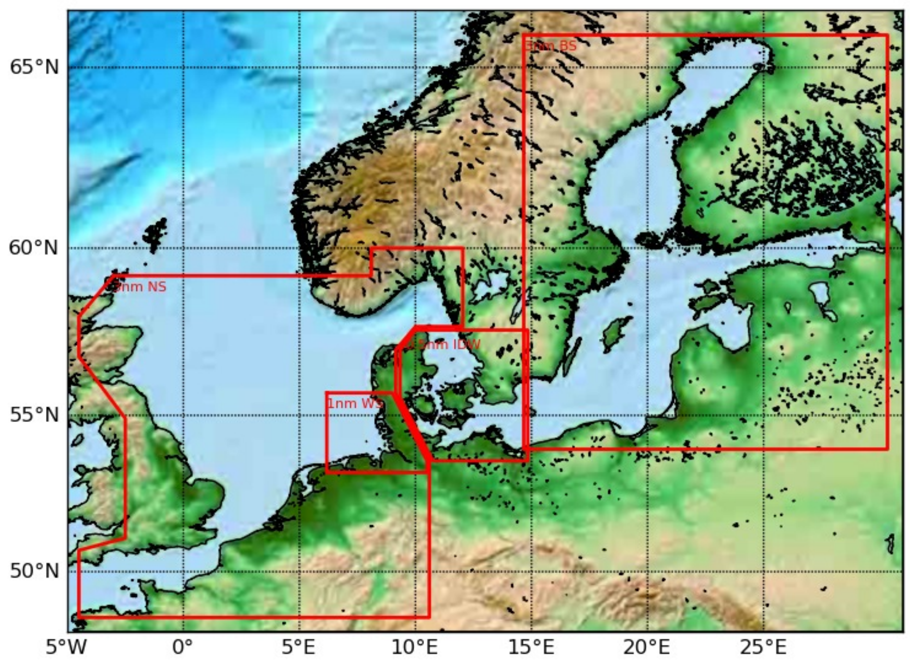
Danish Meteorological Institute
Ministry for Climate and Energy

Technical Report 12-11

Implementation details for HBM

Per Berg and Jacob Weismann Poulsen



www.dmi.dk/dmi/tr12-11.pdf Copenhagen 2012

Colophone

Serial title:
Technical Report 12-11

Title:
Implementation details for HBM

Subtitle:

Authors:
Per Berg and Jacob Weismann Poulsen

Other Contributers:

Responsible Institution:
Danish Meteorological Institute

Language:
English

Keywords:
operational ocean models, code quality, parallelisation, validation, verification, openMP, MPI, openACC, vectorization, nesting, scaling, cache layout, NUMA, nightly tests, automatic testing, geometric decomposition

Url:
www.dmi.dk/dmi/tr12-11.pdf

ISSN:
1399-1388

ISBN:

Version:
1.0

Website:
www.dmi.dk

Copyright:
Danish Meteorological Institute

DMI
Technical Report 12-11

Implementation details for HBM

Per Berg Jacob Weismann Poulsen

April 20, 2012

The aim of this document is to convey relevant information to people that intend to work with the source code of the ocean circulation model HBM, or who want to learn how this model code distinguishes itself from other model codes in (sometimes) unique ways. We assume that the reader is familiar with basic concepts of modelling and scientific computing. We assume a reasonably high level of familiarity with standard Fortran95 as well as with the openMP standard and the MPI standard. Moreover, we assume that the reader is familiar with basic notions within Earth system modelling, especially within oceanography.

That is, this document will focus on those topics that may be considered special for HBM compared to other models, and it will not deal much with "common knowledge". The solutions that we deal with in this document are all invented by the authors and reflect our deep interest in performance both from the numerical point of view and the computational point of view. We have tried to cover the aspects of the implementation that we find most interesting but our ultimate recommendation is still to use *the force* and read the source.

This document is the proper choice if you need a general reference to the HBM implementation. By quoting this document and a suitable source code tag, you can easily make reference to a specific release.

Acknowledgement: We are most grateful to Luiz DeRose from Cray for sponsoring our access to a GPU based test system and to John Levesque from Cray for helping with the GPU port. We also wish to express our gratitude to Maria Grazia Giuffreda from CSCS for granting us access to some of their systems and to Chris Gottbrath and Bill Burns, Rogue Wave software, for adding the requested features to the totalview^{TM} debugger allowing us to establish a debugging framework based on code commentary.

Finally, we would like to thank Rong Fu, DMI, for helping with some of the plots and tables and Jacob Woge Nielsen, DMI, for constructive comments on an early version of this document.

Front page: The figure on the front page shows the geographical extent of the setup used for the MyOcean Baltic Sea Version 2 operational model runs at DMI. This is also the test-case most extensively used throughout this report.

1 Introduction

The origin of the HBM code dates back to the BSHcmod ocean circulation model, the development of which was originally initiated in the 1990s at Bundesamt für Seeschifffahrt und Hydrographie (BSH) in Germany where it since then has been applied operationally for storm surge warning. The first part of the name refers to this origin while the last part is an abbreviation of what it is, namely a **C**irculation **MOD**el. Application and development has branched off from this origin into different institutions around the Baltic Sea.

One branch is the HIROMB[1] which has been operated for ocean forecasts at the Swedish Meteorological and Hydrological Institute (SMHI) since 1995.

Another branch of the model code has been maintained and further developed at the Danish Meteorological Institute (DMI), where it since 2001 has served as the operational storm surge model as well as being heavily used in a number of EU-funded projects. During different periods over the years it became known first as DMI-BSHcmod and later as DMIcmod to reflect the changing affiliation of the core developers.

Nowadays, attempt has been made to merge the model code development into a common project, the intention being to join the forces of experienced modellers from around the Baltic Sea. This model code project carries the name HBM, the HIROMB-BOOS[2] Model, not to promote a single institute but rather to indicate the affiliations of its major user and developer community.

The HBM code started from a snap-shot of the DMIcmod repository in 2009. The source code of the HBM project is currently maintained through a subversion server hosted at DMI. A previous release[3] of the HBM code runs primo 2012 operationally, using different setups, at DMI as the storm surge model and at the four MyOcean[4] Baltic Model Forecasting Centre production units BSH, DMI, SMHI, and FMI (Finnish Meteorological Institute) as the MyOcean Baltic Sea Version 2. The current code implementation is as close as one gets to a total rewrite without it being so from the original

[1]High Resolution Operational Model for the Baltic sea, `http://www.hiromb.org/`
[2]Baltic Operational Oceanographic System, `http://www.boos.org/`
[3]subversion tag `myov2` of November 3, 2011.
[4]MyOcean is the main European project dedicated to the implementation of the GMES Marine Service for ocean monitoring and forecasting, `http://www.myocean.eu/`

BSHcmod, and it is very different from the HIROMB code implementation; this has been necessary in order to meet our quality requirements for the code implementations and thereby also for the model results. The code is written in free format standard Fortran95 and has been parallelized using openMP, MPI and more recently also openACC. Based on experience from real applications, we have defined a set of code styling rules on top the ANSI[5] standard and some validation procedures that help the developers during maintenance, aiming at assuring portability and correct behavior. We have included a summary of the styling rules in appendix B. The development team takes a continued effort in keeping the quality high and the sloccount[6] low. That is,

- we rate correctness higher than added functionality,
- we rate optimization with respect to correctness and functionality higher than optimization with respect to speed,
- we rate optimization with respect to speed very high but never so high that we sacrifice portability,

because we strongly believe that this is the right choice in the long run. It is very important that model results are reproducible at any time, anywhere, by anyone.

Our approach when introducing new and expanding existing functionality, optimizing for speed, implementing support for e.g. accelerators, etc, is basically iterative and incremental[7]. We always start off from a well-defined foundation and on that we grow the new idea, learning how it behaves in real applications, attempting to map potential issues, improving the implementation as needed, and at the end of the day, we accept or reject it. We accept no unjustified deviations from the reference. That is, no code revision can change the results unless it is a genuine bug fix, a new physical feature or another solution algorithm for an existing feature, in which cases appropriate justification must be presented and the entire batch of test-cases

[5]American National Standards Institute, a private non-profit organization that oversees the development of voluntary consensus standards for products, services, processes, systems, and personnel, `http://www.ansi.org/`

[6]`SLOCCOUNT` is a set of tools for counting physical Source Lines of Code, cf. `http://www.dwheeler.com/sloccount/`

[7]Iterative and incremental development is the basis of cyclic software development, as opposed to the more linear waterfall model approach, see e.g. `http://en.wikipedia.org/wiki/Iterative_and_incremental_development/`

must be re-run and their references altered.

There are several issues that will affect code design. We have to be aware of these because they will put limitations on what we can do but also make something easier than with other models. The five most important model features in HBM from this point of view are:

- regular lat/lon grid with fixed but user-specified angular grid spacings at run-time, i.e. equidistant in latitude direction,
- choice at run-time of z-coordinates with fixed spacing varying with depth and time-varying top-layer thickness, or so-called dynamical vertical coordinates with grid spacing varying in time,
- designed for any number of two-way nested domains with any number of nesting levels specified at run-time,
- low-order finite differencing keeping the operator lengths short at +/-1 grid point,
- time stepping is explicit in the horizontal direction, and implicit in the vertical for momentum, diffusion and turbulence.

All of these will have to be addressed when we define proper data structures, memory access, domain decompositions for parallelisation, etc. In this document we will make attempts to cover these issues and to explain what we do to ensure implementation quality. We start off by describing in chapter 2 the major data structures, the basic indexing, and how these relate to the memory cache usage. In chapter 3, we describe how nesting is implemented in HBM and how this affects the computational flow. The subjects covered in chapters 2 and 3 are themselves useful independently, but they also constitute the prerequisites for our approaches towards parallelisation, explained in chapter 4. Finally in chapter 5, devoted to validation and verification, we attempt to describe what we do to ensure that the model behaves as expected such that we with peace in our minds can release the code for production runs.

The present report describes the HBM code as it is today[8]. In the future we will have improved the code, fixed issues, implemented new features, etc, and updates of (parts of) this report will likely be appropriate.

[8]subversion tag `release_2_5` of March 22, 2012.

2 Data structures

This section describes the basic data structures used throughout the model. We will explain the layout of $2D$ and $3D$ variables and describe how one can access them within each nested area. Moreover, we provide technical background on the data permutations that takes place during the initialization. The permutations have caused confusion in the past and they play an important role once we get to deal with cache layout later in this chapter, as well as with the NUMA[9] layout and the task local MPI layout in the chapter on the parallelization of the code.

2.1 Grid structure

The $3D$ grid is a staggered Arakawa C grid. The model uses a regular latitude/longitude grid in the horizontal plane, thus limiting the range of applications to regional models that do not cover the North Pole. In the vertical direction one uses runtime options to choose between classical fixed z-coordinates and so-called dynamical vertical coordinates. Being regular, the horizontal grid has fixed angular grid spacings. The grid spacing in the latitude direction need not be the same as in the longitude direction. The vertical grid spacing varies with depth; when using dynamical vertical coordinates, the vertical grid spacing varies with time too.

A number of nesting levels may be used, between which there is a fully, dynamical two-way nesting, cf. next chapter. The nesting levels are allowed to have different but constant horizontal resolutions, possibly different vertical resolutions, and different time stepping levels. The outermost domain may have a number of open lateral boundaries; the inner, fine(r) domain(s) cannot have any open lateral boundaries, only nesting borders to the enclosing domain(s). All domains have a free surface which exchanges matter and energy with the atmosphere.

The grid origin is the location of the North-West corner grid point. The positive direction of the longitude axis is to the East, the positive direction of the latitude axis is to the North, and the vertical axis is positive upwards. Any constant z-level constitutes a geopotential surface, and $z = 0$ is often chosen near mean sea level.

[9]NUMA stands for Non-Uniform Memory Access

2.2 Basic indexing

Internally in the code, the index set (i, j, k) refers to the three spatial directions. The grid origin has index $(1, 1, 1)$. Each index i, j, and k assume positive values only. Note that in e.g. finite differencing we often run into say $i - 1$ which must be mapped out-of-domain if $i = 1$, see description of indexing in section 2.4 on wet-point compression. The first index, i, refers to the latitude direction, and increases in the southward direction. The second index, j, refers to the longitude direction and increases in the eastward direction. The third index, k, indicates the model layers with $k = 1$ signifying the surface layer and k increasing downwards.

With the above definitions, the index axes $i - j - k$ constitute a left-handed coordinate system as opposed to the physical $lon - lat - z$ system being right-handed, see figure 1. For an oceanographer this might not seem to be the most intuitive choice of axis orientation, but a mathematician is used to having row 1, column 1 at the upper left corner and to him/her the choice is more obvious. Care should therefore be taken during development of e.g. finite difference code.

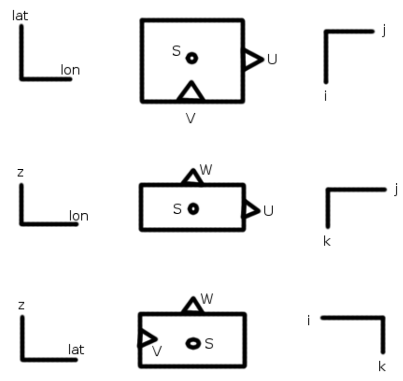


Figure 1: Staggered grid and projection onto coordinate planes in the $lon - lat - z$ and $i - j - k$ systems.

The grid cell node, also known as the grid point, is placed at (i, j, k). Most scalar quantities (like S and T) are located at the grid cell node and their value represent the value of the respective quantity within the grid cell.

The velocity components are located half-way between adjacent grid cell nodes on the face of the grid cell in the respective direction. That is, the components of the velocity vector (u, v, w) have spatially staggered locations. The positive directions of the velocity components follow the respective coordinate axis: For grid cell (i, j, k) the zonal component, u at (i, j, k), is located on the east cell face and is positive towards east; the meridional component, v at (i, j, k), is located on the south face and is positive towards north; and the vertical component, w at (i, j, k), is located on the upper face and is positive upwards. The figure 1 attempts to sketch these relationships by projection of the computational grid cell onto the three perpendicular, coordinate planes.

2.3 Memory module

The application is a 64/32-bit application, i.e 64 bit reals and 32 bit integers. All variable declarations are explicit with respect to kinds for reals and integers, i.e. we always explicitly declare reals as `real(8)` and integers as `integer(4)`.

We have developed a module for memory allocation, especially suited for nested variables. This module, called `cmod_mem`, is based on abstract data types containing pointer arrays which allow for allocating arrays with rows of different type and rank. This is required since nested areas do not in general have the same size. Names of predefined type definitions follow the notation: `cmTD` where `T` is a type identifier:

```
  T = i for integer(4)
  T = r for real(8)
  T = l for logical
```

and `D` identifies the dimension (shape) of the array:

```
  D = 1 for 1D arrays
  D = 2 for 2D arrays
  D = 3 for 3D arrays
```

Each of these 9 predefined abstract data types has one member called `p`, which is a pointer array of type `T` and rank `D`, e.g. the `cmr2` has the following type definition

```
  type cmr2
    real(8), pointer :: p(:,:)
  end type
```

Please note, that global arrays are allocated in the beginning of the program cf. module procedures `AllocPointers` and `AllocArrays` in `cmod_arrays`.

2.4 Wet-point compression

The grid contains both *wet-points*, points beneath the sea bed and true *land-points*, i.e. the $2D$ and $3D$ matrices that should hold the state variables would indeed be pretty *sparse*. This raises the obvious question of whether to use direct or indirect storage and a back-of-the-envelope estimate seems appropriate: Assume that we need to store say 50 `real(8)` matrices in a $500 \times 500 \times 150$ grid. This alone requires $\approx$ 14GB if using direct storage and only $\approx$ 2GB with 15% wet-points, typically of many real applications. Thus, the model uses indirect addressing for all the major data-structures implying that one saves a lot of runtime memory, the price being increased internal book-keeping. There are several ways to do the indirect addressing and we have chosen to use a variant of the *compressed row storage format* also sometimes know as *compression by gathering*.

For area `#ia`, the number of $3D$ wet-points is denoted by `iw3(ia)`. The grid index extremes in the area `#ia` are `mmx(ia)`, `nmx(ia)`, and `kmx(ia)`. The number of $2D$ wet-points, i.e. the number of wet-points confined to the surface layer is denoted by `iw2(ia)`. These are declared and assigned during model initialization in module `cmod_params`.

Below you will find specifications for one of our main setups today. It is a four-domain nested setup, consisting of a 3 n.m.[10] North Sea, a 0.5 n.m. Inner Danish Waters domain, a 1 n.m. German Bight / Wadden Sea domain, and a 3 n.m. Baltic Sea domain. The North Sea and the Baltic Sea domains are separated from each other: NS extends into Skagerrak to Skagen, BS extends eastward from Bornholm. Note that the ratio of wet-points in the largest sub-domains is only 11.1% so we are indeed saving a lot of memory using indirect addressing:
The indirect addressing used is simply a look-up table relating physical location, in terms of the index triplet (i, j, k), to memory location, in terms of a single array index. There is one look-up table for each nested area,

[10]nautical miles, 1 n.m. = 1852 m.

Domain	mmx	nmx	kmx	grid points	iw2	iw3	iw3/g.p.
NS	348	194	50	3375600	18908	479081	14.1%
IDW	482	396	75	14315400	80904	1583550	11.1%
WS	149	156	24	557856	11581	103441	18.5%
BS	248	189	109	5109048	13056	671985	13.1%

`ia`, which is declared and assigned during model initialization in module `cmod_params`:

```
m1(ia)%p(1:mmx(ia),1:nmx(ia),1:kmx(ia)).
```

These look-up tables return the value 0 for points on land or below the sea bed, and a positive integer, indexed `1,...,iw3(ia)`, for a point in the water. Thus, we have:

```
iw3(ia) = maxval( m1(ia)%p(:,:,:) )
iw2(ia) = maxval( m1(ia)%p(:,:,1) )
```

To simplify table look-up during e.g. finite differencing, additional tables are defined which extend the `m1` with a one point zero-padding at both ends of the horizontal coordinate directions and thus return the value 0 for points outside the domain. These index arrays are

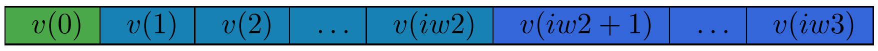
```
mm1(ia)%p(0:mmx(ia)+1,0:nmx(ia)+1,1:kmx(ia))
```

with

```
mm1(ia)%p(1:mmx(ia), 1:nmx(ia), : ) = m1(ia)%p(:,:,:)
mm1(ia)%p(    0     ,     :     , : ) = 0
mm1(ia)%p(    :     ,     0     , : ) = 0
mm1(ia)%p(mmx(ia)+1,     :     , : ) = 0
mm1(ia)%p(    :     , nmx(ia)+1, : ) = 0
```

2.5 Global arrays

Most variables are allocatable real arrays of `kind 8`, being allocated during the initialization process. Each $3D$ variable is stored as rank 1 arrays, one for each nested area, with indices running from 0 (the dummy on-land value) to the number of wet-points. Likewise, each $2D$ variable is stored as rank 1 arrays, one for each nested area, with indices running from 0 to the number of surface wet-points. Note that for $3D$ variables we always have the surface points stored as the first $1 \ldots$ `iw2(ia)` points, cf. the figure below where green denotes an inactive point (landpoint, point beneath the sea bed, point outside the domain), light blue indicates an active surface point and dark blue an active deep water point:



Variables usually have descriptive short names. As an example, the zonal current in area `#ia` is

```
u(ia)%p(0:iw3(ia))
```

and it may be accessed through indirect addressing like

```
u(ia)%p(m1(17,89,1))
```

for the value in the surface layer at horizontal position (i, j) =(17,89).

