D1.1 Model-Specific Dialect Formulations

Nabeeh Jum'ah Julian Kunkel Michel Müller Hisashi Yashiro
Thomas Dubos John Thuburn

Workpackage: WP1 Towards higher-level code design
Responsible institution: Universität Hamburg
Contributing institutions: RIKEN, IPSL
Date of submission: July 2017

Contents

1 Introduction **3**
1.1 Relation to the Project . . . 3
1.2 Motivation . . . 3
1.3 Methodology . . . 4
1.4 Structure of this Document . . . 4

2 Requirements **4**
2.1 Euclidean- and Icosahedral Grid Geometries . . . 6
2.2 Model-specific Needs and Dialects . . . 9
2.3 Functional Requirements . . . 19
2.4 Non-Functional Requirements . . . 19

3 Extending Models' Programming Language **20**
3.1 Collaborative extension development . . . 21
3.2 Extensions and Domain-Specific Concepts . . . 21

4 Code Examples **28**
4.1 ICON . . . 28
4.2 DYNAMICO . . . 35
4.3 NICAM . . . 35
4.4 ASUCA . . . 39

5 Evaluation of Code Quality **42**

6 Related Work **43**

7 Summary and Conclusions **44**
7.1 Dialects . . . 44
7.2 ASUCA and Hybrid Fortran . . . 45

Abstract

Exploiting the power of HPC, is a main concern for the scientists in the climate and atmospheric sciences. The usual software development tools are not sufficient to help them get the optimal use of the computer resources. In the AIMES project WP1, we study the approach of higher level coding to provide performance poratbility. We examine the use of a domain-specific language to provide the scientists a tool to develop software using the domain concepts. We start making domain-based abstractions with suggesting dialects for the three models (DYNAMICO, ICON, and NICAM). Furthermore, a comparison is drawn to a further model (ASUCA) with a different set of requirements and a way to incorporate these requirements is examined. Such approach allows the scientists to write the scientific applications with a readable code without any optimization or hardware-related details.

Under the first task of WP1, we have explored the development of language constructs to extend the Fortran language in each of the three models. In collaboration with the scientists each of whom masters one of the subject models, we have chosen a set of hand optimized Fortran codes from the models, to seek for the possible opportunities for the language extensions development. We have suggested abstractions to extend the Fortran language to serve the models, and based on the suggestions we have rewritten the given codes with the language extentions. We have discussed the suggestions based on the rewritten codes, and formalized the specifications of the language extensions after the agreement on them. In this report we give an overview of the work that has already been achieved for the three models dialects.

1 Introduction

This section describes first the relation to the project according to the project proposal in Section 1.1. Then the motivation for the work that is done under this task is discussed in Section 1.2. A brief description of the methodology that we have used during this task is given in Section 1.3. Section 1.4 describes the structure of this document.

1.1 Relation to the Project

This report describes the work that has been achieved under the work package WP1 of the AIMES project. According to the projects proposal and plan, the tasks under this deliverable are:

- *Higher-level code design for DYNAMICO*
  This task covers the transition of DYNAMICO [DDT+15] towards a higher-level code design. Firstly, the identification of the language dialect relevant for developers of the DYNAMICO model and, secondly, the re-write of relevant operators into the meta-dsl and DYNAMICO-specific dialect is performed.
- *Higher-level code design for ICON*
  This task covers the transition of ICON [ZRRB15] towards a higher-level code design. Firstly, the identification of the language dialect relevant for developers of the ICON model and, secondly, the re-write of relevant operators into the meta-dsl and ICON-specific dialect is performed. While we have already gained experience in ICOMEX regarding the dialect for the ICON ocean and converted two operators, the started process is not complete. In this task, we cover more complex operations and other model parts, and the dialect requires formalization.
- *Higher-level code design for NICAM*
  This task covers the transition of NICAM [STY+14a] towards a higher-level code design. Firstly, the identification of the language dialect relevant for developers of the NICAM model and, secondly, the re-write of relevant operators into the meta-dsl and NICAM-specific dialect is performed.

1.2 Motivation

The compilers of the general-purpose languages apply some optimizations to the code. However, those compilers can not make some optimization decisions on behalf of the programmers. Furthermore, they can not handle some optimizations without an external guidance. Hence, some opportunities to achieve a higher performance are lost.
On the other hand, providing the optimization decisions by the programmers needs an expertise in many programming details which are far from the scientific domain knowledge. The scientists then need to learn many skills besides to their domain. Additionally, applying the optimization within the source code harms the

code structure and the understandability of it and makes it need more effort for any subsequent maintenance or code modifications.

In fact, we have explored some compilers optimizations and how they handle different code structures. We have investigated, for example, the memory layout of the grid variables and the impact on the performance under different compilers and optimization options (fig. 1). The performance of the developed code was sensitive to the coding decisions, e.g. the memory layout.

1.3 Methodology

Our approach to get around the shortcomings of existing compilers is to provide language extensions that lift general purpose code on a higher abstraction level. This leads to a clear separation of concerns:
Scientists need to focus on the scientific problem and not on computing/performance details. So, they should be given the right tools and language to do that. Hardware details and features should not be a concern for domain scientists. Those details will be prepared by scientific programmers fig. 2. Those programmers are experienced with an architecture's details, and how to use its features in the best way to optimize an algorithm's execution performance. So, based on the DSL implementations which are customized to hardware features, the final generated code will be target-machine optimized.
We extend the programming language that is used to write the source code of a model, with higher level constructs. The extensions are based on the domain science concepts.
A scientist who writes a model's code writes only the code which is necessary to deliver the intended results from a scientific perspective. Additional optimization or hardware-specific information are not used to write the source code. The language extensions provide the way to hide such information.
To produce a high-performance software, a code repository which is written with the extended language, is processed by a source-to-source translation tool. This tool transforms the extensions into an original language code that is ready for compilation. The translation process generates the code in a form that is suitable to run with a high performance on the target hardware.
The focus of this document is on the language extensions. Mainly we discuss the extensions which we have developed for the three icosahedral models; Dynamico, Icon, and NICAM. ASUCA and its implementation strategy using "Hybrid Fortran" are also discussed for comparison purposes.

1.4 Structure of this Document

In Section 2 we discuss the requirements which guided and shaped our total approach and the dialects developed to serve each of the three models. The computation in icosahedral models is introduced first, and the model-specific needs for dialects are discussed. The section is finalized with a discussion of the functional and non-functional requirements. Extending the programming language of a model is discussed in Section 3. First, the collaboration with the domain scientists, who has the expertise with the model, is discussed. Then, the extensions that are added to address domain-specific concepts are discussed. Code examples demonstrate the dialects developed for each of the three models in Section 4, besides to code examples from ASUCA. Section 5 discusses the evaluation of the code quality when the dialects are used in coding the models. A review of related work in the literature and known projects is dicussed in Section 6. And finally, Section 7 gives a summary and conclusions of the work described in this document.

2 Requirements

This section describes the requirements for the solution we offer including the language extensions that are developed and the compiler infrastructure to translate the code. We purposely include aspects of the compiler infrastructure development as the concepts of the developed tools influence the design and usage of the language. Therefore, we pursue a co-design approach of tool development and language extension. The requirements express the documented needs that the design, product or process must be able to perform or satisfy (wikipedia).

The existing climate models use hand optimization with multiple code sections to gain performance on different hardware platforms. This is essential, because a code that is optimized to run on one machine, would loose performance on another one. But with the accelerating changes in the hardware features and the addition of different components to heterogeneous systems, the code repositories become a big burden. The support for new hardware features needs the addition of more and more code sections or versions of the software repositories. Maintaining such code repositories would be more complex and redundant. Furthermore, the hand optimization of the code besides to the code structure limits the readability of the code. Normally an algorithm's code is transformed into a code that is not easliy understood. Modifying the code is then a tough and boring mission,

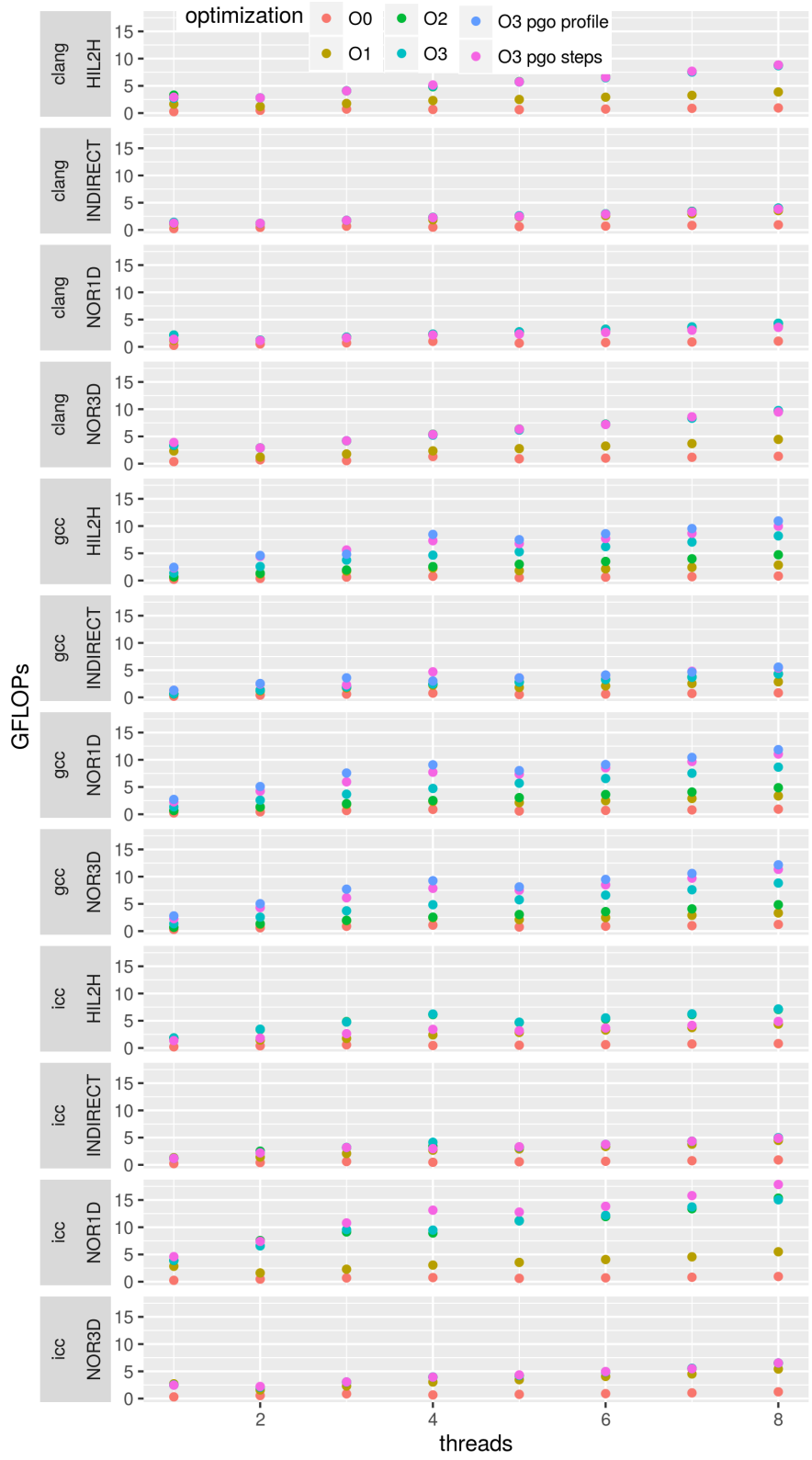


Figure 1: Optimization of different memory layouts by different compilers: performance is sensitive to coding decisions

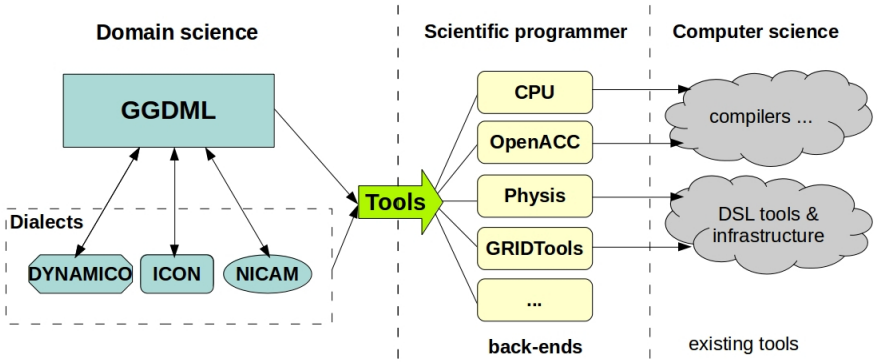


Figure 2: Seperation of Concerns: Scientists work on scientific problem coding and scientific programmers provide the details of code generation

which is done repeatedly over the code sections. The scientists need a high expertise to optimize their algorithms for the various hardware platforms.

To avoid the mentioned problems, we explore higher level coding, discarding any low level details that are needed for the general-purpose language compilers. With the higher-level domain-concept-based coding, we provide a simpler and easier development process for the scientists. A higher-level source code is written only once, without any complex optimization details. It is then translated into a general-purpose-language (or intermediate language) code by a source-to-source translation tool. The translation eventually generates a code that is optimized for a specific target machine. The translation process is driven by configuration information. Specialist scientific programmers, who have expertise in hardware platforms, provide such information in configuration files to lead the translation tool to generate an optimized code.

In order for the language extensions to achieve the intended objectives, the development process had to satisfy a set of requirements and restrictions. In this section we discuss the objectives and requirements, which guide our development effort.

2.1 Euclidean- and Icosahedral Grid Geometries

In climate models, the grid is an essential part for the development of codes which compute the values of the variables that represent fields over some space. Grids are used to discretize the space over which the variables are measured. Some models use rectangular structured grids which simply address data by Euclidean space coordinates. An advantage of modeling with such grids is the simplicity of mapping and addressing of a variable's values. The values of a variable on a regular grid is stored in a multi-dimensional array. To access a variable's value, direct addressing with an explicitly-provided index for each dimension is used. In fact this represents a simpler addressing scheme in computer memory, which has performance advantages with regards to how efficiently an implementation can make use of the available memory bandwidth, especially when running on hardware architectures that are heavily optimized for sequential access performance (e.g. GPUs). However, the main shortcoming of rectangular grids is the difficulty to account for the curvature of the earth, which becomes increasingly problematic with increasing scale of the model, that is, a rectangular grid that covers an increasing surface area contains rectangles with either different areas or different shapes depending on the position of the rectangle. Thus, rectangular grids with longitude/latitude do not fit global climate/atmospheric models. This trade-off leads to the continuing need for different grid types to support global models development. The requirement of some models for a more isotropic and equal-area global grid creates the need to go beyond Euclidean space, e.g. towards the icosahedral geometry.

An icosahedral model is one that uses an icosahedral grid, which represents the earth surface into an icosahedron. The faces of an icosahedron are further divided into smaller traiangles repeatedly to a level that is enough to provide an intended resolution. Further refinements for some triangles allow for nested grids, which provide higher resolution for specific regions on the globe. ICON for instance exhibits such capability, which is not the case for simple structured grids.
In icosahedral grids, hexagons can be synthesized. Yet pentagonal areas still exist then. Thus we see icosahedral models use either traiangular or hexagonal grids. Variables are declared with respect to the grid. They can be declared at the centers of the cells, on the edges of the cells, or at their vertices (Figure 3). Figure 4 and fig. 5 show the variables locations with respect to the triangular and hexagonal grids respectively.

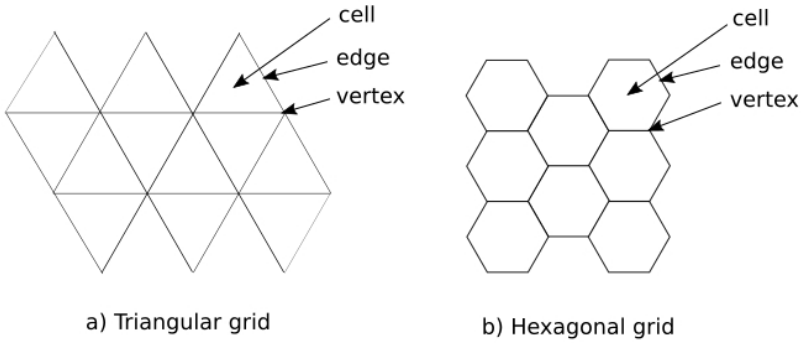
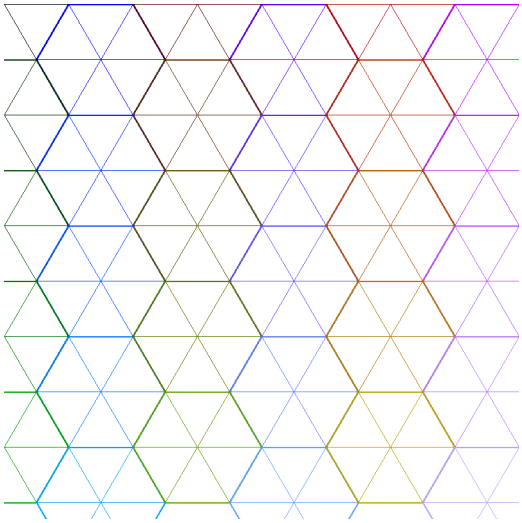
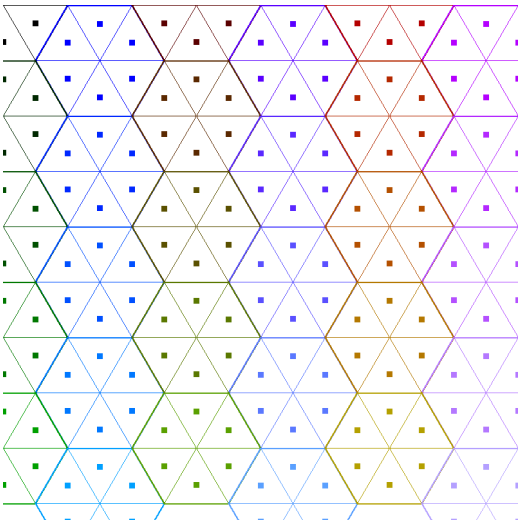


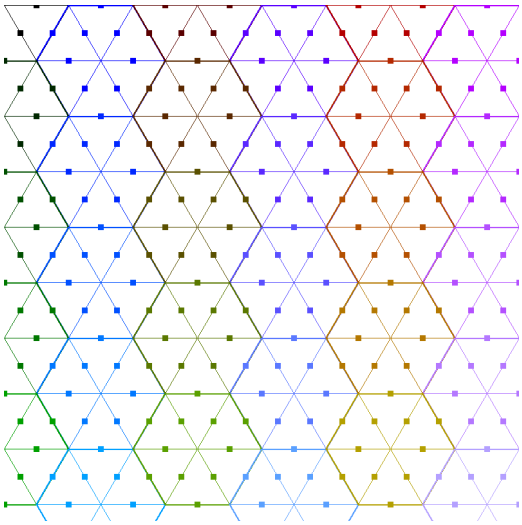
Figure 3: Icosahedral grids and variables



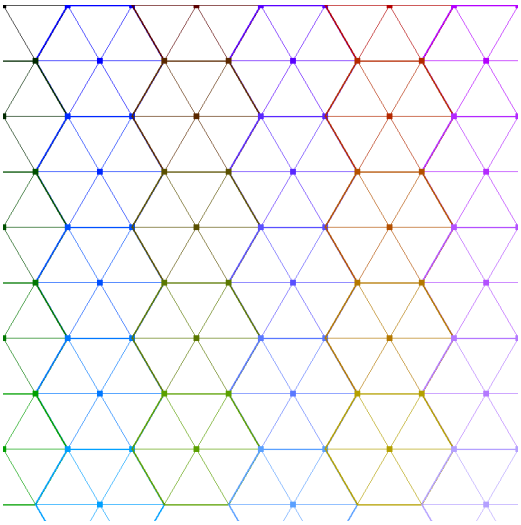
(a) A triangular grid



(b) A triangular grid with a variable defined at the cell centers



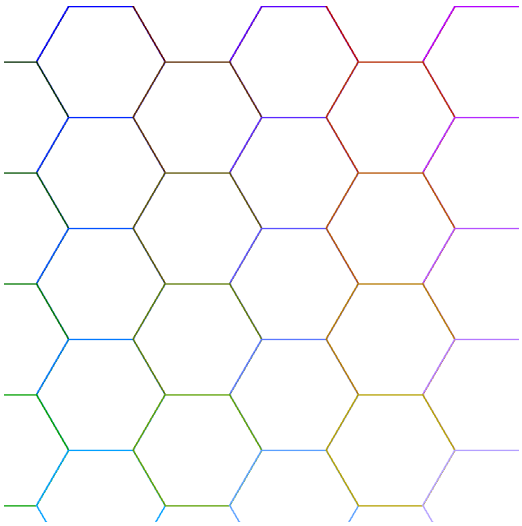
(c) A triangular grid with a variable defined on the edges



