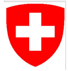


Parallel Model

- Let’s now see in detail how a dusk stencil is meant to be executed.
- The execution model (we also call it *parallel model*) presented here works as a contract with the (DSL) user.
- For each stencil, executing its generated code (which might be the result of various transformations and optimizations) *must* produce the same **effects** on output fields as the parallel model’s execution would when given the same inputs.
- The user thus doesn’t need to worry about what goes under the hood, as dusk&dawn promise that the generated code will behave equivalently to the parallel model’s execution.

Parallel Model

```
...
with levels_upward:
    ...
with levels_upward:
    ...
with levels_downward:
    ...
```

Looking at the code in a top-down fashion, the first “nodes” we encounter are *vertical domain regions*.

These constitute blocks of codes which must be executed **sequentially** in the order in which they appear.

Parallel Model

```
...
with levels_upward:
    STATEMENT1
    STATEMENT2
    STATEMENT3
    ...
```

Within each region:

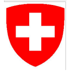
- Iterate **sequentially** through the k-levels (upward or downward, depending on the `with levels_*` statement).
  - Within each k iteration execute the statements (direct children) of the region **sequentially** in the order in which they appear. The execution of a statement can start only when the preceding ones have completed.

Parallel Model

```
...
with levels_upward:
    f_a = f_b
    with sparse[Cell > Edge]:
        f_sparse = f_edge * f_cell
    if condition:
        f_c = 5.0
    ...
```

Each statement (direct child) of the region is executed on each location of the *horizontal* domain (location type depends on the statement) in **any order**.

This makes it an embarrassingly parallel formulation, allowing dawn to produce code that runs on several threads.



Parallel Model

```
...
with levels_upward:
    ...
    if f_c > 0.0:
        f_c = 0.0
    else:
        f_c = - f_c
    ...
```

This also means that statements containing substatements, such as if-then-else constructs and sparse loops, are to be considered atomic: they must be evaluated as a whole for each location.

Parallel Model

```
@stencil
def copy(
    inF: Field[Edge, K],
    inoutF: Field[Edge, K],
    outF: Field[Edge, K]
):

    tempF: Field[Edge, K]

    with levels_upward as k:
        tempF = inF
        outF = tempF
        inoutF = inoutF + tempF
```

API fields (part of the contract with the user):

- Input: must not be changed
- Output: doesn't matter what contained before, at the end of the stencil execution it must contain the correctly computed value
- Input-Output: same as output, but what contains at the beginning matters

Temporary fields (**not** part of the contract with the user):

Compiler has full leeway in what to do with them, e.g. keep or inline, …

The user should think of them as local "variables" with the scope of the stencil.

MeteoSwiss

Parallel Model

```
@stencil
def reduction(
  f: Field[Edge, K]
):

  with levels_upward:
    f = sum_over(
          Edge > Cell > Edge,
          f[Edge > Cell > Edge]
        )
```

In the right hand sides of assignments, *value (copy) semantics* apply to fields being read.
The value is the **field as it was before the statement started being executed**.

Important point, will be clear why you need this later on...

Parallel Model: Execution Safety

We will now try to relax some constraints of the parallel model and highlight some criticalities that arise. This is to show what the compiler can/cannot do in order to optimize code.

Parallel Model: Execution Safety

```
@stencil
def vertical(
  f: Field[Edge, K],
  g: Field[Edge, K]
):
```

Remove sequentiality of **k-loop** iterations, allow **any order**. Opportunity to parallelize.

```
with levels_upward as k:
  f = f + f[k-1]
```

```
with levels_upward as k:
  g = g + 1.0
  f = g[k-1]
```

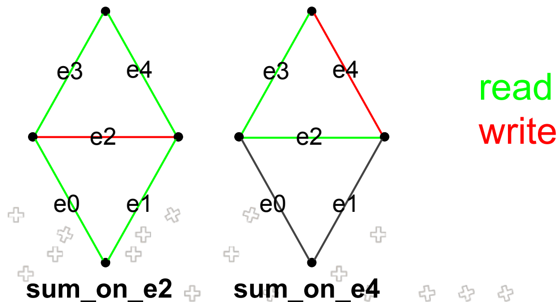
In general not possible when there are *vertical data dependencies* between statements (or of a statement with itself). But there are exceptions… (last slide)

```
with levels_upward:
  f = g + 2.0
```

Any other case is ok.

