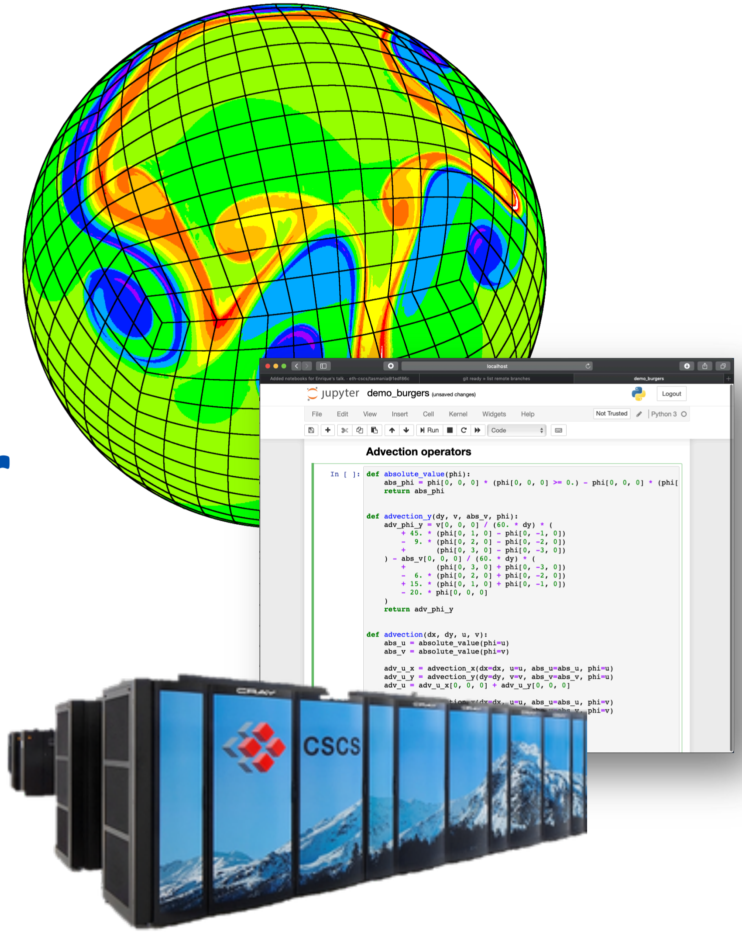
High Performance Computing for Weather and Climate (HPC4WC)



Content: High-Level Programming

Lecturer: Stefano Ubbiali

Block course 701-1270-00L

Summer 2020

Learning Goals

- Learning what a domain-specific language (DSL) is.
- Understanding how a DSL helps in writing hardware-agnostic, maintainable code without sacrificing performance.
- Be able to apply a DSL to a stencil program from a weather and climate model.

Typical Workflow

solver.py

```
import numpy as np
```

Fast prototyping in Python (or MATLAB)

solver.cpp

```
#include <vector>
```

Naïve implementation in a compiled language (e.g. C++)

solver_omp.cpp

```
#pragma omp parallel
```

Multi-threaded version using OpenMP

solver_mpi.cpp

```
MPI_Init(&argc, &argv);
```

Going multi-node with MPI (possibly blended with OpenMP)

solver.cu

```
cudaMalloc(&d_x, …));
```

CUDA version for impressive single-node performance

solver_mpi.cu

```
MPI_Recv(d_x, …);
```

CUDA-aware MPI: getting the best out of Piz Daint

Possible Scenarios

What if …

1. … we want to introduce a modification at the algorithmic/numerical level?

2. … our application has a broad user community and it must run efficiently on a variety of platforms?

3. … our code consists of thousands (if not millions) LOC?

The explosion of hardware architectures made this development model obsolete!

A Real-Case Example : COSMO

- Limited-area model developed by the **Co**nsortium for **S**mall-Scale **Mo**deling.
- Run operationally by 7 national weather services and used by several academic institutions as a research tool.
- Two target architectures: CPUs and GPUs.
- Around 330K lines of F90 code and 90K lines of C/C++ code.
- Cost of porting the full code base to GPU: approx. 20-30 Man-Years!

Separation of Concerns

Domain expert	Performance expert
Answer scientific research questions	Write optimized code for target platform
Declarative programming style: Focus on **what** you want to do	Imperative programming style: Focus on **how** to do it
Common data access interface: e.g. data[i, j, k]	Storage and memory allocation: e.g. C-layout vs F-layout
Computation kernels: Calculations for a single grid point	Control structure (e.g. for loops): Optimized data traversal
Individual operators (“grains”)	Final computation: Detect and exploit parallelism b/w grains

Overarching Goals

- Single **hardware-agnostic** application code.
- Easy to implement.
- Easy to **read**.
- Easy to **maintain**.
- **Performance portable**.

Domain Specific Languages (DSLs)

- Programming language tailored for a specific class of problems.
- Higher level of abstraction w.r.t. a general purpose language.
- Intended to be used by domain experts, who may not be fluent in programming.
- Abstractions and notations much aligned to concepts and rules from the domain.
- Some examples:
  - Machine Learning: TensorFlow (Keras)
  - Scientific Computing: Kokkos, FEniCS, FreeFEM
  - Fluid Dynamics: OpenFOAM
  - Image Processing: Halide
  - Stencils: Ebb, Taichi, GT4Py

GT4Py

- High-performance implementation of a stencil kernel from a high-level definition.
- GT4Py is a domain specific **library** which exposes a domain specific **language** (GTScript) to express the stencil logic.
- GTScript is embedded in Python (**eDSL**).
  - Legal Python syntax and (almost) legal Python semantics.
- GT4Py = **G**rid**T**ools **For** **Py**thon
  - Harnessing the C++ GridTools ecosystem to generate native implementations of the stencils.
- Emphasis on tight integration with scientific Python stack.

