



Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra

Swiss Confederation

HP2C

Preparing a weather prediction and regional climate model for current and emerging hardware architectures.

Oliver Fuhrer (MeteoSwiss), Tobias Gysi (Supercomputing Systems AG), Xavier Lapillonne (C2SM), Carlos Osuna (C2SM), Thomas Schulthess (CSCS/ETH), Mauro Bianco (CSCS)

 **CSCS**
Swiss National Supercomputing Centre

 **CRAY**
THE SUPERCOMPUTER COMPANY

 **SCS**
super computing systems

 **COSMO**
CONSORTIUM FOR SMALL SCALE MODELING

 **C2SM**
Center for Climate
Systems Modeling

 **NVIDIA**

 **DWD**

 **IACETH**

 Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra



Overview

- Motivation
- The COSMO model
- The HP2C Initiative
 - Code analysis
 - Approach
 - Results
 - Outlook
- Conclusions

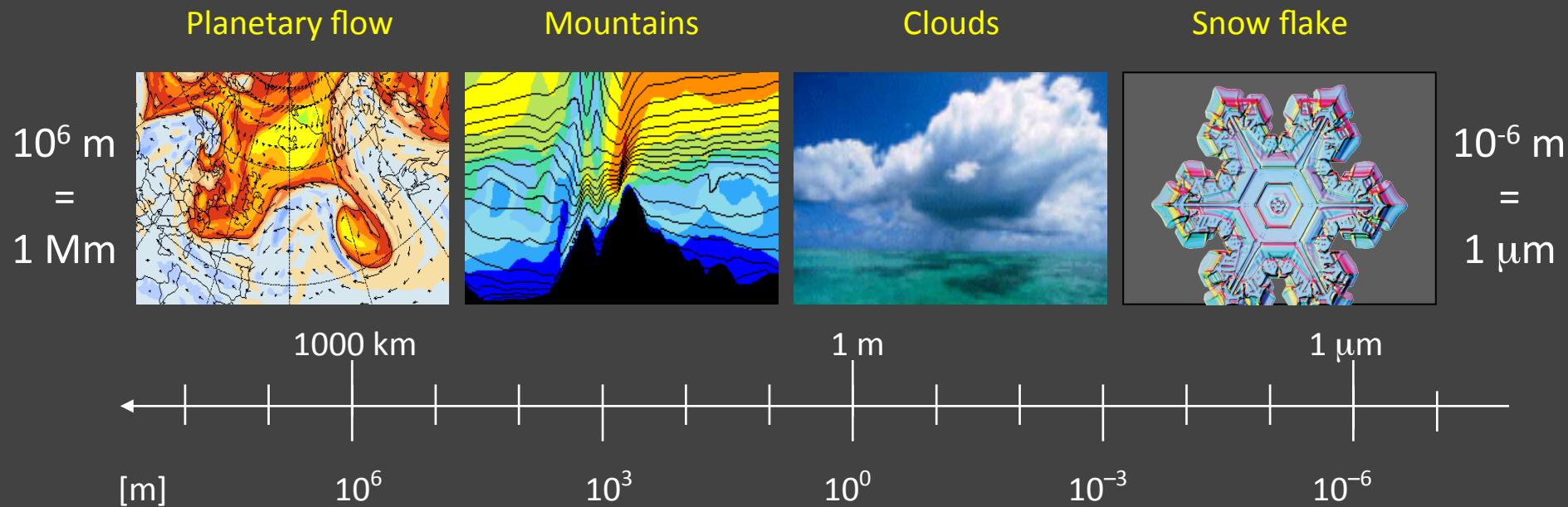


Motivation

What are the reasons that weather and climate simulation require increasing amount of HPC resources?

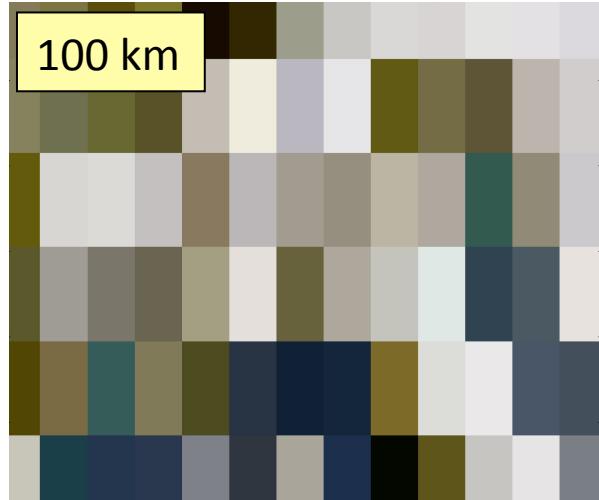
- Higher resolution
- Larger ensembles
- Increasing model complexity

