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

Dr. Ben Cumming
Swiss National Supercomputing Center
Lugano, Switzerland
bcumming@cscs.ch

Dr. Carlos Osuna
Center For Climate Systems Modeling ETHZ
Zürich, Switzerland
carlos.osuna@env.ethz.ch

Mr. Tobias Gysi
Supercomputing Systems AG
Zürich, Switzerland
tobias.gysi@scs.ch

Dr. Mauro Bianco
Swiss National Supercomputing Center
Lugano, Switzerland
mbianco@cscs.ch

Dr. Xavier Lapillonne
MeteoSwiss
Zürich, Switzerland
xavier.lapillonne@meteoswiss.ch

Dr. Oliver Fuhrer
MeteoSwiss
Zürich, Switzerland
oliver.fuhrer@meteoswiss.ch

Prof. Thomas C. Schulthess
ETH Zürich
Zürich, Switzerland
schulthess@cscs.ch

Abstract—**We summarize the results of porting the numerical weather simulation code COSMO to different hybrid Cray HPC systems. COSMO was written in Fortran with MPI, and the aim of the refactoring was to support both many-core systems and GPU-accelerated systems with minimal disruption to the user community. With this in mind, different approaches were taken to refactor the different components of the code: the dynamical core was refactored with a C++-based domain specific language for structured grids which provides both CUDA and OpenMP back ends; and the physical parameterizations were refactored by adding OpenACC and OpenMP directives to the original Fortran code. This report gives a detailed description of the challenges presented by such a large refactoring effort using different languages on Cray systems, along with performance results on three different Cray systems at CSCS: Rosa (XE6), Todi (XK7) and Daint (XC30).**

I. Introduction

The consortium for small-scale modeling (COSMO) is a limited-area, non-hydrostatic atmospheric model for both regional weather forecasting and climate modelling. It is used both for operational weather forecasting by numerous national weather services, including those in Germany, Switzerland and Italy, and for limited-area climate models by the climate community. The different operational requirements of these two communities often mean that each community has a different definition of an "optimal" application. Operational weather forecasting is often subject to a "time to solution" objective: the solution for a model must be obtained inside a time limit imposed by the forecasting schedule. Thus, the aim is to minimise the hardware and power requirements to to obtain the solution in a given time frame. Conversely, climate simulations typically use larger grids and simulate much longer time frames, so the objective is to minimise the time to solution[1].

[1]Minimising the "storage to solution" is also a major concern for climate models, which isn't discussed in this paper.

In this paper we summarize the efforts of a project that was performed as part of the HP2C initiative in Switzerland to port COSMO for different multi-core and many-core hardware architectures. The original COSMO, which is written in Fortran 90 with flat MPI parallelization, is heavily optimized for NEC vector machines at the expense of sub-optimal performance on x86-based machines. This project aims to satisfy the dual objectives of broadening the base of possible architectures on which the application is optimized, while giving users to choose a hardware configuration that is best-suited to the requirements of their use case. An important requirement of the project is that the new code should be readily extended to support new architectures, while retaining a single performance-portable source code. These objectives of supporting wide range of hardware while retaining a single source code are essential for ongoing maintenance of the code and adoption by the user community, but they are very challenging to acheive.

To this end, two approaches are used to port different parts of the code base: a domain-specific embedded language (DSEL) written with C++, OpenMP and CUDA which abstracts the stencil logic from the loop logic with hardware-specific backends; and adding OpenACC and OpenMP directives to the original Fortran code. This paper will give our hands-on experience with these different approaches, with a particular focus on using the Cray toolchain on Cray systems at CSCS.

The structure of this paper is as follows. In Section II we will give a brief overview of the original source code and the programming models chosen for the port. The main strategies for porting, namely a domain-specific embedded language in C++ and directives, and our experiences with implementing them will be discussed in Section III. Numerical benchmarks that compare the CPU and GPU implementations of the ported code to one another and the original code on three Cray systems, namely XK7, XE6 and XC30 systems at CSCS, will then be presented in Section IV. Finally, Section V will summarize the findings to date of the project, along with future plans.

II. The COSMO Source Code

COSMO has about 250,000 lines of Fortran 90 code, with flat MPI parallelization. The number of lines in each component of the original Fortran 90 code base, along with their corresponding run time contributions, are shown in Figure 2(a) and (b) respectively. Each of the components are briefly described below:

• **Dynamics:** The dynamical core solves the equation of equations describing compressible non-hydrostatic atmospheric flow without any scale approximations. The dynamics account for over 40,000 lines of code, which are maintained and updated relatively infrequently small set of developers, and are the most computationally intensive with 61% of the total run time.

• **Physics:** The physical parameterization accounts for processes, such as radiation or turbulence, that are not described by the dynamics. This code accounts for roughly 43,000 lines of code that are regularly updated and extended by many members of the user community.

• **Assimilation:** Data assimilation integrates observation data into the model methods such as nudging [5]. Assimilation accounts for 40% of the code base, at 80,000 lines, yet has low run time contribution of 1%.

• **Diagnostics:** Generation of specific data fields for later analysis inside the model. This has not yet been ported, and is not covered in this paper.

• **IO:** The IO was optimized as part of this project, however this was independent of the work on hybrid architectures, and is not discussed in this paper.

The execution sequence of these components in the time stepping loop is illustrated in Figure 1.

The main focus of the porting effort has been on the dynamics and physics, which combined account for over 80% of the time to solution. However the other components must also be considered carefully to avoid bottlenecks, particularly:

- Ahmdahl's law will come into effect on multi-core and many-core systems where serialized code remains, even if the serialized code has a very small proportion of the runtime in the sequential version.

- The PCI bus can become a bottleneck when using accelerators, such that it may be necessary to port a computationally inexpensive component to the GPU to avoid copying data between host and device over the PCI bus.

The aim of providing performance-portable code while minimising disruption to users is very challenging for a code with a large, and active, community of developers. As a result, different approaches were taken to port different components, as dictated by the requirements of the different developer communities.

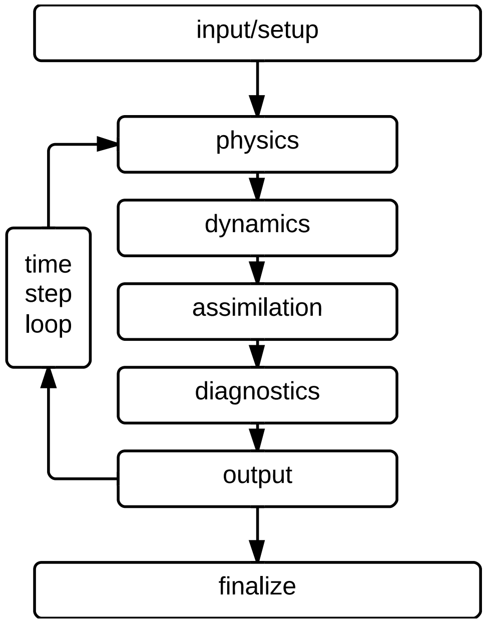


Fig. 1. An abbreviated version of the main time integration loop for COSMO. The size of each step, both in terms of source code and run time are summarized in Figure 2.