Definitions Function

```
import gt4py as gt
from gt4py.gtscript import Field, PARALLEL, computation, interval
import numpy as np

f64 = np.float64

def laplacian_defs(in_field: Field[f64], out_field: Field[f64]):
    with computation(PARALLEL), interval(...):
        out_field = (
          - 4. * in_field[ 0,  0, 0]
          +      in_field[-1,  0, 0]
          +      in_field[+1,  0, 0]
          +      in_field[ 0, -1, 0]
          +      in_field[ 0, +1, 0] )
```

Regular (named) function

Definitions Function

```
import gt4py as gt
from gt4py.gtscript import Field, PARALLEL, computation, interval
import numpy as np

f64 = np.float64

def laplacian_defs(in_field: Field[f64], out_field: Field[f64]):
    with computation(PARALLEL), interval(...):
        out_field = (
          - 4. * in_field[ 0,  0, 0]
          +      in_field[-1,  0, 0]
          +      in_field[+1,  0, 0]
          +      in_field[ 0, -1, 0]
          +      in_field[ 0, +1, 0] )
```

Input and output fields
(object-oriented interface)

Definitions Function

```
import gt4py as gt
from gt4py.gtscript import Field, PARALLEL, computation, interval
import numpy as np

f64 = np.float64

def laplacian_defs(in_field: Field[f64], out_field: Field[f64]):
    with computation(PARALLEL), interval(...):
        out_field = (
            - 4. * in_field[ 0,  0, 0]
            +      in_field[-1,  0, 0]
            +      in_field[+1,  0, 0]
            +      in_field[ 0, -1, 0]
            +      in_field[ 0, +1, 0] )
```

Field descriptors
as type annotations

Definitions Function

```
import gt4py as gt
from gt4py.gtscript import Field, PARALLEL, computation, interval
import numpy as np

f64 = np.float64

def laplacian_defs(in_field: Field[f64], out_field: Field[f64]):
    with computation(PARALLEL), interval(...):
        out_field = (
          - 4. * in_field[ 0,  0, 0]
          +      in_field[-1,  0, 0]
          +      in_field[+1,  0, 0]
          +      in_field[ 0, -1, 0]
          +      in_field[ 0, +1, 0] )
```

Any computation must be wrapped in a **with** construct which can be thought of as being a k-loop

Definitions Function

```
import gt4py as gt
from gt4py.gtscript import Field, PARALLEL, computation, interval
import numpy as np

f64 = np.float64

def laplacian_defs(in_field: Field[f64], out_field: Field[f64]):
    with computation(PARALLEL), interval(...):
        out_field = (
          - 4. * in_field[ 0,  0, 0]
          +     in_field[-1,  0, 0]
          +     in_field[+1,  0, 0]
          +     in_field[ 0, -1, 0]
          +     in_field[ 0, +1, 0] )
```

Iteration order in the vertical direction : PARALLEL, FORWARD, BACKWARD

Definitions Function

```
import gt4py as gt
from gt4py.gtscript import Field, PARALLEL, computation, interval
import numpy as np

f64 = np.float64

def laplacian_defs(in_field: Field[f64], out_field: Field[f64]):
    with computation(PARALLEL), interval(...):
        out_field = (
          - 4. * in_field[ 0,  0, 0]
          +      in_field[-1,  0, 0]
          +      in_field[+1,  0, 0]
          +      in_field[ 0, -1, 0]
          +      in_field[ 0, +1, 0] )
```

Vertical region of application:
... = full column

Definitions Function

```
import gt4py as gt
from gt4py.gtscript import Field, PARALLEL, computation, interval
import numpy as np

f64 = np.float64

def laplacian_defs(in_field: Field[f64], out_field: Field[f64]):
    with computation(PARALLEL), interval(...):
        out_field = (
          - 4. * in_field[ 0,  0, 0]
          +      in_field[-1,  0, 0]
          +      in_field[+1,  0, 0]
          +      in_field[ 0, -1, 0]
          +      in_field[ 0, +1, 0] )
```

Each statement (or **stage**) can be thought of as an ij-loop

Definitions Function

```
import gt4py as gt
from gt4py.gtscript import Field, PARALLEL, computation, interval
import numpy as np

f64 = np.float64

def laplacian_defs(in_field: Field[f64], out_field: Field[f64]):
    with computation(PARALLEL), interval(...):
        out_field = (
          - 4. * in_field[ 0,  0, 0]
          +      in_field[-1,  0, 0]
          +      in_field[+1,  0, 0]
          +      in_field[ 0, -1, 0]
          +      in_field[ 0, +1, 0] )
```

Neighboring points accessed through offsets

Definitions Function

```python
import gt4py as gt
from gt4py.gtscript import Field, PARALLEL, computation, interval
import numpy as np

f64 = np.float64

def laplacian_defs(in_field: Field[f64], out_field: Field[f64]):
    with computation(PARALLEL), interval(...):
        out_field = (
          - 4. * in_field[ 0,  0, 0]
          +      in_field[-1,  0, 0]
          +      in_field[+1,  0, 0]
          +      in_field[ 0, -1, 0]
          +      in_field[ 0, +1, 0] )
```

1st horizontal dimension

2nd horizontal dimension

Vertical dimension

Neighboring points accessed through offsets

Definitions Function

```python
import gt4py as gt
from gt4py.gtscript import Field, PARALLEL, computation, interval
import numpy as np

f64 = np.float64

def laplacian_defs(in_field: Field[f64], out_field: Field[f64]):
    with computation(PARALLEL), interval(...):
        out_field = (
          - 4. * in_field[ 0,  0, 0]
          +      in_field[-1,  0, 0]
          +      in_field[+1,  0, 0]
          +      in_field[ 0, -1, 0]
          +      in_field[ 0, +1, 0] )
```

No for loops!
No return statement!

