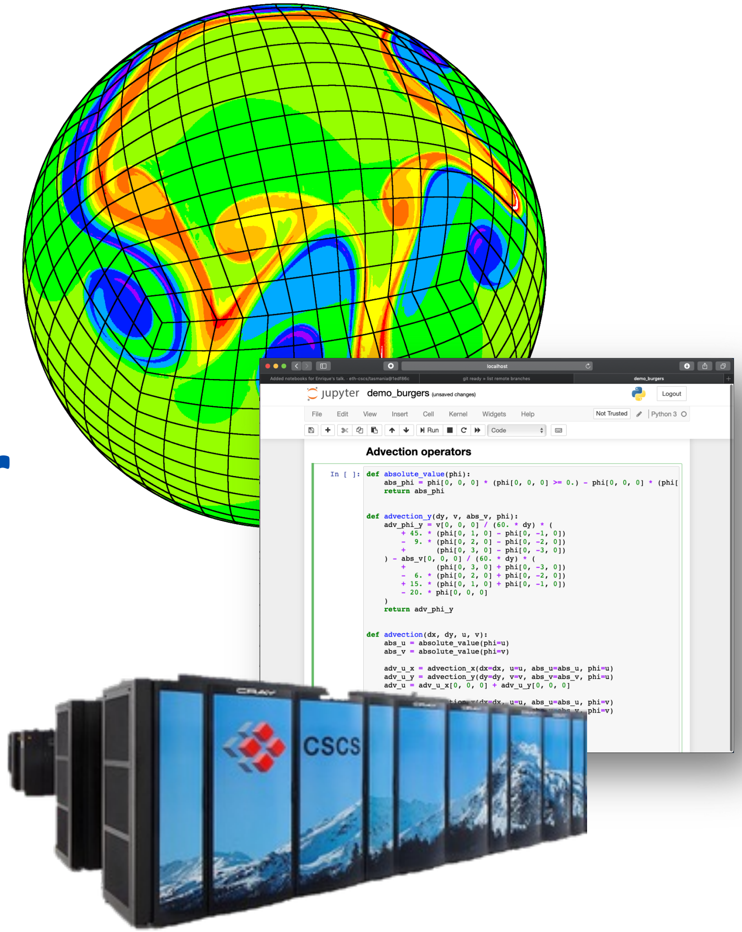
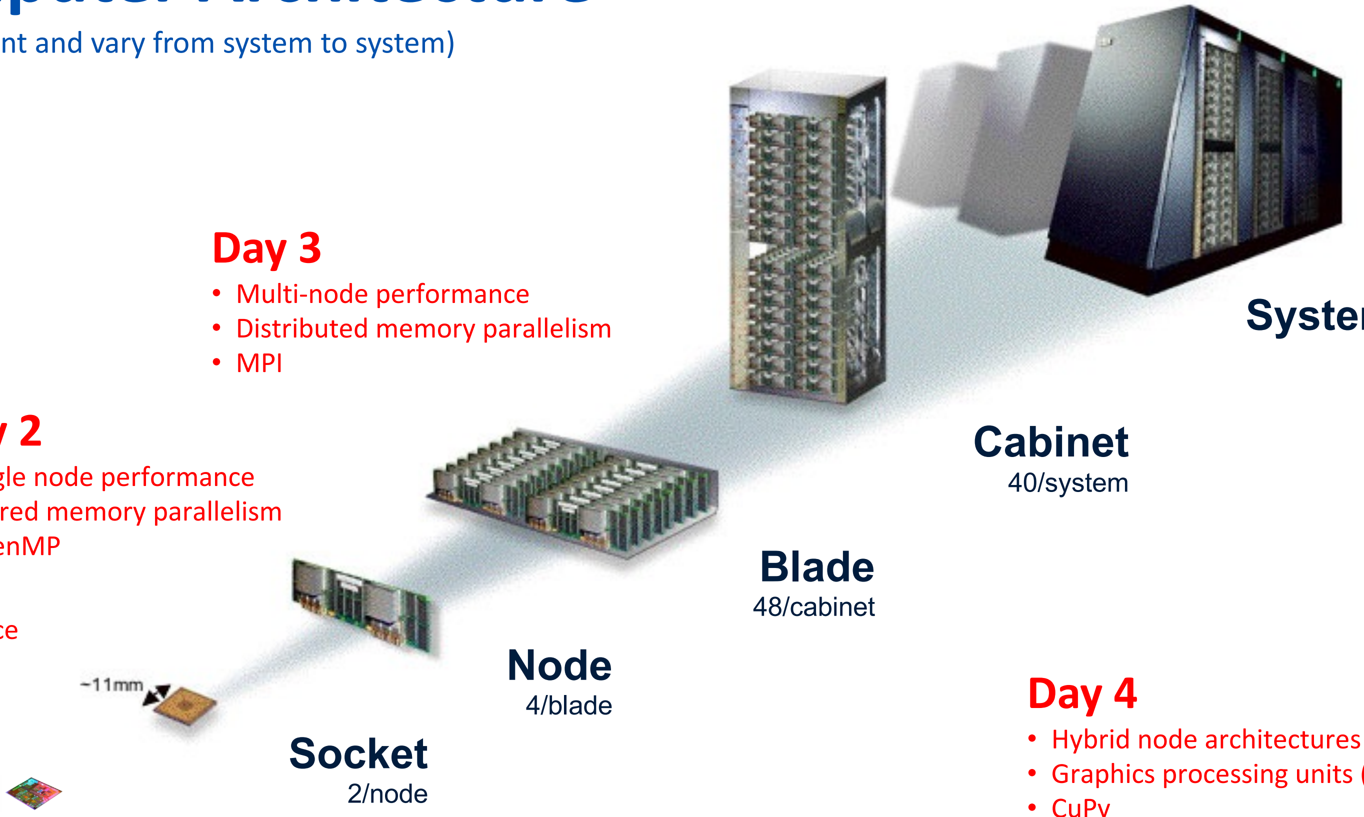
High Performance Computing for Weather and Climate (HPC4WC)