Multi-scale interactions in the weather & climate system

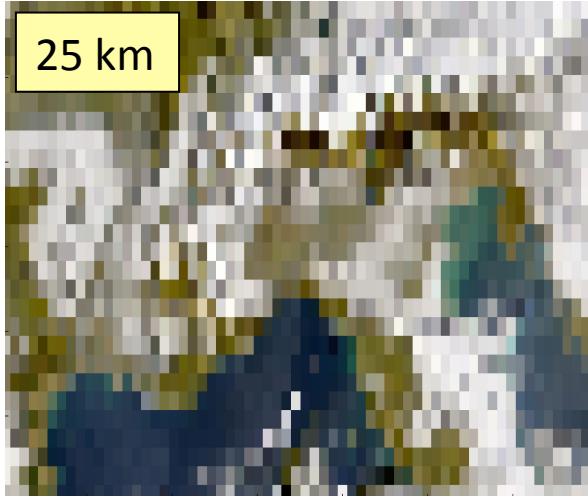




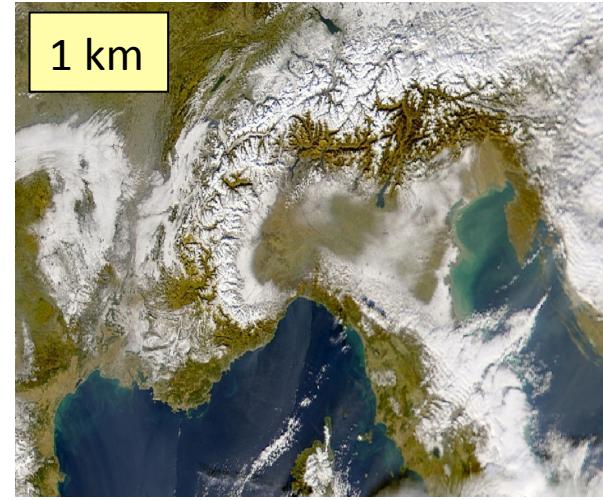
Higher resolution



GCM



RCM



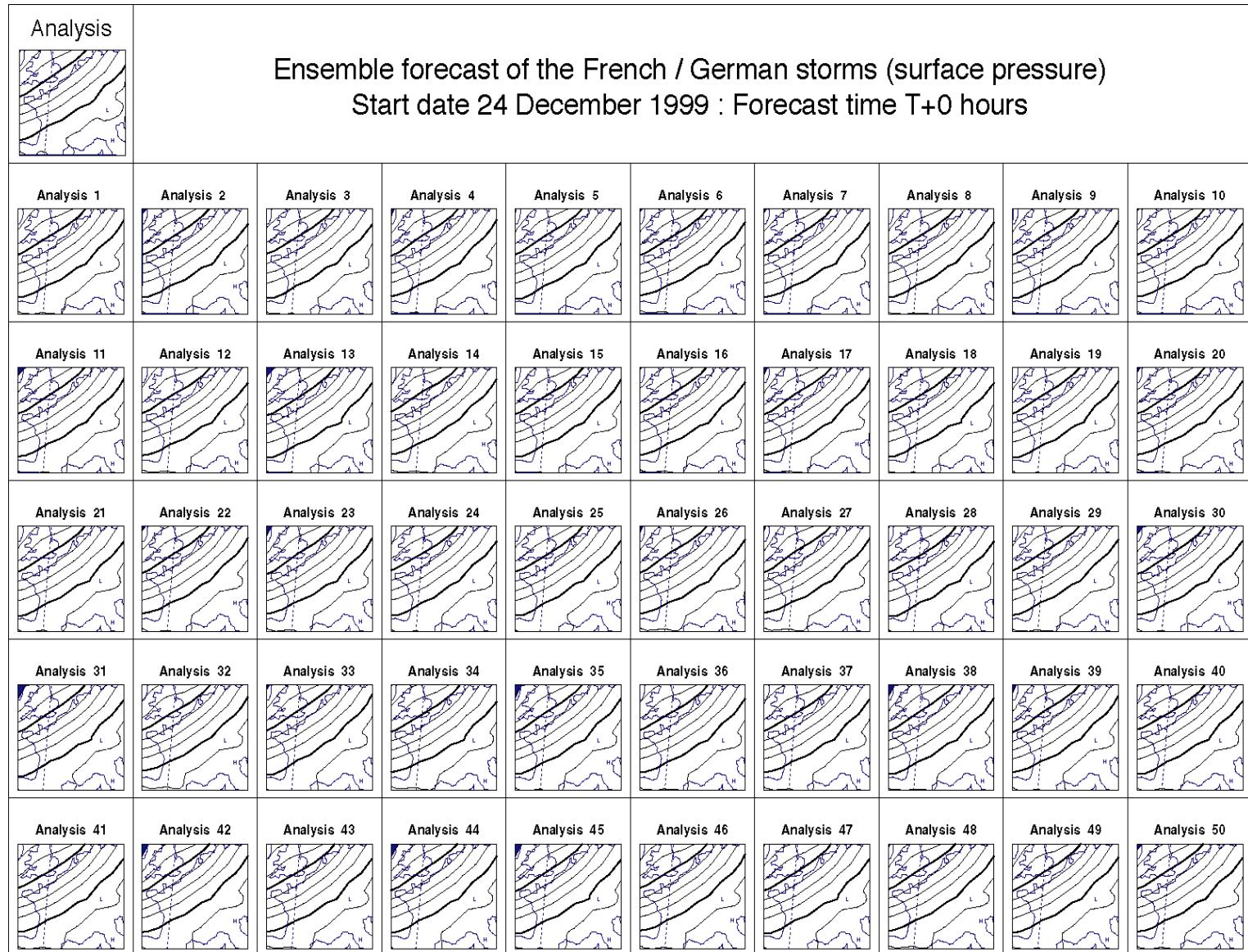
Cloud-Resolving RCM

**Resolution is of key importance to reduce the uncertainty
of climate and weather model predictions**
(e.g. hydrological cycle, extreme events, local predictions)

$2 \times \text{resolution} \approx 10 \times \text{computational cost}$

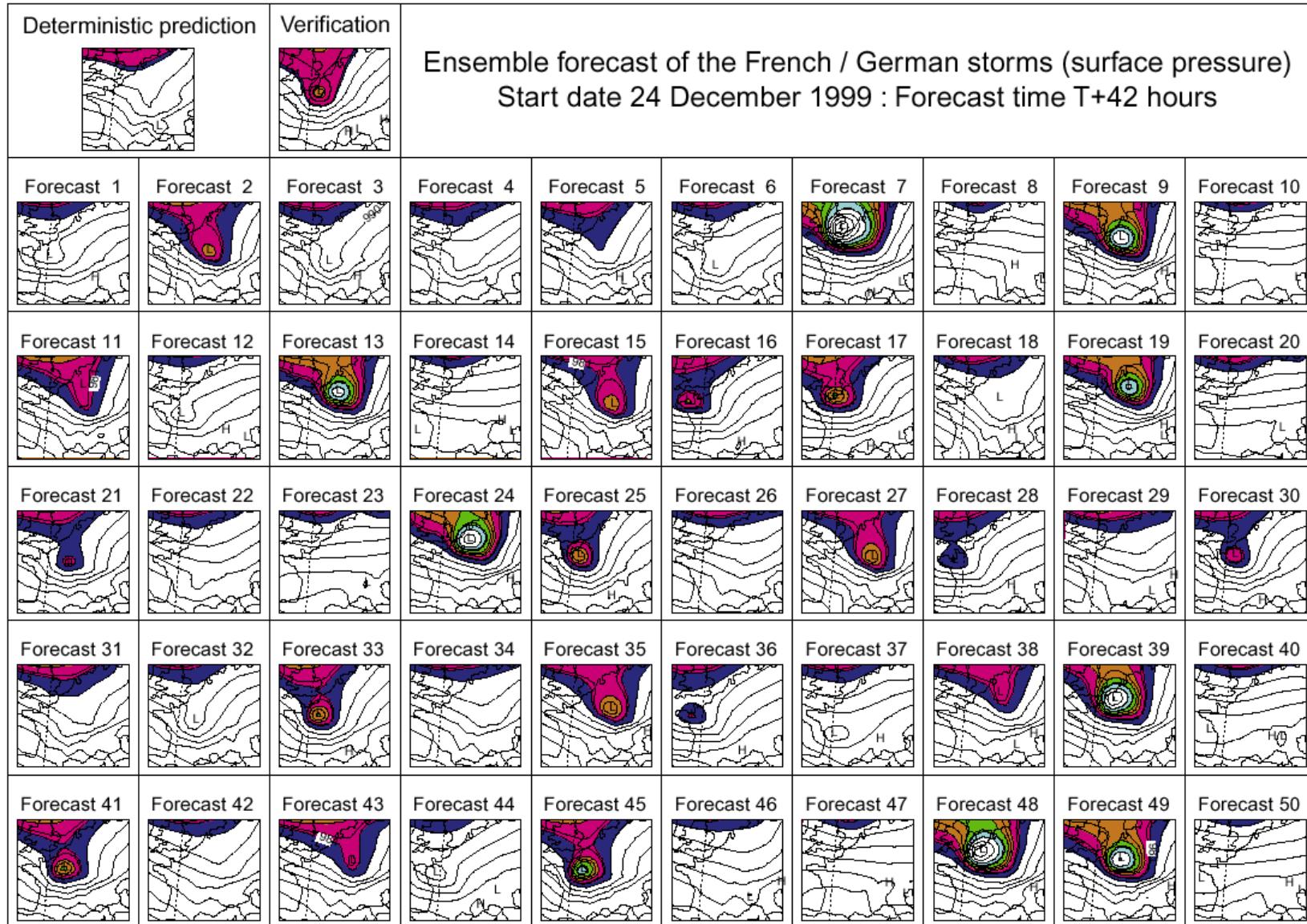


Initial condition (T+0h)



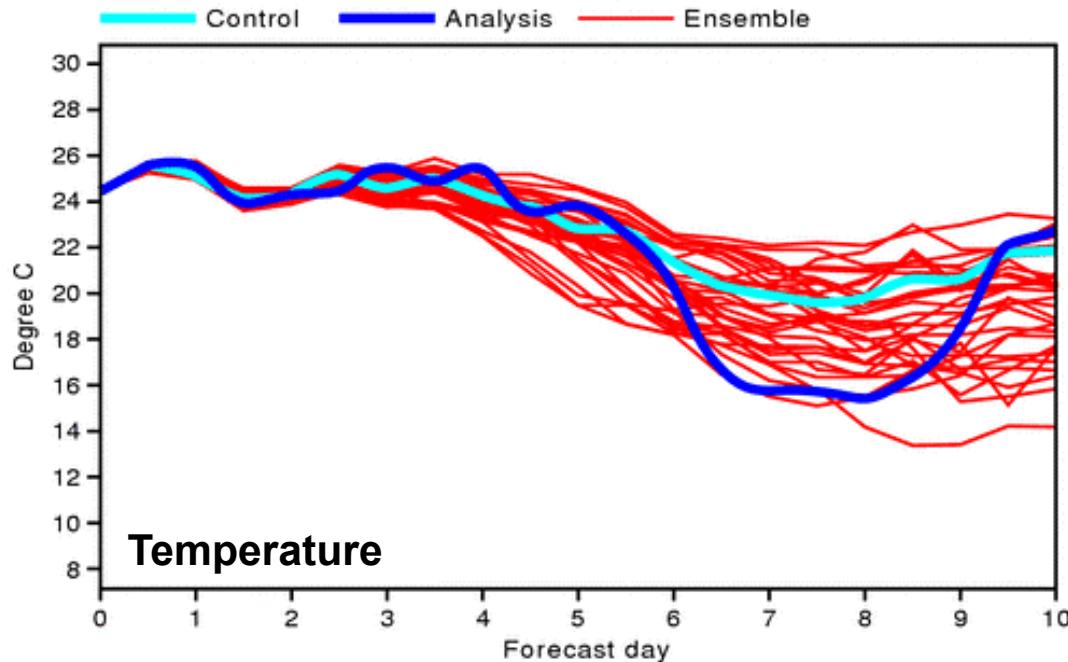


Lothar ensemble forecast (T+42h)