A. Programming Model

The flat MPI parallelism in the original COSMO is replaced with both coarse-grained and fine-grained parallelism. The coarse-grained parallelism corresponds to the flat MPI approach in the original implementation, with domain decomposition and MPI for communication of halo updates. An additional level of fine-grained parallelism is introduced, with OpenMP on multi-core hardware and CUDA parallelization on GPUs.

Currently two target hardware platforms are supported[2], which require different approaches to fine-grained parallelism at the node level:

• **Multi-socket x86 node:** A typical node will have two multi-core sockets. A hybrid MPI-OpenMP approach is taken, with no more than one NUMA domain per MPI process.

• **GPU node:** A node with one or more GPU accelerators. CUDA and OpenACC provide fine-grained parallelism on the GPUs, with one MPI process for each GPU. Current efforts are focussed on Cray XK7 nodes, which have one K20X accelerator per node, and a fat node with two Sandy Bridge sockets and 8 K20 accelerators.

An important consideration when using accelerators is whether to implement a hybrid execution model, whereby work is shared by both host and accelerator. The two choices are presented below, however we note that it is possible to have a solution that lies somewhere between these two extremes:

• **Offload:** All of the data fields and computational work are stored and performed on the accelerator, with the host used only coordinating GPU execution, and auxiliary tasks such as IO. This approach does not utilize the CPU for computation:

[2]Preliminary tests of the new Dynamical Core have been carried out on Intel KNC, however multi-node tests have not yet been tested yet.

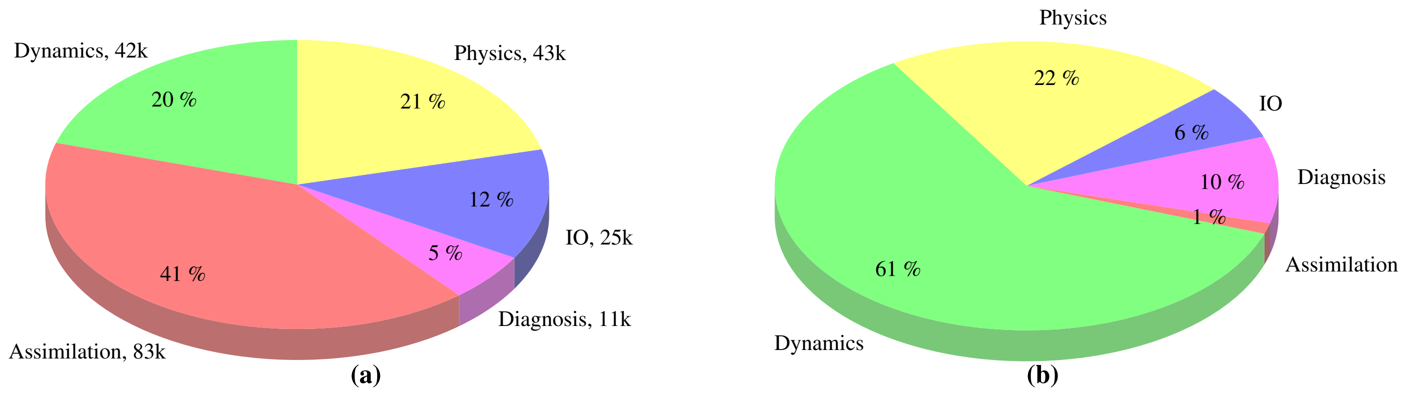


Fig. 2. A breakdown of: (a) the number of source code lines the main components of the COSMO code base; (b) the contribution of each component to the time to solution.

it coordinates the GPU, communication and IO.

• **Hybrid:** The computational work is divided between both host and device to take advantage of all computational resources on the node. This approach potentially maximises utilization of available computational resources, however it is typically more difficult to implement load balancing and synchronization between host and device, and leads to increased data traffic on the PCI bus.

The initial intention was that parts of the time step that were computationally inexpensive, such as data assimilation which has 80,000 lines of code and contributes only 1% of the run time, would remain on the CPU. The computationally intense aspects of the dynamical core and physical parameterizations were to be offloaded to the GPU.

Tests showed that time to transfer 10 prognostic fields required for assimilation from device to host as 118 ms, compared to the combined time for Dynamics and Physics of 253 ms[3]. Despite there being little computational benefit in moving the assimilation to the accelerator, it was nonetheless necessary to avoid allowing the PCI bus to become a bottleneck. So a full offload model is used everywhere in COSMO to avoid copying prognostic fields over the PCIe bus, with full fields only copied from device to host to perform IO.

III. Strategies For Porting

In this section we give an overview of the approaches taken to port the dynamical core and the physical parameterization, along with the changes to the MPI communication infrastructure that this required. In each case the aim of developing a single source code that is performance portable had to be balanced against the requirements of the users who develop and maintain the different components.

A. *The DSEL Stencil Library*

It was decided to that the dynamics, which is the most performance-critical part of COSMO with 60% of the run time, which would be completely rewritten. A rewrite was feasible in this case because the dynamics was developed and maintained by one main developer, which makes collaboration easier than a disperse group of developers as is the case with the physical parameterization. The dynamical core is also updated infrequently which makes it less of a "moving target" for a complete rewrite. With this in mind, a domain-specific embedded language (DSEL) tailored for the stencil algorithms in the dynamics was developed in C++, and the entire dynamical core was rewritten using the DSEL, referred to as *the stencil library*.

The DSEL separates the loop logic from the stencil definition, which allows aggressive optimization targeted at a specific hardware back end to be performed at compile time (performance portability). In this manner, efficient implementations can be generated from a single stencil description for each hardware back end supported by the DSEL (single source code). The Fortran listing for a simple stencil shown in Figure 3(a) illustrates the separate loop and stencil logic for a horizontal Laplacian operator applied in three dimensions, with the loop logic highlighted in green, and the stencil definition in blue. Optimised versions of this stencil on two different architectures would have the same stencil definition, however the optimal loop logic on each architecture could be very different. For example, on the CPU one might pursue a blocking strategy that optimises for cache reuse and distributing the blocks between relatively few CPU cores. Likewise, on the GPU an implementor may want to use software-managed cache and choose thread block sizes that optimize occupancy. Examples of some different considerations include

- The storage order of the data fields.
- The loop ordering.
- Blocking and tiling on the loops.
- Buffering techniques for passing intermediate values between stencils that are applied one after the other.
- Software managed caching.

It is a major undertaking to implement such considerations into every stencil loop of an application like COSMO, which has many stencils, even when only one hardware platform is targeted. Doing this in a performance portable manner becomes far more challenging when multiple hardware platforms, each of which benefit from different optimizations.