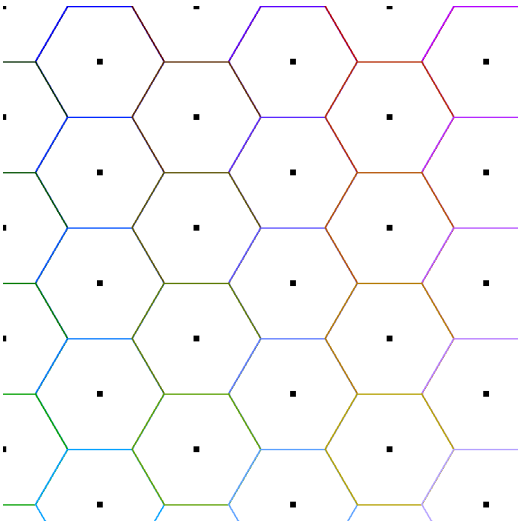
(d) A triangular grid with a variable defined at the vertices

Figure 4: Triangular grid

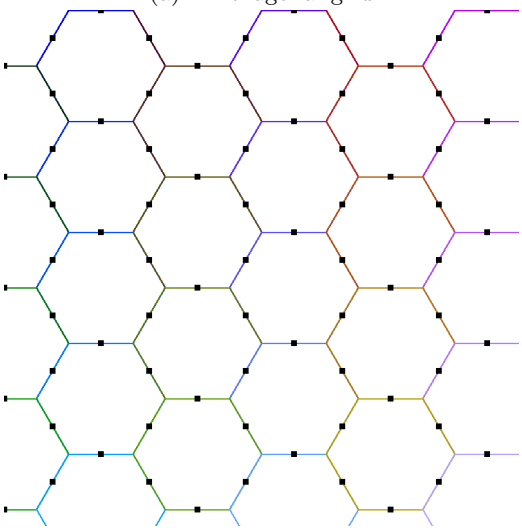
Moving from structured grids to icosahedral grids allows a model to provide new functionalities. However, it complicates the storage of the data of the variables. Multi-dimensional arrays do not directly support the storage of the icosahedral-grid-bound variables like they do in the structured grids. The values of the variables over two-dimensional (surface) grids are stored in a one-dimensional array. A transformation function is then needed to



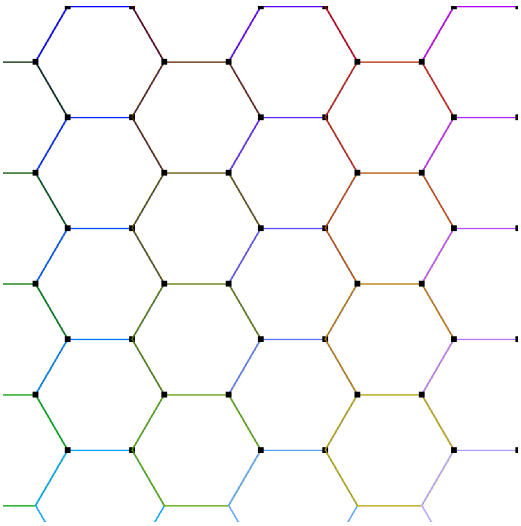
(a) A hexagonal grid



(b) A hexagonal grid with a variable defined at the cell centers



(c) A hexagonal grid with a variable defined on the edges



(d) A hexagonal grid with a variable defined at the vertices

Figure 5: Hexagonal grid

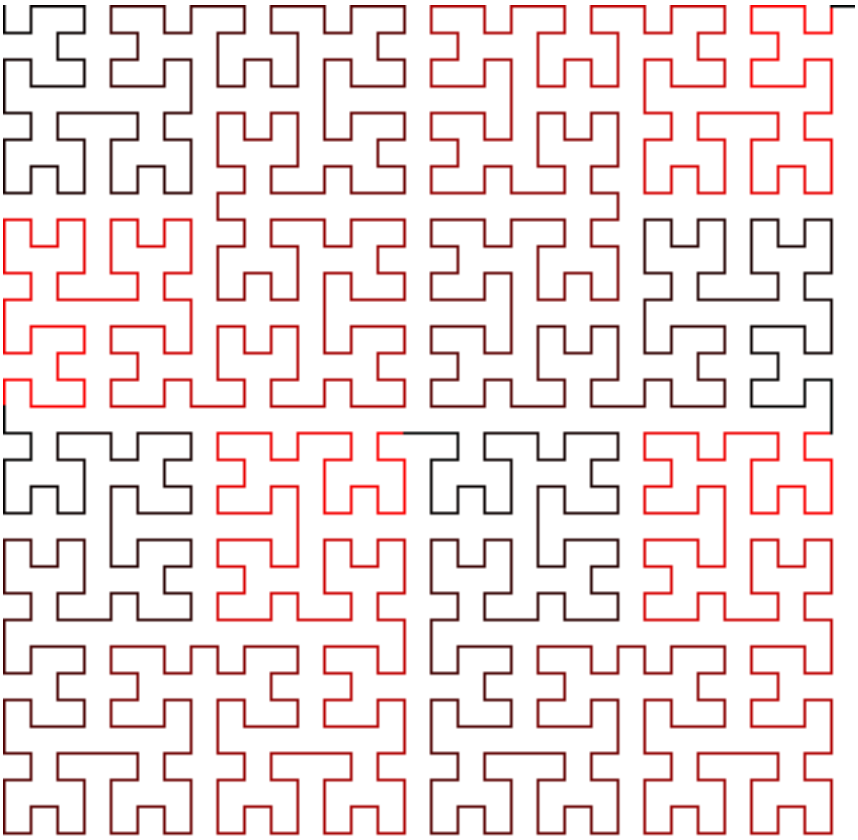


Figure 6: Hilbert space-filling curve

map the indices between the two-dimensional and the one-dimensional layouts. The choice of this transformation function affects the memory bandwidth and hence the model's performance. With such transformations, models need to use indirect addressing to access a variables value. In indirect addressing, special arrays are used to store the actual place of a variable's value in the memory. HEVI methods with a space-filling curve (e.g. Hilbert speace-filling curve – fig. 6) for the horizontal surface enable the indirect addressing for the three-dimensional grids. The developed extensions, which we discuss in this document, are intended mainly to support HEVI-based solutions.
An important feature for climate/atmospheric models is the stencil-based calculation of a grid variable. Thus, the value of a variable at some point in the grid depends on a set of surrounding values of a variable. With rectangular grids and multi-dimensional arrays, the locality of the memory accesses according to the layout of the array allows for better memory bandwidth usage. The space-filling curve (e.g. Hilbert speace-filling curve) provides such feature for icosahedral models. The nature of the curve maps close (neighbouring) points on the two-dimensional surface to close points on the one-dimensional line. This way, the stencil points which need to be accessed to update a variable's value are all situated close to each other in memory, allowing for better memory bandwidth.

2.2 Model-specific Needs and Dialects

In this section we introduce the icosahedral models for a better understanding of the scientific problem and programming issues. ASUCA and its "Hybrid Fortran" based implementation is furthermore introduced for comparison purposes.

2.2.1 DYNAMICO

DYNAMICO is a shallow-atmosphere fully-compressible model based on a mass-based vertical coordinate and a Hamiltonian formulation [DT14]. Prognostic variables are pseudo-density ρ_s, mass-weighted tracers (potential temperature, water species), geopotential Φ, "horizontal" covariant components of momentum and mass-weighted vertical momentum $W = \rho_s g^{-2} D\Phi/Dt = \rho_s g^{-1} w$. Prognostic equations are in flux-form for mass, tracers and W, in advective form for Φ and in vector-invariant form for "horizontal" momentum.

The horizontal mesh is a quasi-uniform icosahedral C-grid obtained by subdivision of a regular icosahedron. Control volumes for mass, tracers and entropy/potential temperature are the hexagonal cells of the Voronoi mesh to avoid the fast numerical modes of the triangular C-grid. The prognostic variables are arranged vertically on a Lorenz grid with all thermodynamical variables collocated with mass. The spatial discretization is obtained from the three-dimensional Hamiltonian formulation. Tracers are transported using a second-order finite volume scheme with slope limiting for positivity.
DYNAMICO uses an additive Runge-Kutta time scheme with two Butcher tableaus, one explicit and one implicit. Currently the second-order 3-stage scheme ARK(2,3,2) is used [GKC13]. A Hamiltonian splitting decides which terms of the equations of motion are treated explicitly or implicitly (T. Dubos and S. Dubey, in preparation). As a result the implicit terms couple the vertical acceleration due to the pressure gradient and the adiabatic pressure change due to vertical displacements of fluid parcels. The resulting implicit problem reduces to independent, scalar, purely vertical, nonlinear problems which are solved to machine precision in two Newton iterations involving one tridiagonal solve each. The overall time scheme has a HEVI (horizontally explicit, vertically implicit) structure.
DYNAMICO is implemented in Fortran 90. It uses hybrid MPI/OpenMP parallelism and a patchwise structured memory layout to allow efficient memory accesses and vectorization. The icosahedral mesh is partitioned into logically Cartesian patches which are distributed amond MPI processes. Correspondingly, data for a given field is accessed through an array of an opaque derived type $field\_t$, essentially containing a allocatable array. Each MPI process may own several patches, Patches owned by an MPI process are distributed among OpenMP threads. A single patch can be dispatched to several OpenMP threads, in which case the vertical dimension is distributed among those threads when possible. Horizontal layers are contiguous in memory, in order to avoid memory conflicts between threads. In the presence of vertical dependencies (e.g. vertical tridiagonal solver), the horizontal direction is distributed among threads, with potentially more memory conflicts. Each patch comes with a halo of nearest neighbours ; whenever necessary, opaque halo exchange routines are called, which perform a mix of non-blocking MPI communication and direct memory-to-memory copies.
The grid partitioning, its distribution among MPI processes and threads and the resulting array sizes and loop ranges are as follows :

- Run-time parameter nbp determines the total number of mesh points. Its role is similar to ICON's $glevel$, with $nbp = 1 + 4^{glevel}$. There are $10(nbp-1)^2+2$ cells (hexagons), $20(nbp-1)^2$ vertices and $30(nbp-1)^2$ edges.
- Run-time parameters $nsplit\_i, nsplit\_j$ determine the mesh partition. If $nsplit\_i = 1, nsplit\_j = 1$, the mesh is partitioned into 10 logically-Cartesian rhombii containing nbp^2 cells, $2(nbp-1)^2$ vertices $(3nbp-1)(nbp-1)$ edges. With $nsplit\_i > 1, nsplit\_j > 1$, the main rhombii are subdivided in $10 \times npslit\_i \times nsplit\_j$ patches of sizes as equal as possible.
- Notice that there is some overlap between the patches. Furthermore stencil calculations require the presence of halo values beyond the boundary of the patch. Consistency of the values stored in different patches and their halos is managed by the halo exchange routines.
- For each patch, 2D data stored at cells is represented by an array of size $iim \times jjm$ where $iim > nbp/nsplit\_i, jjm > nbp/nsplit\_j$ to accomodate for halos. The corresponding horizontal indices (i, j) are lumped into a single horizontal index ij to allow longer loops.
- To store data at vertices, one first decides to attach each vertex to a single cell among its 3 neighboring cells. Conversely, each cell is 'responsible' for two vertices. Thus 2D data stored at vertices is represented by an array of size $2 \times iim \times jjm$.
- To store data at edges, one first decides to 'attach' each edge to a single cell among its 2 neighboring cells. Conversely, each cell is 'responsible' for 3 edges. Thus 2D data stored at edges is represented by an array of size $3 \times iim \times jjm$.
- Vertices and edges are 'attached' to cells in a regular way, which allows the access to neighbouring edges and vertices by adding loop-invariant offsets to index ij. This index ranges either from ii_begin to ij_end or from ii_begin_ext to ij_end_ext. The latter, extended range includes the nearest halo points and guarantee the correctness of certain sequences of computations without exchanging halos in between.
- There are llm full levels in the vertical direction, and $llm + 1$ half-levels.
- In the absence of vertical dependencies, each thread loops the vertical index l between thread-private bounds. The lower bound is ll_begin (resp. $ll\_beginp1$) when the full range starts at $l = 1$ (resp. $l = 2$) and the upper bound is ll_end (resp. ll_endm1, ll_endp1) when the full range ends at $l = llm$ (resp. $llm - 1, llm + 1$).

- In the presence of vertical dependencies, each thread loops the vertical index l from $1/2$ to $llm - 1/llm/llm+1$. The horizontal range ij_begin, ij_end (resp. ij_begin_ext, ij_end_ext) is distributed among threads ; each thread loops ij over ij_omp_begin, ij_omp_end (resp. $ij\_omp\_begin\_ext, ij\_omp\_end\_ext$.

A computation typically involves a two-level process. At the higher level, a subroutine X takes as input variables arrays of $field\_t$, each describing one input or output field. This subroutine loops over patches in the case that one MPI process handles several patches. For each patch, it obtains pointers to the arrays containing the data for that patch. This includes the input and output fields, as well as geometric quantities (areas, lengths, ratios of areas and lenghts) which are computed once and for all at grid generation. Then a lower-level *compute_X*' routine is called, with the arrays containing the field data for that patch as arguments (the static geometric data is made available through global pointer variables). The actual computation is done by *compute_X*, which loops over indices as sketched above, the horizontal loop being innermost for contiguous memory access. Its arguments are declared as Fortran arrays with explicit shapes, in order to optimize the ability of the compiler to optimize and vectorize the loops.
In the first example below, the horizontal divergence of the horizontal mass flux is computed :

$$convm(ij,l) = \frac{1}{A(ij)} \sum_{e \in edges(ij)} n(e) hflux(e,l)$$

where $hflux(e,l)$ is the flux of air through a vertical face between two hexagonal control volumes, $n(e) = \pm 1$ is a sign such that $n(e)F(e,l)$ is positive for outgoing flux and $A(ij)$ is the area of cell ij.

Listing 1: Horizontal divergence operator of DYNAMICO

```
REAL(rstd),INTENT(IN)  :: hflux(3*iim*jjm,llm)  ! hflux in kg/s
REAL(rstd),INTENT(OUT) :: convm(iim*jjm,llm)  ! mass flux convergence
INTEGER :: ij,l

DO l=ll_begin, ll_end
   !DIR$ SIMD
   DO ij=ij_begin,ij_end
      ! convm = -div(mass flux), sign convention as in Ringler et al. 2012, eq. 21
      convm(ij,l)= -1./Ai(ij)*(ne_right*hflux(ij+u_right,l)   +  &
            ne_rup*hflux(ij+u_rup,l)       +  &
            ne_lup*hflux(ij+u_lup,l)       +  &
            ne_left*hflux(ij+u_left,l)     +  &
            ne_ldown*hflux(ij+u_ldown,l)   +  &
            ne_rdown*hflux(ij+u_rdown,l))
   END DO ! ij
END DO ! llm
```

Since all cells have 6 neighbours, the loop over edges is manually unrolled and loop-invariant offsets are used to access data at edges. If a cell is a pentagon, its sixth edge is fictitious and the corresponding flux is zero, so that no special handling of that case is necessary. The SIMD directive helps the compiler vectorize the inner loop.
The second example below is more involved. It computes one part of the time derivative of velocity (wind) :

$$du(e,l) := du(e,l) + \frac{1}{2} \sum_{e' \in edges(e)} w(e,e') hflux(e',l)(qu(e,l) + qu(e',l))$$

where e, e' are edges, $edges(e)$ is the set of edges of the two cells touching e and $w(e,e')$ are precomputed weights.

Listing 2: Corilis operator of DYNAMICO

```
REAL(rstd),INTENT(IN)    :: hflux(3*iim*jjm,llm)  ! hflux in kg/s
REAL(rstd),INTENT(OUT)   :: convm(iim*jjm,llm)    ! mass flux convergence
REAL(rstd),INTENT(INOUT) :: du(3*iim*jjm,llm)
INTEGER :: ij,l

! Compute potential vorticity (Coriolis) contribution to du
DO l=ll_begin,ll_end
   !DIR$ SIMD
   DO ij=ij_begin,ij_end
      uu_right = &
          wee(ij+u_right,1,1)                                          &
            *hflux(ij+u_rup,l)*(qu(ij+u_right,l)+qu(ij+u_rup,l))     &
```

```
            +wee(ij+u_right,2,1)                                              &
               *hflux(ij+u_lup,l)*(qu(ij+u_right,l)+qu(ij+u_lup,l))           &
            +wee(ij+u_right,3,1)*hflux(ij+u_left,l)                           &
               *(qu(ij+u_right,l)+qu(ij+u_left,l))                            &
            +wee(ij+u_right,4,1)*hflux(ij+u_ldown,l)                          &
               *(qu(ij+u_right,l)+qu(ij+u_ldown,l))                           &
            +wee(ij+u_right,5,1)*hflux(ij+u_rdown,l)                          &
               *(qu(ij+u_right,l)+qu(ij+u_rdown,l))                           &
            +wee(ij+u_right,1,2)*hflux(ij+t_right+u_ldown,l)                  &
               *(qu(ij+u_right,l)+qu(ij+t_right+u_ldown,l))                   &
            +wee(ij+u_right,2,2)*hflux(ij+t_right+u_rdown,l)                  &
               *(qu(ij+u_right,l)+qu(ij+t_right+u_rdown,l))                   &
            +wee(ij+u_right,3,2)*hflux(ij+t_right+u_right,l)                  &
               *(qu(ij+u_right,l)+qu(ij+t_right+u_right,l))                   &
            +wee(ij+u_right,4,2)*hflux(ij+t_right+u_rup,l)                    &
               *(qu(ij+u_right,l)+qu(ij+t_right+u_rup,l))                     &
            +wee(ij+u_right,5,2)*hflux(ij+t_right+u_lup,l)                    &
               *(qu(ij+u_right,l)+qu(ij+t_right+u_lup,l))

          uu_lup = &
              wee(ij+u_lup,1,1)                                                                  &
               *hflux(ij+u_left,l)*(qu(ij+u_lup,l)+qu(ij+u_left,l))                              &
            +wee(ij+u_lup,2,1)                                                                   &
               *hflux(ij+u_ldown,l)*(qu(ij+u_lup,l)+qu(ij+u_ldown,l))                            &
            +wee(ij+u_lup,3,1)                                                                   &
               *hflux(ij+u_rdown,l)*(qu(ij+u_lup,l)+qu(ij+u_rdown,l))                            &
            +wee(ij+u_lup,4,1)                                                                   &
               *hflux(ij+u_right,l)*(qu(ij+u_lup,l)+qu(ij+u_right,l))                            &
            +wee(ij+u_lup,5,1)                                                                   &
               *hflux(ij+u_rup,l)*(qu(ij+u_lup,l)+qu(ij+u_rup,l))                                &
            +wee(ij+u_lup,1,2)                                                                   &
               *hflux(ij+t_lup+u_right,l)*(qu(ij+u_lup,l)+qu(ij+t_lup+u_right,l))                &
            +wee(ij+u_lup,2,2)                                                                   &
               *hflux(ij+t_lup+u_rup,l)*(qu(ij+u_lup,l)+qu(ij+t_lup+u_rup,l))                    &
            +wee(ij+u_lup,3,2)                                                                   &
               *hflux(ij+t_lup+u_lup,l)*(qu(ij+u_lup,l)+qu(ij+t_lup+u_lup,l))                    &
            +wee(ij+u_lup,4,2)                                                                   &
               *hflux(ij+t_lup+u_left,l)*(qu(ij+u_lup,l)+qu(ij+t_lup+u_left,l))                  &
            +wee(ij+u_lup,5,2)                                                                   &
               *hflux(ij+t_lup+u_ldown,l)*(qu(ij+u_lup,l)+qu(ij+t_lup+u_ldown,l))

          uu_ldown = &
              wee(ij+u_ldown,1,1)                                                                    &
               *hflux(ij+u_rdown,l)*(qu(ij+u_ldown,l)+qu(ij+u_rdown,l))                              &
            +wee(ij+u_ldown,2,1)                                                                     &
               *hflux(ij+u_right,l)*(qu(ij+u_ldown,l)+qu(ij+u_right,l))                              &
            +wee(ij+u_ldown,3,1)                                                                     &
               *hflux(ij+u_rup,l)*(qu(ij+u_ldown,l)+qu(ij+u_rup,l))                                  &
            +wee(ij+u_ldown,4,1)                                                                     &
               *hflux(ij+u_lup,l)*(qu(ij+u_ldown,l)+qu(ij+u_lup,l))                                  &
            +wee(ij+u_ldown,5,1)                                                                     &
               *hflux(ij+u_left,l)*(qu(ij+u_ldown,l)+qu(ij+u_left,l))                                &
            +wee(ij+u_ldown,1,2)                                                                     &
               *hflux(ij+t_ldown+u_lup,l)*(qu(ij+u_ldown,l)+qu(ij+t_ldown+u_lup,l))                  &
            +wee(ij+u_ldown,2,2)                                                                     &
               *hflux(ij+t_ldown+u_left,l)*(qu(ij+u_ldown,l)+qu(ij+t_ldown+u_left,l))                &
            +wee(ij+u_ldown,3,2)                                                                     &
               *hflux(ij+t_ldown+u_ldown,l)*(qu(ij+u_ldown,l)+qu(ij+t_ldown+u_ldown,l))              &
            +wee(ij+u_ldown,4,2)                                                                     &
               *hflux(ij+t_ldown+u_rdown,l)*(qu(ij+u_ldown,l)+qu(ij+t_ldown+u_rdown,l))              &
            +wee(ij+u_ldown,5,2)                                                                     &
               *hflux(ij+t_ldown+u_right,l)*(qu(ij+u_ldown,l)+qu(ij+t_ldown+u_right,l))

          du(ij+u_right,l) = du(ij+u_right,l) + .5*uu_right
          du(ij+u_lup,l)   = du(ij+u_lup,l)   + .5*uu_lup
          du(ij+u_ldown,l) = du(ij+u_ldown,l)   + .5*uu_ldown
        END DO
      END DO
```

As previously, the loop over the 10 elements of $edges(e)$ is manually unrolled and loop-invariant offsets are used to access data at edges. Furthermore, another typical aspect of DYNAMICO code is exemplified : since the index ij loops over cells, and the output data is at edges, the computation must be done for each of the 3 edges attached to each cell. In practice the corresponding code is manually inlined 3 times.