Ensembles

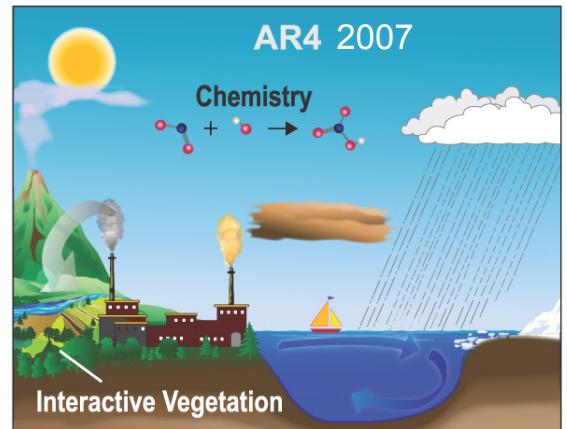
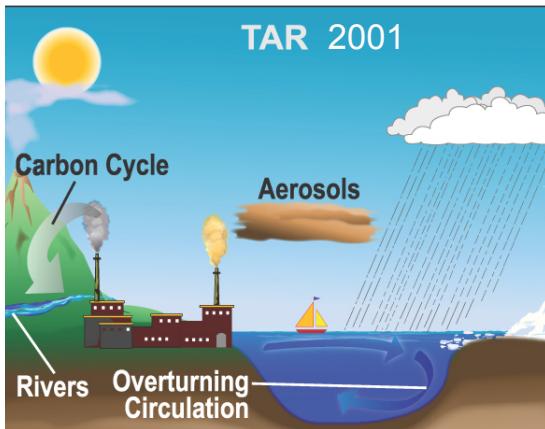
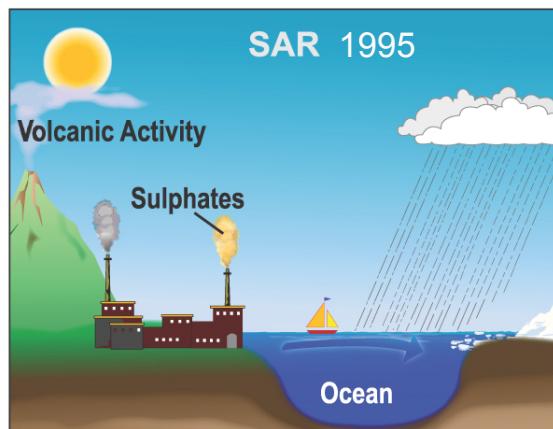
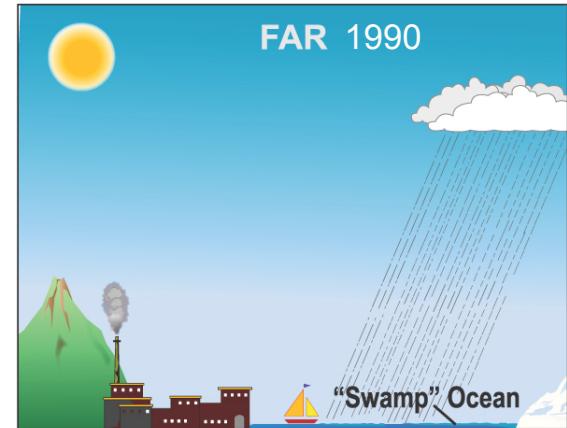
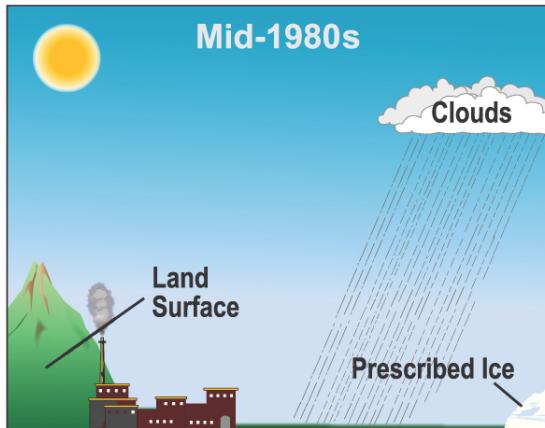
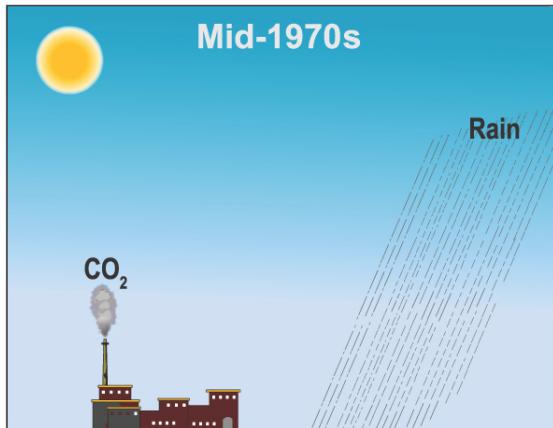


Ensembles are of key importance for quantifying the certainty/uncertainty of a weather or climate prediction

N members \approx N x computational cost



Increasing model complexity





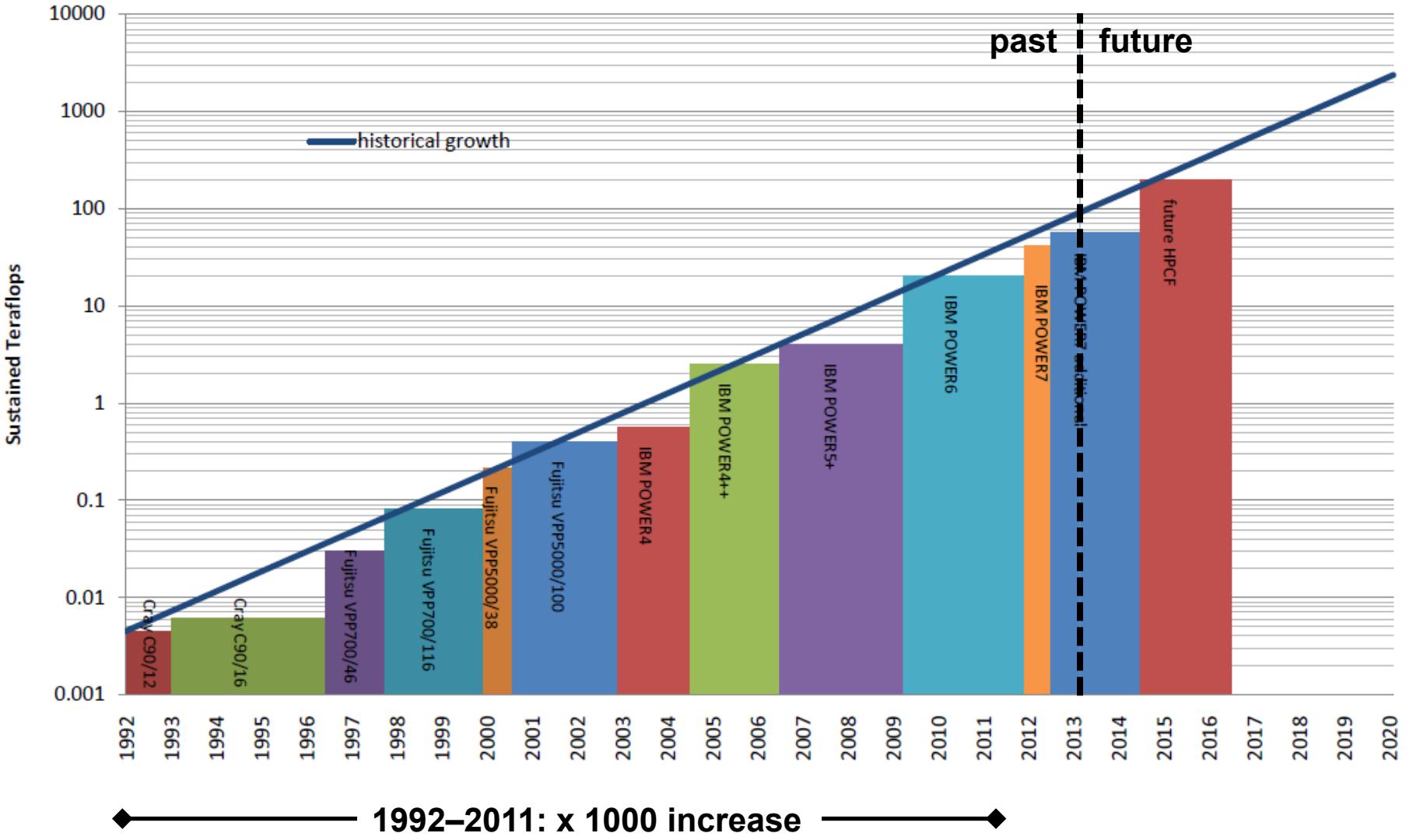
Can the benefit be quantified?

What is the benefit of increasing HPC resources for weather and climate simulation?

- Higher resolution?
- Larger ensembles?
- Increasing model complexity?



