D3.1 Mini-IGCMs Version 1

Nabeeh Jumah Julian Kunkel

Workpackage: WP3 Evaluation
Responsible institution: Yashiro, Maruyama
Contributing institutions: RIKEN, IPSL
Date of submission: February 2017

Contents

Abstract

In this report we present a set of icosahedral kernels which were extracted from three icosahedral models; ICON, DYNAMICO, and NICAM. The models use different icosahedral grids and different ways to access the data of the fields. The kernels are shown in two forms: the original Fortran code as taken from the models, and a DSL-formulated version that uses a set of DSL language extensions. The set of the extensions were developed as part of another work-package (WP1) in the same project. The kernels cover a set of cases that represent important features for the coding productivity and performance. The DSL eases the coding of the models by reducing the coding problem to the scientific part in terms of scientific concepts. So, the connectivity details of the icosahedral grids and the performance aspects like parallelization or memory layout are hidden in the DSL versions of the kernels. The kernels show operations on two- and three-dimensional grids.

1 Introduction

This report describes our effort within the work-package WP3 during the first phase of the project to formulate a set of kernels with higher-level code.
We extracted a set of kernels from the three icosahedral models; ICON, DYNAMICO, and NICAM, and as part of the work-package WP1 we developed a set of dialects to code the kernels with higher-level language. The dialects were used as a starting point for the development of a domain-specific language.
The code that we show in this report was formulated based on the language extensions that we suggested in the work-package WP1.
The kernel versions that are formulated with the DSL show the abstraction of the indices, and the access to the fields using the DSL iterator. DSL extensions are used to simplify the indirect access, to reference neighbors and connected components, e.g. edges of a cell. The connectivity is abstracted through grid concepts.
Different cases for two- and three-dimensional grids are covered by the chosen kernels.

1.1 Relation to the Project

This report describes the testbed suite of icosahedral climate model; source-code and documentation including the description of test-cases. Version 1 includes runnable instances of the simple kernels.
The following text is the description of the project proposal for this task and deliverable:

- *Survey and selection of test cases*
  In this task, we estimate problem sizes of simulations expected on exa-scale computing systems. Parameters such as number of horizontal/vertical grids, duration time, number/size/interval of output variables are selected for two types of simulation: numerical weather prediction and climate simu- lation. For the evaluation of data compression in file I/O, we should carefully choose the value and spatial distribution of the data.

- *Extract simple kernels from climate models*
  Simple kernels are required for testing the benefit of the DSL developed in WP1. At first, we extract several typical and performance critical kernels from each models that are representative of this model. A diverse set of kernels is chosen that e.g. cover direct/indirect access, 2D/3D stencil calculation and conditional branches. Those kernels are packaged that they can be compiled and run. We also prepare simple I/O kernels for testing, if needed.

2 Language Extensions

The project includes the development of a set of language extensions (as part of the work-package WP1) and developing a set of mini applications based on the codes of the three icosahedral models (as part of the work-package WP3). Therefore, the kernels that we provide in this document are extracted from the codes of the models, and rewritten with the language extensions.
The set of the language extensions is developed within work-package WP1 to serve the higher-level coding of the models. The domain scientists recommended some coding problems and kernels to consider for the sake of the development of the language extensions.
Language extensions were suggested and kernels were rewritten with the suggested extensions and discussed with the domain scientists. The suggestions and discussions were done iteratively until we reached into an acceptable form that serves the semantical needs while fitting the scientists.

3 Computation and I/O

The focus of the codes in the scope of this document is on the computational operations. However, we provide also a set of basic I/O kernels to handle the initialization of the modeling fields and to write the necessary fields data at specific time steps.
The I/O operations are implemented with NetCDF in the mini application that will be described in a later document (Deliverable D3.3).

4 Extracted Kernels

In this section, we show a set of extracted kernels from each one of the three models. The kernels are written both in the original Fortran code and the DSL version.

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 1: ICON Fortran code #1

```
#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 DSL:

Listing 2: ICON DSL code

```
foreach edge in grid
        ! 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 the DSL 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 DSL hides the memory layout and connectivity information.

In the following 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 3: ICON Fortran code #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 DSL:

Listing 4: 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 DSL 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 code.

Listing 5: ICON Fortran code #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 DSL:

Listing 6: 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 DSL extensions, (cell%above) and (cell%below) are used to access values vertically neighboring the cell.

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

Listing 7: ICON Fortran code #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 DSL:

Listing 8: ICON DSL code

```
foreach cell in grid2D
  z_thermal_exp(cell) = 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 the DSL extensions.

In the following code, 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 9: ICON Fortran code #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 DSL:

Listing 10: 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 code shows a more complex case 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 11: ICON Fortran code #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 DSL:

Listing 12: 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 13: 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 DSL:

Listing 14: DYNAMICO DSL code

```
foreach cell in grid
 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%neighbor) to refer to the neighbors of a cell simplifies the code of the stencil operation and eliminates the duplicate code over each neighbor.

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 neighbor cells within the stencil operation.

Listing 15: NICAM Fortran code #1

```
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 DSL:

Listing 16: NICAM DSL code

```
foreach cell in grid | g{OPRT_nstart..OPRT_nend}
  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 | 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 DSL to define the RANGE in the first line helps iterating a subset of the grid cells. Within the iterator, the use of (cell%neighbor) removes the complexity of the index calculations which are necessary to reference neighbors. Also the REDUCE extension simplifies the stencil operation over the neighbors.

The following code 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 17: NICAM Fortran code #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 DSL:

Listing 18: NICAM DSL code

```
  foreach cell in grid | 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)))
  end foreach

  foreach cell in grid | g{1..OPRT_nstart-1 , OPRT_nend+1 .. gall}
            scl(cell) = 0.0_PRECISION
  end foreach
```

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 code 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 19: NICAM Fortran code #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 DSL:

Listing 20: NICAM DSL code

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

The REDUCE in the new code allows to represent the computation that involves 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 code.

Listing 21: NICAM Fortran code #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 DSL:

Listing 22: NICAM DSL code

```
  foreach cell in grid | g{OPRT_nstart..OPRT_nend}
          dscl(cell) = REDUCE(+,N={0..6},
      clap(cell%g,cell%l,N) * scl(cell%neighbor(N)))
  end foreach

  foreach cell in grid | g{1..OPRT_nstart-1 , OPRT_nend+1 .. gall}
            dscl(cell) = 0.0_PRECISION
  end foreach
```

5 Summary and Conclusions

We discussed a set of kernels that were extracted from three icosahedral models. The kernels were rewritten with the DSL language extensions. Many coding issues were covered with the choice of codes that we selected. Both two- and three-dimensional grids were considered in the choice of the kernels.
Different access ways from the different models were reformulated in a unified set of DSL constructs. Thus, we see the same iterator used, and the same concepts for memory access with abstract indices used in the DSL version of the kernels. The DSL constructs replace the original general-purpose language iterators and indices.

Acknowledgement

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