As exemplified above, writing performance-critical DYNAMICO code features much manual unrolling and inlining. While this is not difficult, it is error-prone, especially since different offsets must be used for each instance of inlined code. Ideally, this unrolling and inlining should be performed automatically by a source-to-source translator while code of inner loops would be written only once in an adequately expressive DSL.

2.2.2 ICON

ICON is the ICOsahedral Non-hydrostatic modeling framework developed jointly by the German Weather Service (DWD) and the Max Planck Institute for Meteorology (MPI-M). It was developed for unified next-generation global numerical weather prediction (NWP) and climate modeling. One of the main considerations for the development of ICON were the exact mass conservation (and further wish for energy conservation) which needs a non-hydrostatic dynamical core. Another consideration was related to the achieved advances on computing infrastructure where more massively parallel architectures have been arising. So, ICON was meant to scale the software to such highly-parallel architectures. The grids refinement and nesting and their computational impact were also another consideration to develop ICON. ICON uses triangular icosahedral C grids. This decision allowed for a simple way to refine grids with nesting.

The horizontal grid used for ICON is an unstructured grid which projects the earth surface over a spherical icosahedron. The twenty equally-sized faces of the icosahedron are first divided in a root division step by dividing each triangles edge into n parts. This step refines each triangle's area into n^2 smaller equally-sized triangles. This division step is denoted Rn. Then, each resulting rectangle is further divided into four equally-sized rectangles by connecting the midpoints of its edges. This last step is applied recursively k times (Figure 7). This recursive division is denoted Bk. The grid is called an $RnBk$ grid. The grid resolution of an $RnBk$ grid is defined by the number of the cells of the grid n_c, the number of the edges of the cells n_e, and the number of the vertices of the cells n_v calculated by the equations eq. (1),eq. (2), and eq. (3) respectively.

$$n_c = 20n^2 4^k \tag{1}$$

$$n_e = 30n^2 4^k \tag{2}$$

$$n_v = 10n^2 4^k + 2 \tag{3}$$

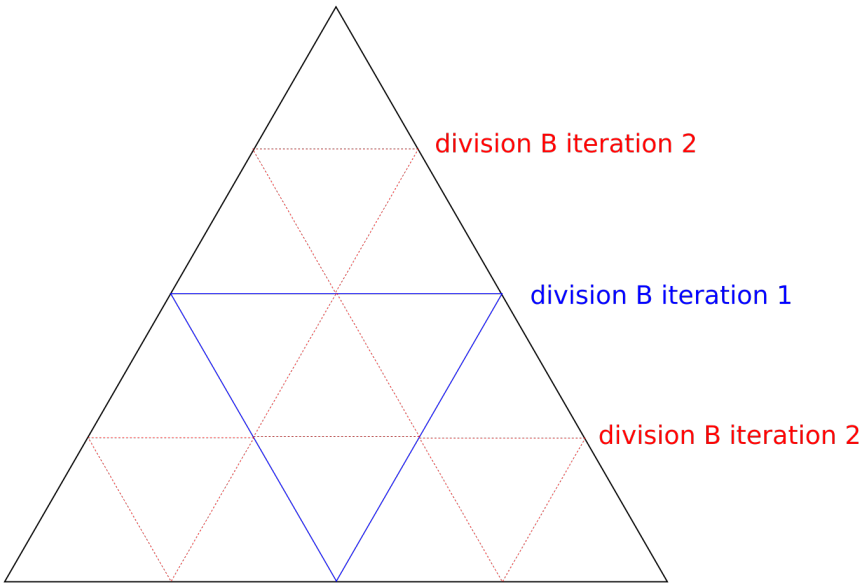


Figure 7: The recursive division of the ICON grid

The nested grids for specific regions are generated by further refining the set of triangles covering that region beyond the mentioned k iterations of the B division step.
The dynamical core is based on the prognostic variables suggested by Gassmann and Herzog [GH08] and [Gas13]. The set of basic equations are described in eq. (4), eq. (5), eq. (6), and eq. (8) respectively

$$\frac{\partial \nu_n}{\partial t} + \frac{\partial K_h}{\partial n} + (\zeta + f)\nu_t + w\frac{\partial \nu_n}{\partial z} = -c_{pd}\theta_\nu \frac{\partial \pi}{\partial n} + F(\nu_n) \tag{4}$$

$$\frac{\partial w}{\partial t} + v_h.\nabla w + w\frac{\partial w}{\partial z} = -c_{pd}\theta_\nu \frac{\partial \pi}{\partial z} - g \tag{5}$$

$$\frac{\partial \rho}{\partial t} + \nabla.(v\rho) = 0 \tag{6}$$

$$\frac{\partial \pi}{\partial t} + \frac{R_d}{c_{vd}}\frac{\pi}{\rho\theta_\nu}\nabla.(v\rho\theta_\nu) = \tilde{Q} \tag{7}$$

where

$$\pi = (\frac{R_d}{P_{00}}\rho\theta_\nu)^{R_d/c_{\nu d}} \tag{8}$$

The models variables are localized at the centers of the grid cells, the edges of the cells, or at the vertices. A horizontally-explicit time-stepping scheme is used in ICON. The model uses MPI to scale up the computation to the compute nodes of a machine. The code of the model has a low halo communication impact. Also, as a result of the horizontally-explicit time-stepping scheme, the model needs no global communication except for I/O and optionally global diagnostics. OpenMP is used to parallelize the traversal of the grid on a node. The layout of the variables in memory is blocked to serve exploiting the memory bandwidth on the compute nodes. The structure of the code is written to run efficiently and make use of the resources on different machines including vector machines and cache-based machines.

2.2.3 NICAM

Non-hydrostatic Icosahedral Model (NICAM; [TS04] [SMTM08] [STY+14b]) is a global high-resolution atmospheric model, which has been continuously developed mainly by Japan Agency for Marine–Earth Science and Technology (JAMSTEC), the University of Tokyo, and the RIKEN Advanced Institute of Computational Science (AICS). Most of the code is written in Fortran 90, except the low-level I/O module. NICAM uses a fully compressible (elastic) non-hydrostatic system and an icosahedral grid configuration. The shape of the horizontal control area is hexagon or pentagon. All of the prognostic variables are co-located at the center of the control area in horizontal. NICAM adopts the Lorenz grid and terrain-following coordinates as a vertical coordinate system. Higher resolution grids are recursively subdivided from a coarser resolution grid [TGS08]. We refer to the grid division level as the glevel. The number of the grid point is given as $10 \times 4^{glevel} + 2$.

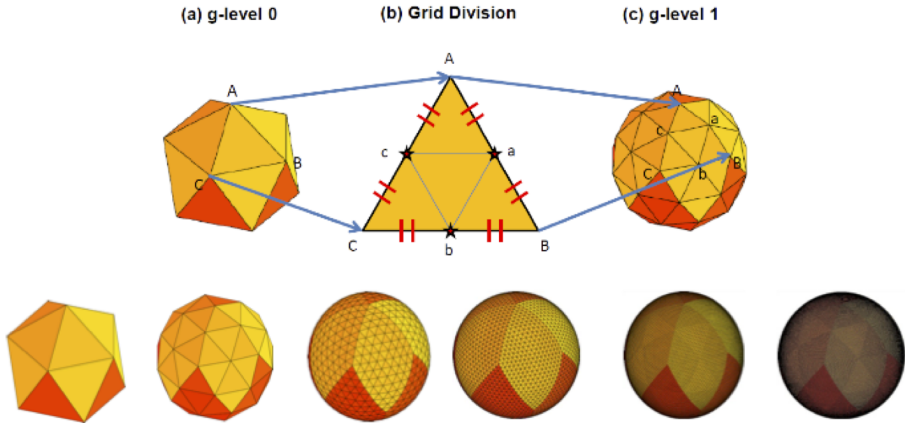


Figure 8: Grid division level (glevel) of NICAM

The icosahedral grid system can be treated as the structured grid in the local scope. At first, we divide the globe by ten rhombus-shaped tiles called "region". Each region has 2-dimentionally structured grids with halo. Regions are also recursively subdivided with keeping the structure of the grids. The region division level is referred as rlevel. The number of the grids in each region is given as $(2^{glevel-rlevel} + 2)^2$.
One or more regions are allocated to each MPI process. The two grids, which typically express the north pole and south pole, are not included in the grid of the regular region except halo. These two grids and surrounding grids are prepared as pole regions. Only a master process computes pole regions. Peer-to-peer communications between regular and regular, and between regular and pole are conducted at the appropriate timing in the time loop. The icosahedral grid system has twelve pentagonal control areas. The two of them are the north pole and the south pole, so the pole regions always have a pentagonal shape. Remaining ten are included in the regular

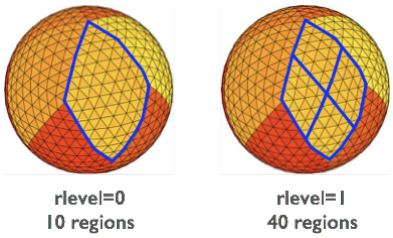


Figure 9: Region division level (rlevel) of NICAM

region. Around the region which has pentagonal grid area, the communication pattern is different from the other regular regions. In this respect, the icosahedral grid system is different from the Euclidian grid system. The grid group in each region is a shape of the rhombus. This 2-d horizontal grid structure is treated as 1-d at the computation to get longer loop for the better computational performance on the supercomputer such as vector supercomputers and GPU-based supercomputers.
Here we show the algorithm for the horizontal divergence operator(fig. 10) as an example of the stencil calculation kernel in NICAM. From Gauss's theorem, we have the divergence of vector $\vec{u}$ at point P_0 as

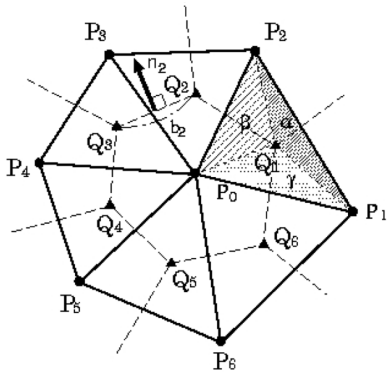


Figure 10: The schematic figure of horizontal divergence operator of NICAM

$$\nabla \cdot \vec{u}(P_0) = \frac{1}{A(P_0)} \sum_{i=1}^{6} b_i \frac{\vec{u}(Q_i) + \vec{u}(Q_{mod(i+1,6)})}{2} \cdot \vec{n_i} \tag{9}$$

where $A(P_0)$ is the area of the control area at point P_0. The $\vec{u}$ at Q_i is calculated by using triangle linear interpolation from the surrounding point. b_i and $\vec{n_i}$ denote the geodesic arc length of $Q_iQ_{mod(i+1,6)}$ and the outward unit vector normal to the arc at the midpoint of $Q_iQ_{mod(i+1,6)}$.
Eq.(9) can be rewriteen as a linear combination

$$\nabla \cdot \vec{u}(P_0) = \sum_{i=0}^{6} \vec{c_i} \cdot \vec{u}(P_i) \tag{10}$$

where vector c_i are constant coefficients. We can calculate c_i only once at the time of setup. The sample of the fortran code is as follows.

Listing 3: Horizontal divergence operator of NICAM

```
integer :: iall ! num. of the horizontal grid for i-axis
integer :: gall = iall * iall
integer :: kall ! num. of the vertical layer
integer :: lall ! num. of the region for this process

real(8) :: scl      (gall,kall,lall)   ! scalar
real(8) :: vx       (gall,kall,lall)   ! horizontal velocity V (x-component)
real(8) :: vy       (gall,kall,lall)   ! horizontal velocity V (x-component)
real(8) :: vz       (gall,kall,lall)   ! horizontal velocity V (x-component)
real(8) :: coef_div(gall,0:6,3,lall) ! constant vector coefficient

integer :: XDIR=1, YDIR=2, ZDIR=3
integer :: gmin, gmax
```

```
integer :: g, k, l

gmin = 1    + iall + 1 ! start point of the inner grid
gmax = gall - iall - 1 ! end   point of the inner grid

do l = 1, lall
do k = 1, kall
do g = gmin, gmax
   scl(g,k,l) = coef_div(g,0,XDIR,l) * Vx(g       ,k,l) &
              + coef_div(g,1,XDIR,l) * Vx(g+1     ,k,l) &
              + coef_div(g,2,XDIR,l) * Vx(g+iall+1,k,l) &
              + coef_div(g,3,XDIR,l) * Vx(g+iall  ,k,l) &
              + coef_div(g,4,XDIR,l) * Vx(g-1     ,k,l) &
              + coef_div(g,5,XDIR,l) * Vx(g-iall-1,k,l) &
              + coef_div(g,6,XDIR,l) * Vx(g-iall  ,k,l) &
              + coef_div(g,0,YDIR,l) * Vy(g       ,k,l) &
              + coef_div(g,1,YDIR,l) * Vy(g+1     ,k,l) &
              + coef_div(g,2,YDIR,l) * Vy(g+iall+1,k,l) &
              + coef_div(g,3,YDIR,l) * Vy(g+iall  ,k,l) &
              + coef_div(g,4,YDIR,l) * Vy(g-1     ,k,l) &
              + coef_div(g,5,YDIR,l) * Vy(g-iall-1,k,l) &
              + coef_div(g,6,YDIR,l) * Vy(g-iall  ,k,l) &
              + coef_div(g,0,ZDIR,l) * Vz(g       ,k,l) &
              + coef_div(g,1,ZDIR,l) * Vz(g+1     ,k,l) &
              + coef_div(g,2,ZDIR,l) * Vz(g+iall+1,k,l) &
              + coef_div(g,3,ZDIR,l) * Vz(g+iall  ,k,l) &
              + coef_div(g,4,ZDIR,l) * Vz(g-1     ,k,l) &
              + coef_div(g,5,ZDIR,l) * Vz(g-iall-1,k,l) &
              + coef_div(g,6,ZDIR,l) * Vz(g-iall  ,k,l)
enddo
enddo
enddo
```

2.2.4 ASUCA

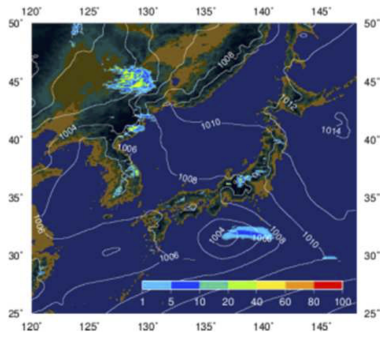


Figure 11: Scale of ASUCA [SIK+]

ASUCA is a mesoscale Weather prediction model developed in Modern Fortran by the Japan Meteorological Agency [IMKK10]. It is in production since 2014 as one of the main models used in operational weather forecast in Japan and covers the area depicted in fig. 11. It uses a traditional Arakawa C-grid (one type of rectangular grid), parallelized in the horizontal domains [SAO14]. Long time steps are implemented using a third order Runge-Kutta method [WS02]. Sound and gravity waves are treated separately using a second order Runge-Kutta method for shorter time steps, employing the HEVI scheme (Horizontally explicit - vertically implicit). Vertical advection of water substances are calculated using a separate timestep for each column, based on the CFL breaking condition [IMKK10]. ASUCA employs generalized coordinates used together with Lambert conformal conic as well as latitude/longitude projections [SAO14]. It is discretized using a finite volume method in three domains: `I` and `J` as interchangeable horizontal domains and `K` as the separately treated (largely sequentially implemented) vertical domain. For performance reasons, ASUCA employs the most ubiquitous and optimized data structure offered by Fortran: Multidimensional arrays. The program structure can roughly be put into two categories: Physical processes (modules depicted under `pp interface` as well as `phys. adjust` in figure 12) and dynamical core (all other modules depicted in figure 12). This distinction is common in weather models such as WRF [MHH14] and COSMO [COG+13]. In operation, ASUCA is used to create nine hour forecasts every hour, calculating the area depicted in fig. 11 in a two kilometer resolution [SIK+]. In this document

ASUCA is used to serve as a comparison case and to examine whether such a case can and shall be supported by a new higher level abstraction.

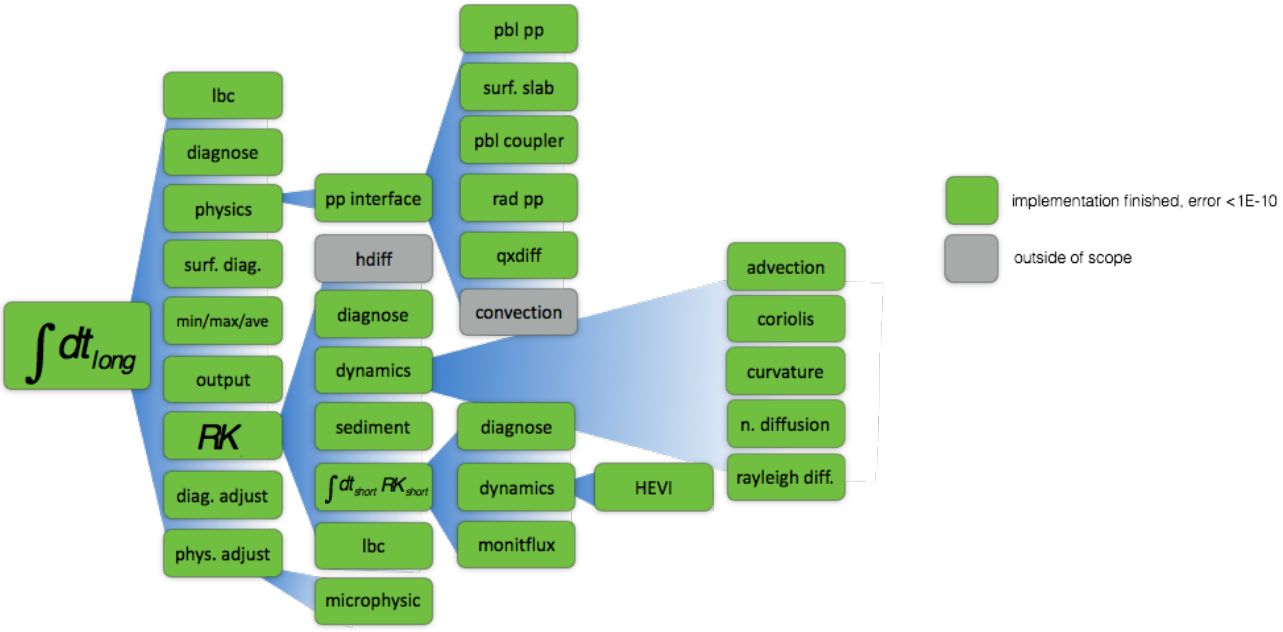


Figure 12: Simplified call graph of ASUCA and status of Hybrid Fortran based implementation

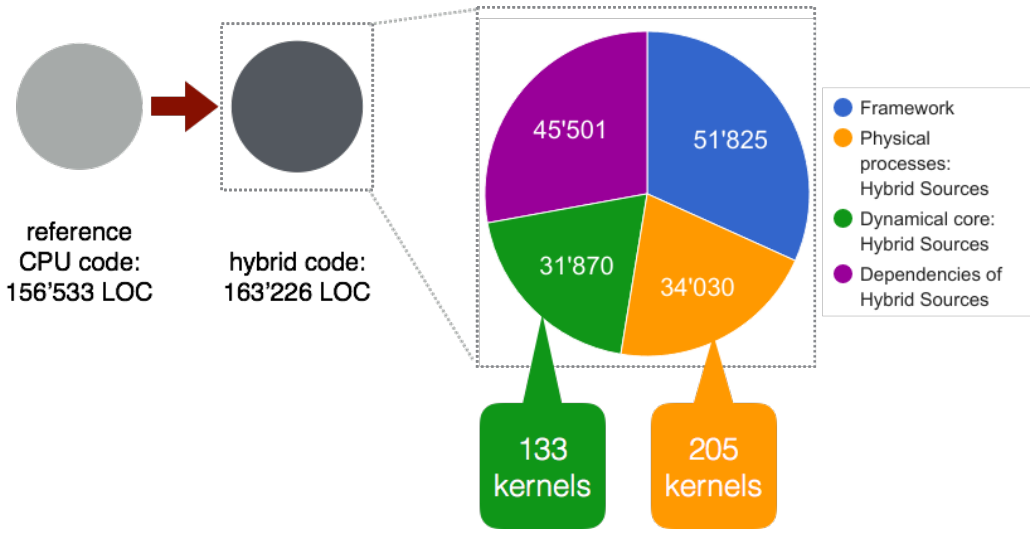


Figure 13: Hybrid ASUCA in numbers

Research at the Tokyo Institute of Technology has lead to a new ”Hybrid Fortran” based implementation of ASUCA. Its main goals are the following:

1. Allowing the same user code to be used on both CPU and GPU architectures. In current research the focus was on Nvidia Tesla and Intel Xeon based architectures, but the switch to others should be possible and streamlined.
2. Staying as close as possible to the original code base to ease the transition.
3. Allowing performance portability.