Exceptions are, for instance, temperature and salinity, which are referring to separate columns of a `real(8)` component array, `cmp`, with `ntracers` columns (`ntracers`≥ 2 is the total number of tracers), i.e. in grid area `#ia` we have the water temperature in `cmp(ia)%p(1,0:iw3(ia))` and the salinity in `cmp(ia)%p(2,0:iw3(ia))`. In appendix A we list the most important global variables, that is all the prognostic variables and the major diagnostic variables, meteorological forcing variables, index variables, and variables used for open boundary conditions and for nesting conditions.

The top layer, $k = 1$, deserves some special attention since a number of processes (wind forcing, heat exchange, ice drift, etc.) are confined to the near-surface region. For simplicity, surface index arrays are introduced as

```
me1(ia)%p(:,:) = mm1(ia)%p(:,:,1)
```

Furthermore, the internal alignment is such that `me1(ia)%p(:,:)` and therefore also `mm1(ia)%p(:,:,1)` and `m1(ia)%p(:,:,1)`, assumes values 0 through `iw2(ia)`.

There are two more very useful index arrays; one relating the surface wet-point number to horizontal grid indices; one that gives the number of wet-points in the water column at a given surface location. The first of these is a rank-2 integer array, `ind(ia)%p(1:2,1:iw2(ia))`. The second is a rank-1 integer array, `kh(ia)%p(0:iw2(ia))`.

2.6 Cache layout

Up to this point we have carefully tried *not* to be explicit about which of the `iw3` points is actually the first, which is the second and so forth. The aim of this section is to describe the exact enumeration of the wet-points. The mapping has been chosen to reflect the way memory works in current processors. It should not come as a big surprise that the performance of applications like this is tightly related to how well the data access patterns match the inner workings of the memory hierarchy.

Before we look at the actual access patterns we summarize how a CPU accesses memory[11] and we use our local Cray XT5 system as a concrete example. The XT5 node at DMI is a 12-way cc-NUMA[12] node with 16GB shared memory. Each node has two sockets and each socket has an AMD opteron six-core Istanbul 2.4 GHz processor installed. Note that AMD caches are *exclusive caches* (as opposed to say Intel) so the cacheline size is fixed for all cache levels. An exclusive cache implies that data is guaranteed to be in at most one of the L1, L2 and L3 caches. When the L1 misses and the L2 hits on an access, the hitting cache line in the L2 is exchanged with a line in the L1 and so forth. In this particular case the cacheline size is 64bytes and there are 3 cache levels with L1 and L2 being private to the core and L3 shared among the cores. Some details are shown below:

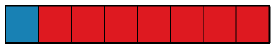
```
L1: 64Kb 2-way set associative data.
  L1 latency is 3 core clocks ~ 3 clocks / 2.4 Gcycle/sec = 1.25 ns.
  L1 bandwidth is NOT symmetric.
  L1 LD bandwidth: 2x128bit LD/cycle ~ 32Bytes/cycle * 2.4GHz = 76.8 GB/s.
  L1 ST bandwidth:  2x64bit ST/cycle ~ 16Bytes/cycle * 2.4GHz = 38.4 GB/s.

L2: 512Kb, 16-way set associative.
  L2 latency is 3+9=12 core clocks ~ 12 clocks/2.4 Gcycle/sec = 5 ns.
  L2 bandwidth is 38.4 GB/s.

L3: 2Mb (shared), 32-way set associative.
  L3 latency is 3+9+33=45 core clocks~45 clocks/2.4 Gcycle/sec = 18.75 ns.
  L3 bandwidth is 32 GB/s per core.
  The quoted L3 bandwidth is per core, but each core is limited to its
  own port, so the bandwidth seen by a core is the same whether one core
  is active or six cores are active.

Main memory (ccNUMA):
 Component  DIMM FSB  frequency Transmission rate DIMM bandwidth Bandwidth
```

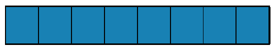
[11]Details can be found in classical textbooks such as [8].

[12]cache coherent Non-Uniform Memory Access, cf. [8].

```
DDR2-667   PC2-5300  333 MHz   667 MT/s          5.3 GB/s       10.6 GB/s
Read latency local socket     :   73 ns
Read latency remote socket    :  115 ns
```

The actual numbers stated above are not so important. The interesting part is their relationship. For instance, note that the latency is 1.25 ns for L1 access and 73 ns/115 ns for main memory. The worst-case usage of the cache system assuming `real(8)` elements (any of the 3 levels) is illustrated below (blue is used and red is unused) where we use only 1/8 elements in the cacheline. Note that the worst-case usage is losing a factor of 8 in memory latency performance as well as memory bandwidth performance compared to the best case usage shown just after the worst-case.

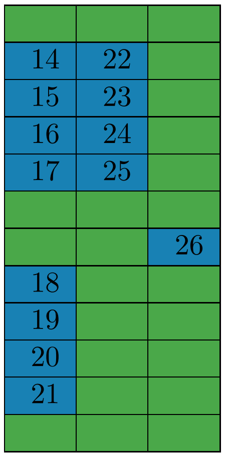




Now we are ready to investigate the way input data is actually stored:

```
mm_input(:,:,:) = 0
iw3 = 0
do k = 1,kmx
  do j = 1,nmx
    do i = 1,mmx
      if (is_wet(i,j,k)) then
        iw3 = iw3 + 1
        mm_input(i,j,k) = iw3
      endif
    enddo
  enddo
enddo
```

There are two obvious problems with this layout. First, the addressing of the index array `mm_input` itself will lead to the worst-case cache usage. Second, all vectors addressed indirectly by this index will have the worst-case cache usage. Let us try to illustrate this by showing the index numbers relating to $k = 2$, i.e. relating to the set of wet-points just beneath the surface. The model has `iw2`=13 wet-points in the surface, i.e. there are 13 water columns. The green points are inactive points whereas the blue points are active wet-points in this small example.



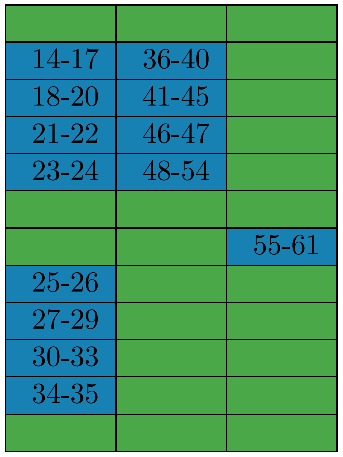
Note that the innermost k-loop will access element number 1,14,27,... in all the indirectly addressed vectors. However, we are free to permute the index in any way we like as long as we stick to the rule that the first `iw2` points constitutes the surface wet-points. This is the only constraint we have set up. If we insist on having the k-loop as the innermost loop then we need to permute the index arrays (`mm`, `ind`, `kh`) and all the input arrays u, v, T, S, etc. And we will need to permute the data so that the k-loop will access memory with stride-1 for $k > 1$, cf. below where we show all wet-points for $k > 1$. The innermost k-loop will access (1,14,15,16,17), (2,18,19,20), (3,21,22), ..., (13,55,56,57,58,59,60,61) in all the vectors. Splitting the innermost k-loop into a surface ($k = 1$) and a below-surface subloop ($k > 1$) allows for vectorization as we shall see later. The permutation of the index arrays and the corresponding input arrays can be found in `perm.f90`.

The explicit loop structure reflected below is another way to express this

```
do j=1,n
  do i=1,m
    do k=1,km
     ... u_permuted(mmk_permuted(k,i,j)) ...
    enddo
  enddo
enddo
```

One should be aware that in the above, `u_permuted` is accessed in all wet-points, `mmk_permuted > 0`, as well as in all land points, `mmk_permuted = 0`.

In the code itself we recommend a loop structure like the one below where



the outermost loop runs over all surface wet-points and the innermost loop runs down through each water column, cf.

```
surfacewetpointloop: do iw = 1,iwet2
  i = ind(1,iw)
  j = ind(2,iw)
  do k = 1,kh(iw)
    ! all wet-points (k,i,j) are reached here
     ... u_permuted(mmk_permuted(k,i,j)) ...
  enddo
enddo
```

This is re-structured for even better performance, by unrolling the surface layer $k = 1$, which then can be out-factored in a separate surface loop:

```
surfacewetpoints: do iw = 1,iwet2
  ! all surface wet-points (1,i,j) are reached here
  ! access here is stride-1, allowing vectorization
      ... u_permuted(iw) ...
enddo

surfacewetpointloop: do iw = 1,iwet2
  i = ind(1,iw)
  j = ind(2,iw)
  do k = 2,kh(iw)
    ! all deeper wet-points (k,i,j) are reached here
    ! access here is stride-1 allowing vectorization
     ... u_permuted(mmk_permuted(k,i,j)) ...
  enddo
enddo
```

Please notice that the innermost loops now are stride-1 which will allow for vectorization. This can be difficult to figure out in the last case, by the human developer and especially by the compiler, so it is recommended to give a little hint by re-writing the code to be more explicit on this fact:

```
surfacewetpointloop: do iw = 1,iwet2
  if (kh(iw) <= 1) cycle surfacewetpointloop
  i = ind(1,iw)
  j = ind(2,iw)
  do mi = mmk_permuted(2,i,j),mmk_permuted(kh(iw),i,j)
    ! all deeper wet-points (k,i,j) are reached here
    ! access here is CLEARLY stride-1
     ... u_permuted(mi) ...
  enddo
enddo
```

2.6.1 Theoretical musings

Admitted, it is not all loops that *only* access the points in the water column at hand. There are several loops that need to look at neighbouring water columns too, i.e. when the loop access element 18 in the compressed vector it also needs element number 41, and when it accesses element 19 it will also need to access element 42 and so forth. This raises the question as to whether or not it is worth considering another permutation than the one presented in the last section, i.e. a mapping τ of the set of subsurface wet-points, $\{iw2+1, iw3\}$, onto itself

$$\tau : \{iw2+1, iw3\} \mapsto \{iw2+1, iw3\}$$

in such a way that the cache access pattern is more efficient.

Before we try to answer this question we better recap a few simple facts about the implementation:

- There are innermost k-loops that do not look at neighbouring points
- There are innermost k-loops that need to access neighbouring points
- There are NO innermost k-loops that need to access neighbouring points whose lat/lon distance is more than 1 point away.

Thus, assume that we would like `mm(i-1,j,k)`, `mm(i+1,j,k)`, `mm(i-1,j-1,k)`, ..., `mm(i+1,j+1,k+1)` to give rise to consecutive values in the arrays they index so that we can access with stride-1. Can we find such a permutation?

Yes, since we can model these wishes using a sparse (wet-points x wet-points) matrix A. That is, $A(x, y) = 1$ if and only if we wish to have wet-point index X close to wet-point index Y and vice versa. If we assume that the matrix is symmetric, i.e. if wet-points X is close to Y then Y must be close to X, then the matrix maps directly into an undirected graph $G = (V, E)$ with vertices corresponding to the wet-points and the edges corresponding to the desire of index closeness. This is a classical problem whose solution, i.e. the *optimal* permutation, is given by a *sparse-matrix-ordering* algorithm. Now, if the closeness criteria really represents the ordering in which the program uses the index, then this approach will ensure an optimal cache usage but as mentioned in the beginning of the section the real problem is that the usage changes from one subroutine to another. Some will need that neighbouring points are close, e.g. `tflow.f90` that implements tracer *advection* and *diffusion*, whereas others will not need neighbouring points close and thus will suffer from such a permutation.

As the next section will show there is no current need for considering alternative permutations but we still find it relevant to keep these considerations in mind as the implementation evolves.

2.6.2 Cache profiling

Below you will find cache profiling on one of our most important test-cases running on the XT5 system today (the four-domain nested setup described earlier).

```
TLB utilization              3628.74 refs/miss      7.087 avg uses
D1 cache hit,miss ratios       99.2% hits            0.8% misses
D1 cache utilization (misses)  120.18 refs/miss     15.022 avg hits
D1 cache utilization (refills)  48.18 refs/refill    6.023 avg uses
D2 cache hit,miss ratio        26.0% hits           74.0% misses
D1+D2 cache hit,miss ratio     99.4% hits            0.6% misses
D1+D2 cache utilization       162.31 refs/miss     20.288 avg hits
System to D1 refill           19.659M/sec    102268075776 lines
System to D1 bandwidth      1199.908MB/sec  6545156849664 bytes
D2 to D1 bandwidth           420.654MB/sec  2294548102784 bytes
```

It seems reasonable to consider cache hits rates a function of cache size and the program's memory reference pattern, so for this test-case we can conclude that the working set size is fitting nicely in D1, but the remaining references are too big to fit into D2.

As for TLB[13] performance, note that the profiling above is with 4KB pages implying that we can have 512 `real(8)` elements per page. We see 3629 references (more than the 512) per miss, so we are almost certainly touching every byte in the page. Moreover, note that "avg uses" is greater than 1 and since the predominant memory access strides over all data in each page, it is likely accessing every word of the page.

It is always worth to consider if we can improve the TLB performance by changing the page size so let us do another back-of-the-envelope estimate and compute how many pages a $3D$ variable will span. For the IDW sub-domain each $3D$ variable will span $1583550/512 \approx 3093$ pages and so forth, cf. table 1.

Domain	**iw3**	**4KB pages**	**2MB pages**
NS	479081	936	3.7
IDW	1583550	3093	12.1
WS	103441	202	0.8
BS	671985	1313	5.1

Table 1: Number of pages required to store a $3D$ variable in each sub-domain.

Looking at this table it should be pretty obvious that going for larger pages is not the way to go for this test-case. If we do that then we will see a lot of remote memory references when running on multiple sockets and we will not be able to openMP scale the application, since a page is the smallest unit that we can place on the sockets.

[13]A translation look-aside buffer (TLB) is a cache that memory management hardware uses to store translations of virtual addresses.

3 Nesting

The word ”nesting” has many interpretations in the oceanographic community and elsewhere. Sometimes, ”nesting” is just stated without any further definition. In the following we will attempt to clarify what we mean by ”nesting” in the framework of HBM.

Often terms like ”coarse grid” and ”fine grid” are used to reflect the relative resolution of two domains. But in the context of nesting in HBM, this can be misleading because we have to deal with more nesting levels and because two domains exchanging information need not have different horizontal resolutions. A more correct way is referring to the domains as ”enclosing” or ”inclosing”.

Nesting in HBM is used as a practical means of setting up models with different demands of high horizontal resolution, high vertical resolution, large toplayer size and small time step size in different parts of the modelled region. With nesting it becomes feasible to run models operationally that would otherwise be too computationally demanding if one e.g. had to set up the model with the smallest grid spacing and the smallest time step size throughout the entire domain. Alternatives to nesting could in some situations be curvi-linear grids or unstructured grids, or even regular structured grids which are computationally decoupled from each other with transfer boundaries between.

3.1 Hydrodynamic nesting procedures

3.1.1 Fully dynamical two-way nesting

The HBM code is designed for any number of fully dynamical two-way nested areas exchanging mass and momentum between each other across their borders at the numerical time step level so as to obtain continuity of transports and thereby also conservation of mass. For practical reasons, we have limited the maximum allowed number of nested areas to 99; we have to date not worked with any setup having more than 12 nested areas and there is no *real operational* application running today which has more than 4 so we expect this number to be sufficient for a while, else it is a simple matter to increase it.

The enclosing and inclosing domains are ”glued” together across their mutual border:

- Nesting from the enclosing (or coarse) grid to the inclosing (or fine) grid uses momentum equation and enclosing grid velocity components to obtain inclosing grid velocity components at border.
- Nesting from inclosing (fine) grid to enclosing (coarse) grid uses inclosing grid velocity components to obtain enclosing grid velocity components just inside the border.

3.1.2 One-way nesting

Another, more simple way of nesting is by use of so-called fjord models. Here the nesting is one-way, from a coarser grid to a finer grid only. The fine grid domain (the fjord) is forced along its open lateral boundaries by prescribed water levels boundary conditions obtained from the coarse grid. There is no feed-back from fine grid to coarse grid.

The one-way nesting facility has, however, not been implemented yet; this is left for a future release of HBM.

3.1.3 Vertical nesting

Nesting from a horizontally coarse grid to a finer grid may also involve nesting to a vertically finer grid. If so, there must be an integer number of vertical fine grid layers inside each coarse grid vertical layer.

3.1.4 Time step levels

When nesting to finer grids, it might be useful to do computations with a smaller time step size. In HBM this is done by having different time step levels. When you go from one time level to next, the time step size is decreased by a factor of 2 if you have more than two nesting domains (this restriction might very well be subject to change in near future). If you have only two nested domains, the time step size may be decreased by any integer factor.

3.2 Computational flow of nested hydrodynamics

The module `cmod_hydrodynamics` wraps the solutions of the nested hydrodynamic equations. It uses recursion, since solution of nesting level `#level` requires solution of the next higher level `#level+1`:
The main area (area `#1`, hard-coded as `mainarea=1`, see subsection Nesting

Variables) is on time level `#1` (`level=1`), defines the master time step size (which again defines the computational cycle) and should make one time step evolution only once per computational cycle. There can be more domains on time step level `#1`.

If there is more than two time step levels, the next subsequent time step levels have the following structure:
On time step level `#2`, which has half the time step size, we run through two time step evolutions per computational cycle (which is also twice pr evolution on the previous level `#1`). Then, on time step level `#3`, which has half the time step size compared to level `#2`, we run through two time step evolutions per previous level (which is 4 times per computational cycle). And so on.

If there is only two time step levels, the time step sizes can be chosen more freely:
The time step size of the main area is an integer multiple of the time step size of time step level `#2`.

In short, this amounts to the following pseudo-code flow chart which starts from the main program by solving for the lowest time step level, which is the time step level of the main area

```
call SolveHydrodynamics ( timelevel(mainarea) )
```

i.e. nesting outside-in. The subroutine `SolveHydrodynamics` goes like this:

```
subroutine SolveHydrodynamics ( level )
  !  run through the partial time levels at the
  ! present time nesting level
  do itl=1,max_iterations_on_this_level
    !  solve momentum equations on this level
    do ia=1,narea
      if (.not.onthislevel(ia,level)) cycle
      call SolveMomEq (ia,itl)
    enddo
    !  solve the next level hydrodynamics
    call SolveNextLevel ( level+1 )
    !  solve the mass equations on this level
    do ia=1,narea
      if (.not.onthislevel(ia,level)) cycle
      itn = timelevelsize(ia)
      call SolveMassEq (ia,itl,itn)
    enddo
  enddo
end subroutine SolveHydrodynamics
```

and the solution of the next level is simply:

```
subroutine SolveNextLevel ( level )
  if (level <= maxtimelevels) then
    call SolveHydrodynamics ( level )
  endif
end subroutine SolveNextLevel
```

In the above, the domains are enumerated by `ia` from `1` to `narea`. This enumeration is used as indices to look-up the relevant data-structures for the domain at hand. The spatial and temporal resolutions associated with each of these domains are user-specified through configuration files read in during model initialisation.