[3]These preliminary results were obtained on a HP SL390 node with PCIe 8x, and a Cray XT6 running COSMO-2 on 45 nodes.

Unlike the optimal loop logic which is hardware specific, the stencil logic (marked in blue in Figure 3(a)) is the same on each architecture. Users of the stencil library implement the stencil logic in a C++ functor, as is done for the Laplacian operator in Figure 3(b). The loop bounds and the direction of the loop in the k-direction are then specified separately from the stencil functor using the DSEL, as shown in Figure 3(c). Because the stencil `Laplace` functor is a type, it can be passed to the `StencilStage` definition as a template parameter, along with parameters that describe the loop logic. There are three important pieces of loop logic specified: `cKIncrement` indicates that the loop is in ascending k order; `IJRange` specifies how many halo levels on each of the horizontal boundaries; and `KRange` indicates the extent of the range of the k-loop. The loop logic and the stencil logic are all passed to the stencil compiler as template parameters, and template meta-programming is used to generate very efficient code at compile time. The benefit of this approach is apparent in the relatively simple stencil description that can be compiled for multiple back ends, with the complicated implementation details are hidden from the user who does not require detailed knowledge of template meta programming. The main drawback with this approach is that the back end implementation are quite complicated, with specialist knowledge required to maintain and add support for new back ends.

```
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
```

(a)

```
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)];
    }
}
```

(b)

```
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
```

(c)

```
 #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>();
      }
    }
  }
}
```

(d)

Fig. 3. Example of how a simple horizontal Laplacian stencil is expressed using the DSEL. (a) The original Fortran implementation with the loop logic and stencil logic highlighted in green and blue respectively. (b) Shows the stencil logic expressed as an update functor, and the corresponding loop logic is illustrated in in (c). An idealized example of the code generated by the OpenMP back end after parsing the stencil definitions in (b) and (c) is shown in (d).

There are two levels of fine-grained parallelism used in the back end. The first level splits the domain into blocks in the horizontal IJ plane that can be processed independently with no need for synchronization and consistency. Then second level can be applied inside each block. This is not the case with the OpenMP back end, which uses one thread per block. The additional parallelism is implemented on the CUDA back end, where one thread is assigned to each vertical column, with the synchronization between the threads in a block. The different back ends use different loop and data orderings. The CUDA back end uses an IJK loop ordering that makes for coalesced memory reads and writes on the IJ plane. The OpenMP back end uses a k-first loop order that improves cache reuse and reduces cache conflicts between threads on adjacent blocks.

There are two two types of stencil motifs used in COSMO. The first motif has no dependencies in any direction, so that the stencil operation can be applied to the points of the grid in any order, which gives greater flexibility in choosing the loop logic. The second motif arises from the semi-implicit temporal integration scheme used in some parts of the time stepping that uses implicit integration in the vertical direction. This temporal discretization requires the solution of an independent tridiagonal linear system for each vertical column, for which the direction of application of the stencil in the k-direction must be specified due to loop dependencies in the forward and backward substitution phases of the Thomas algorithm used to solve the tridiagonal systems. This motif, which also occurs in parts of the physical parameterization, had a large influence on the design of the DSEL:

- **Explicit k-loop direction:** As shown in Figure 3(c), the user specifies the direction of the k-loop as a template parameter in the `define_sweep<>`, while the i-loop and the j-loop directions are defined by the back end.
- **Horizontal blocking/tiling:** The blocking is performed in the horizontal plane only, because blocking in the vertical direction would require syncronization between vertically adjacent blocks.
- **One thread per column:** The GPU back end uses one thread per column, with each thread performing the k-loop that applies the stencil to each point of the column.

The template meta-programming techniques used in the stencil library test the capabilities of C++ compilers. The compile times are slower than for the original Fortran implementation, but have not been as bad as initially feared: a parallel build of the OpenMP version of the dynamical core takes less than two minutes on a Sandy Bridge-based system. The GNU C++ compiler was primarily used to develop the library because if offers very robust and mature support for such codes. The Intel compiler is able to compile the code, however it generally gives executables that are 10–20% slower than those generated with GNU. The Cray C++ compiler has also been tested to generate executable that was competitive with the Intel compiler, however compilation times were much longer, which make it unattractive for development. This is an important point for C++ codes where performance can depend strongly on the compiler: developers make design decisions that affect the performance according to the tools used during development.The new C++11 standard has features that would make the stencil library simpler, however these features are not supported by the nvcc compiler which is used for the CUDA back end.

The DSEL is a significant departure from the original Fortran code, and it is important to involve users in its development. A week-long work shop with was held in Frankfurt in November 2012 for users from the weather and climate communities. The aim of the workshop were two-fold: first to introduce users to the stencil library; and secondly to gather feedback. The response from users was positive, with participants at the workshop able to write reasonably complicated stencils that used most features of the stencil library by the end of the week, and there is an ongoing effort to port a new fast waves solver in the dynamical core from Fortran to the stencil library, in collaboration with its developer at the German weather service (DWD). Feedback was also very valuable, highlighting the need to involve the broad community to determine which features they value the most, and to see their response to different design choices that were made in the library. For example, it was very informative to see how sensitive the specialized user community is to domain-specific nomenclature, which lead to changes in the naming and structure of the DSEL to better reflect community norms.

The development of the stencil library is an ongoing project, with the aim of both improving the design and performance of the library. We are also considering how this work could be applied to other stencil-based codes in atmospheric and geophysical applications. We finish the discussion of the stencil library with a summary of the main pros and cons we have found with this approach.

Pros of the DSEL

- By separating the stencil and loop logic, each stencil is written only once to achieve good performance on the supported hardware back ends, which meets the aim of performance portability with a single source code.
- The separation of stencil and loop logic also removes hardware specific language (e.g. OpenMP and OpenACC directives, NEC directives, CUDA) from user code.
- The modular design is well-suited to unit testing, which accelerates debugging and regression testing.
- The DSEL provides a set of finite difference operators, so that the stencil expression better matches the mathematical formulation.
- Embedding the domain-specific language in C++ gives the user access to the rich features available in C++.
- Template meta-programming makes it possible to implement functions for commonly used stencil operations with no run-time overhead.
- Compilation times are quite reasonable for the OpenMP back end with the GNU and Intel compilers. They are considerably slower for the CUDA back end, however it is also that case that OpenACC compilation with the Cray and PGI compilers is very slow.
- New users who have little or no C++ development experience have been able to start using the stencil library with the tutorial materials that have been developed for the project.
- The work on the dynamical core has not been any slower than the work on the physical parameterizations, which added directives to the original Fortran code, an approach that is often touted as faster. This is partly due to the higher maturity of the C++ and CUDA compilers relative to the early OpenACC implementations, and because significant changes were required in the Fortran code to obtain good OpenACC performance.