Requirement 2 has limited the solution space to Fortran based solutions. It has also ruled out DSLs in this case, in favour of a directive based solution. Ideally the existing user code only needs be extended with directives, without the computational code needing to be changed. Requirements 1 and 3 have lead to a new transpiler that takes Fortran plus directives as an input and outputs either OpenMP Fortran or CUDA Fortran. OpenACC has

been evaluated thoroughly, but various stability issues with PGI's implementation (which is the only available one with a near complete feature set except for the proprietary Cray implementation) lead to employing CUDA Fortran and a transpiler instead. It is however still used as a third possible backend implementation, mainly in order to make use of its reduction features (kernel reductions are only supported in the OpenMP and OpenACC based backends). Figure 13 depicts the number of lines of code and number of kernels contained in this new version of ASUCA. This set of tools and the underlying language extension is from here on referred to as "Hybrid Fortran".

The need for performance portability (requirement 3) has lead to a few core design choices for the Hybrid Fortran language extension and its implementation. These are described in the following paragraphs.

2.2.4.1 Storage Order Since the ASUCA dynamical core is based on stencil computations, it is strongly memory bandwidth bounded. This makes choosing the correct storage order paramount to the application performance, since cache misses directly influence the memory pressure [DNW+09]. Different hardware architectures have different ideal storage orders however, especially going from CPU to GPU since their cache architecture works quite . On the CPU, the innermost loop should always be given stride-1 memory access. Since in the ASUCA physical processes the horizontal domains are on CPU parallelized on a coarse-grained level (which avoids context switching costs [Kwi01]), the natural (i.e. giving the best data locality [DHK+00]) choice for storage order in Fortran is either `KIJ` or `KJI`, of which the first one is chosen. Meanwhile on GPUs the domain that is mapped to the first thread index (i.e. one of the parallel domains) should be given stride-1 access to enable coalesced memory access [Har07], which requires either `IJK` or `JIK` orderings - of which again the first one is chosen. Due to requirement 2, Hybrid Fortran therefore reorders all array specifications and accesses by generating macro wrappings, thus allowing the user a centralized specification of storage order while all user code can remain written in a particular order.

2.2.4.2 Compile-Time Defined Parallelization Granularity Figure 13 depicts ASUCA's physical processes to have 205 kernels. This only refers to the GPU version however. In its original CPU implementation, a vast majority of this code (i.e. all calls depicted under "pp interface" in figure 12) is being run in a single parallel kernel. The main reason for this is a context switching and cache optimization on CPU: The term "physical processes" in the context of ASUCA is used for calculations that can be run on each `K`-column separately, without affecting neighboring columns until the next run of the dynamical core. In order to increase cache locality and decrease the amount of context switching, it is therefore advisable on CPU to run all these calculations on a single thread per column before synchronizing [Kwi01]. On GPUs however this strategy is very problematic:

1. Deep call graphs under a single kernel are difficult to maintain since kernel code relies on compiler inlining, which breaks easily and only supports a subset of language constructs.
2. On GPUs the memory, cache and register resources per thread are much more limited, in favour of supporting a much higher number of parallel threads per microchip area.

In order to ensure performance portability it is therefore necessary to allow the parallelization to be applied at different levels in the call graph, depending on what hardware architecture is targeted. On CPU the physical processes depicted under `pp interface` in figure 12 are to keep their coarse grained parallelization, while on GPU, parallelization is to be applied at a finer-grained, i.e. lowever level in the call graph. Hybrid Fortran solves this by allowing each parallel region definition to specify the target architectures it is to be applied for.

2.2.4.3 Compile-Time Defined Privatization Due to the necessity for compile-time defined parallelization granularity (as described above in section 2.2.4.2), the need for compile-time defined privatization immediately follows. A finer granularity of threading requires that data structures that were previously local to one thread need to be shared among all threads running on a single GPU. Hybrid Fortran solves this by giving a set of directives that allow the programmer to extend data structures with additional domains, depending on whether these domains are needed for parallel regions at a lower level of the call graph. This feature goes hand-in-hand with the partial application of parallel regions for different architectures described in section 2.2.4.2.

2.2.4.4 Device Data Region Since ASUCA is heavily based on stencil operations that have a low arithmetic intensity, it is highly susceptible to memory bandwidth [DNW+09]. The main reason to employ GPUs is then to make use of their high bandwidth between device memory and computational cores. This advantage can however only be used if the data largely stays on the device memory, i.e. avoiding unnecessary copying

using the comparatively slow PCI Express bus. To ensure this it is necessary to specify for each data object in each routine, what its behavior with respect to the device memory is to be (analogous to OpenACC code [WSTaM12]). Hybrid Fortran for this reason allows the programmer to specify a data object to use `present` or `transferHere` semantics. It also uses the Fortran 90 style `intent` clauses to determine what kind of transfer needs to be implemented.

2.3 Functional Requirements

We discuss here the functional requirements[1], which drove the development of the model dialects.

RFPD **Parse extended language**

Within source to source translation, parse source code that uses extended language to be further processed.

RFRS **Read source input files**

Read input source code from the models code repository to be parsed and processed during source-to-source translation.

RFRC **Read configuration files**

Read configuration files to handle source-to-source translation process.

RFGC **Generate target code**

Generate code for various backends based on the configuration files, which drive the translation process.

RFOP **Optimize code**

Apply the relevant optimization procedures and rewrite code in a way get optimal performance on the target machine.

RFWS **Write processed code trees**

After source code tree contents are processed the solution should write the processed form into an output code tree.

RFIO **I/O and communication interfaces**

Parallel applications need to communicate data, e.g., in HALO exchanges, or it performs I/O with the data. Thus, the DSL must allow infrastructure code of the models to either access the data of the grid, or it provides means to directly communicate and read/write this data from infrastructure code.

2.4 Non-Functional Requirements

The non-functional requirements, which drove and guided the development of the language extensions are discussed in this section.

RNPP **Performance portability**

One of our goals in the AIMES project is to give the capability for scientists to write performance portable code. Atmospheric/climate applications are highly demanding for high performance computing. However, rewriting parts of the model's code for different architectures is the price of getting performance. With the higher-level code design, we support performance portability of models' code.

RNCT **Compilation time**

Runtime of the tool must be in the order of the regular compile time (2x of current compile time is acceptable).

RNWO **Write-Once Use-Many**

Code should be written once, for all hardware platforms. It should not include any hardware details, so that there is no need to rewrite same algorithm again for any other hardware platform. In particular, there should not be any repeated "pattern" inside the code that could be extracted and abstracted further. This requirement achieves implicitly the following:

Only one copy of algorithm: Code should be written once, in one place, and not repeated in any other places for any other platforms.

Code maintainability: Once code is written, it can be modified easily, one time, as it is written only in one place.

[1] *The functional requirements describe the actual functionality the solution and its components should provide.*

RNPR **Productivity**

Scientists write the code once, and need not waste time on repeating the coding process for other hardware platforms. Less time is spent to code an algorithm (even when targeting multiple platforms).

RNLI **General-purpose language integration**

The DSL is not a completely standalone language. Neither is it a general-purpose language. It is intended to work as an extension of another language, normally a general-purpose languages. This way a mixture of grammers allows using existing code repositories, and does not need scientists to learn a new language.

RNDA **Domain abstraction**

During the DSL development, the concepts and operations of the domain science have been abstracted. Abstraction is a key for a successful DSL representation of a domain, and performance portability. The right abstractions make programming easy for scientists based on the rationale of their domain. It also allows them to focus on concepts and operations (algorithms), without being lost with the burden of hardware details and optimization.

For example, data representation instead of arrays is described in a level of abstraction that allows performance portability. They are described such that, after compilation, the real layout of data is chosen to be suitable for the target hardware.

RNFL **Flexibility**

The meta-compilation of a DSL-based code provides a flexible solution. The tools will be designed to flexibly support different DSLs which extend different container languages. The same tools are used to handle modified DSL specifications. Also, different container languages are supported without the need to modify the tools.

To offer the flexibility of the tools, the compilation infrastructure do not comprise any hard-coded DSL or hardware related info. Instead, configuration information drive the tools function. DSL specifications, hardware configurations and compilation options are fed as input to these tools.

RNTS **Tools simplicity**

The translation tool is lightweight, compact, and simple to deal with. Scientists should not be concerned about using it. Despite its complex functionality, it simply integrates into build systems(e.g. make).

RNTM **Tools maintainability**

Maintainability of the tools is an important issue for scientists. So, the translation tool is designed as a file that ships with code repositories just like a script or a makefile. The tools have no external dependencies.

RNCS **DSL code simplicity and readability**

The DSL code is easy to read and understand. That is true because the abstraction development is based on domain-science concepts.

RNCC **Commitment to currrent models source code**

The DSL is developed in a bottom-up approach. This guarantees a more flexible incremental way to port a model's code to DSL. Otherwise, it would be impractical to convert the whole model into DSL in one step. The models are big in terms of code size.

RNBI **Tools integratable into existing build systems**

The tools which will be developed should be integratable into existing build systems e.g. make.

3 Extending Models' Programming Language

Improving the software development process and the performance portability of the three icosahedral models is the driver behind this work. A rewrite of the complete code base with hundreds of thousands of lines is not possible. Therefore, we extended Fortran language with domain-specific concepts relevant to the domain science. The development of the appropriate domain abstractions covered the analysis of the requirements, suggesting abstractions, the discussion and agreement on the suggestions, and the specification of the language extensions.

3.1 Collaborative extension development

To develop the language extensions, we worked together with the domain scientists in a co-design approach:

- The domain scientists each of whom is an expert with one of the three models, have suggested the code parts which are the most relevant. They have chosen the most compute-intensive and time-consuming codes which are the most sensitive in terms of performance.
- An abstraction has been extracted by recognizing the domain concepts and operations in these compute intensive code parts. During this process, we tried to identify commonalities in the three models and create a representation that expresses all three models. Technical requirements for performance were considered during this abstraction process.
- We rewrote codes from the models according to the suggestions.
- We discussed with scientists the abstractions and code examples.

We repeated this process and, thus, iteratively refined the specification until all requirements were met.

3.2 Extensions and Domain-Specific Concepts

In this part, we review the achieved work to extend the Fortran language within each of the three models. We have formed the extensions within each model, and kept in mind that the more common extensions between the three models we can form, the closer we are to define the domain-specific extensions. We discuss the extensions which help to declare variables on the grid. We then discuss the grid definition extensions, which are used to define special sets of cells, edges or vertices of the grid. The necessary extensions to reference the grid variables are then dicussed. We discuss the way to declare the grid variables that are defined over a special set of grid cells, edges, or vertices. We provide an iterator construct to allow stencil operations over the whole grid or subsets of it. The original Fortran code that is written within the iterator is kept, but it is given the ability to reference the grid variables with language extensions. Those extensions ease the coding process, and hide memory layout details. The stencil codes in which an operation is applied over multiple neighbours can also make use of a reduction extension.

3.2.1 Declarations

One of the first domain concepts that we need to start at, is the grid. The grid is a core concept for the climate models. Many concepts, which stem from the grid concept, then come together. The mapping of a space into a grid generates the cells of the grid, and the edges and the vertices of the cells. The variables that are defined over the grid are defined either at the centers of the cells of the grid, on their edges, or at their vertices. The support to declare variables this way is sufficient for the three models, and for finite difference method solutions in general. Based on this, we evolve extensions to declare the grid variables.

Basic grid declaration specifiers

The extensions that we evolve for the grid variable declaration add a set of declaration specifiers to the language. The declaration specifiers tell whether a variable is defined at cell center, on its edges, or at its vertices. They also tell whether it is defined for the whole three-dimensional grid, or a projected two-dimensional surface.

To distinguish where the variable is defined with respect to the grid cells, the following specifications describe the offered extensions. A variable of some type is declared by

```
vartype , CELL[,...] :: varname
```

to indicate that it is defined at cell center. The same way the specifiers are used as

```
vartype , EDGE[,...] :: varname
```

and

```
vartype , VERT[,...] :: varname
```

to indicate that a variable is defined on the cell's edges and at the cell's vertices respectively.

In general, the models use three dimensional grids. But, some variables are defined on two-dimensional grids. To distinguish the grid on which a variable is defined, a specifier is added to the declaration statement. A variable that is defined over a three-dimensional grid, is declared with the corresponding specifier as

```
vartype , CELL/EDGE/VERT, 3D[,...] :: varname
```

The same way, a variable that is defined over a two-dimensional grid, is declared as

```
vartype , CELL/EDGE/VERT, 2D[,...] :: varname
```

We could use the same extensions that are mentioned for the grid variable declaration, for the three models.

ICON dialect

In ICON model scientists use a vertically staggered grid on which most variables are defined at full levels (layer centers), whereas some are defined at the layer interfaces, also referred to as half levels. From this follows that the number of half levels exceeds the number of full levels by one. So, this needs to be also supported and specified by the suggested extensions. A variable defined over a cell/edge/vertex in the three dimensional grid, with half level in vertical direction, of some type is declared by:

```
vartype , CELL/EDGE/VERT, 3D, HL[,...] :: varname
```

Multi-value declaration specifiers

Sometimes we need to connect two values for example, or generally N values, to a grid component, like cell for instance. To support that, we use the following

```
vartype [,...], E(numberOfValues,indexOrder)[,...] :: varname
```

to connect (numberOfValues) entries, and to address it by (indexOrder) index position. For example

```
INTEGER , CELL,3D,E(2,1) :: X
```

connects two integer values for each cell in the 3D grid, and to access the first value we use X(1,cell). That is, we used the number of the integer value as 1 in position 1, before (cell), to access the value. The same way, we use X(2,cell) to access the second value connected to the cell.

3.2.2 Grid Structure

The grid concept is an essential part of all earth system models. Although it is an abstract mathematical concept, it is vital to model natural processes. So, we need the grid structure to locate the different measurements of the different variables in a space. For this reason, we need to abstract the grid structure, in order to address the data values that are connected to a grid's cells/edges/vertices.
The grid is a high level abstraction, which scientific computations essentially depend on. The suggested extensions give the opportunity to have a grid defined and ready to use with some notion, but for flexibility, they also support explicit definition of grid structures.

Basic abstractions

Their are different grid strctures used in different models. In order for the extensions to serve a model and the scientific domain, the grid should be abstracted to a level allowing dealing with the variety of grids. Three of the models we deal with now are icosahedral based while one is using a rectangular grid (ASUCA).
The models define grid variables either at its cell centers, on their edges, or at their vertices. To define and access such grid variables, we abstract those grid concepts within our language extensions. We provide the way to traverse specific subsets of grids cells/edges/vertices.
Grids are of some finite dimensionality. To define a set of a grid's cells, for example, we can describe it through dimensions and the boundaries of each dimension. So, in our extensions we define a set of a grid's cells/edges/vertices in an n-dimensional grid as

```
RANGE, CELL/EDGE/VERT, nD rangename =
 DIM1 {from .. to} * DIM2 {from .. to} * ... * DIMn {from .. to}
```

For example, we define a 3D collection of cells in ICON model by

```
RANGE, CELL, 3D  a = index {1 .. nproma}
 * level {1 .. nlev} * block {1 .. nblks_c}
```

and in NICAM by

```
RANGE, CELL, 3D  a = g {1 .. ADM_gall} * k {1 .. ADM_kall}
 * l {1 .. ADM_lall}
```

and in DYNAMICO by

```
RANGE, CELL, 3D  a = ij {ij_begin .. ij_end} * l {ll_begin .. ll_end}
```

Here is another set of examples on defining the surface cells in a 2D grid. We define a 2D collection of cells in ICON model by

```
RANGE, CELL, 2D  a = index {1 .. nproma} * block {1 .. nblks_c}
```

and in NICAM model we define a 2D collection of cells by

```
RANGE, CELL, 2D  a = g {1 .. ADM_gall} * l {1 .. ADM_lall}
```

and in DYNAMICO model we define a 2D collection of cells by

```
RANGE, CELL, 2D  a = ij {ij_begin .. ij_end}
```

Dimensions and grid

The grid's cells/edges/vertices are defined through dimensions. The range is a set of cells/edges/vertices, which is the cartesian product of a number of dimensions, each of which has defined boundaries.
A dimension's description is fed as

```
dimension_name {dimension_lower_bound .. dimension_higher_bound}
```

and these definitions of dimensions are multipled (Cartesian product) to get a higher dimensional collection of grid cells/edges/vertices.
Based on this, by one step back, we can give the dimension a name as a RANGE, and then use it both as a standalone RANGE, and to build higher dimensional RANGEs. For example, in ICON let's define

```
RANGE, CELL, 2D  a = index {1 .. nproma}
```

and

```
RANGE, CELL, 1D  b = level {1 .. nlev}
```

and

```
RANGE, CELL, 0D  c = block {1 .. nblks_c}
```

Now we can use these RANGEs, and we can define

```
RANGE, CELL, 3D  space = a*b*c
```

to address cells in 3D space.
Let's take an example in NICAM model, let's define

```
RANGE, CELL, 2D  a = g {1 .. ADM_gall}
```

and

```
RANGE, CELL, 1D  b = k {1 .. ADM_kall}
```

and

```
RANGE, CELL, 0D  c = l {1 .. ADM_lall}
```

Now we can define the RANGE

```
RANGE, CELL, 3D  space = a*b*c
```

to address cells in 3D space.
Another example in DYNAMICO, let's define

```
RANGE, CELL, 2D  a = ij {ij_begin .. ij_end}
```

and

```
RANGE, CELL, 1D  b = l {ll_begin .. ll_end}
```

Now we can define the RANGE

```
RANGE, CELL, 3D  space = a*b
```

to address cells in 3D space.
This way, we can use n-dimensional RANGEs, and RANGEs with a subdimensionality of them, while keeping all the named RANGEs of different dimensionalities, to make use of them throughout the source code. That is true because dimensions themselves are basically the simplest RANGEs.

3.2.3 Addressing grid connected variables

As we have seen, the grids and subsets of them are represented by RANGEs allowing referencing grid's variables. These ranges are sets that result from the Cartesian product of the grid's dimensions. So, we can reference any cell (for example) in a grid through a tuple containing an element in the RANGE, or elements of RANGEs comprising it. For example, in ICON

```
Let cell be an element in space (in last examples)
Then
some_var(cell)
and
some_var(cell%index,cell%level,cell%block)
are equivalent
```

and so, we can address variables that are defined on cell centers by many ways. The same for NICAM,

```
Let cell be an element in space
Then
some_var(cell)
and
some_var(cell%g,cell%k,cell%l)
are equivalent
```

and for DYNAMICO,

```
Let cell be an element in space
Then
some_var(cell)
and
some_var(cell%ij,cell%l)
are equivalent
```

This offers more flexibility to support existing models and their needs. We could go lower level, form the higher abstraction level, to use original model (current code without extensions) indexing.