Parallel Model: Execution Safety

```
@stencil
def reduction(
  f: Field[Edge, K]
):
  with levels_upward:
    f = sum_over(
          Edge > Cell > Edge,
          f[Edge > Cell > Edge])
```



Reference semantics on rhs instead of copy semantics. Accessing the actual field, as it is now.

This is **dangerous** because, depending on the order in which the statement is executed over the locations, the results change.

Think about swapping the order of sum_on_e2 and sum_on_e4.

An execution with multiple threads has exactly this kind of problem, which is called a **race condition**.

Copy semantics are the only safe option.

MeteoSwiss

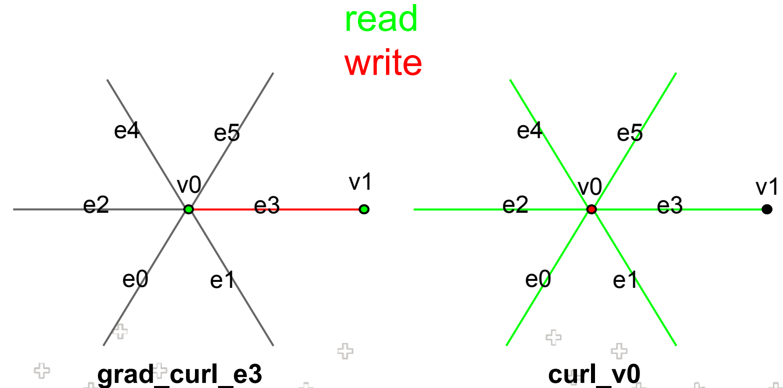
Parallel Model: Execution Safety

```
@stencil
def reduction(
  f: Field[Edge, K],
  grad_curl_f: Field[Edge, K]
):

  curl_f: Field[Edge, K]

  with levels_upward:
    curl_f =
       sum_over(Vertex > Edge,
                f * geofac_curl)
    grad_curl_f =
       sum_over(Edge > Vertex,
                curl_f,
                weights=[-1.0, 1])
```

Remove **sequentiality of statements.**

For example, a threaded execution of statements within a k iteration doesn't guarantee the sequentiality of the statements inside.

One thread executing these statements for each location.



Race condition!

MeteoSwiss

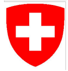
Vertical Solvers

$$\frac{\partial T}{\partial t} = \kappa \nabla^2 T$$

To introduce vertical solvers, let's start from the *heat equation* that we have to solve.

$$\frac{\partial T}{\partial t} = \kappa \frac{\partial^2 T}{\partial y^2}$$

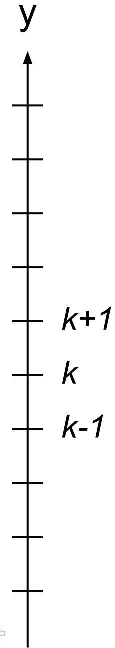
The focus here is to solve it along vertical columns of our domain, therefore we can directly look at the 1D heat equation.



