CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A Review of The Challenges and Results of Refactoring the Climate Code COSMO for Hybrid Cray HPC Systems

Ben Cumming[1], Carlos Osuna[2], Tobias Gysi[3], Mauro Bianco[1], Xavier Lapillonne[2], Oliver Fuhrer[2] and Thomas Schulthess[1]

[1]CSCS, [2]ETHZ, [3]SCS and [4]MeteoSwiss

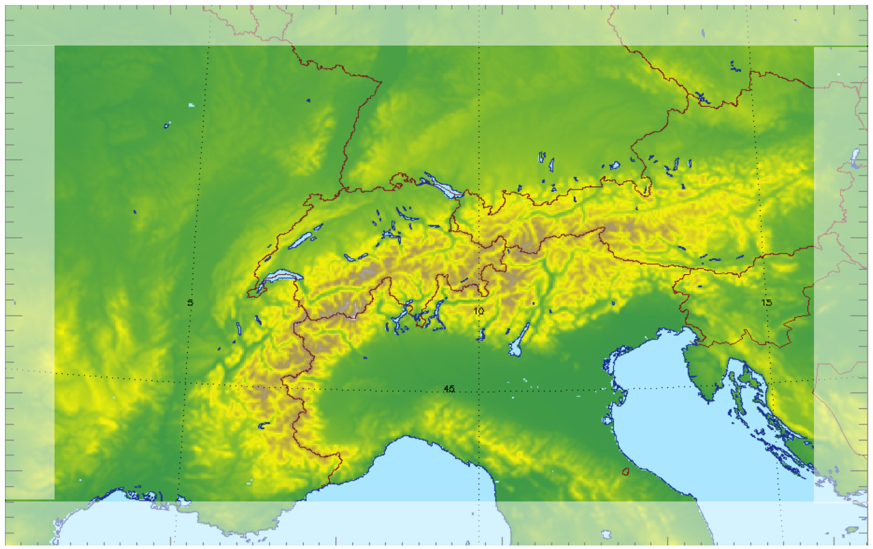
CUG 2013

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

What is COSMO?

- The consortium for small-scale modeling (COSMO) is a limited-area, non-hydrostatic atmospheric model.
- Used for both weather forecasting (MeteoSwiss, DWD, and others) and climate modeling (ETHZ, KIT, and others)
- Stencils computation on a regular 3D grid.



2km model of Alps: 520x350x60 grid

Current Usage

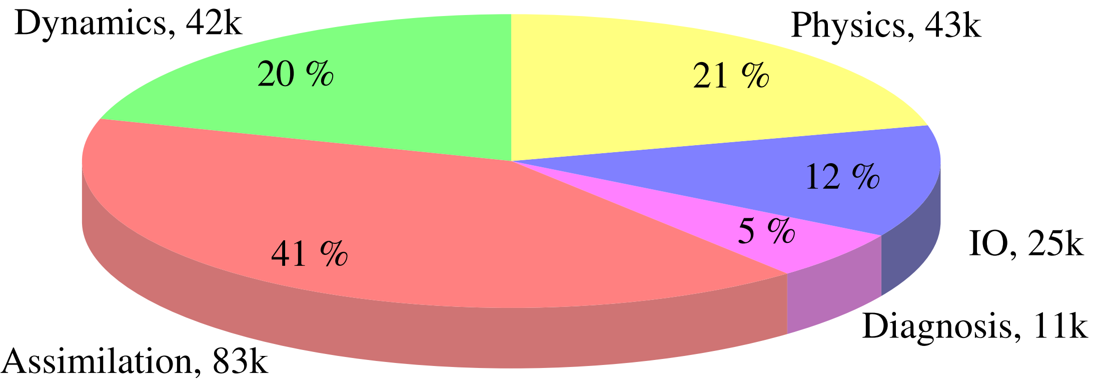
- The current version of COSMO is heavily-optimized for NEC vector machines
  - Poor performance on x86 machines primarily due to poor cache reuse.
- Many users use x86-based machines (Cray XE5 in use at Meteoswiss)
  - The German weather service (DWD) is acquiring an XC30
  - The main trunk will be "tuned" for Sandy/Ivy Bridge.
- Performance portability is a problem with this model
  - Code is modified for optimal performance on a new architecture
  - Optimal support for one architecture at a time
  - Users on other architectures get sub-optimal performance

An Alternative approach

- There has been an ambitious effort in Switzerland as part of the HP2C program, with strong collaboration between MeteoSwiss, ETHZ and CSCS, to port COSMO to better support hybrid multi-core and many-core systems.
- An important aim was to develop a performance-portable code from a single source code.

Code Overview

- COSMO has ~250,000 lines of Fortran 90 code.
- Flat MPI used for parallelization.