Content: High-Level Programming
Lecturer: Oliver Fuhrer
Block course 701-1270-00L
Summer 2023



Supercomputer Architecture

(Numbers are for Piz Daint and vary from system to system)



Day 1

- Single core performance
- Caches

Day 2

- Single node performance
- Shared memory parallelism
- OpenMP

Day 3

- Multi-node performance
- Distributed memory parallelism
- MPI

Day 4

- Hybrid node architectures
- Graphics processing units (GPUs)
- CuPy

Future of HPC in Weather and Climate?

Yesterday

- x86 CPU
- MPI
- Fortran
- OpenMP

Today

- GPU
- C++
- Python
- mpi4py
- CuPy

Tomorrow

- Specialized hardware?
- ASIC?
- FPGA?
- ML?
- Domain-specific languages?
- GT4Py?

Learning Goals

- Understand what a domain-specific language (DSL) is.
- Understand how a DSL helps in writing hardware-agnostic and 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.F90

```
#include <vector>
```

Naïve implementation in a compiled language (e.g. F90, C++)

solver_omp.F90

```
#pragma omp parallel
```

Multi-threaded version using OpenMP

solver_omp_mpi.F90

```
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…

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

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

…our code consists of thousands (if not millions) lines of code?

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 8 national weather services and used by several academic institutions as a research tool.
- Three target architectures: x86 CPUs, NVIDIA GPUs and NEC vector CPUs
- Around 400K lines of F90 code and 90K lines of C/C++ code.
- Cost of porting the full code base to GPU: approx. 20-30 programmer-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 (The 3 P's)

- **Productivity**
  Easy to implement.
  Easy to **read**.
  Easy to **maintain**.

- **Performance**
  Is **fast**.

- **Portability**
  Single **hardware-agnostic** application code.
  Runs efficiently on **different hardware** targets.

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.

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:
  - Typesetting: LaTeX
  - Machine Learning: PyTorch, TensorFlow (Keras)
  - Scientific Computing: Kokkos, FEniCS, FreeFEM
  - Fluid Dynamics: OpenFOAM
  - Image Processing: Halide, Taichi
  - Stencils: Devito, Ebb, 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.

What Does The GT4Py DSL Need?

```
import numpy as np

def laplacian_np(in_field):
    out_field = np.zeros_like(in_field)
    nx, ny, nz = in_field.shape
        for i in range(1, nx - 1):
            for j in range(1, ny - 1):
                for k in range(0, nz):
                    out_field[i, j, k] = (
                        - 4 * in_field[i, j, k]
                        + in_field[i - 1, j, k]
                        + in_field[i + 1, j, k]
                        + in_field[i, j - 1, k]
                        + in_field[i, j + 1, k] )
        return out_field
```

Input, output, and possibly temporary 3D fields

Nested loops iterating along both horizontal and vertical directions

Math operations

Indices and offsets

Definitions Function

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

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

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
(with type annotations)

Definitions Function

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

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

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

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

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 can be thought of as being an ij-loop

Definitions Function

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

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

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

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

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

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 = "gt:cpu_ifirst"
laplacian = gt.stencil(backend, laplacian_defs)
```

- Backends target different purposes, needs, and computer architectures:
  - Python: "numpy" (vectorized syntax);
  - C++: "gt:cpu_ifirst" (x86), "gt:cpu_kfirst" (MIC), "gt:gpu" and "cuda" (NVIDIA GPU).