3.2.4 Declaration using RANGEs

In climate/atmospheric models scientists need to deal with variables defined over a grid. We used some rules (refer to section 3.2.1) to declare such variables. Besides, we can use RANGEs alternatively to declare variables. To declare a variable with a value connected to each element in a RANGE, we can use

```
var_type, ON RANGE(range_name) :: var_name
```

For example, we can declare a real value for each cell in the 3D (space) RANGE defined in the examples before, as

```
REAL(WP), ON RANGE(space) :: X
```

which is equivalent to

```
REAL(WP), CELL, 3D :: X
```

which is equivalent to the Fortran code

```
REAL(WP) :: X (1:nproma , 1:nlev , 1:nblks_c)
```

in ICON model, and

```
REAL(WP) :: X (1:ADM_gall , 1:ADM_kall , 1:ADM_lall)
```

in NICAM model, and

```
REAL(WP) :: X (ij_begin:ij_end , ll_begin:ll_end)
```

in DYNAMICO model.
Declaration of variables using the RANGEs allows defining the scope of the variable, and then the variable is defined over a specific set of cells/edges/vertices.

3.2.5 Special-need arrays

We faced cases in ICON where scientists needed to declare arrays with one dimension or two dimensions out of the three dimensions, and with different orders. The idea is to convert some code parts to use the extensions while keeping the existing codes without any changes. Using RANGEs, we can solve this problem by defining a RANGE for the case and using it to declare the array. An example from ICON is

```
RANGE, CELL, XD special = block {1 .. nblks_c} * level {1 .. nlev}
REAL(WP),  ON RANGE(special) :: levmask
```

and so we can address cells in the array as follows

```
Let (curr) be an element in the RANGE (special)
Then we can reach (levmask) values by
levmask(curr)
or if needed we can use
cell%block and cell%level
```

For single dimension arrays, we use a simple one-dimensional RANGE and declare variables over it.

3.2.6 Frequently used RANGEs

We deal with some RANGEs frequently. For example, the set of all grid cells is frequently used in kernels. In fact we need to write it once and use its name throughout code. But also, we can define such RANGEs as default RANGEs in configuration files. In source files written with extensions we can simply use them in simple abstractions such as

```
GRID%cells
GRID%edges
GRID%vertices
```

Refer to fig. 15.
Based on these predefined RANGEs, alongside with operators (section 3.2.7) we can also derive other needed RANGEs instead of defining them from dimension details. More on this in section 3.2.7.

3.2.7 Operators

We can use operators on RANGEs, to define new RANGEs from operand RANGEs, with different options. We can also apply some operators on RANGE elememts. These operators are described in the following subsections.

3.2.7.1 RANGE operator * We have seen in the previous descriptions, that simple RANGEs of one dimension can be multiplied to define a higher dimensionality RANGE. This is one form of operations applied to RANGEs. We can multiply a RANGE with another RANGE as

```
RANGE, CELL, 1D a = DIM1 {from .. to}
RANGE, CELL, 1D b = DIM2 {from .. to}
RANGE, CELL, 2D c = a * b
RANGE, CELL, 2D d = a * DIM3 {from .. to}
```

Refer to fig. 14.

3.2.7.2 RANGE operator / In case it is needed, the extensions support defining lower dimensionality RANGEs from higher dimensionality ones. For example, let's define the 3D RANGE space as

```
RANGE, CELL, 3D space = DIM1 {from .. to} * DIM2 {from .. to} * DIM3 {from .. to}
```

then, based on this RANGE's definition we can define a new 2D RANGE as

```
RANGE, CELL, 2D surface = space / DIM2
```

which is the (space) 3d RANGE with the vertical (assuming DIM2 is the vertical) dimension droped. Refer to fig. 14.

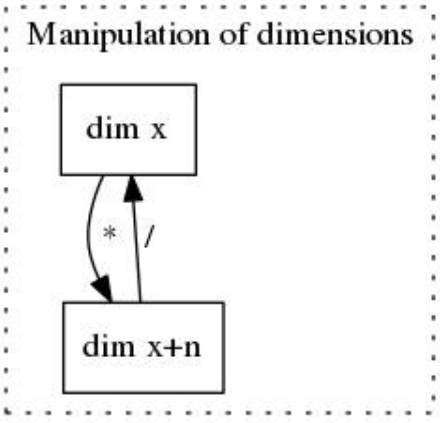


Figure 14: Dimension Operators: higher to lower dimensional grids and vice versa

3.2.7.3 RANGE operator | To inherit a RANGE with its dimention boundaries modified, we use the operator |. For example, assume we have the RANGE

```
RANGE, CELL, 3D a = DIM1 {from .. to} * DIM2 {from .. to} * DIM3 {from .. to}
```

Based on the definition of this RANGE we can define the RANGE

```
RANGE, CELL, 3D b = a | DIM3 {new_from .. new_to}
```

to inherit the RANGE a but with DIM3 dimension modified to new boundaries. This simplifies RANGEs definition instead of defining RANGEs completely from scratch always.

3.2.7.4 RANGE union operator + To simplify referencing grid over regions already defined by existing ranges, the extensions support the set union operation. The extensions offer the '+' operator to represent the union operation of defined ranges. For example, assume we have the ranges

```
RANGE, CELL, 3D a = DIM1 {1 .. 10} * DIM2 {1 .. 10} * levels {1 .. 5}
```

and

```
RANGE, CELL, 3D b = DIM1 {1 .. 10} * DIM2 {1 .. 10} * levels {6 .. 10}
```

then we can use

```
a + b
```

to reference the grid over levels 1 to 10.

3.2.7.5 RANGE exclusion operator - The exclusion of ranges is also supported. This is done by the '-' operator. For example, assume

```
RANGE, CELL, 3D a = DIM1 {1 .. 10} * DIM2 {1 .. 10} * levels {1 .. 10}
```

and

```
RANGE, CELL, 3D a = DIM1 {1 .. 10} * DIM2 {1 .. 10} * levels {1}
```

then we can use

```
a - b
```

to reference grid at levels 2 to 10.

3.2.7.6 RANGE element operator % Another important operator is the % operator used with a grid's component to reach other related components. For example, to reach an edge of a cell we can use cell%edge, and so on. And to reach a cell's neighbouring cell we could use cell%neighbour. Refer to Figure 15.

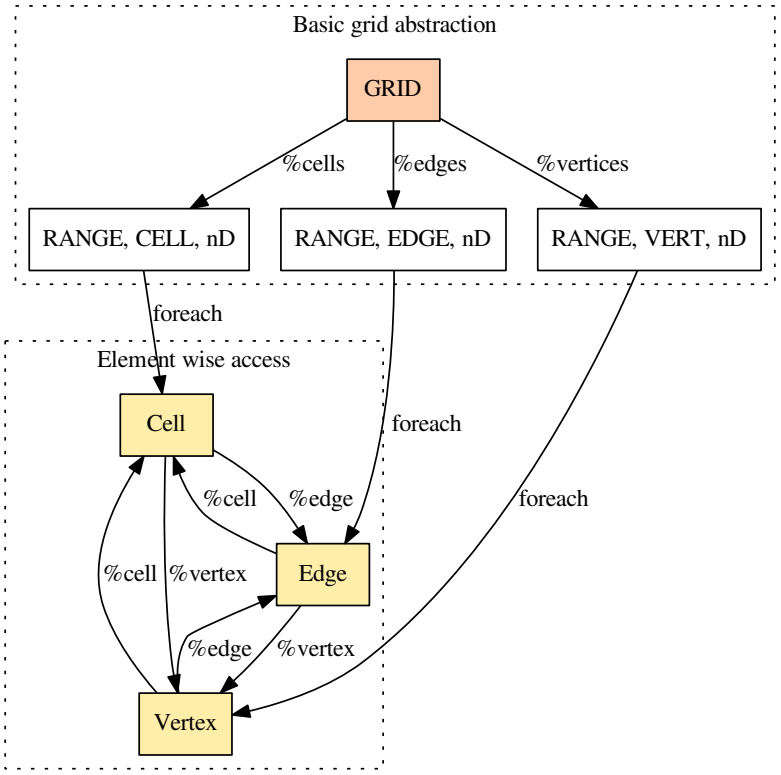


Figure 15: Grid Element Relationships: Simplifying the traversal of the grid elements with language extensions

3.2.8 foreach constructs

The extensions offer a construct to operate on variables defined over a grid. This is essential to write kernel codes in climate models. Based on the RANGEs defined to reflect a grid's structure, we use (foreach) construct to operate on the variables defined over those grids. Refer to fig. 15.

```
Fortran code
FOREACH element IN somerange
  Fortran code with variables referenced by (element)
END FOREACH
Fortran code
```

A foreach statement traverses a range given as one part of the statement. The given range defines which parts of the grid will be traversed. Data of the variables that are defined over those grid elements are accesed within the foreach statement. Also, those grid elements are used to access data indirectly, using grid elements relationships.

```
FOREACH cell IN some_range
  var1(cell)=var2(cell)-var2(cell%below)
END FOREACH
```

Besides to the range to traverse, an element name is given as part of the foreach statement. This allows referencing the current element of the grid we are traversing currently.

3.2.9 reduce constructs

Some mathematical operators are applied over a set of elements in an equation that is used to update a grid variable. We provide an extension to simplify writing such codes. The REDUCE construct applies a mathematical operator to a given code that is repeated with a vriable defined over some range of numbers.

```
REDUCE(operator,variable={from..to},code)
```

The REDUCE construct helps mostly with stencil codes.

4 Code Examples

In this part, we provide code examples from the three models before and after rewriting with the developed dialects. The examples demonstrate using the dialects to define grids, and to manipulate the grid-connected variables. RANGE statements, FOREACH statements, components and neighbor references, and stencil REUDCE constructs are demonstrated. Code examples from ASUCA are also shown at the end of this section.

4.1 ICON

The following Fortran code from the ICON model uses directives to handle optimization for different architectures. The loops are interchanged in order to fit the target architecture in each section. The loop indices, which iterate the grid edges, are used in an indirect addressing to reference some variables defined over the cells around the iterated edges.

Listing 4: ICON Fortran code

```
#ifdef __LOOP_EXCHANGE
 DO je = i_startidx, i_endidx
!DIR$ IVDEP, PREFERVECTOR
  DO jk = nflat_gradp(jg)+1, nlev
#else
 DO jk = nflat_gradp(jg)+1, nlev
  DO je = i_startidx, i_endidx
#endif
   ! horizontal gradient of Exner pressure,
   ! Taylor-expansion-based reconstruction
   z_gradh_exner(je,jk,jb) = p_patch%edges%inv_dual_edge_length(je,jb)* &
    (z_exner_ex_pr(icidx(je,jb,2),ikidx(2,je,jk,jb),icblk(je,jb,2)) +   &
     p_nh%metrics%zdiff_gradp(2,je,jk,jb)*                              &
    (z_dexner_dz_c(1,icidx(je,jb,2),ikidx(2,je,jk,jb),icblk(je,jb,2)) + &
     p_nh%metrics%zdiff_gradp(2,je,jk,jb)*                              &
     z_dexner_dz_c(2,icidx(je,jb,2),ikidx(2,je,jk,jb),icblk(je,jb,2)))- &
    (z_exner_ex_pr(icidx(je,jb,1),ikidx(1,je,jk,jb),icblk(je,jb,1)) +   &
     p_nh%metrics%zdiff_gradp(1,je,jk,jb)*                              &
    (z_dexner_dz_c(1,icidx(je,jb,1),ikidx(1,je,jk,jb),icblk(je,jb,1)) + &
     p_nh%metrics%zdiff_gradp(1,je,jk,jb)*                              &
     z_dexner_dz_c(2,icidx(je,jb,1),ikidx(1,je,jk,jb),icblk(je,jb,1)))))
  ENDDO
 ENDDO
```

Equivalent code rewritten with the ICON's dialect:

Listing 5: ICON DSL code

```
FOREACH edge IN grid%edges
        ! horizontal gradient of Exner pressure,
        ! Taylor-expansion-based reconstruction
            z_gradh_exner(edge) = edge%inv_dual_edge_length*  &
             (z_exner_ex_pr(edge%cell(2)) +                   &
              p_nh%metrics%zdiff_gradp(2,edge)*               &
             (z_dexner_dz_c(edge%cell(2),1) +                 &
              p_nh%metrics%zdiff_gradp(2,edge)*               &
              z_dexner_dz_c(edge%cell(2),2)) -                &
             (z_exner_ex_pr(edge%cell(1)) +                   &
              p_nh%metrics%zdiff_gradp(1,edge)*               &
             (z_dexner_dz_c(edge%cell(1),1) +                 &
              p_nh%metrics%zdiff_gradp(1,edge)*               &
              z_dexner_dz_c(edge%cell(1),2))))
END FOREACH
```

The code written with ICON's dialect uses the iterator that iterates the edges of the grid. The abstract (edge) index is used to reference the variables instead of explicitly using indices that impact the performance because of memory layout. Using (edge%cell) to refer to the cells around an edge simplifies the indirect addressing. This way, the dialect hides the memory layout and connectivity information.

In the following example Fortran code, a variable at the cell center is updated based on the values of a variable on the three edges of the cell. The values of the variable on the edges are accessed indirectly through special arrays.

Listing 6: ICON Fortran code - Example(2)

```
!$OMP DO PRIVATE(jb,i_startidx,i_endidx,jk,jc,z_w_concorr_mc) ICON_OMP_DEFAULT_SCHEDULE
DO jb = i_startblk, i_endblk

  CALL get_indices_c(p_patch, jb, i_startblk, i_endblk, &
                           i_startidx, i_endidx, rl_start, rl_end)

 ! Interpolate contravariant correction to cell centers...
#ifdef __LOOP_EXCHANGE
  DO jc = i_startidx, i_endidx
!DIR$ IVDEP
   DO jk = nflatlev(jg), nlev
#else
  DO jk = nflatlev(jg), nlev
   DO jc = i_startidx, i_endidx
#endif

     z_w_concorr_mc(jc,jk) =  &
       p_int%e_bln_c_s(jc,1,jb)*z_w_concorr_me(ieidx(jc,jb,1),jk,ieblk(jc,jb,1))+&
       p_int%e_bln_c_s(jc,2,jb)*z_w_concorr_me(ieidx(jc,jb,2),jk,ieblk(jc,jb,2))+&
       p_int%e_bln_c_s(jc,3,jb)*z_w_concorr_me(ieidx(jc,jb,3),jk,ieblk(jc,jb,3))
```

Equivalent code rewritten with the ICON's dialect:

Listing 7: ICON DSL code

```
FOREACH cell IN GRID
     z_w_concorr_mc(cell) =  &
       REDUCE(+,N={1..3},p_int%e_bln_c_s(cell,N)*z_w_concorr_me(cell%edge(N))) )
END FOREACH
```

In the code rewritten with the ICON extensions, the arrays used to indirectly access the edge-localized values are removed and special extensions are used to hide connectivity and memory layout. The stencil operation is also reduced by the REDUCE construct.

The extensions not only hide indirect access and connectivity details for the horizontal connectivity, but also vertical neighborhood is also accessible with special extensions as we see in the next example.

Listing 8: ICON Fortran code - Example(3)

```
DO jb = i_startblk, i_endblk

  CALL get_indices_c(p_patch, jb, i_startblk, i_endblk, &
    i_startidx, i_endidx, rl_start, rl_end)
...

    DO jk = 2, nlev
      DO jc = i_startidx, i_endidx
        ! backward trajectory - use w(nnew) in order to be at the same time level as w_concorr
        z_w_backtraj = - (p_nh%prog(nnew)%w(jc,jk,jb) - p_nh%diag%w_concorr_c(jc,jk,jb)) * &
          dtime*0.5_wp/p_nh%metrics%ddqz_z_half(jc,jk,jb)

        ! temporally averaged density and virtual potential temperature depending
        ! on rhotheta_offctr (see pre-computation above)
        z_rho_tavg_m1 = wgt_nnow_rth*p_nh%prog(nnow)%rho(jc,jk-1,jb) + &
                        wgt_nnew_rth*p_nh%prog(nvar)%rho(jc,jk-1,jb)
        z_theta_tavg_m1 = wgt_nnow_rth*p_nh%prog(nnow)%theta_v(jc,jk-1,jb) + &
                          wgt_nnew_rth*p_nh%prog(nvar)%theta_v(jc,jk-1,jb)

        z_rho_tavg = wgt_nnow_rth*p_nh%prog(nnow)%rho(jc,jk,jb) + &
                     wgt_nnew_rth*p_nh%prog(nvar)%rho(jc,jk,jb)
        z_theta_tavg = wgt_nnow_rth*p_nh%prog(nnow)%theta_v(jc,jk,jb) + &
                       wgt_nnew_rth*p_nh%prog(nvar)%theta_v(jc,jk,jb)

        ! density at interface levels for vertical flux divergence computation
        p_nh%diag%rho_ic(jc,jk,jb) = p_nh%metrics%wgtfac_c(jc,jk,jb) *z_rho_tavg    + &
                                (1._wp-p_nh%metrics%wgtfac_c(jc,jk,jb))*z_rho_tavg_m1 + &
          z_w_backtraj*(z_rho_tavg_m1-z_rho_tavg)

        ! perturbation virtual potential temperature at main levels
        z_theta_v_pr_mc_m1  = z_theta_tavg_m1 - p_nh%metrics%theta_ref_mc(jc,jk-1,jb)
        z_theta_v_pr_mc     = z_theta_tavg    - p_nh%metrics%theta_ref_mc(jc,jk,jb)
```

```
        ! perturbation virtual potential temperature at interface levels
        z_theta_v_pr_ic(jc,jk) =                                          &
                 p_nh%metrics%wgtfac_c(jc,jk,jb) *z_theta_v_pr_mc +    &
          (1._vp-p_nh%metrics%wgtfac_c(jc,jk,jb))*z_theta_v_pr_mc_m1

        ! virtual potential temperature at interface levels
        p_nh%diag%theta_v_ic(jc,jk,jb) = p_nh%metrics%wgtfac_c(jc,jk,jb) *z_theta_tavg    +  &
                                  (1._wp-p_nh%metrics%wgtfac_c(jc,jk,jb))*z_theta_tavg_m1 +  &
          z_w_backtraj*(z_theta_tavg_m1-z_theta_tavg)

        ! vertical pressure gradient * theta_v
        z_th_ddz_exner_c(jc,jk,jb) = p_nh%metrics%vwind_expl_wgt(jc,jb)* &
          p_nh%diag%theta_v_ic(jc,jk,jb) * (z_exner_pr(jc,jk-1,jb)-      &
          z_exner_pr(jc,jk,jb)) / p_nh%metrics%ddqz_z_half(jc,jk,jb) +   &
          z_theta_v_pr_ic(jc,jk)*p_nh%metrics%d_exner_dz_ref_ic(jc,jk,jb)
      ENDDO
    ENDDO
...
ENDDO
```

Equivalent code rewritten with the ICON's dialect:

Listing 9: ICON DSL code

```
foreach cell in grid
        ! backward trajectory - use w(nnew) in order to be at the same time level as w_concorr
        z_w_backtraj = - (p_nh%prog(nnew)%w(cell) - p_nh%diag%w_concorr_c(cell)) * &
          dtime*0.5_wp/p_nh%metrics%ddqz_z_half(cell)

        ! temporally averaged density and virtual potential temperature depending on
        ! rhotheta_offctr (see pre-computation above)
        z_rho_tavg_m1 = wgt_nnow_rth*p_nh%prog(nnow)%rho(cell%below) + &
                        wgt_nnew_rth*p_nh%prog(nvar)%rho(cell%below)
        z_theta_tavg_m1 = wgt_nnow_rth*p_nh%prog(nnow)%theta_v(cell%below) + &
                          wgt_nnew_rth*p_nh%prog(nvar)%theta_v(cell%below)

        z_rho_tavg = wgt_nnow_rth*p_nh%prog(nnow)%rho(cell) + &
                     wgt_nnew_rth*p_nh%prog(nvar)%rho(cell)
        z_theta_tavg = wgt_nnow_rth*p_nh%prog(nnow)%theta_v(cell) + &
                       wgt_nnew_rth*p_nh%prog(nvar)%theta_v(cell)

        ! density at interface levels for vertical flux divergence computation
        p_nh%diag%rho_ic(cell) = p_nh%metrics%wgtfac_c(cell) *z_rho_tavg    + &
                                 (1._wp-p_nh%metrics%wgtfac_c(cell))*z_rho_tavg_m1 + &
          z_w_backtraj*(z_rho_tavg_m1-z_rho_tavg)

        ! perturbation virtual potential temperature at main levels
        z_theta_v_pr_mc_m1  = z_theta_tavg_m1 - p_nh%metrics%theta_ref_mc(cell%below)
        z_theta_v_pr_mc     = z_theta_tavg    - p_nh%metrics%theta_ref_mc(cell)

        ! perturbation virtual potential temperature at interface levels
        z_theta_v_pr_ic(cell) =                                           &
                 p_nh%metrics%wgtfac_c(cell) *z_theta_v_pr_mc +    &
          (1._vp-p_nh%metrics%wgtfac_c(cell))*z_theta_v_pr_mc_m1

        ! virtual potential temperature at interface levels
        p_nh%diag%theta_v_ic(cell) = p_nh%metrics%wgtfac_c(cell) *z_theta_tavg    +  &
                                  (1._wp-p_nh%metrics%wgtfac_c(cell))*z_theta_tavg_m1 +  &
          z_w_backtraj*(z_theta_tavg_m1-z_theta_tavg)

        ! vertical pressure gradient * theta_v
        z_th_ddz_exner_c(cell) = p_nh%metrics%vwind_expl_wgt(cell)* &
          p_nh%diag%theta_v_ic(cell) * (z_exner_pr(cell%below)-      &
          z_exner_pr(cell)) / p_nh%metrics%ddqz_z_half(cell) +   &
          z_theta_v_pr_ic(cell)*p_nh%metrics%d_exner_dz_ref_ic(cell)
end foreach
```

In the code version that is written using the ICON extensions, (cell%above) and (cell%below) are used to access values vertically neighboring the cell.