Cons of the DSEL

- The DSEL is a significant departure from the approach currently used by the community, requiring considerable time and effort from users to adapt.
- The scope of the library is limited to functionality required to implement the dynamical core as it is. New developments in the dynamical core might require that new functionality is added to the stencil library.
- There is some repetitive boilerplate code when declaring stencils, which is often due to restrictions imposed by C++.
- Users who are not familiar with C++ can be distracted by having to learn a sub-set of C++ at the same time as the DSEL.
- The engineering effort to add support for a new hardware back end to the library is not trivial, requiring advanced C++ knowledge.
- Finding an appropriate balance between generic and domain-specific language is difficult. Some of the library features are specific to COSMO, which makes it easier to optimize them, but limits the usefulness of the DSEL for other applications.

B. *Directive-Based Fortran*

Compiler directives provided by OpenMP and OpenACC are non-executable statements that indicate to the compiler how code can be parallelized. OpenMP is a widely-used and supported set of compiler directives for multi-threading on multi-core machines[3]. The recently-proposed OpenACC directives [2] allow the programmer to specify which parts

```
DO j=1,je
  ! forward substitution
  DO k=2,ke
    DO i=1,ie
      c(i,j,k) = 1.0/( b(i,j,k)-c(i,j,k-1)*a(i,j,k) )
      d(i,j,k) = (d(i,j,k) - d(i,j,k-1) * a(i,j,k)) &
                 * c(i,j,k)
    END DO
  END DO
  ! Back substitution
  DO k=n-1,1,-1
    DO i=1,ie
      x(i,j,k) = d(i,j,k) - c(i,j,k) * x(i,j,k+1)
    END DO
  END DO
END DO
```

(a)

Note: the Thomas algorithm used to solve the independent tridiagonal system in each vertical column on the left has loop dependencies in k for the c, d and x fields, necessitating explicit k-loop order in the forward and backward solution phases. Whereas the stencil below can be processed in any order.

```
DO k=1,ie
  DO j=1,je
    DO i=1,ie
      a(i,j,k) = w1(i,j,k) * b(i+1,j,k) &
               + w2(i,j,k) * b(i  ,j,k) &
               + w3(i,j,k) * b(i-1,j,k)
    END DO
  END DO
END DO
```

(b)

Fig. 4. Examples of the two stencil motifs used in COSMO: (a) tridiagonal solve with k-loop dependency; (b) "normal" stencil with no loop dependencies.

of code to offload to an accelerator, with additional directives for controlling the transfer of data fields between host and device memory. A significant advantage of using directives over more ambitious approaches, such as rewriting the code, is that code parallelized by inserting directives is familiar to the original authors and users of the code. Because the physical parameterization has a large user community, and is regularly updated and extended, directives were chosen to minimize disruption to users, and keep pace with changes in the original source. The work on directives presented here has been covered in more detail in the paper [1].

The intention of this project is to use both OpenMP and OpenACC directives for a portable code that runs both on multi-core and accelerators. However, the main focus to date has been on the OpenACC port for the GPU, which was performed in two stages. The first stage added directives to store the required data fields on the device, and perform the stencil loops on the device. The second phase was to then investigate and tune the performance of the physical parameterizations. We now summarize the most important optimizations that had to be considered during the optimzation phase.

1) Exploit SIMD parallelization: The physical parameterizations perform computation on each vertical column in the domain independently. So given that each field has an (i, j, k) structure, a kernel can be applied in parallel over all i, j indexes at each k-level. This implies a loop ordering, like that used for implicit vertical integration in the dynamical core, with parallel execution in the horizontal plane, and sequential execution in the k-direction. The variable `N` in the OpenACC directive `vector_length(N)` can then be used to control vectorization.

2) Minimize transfers: To avoid bottlenecks caused by transferring fields between host and device memory, all data fields are copied to the device during initialization where they remain throughout time stepping.

3) Loop restructuring: In most cases it is sufficient to simply add directives to the loops. However, in some cases some restructuring of the loop is required to improve performance. By restructuring the loop in Figure 5(a) to that in Figure 5(b), the number of kernels is reduced from two to one, and the temporary variable c_2 is cached in a register between k loops. A further optimization that is employed for some performance-critical kernels is show in Figure 5(c). In this case, the original three-dimensional field with dimensions (n_x, n_y, n_z) is decomposed into two-dimensional blocks, each with dimension (n_{proma}, n_z). This new layout makes it easier to coalesce memory loads when n_x is not a multiple of 32, and gives greater flexibility in the choice of threads per block.

4) Remove automatic arrays: Automatic arrays in OpenACC incur a call to `cudaMalloc`, which has a significant overhead on the GPU. These are avoided by passing the array as an argument, and allocating the memory at a higher level in the code.

5) Occupancy: The occupancy of a kernel is influenced by the number of registers required per warp. Some experimentation is required to find the balance between increased occupancy (more, smaller kernels) and reducing kernel launch overheads (fewer, larger kernels).

6) Profiling and tuning dominant kernels: After the first round of implementation, the kernels are profiled with a GPU profiler, and tuning of parameters such as the vector length is performed on the dominant kernels.

By taking the steps outlined above, the performance of the code on the GPU can be improved significantly. However, the tuned code often does not perform well on the CPU. Indeed, the tuned GPU version of the code is considerably slower than the original code when compiled for the CPU. One most obvious method for keeping performance portability is to have two versions of each physical parameterization: one tuned for OpenACC on GPUs; and the other tuned for OpenMP on multi-core. This approach has the drawback of requiring that two versions of each parameterization must be maintained (or more if further hardware back ends are to be supported), and multiple versions of new parameterizations have to be written.

Currently we are investigating whether it is possible to have performance portable code that uses OpenMP and OpenACC directives in the source code. Our experiences to date have suggested that it is not possible to do this in an elegant manner for many of the kernels (particularly if we want to target more than one OpenACC compiler). The addition of accelerator directives in the proposed OpenMP 4.0 standard [4] has the potential to make this easier. However if the CPU and GPU versions of kernel have optimizations that differ greatly, it will probably still not be possible.

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

(a)

```
!$acc data present(a,c1)
!vertical loop
!$acc parallel vector_length(N) private(i,j,k,c2)
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
```

(b)

```
!$acc data present(a,c1)
!$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
```

(c)

Fig. 5. Example of restructuring performed to improve performance when adding OpenACC directives to the physical parameterization. In (a) two separate horizontal loops (in the ij-plane) are nested inside a vertical loop, with an intermediate result `c2` stored in an array (an optimization for NEC vector machines). By merging the two horizontal loops in (b) the loops are performed with one kernel, which stores the intermediate result `c2` in a register.

The physical parameterization has been quite frustrating at times due to immature OpenACC implementations. The support for OpenACC has developed a lot recently, however there are still compiler-specific workarounds throughout the code for compiler bugs. Also, there are currently only three compilers with support for OpenACC: Cray, PGI and CAPS. Of these, only Cray and PGI are mature enough to use for a large project like this, and furthermore, we are reliant on one compiler vendor on non-Cray systems. This is a major obstacle to uptake of OpenACC in the future. Again, the inclusion of accelerator directives in OpenMP 4 will not necessarily address this concern, because each compiler vendor will most likely support a limited subset of the available accelerators: for example, the GNU compilers will certainly support the new standard, however it will still most likely be left for hardware vendors to add support for their accelerators to GNU.