Compilation

- A stencil needs to be compiled for a given **backend**:

```
backend = “gtx86”
laplacian = gt.stencil(backend, laplacian_defs)
```

- Available backends:
  - Python: “debug” (for loops), “numpy” (vectorized syntax);
  - C++: “gtx86” (x86), “gtmc” (MIC), “gtcuda” (NVIDIA GPU).

- For GT-based backends, compilation consists of three steps:
  1) Generate optimized code for the target architecture.
  2) Compile the automatically generated code.
  3) Build Python bindings to that code.

Storages

- The compilation returns a callable object which can be invoked on GT4Py storages.
- Storages have optimal memory **strides**, **alignment** and **padding**.
- gt.storage provides functionalities to allocate storages …

```
nx, ny, nz = 128, 128, 64
def_orig = (1, 1, 0)
out_field = gt.storage.zeros(
        backend, def_orig, (nx, ny, nz), dtype=f64 )
```

… and convert NumPy arrays into valid storages:

```
in_field = gt.storage.from_array(
        np.randon.rand(nx, ny, nz), backend, def_orig, dtype=f64 )
```

Storages

- Storages can be accessed as NumPy arrays:

```
in_field[0, 0, 0] = 4.
print(in_field[0, 0, 0])
```

Running

- Running computations is as simple as a function call:

```
laplacian(
        in_field=in_field,
        out_field=out_field,
        origin=(1, 1, 0),
        domain=(nx – 2, ny – 2, nz)
)
```

Bindings b/w the symbols used within the definitions fct. and the arrays holding the data

Running

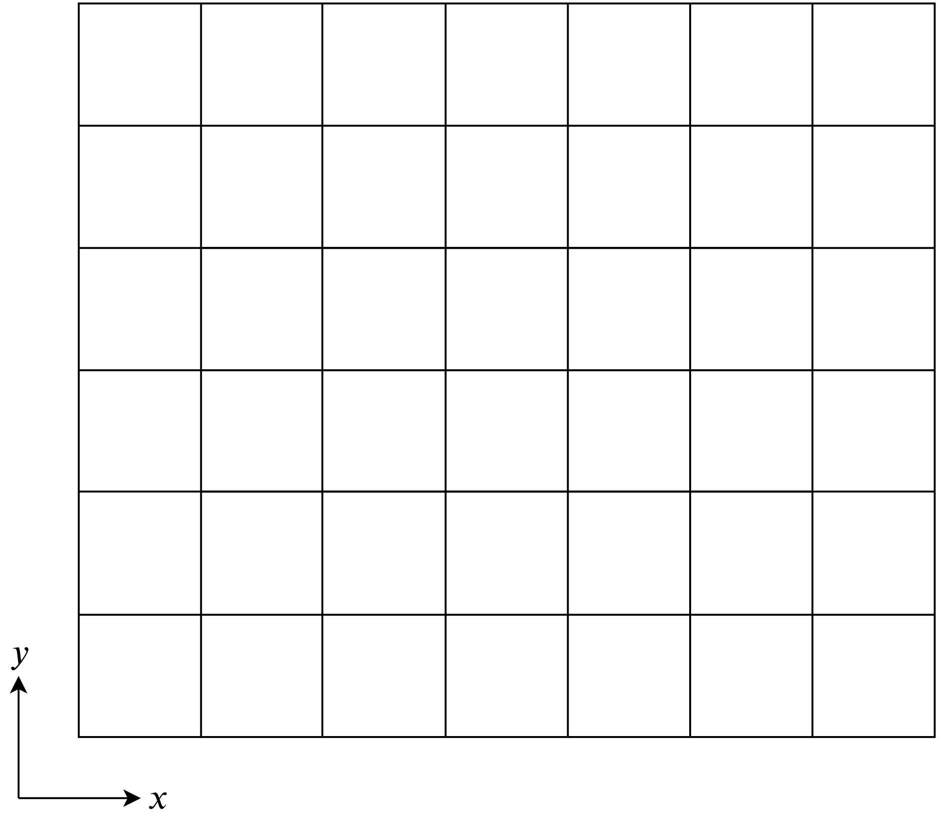
- Running computations is as simple as a function call:

```
laplacian(
        in_field=in_field,
        out_field=out_field,
        origin=(1, 1, 0),
        domain=(nx – 2, ny – 2, nz)
)
```