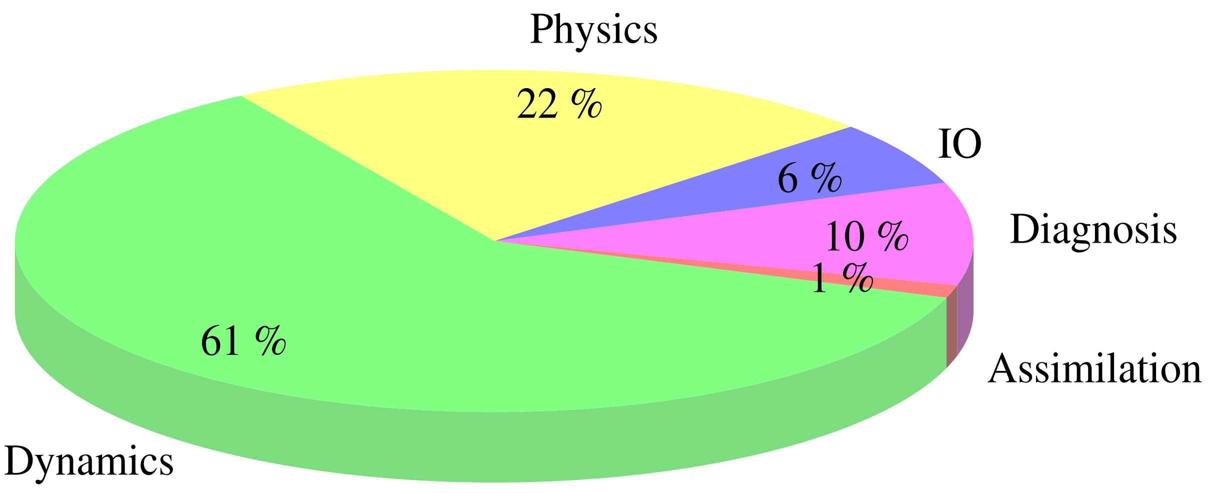


Of key interest when porting:

- Dynamics : 42k lines
  - Solution of atmospheric flow equations.
  - One main developer at DWD
  - Updated infrequently
- Physics : 43k lines
  - Parameterization of processes not modeled in dynamics (e.g. radiation)
  - Many contributors and frequent updates

Runtime Breakdown

- The dynamical core and physical parameterization account for over 80% of the runtime
  - These will be the main focus of hardware-specific optimizations

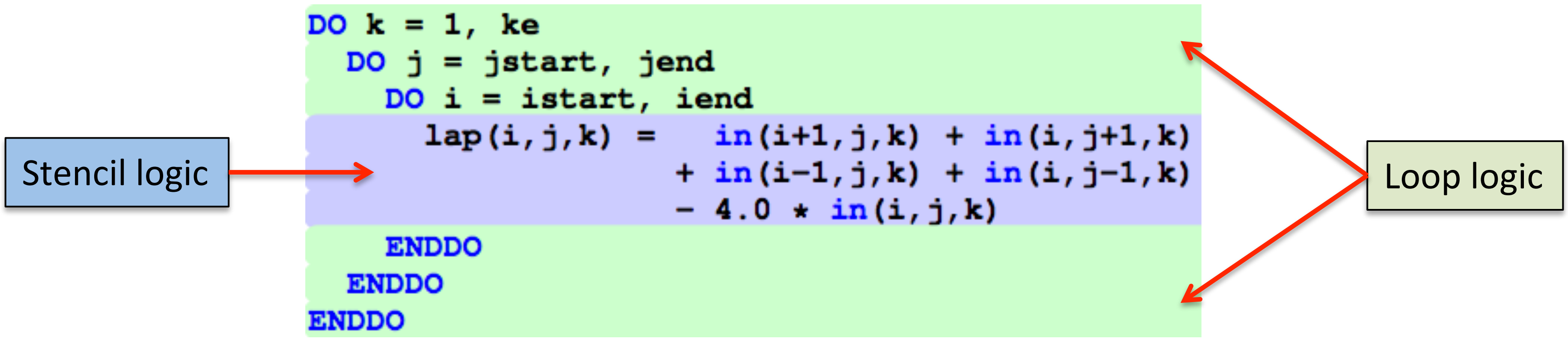


Porting Strategies

- Different approaches are used for each part of COSMO:
  1. A domain-specific embedded language (DSEL) for stencil operations was written in C++, and the dynamics fully ported to the DSEL.
     - Easier to collaborate with small developer base
     - Potential for greatest performance gain
  2. OpenACC directives were added to the physical parameterization and assimilation.
     - Less intrusive than a complete rewrite
     - Easier to minimize disruption to users, and keep up with frequent changes to code.