Vertical integration is show in the next example. A two-dimensional variable is updated based on the values of another three-dimensional variable.

Listing 10: ICON Fortran code - Example(4)

```
DO jb = i_startblk, i_endblk

  CALL get_indices_c(p_patch, jb, i_startblk, i_endblk, &
    i_startidx, i_endidx, rl_start, rl_end)

...

      z_thermal_exp(:,jb) = 0._wp
      DO jk = 1, nlev
        DO jc = i_startidx, i_endidx
          z_thermal_exp(jc,jb) = z_thermal_exp(jc,jb) + cvd_o_rd                    &
            * p_nh%diag%ddt_exner_phy(jc,jk,jb)                                     &
            /  (p_nh%prog(nnow)%exner(jc,jk,jb)*p_nh%metrics%inv_ddqz_z_full(jc,jk,jb))
        ENDDO
      ENDDO

...
ENDDO
```

Equivalent code rewritten with the ICON's dialect:

Listing 11: ICON DSL code

```
foreach cell in grid2D
  z_thermal_exp(cell%horizontal) = 0._wp
end foreach

foreach cell in grid
  z_thermal_exp(cell%horizontal) = z_thermal_exp(cell%horizontal) + cvd_o_rd    &
    * p_nh%diag%ddt_exner_phy(cell)                                             &
    /  (p_nh%prog(nnow)%exner(cell)*p_nh%metrics%inv_ddqz_z_full(cell))
end foreach
```

The two- and three-dimensional grid variables are accesses with ICON extensions.

In the following example, the values of a variable at the vertices of an edge and on the cells sharing the edge are accessed indirectly using specialized arrays.

Listing 12: ICON Fortran code - Example(5)

```
DO jb = i_startblk, i_endblk

  CALL get_indices_e(p_patch, jb, i_startblk, i_endblk, &
                     i_startidx, i_endidx, rl_start, rl_end)

...

#ifdef __LOOP_EXCHANGE
    DO je = i_startidx, i_endidx
      DO jk = 1, nlev
#else
    DO jk = 1, nlev
      DO je = i_startidx, i_endidx
#endif

! Compute upwind-biased values for rho and theta starting from centered differences
! Note: the length of the backward trajectory should be 0.5*dtime*(vn,vt) in order to
! arrive at a second-order accurate FV discretization, but twice the length is needed
! for numerical stability
z_rho_e(je,jk,jb) =                                                             &
  p_int%c_lin_e(je,1,jb)*p_nh%prog(nnow)%rho(icidx(je,jb,1),jk,icblk(je,jb,1)) +  &
  p_int%c_lin_e(je,2,jb)*p_nh%prog(nnow)%rho(icidx(je,jb,2),jk,icblk(je,jb,2)) -  &
  dtime * (p_nh%prog(nnow)%vn(je,jk,jb)*p_patch%edges%inv_dual_edge_length(je,jb)*  &
 (p_nh%prog(nnow)%rho(icidx(je,jb,2),jk,icblk(je,jb,2)) -                         &
  p_nh%prog(nnow)%rho(icidx(je,jb,1),jk,icblk(je,jb,1)) ) + p_nh%diag%vt(je,jk,jb) *  &
  p_patch%edges%inv_primal_edge_length(je,jb) * p_patch%edges%tangent_orientation(je,jb) *  &
 (z_rho_v(ividx(je,jb,2),jk,ivblk(je,jb,2)) - z_rho_v(ividx(je,jb,1),jk,ivblk(je,jb,1)) ) )

z_theta_v_e(je,jk,jb) =                                                         &
  p_int%c_lin_e(je,1,jb)*p_nh%prog(nnow)%theta_v(icidx(je,jb,1),jk,icblk(je,jb,1)) +  &
  p_int%c_lin_e(je,2,jb)*p_nh%prog(nnow)%theta_v(icidx(je,jb,2),jk,icblk(je,jb,2)) -  &
  dtime * (p_nh%prog(nnow)%vn(je,jk,jb)*p_patch%edges%inv_dual_edge_length(je,jb)*  &
```

```
  (p_nh%prog(nnow)%theta_v(icidx(je,jb,2),jk,icblk(je,jb,2)) -                                    &
   p_nh%prog(nnow)%theta_v(icidx(je,jb,1),jk,icblk(je,jb,1)) ) + p_nh%diag%vt(je,jk,jb) *         &
   p_patch%edges%inv_primal_edge_length(je,jb) * p_patch%edges%tangent_orientation(je,jb) *       &
  (z_theta_v_v(ividx(je,jb,2),jk,ivblk(je,jb,2)) - z_theta_v_v(ividx(je,jb,1),jk,ivblk(je,jb,1))))

      ENDDO ! loop over edges
    ENDDO   ! loop over vertical levels

...
ENDDO
```

Equivalent code rewritten with the ICON's dialect:

Listing 13: ICON DSL code

```
foreach edge in grid
        ! Compute upwind-biased values for rho and theta starting from centered differences
        ! Note: the length of the backward trajectory should be 0.5*dtime*(vn,vt) in order to
        ! arrive at a second-order accurate FV discretization, but twice the length is needed
        ! for numerical stability
        z_rho_e(edge) =                                                                  &
          p_int%c_lin_e(edge,1)*p_nh%prog(nnow)%rho(edge%cell(1)) +                      &
          p_int%c_lin_e(edge,2)*p_nh%prog(nnow)%rho(edge%cell(2)) -                      &
          dtime * (p_nh%prog(nnow)%vn(edge)*p_patch%edges%inv_dual_edge_length(edge)* &
         (p_nh%prog(nnow)%rho(edge%cell(2)) -                                            &
          p_nh%prog(nnow)%rho(edge%cell(1)) ) + p_nh%diag%vt(edge) *                     &
          p_patch%edges%inv_primal_edge_length(edge) * p_patch%edges%tangent_orientation(edge) *&
         (z_rho_v(edge%vertex(2)) - z_rho_v(edge%vertex(1)) ) )

        z_theta_v_e(edge) =                                                              &
          p_int%c_lin_e(edge,1)*p_nh%prog(nnow)%theta_v(edge%cell(1)) +                  &
          p_int%c_lin_e(edge,2)*p_nh%prog(nnow)%theta_v(edge%cell(2)) -                  &
          dtime * (p_nh%prog(nnow)%vn(edge)*p_patch%edges%inv_dual_edge_length(edge)* &
         (p_nh%prog(nnow)%theta_v(edge%cell(2)) -                                        &
          p_nh%prog(nnow)%theta_v(edge%cell(1)) ) + p_nh%diag%vt(edge) *                 &
          p_patch%edges%inv_primal_edge_length(edge) * p_patch%edges%tangent_orientation(edge) *&
         (z_theta_v_v(edge%vertex(2)) - z_theta_v_v(edge%vertex(1)) ))

end foreach
```

In the new code that uses the ICON extensions, the indirect access to the variables at the vertices and on the cells is handled with the abstractions (edge%vertex) and (edge%cell) respectively.

The last example shows a more complex code that was written with sections to account for architecture. Scalar variables in one section are rewritten as arrays in the other section to transfer temporary calculation results between loops.

Listing 14: ICON Fortran code - Example(6)

```
DO jb = i_startblk, i_endblk

  CALL get_indices_e(p_patch, jb, i_startblk, i_endblk, &
                     i_startidx, i_endidx, rl_start, rl_end)

...


#ifdef __LOOP_EXCHANGE
    DO je = i_startidx, i_endidx
      DO jk = 1, nlev

        lvn_pos = p_nh%prog(nnow)%vn(je,jk,jb) >= 0._wp

        ! line and block indices of upwind neighbor cell
        ilc0 = MERGE(p_patch%edges%cell_idx(je,jb,1),p_patch%edges%cell_idx(je,jb,2),lvn_pos)
        ibc0 = MERGE(p_patch%edges%cell_blk(je,jb,1),p_patch%edges%cell_blk(je,jb,2),lvn_pos)

        ! distances from upwind mass point to the end point of the backward trajectory
        ! in edge-normal and tangential directions
        z_ntdistv_bary_1 =  - ( p_nh%prog(nnow)%vn(je,jk,jb) * dthalf +    &
          MERGE(p_int%pos_on_tplane_e(je,jb,1,1), p_int%pos_on_tplane_e(je,jb,2,1),lvn_pos))

        z_ntdistv_bary_2 =  - ( p_nh%diag%vt(je,jk,jb) * dthalf +    &
```

```
            MERGE(p_int%pos_on_tplane_e(je,jb,1,2), p_int%pos_on_tplane_e(je,jb,2,2),lvn_pos))

          ! rotate distance vectors into local lat-lon coordinates:
          !
          ! component in longitudinal direction
          distv_bary_1 =                                                                  &
                z_ntdistv_bary_1*MERGE(p_patch%edges%primal_normal_cell(je,jb,1)%v1,      &
                                       p_patch%edges%primal_normal_cell(je,jb,2)%v1,lvn_pos) &
              + z_ntdistv_bary_2*MERGE(p_patch%edges%dual_normal_cell(je,jb,1)%v1,       &
                                       p_patch%edges%dual_normal_cell(je,jb,2)%v1,lvn_pos)

          ! component in latitudinal direction
          distv_bary_2 =                                                                  &
                z_ntdistv_bary_1*MERGE(p_patch%edges%primal_normal_cell(je,jb,1)%v2,      &
                                       p_patch%edges%primal_normal_cell(je,jb,2)%v2,lvn_pos) &
              + z_ntdistv_bary_2*MERGE(p_patch%edges%dual_normal_cell(je,jb,1)%v2,       &
                                       p_patch%edges%dual_normal_cell(je,jb,2)%v2,lvn_pos)


          ! Calculate "edge values" of rho and theta_v
          ! Note: z_rth_pr contains the perturbation values of rho and theta_v,
          ! and the corresponding gradients are stored in z_grad_rth.
          z_rho_e(je,jk,jb) = p_nh%metrics%rho_ref_me(je,jk,jb)     &
            +                           z_rth_pr(1,ilc0,jk,ibc0)   &
            + distv_bary_1 * z_grad_rth(1,ilc0,jk,ibc0) &
            + distv_bary_2 * z_grad_rth(2,ilc0,jk,ibc0)

          z_theta_v_e(je,jk,jb) = p_nh%metrics%theta_ref_me(je,jk,jb) &
            +                           z_rth_pr(2,ilc0,jk,ibc0)     &
            + distv_bary_1 * z_grad_rth(3,ilc0,jk,ibc0)   &
            + distv_bary_2 * z_grad_rth(4,ilc0,jk,ibc0)

        ENDDO ! loop over edges
      ENDDO   ! loop over vertical levels

#else

! [first loop encapsulated in subroutine 'btraj' in ICON code]

      DO jk = slev, elev
        DO je = i_startidx, i_endidx

          !
          ! Calculate backward trajectories
          !

          ! position of barycenter in normal direction
          ! pos_barycenter(1) = - p_vn(je,jk,jb) * p_dthalf

          ! position of barycenter in tangential direction
          ! pos_barycenter(2) = - p_vt(je,jk,jb) * p_dthalf

          ! logical auxiliary for MERGE operations: .TRUE. for vn >= 0
          lvn_pos = p_vn(je,jk,jb) >= 0._wp

          ! If vn > 0 (vn < 0), the upwind cell is cell 1 (cell 2)

          ! line and block indices of neighbor cell with barycenter
          z_cell_idx(je,jk,jb) = &
             &   MERGE(ptr_p%edges%cell_idx(je,jb,1),ptr_p%edges%cell_idx(je,jb,2),lvn_pos)

          z_cell_blk(je,jk,jb) = &
             &   MERGE(ptr_p%edges%cell_blk(je,jb,1),ptr_p%edges%cell_blk(je,jb,2),lvn_pos)


          ! Calculate the distance cell center --> barycenter for the cell,
          ! in which the barycenter is located. The distance vector points
          ! from the cell center to the barycenter.
          z_ntdistv_bary(1) =  - ( p_vn(je,jk,jb) * p_dthalf     &
               & + MERGE(ptr_int%pos_on_tplane_e(je,jb,1,1),        &
               &         ptr_int%pos_on_tplane_e(je,jb,2,1),lvn_pos))

          z_ntdistv_bary(2) =  - ( p_vt(je,jk,jb) * p_dthalf     &
               & + MERGE(ptr_int%pos_on_tplane_e(je,jb,1,2),        &
               &         ptr_int%pos_on_tplane_e(je,jb,2,2),lvn_pos))
```

```
          ! In a last step, transform this distance vector into a rotated
          ! geographical coordinate system with its origin at the circumcenter
          ! of the upstream cell. Coordinate axes point to local East and local
          ! North.

          ! component in longitudinal direction
          z_distv_bary(je,jk,jb,1) =                                                    &
               &    z_ntdistv_bary(1)*MERGE(ptr_p%edges%primal_normal_cell(je,jb,1)%v1,          &
               &                            ptr_p%edges%primal_normal_cell(je,jb,2)%v1,lvn_pos) &
               & + z_ntdistv_bary(2)*MERGE(ptr_p%edges%dual_normal_cell(je,jb,1)%v1,            &
               &                            ptr_p%edges%dual_normal_cell(je,jb,2)%v1,lvn_pos)

          ! component in latitudinal direction
          z_distv_bary(je,jk,jb,2) =                                                    &
               &    z_ntdistv_bary(1)*MERGE(ptr_p%edges%primal_normal_cell(je,jb,1)%v2,          &
               &                            ptr_p%edges%primal_normal_cell(je,jb,2)%v2,lvn_pos) &
               & + z_ntdistv_bary(2)*MERGE(ptr_p%edges%dual_normal_cell(je,jb,1)%v2,            &
               &                            ptr_p%edges%dual_normal_cell(je,jb,2)%v2,lvn_pos)

        ENDDO ! loop over edges
      ENDDO   ! loop over vertical levels



          DO jk = 1, nlev
            DO je = i_startidx, i_endidx

              ilc0 = z_cell_idx(je,jk,jb)
              ibc0 = z_cell_blk(je,jk,jb)

              ! Calculate "edge values" of rho and theta_v
              ! Note: z_rth_pr contains the perturbation values of rho and theta_v,
              ! and the corresponding gradients are stored in z_grad_rth.
              z_rho_e(je,jk,jb) = p_nh%metrics%rho_ref_me(je,jk,jb)     &
                +                         z_rth_pr(1,ilc0,jk,ibc0)    &
                + z_distv_bary(je,jk,jb,1) * z_grad_rth(1,ilc0,jk,ibc0) &
                + z_distv_bary(je,jk,jb,2) * z_grad_rth(2,ilc0,jk,ibc0)

              z_theta_v_e(je,jk,jb) = p_nh%metrics%theta_ref_me(je,jk,jb) &
                +                         z_rth_pr(2,ilc0,jk,ibc0)     &
                + z_distv_bary(je,jk,jb,1) * z_grad_rth(3,ilc0,jk,ibc0)   &
                + z_distv_bary(je,jk,jb,2) * z_grad_rth(4,ilc0,jk,ibc0)

            ENDDO ! loop over edges
          ENDDO   ! loop over vertical levels


#endif

...
        ENDDO
```

Equivalent code rewritten with the ICON's dialect:

Listing 15: ICON DSL code

```
foreach edge in grid
    if( p_nh%prog(nnow)%vn(edge) >= 0._wp)
          z_ntdistv_bary_1 =  - ( p_nh%prog(nnow)%vn(edge) * dthalf
                              + p_int%pos_on_tplane_e(edge%cell(1),1) )
          z_ntdistv_bary_2 =  - ( p_nh%diag%vt(edge) * dthalf
                              + p_int%pos_on_tplane_e(edge%cell(1),2) )

          distv_bary_1 = z_ntdistv_bary_1*edge%normalPcell(1)%v1
                       + z_ntdistv_bary_2*edge%normalDcell(1)%v1
          distv_bary_2 = z_ntdistv_bary_1*edge%normalPcell(1)%v2
                       + z_ntdistv_bary_2*edge%normalDcell(1)%v2

          z_rho_e(edge) = p_nh%metrics%rho_ref_me(edge) + z_rth_pr(1,edge%cell(1))
                        + distv_bary_1 * z_grad_rth(1,edge%cell(1))
                        + distv_bary_2 * z_grad_rth(2,edge%cell(1))
          z_theta_v_e(edge) = p_nh%metrics%theta_ref_me(edge) + z_rth_pr(2,edge%cell(1))
                            + distv_bary_1 * z_grad_rth(3,edge%cell(1))
                            + distv_bary_2 * z_grad_rth(4,edge%cell(1))
```

```
      else
            z_ntdistv_bary_1 =  - ( p_nh%prog(nnow)%vn(edge) * dthalf
                                + p_int%pos_on_tplane_e(edge%cell(2),1) )
            z_ntdistv_bary_2 =  - ( p_nh%diag%vt(edge) * dthalf
                                + p_int%pos_on_tplane_e(edge%cell(2),2) )

            distv_bary_1 = z_ntdistv_bary_1*edge%normalPcell(2)%v1
                         + z_ntdistv_bary_2*edge%normalDcell(2)%v1
            distv_bary_2 = z_ntdistv_bary_1*edge%normalPcell(2)%v2
                         + z_ntdistv_bary_2*edge%normalDcell(2)%v2

            z_rho_e(edge) = p_nh%metrics%rho_ref_me(edge) + z_rth_pr(1,edge%cell(2))
                          + distv_bary_1 * z_grad_rth(1,edge%cell(2))
                          + distv_bary_2 * z_grad_rth(2,edge%cell(2))
            z_theta_v_e(edge) = p_nh%metrics%theta_ref_me(edge) + z_rth_pr(2,edge%cell(2))
                            + distv_bary_1 * z_grad_rth(3,edge%cell(2))
                            + distv_bary_2 * z_grad_rth(4,edge%cell(2))
      endif
   end foreach
```

4.2 DYNAMICO

The following Fortran code from the DYNAMICO model uses two nested loops with a directive to vectorize the inner loop which iterates the horizontal grid. The horizontal loop index is used to calculate the indices of the neighbors in a stencil operation.

Listing 16: DYNAMICO Fortran code

```
DO l=ll_begin,ll_end
!DIR$ SIMD
  DO ij=ij_begin,ij_end

   berni(ij,l) = .5*(geopot(ij,l)+geopot(ij,l+1)) &
    + 1/(4*Ai(ij))*(le(ij+u_right)*de(ij+u_right)*u(ij+u_right,l)**2 + &
                    le(ij+u_rup)*de(ij+u_rup)*u(ij+u_rup,l)**2 +       &
                    le(ij+u_lup)*de(ij+u_lup)*u(ij+u_lup,l)**2 +       &
                    le(ij+u_left)*de(ij+u_left)*u(ij+u_left,l)**2 +    &
                    le(ij+u_ldown)*de(ij+u_ldown)*u(ij+u_ldown,l)**2 + &
                    le(ij+u_rdown)*de(ij+u_rdown)*u(ij+u_rdown,l)**2 )
  ENDDO
ENDDO
```

Equivalent code rewritten with the DYNAMICO's dialect:

Listing 17: DYNAMICO DSL code

```
RANGE,CELL,3D gc = ij{ ij_omp_begin_ext .. ij_omp_end_ext }*l {1 .. llm}

FOREACH cell IN gc
 berni(cell) = .5*(geopot(cell)+geopot(cell%above)) + 1/(4*Ai(cell%ij))
  * REDUCE(+, N={1..6}
  le(cell%neighbour(N)%ij)*de(cell%neighbour(N)%ij)
                                          *u(cell%neighbour(N))**2)
END FOREACH
```

The rewritten code uses the FOREACH statement to iterate the set of cells and update the variable (berni) at each of the iterated cells. Using the REDUCE extension along with the (cell%neighbour) to refer to the neighbours of a cell simplifies the code of the stencil operation and eliminates the duplicate code over each neighbour.

4.3 NICAM

The following Fortran code from the NICAM model uses three nested loops with an OpenCL directive to harness parallel execution capabilities. The code defines variables to help calculate the indices that are necessary to reference the variables over the neighbour cells within the stencil operation.