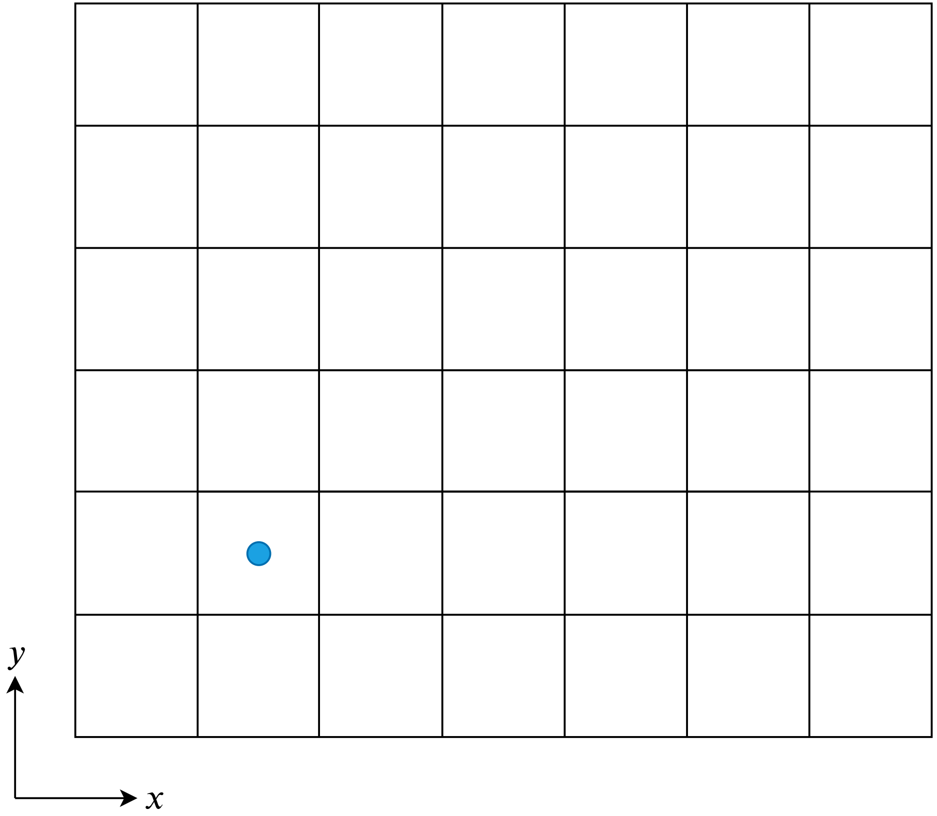
Origin and extent of the computation domain

- out_field now contains the results of the computation.