Porting The Dynamics With a DSEL

- The DSEL, called “The Stencil Library”, is a domain specific language embedded in C++.
- All stencils in COSMO can be expressed in terms of loop logic and stencil logic.
  - The DSEL separates the loop logic and stencil logic.
- At compile time either a hardware-specific backend generates code with loop ordering, block sizes and buffering/caching optimized for the target hardware (currently we have OpenMP and CUDA backends)



```
DO k = 1, ke
  DO j = jstart, jend
    DO i = istart, iend
      lap(i,j,k) =   in(i+1,j,k) + in(i,j+1,k)
                   + in(i-1,j,k) + in(i,j-1,k)
                   - 4.0 * in(i,j,k)
    ENDDO
  ENDDO
ENDDO
```

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

DSEL Walkthrough : Loop Logic

```
struct Laplace{
  static void Do(Context ctx){
    ctx[lap(center)] = ctx[in(iplus1)] + ctx[in(iminus1)]
                      +ctx[in(jplus1)] + ctx[in(jminus1)]
                      -4.0 * ctx[in(center)];
    }
}
```

- The stencil logic is expressed in a C++ functor
- The functor is pure type information

- Loop logic is specified when the stencil is compiled
- Loop bounds and direction are passed as template parameters, along with the stencil functor.

```
Stencil stencil;
StencilCompiler::Build(  // compile the stencil
  stencil,
  ...
  define_sweep<cKIncrement>(
    define_stages(
      StencilStage< Laplace,
                    IJRange<0,0,0,0>,
                    KRange<0,0> >()
    )
  ),
  ...
);
stencil.Apply();      // apply the stencil
```

DSEL Pros and Cons

Pros

- Performance portability
- Single source code for multiple hardware targets
- No hardware-specific language in user code
- Users with little/no C++ experience have been able to use the DSEL quite quickly (after a week long workshop)

Cons

- Significant departure from what users are used to
- There can be a lot of "boiler plate" code as a result of using C++
- Adding support for new hardware back ends requires highly-specialized knowledge and takes time

Porting The Physics with OpenACC Directives

- Porting with OpenACC was a two-step process:
  1. Add data directives (requires detailed analysis of the code) and add loop directives to the loops.
  2. Tune the slower loops.
- For many loops step 1 was sufficient, however some key stencils benefitted from extensive tuning.

Tuning OpenACC

Move the k-loop into the inner loop

- One thread per column in the k direction performing a sequential loop

Copy blocks of 3D field (nx,ny,nz) into 2D blocks (nproma,nz)

- Makes it easier to coalesce loads and stores when nx is not a multiple of 32
- Gives greater flexibility in choosing number of threadblocks

```
!vertical loop
!$acc parallel vector_length(N)
do k=2,Nz
   !$acc loop
   do j=1,Ny
      !$acc loop vector
      do i=1,Nx
         c2=c1(i,j,k)*a(i,j,k-1)
         a(i,j,k)=c2*a(i,j,k-1)
      end do
   end do
end do
!$acc end parallel
!$acc end data
```

```
!$acc data present(a,c1)
!$acc parallel loop vector_length(N)
do ip=1,nproma
   !vertical loop
   do k=2,Nz
      !work 1
      c2=c1(ip,k)*a(ip,k-1)
      !work 2
      a(ip,k)=c2*a(ip,k-1)
   end do
end do
!$acc end parallel loop
!$acc end data
```

OpenACC & OpenMP: Portability?

- A key aim of this work is to have performance portable code
  - Important for future maintenance.
  - Having multiple versions of loops for different architectures would hinder adoption by the user community
- The focus to date has been on OpenACC for GPU support
- Ideally we should get good performance on the same code by adding OpenMP directives…

Loop Order Example

- The key stencil loops have dependencies in the k-direction: the input at level k+1 requires the computed value at level k.
- The optimal loop structure for this is different on x86 (OpenMP) and GPU (OpenACC)

Vectorizable inner loop optimal for OpenMP

```
!$omp parallel do
do k=2,Nz
  do ip=1,nproma
    c2=c1(ip,k)*a(ip,k-1)
    a(ip,k)=c2*a(ip,k-1)
  enddo
enddo
!$omp end parallel do
```

Sequential inner k-loop optimal with OpenACC

```
!$acc parallel loop vector_length(N)
do ip=1,nproma
  do k=2,Nz
    c2=c1(ip,k)*a(ip,k-1)
    a(ip,k)=c2*a(ip,k-1)
  enddo
enddo
!$acc end parallel loop
```

Communication

- A generic communication library for structured grids was written in C++.
  - Transparently handles fields stored on the GPU or host memory.
  - Uses direct GPU-to-GPU (G2G) communication in MPI (tested with MVAPIC2, OpenMPI and Cray MPI)
  - Breaks communication into send and wait phases so that communication and computation can be overlapped.
  - Arbitrary orientation (IJK, KIJ) storage order.
  - Called from the Fortran code via a C++ wrapper.