Listing 18: NICAM Fortran code

```
      do d = 1, ADM_nxyz
```

```
   do l = 1, ADM_lall
   do k = 1, ADM_kall
      do n = OPRT_nstart, OPRT_nend
         ij     = n
         ip1j   = n + 1
         ijp1   = n     + ADM_gall_1d
         ip1jp1 = n + 1 + ADM_gall_1d
         im1j   = n - 1
         ijm1   = n     - ADM_gall_1d
         im1jm1 = n - 1 - ADM_gall_1d

         grad(n,k,l,d) = cgrad(n,l,0,d) * scl(ij    ,k,l) &
                       + cgrad(n,l,1,d) * scl(ip1j  ,k,l) &
                       + cgrad(n,l,2,d) * scl(ip1jp1,k,l) &
                       + cgrad(n,l,3,d) * scl(ijp1  ,k,l) &
                       + cgrad(n,l,4,d) * scl(im1j  ,k,l) &
                       + cgrad(n,l,5,d) * scl(im1jm1,k,l) &
                       + cgrad(n,l,6,d) * scl(ijm1  ,k,l)
      enddo
      grad(          1:OPRT_nstart-1,k,l,d) = 0.0_RP
      grad(OPRT_nend+1:ADM_gall     ,k,l,d) = 0.0_RP
   enddo
   enddo
   enddo
```

Equivalent code rewritten with the NICAM's dialect:

Listing 19: NICAM DSL code

```
RANGE, CELL, 3D g1 =  GRID%cells | g{OPRT_nstart..OPRT_nend}
FOREACH cell in g1
  do d = 1, ADM_nxyz
       grad(cell,d) = REDUCE(+,N={0..6},
                     cgrad(cell%g,cell%l,N,d) * scl(cell%neighbor(N)) )
  enddo
END FOREACH

FOREACH cell in GRID%cells | g{1..OPRT_nstart-1 , OPRT_nend+1 .. gall}
  do d = 1, ADM_nxyz
          grad(cell,d) = 0.0_PRECISION
  enddo
END FOREACH
```

Using the operators of the NICAM model's dialect to define the RANGE in the first line helps iterating a subset of the grid cells. Within the iterator, the use of (cell%neighbour) removes the complexity of the index calculations which are necessary to reference neighbours. Also the REDUCE extension simplifies the stencil operation over the neighbours.

The following example traverses six neighbors to compute a variable with three components –x,y, and z. The code computes the memory location of each value used in the computation.

Listing 20: NICAM Fortran code -example(2)

```
   gall    = ADM_gall
   gall_1d = ADM_gall_1d
   kall    = ADM_kall

   do l = 1, ADM_lall
      !$omp parallel default(none),private(n,k,ij,ip1j,ip1jp1,ijp1,im1j,ijm1,im1jm1), &
      !$omp shared(OPRT_nstart,OPRT_nend,gall,gall_1d,kall,l,scl,sclx,scly,sclz,cdiv,vx,vy,vz)
      do k = 1, kall

         !$omp do
         do n = OPRT_nstart, OPRT_nend
            ij     = n
            ip1j   = n + 1
            ijp1   = n     + gall_1d
            ip1jp1 = n + 1 + gall_1d
            im1j   = n - 1
            ijm1   = n     - gall_1d
            im1jm1 = n - 1 - gall_1d

            sclx(n) = cdiv(n,l,0,1) * vx(ij    ,k,l) &
```

```
                    + cdiv(n,l,1,1) * vx(ip1j  ,k,l) &
                    + cdiv(n,l,2,1) * vx(ip1jp1,k,l) &
                    + cdiv(n,l,3,1) * vx(ijp1  ,k,l) &
                    + cdiv(n,l,4,1) * vx(im1j  ,k,l) &
                    + cdiv(n,l,5,1) * vx(im1jm1,k,l) &
                    + cdiv(n,l,6,1) * vx(ijm1  ,k,l)
          enddo
          !$omp end do nowait

          !$omp do
          do n = OPRT_nstart, OPRT_nend
             ij     = n
             ip1j   = n + 1
             ijp1   = n     + gall_1d
             ip1jp1 = n + 1 + gall_1d
             im1j   = n - 1
             ijm1   = n     - gall_1d
             im1jm1 = n - 1 - gall_1d

             scly(n) = cdiv(n,l,0,2) * vy(ij    ,k,l) &
                     + cdiv(n,l,1,2) * vy(ip1j  ,k,l) &
                     + cdiv(n,l,2,2) * vy(ip1jp1,k,l) &
                     + cdiv(n,l,3,2) * vy(ijp1  ,k,l) &
                     + cdiv(n,l,4,2) * vy(im1j  ,k,l) &
                     + cdiv(n,l,5,2) * vy(im1jm1,k,l) &
                     + cdiv(n,l,6,2) * vy(ijm1  ,k,l)
          enddo
          !$omp end do nowait

          !$omp do
          do n = OPRT_nstart, OPRT_nend
             ij     = n
             ip1j   = n + 1
             ijp1   = n     + gall_1d
             ip1jp1 = n + 1 + gall_1d
             im1j   = n - 1
             ijm1   = n     - gall_1d
             im1jm1 = n - 1 - gall_1d

             sclz(n) = cdiv(n,l,0,3) * vz(ij    ,k,l) &
                     + cdiv(n,l,1,3) * vz(ip1j  ,k,l) &
                     + cdiv(n,l,2,3) * vz(ip1jp1,k,l) &
                     + cdiv(n,l,3,3) * vz(ijp1  ,k,l) &
                     + cdiv(n,l,4,3) * vz(im1j  ,k,l) &
                     + cdiv(n,l,5,3) * vz(im1jm1,k,l) &
                     + cdiv(n,l,6,3) * vz(ijm1  ,k,l)
          enddo
          !$omp end do nowait

          !$omp do
          do n = 1, OPRT_nstart-1
             scl(n,k,l) = 0.0_RP
          enddo
          !$omp end do nowait

          !$omp do
          do n = OPRT_nend+1, gall
             scl(n,k,l) = 0.0_RP
          enddo
          !$omp end do

          !$omp do
          do n = OPRT_nstart, OPRT_nend
             scl(n,k,l) = sclx(n) + scly(n) + sclz(n)
          enddo
          !$omp end do

       enddo
       !$omp end parallel
    enddo
```

Equivalent code rewritten with the NICAM's dialect:

Listing 21: NICAM DSL code

```
RANGE, CELL, 3D g1 =  GRID%cells | g{OPRT_nstart..OPRT_nend}
foreach cell in g1 !!! or directly without g1: in GRID%cells | g{OPRT_nstart..OPRT_nend}

           scl(cell) = REDUCE(+,D={1..3},
                REDUCE(+,N={0..6},
      cdiv(cell%g,cell%l,N,D) * v(cell%neighbor(N),D)
      )
      )
endforeach

foreach cell in GRID%cells | g{1..OPRT_nstart-1 , OPRT_nend+1 .. gall}
           scl(cell) = 0.0_PRECISION
endforeach
```

In the new code rewritten with NICAM extensions, the code computation is reduced twice; for both the three components, and the six neighbors. The memory calculations are dropped from the source code. The memory layout is abstracted by the extensions.

The next example also uses memory location calculation to access values on six neighbors. The computed variable is composed of as many components as the number of dimensions.

Listing 22: NICAM Fortran code -example(3)

```
   do d = 1, ADM_nxyz
   do l = 1, ADM_lall
!OCL PARALLEL
   do k = 1, ADM_kall
      do n = OPRT_nstart, OPRT_nend
         ij     = n
         ip1j   = n + 1
         ijp1   = n     + ADM_gall_1d
         ip1jp1 = n + 1 + ADM_gall_1d
         im1j   = n - 1
         ijm1   = n     - ADM_gall_1d
         im1jm1 = n - 1 - ADM_gall_1d

         grad(n,k,l,d) = cgrad(n,l,0,d) * scl(ij    ,k,l) &
                       + cgrad(n,l,1,d) * scl(ip1j  ,k,l) &
                       + cgrad(n,l,2,d) * scl(ip1jp1,k,l) &
                       + cgrad(n,l,3,d) * scl(ijp1  ,k,l) &
                       + cgrad(n,l,4,d) * scl(im1j  ,k,l) &
                       + cgrad(n,l,5,d) * scl(im1jm1,k,l) &
                       + cgrad(n,l,6,d) * scl(ijm1  ,k,l)
      enddo
      grad(          1:OPRT_nstart-1,k,l,d) = 0.0_RP
      grad(OPRT_nend+1:ADM_gall     ,k,l,d) = 0.0_RP
   enddo
   enddo
   enddo
```

Equivalent code rewritten with the NICAM's dialect:

Listing 23: NICAM DSL code

```
RANGE, CELL, 3D g1 =  GRID%cells | g{OPRT_nstart..OPRT_nend}
foreach cell in g1
  do d = 1, ADM_nxyz
        grad(cell,d) = REDUCE(+,N={0..6},
    cgrad(cell%g,cell%l,N,d) * scl(cell%neighbor(N))
    )
  enddo
endforeach

foreach cell in GRID%cells | g{1..OPRT_nstart-1 , OPRT_nend+1 .. gall}
  do d = 1, ADM_nxyz
           grad(cell,d) = 0.0_PRECISION
  enddo
endforeach
```

The REDUCE in the new code allows to represent the computation that envolves the six neighbors. The variable components are iterated in the loop within the iterator.

The same way of neighbor-based computation, but with one-component variable is shown in the next example.

Listing 24: NICAM Fortran code -example(4)

```
    do l = 1, ADM_lall
!OCL PARALLEL
    do k = 1, ADM_kall
       do n = OPRT_nstart, OPRT_nend
          ij     = n
          ip1j   = n + 1
          ijp1   = n     + ADM_gall_1d
          ip1jp1 = n + 1 + ADM_gall_1d
          im1j   = n - 1
          ijm1   = n     - ADM_gall_1d
          im1jm1 = n - 1 - ADM_gall_1d

          dscl(n,k,l) = clap(n,l,0) * scl(ij    ,k,l) &
                      + clap(n,l,1) * scl(ip1j  ,k,l) &
                      + clap(n,l,2) * scl(ip1jp1,k,l) &
                      + clap(n,l,3) * scl(ijp1  ,k,l) &
                      + clap(n,l,4) * scl(im1j  ,k,l) &
                      + clap(n,l,5) * scl(im1jm1,k,l) &
                      + clap(n,l,6) * scl(ijm1  ,k,l)
       enddo
       dscl(          1:OPRT_nstart-1,k,l) = 0.0_RP
       dscl(OPRT_nend+1:ADM_gall     ,k,l) = 0.0_RP
    enddo
    enddo
```

Equivalent code rewritten with the NICAM's dialect:

Listing 25: NICAM DSL code

```
  RANGE, CELL, 3D g1 =  GRID%cells | g{OPRT_nstart..OPRT_nend}
  foreach cell in g1
          dscl(cell) = REDUCE(+,N={0..6},
      clap(cell%g,cell%l,N) * scl(cell%neighbor(N))
      )
  endforeach

  foreach cell in GRID%cells | g{1..OPRT_nstart-1 , OPRT_nend+1 .. gall}
            dscl(cell) = 0.0_PRECISION
  endforeach
```

4.4 ASUCA

The following Listings show simplified extracts of the Hybrid Fortran implementation of ASUCA.

Listing 26: Data module containing the data objects used in the following code examples

```
    module example_data_module
        real, allocatable:: a(:,:,:)
        real, allocatable:: sum_a(:,:)

        @domainDependant{ &
            & domName(k,i,j), domSize(NZ,NX,NY), &
            & attribute(host), &
            & accPP(AT_KIJ), domPP(DOM_KIJ) &
        }
        a
        @end domainDependant

        @domainDependant{domName(i,j), domSize(NX,NY), attribute(host)}
        sum_a
        @end domainDependant

        subroutine allocate_example_data
            allocate( a(NZ, NX, NY) )
            allocate( sum_a(NX, NY) )
        end subroutine
    end module
```

Listing 27: Main program loop showing the implementation of the device data region

```
subroutine run_asuca
    use example_data_module, only: allocate_example_data, a, sum_a

    @domainDependant{attribute(autoDom, transferHere)}
    a, sum_a
    @end domainDependant

    call allocate_example_data
    ! ... further host allocation calls left out

    timestep_long: do
        ! ... setup of timestep and diagnose step left out
        call run_physics
        ! ... output and statistics left out
        call run_rungekutta_long ! will also call lateral_boundary_conditions_damping
        ! ... finalizing time step calls left out
        if ( ..exit_condition_reached.. ) then
          exit timestep_long
        end if
    end do timestep_long

    ! ... host cleanup calls left out
end subroutine
```

Listing 28: Coarse grained parallelization of physical processes - as applied to the root of their call graph

```
subroutine run_physics
    use example_data_module, only: a, sum_a

    real, intent(in), dimension(NZ, NX, NY) :: a
    real, intent(out), dimension(NX, NY) :: sum_a

    @domainDependant{attribute(autoDom, present)}
    a, sum_a
    @end domainDependant

    @parallelRegion{appliesTo(CPU), domName(i,j), domSize(NX,NY)}

    call sum_column(a, sum_a)
    ! .. more calls to deep graphs of subroutines

    @end parallelRegion
end subroutine
```

Listing 29: Fine grained parallelization of physical processes - as applied to all leafs of their call graph. Please note: This is a dummy routine with no physical correspondence, it is used here purely to show how Hybrid Fortran directives are applied.

```
subroutine sum_column(a, sum_a)
    real, intent(in), dimension(NZ) :: a
    real, intent(out) :: sum_a
    integer(4) :: k

    @domainDependant(attribute(autoDom, present), domName(i,j), domSize(NX,NY)}
    a, sum_a
    @end domainDependant

    @parallelRegion{appliesTo(GPU), domName(i,j), domSize(NX,NY)}

    sum_a = 0.0
    do k=1, NZ
        sum_a = sum_a + a(k)
    end do

    @end parallelRegion
end subroutine
```

Listing 30: Real world example from the dynamical core of ASUCA

```
subroutine lateral_boundary_conditions_damping(nx_u, nx_v)
  use prm, only: nx0, ny0, basex, basey
```

```
      use dvar, only: dens_ptb
      use ref, only: dens_ref_f_x, dens_ref_f_y, dens_ref_f
      use metrics, only: jd_uf, jd_vf, dxidx_uf, dyidy_vf
      use svar, only: vel_x_damp, vel_y_damp
      implicit none

      integer(4), intent(in) :: nx_u, ny_v

      integer(4):: k
      real :: adj
      real :: mom_x_v_damp
      real :: mom_y_v_damp
      real :: rdens_f
      real :: rnx0
      real :: rny0

      @domainDependant{attribute(autoDom, host)}
      adj0, adj1, adj2
      @end domainDependant

      @domainDependant{attribute(autoDom, present)}
      dens_ptb, dens_ref_f_x, dens_ref_f_y, dens_ref_f, jd_uf, jd_vf, dxidx_uf, dyidy_vf,
      vel_x_damp, vel_y_damp, vel_z_damp,
      mom_x_v_bnd, mom_y_v_bnd,
      rqa_v_bnd
      @end domainDependant

      rnx0 = 1._rp / real(nx0)
      rny0 = 1._rp / real(ny0)
      adj = adj0(ktb)                                    &
        & + adj1(ktb) * (tratio_bnd * hour2sec)          &
        & + adj2(ktb) * (tratio_bnd * hour2sec) ** 2

      @parallelRegion{ &
        & domName(i,j), domSize(NX_MN:NX_MX,NY_MN:NY_MX), startAt(1,1), endAt(nx_u,NY) &
      }
      do k = 1, NZ
        mom_x_v_damp =  mtratio_bnd * mom_x_v_bnd(k,i,j,1) &
          &           +   tratio_bnd * mom_x_v_bnd(k,i,j,2) &
          &           + adj / dxidx_uf(k,i,j)               &
          &           * real( nx0 - 2 * (basex + i) ) * rnx0
        rdens_f = 1._rp / ( dens_ref_f_x(k,i,j) &
          & + 0.5_rp * (dens_ptb(k,i,j) + dens_ptb(k,i+1,j)) )
        vel_x_damp(k,i,j) = mom_x_v_damp * jd_uf(k,i,j) * rdens_f
      end do
      @end parallelRegion

      @parallelRegion{ &
        & domName(i,j), domSize(NX_MN:NX_MX,NY_MN:NY_MX), startAt(1,1), endAt(NX,ny_v) &
      }
      do k = 1, NZ
        mom_y_v_damp =  mtratio_bnd * mom_y_v_bnd(k,i,j,1) &
          &           +   tratio_bnd * mom_y_v_bnd(k,i,j,2) &
          &           + adj / dyidy_vf(k,i,j)               &
          &           * real( ny0 - 2 * (basey + j) ) * rny0
        rdens_f = 1._rp / ( dens_ref_f_y(k,i,j) &
          & + 0.5_rp * (dens_ptb(k,i,j) + dens_ptb(k,i,j+1)) )
        vel_y_damp(k,i,j) = mom_y_v_damp * jd_vf(k,i,j) * rdens_f
      end do
      @end parallelRegion
    end subroutine
```

The following features are observable from these listings:

1. `@parallelRegion` directives are used to specify a parallel computation in a specified domain, here the horizontal `IJ`-domain. `domName` and `domSize` attributes specify the iterators and size of the domain as seen by the involved data objects. Domains are assumed to be 1-based, other starting points can be chosen by using a column (`:`) separator in the `domSize` attribute. `startAt` and `endAt` clauses can be used if only a partial domain is to be iterated in parallel. Both `domSize` and `startAt` / `endAt` clauses are sometimes needed when otherwise the compile-time defined privatization of data objects would be ambiguous (see also 2.2.4.3).

2. The `appliesTo` attributes in `@parallelRegion` directives can be used to apply such regions only

to a partial set of hardware architectures. For the other architectures the code within the region will be executed sequentially, e.g. `compute_all_columns` becomes a sequence of kernel calls. This implements compile-time defined parallelization granularity as described in section 2.2.4.2.

3. `@domainDependant` directives are used to give additional information about data objects:
   - The `autoDom` attribute directs Hybrid Fortran to pick up the declared dimensions in the Fortran data object specification, and merge them with the template dimensions given by `domSize` (see listing 29 as an example)
   - The `host` attribute makes sure that Hybrid Fortran will not treat a data object as device data in a particular subroutine. It is mainly used in code that is outside of the main call tree, such us module data specifications.
   - The `present` attribute is the opposite of `host`: It directs Hybrid Fortran to treat a data object as device present. See also section 2.2.4.4.
   - The `transferHere` attribute instructs Hybrid Fortran to implement a device allocation and transfer at this point in the code. See also section 2.2.4.4.
   - The `domName` clause gives the domain names (i.e. iterator names) associated with the sizes given by `domSize`. This gives important hints to Hybrid Fortran about how to treat a data object with respect to its privatization described in section 2.2.4.3.
   - The `accPP` and `domPP` clauses are used to specify the preprocessor macros used for reordering, in case there are different reorderings needed in an application. These macros can then be specified application-wide (in the file `storage_order.F90` that automatically gets included in all Hybrid Fortran source files). In ASUCA's case this is required since the synthetically privatized versions of `a` and `sum_a` in `sum_column` have a different reordering scheme than the user-written orderings of data objects like `vel_x_damp` in `lateral_boundary_conditions_damping`. Hybrid Fortran always adds the privatization before user specified domains, i.e. `a(k)` in `sum_column` gets converted to `a(AT(i,j,k))` using the default reordering macro `AT`. Other data objects therefore need a separate reordering macro, here `AT_KIJ`, so original code written in `KIJ` order can be retained. Employing this combination of transpiler and preprocessor macros implements the feature described in section 2.2.4.1.

   .

4. Other than in pure CUDA Fortran, multiple parallel regions per subroutine are supported (see listing 30 as an example). To achieve this, Hybrid Fortran employs a kernel routine synthesis.
5. Module data arrays are supported.
6. Outside of parallel regions, a mix of host and device data can be used. Local data objects will be passed into kernels correctly. See the data objects `adj`, `adj0`, `adj1` and `adj2` in listing 30 as an example.