Spatial nesting levels and time step levels need not be the same. The constraint made here is that we select one domain, the main area, which is on nesting level `#1` and time step level `#1`. Only this main area can have open lateral boundaries.

You can nest to area(s) on a higher spatial nesting level (with any positive integer factor between spatial resolution grid spacings) and/or a higher time step level.

Possibly, at a later stage we will make the model even more general by relaxing those restrictions. It can be done, but it requires some more careful design. But at the moment a further generalisation is not justified.

We expand the above-shown pseudo-code a little to put more focus on where the actual nesting of different prognostic hydrodynamic variables takes place. First, in `SolveMomEq`, we solve the momentum equations of area `#ia` in the interior of the domain i.e. updating `u,v` to `un,vn`, and then we can obtain the velocity components near the outer borders of domain `#ia` by solving the momentum equations for `un,vn` with latest velocity components from enclosing domain(s) `#iia` prescribed at the borders:

```
subroutine SolveMomEq ( ia, ... )
  !  solve momentum equations for interior points:
  call momeqs( u(ia)%p, v(ia)%p, un(ia)%p, vn(ia)%p, ... )

  !  fix velocity by nesting:
```

```
  do iao=1,nestingfrom(ia)%p(0)
    !  ia: inclosing domain, iia: enclosing domain
    iia = nestingfrom(ia)%p(iao)
    call mom_c_f( u(ia)%p, v(ia)%p, un(ia)%p, vn(ia)%p, ...  &
                  un(iia)%p, vn(iia)%p )
  enddo
end subroutine SolveMomEq
```

Then, in `SolveMassEq`, we use the newest `un,vn` of present area `#ia` to obtain `un,vn` of any possible enclosing domain `#iia` at cell faces just inside the region covered by the inclosing area `#ia`; this is done by simple conservation of cell face fluxes. Since we at this stage of the computational cycle have all velocity components of present domain `#ia` updated, we can solve the mass equations in the interior of the present domain, updating `z` to `zn` and also updating diagnostic variable `w`. With `zn(ia)` in place we can distribute the solution to the enclosing domains in `copy_f_g_sfc`. Finally, we can update `zn(ia)` and `w(ia)` along the border points using mass equation with fine grid velocities at interior cell faces and coarse grid velocities at exterior cell faces.

```
subroutine SolveMassEq ( ia, ... )
  !  fix velocity along border:
  do ii=1,nestinglevels(ia)
    ! iia: enclosing domain, ia: inclosing domain
    iia = nestingto(ia)%p(ii)
    call mom_f_c ( un(ia)%p, vn(ia)%p, un(iia)%p, vn(iia)%p, ... )
  enddo

  !  solve mass equations for zn and w at interior points:
  call masseqs( z(ia)%p, zn(ia)%p, w(ia)%p, un(ia)%p, vn(ia)%p, ... )

  !  distribute z to enclosing domains:
  do ii=1,nestinglevels(ia)
    ! iia: enclosing domain, ia: inclosing domain
    iia = nestingto(ia)%p(ii)
    call copy_f_g_sfc ( zn(ia)%p, zn(iia)%p, ... )
  enddo

  !  update z and w in fine grid along border:
  do iao=1,nestingfrom(ia)%p(0)
    !  ia:  fine grid,  iia: coarse grid
    iia = nestingfrom(ia)%p(iao)
    call rand_z( zn(ia)%p, z(ia)%p, un(ia)%p, vn(ia)%p,           &
                 un(iia)%p, vn(iia)%p, ... )
    call w_c_f( w(ia)%p, zn(ia)%p, z(ia)%p, un(ia)%p, vn(ia)%p,   &
                 un(iia)%p, vn(iia)%p, ... )
  enddo
end subroutine SolveMassEq
```

3.3 Nesting variables

Most of the nesting is defined though the basic parameters set up in module `cmod_params`. All of these are configurable from user input, except these two which have predefined values:

```
integer(4), parameter, public :: maxareas = 99 ! Max No. of areas
integer(4), parameter, public :: mainarea = 1  ! No. of the main area/domain
```

Other simple nesting variables describe the number of two-way nested areas and one-way nested domains (so-called fjord models), as well as the total number of domains:

```
integer(4), save, public :: narea      ! No. of dynamically two-way nested areas
integer(4), save, public :: nfjord     ! No. of one-way nested fjord models
integer(4), save, public :: ntotal     ! Total No. of nested domains
```

Obviously, the last is the sum of the other two. The one-way nesting facility has, however, not been implemented yet; this is left for a future release of HBM.

The `cmod_params` module contains the following arrays storing the definitions of the spatial and temporal nesting:

```
logical,    allocatable, save, public :: onthislevel(:,:)
integer(4), allocatable, save, public :: timelevel(:), nestinglevels(:)
integer(4), allocatable, save, public :: enclosing(:,:), timelevelsize(:)
type (cmi1),    pointer, save, public :: nestingto(:), nestingfrom(:)
type (cmi2),    pointer, save, public :: kfg(:), iga(:), jga(:)
type (cmi2),    pointer, save, public :: znesting(:), unesting(:), vnesting(:)
```

which we shall attempt to document in the following.

The four arrays `timelevel(:)`, `timelevelsize(:)`, `nestinglevels(:)` and `nestingto(:)%p(:)` are initialised according to user-specifications. Attempt is made to have them "sanity-checked" to avoid the most obvious specification errors, but there is no guarantee that cases can occur that make the bookkeeping of the nesting crash.

```
timelevel(ia) :        identifies the time step nesting level for area #ia; it
                       should equal 1 for the coarse grid (mainarea) with the
                       largest time step and for those domains on the same
                       time-level. A higher value means higher time-nesting
                       level (i.e. smaller time step).
timelevelsize(ia) :    the No. of subdivision of time step size for area #ia
                       compared to the time step size of the main area.
nestinglevels(ia) :    gives the number of immediately inclosed nesting areas
                       to area #ia, i.e. how many domains does area #ia nest
                       to. It should be zero for domains on the highest
                       nesting level.
nestingto(ia)%p(nl) :  for nl in {1:nestinglevels(ia)} this gives the number
                       of the area(s) that area #ia is nesting to.
```

It is useful to define other look-up tables to keep track of the nesting:

```
is area #ia on time level #itl:                 onthislevel(ia,itl)
order of area #iia enclosing area #ia:          enclosing(ia,iia)
No. of areas nesting from area #ia:             nestingfrom(ia)%p(0)
areas nesting from area #ia in given order:     nestingfrom(ia)%p(1:)
```

i.e. if area `#iia` is nesting to area `#ia`, the value of the simple index variable:

```
iia
```

equals the value of the rather complex construction:

```
nestingfrom(ia)%p(enclosing(ia,iia))
```

The last six arrays are used for storing the grid indices of the nesting.

```
iga(iia)%p(ir,iao) :   with fine grid area #iia inclosed into a coarse grid
                       area of order iao, this is in the north -> south
                       direction

                       for ir = 1 : the index of the fine grid origin in
                                    coarse grid coordinates

                       for ir = 2 : the number of fine grid points in one
                                    coarse grid point; usually this number is
                                    even

                       for ir = 3 : displacement of fine grid relative to
                                    coarse grid; usually
                                    iga(iia)%p(3,iao) = iga(iia)%p(2,iao)/2

jga(iia)%p(ir,iao) :   same as iga(:)%p(:,:) but for the west -> east index
```

```
kfg(ia)%p(k,ii)     :   for coarse grid #ia and fine grid at nesting level #ii,
                        this gives the number of fine grid vertical layers
                        inside coarse grid layer #k.

znesting(iia)%p(iao,1:2) : first and last index of the border of fine grid
                           #iia inclosed in coarse grid of order ioa.

unesting(ia)%p(ii,1:2)   : first and last index of the u-border of coarse
                           grid #ia nesting to fine grid level #ii.

vnesting(ia)%p(ii,1:2)   : first and last index of the v-border of coarse
                           grid #ia nesting to fine grid level #ii.
```

In the above, please note that

```
ii = 1:nestinglevels(ia)
```

and an easy way to obtain `iao` is through

```
iao = enclosing(ia,iia)
```

3.4 Nesting details

Here we attempt to describe in detail some implementation features of the nesting in HBM. To follow the computational flow of nested hydrodynamics, have a look at the pseudo-codes presented in section 3.2.

The following border types are defined in the variables `krz, kru`, and `krv`, see table 27, for the respective border:

```
unknown border type:  < 0, no action
not a border:         0, no action
W border:             1
N border:             2
E border:             3
S border:             4
```

That is, on a z-border the type (W/N/E/S) is seen from inclosed/fine grid, while for u- and v-borders the type is as seen from the enclosing/coarse grid.

3.4.1 Momentum

The nesting from enclosing grid to the inclosed grid uses momentum equation and velocity components of the enclosing grid to obtain velocity components of the inclosed grid at the border. The z-border points (as seen from the inclosed/fine grid) along each border must be enumerated in a specific order in increments of 1:

```
W: from S to N
N: from W to E
E: from N to S
S: from E to W
```

Each border must be fully connected in the specifications. There can be more than one border of each type. Convex corners (as seen from inclosed grid) need special treatment. The corner point (in inclosed grid) must be defined on **both** of the joined borders, enumerated by an increment of 1 clockwise, eg for a W/N corner, where the W border ends at (say) border-point No. 7, the N border must start at border-point No. 8. Concave corners (as seen from inclosed grid) need no special treatment with the present method.

The nesting from inclosed grid to enclosing grid uses velocity components of the inclosed grid to obtain velocity components of the enclosing grid just inside the border. The u and v cell faces (as seen from the enclosing/coarse grid) along each border must be enumerated in a specific order in increments of 1:

```
E: from S to N
S: from W to E
W: from N to S
N: from E to W
```

There can be more than one border of each type. Each border must be fully connected in the specifications. The u- and v-borders must be specified such that all momentum points of the enclosing grid one point inside the border are covered with start and end either at a land point or continuing at/from another u- or v-border of another border type (i.e. a corner). Corner points must not be duplicated to model nesting corners, i.e. concave corners points (as seen from enclosing grid) should NOT appear in the specification since this would complicate this nesting procedure more than intended, and convex corners (as seen from enclosing grid) should be specified in only one of the joining borders.

The algorithm can be sketched as follows. With subscript f denoting the inclosed (or fine) grid and subscript c denoting the enclosing (or coarse) grid, we have at any point:

grid spacing factor:	$F = dx_f/dx_c$
area of cell face:	$A_c = A_f = A$
transport across cell face:	$T_c = T_f = T$

As seen from the inclosed grid, we have

$$\begin{aligned} \text{area:} \quad & A = \sum_{i=1}^{N} h_{f,i} dx_f \\ \text{transport:} \quad & T = \sum_{i=1}^{N} h_{f,i} u_{f,i} dx_f \end{aligned}$$

with a $1 : N$ nesting ratio in the considered direction (x as an example) and h denoting the cell face height. As seen from the enclosing grid, we can then assign

$$\begin{aligned} h_c &= A/dx_c = F\sum_{i=1}^{N} h_{f,i} \\ u_c &= T/A = T/(h_c dx_c) = \sum_{i=1}^{N} h_{f,i} u_{f,i} dx_f / \sum_{i=1}^{N} h_{f,i} dx_f \end{aligned}$$

implying continuity of transports and thereby also conservation of mass.

3.4.2 Mass

Nesting of water level from enclosing grid to inclosed grid is performed by solving the fine grid mass equations with enclosing grid velocity components prescribed at the border faces of the grid cells.

When levels have of the inclosed grid have been obtained, we extract these values where they (partially) cover a grid cell in the enclosing grid, make a grid cell averaging and assign at the enclosing grid points

$$z_c = \sum_{i=1}^{NM} z_{f,i} dx_{f,i} dy_{f,i} / \sum_{i=1}^{NM} dx_{f,i} dy_{f,i}$$

assuming a $1 : N$ by $1 : M$ nesting.

3.4.3 Tracers

Nesting of tracers, i.e. salinity and temperature, and passive tracers from e.g. biogeochemical models, is best described by describing the major computational flow including nesting procedures. The following pseudo-codes are snipped from inside the time loop, after we have updated `u,v,z` in `SolveHydrodynamics`. It starts by performing nesting of the tracers followed by tracer advection:

```
!  Tracer nesting:
do ia=1,narea
  do ii=1,nestinglevels(ia)
    !  iia: fine grid  ia:  coarse grid
    iia = nestingto(ia)%p(ii)

    !  Tracer boundary values coarse grid-->fine grid:
    !  step 1: find coarse grid values just outside border:
    call bndstz( cmp(ia)%p, bndz(iia)%p, ... )
    !  step 2: advect boundary values at inflow points:
    call bndzst( cmp(iia)%p, bndz(iia)%p, ui(iia)%p, vi(iia)%p, ... )

    !  Tracer boundary values fine grid-->coarse grid:
    !  step 1: coarse grid S/T by cell-averaging from fine grid
    !          (performed in copy_g_f during previous time step).
    !  step 2: advect boundary values at inflow points (f-->c):
    call bnduvst( cmp(ia)%p, cmp(iia)%p, ui(ia)%p, vi(ia)%p, ... )
  enddo
enddo

!  Run through all areas and do advection
do ia=1,narea
  call tflow( cmp(ia)%p, ui(ia)%p, vi(ia)%p, w(ia)%p, DoAdvec, ... )
enddo
```

Then, before we can perform diffusion, we need to nest the results of the tracer advection step from the inclosed grid to the enclosing grid near the borders, and after we have finished the calculation of diffusion we can extract the inclosed grid values to the enclosing grid where the two grids overlap:

```
!  do tracer nesting 'coz it was likely modified during advection:
do ia=1,narea
  do ii=1,nestinglevels(ia)
    !  iia: fine grid,  ia:  coarse grid
    iia = nestingto(ia)%p(ii)
    !  Tracer boundary values fine grid-->coarse grid:
    call brdcopy( cmp(ia)%p, cmp(iia)%p, ... )
  enddo
enddo

!  do the diffusion:
do ia=1,narea
  call tflow( cmp(ia)%p, dispv(ia)%p, eddyh(ia)%p, DODiff, ... )
enddo

do ia=1,narea
 do iao=1,nestingfrom(ia)%p(0)
   !  ia:  fine grid,  iia: coarse grid
   iia = nestingfrom(ia)%p(iao)
   !  do fine-->coarse grid copy
   call copy_f_g ( cmp(ia)%p, cmp(iia)%p, ... )
  enddo
enddo
```

The extraction performed in `brdcopy` and in `copy_f_g` is similar to what is done in `copy_f_g_sfc` and described in the previous subsection for water level, except that we need to take into account the vertical grid refinement, too:

$$T_c = \sum_{i=1}^{NMK} T_{f,i} h_{f,i} dx_{f,i} dy_{f,i} / \sum_{i=1}^{NMK} h_{f,i} dx_{f,i} dy_{f,i}$$

where T signifies the grid point value of any tracer, and we assume a $1:N$ by $1:M$ by $1:K$ nesting at the considered coarse grid point.

If there are any processes involved other than advection/diffusion, we must repeat the `copy_f_g` for the relevant tracers after such processes have finished. These processes are thermodynamics which affect the temperature tracer, and the biogeochemical processes which affect the passive biogeochemical tracers.

3.5 Setting up nested models

The general model setup is specified through the HBM configuration file called `cfg.nml` which contains `NAMELIST`s. In the first, called `cfglist`, you specify the number of two-way nested areas. In the second and subsequent, one for each domain, all called `cfgfn`, you give the file names for the grid specification file and for the bathymetry file as well as information on time level and nesting. Thus, a case with two domains is specified with three `NAMELIST`s as follows:

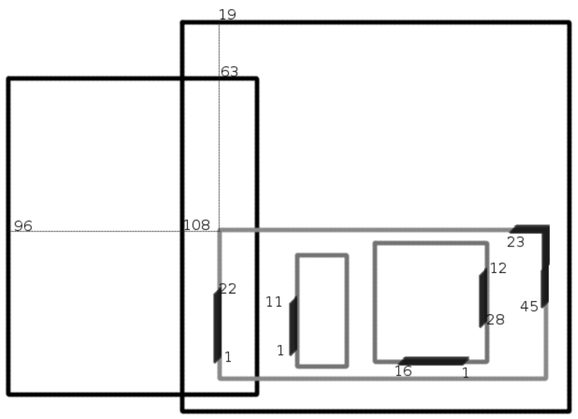
```
&cfglist
  narea = 2
/
&cfgfn
  cfgfile_in = "data_coarse"
  bathyfile_in = "coarse_grid_depth"
  timelevel_in = 1
  nestinglevels_in = 1
  nestingto_in = 2
/
&cfgfn
  cfgfile_in = "data_fine"
  bathyfile_in = "fine_grid_depth"
  timelevel_in = 2
  nestinglevels_in = 0
  nestingto_in = 0
/
```

The origin of a specific inclosed sub-domain with respect to ALL the enclosing domains nesting to that specific domain must be specified together with the nesting ratios and grid displacements. This is done in the respective `cfgfile_in` file.

In case there are more than two domains, you need to specify which parts of the z-, u- and v-borders that are nesting to/from which sub domain. This is done at the bottom of the respective `cfgfile_in` file. First, for each finer, inclosed domain, you specify which z-border points are nesting from which enclosing areas. Then, for each enclosing domain, you specify which uv-border points are nesting to which inclosing areas.

In the following we give some examples to clarify various issues in setting up nested models.