Integration

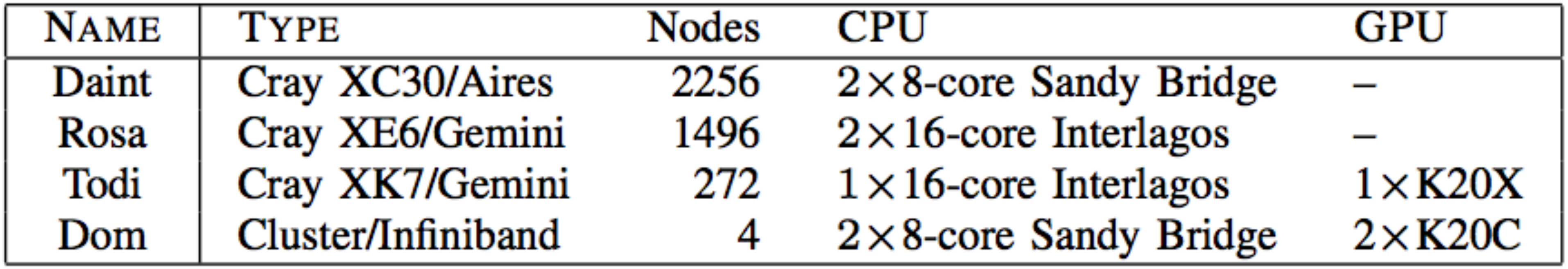
- Getting the complete application to gel with C++, CUDA, Fortran and OpenACC has been a challenge.
- The dynamical core and communication framework are compiled as a library with GNU C++ and NVCC compilers.
- The application is compiled with either Cray or PGI Fortran/OpenACC compilers.
- The OpenACC work has not been straightforward
  - OpenACC support from CCE and PGI is reasonably mature now, but as early adopters a lot of bugs and missing features hindered progress
  - We still have open bugs with both compiler's OpenACC implementations (only fully validated GPU run with PGI 12.10)
  - The DSEL development and complete rewrite of Dycore was finished earlier than the OpenACC update of the Fortran code!
  - Currently the PGI compiler is the only OpenACC compiler supported on non-Cray systems, which is an impediment to community acceptance.
  - Still undecided how to get performance-portable code with OpenACC and OpenMP.

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Benchmarks

- Three Cray systems and an Infiniband GPU cluster at CSCS were used for benchmarks.
- Current focus is on correctness and validation of code
  - Results will improve significantly once we start tuning for performance

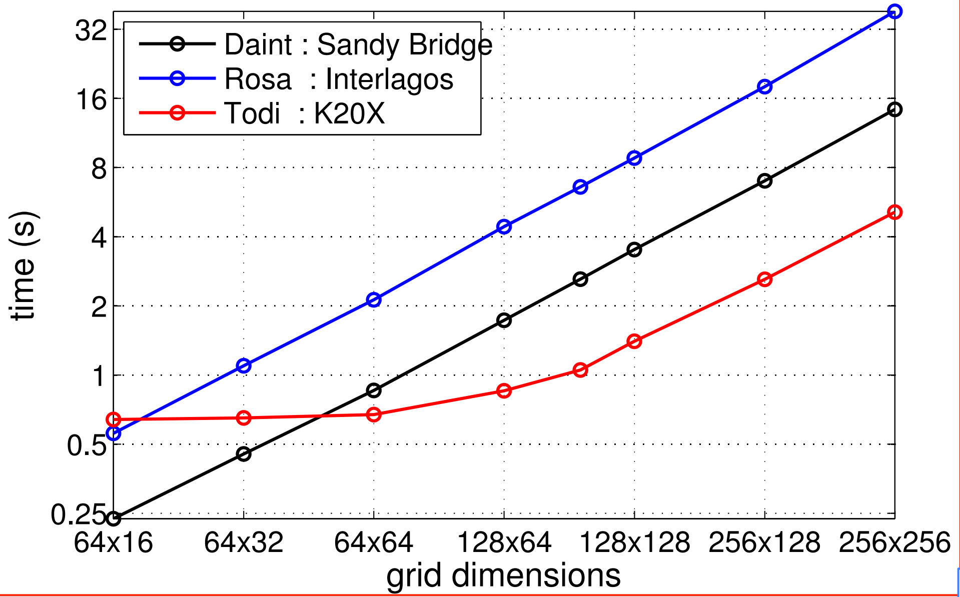


NAME	TYPE	Nodes	CPU	GPU
Daint	Cray XC30/Aires	2256	2×8-core Sandy Bridge	–
Rosa	Cray XE6/Gemini	1496	2×16-core Interlagos	–
Todi	Cray XK7/Gemini	272	1×16-core Interlagos	1×K20X
Dom	Cluster/Infiniband	4	2×8-core Sandy Bridge	2×K20C