C. *Generic Communication Library*

The original Fortran COSMO code uses a one-size-fits-all halo exchange routine, with blocking MPI messaging. The rewrite of COSMO require a more flexible framework for halo updates, with the following requirements:

• **Back ends:** transparent support for fields stored on either CPU or GPU.

• **Layout:** support for fields with arbitrary orientation and padding.

• **Portability:** can be called from both the C++ and Fortran codes.

• **Boundary conditions:** handle related boundary conditions updates and halo updates at the same time via a common interface.

• **Communication hiding:** be able to use asynchronous communication to overlap communication and computation, which has a significant impact on strong scaling, where the ratio of halo regions to computation increases.

It was decided that a significantly different and more flexible solution was required for halo updates.

The Generic Communication Library (GCL) was developed to meet the halo exchange requirements of the COSMO rewrite. It is a generic C++ library for performing halo exchanges on two-dimensional and three-dimensional structured grids. It uses generic programming techniques to support data fields with arbitrary data layouts (IJK, KIJ, etc.) and arbitrary halo dimensions on each boundary (including communication with corner neighbours, i.e. all 26 neighbours in 3D). The library performs halo exchanges for data stored on different memory spaces, with CPU and GPU support currently implemented, via the same interface. It takes advantage of direct GPU to GPU communication, whereby the MPI API calls are passed device pointers directly. This feature is currently supported by the MVAPICH2 for Infiniband[4], OpenMPI[5] and by the Cray MPI library[6]

Halo exchange is composed of four stages:

• **Pack:** The 3D slices of the sub-domain that correspond to halos that are to be sent to neighboring sub-domains are packed into flat buffers.

• **Start exchange:** Post asynchronous receive and send calls with MPI.

• **Wait:** Wait for all pending MPI receives to finish, after which the halo contributions from neighbouring nodes will be stored locally in flat buffers.

• **Unpack:** Unpack the received halo information from the flat buffers back into the three-dimensional fields.

[4]Validated with MVAPICH2 version 1.8 and 1.9.

[5]Validated with version 1.7

[6]Validated with version `cray-mpich2/5.6.3`.

The original halo exchange performed the above stages in one routine with blocking communication. The GCL decouples the four stages, so that it is possible to first call pack and start the exchange, perform unrelated computational work, then perform the wait and unpack the fields. In this manner it should be possible to hide communication with computation, so long as asynchronous progress is enabled in the MPI library.

GCL supports MPI data types for the packing and unpacking phases, however these are only practical on the CPU implementation. Support for MPI data types on the GPU is either very slow (MVAPICH2 on Infiniband) or unsupported (Cray MPI). So custom CUDA kernels were developed for packing and unpacking 3D halos, which are very efficient compared to the OpenMP implementations due to the greater memory bandwidth available on GPUs.

GCL is used both in the C++ rewrite of the dynamical core and in the Fortran/OpenACC physical parameterization. The halo and boundary updates are handled by a C++ halo framework which has C++ and Fortran calling interfaces. The halo framework uses GCL for the halo exchanges, and implements the different types of boundary conditions applied at the global domain boundary. The framework then applies either the appropriate boundary condition or halo exchange at each boundary of a sub-domain via an "update-halo" call. We note that though GCL was developed primarily for use in the COSMO rewrite, it was designed to be fully generic, and COSMO uses only a subset of its features. It is being tested in other structured grid applications, and it is hoped that it will receive better support and testing as a result of being used in other applications.

D. Integration

Developing the full COSMO application using the different techniques outlined above has been a time-consuming and often frustrating task. This has been partly due to the challenges inherent in developing complex mixed C++ and Fortran codes, which are unavoidable. However, it has primarily been due to incomplete feature support provided by the OpenACC compilers and the MPI implementations. Throughout the integration effort, there have been many compiler-specific workarounds for missing features or bugs in OpenACC. This is the cost of being early adopters of a new technology like OpenACC, and the quality of these implementations has improved significantly to the point where many of the issues faced in this project no longer remain. However, the lessons learnt here are important, because such a "mixed" model may be required when porting other large codes to multi-core and many-core architectures.

The most significant challenges have been related to sharing data fields between the dynamical core and physics. Ideally, one would like to simply pass pointers for large three-dimensional fields between the C++ and Fortran, without any copying. However, this can be challenging in practice. The stencil library can add padding to fields, which gives significant speedup for the CUDA back end in particular (about 10% of both Fermi and Kepler). However padding is not simple to support in the current Fortran implementation, so the padding has to be turned off in the DSEL to avoid having to copy fields between C++ and Fortran. Additionally, The OpenACC `host` data directive, which is required for passing pointers to OpenACC regions in Fortran, was not implemented in the PGI compiler until version 13.1, which required an elaborate hack to work around.

The techniques used to keep track of which copy of a field is the current and previous steps are also complicated by the different approaches used by the C++ and Fortran implementations. The stencil library uses double buffering to perform this, with a pointer swap performed between time levels. To handle a three-dimensional field that is stored at two time levels the original Fortran implementation uses a four-dimensional array `u(nx,ny,nz,2)`, where the fourth index is "flipped" between time levels. It is very challenging, to determine *a priori* how many swaps or flips have been performed between entering and exiting each component. When it is not possible to determine how many swaps have been performed, it is necessary to perform a hard copy of the current in/out fields, which imposes significant storage and run time overheads.

Users in the community often have strong compiler preferences, so we want for users to be able to mix and match compiler toolchains for the different components. For the C++/CUDA implementation, the GNU and NVIDIA compilers must be used to build the stencil library, which is then linked with the Fortran code compiled with Cray or PGI compilers. The PGI compiler had limited features and bugs that placed onerous restrictions on the versions of the CUDA library and GNU toolchain that could be used for the dynamical core. It was also difficult to mix compilers when building the OpenMP version. We would like to be able to compile the stencil library with the GNU toolchain which gives the best performance, while being able to choose another compiler like PGI for the Fortran code. Unfortunately the OpenMP runtimes of the different toolchains are incompatible.

IV. Results

We now perform some benchmarks of the ported COSMO code on systems at CSCS, summarised in Table I. It is important to note that the code used in these tests is still under heavy development, and is currently being tested for validation and functionality, not performance. We expect to see considerable performance improvements after tuning, and as OpenACC and GPU communication support continues to evolve.

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	$1\times$K20X
Dom	Cluster/Infiniband	4	2×8-core Sandy Bridge	$2\times$K20C

TABLE I. A summary of the systems used for benchmarking.

A. Generic Communication Library

We investigate a GCL microbenchmark to understand the communication library performance on the different machines, in particular the performance of direct GPU to GPU (G2G) communication. The microbenchmark has a sub-domain of dimensions $128\times128\times60$ on each node, with an additional three halo levels on each i and j edges of each sub domain (for a total of 6 additional halo lines in each horizontal direction).