3.5.1 Defining borders



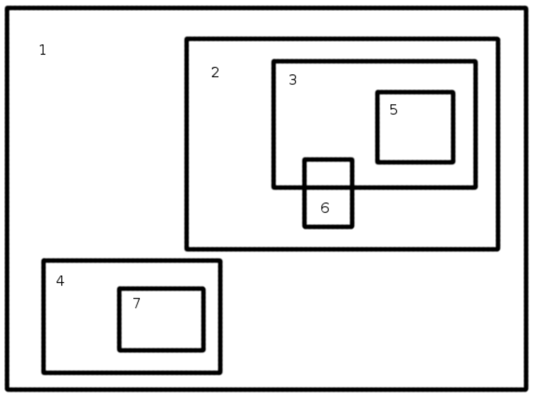
The figure above is an example that demonstrates nesting borders as seen from enclosing domains and from inclosed domains. Thus focus here is on the sub-domain that covers two inclosed domains and which itself is inclosed into two other domains. That is, this particular sub-domain is nesting from two enclosing domains and is nesting to two inclosed domains. Its origin is

(96,63) in the left enclosing grid and (108,19) in the enclosing grid to the right. In both cases, we have a 1:6 nesting with a -3 displacement. The subdomain has 45 z-border points, where the first 22 are nesting from enclosing area of order 1 and the rest (23-45) is nesting from the next enclosing area. The nesting to inclosed area of order 1 is through the first 11 u-border points, while nesting to inclosed area of order 2 is through the 17 u-border points (numbers 12-28) and through 16 v-border points. This would be specified like this:

```
============================================================
 Origin of fn grid in enclosing crs grid    I    J
 --------------------------------------------------
 z(1,1) is located in row(I), column(J):   96   63
 No. of fn grid cells in crs grid cell :    6    6
 Displacement of z(1,1) relative to crs:   -3   -3
 z(1,1) is located in row(I), column(J):  108   19
 No. of fn grid cells in crs grid cell :    6    6
 Displacement of z(1,1) relative to crs:   -3   -3

============================================================
 Z-borders                                nz1  nz2
 -----------------------------------------------------------
 Nest from order 1 at these:                1   22
 Nest from order 2 at these:               23   45

============================================================
 UV-borders                               nu1  nu2  nv1  nv2
 -----------------------------------------------------------
 Nest to order 1 at these:                  1   11    0    0
 Nest to order 2 at these:                 12   28    1   16
```

3.5.2 Other aspects

In the previous subsection we saw how to define the border points for $u-$, $v-$ and z-borders. In this subsection, we focus on defining the nesting hierarchy. We treat a nested configuration with seven domains and a layout of the domains as sketched in the figure below.



The main area, #1, is on time level #1. The main area nests to areas #2 and #4 which are both on time level #2. Area #2 encloses area #3 which is on time level #3. Area #5 is fully inclosed into are #3, while area #6 is nesting the two areas #2 and #3. Both area #5 and #6 are on time level #4. Finally, area #4 has one inclosed area, #7, which is on the same time level, #3, as area #3.

This configuration can be specified in the `cfg.nml` as:

```
&cfglist
  narea = 7
/
&cfgfn
  cfgfile_in = "data_1"
  bathyfile_in = "grid_1"
  timelevel_in = 1
  nestinglevels_in = 2
  nestingto_in = 2, 4
/
&cfgfn
  cfgfile_in = "data_2"
  bathyfile_in = "grid_2"
  timelevel_in = 2
  nestinglevels_in = 2
  nestingto_in = 3, 6
/
&cfgfn
  cfgfile_in = "data_3"
  bathyfile_in = "grid_3"
  timelevel_in = 3
  nestinglevels_in = 2
  nestingto_in = 5, 6
/
&cfgfn
  cfgfile_in = "data_4"
  bathyfile_in = "grid_4"
  timelevel_in = 2
  nestinglevels_in = 1
  nestingto_in = 7
/
&cfgfn
  cfgfile_in = "data_5"
  bathyfile_in = "grid_5"
  timelevel_in = 4
  nestinglevels_in = 0
/
&cfgfn
  cfgfile_in = "data_6"
  bathyfile_in = "grid_6"
  timelevel_in = 4
  nestinglevels_in = 0
/
&cfgfn
  cfgfile_in = "data_7"
  bathyfile_in = "grid_7"
  timelevel_in = 3
  nestinglevels_in = 0
/
```

There is some help to get. By switching on the namelist parameter `ldebug` in the namelist called `optionlist` in the file `option.nml`, i.e.:

```
&optionlist
  ldebug = .false.
/
```

you will get a lot of printouts when running HBM, some of which might prove very useful when validating your model setup. In particular, you can validate that your nested model setup is interpreted as intended. For the seven area case shown above, you can get the explanation shown below from which you may convince yourself that the domain hierarchy is as expected (left as an exercise for the reader to enjoy :)).

```
 Area
 sub level, domain, on time level
           1
           1            2            2
```

```
          1
          2           4           2

          2
          1           3           3

          2
          2           6           4

          3
          1           5           4

          3
          2           6           4

          4
          1           7           3

Area:            1
  nesting to:            2           4
  not nesting from any enclosing domain.
Area:            2
  nesting to:            3           6
  nesting from:          1
Area:            3
  nesting to:            5           6
  nesting from:          2
Area:            4
  nesting to:            7
  nesting from:          1
Area:            5
  not nesting to any sub domain.
  nesting from:          3
Area:            6
  not nesting to any sub domain.
  nesting from:          2           3
Area:            7
  not nesting to any sub domain.
  nesting from:          4

Loop inside-->out
  fine grid, coarse grid
          6           2
          6           3
          5           3
          7           4
          3           2
          4           1
          2           1
```

4 Parallelization

Before we describe the different approaches we present our initial design goals:

- The code will run in serial (and we must be able to build it on systems that have *no* support of MPI, openMP or openACC).
- The code will run with openMP solely (and we must be able to build it on systems that have *no* support of MPI and openACC). It must be possible to run it with a single thread.
- The code will run with MPI solely (and we must be able to build it on systems that have *no* support of openMP and openACC). It must be possible to run with a single MPI task too.
- The code will run with MPI and openMP. This is the default way of running HBM (and we must be able to build it on systems that have *no* support of openACC). It must be possible to run with a single MPI task and/or a single thread.

The actual decomposition into a number of openMP threads and/or MPI tasks must be user-configurable at runtime. We do not want to re-compile just because we choose to run with a different number of threads and/or tasks: Each new compilation produces a new executable code which in principle must be tested, so re-compilation means re-validation, and an endless loop has begun.

Moreover, the application must be able to run in all configure incarnations serial, openMP, MPI, and combinations hereof - and produce the exact same results on any given test-case with any number of openMP threads and MPI tasks. The experimental openACC port is a bit more involved when it comes to cross comparisons as explained in details in the openACC subsection.

In the following subsections we describe how the different parallelization paradigms have been introduced into the code. We start the survey by describing our considerations on geometric decomposition for the irregularly shaped domains that we are generally dealing with. Then we proceed with a short note on vectorization, followed by in-depth description of our work on openMP, MPI and openACC.

4.1 Geometric decomposition

We have used the *geometric decomposition pattern* to split the overall problem into smaller sub-problems. For openMP we split the overall problem either into sets of horizontally irregular $1D$ chunks or irregular $2D$ sets of water columns (depending on whether or not the subroutine at hand operates in $2D$ or $3D$). For MPI we split the overall problem into regular latitude-longitude sub-rectangles.

For many computational problems the task of splitting the problem into a balanced set of subproblems is trivial. For this particular problem this is far from true. We illustrate this in figure 2 where we have plotted the IDW subdomain pertaining to the usual test-case combined with histograms: The colour coding shows the number of wet-points below each surface points, white is on land, and the colour scale runs from dark blue for 1 point to dark red for 75 points. The histogram to the right shows the distribution of number of wet-points along each zonal (i.e. constant latitude) grid line. The upper histogram shows the distribution of number of wet-points along each meridional (constant longitude) grid line. The problem at hand is how to define a decomposition into sub-domains for real examples like this one, to meet criteria like each sub-domain should have approximately the same number of wet-points, the same number of neighbour water columns, the same distribution of column lengths within each domain, etc. It is worth mentioning that this problem is much more relavant for the regional models with very fine resolution than it is to the global ocean models. The resolution in the global models is so coarse that the domain looks more like a bathtub with around 50-60% of the grid-points being wet-points, cf. appendix C.

Before we describe the actual decomposition strategies we present the problem of geometric decomposition in a more general fashion. It can formally be stated as an optimization problem with constrains:

The input is described by a $2D$ grid defined by the surface points $(p_1, ..., p_M)$ with corresponding weights $(w_1, ..., w_M)$. The weights are simply defined as the number of wet-points in the vertical direction below each surface point p_i (i.e. the k-dimension). This grid must be covered by N subsets $r_1, ..., r_N$ with weights v_i such that the balance score C is minimized:

$$v_i = v(r_i) = \sum_{p_j \in r_i} w_j$$

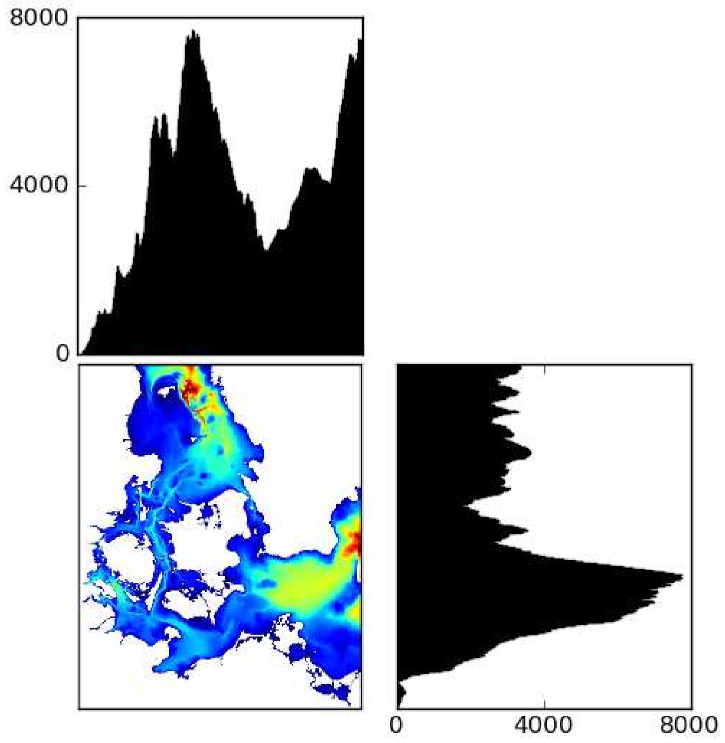


Figure 2: Illustration of how irregular the IDW sub-domain is in terms of 3D wet-points. See text for explanation.

$$C = \max_{1 \le i \le N} v_i - \min_{1 \le i \le N} v_i$$

Note that this is a classical *Knap-Sack problem*, cf. [4] and consequently **NP-hard**, cf. [2]. Below we present an alternative *set-partitioning* formulation of (2) above :

$$\min_{1 \le i,j \le N} |v_i - v_j| \quad s.t. \quad Ax = 1, x \in \{0,1\}^N$$

That is, each column in the matrix A corresponds to a possible subset and each row corresponds to a point in the grid. The coefficients a_{ij} tell whether or not point p_i belongs to subset r_j.

For practical purposes it is often more convenient to describe the balance score in terms of a quotient instead of a difference:

$$C^* = \frac{\max_{1 \leq i \leq N} v_i}{\min_{1 \leq i \leq N} v_i}$$

Note that the general decomposition problem described above where we basically consider all water columns as being totally independent is very idealized. It does not deal with any of the practical issues that one has to take into account when doing the implementation using one of the parallel models, e.g.

- It does not take water column dependencies into account, i.e. neither implicit nor explicit halo communication is taken into account.
- It does not take communication related to nesting, boundary conditions or IO into account.
- It does not take irregular dynamics on top of the overall geometry into account, e.g. the ice dynamics is typically not active in all wet-points but present in geographically isolated sub-areas only.
- It does not take inactive points into account. That is, it assumes that the work required to handle 150 water columns in shallow area (say all with a single layer) should be equivalent to the work required to do a single water column with 150 layers, which most likely will not be the case.

Even if the problem was solvable in theory it would still be quite complicated in practice, handling the book-keeping of the halo-regions stemming from an irregular, but optimal decomposition. Let us look at the bright side: The fact that we cannot solve it in theory relieves us from the burden of trying to deal with these practical problems :). Instead we will focus our attention on implementing various *heuristics* and then measure (using relevant test-cases) whether or not they seem sufficient for our purpose. That is, we will follow the same pragmatic approach as we did when we dealt with the cache permutations earlier and simply lean back and observe (by profiling) how far it will take us, and then judge how good or how bad that is.

4.2 Vectorization

As we have seen in chapter 2 on data structures it is possible to vectorize most of the innermost k-loops but in the test-cases we have today k will not exceed 150. This is still reasonable for SSE and VEX instructions with a vector-length of 2 and 4 for `real(8)` elements, respectively, but for true vector machines a maximal trip-count of 150 is not sufficient for good performance. In the advection and diffusion code, cf. `tflow.f90`, there are several innermost k-loops that have vector instructions of a size equal to the number of tracers. In pure hydrodynamics there are just two tracers (S and T) but when for example a bio-geo-chemical model is included we can have many tracers, currently up to 32, and then for these loops one could consider *loop-fusion* and thereby obtain a maximal vector-length of 32*150. For all the other loops (and also for the cases without the many passive tracers) one have to do something more involved than simple loop fusion. One thing could be to *interchange* the loops like this:

```
do k = 1,kmax
  surfacewet-pointloop: do iw = 1,iw2
    if (k<kh(iw)) then
      i = ind(1,iw)
      j = ind(2,iw)
      ! all wet-points (k,i,j) are reached here
    endue
  enddo
enddo
```

Alternatively, one can construct yet another index `indv` and according bounds `indvl(1:kmax)`, `indvu(1:kmax)` and then sort the wet-points according to `kh`. Then the innermost loop iterates over all wet-points with `kh(iw)` $\geq$ `k`. That is:

```
do k = 1,kmax
  do vi = indvl(k),indvu(k)
    iw = indv(vi)
    i = ind(1,iw)
    j = ind(2,iw)
    ! all wet-points (k,i,j) are reached here
  enddo
enddo
```

or one can permute all arrays according to this layout and re-structure the entire code to have the k-loop as the outermost loop, cf. section 2.6, and then do something like this:

```
do k = 1,kmax
  do iw = vindl(k),vindu(k)
    i = ind(1,iw)
    j = ind(2,iw)
    ! all wet-points (k,i,j) are reached here
  enddo
enddo
```

We must, however, remember the five important constraints set up in the introduction section; here we may be struck by the last one: The above strategies might turn out to be of limited use since the major, most computationally heavy routines (e.g. the `momeqs`) make use of implicit solvers in the k-direction, so re-constructing the code to having outermost k-loops is not a straight-forward thing to do.

4.3 openMP

The implementation of openMP in HBM has generally been accomplished through *code pushing*, i.e. the parallel regions have been pushed into the caller to ensure as large parallel chunks as possible. That is, the code looks like this:

```
...
!$OMP PARALLEL DEFAULT(SHARED)
call foo( ... )
call bar( ... )
!$OMP BARRIER
call baz( ... )
!$OMP END PARALLEL
...
```

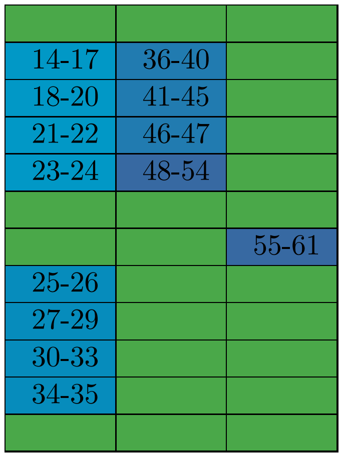
and `foo()`, `bar()`, `baz()` are modified so that the outer loops confine to a subset determined by the current thread number. Moreover, each subroutine has been carefully reviewed for required memory barriers.

The user specifies the number of openMP threads at run time. The domain decomposition is controlled by overloading the subroutine `domp_get_domain` with various *heuristics* depending on the situation at hand, e.g. one routine may need to split the work on active ice-points whereas another may need to split work on wet-points and yet another will split in the west-east index j. Having said that, there is one splitting heuristics that is used most intensively and which might serve as the example here. It goes like this:

```
call domp_get_domain(kh, 1, iw2, nl, nu, idx)
do nsurf=nl,nu
  i = ind(1,nsurf)
  j = ind(2,nsurf)
  ! all threadlocal wet-points (:,:,:) are reached here
  ...
enddo
```

Note that it is a good idea to use the same decomposition for the same set of variables in all the openMP blocks where they appear. Otherwise one will see NUMA effects when trying to scale the application (see e.g. the next subsection).

The actual implementation of `domp_get_domain` used above will try to load-balance the set 1:`iw2` into subsets such that each thread t_i gets approximately `iw3`/`nt` computational points with `nt` being the number of threads used at runtime, cf. the figure below where `nt`=4, `iw3`=61 and where t_1 gets to handle 1:4,14:24, t_2 handles 5:8,25:35, t_3 handles 9:11,36:47 and finally t_4 handles 12:13,48:61.



Note that this strategy may lead to a quite reasonable load-balancing in practice but as revealed in the section on geometric decomposition this is *not* an optimal solution and it is easy to imagine an input-set where this will be a totally hopeless approach. However, as we will see in the profiling section this approach works well for our relevant test-cases. Moreover, as the following estimate shows this gives rise to a pretty well-balanced problem:

In table 2 we first show the balance score C^* per area for our test-case at different numbers of threads (1, 4, ..., 128) and then we weight each area according to the number of wet-points. Thus, in the last row of the table, we have weighted the fact that the number of wet-points differ from one sub-domain to another and the fact that the IDW area has twice as many momentum equation computations per computational cycle as the other areas, and we show a weighted balance score for the entire setup. It shows that our decomposition strategy has potential to scale this problem far beyond our local compute capabilities where we only have 12 threads available.

	1	4	8	16	32	64	128
$C^*(NS)$	1.0	1.0003	1.0029	1.0103	1.0360	1.1395	1.8633
$C^*(IDW)$	1.0	1.0002	1.0004	1.0022	1.0069	1.0362	1.1627
$C^*(WS)$	1.0	1.0012	1.0022	1.0162	1.0740	1.3454	25.0303
$C^*(BS)$	1.0	1.0012	1.0030	1.0188	1.0628	1.2759	9.5645
$\mathbf{C^*}$	**1.0**	**1.0004**	**1.0013**	**1.0071**	**1.0245**	**1.1094**	**3.7132**

Table 2: Balance score per area and weighted.

4.3.1 NUMA tuning

If we insist on scaling our application with openMP then we have to consider our NUMA layout of the variables too. A reasonably well-balanced load distribution will not ensure good scaling by itself. The variables must be placed properly on the CPU sockets too. This subsection aims to describe what we have done in this area.

We have subroutines that consider individual water columns independently whereas others need to communicate with neighbouring water columns, due to e.g. horizontal finite differencing. Thus, if we take a prognostic variable say T and track its usage throughout the program then we will find that it is used in many subroutines and there is *no* universal access pattern (it is used both in routines that need to communicate with neighbours and routines that do not need it) so we have to make a choice. It is obviously not a good idea if we do not try to place and use it consistently, i.e. each thread should pick a subset that is fixed throughout the lifetime of the program. That is, the actual overloading of `domp_get_domain` which basically constitutes our choice must be used consistently throughout the program. We have intentionally done the implementation in such a way that one can easily overload the implementation of `domp_get_domain` if one wishes to experiment with other distribution heuristics.