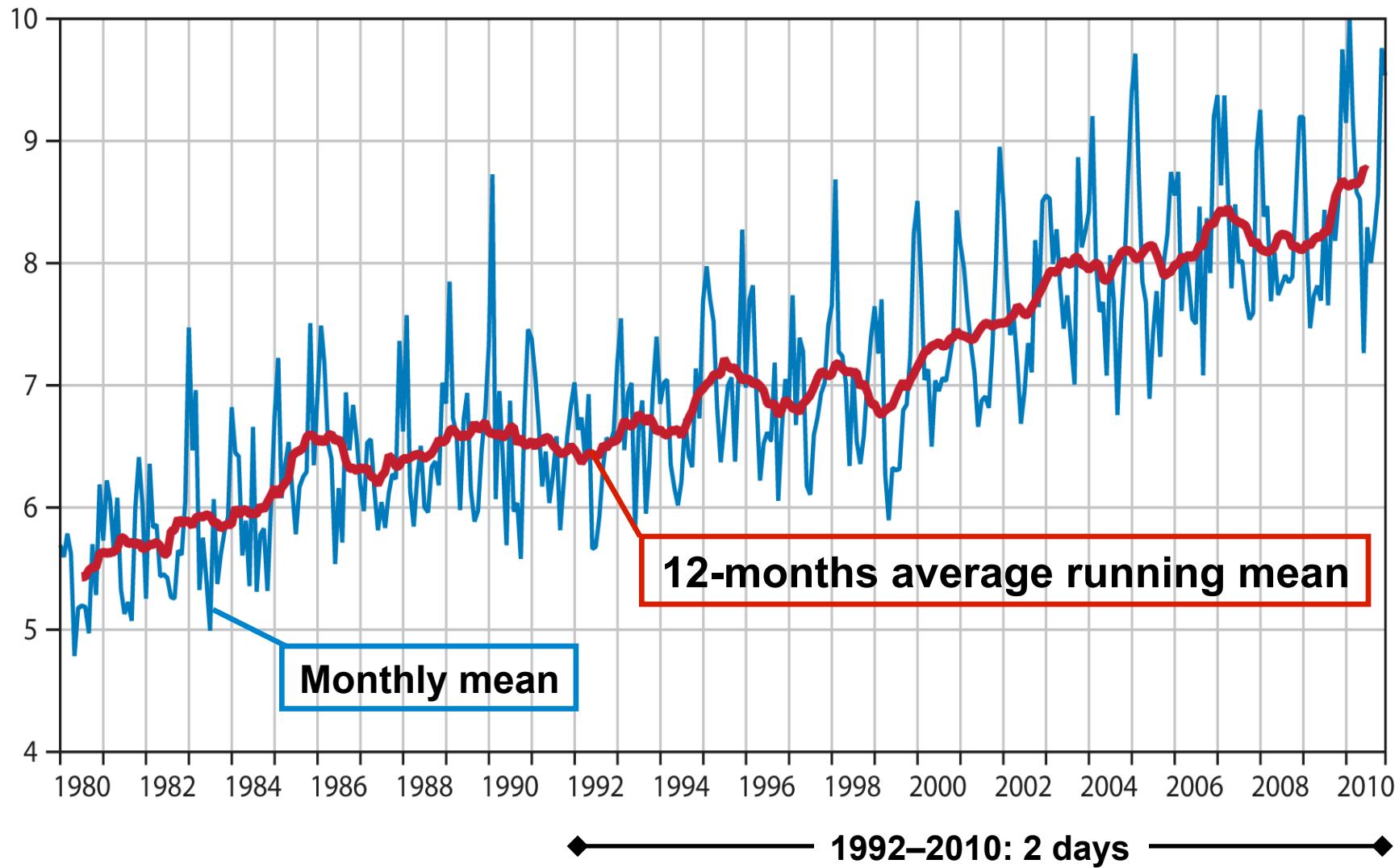
ECWMF sustained peformance





Long-term evolution of NWP scores

ECMWF, ACC 500hPa reaches 60%





Supercomputers are changing!

- **Massive parallelism** – increase in number of cores, stagnant or decreasing clock frequency
- **Less and “slower” memory per thread** – memory bandwidth per instruction/second and thread will decrease, more complex memory hierarchies
- **Heterogeneous hardware** – mixed clusters of CPUs and accelerators (GPUs)
- **Only slow improvements of inter-processor and inter-thread communication** – interconnect bandwidth will improve only slowly
- **Stagnant I/O sub-systems** – technology for long-term data storage will stagnate compared to compute performance

We need to adapt our codes in order be efficient in the future!



COSMO Model

- **Regional weather and climate prediction model**
- Community model
<http://www.clm-community.eu/>
<http://www.cosmo-model.org/>
- O(70) universities and research institutes
- Operational at 7 national weather services

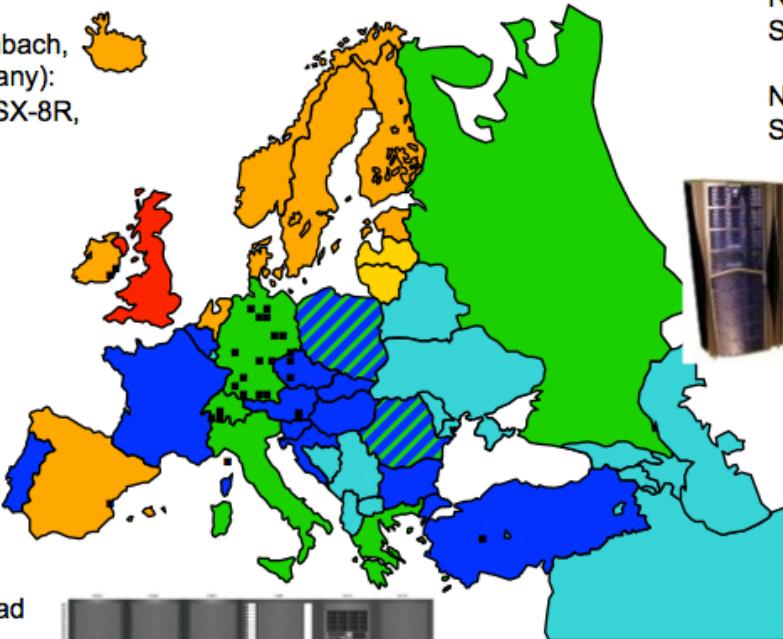


Operational implementations



DWD
(Offenbach,
Germany):
NEC SX-8R,
SX-9

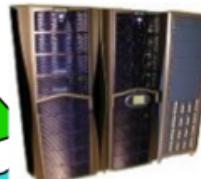
COSMO NWP-Applications



MeteoSwiss:
Cray XT4: COSMO-7 and
COSMO-2 use 980+4 MPI-
Tasks on 246 out of 260 quad
core AMD nodes



ARPA-SIM (Bologna, Italy):
Linux-Intel x86-64 Cluster for
testing (uses 56 of 120 cores)



USAM (Rome, Italy):
HP Linux Cluster
XEON biproc quadcore
System in preparation



ARPA-SIM (Bologna, Italy):
IBM pwr5: up to 160 of 512
nodes at CINECA

COSMO-LEPS (at ECMWF):
running on ECMWF pwr6 as
member-state time-critical
application

HNMS (Athens, Greece):
IBM pwr4: 120 of 256 nodes



COSMO CPU Usage (CSCS)

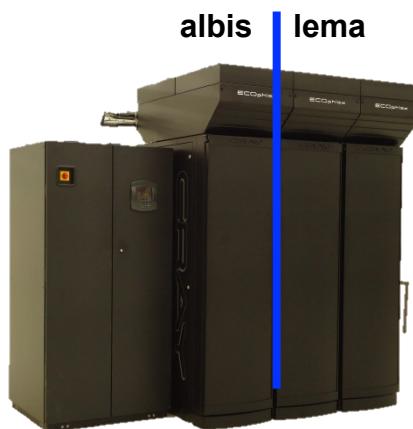
- CPU allocated to ETH groups for COSMO on Cray XE6