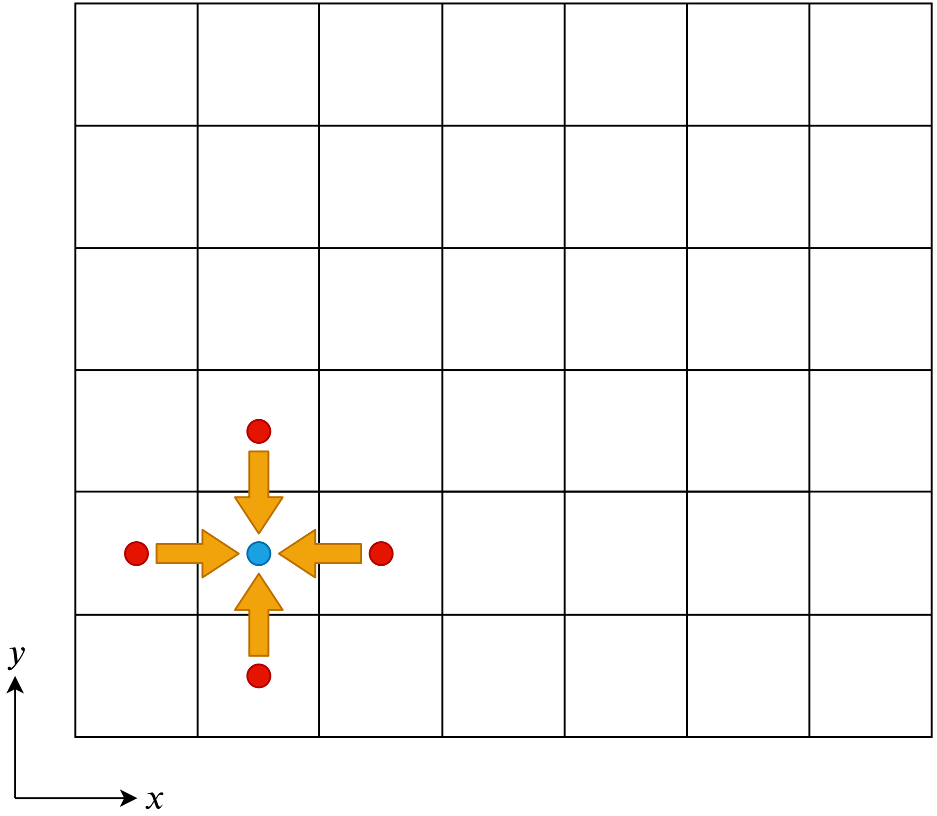
Region of application



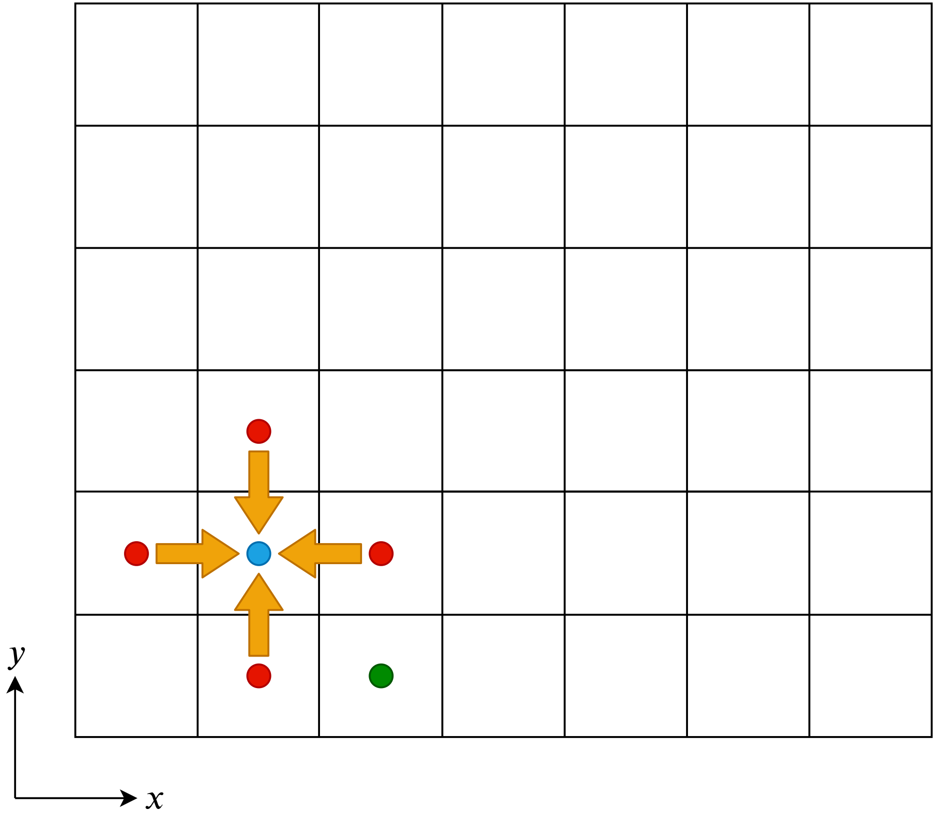
Region of application



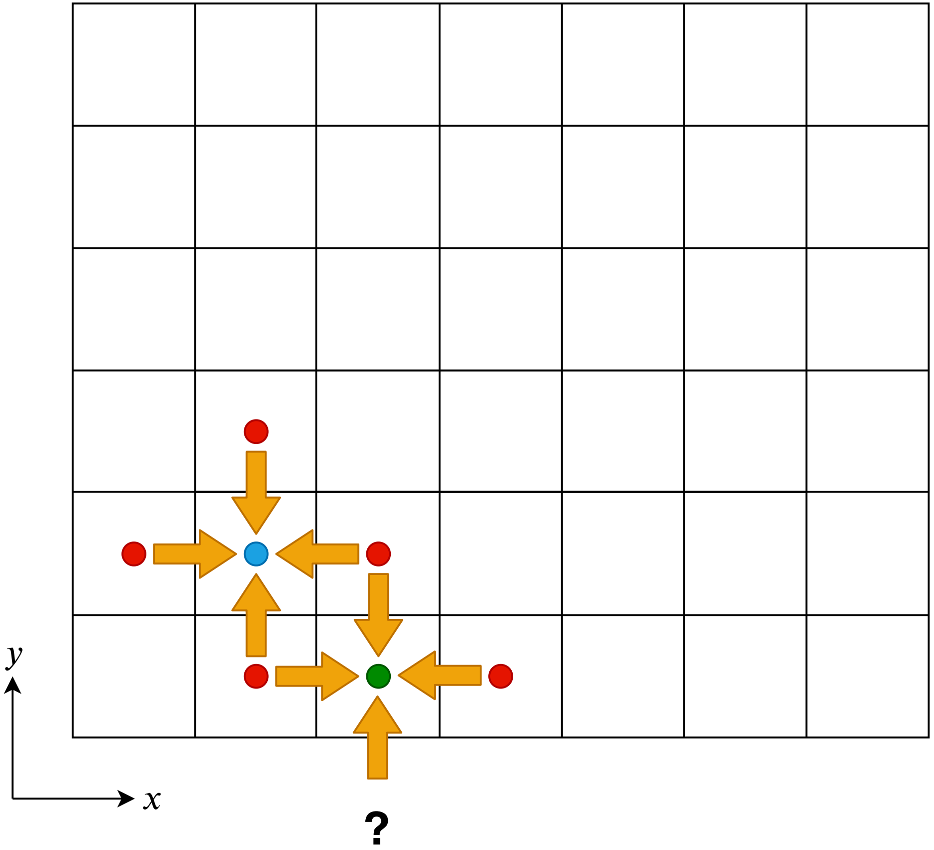
Region of application



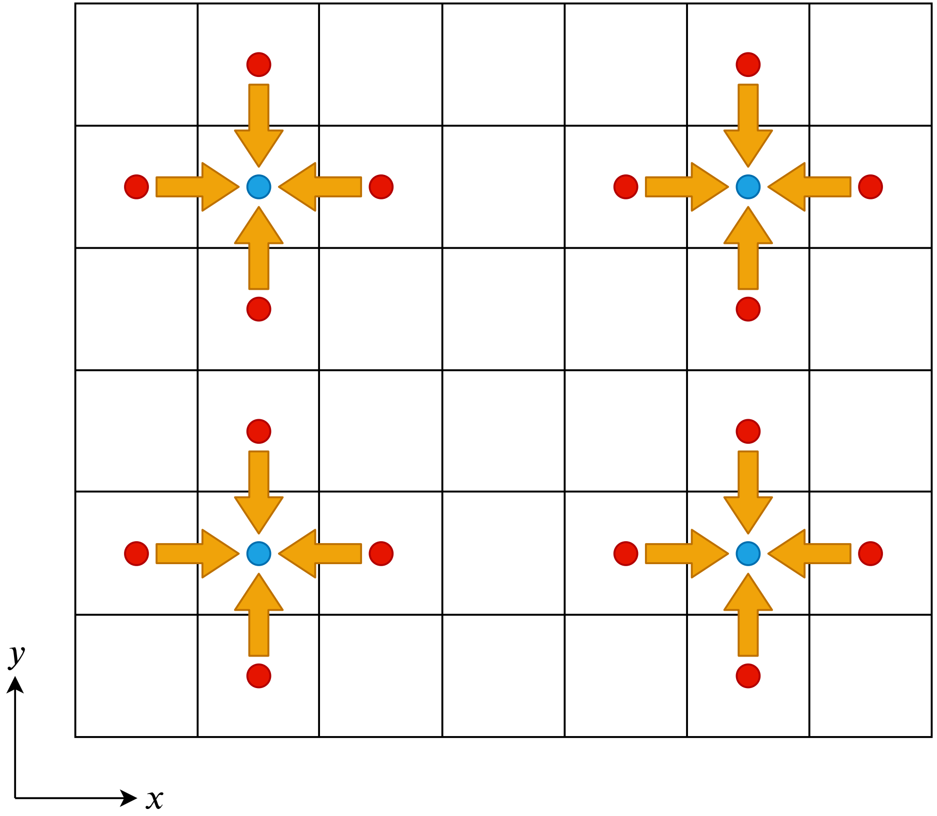
Region of application



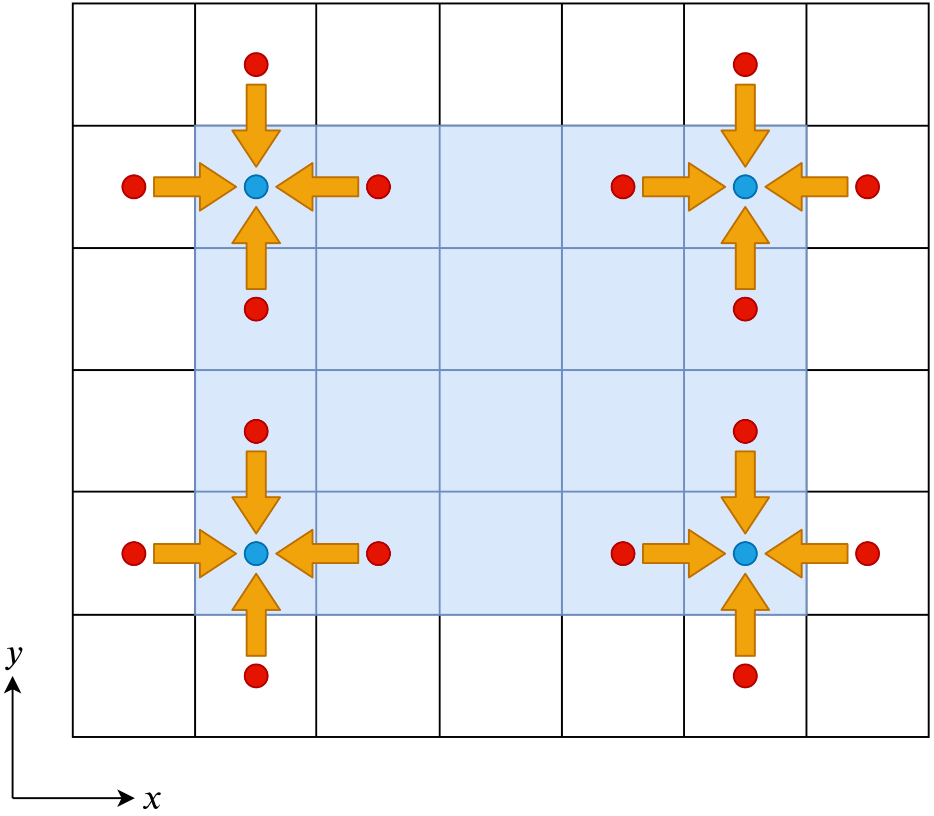
Region of application



Region of application

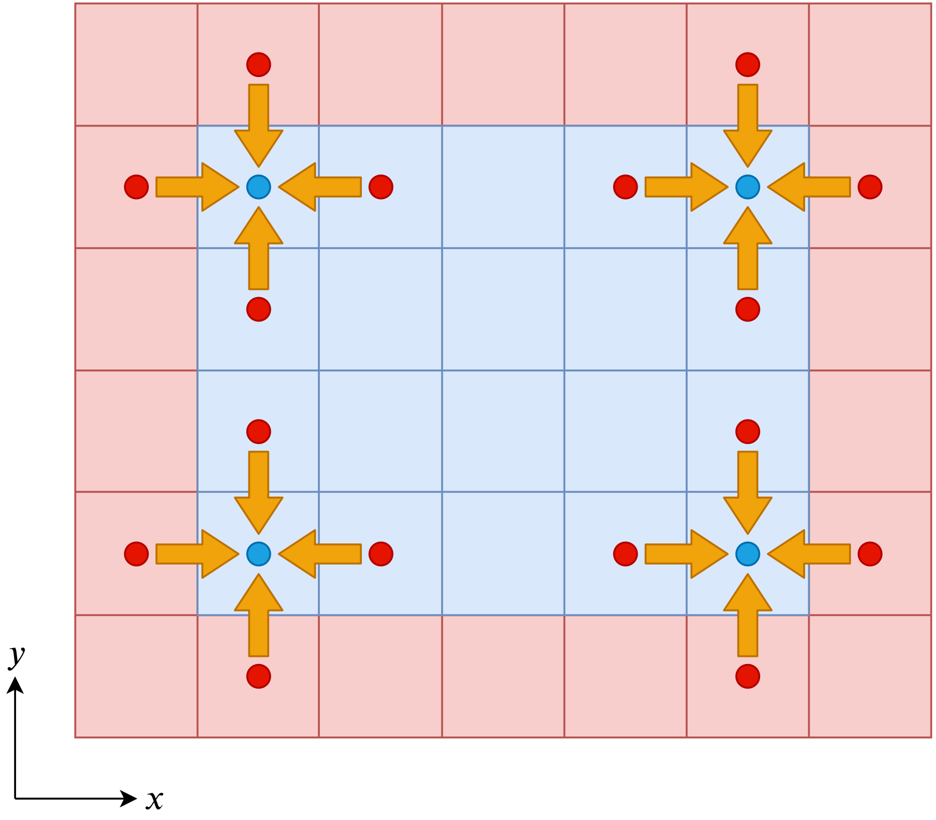


Region of application



(Largest) computation domain

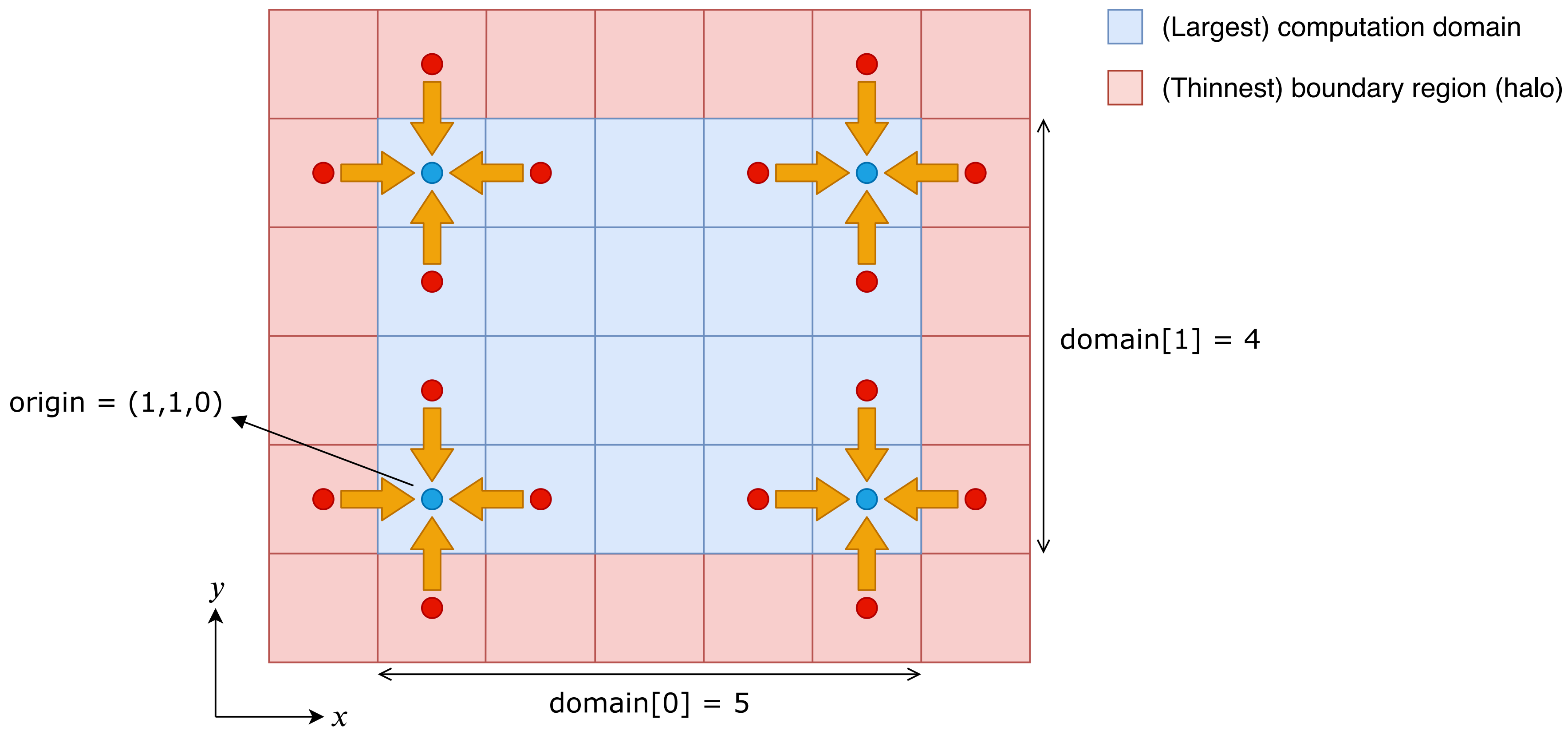
Region of application



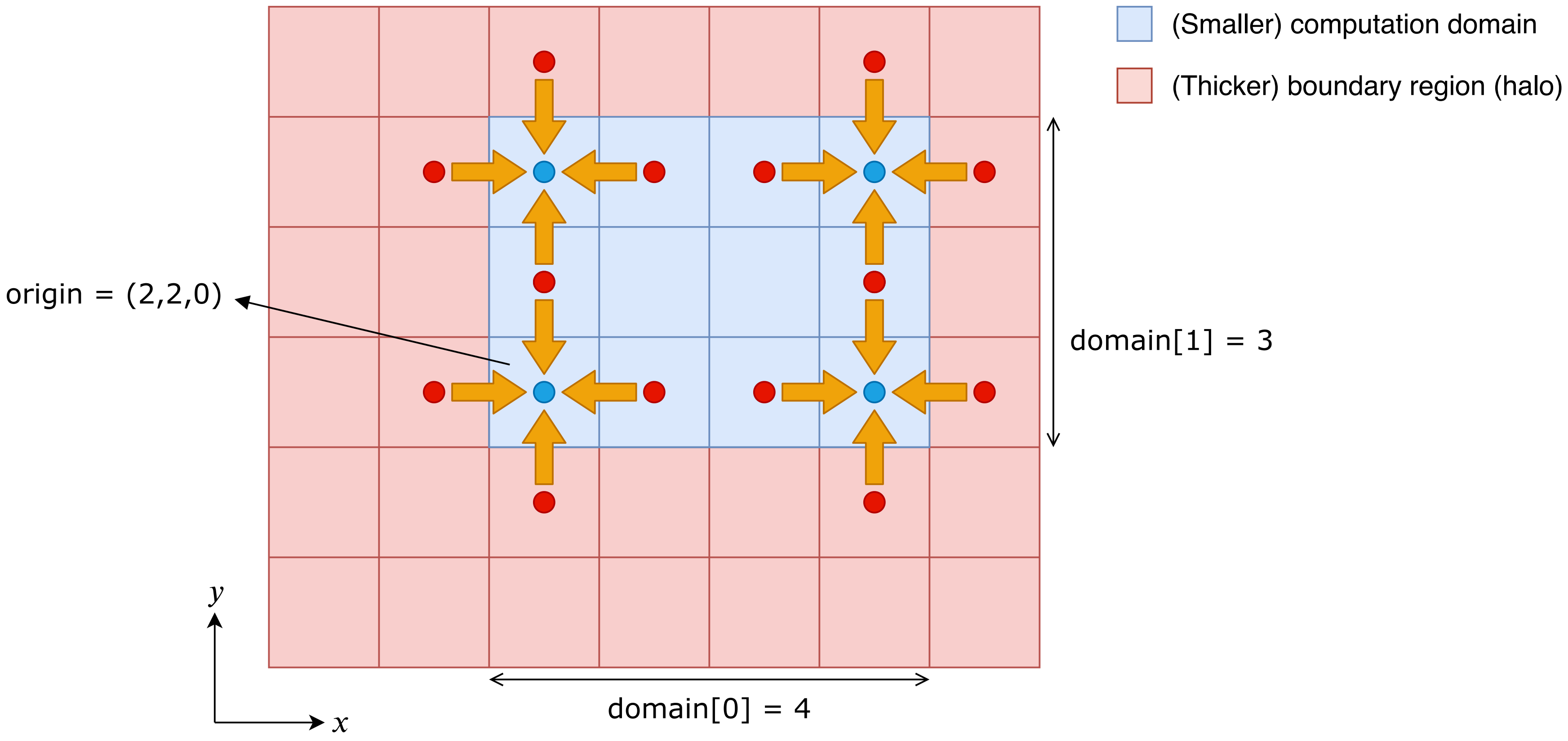
(Largest) computation domain

(Thinnest) boundary region (halo)

Region of application



Region of application