Before we describe the details, let us make a short recap of how *memory placement* works on NUMA architectures. Regardless of the memory segment type (heap, stack, or bss-uninitialised data[14]), page faults occurs when each new page is touched. When the fault is serviced by the operating system (abbreviated OS in the following), a physical memory page (a so-called *memory frame*) is allocated and assigned to the virtual page address. It is at this step where the location of the physical memory is decided. All physical memory allocation is always in units of pages, which are 4KB by default on our XT5 system. There is a variety of practical reasons that physical memory is always allocated in page-aligned page-size chunks. When a fault occurs, the default *memory placement policy* is to allocate from memory local to the faulting core, and use memory from other NUMA nodes only if the local NUMA node is low on memory. In the case where all of the cores of the compute node are in use by the application, this usually means all memory is local. This rule can be violated if per-process memory requirements vary a lot, such as if MPI rank 0 uses a lot more memory than any other process. The programming environments require an underlying set of *malloc routines*, which are usually provided by GNU libc in our case. The `malloc` library manages memory on a memory segment basis (as opposed to page-level) so it manages the heap, expanding it or trimming it as necessary, and `mmap`ing additional segments as well. It is *not* NUMA-aware and it is *not* aware of OS-level activity such as page faulting. Each process has its own malloc environment, and thus manages its memory segments independently. Bottom line is, if we see poor NUMA performance we cannot blame the compiler nor the runtime environment. We must deal with this ourselves in our application, so let us try to describe what we have done in this area:

Note that we can build the code with pure openMP support or with mixed openMP+MPI support. The latter is our default configuration and thus the one on which we have emphasized our efforts. For the pure openMP builds (i.e. without MPI) we have chosen to confine ourselves to a proper layout for the variables that we are already permuting for cache optimizations. There

[14]bss (Block Started by Symbol) is used by many compilers and linkers for a part of the data segment containing statically-allocated variables represented solely by zero-valued bits initially (i.e., when execution begins). It is often referred to as the "bss section" or "bss segment".

is one obstacle that we have to overcome, namely that `kh` must be read in - and consequently touched - before we can deduce how we should have touched it. This bootstrapping issue has been resolved by reading it into a shadow variable `kh_fake` and then using this to do proper first-touch and initialization of the real `kh`. That is:

```
!$OMP PARALLEL DEFAULT (shared) PRIVATE(ia)
  ! NUMA first touch layout of kh
  do ia=1,narea
    call kh_numa(iw2(ia), iw3(ia), kh(ia)%p, kh_fake(ia)%p)
  enddo
!$OMP END PARALLEL
  do ia=1,narea
    deallocate( kh_fake(ia)%p )
  enddo
  deallocate( kh_fake )
  ...
  do ia=1,narea
    ! allocates numa arrays: u_numa(ia),..
    call numa_re_AllocArrays(ia)
  enddo
!$OMP PARALLEL DEFAULT(shared) PRIVATE(ia)
  ! will do first_touch on numa arrays allocated above
  do ia=1,narea
    call permute_numa_ft(mm1(ia)%p,mm1_numa(ia)%p,
             mm1k_numa(ia)%p,iw2(ia),ind(ia)%p,kh(ia)%p,
             u_numa(ia)%p,...)
  enddo
!$OMP END PARALLEL
  do ia=1,narea
    ! define u_numa(ia),...  based on u(ia),...
    call permute(mmx(ia),nmx(ia),mm1(ia)%p,iw2(ia),
                 kh(ia)%p,u(ia)%p,...)
  enddo
  ! will make u_l, v_l,... point to u_numa, v_numa,...
  call numa_pl2g(narea) ! should also deallocate originals
```

Note that the tuning above confines itself to the arrays that are already being cache permuted. Those that are not will *not* have a proper NUMA layout when running solely with openMP.

It is actually much more obvious to NUMA tune the mixed openMP+MPI version of the code since we are already defining new task local variables (the details are found in section 4.4) for all the variables. Alas, even this is not totally straightforward since again we cannot initialise task local array v_l in a NUMA-friendly way before we know the task local kh_l, needed to do the decomposition:

```
call domp_get_domain(kh_l, 1, iw2_l, nl, nu, idx)
```

Originally, both the local indices mmk_l and the local kh_l as well as many of the variable local arrays v_l were handled simultaneously in task_local_arrays() which run only on the MASTER thread.

If we should do the initialization in a NUMA-optimal way, we must first have mmk_l and kh_l initialized in a NUMA-friendly way and then it is straightforward (more or less) to initialize and to use all the v_l arrays appropriately. Here is how we do it. First, we construct a temporary but task local tmp_kh_l and insert the correct values herein. Then, we use tmp_kh_l to prepare a thread decomposition like this:

```
call domp_get_domain(tmp_kh_l, 1, iw2_l, nl, nu, idx)
```

on each local task. Now, we can make a NUMA-friendly initialization of kh_l using nl,nu from above:

```
do np=nl,nu
  kh_l(np) = tmp_kh_l(np)
enddo
```

At this point of time we no longer need tmp_kh_l and can safely deallocate it. We treat ind_l in the same way, i.e. something like

```
tmp_ind(:,:) = 0
do i="on this task plus halo"
  do j="on this task plus halo"
    np = "according to task permutation"
    tmp_ind(1,np) = i
    tmp_ind(2,np) = j
  enddo
enddo

call domp_get_domain(n3d_l, kh_l, 0, n2d_l, nl, nu)

do np=nl,nu
  i = tmp_ind(1,np)
  j = tmp_ind(2,np)
  ind_l(1,np) = i
  ind_l(2,np) = j
  ! we need zeroes also below kh_l(np):
  mmk_l(1:,i,j) = 0
enddo

! Run through i==0 and imax, and j==0 and jmax on MASTER:
!$OMP MASTER
ind_l(1,0) = 0
ind_l(2,0) = 0
mmk_l(1:,0,    0:  ) = 0
mmk_l(1:,imax,0:  ) = 0
mmk_l(1:,0:,   0:  ) = 0
mmk_l(1:,0:,   jmax) = 0
!$OMP END MASTER

! Run through halo:
halo-loop:
  ind_l(1,np) = i
  ind_l(2,np) = j
  mmk_l(1:,i,j) = 0
end-halo-loop

deallocate( tmp_ind )

! assign correct values to mmk_l
loop:
  mmk_l(k,i,j) = ...
end
```

Possibly, we can assign the correct values for mmk_l at the first access instead of the above-shown two-step procedure with first a zero and then the correct value. And then, it is straight-forward:

```
  ! stride-1 loops as an example:
  v_l(nl:nu) = zero  ! or the correct value if it exists
  do np=nl,nu
    i = ind_l(1,np)
    j = ind_l(2,np)
    ml = mmk_l(2,i,j)
    mu = mmk_l(kh_l(np),i,j)
    v_l(ml:mu) = zero  ! or the correct value if it exists
  enddo

  ! Run through halo:
  halo-loop:
    v_l(:) =  ...
  end-halo-loop
```

We highly recommend that you study mpi_task_permute.f90 for further details.

As revealed in the beginning of this section we have to make a placement choice and there is *no* universally correct choice. When we choose the implementation by the `domp_get_domain` described above we are NUMA tuning the subroutines that are not looking at neighbours. Thus, we expect to see a penalty for this choice as we try to scale say the advection subroutines where neighbours are needed.

4.3.2 openMP profiling

In section 2.6 on cache layout we revealed that although our permutation heuristic was not perfect it gave pretty good results. Now, let us see if this holds for the openMP strategy too. Beforehand though, we better recap Amdahl's law: Let α be the proportion of a program that can be made parallel (i.e. benefit from parallelization) and $(1 - \alpha)$ the proportion that cannot be parallelized (i.e. that remains serial), and assume the parallelization overhead is described by γ_N. Then we can derive the computation time T_N required using N threads from the serial computation time T_1 by:

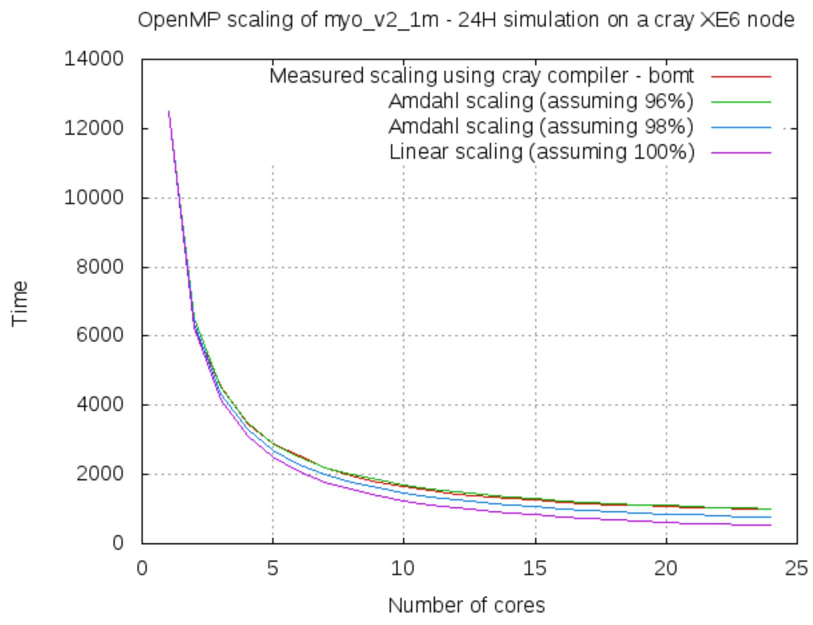
$$T_N = (1-\alpha)T_1 + \frac{\alpha}{N}T_1 + \gamma_N$$

And the theoretical (assuming there is no parallelization overhead, i.e. assuming $\gamma_N = 0$) maximum speedup that can be achieved by using N threads is:

$$S_N = \frac{1}{(1-\alpha) + \frac{\alpha}{N}}$$

As an example, a code with a parallel portion of 50% has a scaling potential of 2, i.e. the theoretical maximum speedup that can be achieved with that code if we had unlimited access to processors is 2. Likewise, codes with a parallel portion of 75% and 96% have scaling potentials of 4 and 25, respectively.

In figure 3 we have plotted the sustained performance on two AMD 12-core Magny-Cours, using from 1 to 24 cores. In the upper figure we have compared the elapsed time for our model run versus number of cores to perfect Amdahl scaling (perfect in the sense that we assume that $\gamma_N = 0$) of 96%, 98% and 100%, and in the lower figure we display the sustained speedup, i.e. elapsed time using N cores divided by the elapsed time using 1 core. From the red curve on the speedup figure, we see that at 24 cores we have obtained a speedup very close to 12.5 which agrees with the theoretical speedup for



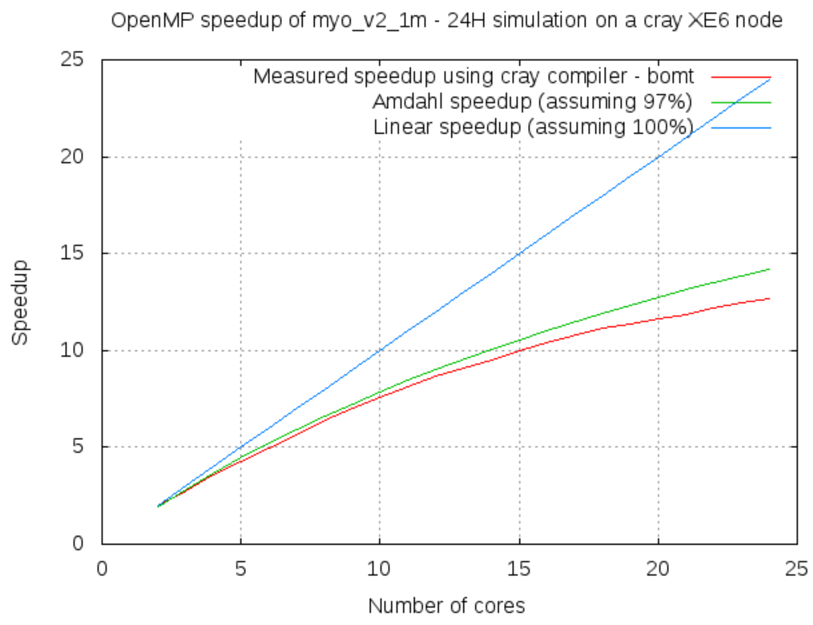


Figure 3: Scaling on two AMD 12-core Magny-Cours using openMP.

a parallel portion of 96%. It is important to keep in mind that all results in this experiment meet our design criterion that the result must be identical regardless of number of cores used; if the results were not identical across number of cores the scalability study would be almost useless and only of very little interest to the developer community. Moreover, the test-case used here is not artificial in any sense; it is the setup that we use for our current operational MyOcean V2 production at DMI. Given that we are aware of serial portions (e.g. all IO) the sustained performance is not too bad.

As discussed above, we have at 24 cores reached approximately half of the scaling potential for this setup. It is interesting to ask: *What happens with a huge number of cores? Can we really reach the theoretical scaling potential?* The answer is: *No, not with this test-case.* The test-case is - even though it is the largest one known to run operationally in the entire Baltic Sea community - far too small. There will simply be too few computational points to distribute among the threads such that each thread has a sufficiently large and well-shaped data set to work on. But as indicated in table 2, we still should be able to scale pretty well to say 64 cores. This will, however, require that the amount of available memory per core is sufficient. But then again, we have not yet seen such a memory-per-core problem with our current setups on the larger systems we have had access to, but for sure, we should keep the possibility in mind when we increase our model size in the future.

Moreover, it is also interesting to cross-compare the scaling on different hardware platforms. In figures 4 - 5 we have compared the scaling of this application on a brand new dedicated AMD Interlagos and a somewhat older not-dedicated Intel Xeon x7550 processor.

4.4 MPI

The MPI parallelization of HBM is characterized by the overall problem decomposition, the data structures used for the task local variables and finally the different types of explicit communication patterns needed to make the implementation correct. The explicit communication patterns require some extended book-keeping, e.g. each task must know which neighbour tasks it needs to communicate with; the index ranges of each task must be known; each task must (of course) know its own wet-points but also the wet-points in the halo-zone which actually reside on the neighbour tasks. Further complications arise with nesting since, with our general approach, we cannot

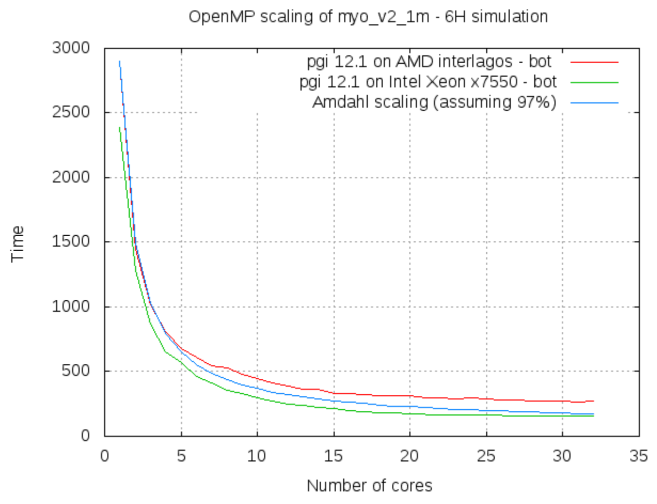


Figure 4: Scaling on AMD versus Intel using the same compiler and same compiler flags on the two different architectures.

assume that points in the vicinity of a nesting border in the enclosing and in the inclosing areas reside on the same MPI task even though they are located in geographically overlapping regions.

We will try to describe each of these aspects in the following and summarize it all by a study of the MPI-performance sustained with our current implementation and the usual test-case.

4.4.1 MPI decomposition

As described in section 4.1, the MPI decomposition splits the overall problem into regular latitude-longitude sub-rectangles. In table 3 we show the example that we used in section 2.6 to show the surface numbers with 4 MPI tasks, consisting of a 2 by 2 decomposition of the domain (first task is i=1:6, j=1; second task is i=7:12, j=1; third is i=1:6, j=2:3; and fourth i=7:12, j=2:3). We have shown both the local numbering (specified in `mm_l`) as well as the global numbering (specified in `mm`). Moreover, table 4 shows the numbering of the neighbours. Each MPI task T will have up to 8 halo-neighbour tasks with 1 being the west neighbour, 2 being the north neighbour, . . . and 8 being the south-west neighbour.

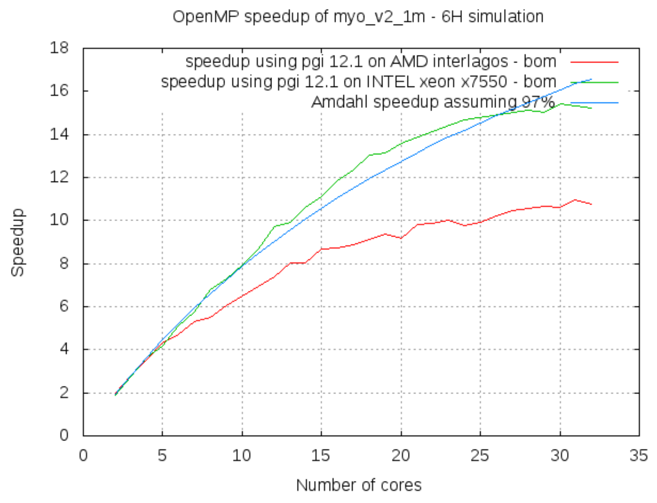


Figure 5: Speedup on AMD versus Intel using the same compiler and same compiler flags on the two different architectures.

1	1=9	
2	2=10	
3	3=11	
4	4=12	
		1=13
1=5		
2=6		
3=7		
4=8		

Table 3: MPI decomposition seen from the surface when using 4 MPI tasks.

5	2	6
1	T	3
8	4	7

Table 4: The 8 halo-neighbour to MPI task T.

The decomposition itself can be specified in a line oriented file. The first line contains the total number of tasks and the rest of the lines come in triples describing the setup for each (task, area), i.e.

```
<total_no_of_tasks>
<task_id_1> <area_id_1> <i_low> <i_up> <j_low> <j_up>
                        <west> <north> <east> <south>
                        <north-west> <north-east> <south-east> <south-west>
...
<task_id_1> <area_id_n> <i_low> <i_up> <j_low> <j_up>
                        <west> <north> <east> <south>
                        <north-west> <north-east> <south-east> <south-west>
<task_id_2> <area_id_1> <i_low> <i_up> <j_low> <j_up>
                        <west> <north> <east> <south>
                        <north-west> <north-east> <south-east> <south-west>
...
```

All numbers are assumed to be read in as format `I5`. Zero in `<low>/<up>` numbers means no active points for the particular task. There can be empty tasks in some but not in all nested areas (with the exception that the first task can be set up as empty on all tasks and thus serve as a dedicated IO task). Minus one in the direction specification means we do not need to do MPI-communication to/from any other MPI-task in that direction (when no other MPI task exists in that direction or that MPI-communication is shielded by land points). Below you will find a sample specification with 5 MPI tasks for a setup with two nested areas. This decomposition is rather arbitrary, used only for testing correctness not for performance. It has four tasks (1, 2, 4 and 5) in area number 1, while in area 2 there is three tasks (1, 2 and 3).

```
    5
    1    1  150  200  163  212
              4   -1   -1   -1
              4   -1   -1   -1
    1    2    1  204    1  300
```

```
              -1   -1   -1    3
              -1   -1   -1   -1
    2    1   189  250  213  414
              -1    5   -1   -1
              -1   -1   -1   -1
    2    2   139  482  301  396
               3   -1   -1   -1
              -1   -1   -1   -1
    3    1     0    0    0    0
              -1   -1   -1   -1
              -1   -1   -1   -1
    3    2   205  482    1  300
              -1    1    2   -1
              -1   -1   -1   -1
    4    1     1  348    1  162
              -1   -1    1   -1
              -1   -1   -1   -1
    4    2     0    0    0    0
              -1   -1   -1   -1
              -1   -1   -1   -1
    5    1     1  188  213  414
              -1   -1   -1    2
              -1   -1   -1   -1
    5    2     0    0    0    0
              -1   -1   -1   -1
              -1   -1   -1   -1
```

A load-balanced (based on a nearly-even split in wet-points, cf. next section on automatic decomposition) 1x5 decomposition of the usual test-case is shown in figures 6 - 9. Each figure shows one of the nested domains, the colour coding indicates the local depth (blue is the most shallow, red is the deepest in each domain), white background is land or out-of-domain, red lines indicate approximate land contours, and the green lines indicate the boundaries of the task decomposition.