The global boundaries are non-periodic, so sub-domains that lie on the edge of the domain will have at least one boundary on which no halo exchange is performed. The halo exchange was performed 50 times, and the average time across all processes and all tests was taken. A small Inifinband-based GPU cluster called Dom was included in these benchmarks, to provide a comparison with Tödi for G2G communication. Dom has slightly less-powerful NVIDIA Kepler K20 GPUs than the K20X devices in Tödi, however this benchmark is concerned only with communication, not floating point performance.

The CPU support in GCL was tested on the XE6 and XC30 systems Rosa and Daint, with the total pack (combined pack and unpack times) and exchange times shown in Table II. On both systems a flat MPI approach was used, with 16 MPI tasks on each node (one per Sandy Bridge core on Daint, and one per Interlagos module on Rosa). This meant that each MPI task had a sub-domain of dimensions $32 \times 32 \times 60$, for a combined grid dimension of $128 \times 128 \times 60$ per node. We also tested using a hybrid MPI-OpenMP approach with fewer tasks per node. This approach aims to optimize communication by reducing the total amount of halo data to be exchanged, while increasing the average message size and reducing the number of MPI messages that are sent. Indeed, when using two MPI tasks per NUMA domain we the exchange communication time was reduced (results not presented here), however the total halo exchange times were longer because the pack and unpack routines have not yet been fully optimised with OpenMP. A minimum of four nodes are required to obtain the maximum MPI complexity in this case, as is evident in the sharp increase in exchange times when going from two to four nodes. After this, the weak scaling times for the exchange times are very good up to 128 nodes, with the XC30 system having the best pack and exchange times in each case.

When using the GPU, a single sub-domain of size $128 \times 128 \times 60$ is stored on the GPU on each node. Because Dom has two GPUs per node, results for both one and two GPUs per node are presented in Table II (results marked with a star use 2 GPUs per node, so 8^* is 4 nodes, with 2 GPUs on each node). The packing times on both Dom and Tödi are much faster than the CPU implementation, which is reasonable given the increased bandwidth available on the GPU. However, despite being twice as fast as those on Daint, the packing times on Tödi are over 3 times slower than on Dom. Furthermore the exchange times are much higher on Tödi: the exchange time for 4 nodes is 1.13 ms on Dom, compared to 3.12 ms on Tödi. The total halo exchange times for the GPU compare favorably to those obtained on Rosa and Daint. This is largely by reducing the packing time, which at $\tilde{3}0\%$ for four nodes on Daint and Rosa is a significant proportion of the total halo exchange time, to 15% on Tödi.

Part of the discrepancy in the exchange times between Tödi and Dom is due to a high degree of variability in the Tödi results. In Figure 6 the mean, minimum and maximum of the mean communication time across all processors are plotted. Both the minimum and maximum exchange time increase significantly when going from 2 to 4 nodes, and change little after that. However, communication on a small subset of the nodes is considerably slower, by a factor of about 6 times. The reason for this variation between runs is not immediately apparent, and is something that we will investigate further. We also note that the main time increases going from 8 to 16 nodes, which is to be expected because 16 is the minimum number of nodes required to maximise MPI communication complexity with 1 MPI task per node. The results on Dom were generally much faster (indeed, the mean values on Dom were faster than the minimum recorded times for Tödi). It would be very interesting to have access to a larger Infiniband-based GPU cluster to test how the communication results scale.

These test show that the total communication time, including both pack and exchange time, on Tödi is comparable to those on the CPU-only systems. However, there is still considerable scope for improvement, in particular the high amount of variability observed in the benchmarks and the relatively poor performance relative to the Infiniband cluster. Bug-free support for G2G communication has only been available since version 5.6.3 of the Cray MPI implementation, and as the implementation matures and GCL is tuned, we expect that halo communication will not be a disadvantage with GPU systems.

SYSTEM	NODES	PACK	EXCHANGE	TOTAL
Rosa	2	1.77	2.77	4.54
	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
	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
	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
	4	0.12	1.13	1.25
	2^*	0.13	0.99	1.12
	4^*	0.13	1.58	1.71
	8^*	0.13	1.63	1.76

TABLE II. AVERAGE TIME IN MS TO PERFORM HALO EXCHANGE ON THE DIFFERENT TEST PLATFORMS. THE *pack* AND *exchange* COLUMNS GIVE THE MEAN TIME IN MS TAKEN TO PACK/UNPACK AND PERFORM MPI COMMUNICATION FOR HALO EXCHANGES.

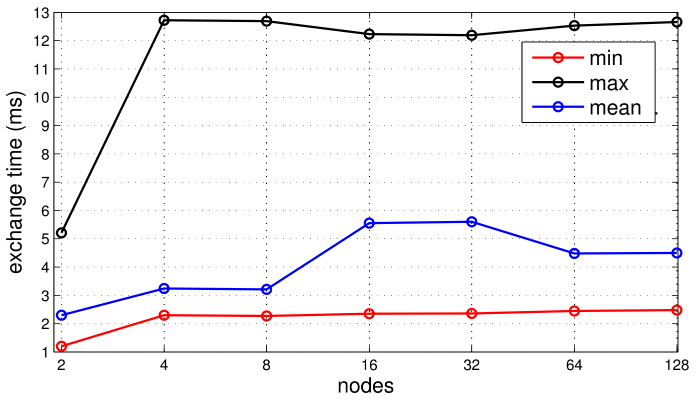


Fig. 6. The min, max and mean exchange times for the GCL microbenchmark with G2G communications on Tödi.

B. Strong Scaling

We now investigate the strong scaling of the physics and dynamics on a single node of each of the three Cray machines listed in Table I. The tests were performed on one node so that the communication could be decoupled from the stencil