- For non-Python backends, compilation consists of three steps:
  1) Generate optimized code for the target architecture (cached in .gt_cache).
  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])
# Output: 4.0
```

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

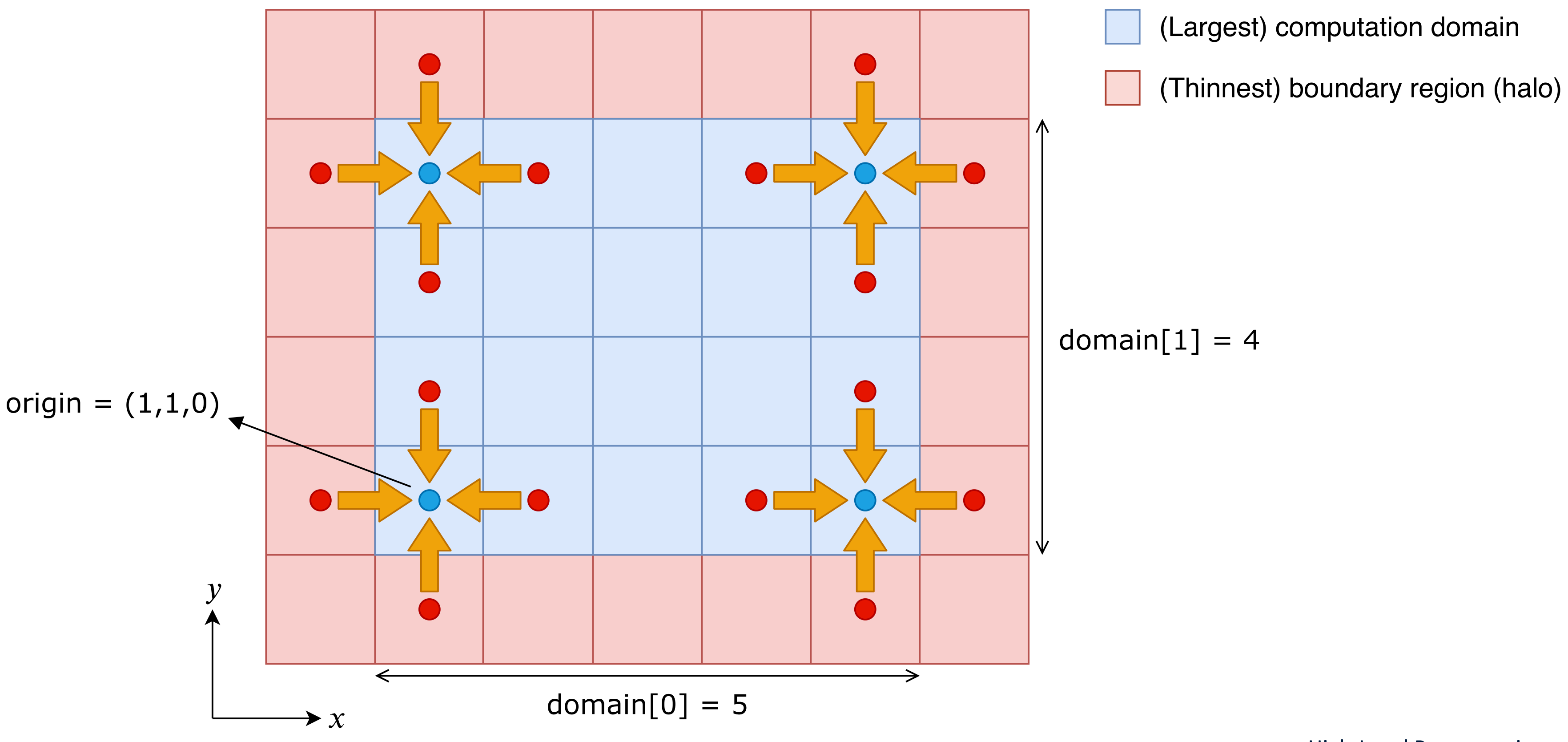
- Running computations is as simple as a function call:

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

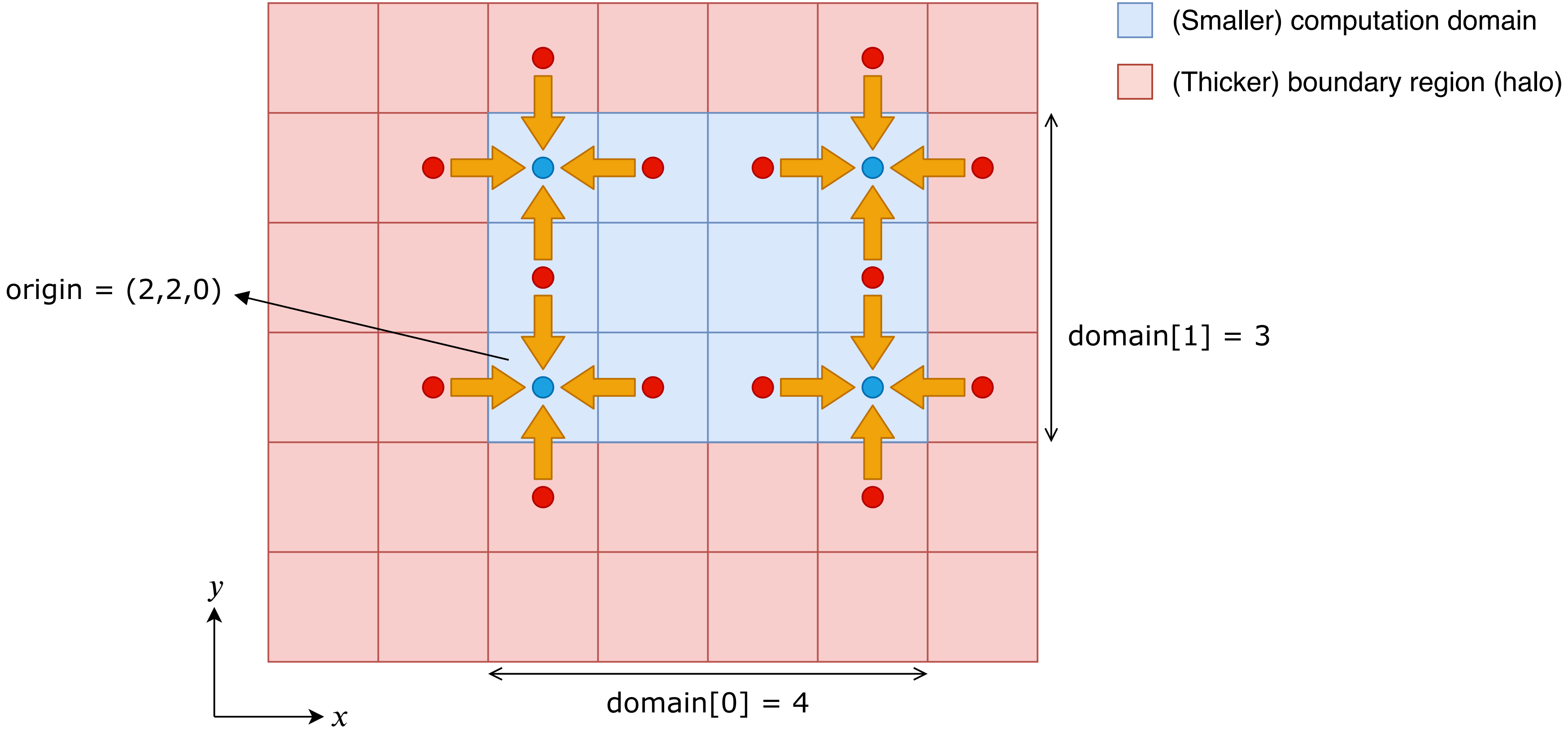
Origin and extent of the computation domain

- out_field now contains the results of the computation.

Region of application



Region of application



Weather and Climate on DSLs

- Several models (FV3, FVM and ICON) being ported to GT4Py
- Other approaches
  - COSMO (MeteoSwiss) dynamical was re-written in C++ using GridTools library.
  - E3SM (US DOE) using the Kokkos library for on-node parallelism.
  - LFric (UK MetOffice)
- Who knows what the future will bring…

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-motivation.ipynb

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

02-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

References

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

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

More in-depth introduction to GT4Py:
https://github.com/VulcanClimateModeling/dsl_workshop