4.4.2 Automatic task decomposition

The MPI decomposition file may be hand-edited and fed to HBM at run-time. We have, however, implemented a couple of different methods to generate decompositions automatically. In both cases, some verification is performed of the decomposition layout, but we can unfortunately not guarantee to catch all odd combinations of geometry, decomposition, nesting, etc, so each new setup, including the decomposition, should, as always, be thoroughly verified by the user.

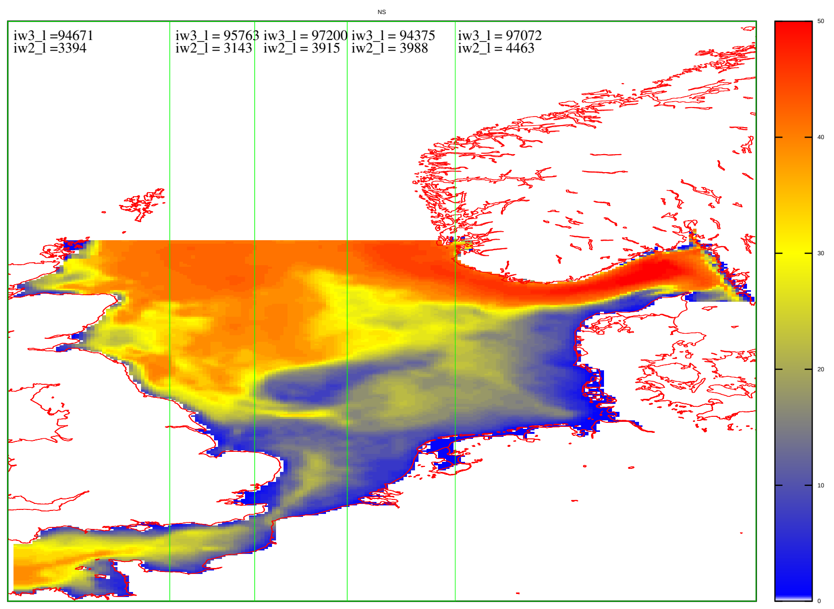


Figure 6: Automatically generated MPI decomposition (1x5) - North sea.

In a straight-forward automatic decomposition the user can specify to decompose into `nproci` tasks in the latitude direction and `nprocj` tasks in the longitude direction. The numbers are specified in the namelist `optionlist` in the `options.nml` file like this:

```
&optionlist
  nproci = 37
  nprocj = 42
/
```

The product of `nproci` and `nprocj` must match the requested number of MPI tasks at run-time. Attempt will then be made to subdivide the latitude-longitude grid into rectangles of approximately the same size. Due to the very irregular domains we are dealing with, see e.g. figure 2, such a strategy will unavoidably lead to poor balance scores and even to empty tasks. It is, however, easy to locate empty tasks and prepare a pruned decomposition file which is stripped from these singularities, but it is in general difficult to treat poor balance scores automatically. We have limited this approach to cases with only one area because it will obviously not work well on more general nested model setups.

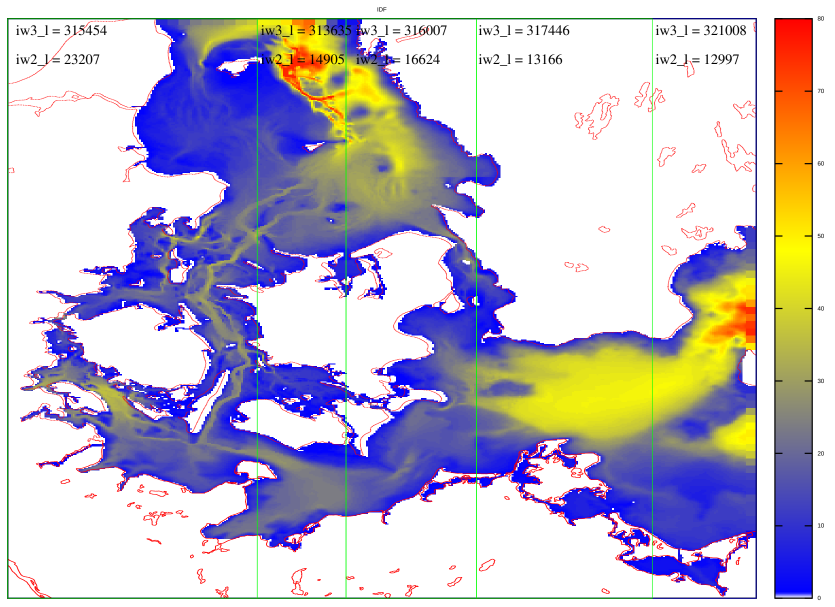


Figure 7: Automatically generated MPI decomposition (1x5) - Inner Danish water.

For nested models with more than one area we have chosen another approach which starts by subdividing each area into `nprocj` meridional sections that for a given nested area contain approximately the same number of wet-points. We call these sections for *semi-optimal J-slices*; an example of decomposing our 4 domain test-case into 5 semi-optimal J-slices is shown in figures 6 - 9. Then, the largest of the J-slices (i.e. the one containing most wet-points) is selected and is subdivided into `nproci` pieces in the latitude direction, containing approximately the same number of wet-points. This determines the zonal decomposition throughout each area. But also with this approach it is difficult to obtain well-balanced decompositions for each nested area. Empty tasks can easily be discarded, but tasks with a relatively small number of wet-points may remain.

We have implemented this `nproci` by `nprocj` heuristic for a multi-domain case such that when `narea > 1`, `nproci = 1` and `nprocj > 1`, then all the possible `nprocj` times `nprocj` pruned decompositions 1x2, 1x3, ..., `nprocj` x `nprocj` are written to files and statistics of the balancing is calculated.

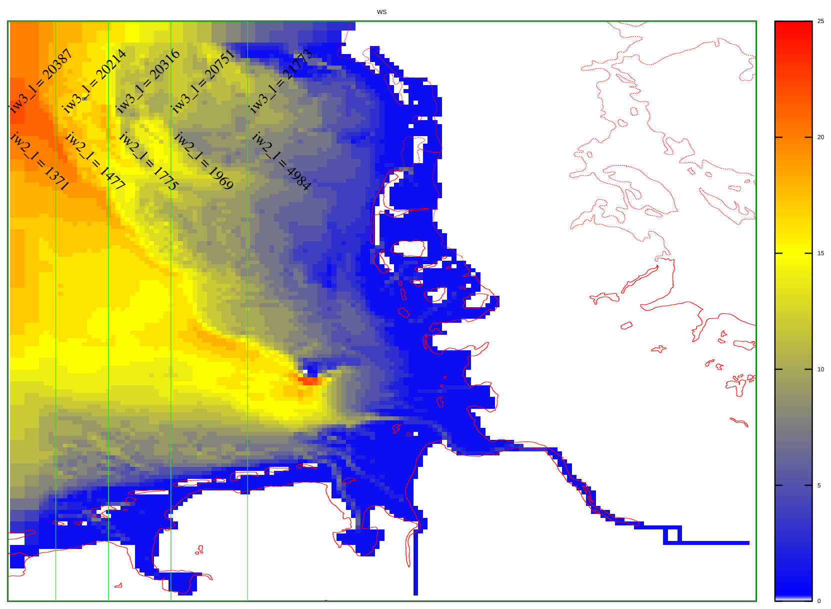


Figure 8: Automatically generated MPI decomposition (1x5) - Wadden sea.

Then we can simply select the best decomposition (based on the balance statistics). That is, if we set `nproci=1` and `nprocj=100` and run it with our usual 4 area test-case, we generate 10000 decompositions, and if we then want to run on say 50 tasks we select from all possible decompositions that resulted in 50 non-empty tasks the decomposition with the lowest balance score. In practice, however, this strategy of selecting a M-tasks decomposition turned out to be merely a fancy way of selecting M by 1 or 1 by M decompositions.

The alternative decomposition strategy, where each nested area is subdivided into `nproci` zonal sections that contain approximately the same number of wet-points is called *semi-optimal I-slices*. Theoretically, we will expect a J-sliced MPI application, when run with one thread on each task, to be comparable in performance to our pure openMP implementation on the same number of cores on a system such as our local Cray XT5, except for a possible different overhead due to explicit MPI communication or due to time required to fork and join the team of openMP threads; note, however, that while the number of times we need to do explicit (MPI) or implicit (openMP) communication may differ a lot so will the time required to com-

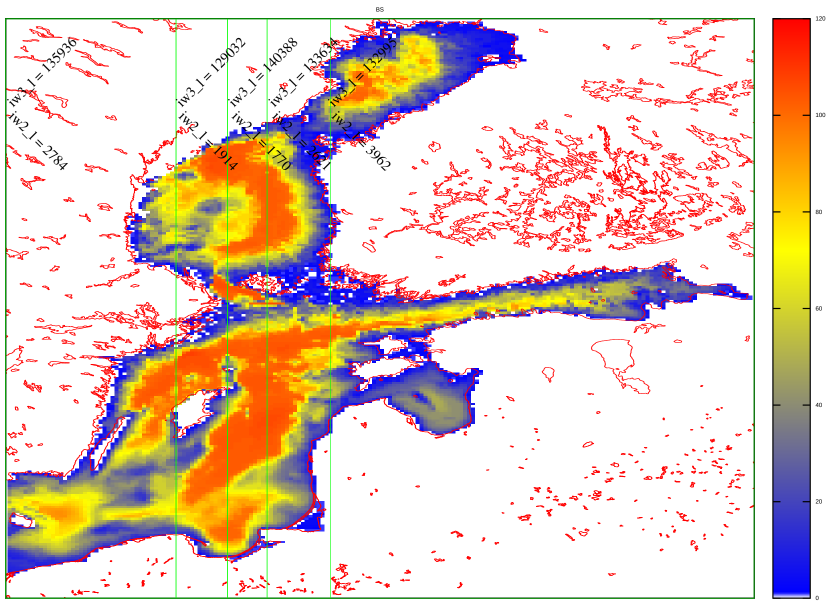


Figure 9: Automatically generated MPI decomposition (1x5) - Baltic sea.

plete the communication and at the end of the day the overhead differences between the two approaches may not be that far from each other. And we will expect I-slices to be more beneficial when we use the maximum number of openMP threads on each task. This performance behavior is indeed the case, as we shall see later: I-slices do perform better than J-slices on our local HPC system when run with the maximal 12 openMP threads on each task. Both I- and J-slices do a very decent job, though, so we see no urgent need for implementing more sophisticated decomposition heuristics with the test-cases we have today and the HPC systems we currently run these on operationally. But we must for sure keep in mind that the use of simple sliced decompositions may turn out to be inefficient in future applications; this is indeed a subject for improvement.

4.4.3 MPI data structures

All outer loops throughout the code essentially look like this:

```
do iw=iw_low,iw_high
   ...
enddo
```

and thus impose the assumption that surface wet-points are placed with stride 1. If we do not permute the data structures according to the MPI decomposition, then this is no longer true, i.e. the set of corner points `i_low`, `i_up`, `j_low`, `j_up` that implicitly defines `iw_low_l` and `iw_high_l` do not represent a stride 1 set of surface points for `task_l`. It is really just a subset of the surface wet-points that are related to `task_l`. In order to ensure that `iw_low_l : iw_high_l` really represent the set of surface wet-points related to `task_l`, we must generate a new index array `mm_l` and all the task local arrays `a_l` that use this index array must be permuted accordingly such that this is indeed the case afterwards. Please consult `mpi_task_permute.f90` for the details. In summary, each task will have the following inner subsets accessed as well as all the data-structures that uses these indices for indirect addressing with stride 1. That is,

```
low_i       : up_i        ; stride 1
low_j       : up_j        ; stride 1
low_ws      : up_ws       ; surface numbers, stride 1
a_l(low_ws) : a_l(up_ws)  ; surface data, stride 1
low_w3      : nwet3       ; subsurface numbers, stride 1
a_l(low_w3) : a_l(nwet3)  ; subsurface data, stride 1
```

We need to figure out where to store the halo values. As indicated above, surface and subsurface indices and data need not constitute contiguous sets. This leaves us with some degree of freedom. We have decided to simply append surface halo points after the inner surface points and the subsurface halo points after the inner subsurface points in the following way:

Let `halo2` be the number of wet surface points in the halo and `halo3` the total number of wet-points in the halo. Each data set then comes in four chunks:

```
a_l(0 : iw2_l)                          ! stride 1
a_l(iw2_l + 1 : iw2_l + halo2)          ! not necessarily stride 1
a_l(iw2_l + halo2 + 1 : halo2 + iw3_l) ! stride 1
a_l(halo2 + iw3_l + 1 : iw3_l + halo3) ! not necessarily stride 1
```

The advantage of this approach is that it is strictly consistent with the approach applied if we had no knowledge of MPI, i.e. surface wet-points first, then the subsurface wet-points; the only practical difference being that we have appended `_l` to all task local variable names as compared to their global counterparts.

As for strides, one should notice that the entire subset for each task is not necessarily accessed with stride 1, only the inner sets are. It has even been possible to keep the property of stride-1 subsurface k-loops for all surface points `nsurf` in the inner set as well as in the halo:

```
do k=2,kh_l(nsurf)
  ! access a_l(mm_l(k,i,j)) in strides of 1 here
enddo
```

As should be evident by now, most global variables now come in a local variant as well as a global variant, e.g.:

Global	Local
mm	mm_l
kh	kh_l
iw2	iw2_l
iw3	iw3_l
un	un_l
vn	vn_l
...	...

This means that in principle all we need to do is from the caller to replace the actual argument from a task global variable to its task local counterpart.

In the serial and pure openMP variants of the code, the local variants are set to point to the global variants and explicit MPI communication operations such as gathers/scatters between local and global structures are skipped in this case.

In the MPI and MPI+openMP variants of the code, the local variants are new data structures allocated in the `Alloc_Local_Arrays()` subroutine which is found in `cmod_local_arrays.f90`, and they are assigned in `task_local_arrays()` in `mpi_task_permute.f90`.

Note that some variables have global meaning only. For instance, we do not change the grid spacing size with tasks, therefore `dxx`, `dyy`, `dz` are the same on all tasks. Also, we do not touch the underlying grid, so `mmx`, `nmx`, `kmx` are the same on all tasks; we might use only a fraction of the global grid, the task-local portion picked by the local index like `mm1_l`.

Please consult `cmod_array.f90` and `cmod_local_array.f90` for the details on the global vs. local arrays.

4.4.4 MPI communication patterns

We need three types of explicit communication patterns, namely one for ordinary *halo-communication*, one for *nesting-communication* and finally one for *global communication*. We highly recommend that you study the file `dmi_mpi.F90` for the details.

The halo-communication deals with exchanging information between neighbour tasks where it is needed for performing e.g. horizontal finite-differencing. The actual communication is controlled by overloading the subroutine `dmpi_halo` with various methods depending on the situation at hand. Thus, one routine is for non-blocking halo swap of `real(8)` variables, one treats the `real(8)` tracer component array `cmp`, and one is for the logical sea ice mask `casus`.

The MPI communication needed for nesting requires that information is gathered from different tasks in different areas. This is implemented by overloading the subroutine `dmpi_gather` with different functionality for different types of data as needed, e.g. for `u`, `v` and `cmp`. Sometimes information needs to be scattered from a global data set to the local data sets. This is done through overloading of the subroutine `dmpi_scatter`.

Global MPI communication deals with exchange of information to/from all local tasks from/to a global data set on a dedicated tasks, named `mpi_io_rank`. Examples are: When meteorological forcing is read in from a binary file on task `mpi_io_rank`, the global data is broadcast to all local tasks by calling `dmpi_broadcast` (appropriately overloaded). The same method is used for open boundary data. The opposite way, i.e. gathering data from all local data sets to one global data set on task `mpi_io_rank`, e.g. for output, is performed through overloading of the `dmpi_gather` subroutine.

4.4.5 MPI performance

As mentioned before, we know that our heuristic for MPI task decomposition is not optimal. Let us see how it performs regardless, just as we did with the permutation for improved cache layout and with our openMP decomposition, both of which are not optimal either but which still do a pretty good job. To evaluate the performance and to estimate the scaling potential of our MPI implementation, we again stick to Amdahl's law which we this time write as

$$T_M = (1-a)T_1 + \frac{a}{M}T_1 + b$$

where T_M is the elapsed time when using M MPI tasks and a is the portion of the application that can be parallelized with MPI. The serial time T_1 may either be a purely serial time (when running without openMP) or it may be the time of running with openMP alone using N threads. The penalty term b takes into account the extra time associated with the explicit communication in the MPI implementation; it may be a constant or an increasing function of M but it is (hopefully) small in magnitude. If we assume that the implementation is such that the openMP and the MPI decompositions are independent, we can write Amdahl's law as

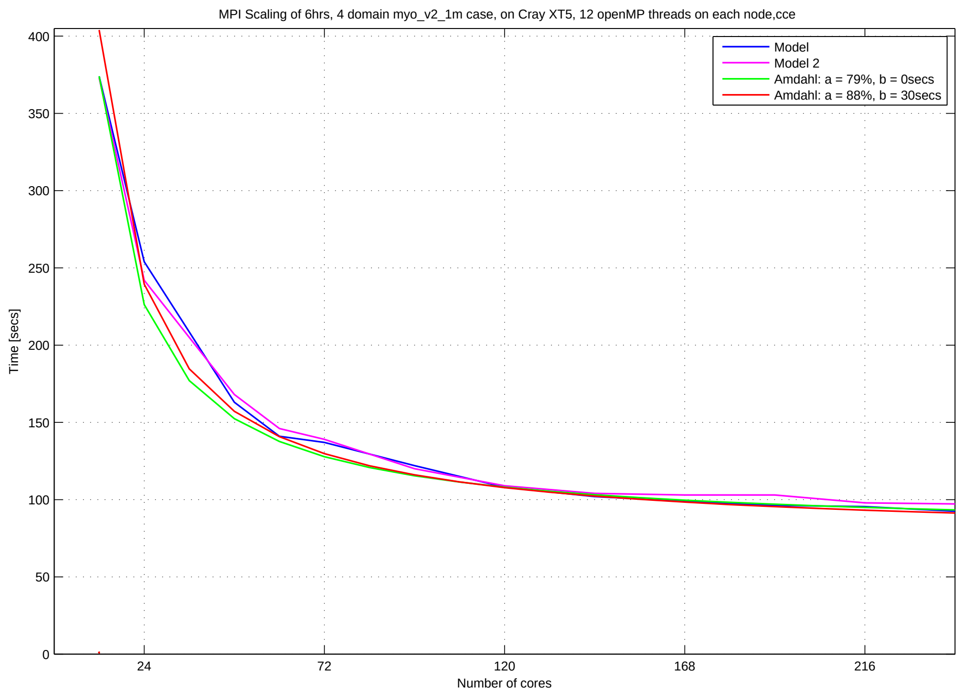
$$T_{M,N} = \left[(1-a) + \frac{a}{M}\right]\left[(1-\alpha) + \frac{\alpha}{N}\right] T_{1,1} + b$$

where $T_{M,N}$ is the time for a run with M MPI tasks and N openMP threads, and $T_{1,1}$ is the pure serial time. Our openMP and MPI decompositions are, however, not fully independent and we can therefore only expect to use the last equation as an optimistic estimate of the scaling potential of the HBM code when we have obtained the parallel portions a and α from independent MPI and openMP experiments, respectively. But in that case we find the theoretical maximum speedup to be:

$$S_{M,N} = \frac{1}{[(1-a) + \frac{a}{M}]\,[(1-\alpha) + \frac{\alpha}{N}]}$$

In figure 10 we show two examples of the performance of the HBM code on a Cray XT5 populated with 12-way AMD Istanbul processors. We have used the full openMP potential on this system, i.e. run with 12 openMP threads on each node, and one MPI task per node. We have used the automatic decomposition into semi-optimal J-slices, and the test-case is the usual MyOcean V2 setup with 4 nested domains (but only run for a 6 hours simulation time in these examples). The result for 60 cores is shown in figure 10 is using the decomposition into 1 by 5 MPI tasks as shown in figures 6 - 9.