Peak 402 TFlops

**> 15'000'000 CPUh per year
(over the period 2010-2012)**

- MeteoSwiss currently has a dedicated Cray XE6



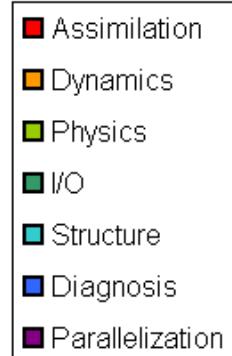
Peak 48 TFlops

~ 15'000'000 CPUh per year

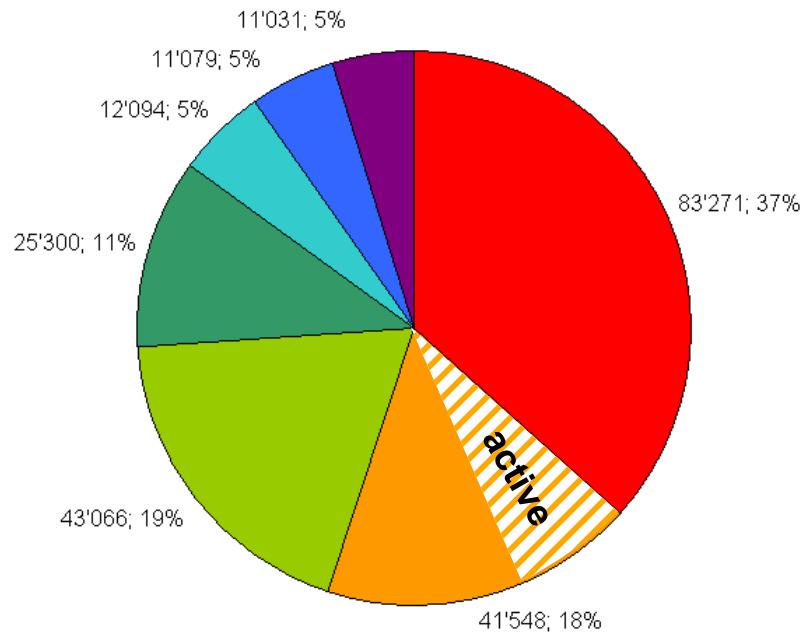


Lines vs. Runtime

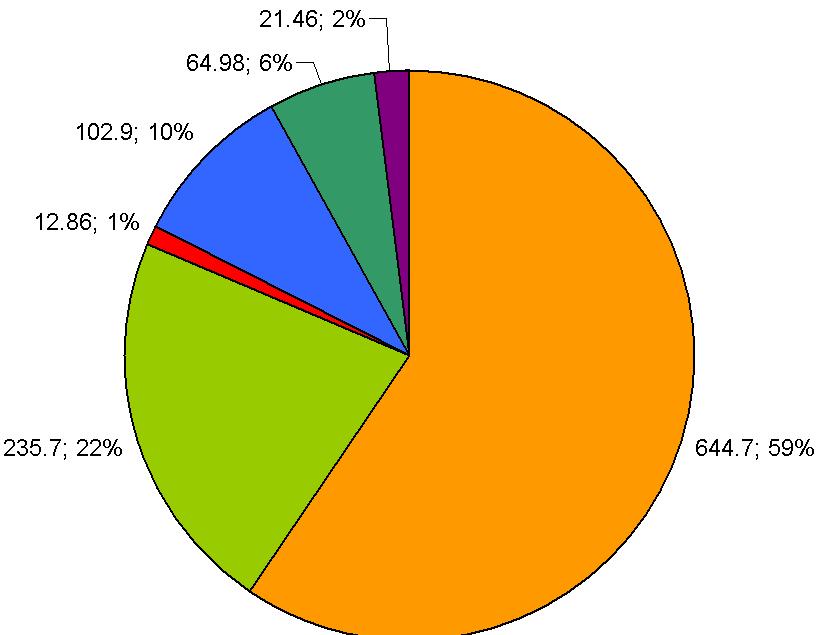
- 250'000 lines of Fortran 90 code



% Code Lines

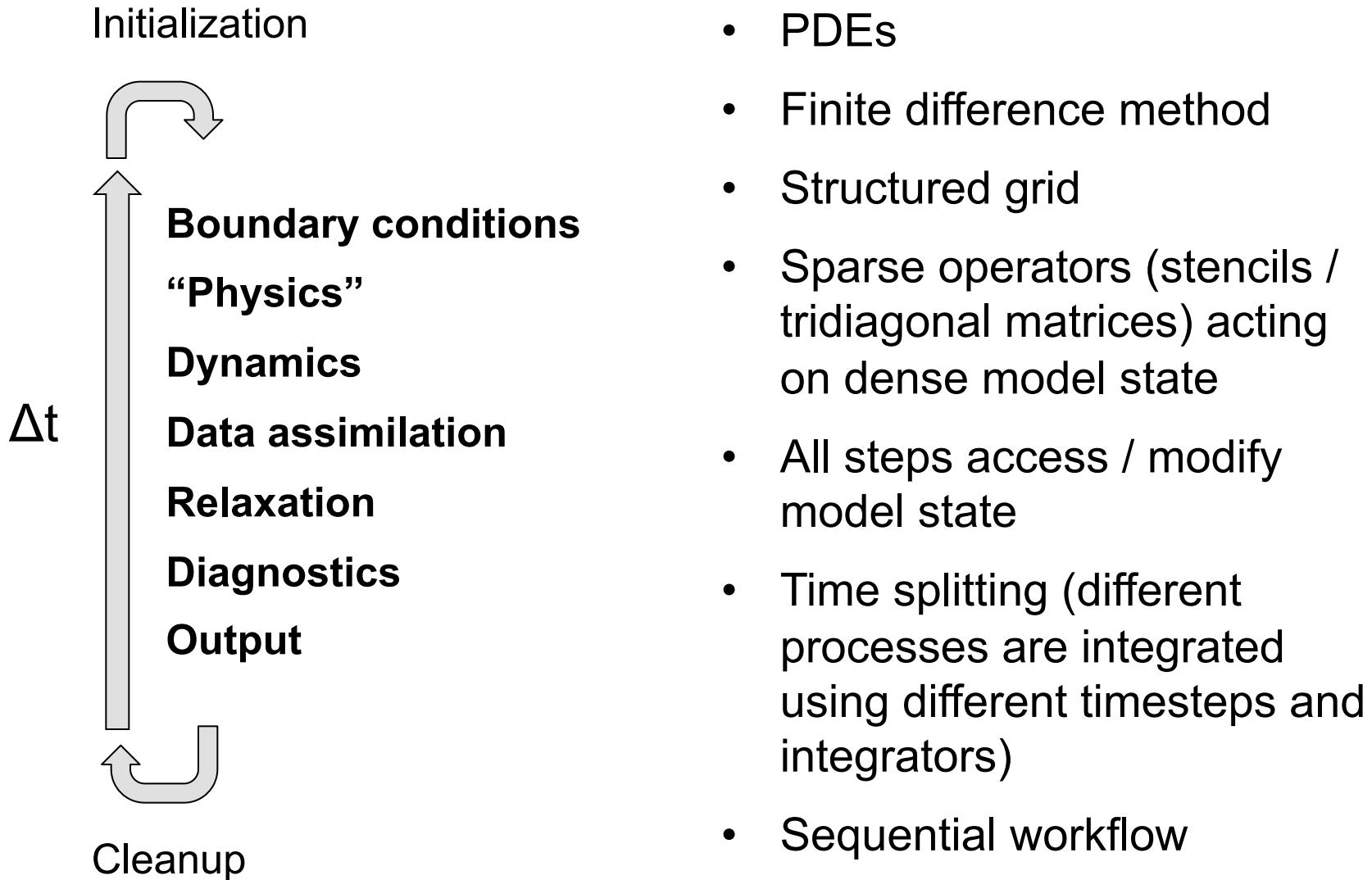


% Runtime





COSMO Workflow





Stencils

- “Dynamics” code (niter = 48, nwork = 4096000)

```
do j = 1, niter
  do i = 1, nwork
    c(i) = a(i) + b(i) * ( a(i+1) - 2.0d0*a(i) + a(i-1) )
  end do
end do
```

$$\frac{\partial a}{\partial t} = k \frac{\partial^2 a}{\partial x^2}$$

Machine	Cray XT4 2008	Cray XT5 2009	Cray XE6 2010	Cray XK6 2011
# cores	4	6	12	16
Single core	0.80 s	0.84 s	0.63 s	0.65 s
All cores	0.56 s	0.46 s	0.18 s	0.16 s
Speedup	1.4	1.8	3.4	4.0

- Code does not scale with the number of cores!



Code Analysis

- How are the two examples different?

“Physics”

```
do j = 1, niter
  do i = 1, nwork
    c(i) = a(i)*b(i) + sin(b(i)) * log(a(i))
    end do
  end do
```

3 memory accesses
136 FLOPs
→ compute bound

“Dynamics”

```
do j = 1, niter
  do i = 1, nwork
    c(i) = a(i) + b(i) * ( a(i+1) - 2.0d0*a(i) + a(i-1) )
    end do
  end do
```

3 memory accesses
5 FLOPs
→ memory bound

- Arithmetic throughput is a **per core resource** and scales with the number of cores
- Memory bandwidth is **shared resource** between cores on a single socket



Algorithmic motifs

- **Arithmetic Intensity (= FLOPs per memory access)**
 - High arithmetic intensity → processor bound → high %peak
 - Low arithmetic intensity → memory bound → low %peak



**COSMO dynamical core
(stencils on structured grid)**

Top500 (Linpack)
Focus of HPC system design

- Example: COSMO runs with ~4% peak on Cray XE6



Summary: Code Analysis

- COSMO is part of the class of finite difference methods on structured grids
 - Strongly memory bandwidth bound
 - Low %peak
- Two main algorithmic motifs
 - Stencils
 - Tridiagonal solve
- **Exploiting data locality is critical!**
- **How to rewrite the code in order to be ready for (different) emerging / future hardware architectures?**



Code example

- Solution of tridiagonal linear system

$$\frac{\partial s}{\partial t} = \frac{\partial}{\partial z} \left(\tilde{k} \frac{\partial s}{\partial z} \right) \rightarrow \begin{bmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & \ddots & \ddots & c_{n-1} & \\ 0 & & a_n & b_n & \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{bmatrix}$$

- Typical for vertically implicit schemes (advection, diffusion, radiation, ...)
- Abundant and performance relevant in many dynamical cores (e.g. COSMO)



COSMO Version

```
! solve tridiag(a,b,c) * x = d
!
! pre-computation
...
do j = jstart, jend
    !
    ! forward elimination
    do k = nk, 2, -1
        do i = istart, iend
            !CDIR ON ADB(d)
                d(i,j,k) = ( d(i,j,k) - d(i,j,k+1) * c(i,j,k) ) * b(i,j,k)
            end do
        end do
    !
    ! back substitution
    do k = 1, nk-1
        do i = istart, iend
            !CDIR ON ADB(x)
                x(i,j,k+1) = a(i,j,k+1) * x(i,j,k) + d(i,j,k+1)
            end do
        end do
    end do
```

- Algorithm: TDMA
- Language: Fortran
- Grid: Structured
- Data layout: (i,j,k)
- Parallelization: MPI in (i,j)
- Loop order: (jki)
- Blocking: (j)
- Vectorization: (i)
- Directives: NEC
- ...