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

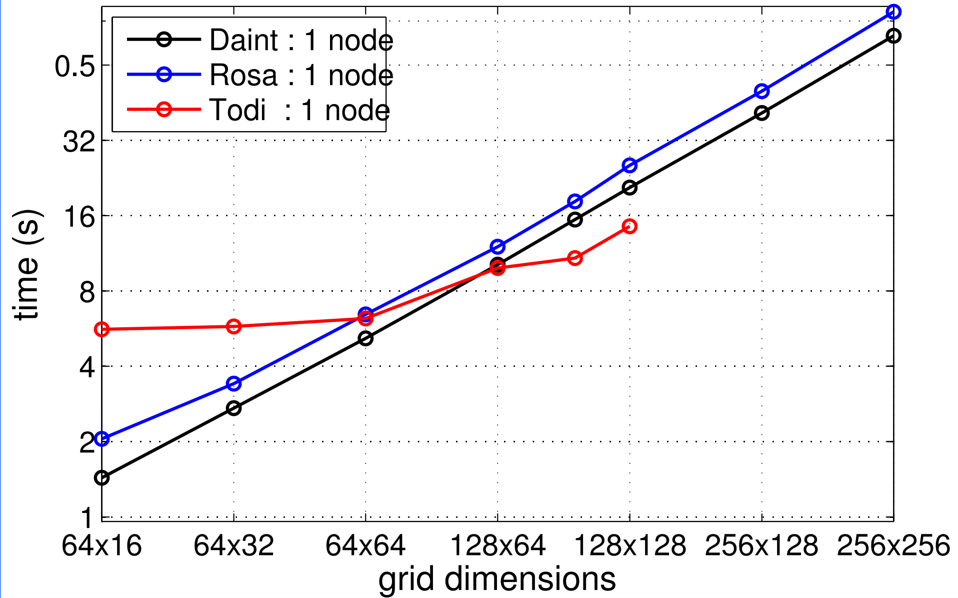
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Strong Scaling



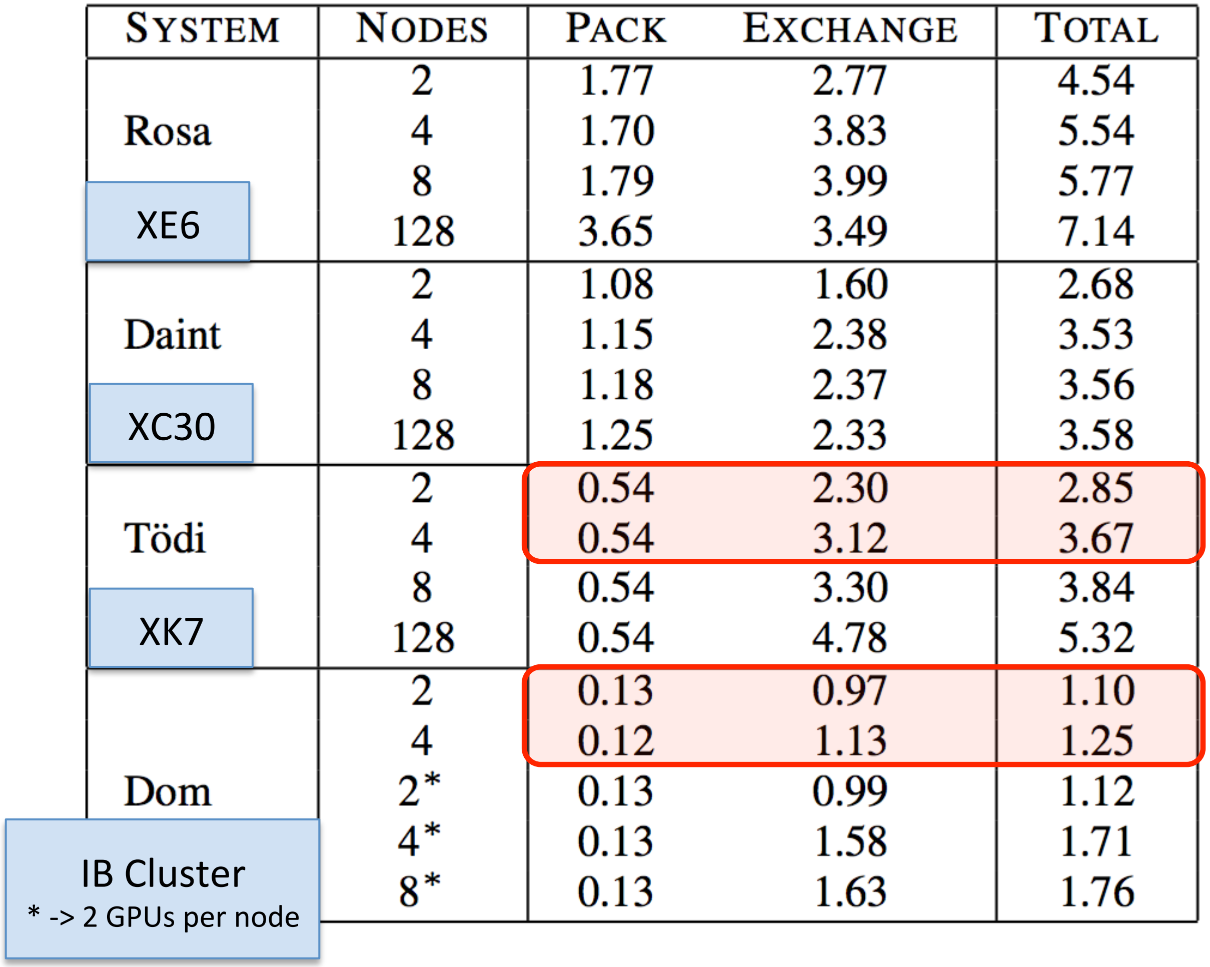
- 10 time steps of the dynamical core
- No MPI communication
- One SB socket with 8 threads on Daint (XC30)
- One IL die with 8 threads on Rosa (XE6)
- One K20X GPU on Todi (XK7)