First, we notice that it matters quite a lot which compiler we apply; the Cray generated executable runs consistently faster than the one from PathScale; thus, using the Cray compiler we can do a 24 hour simulation in approximately 6 minutes but with PathScale it takes almost 8 minutes. We have compared to Amdahl theory, assuming a penalty of both b =0 seconds (green curves) and b =30 seconds (red curves). For Cray we have plotted parallel portions of a =79% and 88%, respectively, while for PathScale the fit is better with the somewhat smaller a =75% and 83%.



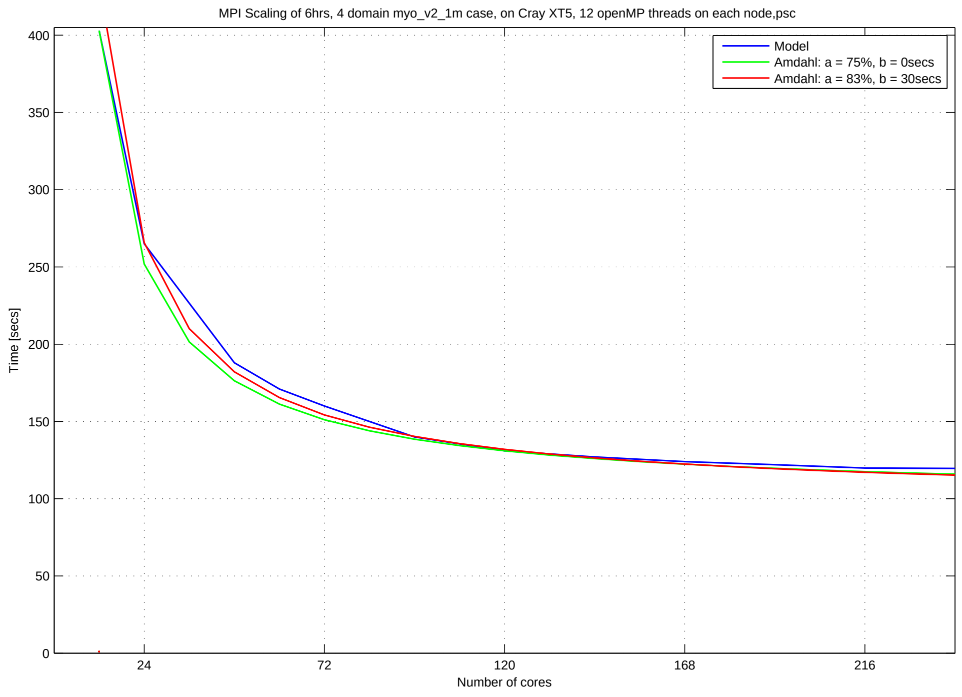


Figure 10: MPI scaling, upper figure is with the Cray compiler and the lower figure is with PathScale.

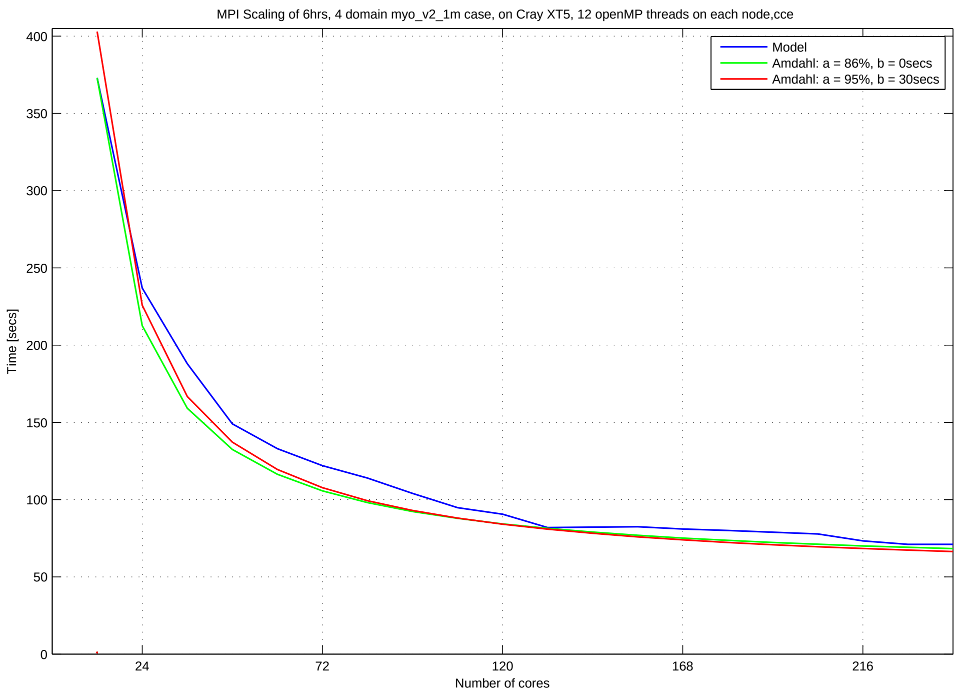
Then, on the upper figure with two identical model experiments (blue and pink curves) we see that there is some noticeable deviations from run to run which we do not see with the openMP scaling experiments; this is expected because we communicate between tasks over a network that is shared with other users of the system whereas openMP experiments are done on a single dedicated node.

Finally, we note that we have reached the maximum scalability for this test-case using the semi-optimal J-slice heuristic: The curves with Cray results are almost flat at the right end, and the curve with PathScale results is actually bending upwards from 216 cores. This is due to the tasks becoming too narrow such that our openMP decompositions become thin strips. In other words, this case is too small to scale further with the applied decomposition strategies.

In figure 11 we show results of scaling experiments using semi-optimal I-slices, this time we use only the Cray compiler. The platform and the test-case are both the same as before. The upper figure is for 12 openMP threads per MPI task and is thus directly comparable to the upper part of figure 10. We see that it matters a lot that we have changed the direction of the decomposition slices; we can now do a 24 hours simulation in 4 minutes 45 seconds using exactly the same computer resources as before due to a more beneficial thread decomposition on each task. For comparison, Amdahl parameters $(a, b) =$(86%,0 seconds) or (95%,30 seconds) fit well in this case.

If we make a pure MPI scaling study, i.e. with 1 openMP thread, we obtain the results like those shown in the lower part of figure 11. Again, same HPC system and test-case as before. We have used the Cray compiler and I-slices as in the upper part of the figure. We have compared to Amdahl scaling with $(a, b) =$(96%,0 seconds) and $(a, b) =$(97%,30 seconds), and we observe that it still scales beyond the 20 cores, just like with pure openMP.

Estimating the scaling potential from the independently observed parallel portions for openMP (figure 10) and for MPI (figure 11, lower part) of 95% - 96%, we get a maximum speedup of 400 - 625. In order to reach this speedup we probably need much larger test-cases if the threads/tasks should not run out of wet-points. In figure 12 we extend the lower part of figure 11 to 90 cores. Taking into account that this case has tasks that are only one single grid point wide in some domains from approximately 60 tasks and



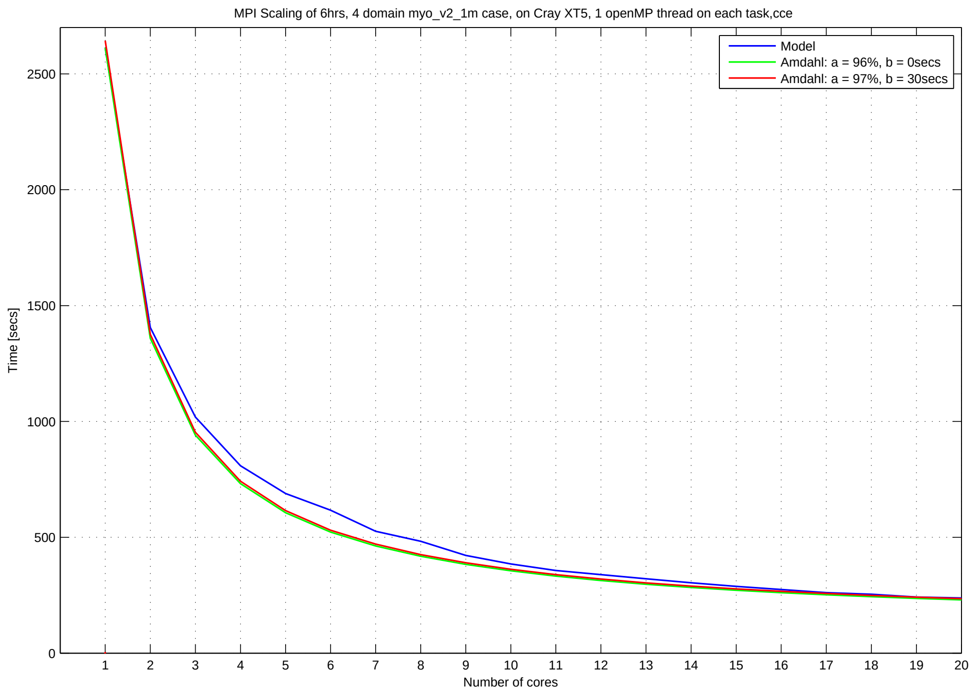


Figure 11: MPI scaling with the Cray compiler. Upper figure: 12 openMP threads per MPI task, 1 MPI task per node. Lower figure: 1 openMP thread per MPI task, up to 12 MPI tasks per node.

up (i.e. that the inner sets become reduced to the same size as their halo), and that there are empty tasks on the largest number of decompositions, it scales reasonable well in the shown interval of cores.

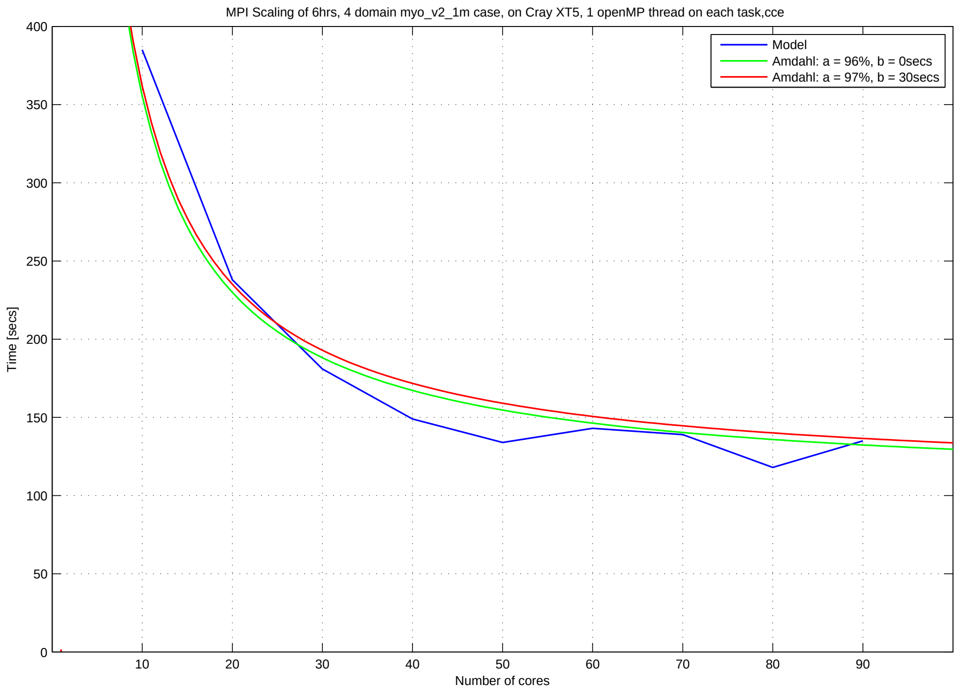


Figure 12: MPI scaling with the Cray compiler from 10 to 90 tasks.

Finally, it should be emphasized, that we have made only one compilation with each compiler, i.e. the same executable is used for all the Cray runs in the upper part of figure 10, in both parts of figure 11 and in figure 12; the only difference between each run is run-time specifications of the number of threads, of the number of tasks and of the extent of each task. All other input are identical and the produced model results are also binary identical.

4.5 Performance perspective

Frankly speaking, the MPI parallelization has mostly been developed to be prepared for future demands. The largest case that we run operationally today is the MyOcean V2 case described throughout this paper and we can run this production sufficiently fast (approx. 24 minutes for a 24 hour simulation) on a single node, i.e. with a single MPI task. For this test-case, MPI is only convenient if we are to do longer simulations fast, say 10 years re-analysis studies. Actually, to the best of our knowledge there are no operational setups in the Baltic region that are larger (measured using the computational intensity defined below) than this case that DMI run for the MyOcean Baltic Model Forecasting Centre production, cf. table 5 and table 6.

We define the *computational intensity* I for a model setup as the sum over all nested areas of the number of potentially active water points (some might occasionally be dry) in that area divided by the time step size (in seconds) used in that area, i.e.

$$I = \sum_{ia=1}^{narea} iw3(ia)/dt(ia)$$

To ease comparison between the computation requirements of different models and different setups, it is useful to look at the relative computational intensity, I_r, which is the intensity of the model setup in question normalized by the intensity of a well-known model setup. Here, we choose to normalize by the specific computational intensity of a certain test-case[15] that has been commonly used at DMI for quick tests as well as longer model simulations (climate studies, re-analysis). This test-case has $I = 10552.5$.

In order to evaluate whether or not the proof-of-concept MPI implementation was sufficient to scale larger cases we made an artificial test-case with a 10-fold computational intensity which is simply the entire North Sea - Baltic Sea region in 1 n.m. resolution, cf. *DMI large* in table 6. We ran this case using up to 40 MPI tasks, cf. figure 13, and it seems to scale well up to 400 cores. Thus, it is certainly feasible to run this case in production, e.g. using 20 MPI tasks we can run a 24 hour simulation in less than 30 minutes. We are planning to make more relevant setups which will probably become even larger in terms of I_r. It is important to realize that the I_r numbers in table 6 should be related to the number of computational resources that is required to complete the run. Please consult appendix C for more details on the setups above as well as setups for other ocean models and the computational resources required to run them.

[15]This test-case is formally known as the test03 test-case.

SETUP	iw3	dt (sec)
DMI ($I_r = 14.0$)		
NS	479081	30
IDW	1583550	15
WS	103441	30
BS	671985	30
FMI ($I_r = 12.1$)		
NSBS	784176	30
IDW	1479208	15
WS	103441	30
BSH ($I_r = 9.1$)		
Coarse	643922	30
Fine	1117390	15
SMHI ($I_r = 1.6$)		
NSBS	506074	30

Table 5: Baltic Sea model setups and their related relative computational intensity used in the MyOcean project. All setups are using the same version of the HBM code.

Institute	Model	Extent	iw3	dt (sec)	I_r
NRL 1/25°	HYCOM	global	861600000	100	816.5
DMI large	HBM	regional	14524110	10	137.6
Mercator 1/12°	Nemo	global	342299000	360	90.1
NRL 1/12°	HYCOM	global	218400000	240	86.2
FCOO dk600	GETM	regional	10668360	90	11.2
FCOO ns1c	GETM	regional	20149380	180	10.6
SMHI bs01	HIROMB	regional	1918522	25	7.3
ECMWF orca25	nemo	global	60731632	1200	4.8
DMI naa	HYCOM-CICE	regional	11099883	300	3.5
ECMWF orca1	nemo	global	2959712	3600	0.07

Table 6: Various ocean models and related setups and their related relative computational intensity. GETM run at FCOO and HIROMB run at SMHI are also regional North Sea - Baltic Sea models. For further details, please consult appendix C.

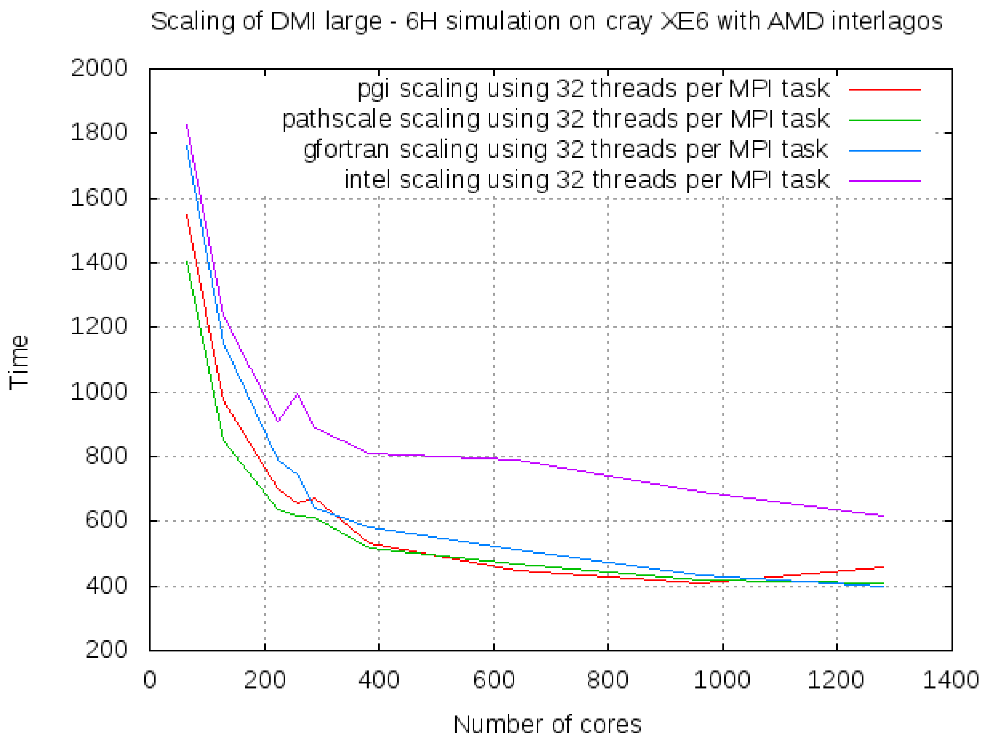


Figure 13: Scaling with different compiler on a cray XE6 with AMD interlagos processors.