Disadvantages of a DSL

- Lack of generality: A DSL is not a complete ontology!
- Debugging on the generated code.
- Cost of developing and maintaining the DSL compiler toolchain.

Conclusions

- High-level programming techniques hide the complexities of the underlying architecture to the end user.
- DSL allows to target multiple platforms without polluting the application code with hardware-specific boilerplate code.
- GT4Py is a Python framework to write performance portable applications in the weather and climate area. It ships with a DSL to write stencil computations.

Lab Exercises

01-GT4Py-sumdiff.ipynb

- Compare NumPy, CuPy and GT4Py on the sum-diff stencil (demo).

02-GT4Py-laplacian.ipynb

- Compare NumPy, CuPy and GT4Py on the Laplacian stencil (demo).

03-GT4Py-concepts.ipynb

- Digest the main concepts of GT4Py.
- Get familiar with writing, compiling and running stencils.
- Get insights on the internal data-layout of the storages.

04-GT4Py-stencil2d.ipynb

- Step-by-step porting of stencil2d.py to GT4Py.
- Write two alternative versions of stencil2d-gt4py-v0.py

Before Starting

1. Pull the latest commit from the Github repo.

2. Make sure that your .jupyterhub.env contains the following lines:

```
module load Boost
module load cudatoolkit
NVCC_PATH=$(which nvcc)
CUDA_PATH=$(echo $NVCC_PATH | sed -e "s/\/bin\/nvcc//g")
export CUDA_HOME=$CUDA_PATH
export LD_LIBRARY_PATH=$CUDA_PATH/lib64:$LD_LIBRARY_PATH
```

After updating your .jupyterhub.env from a terminal (see Oli’s post in #general): terminate the JupyterLab session and fire up a new one.

References

Broad introduction to DSLs:
https://www.jetbrains.com/mps/concepts/domain-specific-languages/

Designing APIs - The Case of GridTools (M. Bianco):
https://www.youtube.com/watch?v=IzWxgFcJFdk&list=PL1tk5lGm7zvQOXi24s586pwDF_yseZ-80&index=7
https://www.youtube.com/watch?v=2tCVOkbediU&list=PL1tk5lGm7zvQOXi24s586pwDF_yseZ-80&index=9

GT repo: https://github.com/GridTools

GT4Py repo: https://github.com/GridTools/gt4py