- Average time spent in physics for per simulated hour of weather
- Exclude MPI communication
- 16 MPI processes on one node of Daint (XC30)
- 32 MPI processes on one node of Rosa (XE6)
- One K20X GPU on Todi (XK7)



Communication

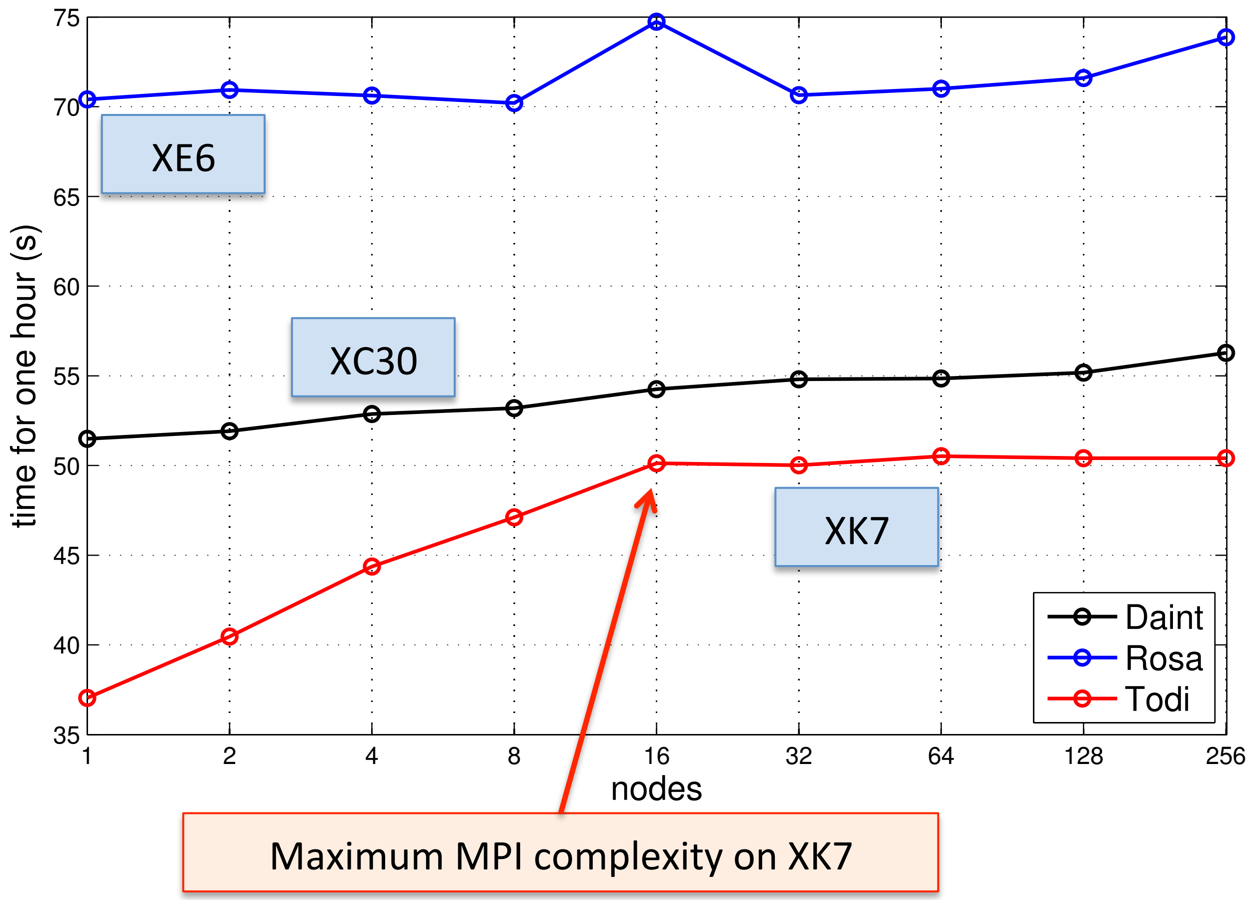
- Weak scaling test
- Each node has fields of dimension 128x128x60 with halo of width 3 in the horizontal boundaries
- Pure MPI use on XE6 and XC30 systems with 16 sub-domains per node.
- One sub-domain per node/GPU on XK7
- Perform exchange of 3 fields 50 times.