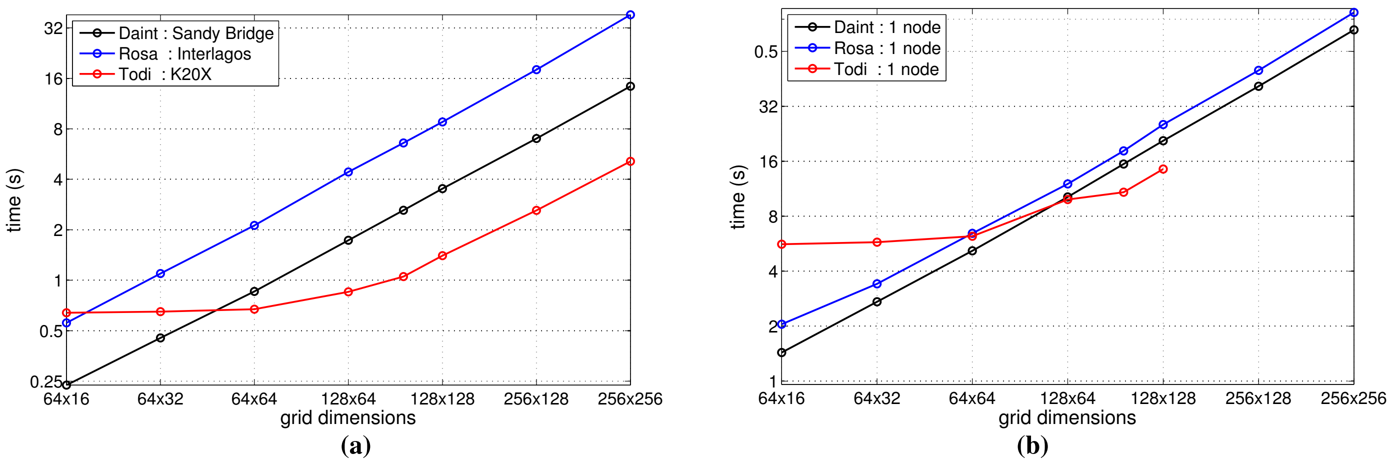


Fig. 7. Strong scaling results. In (a) the time to solution for 10 time steps is shown in seconds for one NUMA domain on Daint and Rosa (8 OpenMP threads in each case) and one K20X GPU on Todi. The average time in seconds taken to compute an hour of simulated weather on a node of each test machine is plotted in (b).

computation, to better focus on the scaling of the stencil computations.

First, we investigate the dynamical core implemented with the stencil library DSEL. The new dynamical core uses asynchronous communication, which has not yet been tested or optimized *in situ* with the dynamical core. To remove the communication overheads from tests of stencil performance we use a performance test with one MPI process and non-periodic boundary conditions. The OpenMP results presented here do not use all of the CPU resources on a node, because they are obtained with one NUMA domain[7] to avoid NUMA issues, so the results are roughly a factor of 2 and 4 times slower than those for a full node with multiple MPI processes on Daint and Rosa respectively.

The benchmark performs 10 full time steps of the dynamical core, and the total time taken at mesh resolutions from $64\times16\times60$ to $256\times256\times60$ are tabulated for each of the Cray systems in Table III, and plotted in Figure 7(a). The OpenMP results on Rosa and Daint were computed using 8 OpenMP threads, and the results on Todi were computed using one K20X GPU.

	Dynamical Core			Physical Parameterization		
Grid	Rosa	Daint	Tödi	Rosa	Daint	Tödi
64×16	0.56	0.24	0.64	1.02	0.72	2.81
64×32	1.10	0.45	0.65	1.71	1.36	2.89
64×64	2.12	0.86	0.67	3.22	2.59	3.11
128×64	4.42	1.73	0.85	6.02	5.10	4.94
128×96	6.59	2.62	1.05	9.12	7.72	5.41
128×128	8.79	3.52	1.40	12.71	10.36	7.25
256×128	18.02	7.01	2.61	25.16	20.59	–
256×256	38.32	14.35	5.12	52.25	41.94	–

TABLE III. Strong scaling results on Rosa, Daint and Tödi for the dynamical core and physical parameterization. Times in seconds.

The time to solution for the OpenMP implementation scales linearly with the grid resolution from the very small $64\times16\times60$ base domain size.The scaling behaviour is quite different on the GPU, with linear scaling only above a threshold grid resolution of $128\times96\times60$. The GPU implementation does not scale below this threshold because there is not enough work to utilize the computational cores and bandwidth of the device. We note that the scaling tests were not performed on grids larger than $256\times256\times60$ as this is the largest grid that can be stored on a GPU with 6GB of memory.

Similar strong scaling tests were performed for the physical parameterization, with the average time to simulate an hour of weather in a 12 hour simulation recorded. Tests on the full node were performed because the physical parameterization uses blocking MPI communication, so it is possible to decouple the computation from communication[8]. The CPU tests use flat MPI with 16 and 32 MPI tasks on a node on Daint and Rosa respectively, and the GPU results were obtained with one MPI task on one node of Tödi. The total time spent in stencil computation is tabulated in Table III and is plotted in Figure 7(b). Note that results are not available for grid dimensions greater than $128\times128\times60$ on Tödi due to memory limitations on the GPU[9]. The physical parameterization exhibits the same strong scaling that was observed for the dynamical core: linear scaling for all mesh resolutions on the CPU, and linear scaling above $128\times96\times60$ on the GPU.

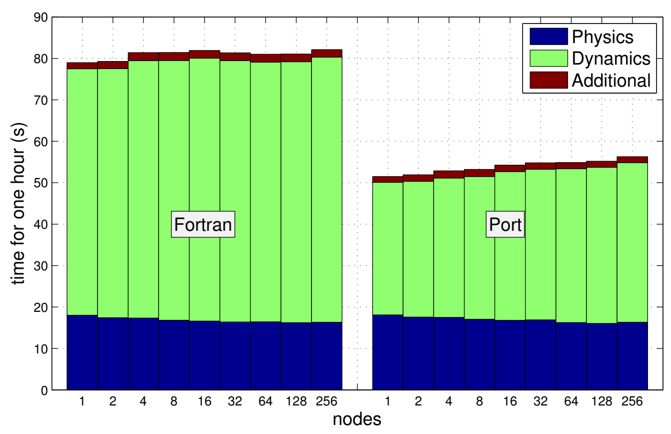
The scaling results for both the dynamical core and the physical parameterization suggest that the fastest time to solution may be achieved with strong scaling and the OpenMP back end. Conversely, the GPU back end is the fastest, by a factor of between 1.4 and 1.45 times when subdomains larger than $128\times96\times60$ are used on each node. This would be significant in weather forecasting where the solution is obtained inside a set time limit with the least resources, or for climate simulations on very large grids.

Finally, we note that the scaling of the CUDA and OpenACC used in the dynamics and physics respectively are very

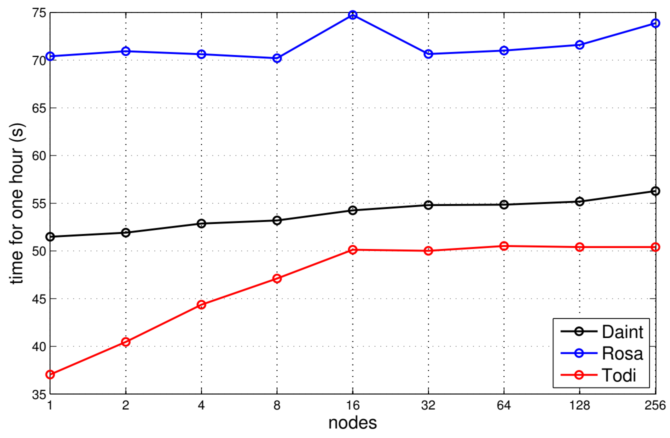
[7]On Rosa a NUMA domain corresponds to one Interlagos die, of which there are 4 on a node, each with 8 cores. On Daint one NUMA domain corresponds to 1 Sandy Bridge socket, of which there are 2 on each node, also with 8 cores.