Optimized CPU Version

```
! solve tridiag(a,b,c) * x = d

!$OMP PARALLEL DO SHARED(x) PRIVATE(a,b,c,d) COLLAPSE(2)
do ib = 1, nblock_i
do jb = 1, nblock_j

  ! pre-computation
  ...

  do i = istart_block, iend_block
  do j = jstart_block, jend_block

    ! forward elimination
    do k = nk, 2, -1
      d(k,j,i) = ( d(k,j,i) - d(k+1,j,i) * c(k,j,i) )
    end do

    ! back substitution
    do k = 1, nk-1
      x(k+1,j,i) = a(k+1,j,i) * x(k,j,i) + d(k+1,j,i)
    end do

  end do
end do

end do
end do
!$OMP END PARALLEL DO
```

- Algorithm: TDMA
- Language: Fortran
- Grid: Structured
- Data layout: (k,j,i)
- Parallelization: MPI and OpenMP in (i,j)
- Loop order: (ijk)
- Blocking: (i,j)
- No vectorization
- Directives: OpenMP
- ...



Optimized GPU Version

```
! solve tridiag(a,b,c) * x = d
!$ACC DATA COPYIN(a,b,c,d) COPYOUT(x)
!$ACC KERNELS LOOP, GANG(32), WORKER(8)
do i = istart, iend
do j = jstart, jend

    ! pre-computation
    ...

    ! forward elimination
    do k = nk, 2, -1
        d(i,j,k) = ( d(i,j,k) - d(i,j,k+1) * c(i,j),
    end do

    ! back substitution
    do k = 1, nk-1
        x(i,j,k+1) = a(i,j,k+1) * x(i,j,k) + d(i,j,k+1)
    end do

end do
end do
!$OMP END KERNELS LOOPS
!$ACC END DATA
```

- Algorithm: TDMA
- Language: Fortran
- Grid: Structured
- Data layout: (i,j,k)
- Parallelization: MPI (i,j) and Threads (i,j)
- Loop order: (ijk)
- No Blocking
- Vectorization: SIMD Threads (i,j)
- Directives: OpenACC
- ...



Learnings

- **Code is a mix** of mathematical model, numerical discretization, solution algorithm, and implementation details → **separation of concerns?**
- Optimizations are **hardware dependent** and **increase code complexity**
- Consequences
 - Hard to achieve performance portability with a single source code!
 - Hard to understand and modify
 - Hard to validate and debug
 - Hard to re-use
 - Hard to attract good programmers



Solutions?

- Good compromise (if it exists!)
- Several efficient source codes
- Separate model and algorithm from hardware specific implementation and optimization

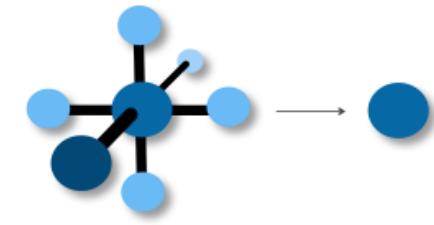
Challenging computer science problem!



Stencil Code Analysis

- Operator has two main components:
 - Loop-logic defining the stencil application domain and order
 - Stencil defining the operator to be applied

```
do k = kstart, kend
    do j = jstart, jend
        do i = istart, iend
            lap(i, j, k) = -4.0_ir * data(i, j, k) +  &
                data(i+1, j , k) + data(i-1, j , k) +  &
                data(i , j+1, k) + data(i , j-1, k)
        end do
    end do
end do
```



Laplacian

- No straight-forward API similar to matrix operations
 - Platform specific details (e.g. loop-logic and storage order) implemented by a library
 - Operator specific stencil needs to be implemented by the user



Stencil Library (C++/DSEL)

- C++ allows to define a **domain specific embedded language (DSEL)** using template meta programming
- At compile-time the DSEL statements are expanded into a sequence of operations
- Library contains hardware dependent code fragments which are assembled into loop-logic
- The backend of the compiler “sees” loops as if they were hand-coded
- We use this approach to generate the hardware dependent parts of the code



Stencil Library (User Code)

```
enum { data, lap } ;

template<typename TEnv>
struct Laplace
{
    STENCIL_STAGE(Tenv)
    STAGE_PARAMETER(FullDomain, data)
    STAGE_PARAMETER(FullDomain, lap)

    static void Do()
    {
        lap::Center() =
            -4.0 * data::Center() +
            data::At(iplus1) +
            data::At(iminus1) +
            data::At(jplus1) +
            data::At(jminus1);
    }
};
```

```
IJKRealField lapfield, datafield;
Stencil stencil;

StencilCompiler::Build(
    pack_parameters(
        Param<lap, cInOut>(lapfield),
        Param<data, cIn>(datafield)
    ),
    concatenate_sweeps(
        define_sweep<KLoopFullDomain>(
            define_stages(
                StencilStage<Laplace, IJRangeComplete>()
            )
        )
    )
);

stencil.Apply();
```



Stencil Library (User Code)

Stencil

```
enum { data, lap } ;  
  
template<typename TEnv>  
struct Laplace  
{  
    STENCIL_STAGE(Tenv)  
    STAGE_PARAMETER(FullDomain, data)  
    STAGE_PARAMETER(FullDomain, lap)  
  
    static void Do()  
    {  
        lap::Center() =  
            -4.0 * data::Center() +  
            data::At(iplus1) +  
            data::At(iminus1) +  
            data::At(jplus1) +  
            data::At(jminus1);  
    }  
};
```

Loop-logic

```
IJKRealField lapfield, datafield;  
Stencil stencil;  
  
StencilCompiler::Build(  
    pack_parameters(  
        Param<lap, cInOut>(lapfield),  
        Param<data, cIn>(datafield)  
    ),  
    concatenate_sweeps(  
        define_sweep<KLoopFullDomain>(  
            define_stages(  
                StencilStage<Laplace, IJRangeComplete>()  
            )  
        )  
    )  
);  
  
stencil.Apply();
```



Code Expansion

- “Expanded” code (for x86 CPU backend)

```
#pragma omp parallel for
for(int block=0; block < num0fBlocks; ++block)
{
    context.MoveToBlock(block);

    for(int i=iBlockStart; i < iBlockEnd; ++i) {
        for(int j=jBlockStart; j < jBlockEnd; ++j) {

            context.MoveTo(i, j, kstart);

            for(int k=kstart; k < kend; ++k) {
                Laplacian::Do(context, FullDomain);
                context.Advance<0,0,1>();
            }
        }
    }
}
```



Stencil Library Features

- Loop definitions
- Boundary conditions
- Functions
- Buffers
- Software managed caching
- Parallelization
- ...



Loop Definition

```
concatenate_sweeps(  
    define_sweep<cKIncrement>(  
        define_stages(  
            StencilStage<Operator1,  
                IJRange<-1,1,-1,1>,  
                KRangeFullDomain>(),  
            StencilStage<Operator2,  
                IJRange<0,0,0,0>,  
                KRangeFullDomain>()  
        )  
,  
    define_sweep<cKDecrement>(  
        define_stages(  
            StencilStage<Operator3,  
                IJRange<0,0,0,0>,  
                KRangeFullDomain>()  
        )  
)  
)
```

No one-to-one correspondence!
This is just for illustration.

```
DO k = 1, ke  
    DO j = jstart-1, jend+1  
        DO i = istart-1, iend+1  
            ! Operator1  
        ENDDO  
    ENDDO  
    DO j = jstart, jend  
        DO i = istart, iend  
            ! Operator2  
        ENDDO  
    ENDDO  
    DO k = ke, 1, -1  
        DO j = jstart, jend  
            DO i = istart, iend  
                ! Operator3  
            ENDDO  
        ENDDO  
    ENDDO
```



Functions

- Operators can be defined as functions (e.g. Laplacian, ...)
- Functions can be nested

```
static void Do() {  
    res::Center() = Call<lap>::With( Call<lap>::With( data::Center() ) );  
}
```

- Code is closer discretized mathematical model

```
utens::Center() +=  
    Call<Average>::With( mass2u,  
        fc::Center() * Call<Average>::With( v2mass, v::Center() ) ) );
```

```
z_fv_north = fc(i,j) * ( v(i,j,k,nn) + v(i+1,j,k,nn) )  
z_fv_south = fc(i,j-1) * ( v(i,j-1,k,nn) + v(i+1,j-1,k,nn) )  
    zfq = 0.25_ireals * ( z_fv_north + z_fv_south )  
utens(i,j,k) = utens(i,j,k) + zfq
```



Buffers

- Buffers (of optimal size and placement) are used for passing information between operators

```
define_buffers(  
    StencilBuffer<a, Real, IJKColumn>(),  
)  
concatenate_sweeps(  
    define_sweep<cKIncrement>(  
        define_stages(  
            StencilStage<Operator1,  
                IJRange<-1,1,-1,1>,  
                KRangeFullDomain>(),  
            StencilStage<Operator2,  
                IJRange<0,0,0,0>,  
                KRangeFullDomain>()  
        )  
    )  
)
```

```
DO k = 1, ke  
    DO j = jstart-1, jend+1  
        DO i = istart-1, iend+1  
            a(i,j,k) = Operator1  
        ENDDO  
    ENDDO  
    DO j = jstart, jend  
        DO i = istart, iend  
            ! Operator2 = f(a)  
        ENDDO  
    ENDDO  
ENDDO
```