Vertical Solvers

Usually in NWP, along the vertical, a fully **implicit discretization** scheme (backward Euler for time and second-order central finite difference for space) is employed (always numerically stable):

$$\frac{T_k^{n+1} - T_k^n}{\Delta t} = \kappa \frac{T_{k+1}^{n+1} - 2T_k^{n+1} + T_{k-1}^{n+1}}{(\Delta y)^2}$$



n: time point, *k*: vertical point

Vertical Solvers

Rearranging the recurrence equation:

$$T_k^{n+1} + \Delta t \kappa \frac{-T_{k+1}^{n+1} + 2T_k^{n+1} - T_{k-1}^{n+1}}{(\Delta y)^2} = T_k^n$$

In matrix form:

$$\left(\mathbb{I} + \frac{\Delta t \kappa}{(\Delta y)^2} \begin{bmatrix} 2 & -1 & & & 0 \\ -1 & 2 & -1 & & \\ & -1 & 2 & \ddots & \\ & & \ddots & \ddots & -1 \\ 0 & & & -1 & 2 \end{bmatrix}\right) \begin{bmatrix} T_0^{n+1} \\ T_1^{n+1} \\ T_2^{n+1} \\ \vdots \end{bmatrix} = \begin{bmatrix} T_0^n \\ T_1^n \\ T_2^n \\ \vdots \end{bmatrix}$$

Unknown

Vertical Solvers

$$\begin{bmatrix} b_0 & c_0 & & & 0 \\ a_1 & b_1 & c_1 & & \\ & a_2 & b_2 & \ddots & \\ & & \ddots & \ddots & c_{i-2} \\ 0 & & & a_{i-1} & b_{i-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{i-1} \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{i-1} \end{bmatrix}$$

A system of linear equations expressed through a tridiagonal matrix is solvable in linear time.

Vertical Solvers

```
@stencil
def TDMA(
 a: Field[Edge, K], b: Field[Edge, K], c: Field[Edge, K],
 d: Field[Edge, K], x: Field[Edge, K] ):

 g: Field[Edge, K]

 with levels_upward[0:0] as k:
   c = c / b
   d = d / b
 with levels_upward[1:] as k:
   g = 1.0 / (b - a * c[k-1])
   c = c * g
   d = (d - a * d[k-1]) * g
 with levels_downward[0:-1] as k:
   d -= c * d[k+1]
 with levels_upward:
   x = d
```

Forward sweep

Backward sweep

Thomas' algorithm to solve tridiagonal system of equations.

Applied column-wise over the whole domain.

Vertical Solvers

```
...
with levels_upward[1:] as k:
  f = f + f[k-1]
with levels_downward[0:-1] as k:
  g = g + g[k+1]
```

Solver-like access: vertically offset access to value written by previous iteration of k-loop.

If there is at least 1 solver-like access: **cannot parallelize k-loop**.

```
with levels_upward[0:-1] as k:
  f = f + f[k+1]
with levels_downward[1:] as k:
  g = g + g[k-1]
```

Stencil-like access: vertically offset access to value present before the k-loop.

If only stencil-like accesses: **can parallelize k-loop**.

MeteoSwiss

Q&A

Questions?

Refresher

We have learned how to express basic stencil operators in dusk

```
@stencil
def grad_n(f_n: Field[Edge], dualL: Field[Edge], f: Field[Cell]):
   with levels_downward:
       f_n = sum_over(Edge > Cell, f, weights=[1,-1]) / dualL


@stencil
def divergence(vn: Field[Edge], L: Field[Edge], A: Field[Cell], edge_orientation:
Field[Cell > Edge], div: Field[Cell]):
  with levels_downward:
      div = sum_over(Cell > Edge, vn * L * edge_orientation) / A
```

MeteoSwiss

Combining operators

An typical PDE operator needs to be expressed as a combination of various basic stencil operators.

E.g. the FVM vector laplacian:

$$\nabla^2 \mathbf{v} = \nabla(\nabla \cdot \mathbf{v}) - \nabla \times (\nabla \times \mathbf{v})$$

In its discretized form:

$$\langle \nabla^2 \mathbf{v} \rangle_\mathbf{e} = \langle \nabla \langle \nabla \cdot \mathbf{v} \rangle_\mathbf{c} \rangle_\mathbf{e} - \langle \nabla \times \langle \mathbf{v} \rangle_\mathbf{v} \rangle_\mathbf{e}$$