[8]The GCL is used for communication in the physics, however communication is no asynchronous because the communication steps are performed sequentially.

[9]The memory footprint on the GPU will decrease when redundant copies between the dynamics and physics are eliminated.



(a)



(b)

Fig. 8. Weak scaling results. In (a) the time to solution for 10 time steps is shown in seconds for one NUMA domain on Daint and Rosa (8 OpenMP threads in each case) and one K20X GPU on Todi. The average time in seconds taken to compute an hour of simulated weather on a node of each test machine is plotted in (b).

similar. Currently some non-trivial physical parameterizations are being ported using the DSEL stencil library to give a direct comparison between the DSEL and OpenACC.

C. Weak Scaling

We now perform weak scaling tests to better understand the MPI communication overheads, and to compare the original Fortran implementation with the new code. The weak scaling tests were performed with a sub-domain dimension 128×128×60 plus halos of width 3 on each node. Weather was simulated for 12 hours, with pseudo-random intitial conditions[10], and the average time to simulate one hour was recorded. Weak scaling is performed from 1 node to 256 nodes on each machine, for a total grid dimension of 2048×2048×60 at 256 nodes. Flat MPI model used for the CPU implementations because the physical parameterization does not yet support OpenMP, and the GPU implementation used one MPI task per node.

First we compare the original Fortran implementation to the new dynamical core and GCL communication on Daint. The total time to solution for both cases, along with a break down of the physics and dynamics, are plotted in Figure 8(a). The physical parameterization is the same in both cases, because its implementation is unchanged in the new version[11]. The dynamical core is significantly faster in the new implementation, which is mostly due to the tuning of the OpenMP back end for x86 cache performance. The improved dynamical core performance gives a total speedup of 1.45 times at 256 nodes relative to the original code. The weak scaling of the new dynamical core is good, but not as good as that for the old version, which used blocking MPI communication. The new dynamical core uses asynchronous communication, with more frequent, smaller messages to hide communication with computation. However, asynchronous progress was not enabled for these tests, so it is not possible to comment on the efficacy of asynchronous communication, although early tests suggest that a considerable amount of tuning will be required to optimize the weak scaling of the communication.

Next we compare the weak scaling of the new version of COSMO on each of the three Cray test machines, the results of which are plotted in Figure 8(b). The weak scaling on Daint from 16 nodes is very good, which is based on a 4×4 node grid, at which point MPI complexity is maximised. Scaling on Rosa is similar to Daint, with an outlier at 16 nodes, which is due to sub-optimal node ordering to minimise inter-node communication, which was not tuned for these benchmarks.

It was not possible to use direct GPU-to-GPU communication on Tödi because we have not had enough time to validate results with the latest Cray MPI implementation for GPUs. As a result, GCL was modified to allocate page-locked buffers in host memory, and the buffers on the GPU were first copied from device to host before using host pointers to perform communication, then the received results were copied back to the device. The additional host-device transfers have a negative impact on the weak scaling on Tödi, as illustrated in Figure 8(b). As the complexity of MPI communication increases from 2 to 16 nodes, the weak scaling is very poor, due entirely to increased communication overheads. Weak scaling is very good from 16 or more nodes, because the message passing complexity does not increase for more than 16 nodes[12]. Despite the poor initial scaling, the GPU results on Tödi are still faster than the dual socket Sandy Bridge system Daint up to 256 nodes. Based on our observations for the GCL benchmark in Section IV-A, it is reasonable to expect that the communication overheads on the Tödi will be reduced significantly once full GPU-to-GPU communication has been implemented and tuned. These results highlight the importance of having an efficient MPI implementation for direct GPU communication for stencil codes like COSMO.

[10]Pseudo random initial conditions are set withing physically realistic ranges, and the results are validated to ensure that physically reasonable results were obtained.

[11]future work on the physical parameterization will include loop restructuring and the addition of OpenMP directives, which will improve performance on multi-core x86 nodes.

[12]A 4×4 process grid is used with 16 MPI processes, which has 4 processes that perform halo exchanges with all 8 neighbours (including corners). For 8 nodes on a 4×2 process grid, the most neighbours any process has to exchange with is 5.

V. Discussion

In this paper we have presented an overview of the strategies used to port a large, real-world atmospheric simulation code from flat MPI to hybrid systems, along with some preliminary results. Two different approaches were taken to port the original Fortran 90 code to hybrid architectures. The first was to port the dynamical core using a DSEL in C++ that can compile efficient code for both OpenMP and CUDA back ends from a single source file. The second approach added OpenACC directives to the original Fortran 90 code for the physical parameterizations. The results presented here have shown that both approaches are viable: a node to node comparison between a Cray XK7 (K20X GPU) and Cray XC30 (dual socket SandyBridge) gave speedup in the range of 1.4–1.45 times on the GPU for both the dynamical core and the physical parameterization.

The ongoing viability of this project will depend on improvements in both the OpenACC compilers, and the implementation of direct GPU-to-GPU communication in MPI. Currently, the main bottleneck for the GPU implementation of our library is the direct GPU-to-GPU communication: if this improves we expect to have very competative hybrid GPU performance. Another significant challenge, for which we currently have no solution, is how to write performance portable code for both OpenMP and OpenACC in a single source code. This is an important question that has to be addressed, because with current tools this is most likely not possible.

The issue of performance-portability is addressed by the DSEL, which can generate efficient code for both OpenMP and CUDA back ends from a single user-defined stencil description. This approach has the additional benefit of removing hardware-specific code from user code, and allowing users to express the stencil operations in a language that is closer to that used in their mathematical formulation. The main impediment to user adoption is the radically different approach taken by the DSEL, which can be intimidating to domain scientists who are uncomfortable with migrating from Fortran to C++. However, the programming model will have to change from the flat MPI model if multi-core and many-core systems are to be utilized efficiently, and it is likely that it will involve either a DSEL-like approach or compiler directives.

Acknowledgment

The authors would like to thank Roberto Ansaloni from Cray and Peter Messmer from NVIDIA for their contributions and assistance throughout this project. This project was supported by the HP2C initiative in Switzerland, and through the POMPA working group of the COSMO consortium.

References

[1] X. Lapillonne and O. Fuhrer. Using compiler directives to port a large scientific application to GPUs: and example from atmospheric science. *Parallel Processing Letters*, submitted for review.

[2] OpenACC. *The OpenACC application programming interface v 1.0*, 2011.

[3] OpenMP Architecture Review Board. *OpenMP Application Programming Interface v 3.1*, 2011.

[4] OpenMP Architecture Review Board. *OpenMP Application Programming Interface Version 4.0 - RC 2*, March 2013.

[5] C Schraff. Data assimilation and mesoscale weather prediction: A study with a forecast model for the alpine region. Publication 56. Technical report, Swiss Meteorological Institute, Zurich, 1996.