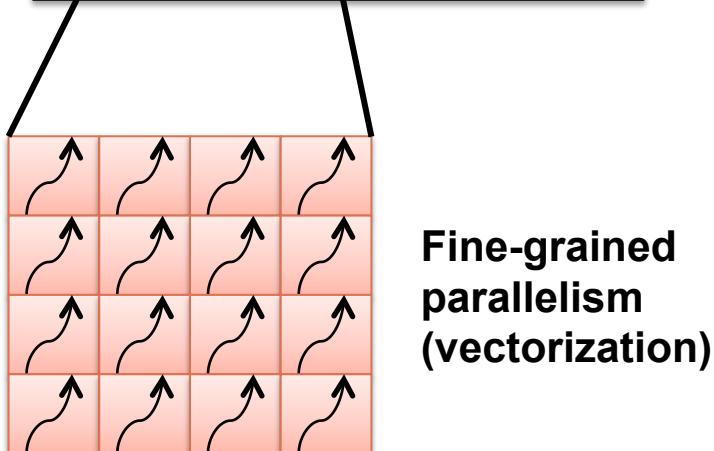
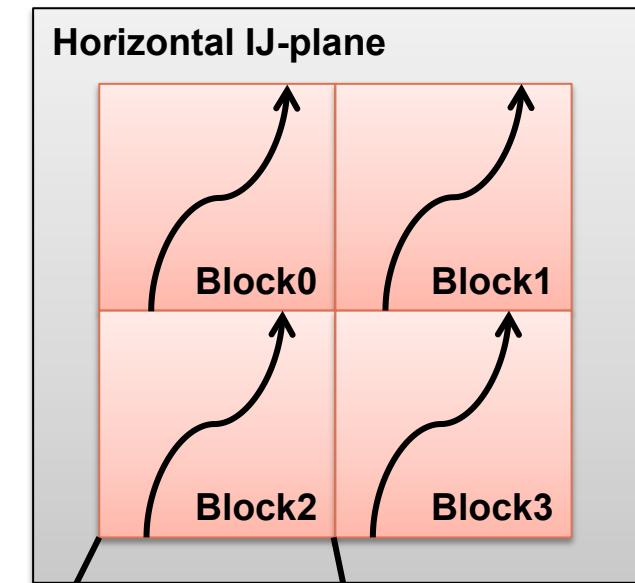


Parallelization Model

- Multi-node parallelization (MPI)
- Shared memory parallelization
 - Support for 2 levels of parallelism
- Coarse-grained parallelism
 - Split domain into blocks
 - Distribute blocks to cores
 - No synchronization & consistency required
- Fine-grained parallelism
 - Update block on a single core
 - Lightweight threads / vectors
 - Synchronization & consistency required

Similar to CUDA programming model
(is a good match for other platforms as well)

Coarse-grained parallelism (multicore)

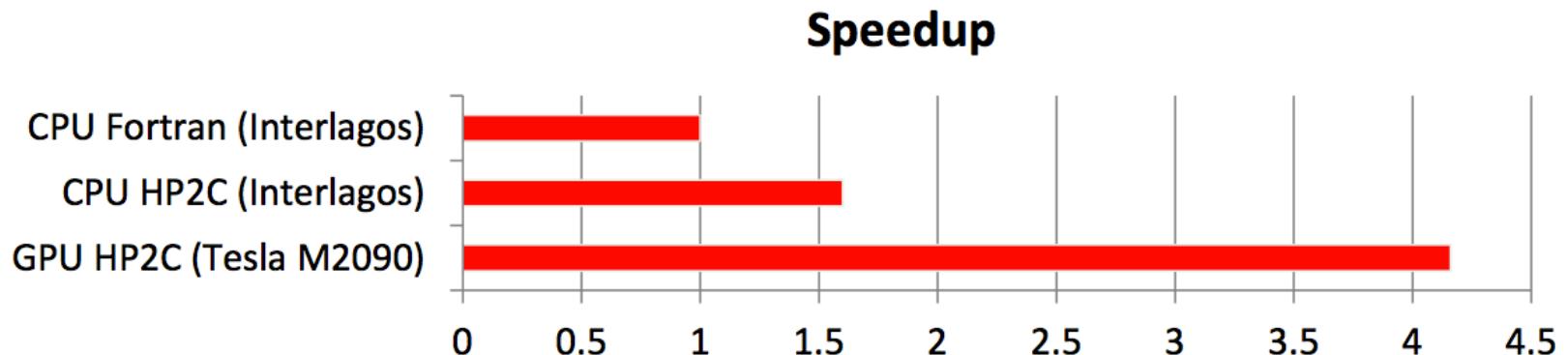




Dynamics Performance

- Test domain: 128 x 128 x 60 on a single CPU / GPU
- **CPU** (OpenMP, kji-storage)
 - Factor 1.6x – 1.8x faster than original COSMO
 - No explicit use of vector instructions (up to 30% improvement)
- **GPU** (CUDA, ijk-storage)
 - CPU vs. GPU is a factor 2.8x faster (Interlagos vs. X2090)
 - Ongoing performance optimization

A single switch in order to compile for GPU





Pros and Cons

Pros

- Better separation of algorithm and implementation strategy
- Performance and portability
- Single source code
- Library suggests / enforces coding conventions and styles
- Flexibility to extend to other architectures
- Based on standard programming language and compiler
- Unit testing

Cons

- Big change with respect to original code
- Boilerplate code
- Library provides fixed set of features
- Adding new architectures requires deep understanding



Demonstrator

- Prototype implementation of the COSMO production suite of MeteoSwiss making aggressive use of GPU technology



1 cabinet Cray XE6

144 CPUs
(1728 cores)

8 GPUs
(19968 cores)



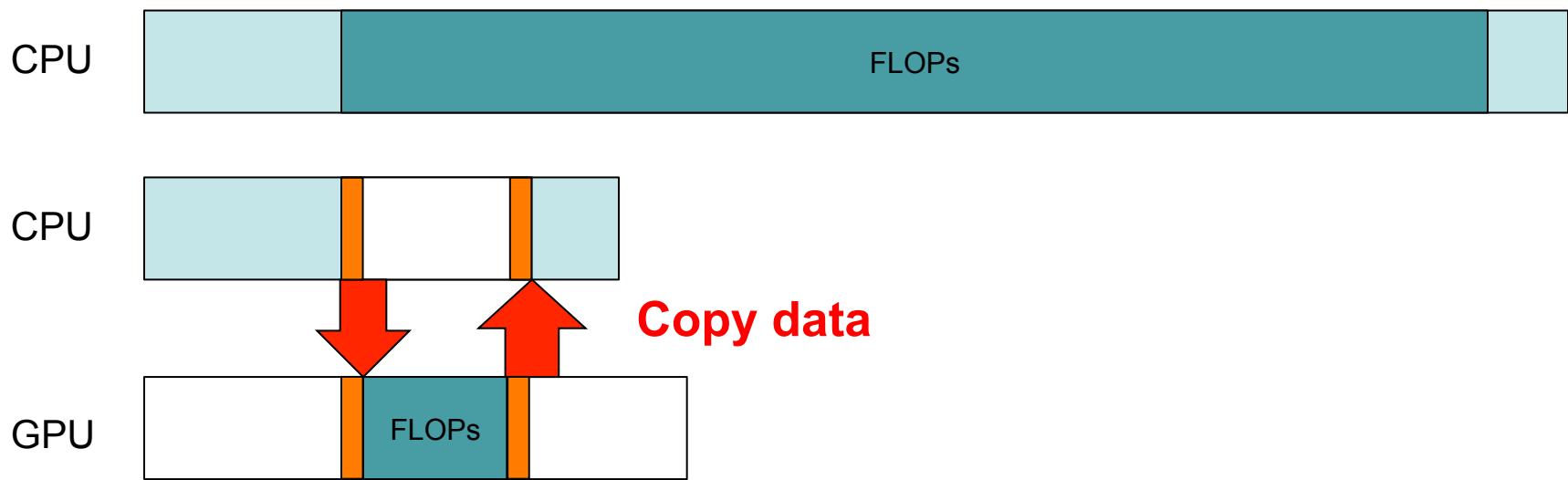
Tyan Server FT77A

- Same time-to-solution on substantially cheaper hardware:
Factor ~3x in price, factor ~9x in power consumption



Accelerator approach

- Leverage high peak performance of GPU
- CPU and GPU have different memories



- This approach does not work for COSMO!
 - Low FLOPs count per load/store (stencils!)
 - Transfer of data on each timestep too expensive



Why does this not work for COSMO?