5 Evaluation of Code Quality

We have taken two relevant kernels from each of the three models, and analyzed the achieved code reduction. An overview of the results is shown in Table 1 and Figure 16. The numbers demonstrate the impact on code length when porting code to the developed dialects.

	lines (LOC)		words		characters	
Model, kernel	before DSL	with DSL	before DSL	with DSL	before DSL	with DSL
ICON 1	13	7	238	174	317	258
ICON 2	53	24	163	83	2002	916
NICAM 1	7	4	40	27	76	86
NICAM 2	90	11	344	53	1487	363
DYNAMICO 1	7	4	96	73	137	150
DYNAMICO 2	13	5	30	20	402	218
total	183	55	911	430	4421	1991
percentage	30.05%		47.20%		45.04%	

Table 1: Model dialects impact on code size

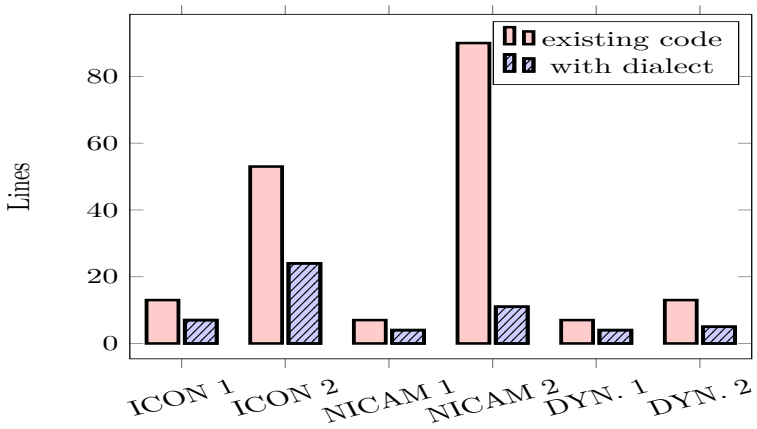


Figure 16: Dialects impact on LOC

In average, we cut down the LOC to less than one third (30%) of the original code. Better reductions are achieved in stencil codes (NICAM example No.2, reduced to 12.22% of the original LOC).
Influence on readability and maintainability: Reducing the important code metrics like **code duplication**, WTF/Minute – in code review, in some cases, boundary conditions could be removed thus reducing the cyclomatic complexity.
Code reduction reduces development costs. By applying COCOMO, we provide the estimated benefits in Table 2.

Software project	Codebase	Effort Applied	Dev. Time (months)	People require	dev. costs (M€)
Semi-detached	Fortran	2462	38.5	64	12.3
	DSL	1133	29.3	39	5.7
Organic	Fortran	1295	38.1	34	6.5
	DSL	625	28.9	22	3.1

Table 2: COCOMO cost estimates

6 Related Work

Many research efforts were directed towards solving the problem of performance portability. Approaches range from using domain libraries, to compiler directives and annotations, to general-purpose language embedded DSL constructs like C++ template programming, to standalone DSLs that replace general-purpose languages, and finally to language extensions.

Library approaches provide high-level functions that models can use to perform some computation with high performance. Bianco et al. [BV12] provide a library for stencil computations. They use generic programming capabilities of C++ language to provide a solution for regular (structured) grid based applications. The active library OP2 [MGR+12] provides a framework to generate optimized code for multiple back-ends. It provides an API to define unstructured meshes and connectivity maps with C/C++ and Fortran. OPS [RMG+14] is another active library that provides C++ domain specific abstractions for multi-block structured grid based applications. It uses source-to-source translation to generate platform specific code that makes use of optimized back-end libraries for different configurations. Tangram [CDRH15] provides optimized code for CPU and GPU platforms through data-structure-based libraries. Besides, it allows programmers to explicitly specify optimizations through using rewriting-rules within code. Shimokawabe et al. [SAO14] [SAO16] provide a C++ based framework that is added to user code to provide performance portability for regular structured grids. Programmers provide stencil computations as C++ functions that update grid points, and the framework translates them to CPU/GPU optimized codes. It also produces any needed CUDA and MPI codes.

Source code preprocessing based solutions use compiler directives to annotate parts of code that is preprocessed before being submitted to backend compiler. Those solutions use a special front-end compiler/preprocessor to process annotated code. HMPP [DBB07] uses directives along with a runtime to generate accelerator high performance code. Parallelization and optimization decisions are provided by source code also in the work

of Christen et al. [CSB11], where they provide a DSL for code generation and auto tuning of stencil codes for manycore and multicore processor based systems. Mint [UCB11] uses annotations to translate stencil computations from C to optimized CUDA C. Annotations within the source code drive the optimization process. In Gung Ho [FGH+13], scientific code is separated into high-level operations acting on whole fields (the algorithm layer) and low-level operations that explicitly compute with the data (kernels). In between sits a layer of autogenerated code, driven partly by directives, that handles looping over data and attempts to optimize performance for different architectures and parallelization strategies. In CLAW [Cla] project, optimization and implementation details like loop optimization and domain decomposition are explicitly specified with annotations added within source code.

General-purpose language embedded constructs, like templates in C++ or regular expressions are used in some solutions. Domain code takes benefit of higher level abstractions built with such constructs. Lower level implementations provide performance for a specific platform. In the C++ library Kokkos [ETS14] C++ constructs are used to support different memory layouts for manycore architectures. The C++ stencil library Stella [FOL+14] was developed for structured grids in climate models. It uses domain concepts through a DSL to code kernels logic using C++ constructs. GridTools [Gri] generalize Stella and add support for other grid types. In addition to C++, Gridtools support the translation of regular stencil code in Python into C++ Gridtools code. Berényi's work [Ber15] extends C++ language with an embedded DSL for AST manipulation. Constructs of this embedded DSL provide parallelization. YASK [You25] is a C++ framework that provides constructs to specify stencils and kernels. It provides a specialized source-to-source translator to convert scalar C++ stencil code into optimized C++ code. Optimization includes SIMD optimization in addition to many other optimizations that harness the power of Xeon-Phi processor.

Some source-to-source translation solutions specify language constructs in a domain-specific language that provides a new syntax which replaces general-purpose languages. Compilation of such DSLs code generates code for different architectures. These DSLs need to support further language features like expressions, operators, and may cover program flow and control. Acceptance of such solutions is crucial, e.g., declarative and functional programming differs significantly from usually used coding styles and thus, is not easily accepted from the domain scientists. Example DSLs that are tight to the scientific domain are Atmol [AvE01] and Liszt [DJP+11]. Such solutions require modification to existing compilers or creation of a new language compiler and force users to rewrite kernels completely with the new syntax. Additional work is needed to integrate the generated code with other parts of the application code.

In contrast to standalone DSLs, Language extension depends on adding new types and constructs to a general-purpose language to support domain concepts and needs modifying a compiler accordingly to generate code. In Physis [MSNM11], code is written in C++ but extended with some domain concepts. It provides a source-to-source translator built on top of ROSE [Qui00]. Code is translated by this source-to-source translator into the target platform code; CUDA code for GPU platforms and C for CPUs. It also uses a runtime component for each platform to achieve high performance. It generates MPI code for distributed compute resources. PyOP2 [RMM+12] provides a parallelization solution for numerical kernels over unstructured meshes through an embedded DSL. On problem-specific parameter unavailability, PyOP2 uses just-in-time kernel compilation and parallelization scheduling. Torres et al. [TLKL13] extend Fortran language to support different index permutations in multidimensional arrays in ICON model. They also use ROSE to do source-to-source translation. However, the solution was heavy-weight and the Fortran parser required many adjustments to run on the complex model code.

We build on the concept of a language extension DSL, with a more compact and dynamic configurable compilation tool. The concept applies to various general-purpose languages in general.

7 Summary and Conclusions

In this section we conclude with a summary about the work described in this document. A summary about the development of the dialects for the three icosahedral models is discussed first. Then, we discuss the opportunities to integrate the suggested solution with ASUCA as a backend for source-to-source translation tools of user code using the developed dialects.

7.1 Dialects

Scientific applications like climate and atmospheric models are highly demanding for performance. However, the software development process using the general-purpose languages (e.g. Fortran) and their compilers still carries some weaknesses relating to the source code optimization. The source code manual optimization harms

its readability and hence maintainability. The performance portability of the code is then also comprised. Besides, the scientists who develop the models would need to learn further lower-level details related to the target architecture on which the model will be run in order to be able to manually optimize the source code. All those factors led us (under the project AIMES) to think of alternative development method for the development of icosahedral models. What we provide is an alternative to the general-purpose languages and the convensional development tools that are used in climate/atmosheric modeling.
Our approach provides the scientists a language that is the same language which they use to write their model, with some additional extensions. The scientists do not need to care about the manual optimization of the source code. The extensions allow symantically to generate an optimized code for a target architecture.
We provide extensions to support three different icosahedral models: DYNAMICO, ICON, and NICAM. In this document we discuss extending each of the three models with a dialect. However, we kept in mind –whenever possible– to reuse the extensions among the models such that we eventually define a common DSL to support icosahedral modeling. In the document we discuss the evolved dialects, and some code examples from the three models before and after using the extensions. Code examples from ASUCA are also shown in the same section. To show the impact of using the dialects, we discussed the code quality and financial impact. We showed that the dialects reduced LOC count to less than one third the Fortran code. We showed also that the use of the dialects reduces the model development costs to less than half the development costs using Fortran.

7.2 ASUCA and Hybrid Fortran

Due to rather conservative stance of the application owners behind ASUCA, expressed in requirement 2 in section 2.2.4, as well as the large code size shown in figure 13, we come to the conclusion that at this point in time the ASUCA model is not a good candidate for a rewrite in a new DSL. However, it serves as an interesting example for the requirements towards performance portability in similar structured grid models such as COSMO and WRF. The following features would make a DSL attractive for a re-implementation of such structured grid models:

1. Support for multidimensional arrays in the backend implementation (i.e. mapping cells to array entries where applicable).
2. Support for a flexible storage order (see also section 2.2.4.1).
3. Support for the partial application of parallel ranges, depending on the target architecture (see also section 2.2.4.2).
4. Support for compile-time defined privatization, depending on the partial application of ranges (see also section 2.2.4.3).
5. Support for device data regions (see also section 2.2.4.4).

This can be achieved in one of the following ways:

1. Support for all of the above in the DSL's frontend, then generate Hybrid Fortran code in the backend using a source-to-source translation from the intermediate language to Hybrid Fortran.
2. Support for all of the above in the DSL's frontend and reimplement the functionality of Hybrid Fortran both for the intermediate language as well as the backend implementation.
3. Separate the point-wise DSL frontend from the `RANGE` implementation. That is, allow other tools like Hybrid Fortran to bring their own implementation of `RANGE` through a standardized coding interface for the iterators. This approach would pass off the responsibility for the problems described in sections 2.2.4.2 and 2.2.4.4 to the implementation provider of `RANGE`, or the analogous construct in the provider's framework (in Hybrid Fortran's case this would be `@parallelRegion`). The higher order DSL would then just be responsible to implement the point-wise code within the region specification. Hybrid Fortran would in this approach rather be a wrapper than an additional backend to the DSL. Such an approach could provide maximum flexibility for tool builders to mix and match technologies that work best for a particular project to achieve performance portability.

Glossary

DSL Domain-Specific Language.

HEVI Horizontal Explicit Vertical Implicit computational solutions for PDEs.

ICOMEX ICOsahedral-grid Models for EXascale earth system simulations, a previous project funded by the G8 before AIMES to develop methods for fine resolution climate models based on icosahedral grids.

Meta-DSL The configuration describing the grammer of the DSL and how it is handled .

Model's Dialect The extensions added to the modeling language to support a specific model.

Transpiler Source-to-source translation compiler.

Acknowledgement

This work was supported by the German Research Foundation (DFG) through the Priority Programme 1648 „Software for Exascale Computing“ (SPPEXA).

References

[AvE01] Robert A van Engelen. Atmol: A domain-specific language for atmospheric modeling. *CIT. Journal of computing and information technology*, 9(4):289–303, 2001.

[Ber15] Dániel Berényi. C++ EDSL for parallel code generation. In *Grid, Cloud & High Performance Computing in Science (ROLCG), 2015 Conference*, pages 1–5. IEEE, 2015.

[BV12] Mauro Bianco and Ugo Varetto. A generic library for stencil computations. *arXiv preprint arXiv:1207.1746*, 2012.

[CDRH15] Li-Wen Chang, Abdul Dakkak, Christopher I Rodrigues, and Wenmei Hwu. Tangram: a high-level language for performance portable code synthesis. *In Programmability Issues for Heterogeneous Multicores*, 2015.

[Cla] CSCS Claw. `https://github.com/C2SM-RCM`. Accessed: 2016-11-22.

[COG+13] Ben Cumming, Carlos Osuna, Tobias Gysi, Mauro Bianco, Xavier Lapillonne, Oliver Fuhrer, and Thomas C. Schulthess. A review of the challenges and results of refactoring the community climate code cosmo for hybrid cray hpc systems. In *CUG2013 Proceedings*, 2013.

[CSB11] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687. IEEE, 2011.

[DBB07] Romain Dolbeau, Stéphane Bihan, and François Bodin. Hmpp: A hybrid multi-core parallel programming environment. In *Workshop on general purpose processing on graphics processing units (GPGPU 2007)*, volume 28, 2007.

[DDT+15] Thomas Dubos, Sarvesh Dubey, Marine Tort, Rashmi Mittal, Yann Meurdesoif, and Frédéric Hourdin. Dynamico, an icosahedral hydrostatic dynamical core designed for consistency and versatility. *Geoscientific Model Development Discussions*, 8(2):1749–1800, 2015.

[DHK+00] Craig C Douglas, Jonathan Hu, Markus Kowarschik, Ulrich Rüde, and Christian Weiß. Cache optimization for structured and unstructured grid multigrid. *Electronic Transactions on Numerical Analysis*, 10:21–40, 2000.

[DJP+11] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, et al. Liszt: a domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 9. ACM, 2011.

[DNW+09] Hikmet Dursun, Ken-ichi Nomura, Weiqiang Wang, Manaschai Kunaseth, Liu Peng, Richard Seymour, Rajiv K Kalia, Aiichiro Nakano, and Priya Vashishta. In-core optimization of high-order stencil computations. In *PDPTA*, pages 533–538, 2009.

[DT14] Thomas Dubos and Marine Tort. Equations of atmospheric motion in non-eulerian vertical coordinates: Vector-invariant form and quasi-hamiltonian formulation. *Monthly Weather Review*, 142(10):3860–3880, 2014.

[ETS14] H Carter Edwards, Christian R Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[FGH+13] R Ford, MJ Glover, DA Ham, CM Maynard, SM Pickles, G Riley, and N Wood. Gung ho: A code design for weather and climate prediction on exascale machines. In *Proceedings of the Exascale Applications and Software Conference*, 2013.

[FOL+14] Oliver Fuhrer, Carlos Osuna, Xavier Lapillonne, Tobias Gysi, Ben Cumming, Mauro Bianco, Andrea Arteaga, and Thomas Christoph Schulthess. Towards a performance portable, architecture agnostic implementation strategy for weather and climate models. *Supercomputing frontiers and innovations*, 1(1):45–62, 2014.

[Gas13] Almut Gassmann. A global hexagonal c-grid non-hydrostatic dynamical core (icon-iap) designed for energetic consistency. *Quarterly Journal of the Royal Meteorological Society*, 139(670):152–175, 2013.

[GH08] Almut Gassmann and Hans-Joachim Herzog. Towards a consistent numerical compressible nonhydrostatic model using generalized hamiltonian tools. *Quarterly Journal of the Royal Meteorological Society*, 134(635):1597–1613, 2008.

[GKC13] Francis X Giraldo, James F Kelly, and Emil M Constantinescu. Implicit-explicit formulations of a three-dimensional nonhydrostatic unified model of the atmosphere (numa). *SIAM Journal on Scientific Computing*, 35(5):B1162–B1194, 2013.

[Gri] CSCS GridTools. `http://www2.cosmo-model.org/content/consortium/developers/2016_01/Gridtools_python.pdf`. Accessed: 2016-11-22.

[Har07] Mark Harris. Optimizing cuda. *SC07: High Performance Computing With CUDA*, 2007.

[IMKK10] Junichi Ishida, Chiashi Muroi, Kohei Kawano, and Yuji Kitamura. Development of a new nonhydrostatic model asuca at jma. *CAS/JSC WGNE Research Activities in Atmospheric and Oceanic Modelling*, 40:0511–0512, 2010.

[Kwi01] Jan Kwiatkowski. Evaluation of parallel programs by measurement of its granularity. In *International Conference on Parallel Processing and Applied Mathematics*, pages 145–153. Springer, 2001.

[MGR+12] GR Mudalige, MB Giles, I Reguly, C Bertolli, and PH J Kelly. Op2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In *Innovative Parallel Computing (InPar), 2012*, pages 1–12. IEEE, 2012.

[MHH14] Jarno Mielikainen, Bormin Huang, and Allen Huang. Using intel xeon phi to accelerate the wrf temf planetary boundary layer scheme, 2014.

[MSNM11] Naoya Maruyama, Kento Sato, Tatsuo Nomura, and Satoshi Matsuoka. Physis: an implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12. IEEE, 2011.

[Qui00] Dan Quinlan. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000.

[RMG+14] István Z Reguly, Gihan R Mudalige, Michael B Giles, Dan Curran, and Simon McIntosh-Smith. The ops domain specific abstraction for multi-block structured grid computations. In *Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC), 2014 Fourth International Workshop on*, pages 58–67. IEEE, 2014.

[RMM+12] Florian Rathgeber, Graham R Markall, Lawrence Mitchell, Nicolas Loriant, David A Ham, Carlo Bertolli, and Paul HJ Kelly. Pyop2: A high-level framework for performance-portable simulations on unstructured meshes. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1116–1123. IEEE, 2012.

[SAO14] Takashi Shimokawabe, Takayuki Aoki, and Naoyuki Onodera. High-productivity framework on gpu-rich supercomputers for operational weather prediction code asuca. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 251–261. IEEE Press, 2014.

[SAO16] Takashi Shimokawabe, Takayuki Aoki, and Naoyuki Onodera. High-productivity framework for large-scale gpu/cpu stencil applications. *Procedia Computer Science*, 80:1646–1657, 2016.

[SIK+] M Sakamoto, J Ishida, K Kawano, K Matsubayashi, K Aranami, T Hara, H Kusabiraki, C Muroi, and Y Kitamura. Development of yin-yang grid global model using a new dynamical core asuca.

[SMTM08] M Satoh, T Matsuno, H Tomita, and H Miura. Nonhydrostatic icosahedral atmospheric model (NICAM) for global cloud resolving simulations. *JOURNAL OF ...*, 2008.

[STY+14a] Masaki Satoh, Hirofumi Tomita, Hisashi Yashiro, Hiroaki Miura, Chihiro Kodama, Tatsuya Seiki, Akira T Noda, Yohei Yamada, Daisuke Goto, Masahiro Sawada, et al. The non-hydrostatic icosahedral atmospheric model: Description and development. *Progress in Earth and Planetary Science*, 1(1):1, 2014.

[STY+14b] Masaki Satoh, Hirofumi Tomita, Hisashi Yashiro, Hiroaki Miura, Chihiro Kodama, Tatsuya Seiki, Akira T Noda, Yohei Yamada, Daisuke Goto, Masahiro Sawada, Takemasa Miyoshi, Yosuke NIWA, Masayuki HARA, Tomoki Ohno, Shin-ichi Iga, Takashi Arakawa, Takahiro Inoue, and Hiroyasu Kubokawa. The Non-hydrostatic Icosahedral Atmospheric Model: description and development. *Progress in Earth and Planetary Science*, 1(1):2293, October 2014.

[TGS08] H Tomita, K Goto, and M Satoh. A new approach to atmospheric general circulation model: Global cloud resolving model NICAM and its computational performance. *SIAM Journal on Scientific Computing*, 30(6):2755–2776, 2008.

[TLKL13] Raul Torres, Leonidas Linardakis, TL Julian Kunkel, and Thomas Ludwig. Icon dsl: A domain-specific language for climate modeling. In *International Conference for High Performance Computing, Networking, Storage and Analysis, Denver, Colo.[Available at h ttp://sc13. supercomputing. org/sites/default/files/WorkshopsArchive/track139. html.]*, 2013.

[TS04] Hirofumi Tomita and Masaki Satoh. A new dynamical framework of nonhydrostatic global model using the icosahedral grid. *Fluid Dynamics Research*, 34(6):357–400, 2004.

[UCB11] Didem Unat, Xing Cai, and Scott B Baden. Mint: realizing cuda performance in 3d stencil methods with annotated c. In *Proceedings of the international conference on Supercomputing*, pages 214–224. ACM, 2011.

[WS02] Louis J Wicker and William C Skamarock. Time-splitting methods for elastic models using forward time schemes. *Monthly weather review*, 130(8):2088–2097, 2002.

[WSTaM12] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. Openacc—first experiences with real-world applications. In *European Conference on Parallel Processing*, pages 859–870. Springer, 2012.

[You25] Chuck Yount. Recipe: Building and Running YASK (Yet Another Stencil Kernel) on Intel® Processors. `https://software.intel.com/en-us/articles/recipe-building-and-running-yask-yet-another-stencil-kernel-on-intel-processors`, 2016 (Accessed: 2016-11-25).

[ZRRB15] Günther Zängl, Daniel Reinert, Pilar Rípodas, and Michael Baldauf. The icon (icosahedral non-hydrostatic) modelling framework of dwd and mpi-m: Description of the non-hydrostatic dynamical core. *Quarterly Journal of the Royal Meteorological Society*, 141(687):563–579, 2015.