SYSTEM	NODES	PACK	EXCHANGE	TOTAL
Rosa	2	1.77	2.77	4.54
XE6	4	1.70	3.83	5.54
	8	1.79	3.99	5.77
	128	3.65	3.49	7.14
Daint	2	1.08	1.60	2.68
XC30	4	1.15	2.38	3.53
	8	1.18	2.37	3.56
	128	1.25	2.33	3.58
Tödi	2	0.54	2.30	2.85
XK7	4	0.54	3.12	3.67
	8	0.54	3.30	3.84
	128	0.54	4.78	5.32
Dom	2	0.13	0.97	1.10
IB Cluster	4	0.12	1.13	1.25
* -> 2 GPUs per node	2*	0.13	0.99	1.12
	4*	0.13	1.58	1.71
	8*	0.13	1.63	1.76

Weak Scaling

- 128x128x60 with halo of width 3 in the horizontal boundaries
- Pure MPI on XE6/ XC30 systems with 32/16 sub-domains per node.
- One sub-domain per node/GPU on XK7
- Average time to simulate one hour of weather



Summary

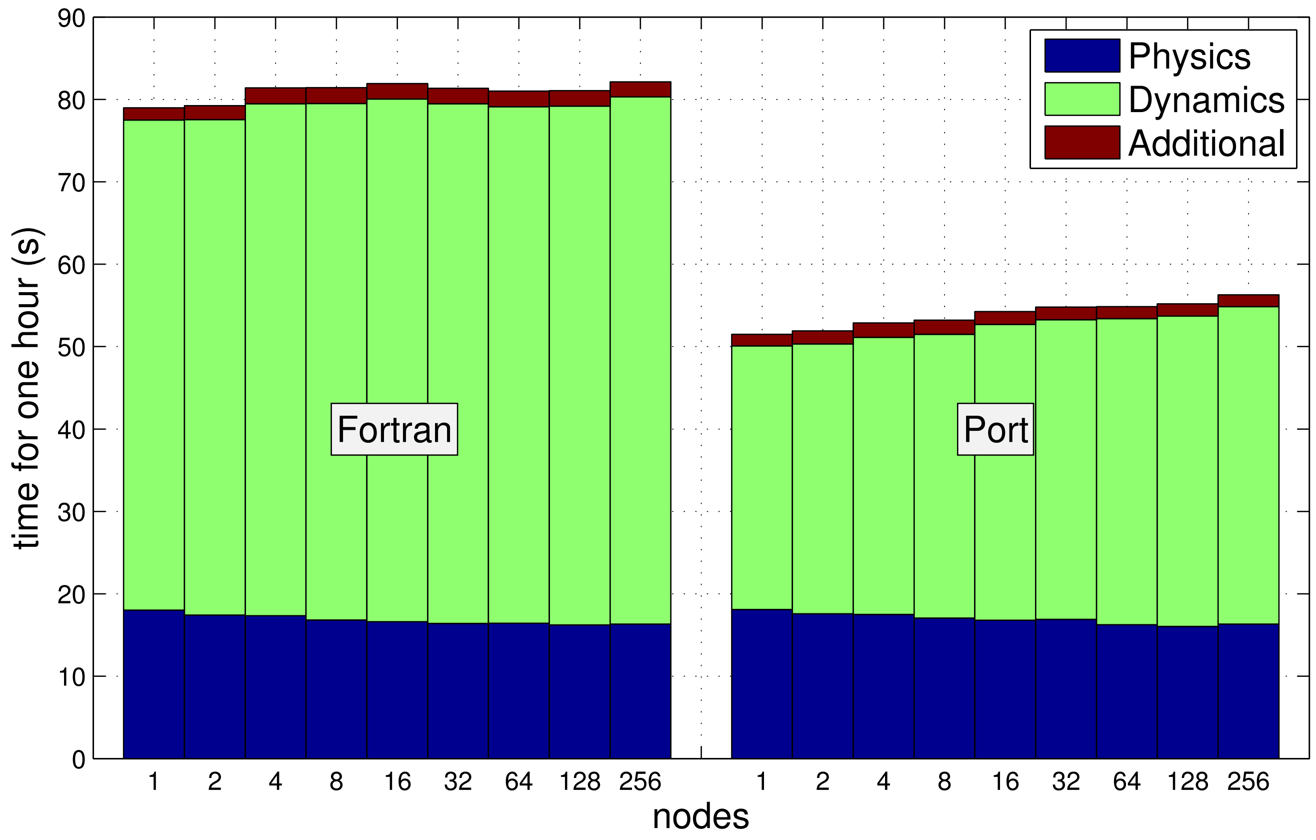
- Performance portability isn't easy!
  - Probably not possible with directives (sorry for the controversy)
- Domain specific languages can be a performance portable solution
  - Somebody has to maintain them
  - Getting the design right is a challenge
  - You have to get users on board
- Full production runs on GPU in June/July 2013
- Hybrid XC30 system at CSCS later in 2013