- Low FLOP count per load/store (stencils!)
- Transfer of data on each timestep too expensive

* Part	Time/ Δt
Dynamics	172 ms
Physics	36 ms
Total	253 ms

vs

§ Transfer of ten prognostic variables
118 ms

All code which touches the prognostic variables within timestep has to be ported



Full GPU Port

HP2C COSMO / HP2C OPCODE follow a common goal...

GPU-implementation of “full” timestep of COSMO

Aim for...

- Completeness (i.e. full COSMO model)
- Performance (i.e. lower time-to-solution)
- Portability / Maintainability (i.e. no hacks)
- Durability (i.e. knowledge transfer and documentation)
- **Time/resource constraints lead to compromises**

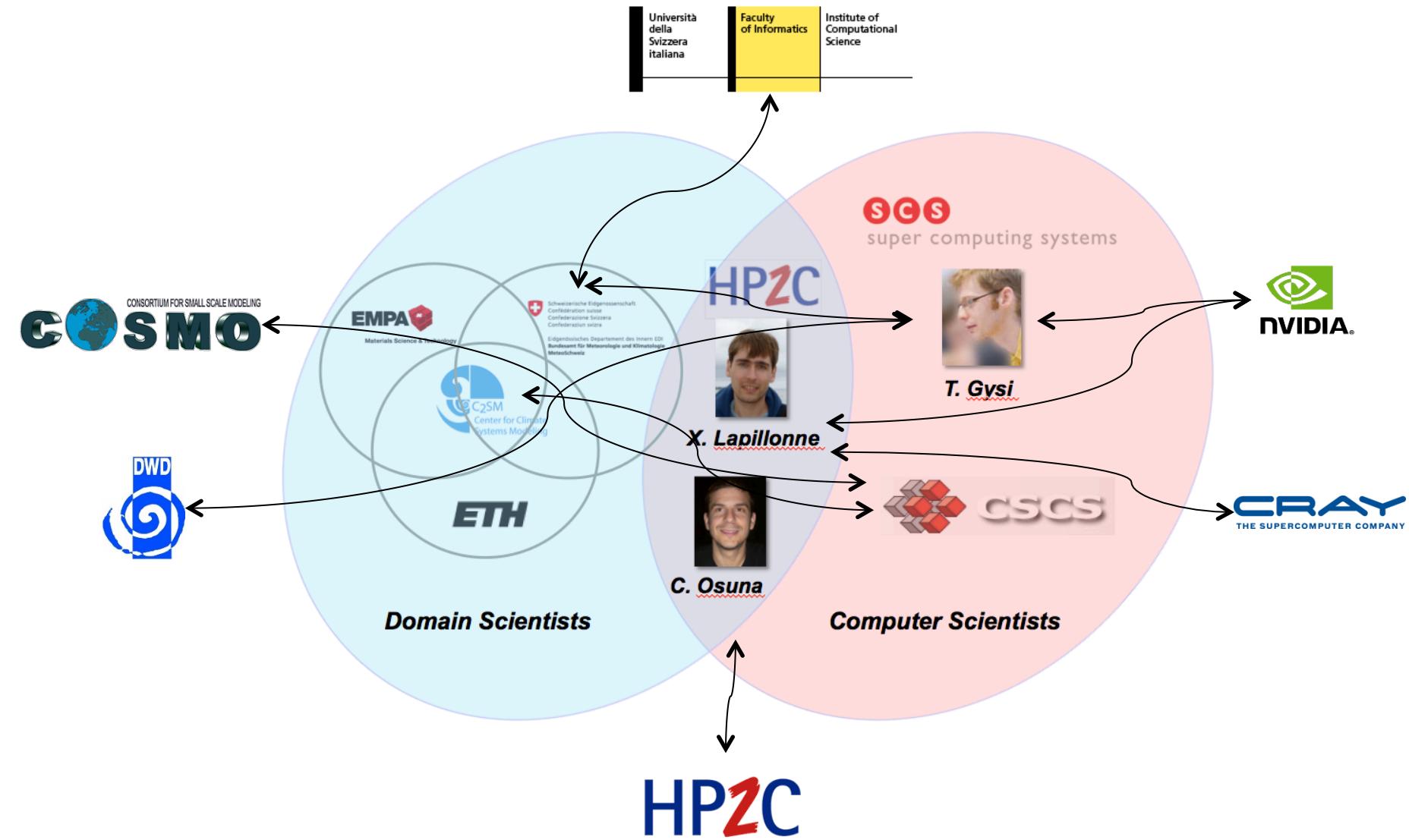


Conclusions

- Changing hardware architectures / programming paradigms require to (continually) adapt our codes
- Model codes are growing in length and complexity
- No consensus has emerged to deliver both high performance with high programmer productivity
- Stencil library (DSEL) can help by...
 - freeing model developer from implementation details
 - increasing programmer productivity
 - retaining efficiency with single source code
 - making our codes more reusable and adaptable
 - joining efforts

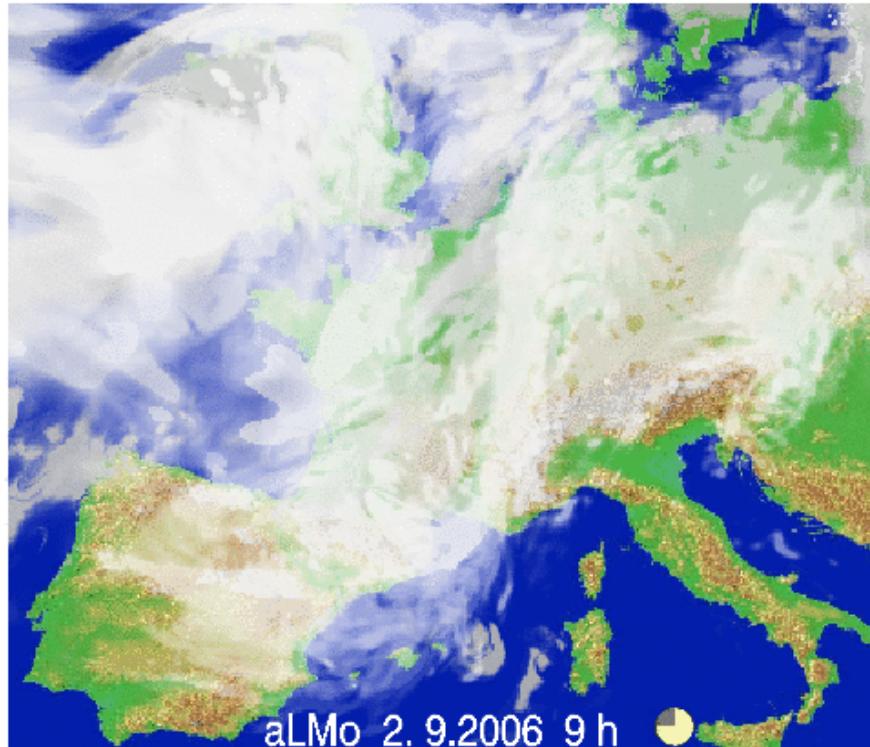


Key ingredient!

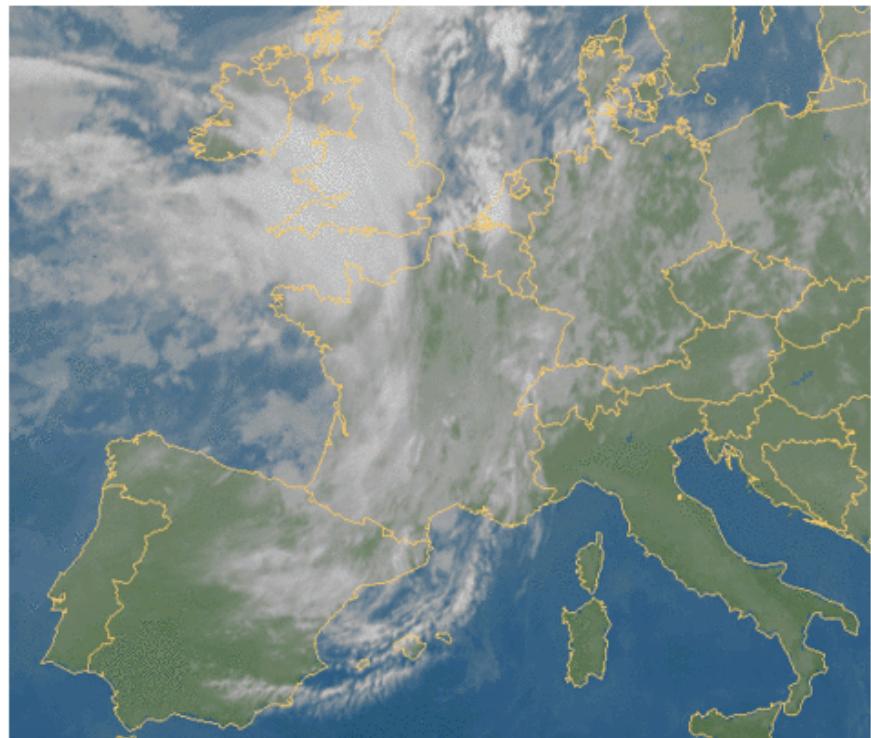




Thank you!



COSMO running on GPUs



Satellite image