Any Questions?

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Weak Scaling of New Dynamical Core



- Time for an hour of simulated weather on Daint (XC30)
- Weak scaling with and without the new dynamics (implemented with the DSEL)

Programming model: Hybrid or Offload?

- The first question was which programming model?
  - **Hybrid** share work between both host and accelerator
  - **Offload** move all fields to, and perform all stencil computation, on accelerator
- The offload model was chosen
  - The overheads of copying prognostic fields over PCIx bus would have negated any computational improvements on GPUs*.

*Preliminary tests performed on HP SL390 node with PCIe 8x, and a Cray XT6 running COSMO-2 on 45 nodes showed that transferring 10 prognostic fields required for assimilation from device to host takes 118 ms, compared to the combined time for Dynamics and Physics of 253 ms

Tuning OpenACC

```
!$acc data present(a,c1,c2)
!vertical loop
do k=2,Nz
  !$acc parallel vector_length(N)
  !$acc loop
  do j=1,Ny
    !$acc loop vector
    do i=1,Nx
      c2(i,j)=c1(i,j,k)*a(i,j,k-1)
    end do
  end do
  !$acc end parallel
  !$acc parallel vector_length(N)
  !$acc loop
  do j=1,Ny
    !$acc loop vector
    do i=1,Nx
      a(i,j,k)=c2(i,j)*a(i,j,k-1)
    end do
  end do
  !$acc end parallel
end do
!$acc end data
```

Fuse loops that were optimized for NEC:

- Minimize DRAM traffic
- “cache” c2 in a register

```
!vertical loop
!$acc parallel vector_length(N)
do k=2,Nz
  !$acc loop
  do j=1,Ny
    !$acc loop vector
    do i=1,Nx
      c2=c1(i,j,k)*a(i,j,k-1)
      a(i,j,k)=c2*a(i,j,k-1)
    end do
  end do
end do
acc end parallel
acc end data
```

Note the k-loop dependency

DSEL Compilation Walkthrough

- The backend generates hardware-specific code
- The example below is an approximation of what the OpenMP backend would produce for our Laplace example
  - The CUDA back end generates CUDA kernels

```
 #pragma omp parallel for
for(int block=0; block < numOfBlocks; ++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);
        context.Advance<0,0,1>();
      }
    }
  }
}
```

DSEL Walkthrough : Stencil Logic

- The stencil logic is expressed in a C++ functor
- The functor is pure type information

```
struct Laplace{
  static void Do(Context ctx){
    ctx[lap(center)] = ctx[in(iplus1)] + ctx[in(iminus1)]
                      +ctx[in(jplus1)] + ctx[in(jminus1)]
                      -4.0 * ctx[in(center)];
    }
}
```

DSEL Walkthrough : Loop Logic

- The stencil compiler is used to specify loop logic
- Note the Laplace functor and loop bounds & directions passed as template parameters
- A compile-time flag selects the specific hardware back end

```
Stencil stencil;
StencilCompiler::Build(  // compile the stencil
  stencil,
  ...
  define_sweep<cKIncrement>(
    define_stages(
      StencilStage< Laplace,
                    IJRange<0,0,0,0>,
                    KRange<0,0> >()
    )
  ),
  ...
);
stencil.Apply();     // apply the stencil
```