4.6 openACC

We have discussed what it would take to prepare the code for alternative hardware too, say accelerators like GPUs. Alternative hardware may turn out to be the only candidate choice for us in a not so far future when increasing demands on low power consumption may dictate what the HPC installation will be. The aim of this subsection is to share our current thinking in this area with respect to HBM.

The openACC standard is of a very recent date: It was announced in November 2011[16] and there are no compiler vendors that officially supports it today. Thus, all findings presented here are results of running on a alpha system

[16]see e.g. http://www.openacc-standard.org/

with nightly builds of the alpha compiler. We are most grateful to Cray for allowing us access to this system. It should be kept in mind that we naturally consider the GPU port very experimental. But without experiments we will never be able to make it, so we might just head-dive into it ...

4.6.1 Introduction

First, let us try to do something very simple allowing us to evaluate on the overall design before we start to consider how we could re-factor the code to make it run well on GPUs too.

We need multiple levels of parallelism in order to make good use of a GPU. The subroutines contain two-level nested loops like this:

```
do iw = 1,iwet2
  i = ind(1,iw)
  j = ind(2,iw)
  do k = 1,1
    ! wet-points (1,i,j) are reached here
  enddo
  do mi=mm(2,i,j),mm(kh(iw),i,j)
    ! wet-points (2,i,j) ... (kh(iw),i,j) are reached here
  enddo
enddo
```

and we would need at least one other loop-nesting level to make good use of GPUs. That is, we need to stripmine the outermost loop with a stride being a multiple of 32. Current compilers cannot handle this automatically so we will do it manually, e.g. something like:

```
do iwo=1,iwet2,stride
  nsurfl = iwo
  nsurfu = min(iwet2,nsurfl+stride-1)
  do iw = nsurfl,nsurfu
    i = ind(1,iw)
    j = ind(2,iw)
    do k = 1,1
      ! wet-points (1,i,j) are reached here
    enddo
    do mi=mm(2,i,j),mm(kh(iw),i,j)
      ! wet-points (2,i,j) ... (kh(iw),i,j) are reached here
    enddo
  enddo
enddo
```

In *CUDA*[17] terminology, we expect that the compiler will use the `threadidx.x` to index the threads within a `nsurfl,nsurfu` subloop and `blockidx.x` to index the different subloops. Moreover, the *CUDA* blocksize will be one dimensional with a size of 32*N threads. The variable `threadIdx.x` would simultaneously be 0,1,2,3,..., 31 inside each block and the overall parallel index would be something like:

```
int iw = blockDim.x*blockIdx.x + threadIdx.x
```

In order to make this work well, we must ensure that the blocks get balanced, i.e. we need something like `domp_get_domain` to ensure that. Moreover, we need to consider what we can do to prevent the strides from serializing.

4.6.2 GPU port of `default_momeqs`

The first goal is really modest: We intend to port a single but computationally expensive subroutine - the momentum equation solver, and then, after evaluating such an experiment we will start considering how to design a solution for the whole code. We hope that we will be able to produce correct results on the GPU and of course, that the GPU solution will have reasonable performance in comparison with the CPU solution, but correctness is our main concern at this point of time. One can enable the experimental code described above with the configure option `--enable-gpu`.

It was not possible to take the loop in `momeqs` as is (too large) so we used 3 simple code transforms: *loop distribution*, *loop interchange* and *strip-mining* to allow the compilers a decent attempt - basically introducing another level of loop-nesting into the big outermost loop and ensuring that the loops are of sizes that the compiler is capable of handling. The single large loop eventually became 12 smaller loops. There are 6 sub-loops for u and v respectively. Note that this distribution itself does not lead to more parallelism within the sub-chunks for u and v, respectively, but that one could indeed handle u and v at the same time if one did not use the same variables for setting up the equations.

The fact that we have done quite some rewrites to allow it to generate GPU code forces us to make a careful study of whether or not our rewrites harm our results when used on the CPU and also study whether or not the rewrites

[17]Compute Unified Device Architecture (CUDA) is a parallel computing architecture developed by Nvidia, cf. [5].

perform better on the CPU. It is important to realize that even though the computations specified by the Fortran statements are *exactly* the same in the two versions of the code the shuffling of the order of the operations makes a big difference to the compilers, i.e. the analysis that they make when generating assembler code can be quite different from one version of the source code to the other; the re-write *must* be verified for correctness. The methods we use to do this analysis is described in details in section 5.

We choose to compare restart files because they contain *all* prognostic variables in full precision and therefore any important discrepancy will show up here. We use the `md5sum` as a simple tool to compare binary files; since we are at this point only interested in knowing if binary files are identical or not.

Cross-comparing, see table 7, the restart file after a serial 6 hours simulation running on the same CPU using the usual test-case we find that we get binary identical results for all but the cray compiler when building with IEEE[18] safe flags (this flaw is being investigated). Moreover, note that we even get binary identical results across a few compilers: gfortran, sun, open64. Thus, we conclude that the manually constructed Fortran code transformations are indeed correct.

Cross-comparing the restart file after a serial 6 hours simulation using the usual test-case but with binaries built with optimization flags we find - not surprisingly - that we no longer get any binary identical files across different compilers, see table 8. Some compilers (gfortran and intel) even produce different results across the two equivalent (in terms of Fortran source) programs. Finally, the results for lahey, sun and open64 did not change when adding tuning flags, i.e. they are binary identical with the IEEE results. Note that the cray compiler that failed to produce binary identical results in the IEEE case actually produces binary identical results in the TUNE case.

Now, let us try to cross-compare the results that emerge from the different compilers for each nested domain. In table 9, we show the worst-case absolute (ε) and relative difference (δ) between all the statistics that the model produces for the 8*4=32 runs on the very same CPU, i.e. 8 compilers with

[18]IEEE: Institute of Electrical and Electronics Engineers. Here we refer to the IEEE Standard for Floating-Point Arithmetic (IEEE 754) which is a technical standard for floating-point computation.

md5sum	**6 hours restart file**
5bb3d02a1dc8488b97a2352c5343d525	pgi/ieee/restart
5bb3d02a1dc8488b97a2352c5343d525	pgi/openacc_ieee/restart
55f52fa4fb22320dbe2877aac8f24a45	gfortran/ieee/restart
55f52fa4fb22320dbe2877aac8f24a45	gfortran/openacc_ieee/restart
f106cd2b2001ceb6b510aac784dcdb8b	pathscale/ieee/restart
f106cd2b2001ceb6b510aac784dcdb8b	pathscale/openacc_ieee/restart
55f52fa4fb22320dbe2877aac8f24a45	sun/ieee/restart
55f52fa4fb22320dbe2877aac8f24a45	sun/openacc_ieee/restart
55f52fa4fb22320dbe2877aac8f24a45	open64/ieee/restart
55f52fa4fb22320dbe2877aac8f24a45	open64/openacc_ieee/restart
8094b126a1f8b7ec1b6675c2ac8b9e1c	lahey/ieee/restart
8094b126a1f8b7ec1b6675c2ac8b9e1c	lahey/openacc_ieee/restart
7c69c9c1e546e711ea3d0072c8e57cd9	intel/ieee/restart
7c69c9c1e546e711ea3d0072c8e57cd9	intel/openacc_ieee/restart
0fc77f2f950e9530f273a10f90347128	cray/ieee/restart
d2a659286ba87150792857152e166f1b	cray/openacc_ieee/restart

Table 7: `md5sums` for the restart file produced by either the default implementation or the openACC implementation of `momeqs` when using IEEE flags for different compilers.

md5sum	**6 hours restart file**
60adac8476115f8689d07dbf42141e05	pgi/tune/restart
60adac8476115f8689d07dbf42141e05	pgi/openacc_tune/restart
78e4eb32e11a347c7c6e1f1df359e7a9	gfortran/tune/restart
aa8975e521f9accfe85ae0760a4de849	gfortran/openacc_tune/restart
22e81c05adadcbab87de7f45b1c61746	pathscale/tune/restart
22e81c05adadcbab87de7f45b1c61746	pathscale/openacc_tune/restart
55f52fa4fb22320dbe2877aac8f24a45	sun/tune/restart
55f52fa4fb22320dbe2877aac8f24a45	sun/openacc_tune/restart
84f3a124fa1e8db98f090201ca8ba601	open64/tune/restart
84f3a124fa1e8db98f090201ca8ba601	open64/openacc_tune/restart
8094b126a1f8b7ec1b6675c2ac8b9e1c	lahey/tune/restart
8094b126a1f8b7ec1b6675c2ac8b9e1c	lahey/openacc_tune/restart
aef2e726544ce0eef7db332bef5a523c	intel/tune/restart
acf89955dcd070182dc78f7a1b0d8c98	intel/openacc_tune/restart
f24723478441685a630601e78e47e1c4	cray/tune/restart
f24723478441685a630601e78e47e1c4	cray/openacc_tune/restart

Table 8: `md5sums` for the restart file produced by either the default implementation or the openACC implementation of `momeqs` when using TUNE flags for different compilers.

4 cases each: ieee, tune, openacc_ieee, openacc_tune.

	NS (ε/δ)	**IDW** (ε/δ)	**WS** (ε/δ)	**BS** (ε/δ)
Mean salinity	2.19e-07 / 6.29e-09	7.29e-07 / 4.50e-08	3.16e-08 / 9.26e-10	1.92e-07 / 2.84e-08
RMS for salinity	2.05e-07 / 5.88e-09	6.90e-07 / 3.79e-08	3.12e-08 / 9.13e-10	1.12e-07 / 1.61e-08
STD for salinity	4.60e-07 / 4.24e-07	4.28e-07 / 5.19e-08	5.96e-09 / 3.05e-09	3.45e-07 / 2.09e-07
Min salinity	0.00e+00 / 0.00e+00	0.00e+00 / 0.00e+00	0.00e+00 / 0.00e+00	0.00e+00 / 0.00e+00
Max salinity	7.51e-10 / 2.12e-11	6.90e-08 / 1.98e-09	1.63e-09 / 4.65e-11	5.35e-11 / 3.40e-12
Mean temp [°C]	4.63e-07 / 5.01e-08	7.45e-07 / 6.66e-08	7.83e-08 / 6.41e-09	9.78e-07 / 1.69e-07
RMS for temp [°C]	4.34e-07 / 4.56e-08	9.39e-07 / 8.11e-08	6.86e-08 / 5.58e-09	2.65e-06 / 4.16e-07
STD for temp [°C]	4.82e-07 / 2.14e-07	1.19e-06 / 3.98e-07	8.06e-08 / 5.74e-08	6.27e-06 / 2.38e-06
Min temp [°C]	7.37e-09 / 1.28e-09	1.31e-09 / 3.27e-10	1.41e-08 / 1.77e-09	3.31e-03 / 1.91e-02
Max temp [°C]	6.00e-13 / 3.23e-14	6.60e-12 / 3.30e-13	9.95e-14 / 5.35e-15	5.92e-10 / 3.19e-11
Mean z [m]	1.32e-06 / 7.12e-05	5.12e-07 / 1.33e-06	1.10e-06 / 2.73e-05	6.10e-08 / 1.37e-07
RMS for z [m]	1.32e-05 / 4.16e-05	4.18e-07 / 1.07e-06	9.04e-07 / 2.41e-06	1.72e-07 / 3.85e-07
STD for z [m]	1.32e-05 / 4.19e-05	1.89e-06 / 2.58e-05	9.98e-07 / 2.67e-06	1.03e-06 / 2.07e-05
Min z [m]	2.87e-10 / 2.71e-10	1.10e-05 / 1.13e-04	1.00e-13 / 6.98e-14	7.78e-07 / 2.23e-06
Max z [m]	3.38e-05 / 2.18e-05	1.10e-06 / 1.97e-06	0.00e+00 / 0.00e+00	1.15e-05 / 1.74e-05
Min u [m/s]	9.99e-14 / 9.04e-14	5.16e-06 / 5.37e-06	0.00e+00 / 0.00e+00	5.53e-08 / 1.49e-07
Max u [m/s]	2.27e-04 / 1.30e-04	2.37e-06 / 4.45e-06	1.24e-08 / 1.04e-08	3.24e-07 / 1.42e-06
Min v [m/s]	3.72e-10 / 2.67e-10	1.11e-06 / 1.33e-06	1.03e-08 / 6.42e-09	4.05e-07 / 9.27e-07
Max v [m/s]	1.31e-04 / 6.21e-05	2.22e-06 / 2.37e-06	9.99e-14 / 9.65e-14	3.75e-05 / 9.55e-05

Table 9: Worst case differences on statistics between the 32 runs.

Next, in table 10, we add two runs on the GPU to the population. That is, we have the 32 runs from before (again 8 compilers, 4 cases: ieee, tune, openacc_ieee, openacc_tune) on the CPU plus two runs (openacc_ieee, openacc_tune) with 1 compiler[19] on the CPU/GPU. Note that the differences between the 34 runs are significantly larger than between the 32 runs above. This finding requires a careful analysis. For instance, the worst case difference on `min z` for the 32 runs on the CPU was 2.87 10^{-10} m whereas it is 4.45 10^{-01} m when we look at the GPU results too and for `min u`, the numbers are 9.99 10^{-14} m/s and 4.45 10^{-1} m/s, respectively. This is obviously not a good sign but looking at the pointwise results in table 11 indicates that it could be a compiler outlier.

[19]At the time of writing it was only possible to perform the GPU experiments with one compiler, namely the alpha version of the compiler from Cray.

	NS (ε/δ)	**IDW** (ε/δ)	**WS** (ε/δ)	**BS** (ε/δ)
Mean salinity	3.05e-06 / 8.76e-08	2.76e-04 / 1.70e-05	6.44e-05 / 1.89e-06	3.76e-06 / 5.56e-07
RMS for salinity	7.88e-06 / 2.26e-07	3.57e-04 / 1.97e-05	7.28e-05 / 2.13e-06	4.58e-07 / 6.57e-08
STD for salinity	1.56e-04 / 1.44e-04	2.46e-04 / 2.98e-05	1.47e-04 / 7.52e-05	1.35e-05 / 8.18e-06
Min salinity	1.04e-03 / 1.78e-01	0.00e+00 / 0.00e+00	0.00e+00 / 0.00e+00	0.00e+00 / 0.00e+00
Max salinity	3.16e-07 / 8.92e-09	3.12e-05 / 8.97e-07	1.15e-05 / 3.28e-07	1.89e-06 / 1.20e-07
Mean temp [°C]	1.26e-04 / 1.36e-05	8.97e-05 / 8.02e-06	2.02e-04 / 1.65e-05	3.43e-06 / 5.92e-07
RMS for temp [°C]	1.65e-04 / 1.73e-05	7.64e-05 / 6.60e-06	2.29e-04 / 1.86e-05	7.44e-06 / 1.17e-06
STD for temp [°C]	1.79e-04 / 7.94e-05	4.06e-05 / 1.36e-05	2.51e-04 / 1.79e-04	1.08e-05 / 4.10e-06
Min temp [°C]	2.08e-05 / 3.61e-06	1.94e-04 / 4.85e-05	4.72e-04 / 5.93e-05	4.54e-03 / 2.64e-02
Max temp [°C]	9.81e-03 / 5.27e-04	1.28e-05 / 6.42e-07	9.83e-03 / 5.28e-04	1.53e-04 / 8.26e-06
Mean z [m]	9.41e-04 / 5.33e-02	5.19e-04 / 1.35e-03	9.47e-03 / 1.91e-01	1.16e-05 / 2.61e-05
RMS for z [m]	8.38e-03 / 2.58e-02	4.95e-04 / 1.27e-03	3.20e-03 / 8.46e-03	2.29e-05 / 5.13e-05
STD for z [m]	8.45e-03 / 2.60e-02	7.95e-05 / 1.08e-03	2.09e-03 / 5.57e-03	1.03e-04 / 2.07e-03
Min z [m]	4.45e-01 / 4.20e-01	3.49e-03 / 3.60e-02	1.65e-02 / 1.15e-02	1.75e-03 / 5.01e-03
Max z [m]	8.42e-02 / 5.15e-02	1.10e-03 / 1.96e-03	0.00e+00 / 0.00e+00	2.70e-02 / 4.09e-02
Min u [m/s]	4.14e-01 / 3.74e-01	1.86e-03 / 1.94e-03	1.33e-01 / 1.67e-01	1.09e-04 / 2.93e-04
Max u [m/s]	1.16e-02 / 6.66e-03	3.98e-02 / 6.96e-02	1.23e-01 / 9.35e-02	1.08e-03 / 4.71e-03
Min v [m/s]	5.42e-03 / 3.89e-03	5.52e-02 / 6.63e-02	1.82e-02 / 1.15e-02	4.08e-04 / 9.32e-04
Max v [m/s]	5.49e-02 / 2.54e-02	1.67e-02 / 1.78e-02	4.29e-02 / 3.97e-02	5.05e-02 / 1.29e-01

Table 10: Worst case differences on statistics between the 34 runs.

In table 11 we see pointwise difference between a run on the CPU solely and a combined CPU/GPU run. The two binaries are both generated by the cray compiler. The reader is encouraged to cross compare these numbers with those found in table 17 in the next section.

	NS (ε/δ)	**IDW** (ε/δ)	**WS** (ε/δ)	**BS** (ε/δ)
cray TUNE				
$\varepsilon_p(zlev,1)$	5.22e-04/3.36e-04	2.29e-04 /4.11e-04	1.35e-05 /6.98e-06	4.16e-05/6.32e-05
$\varepsilon_p(salt,0)$	8.99e-03/9.08e-05	1.15e-01 /1.16e-03	1.17e-05 /1.18e-07	7.27e-03/7.35e-05
$\varepsilon_p(salt,1)$	4.16e-03/4.20e-05	2.82e-02 /2.84e-04	1.17e-05 /1.18e-07	3.03e-03/3.06e-05
$\varepsilon_p(temp,0)$	2.11e-02/2.13e-04	1.57e-01 /1.58e-03	5.70e-06 /5.76e-08	5.42e-02/5.48e-04
$\varepsilon_p(temp,1)$	9.59e-03/9.69e-05	3.64e-02 /3.68e-04	5.70e-06 /5.76e-08	3.44e-02/3.48e-04

Table 11: Worst case pointwise differences when comparing CPU versus GPU results attained with the cray compiler with configure options `--enable-gpu` and with compiler flags being TUNE on both the CPU and the combined CPU/GPU.

Finally, we have looked at the timings emerging from the 32 serial runs and the 2 partly parallel GPU runs. Note that the openACC rewrite of `momeqs` perform worse than our original code for all compilers when we run it on the CPU. Moreover, note that the GPU performance is disappointing. We cannot even beat the serial CPU performance and if we were to beat a thread parallel CPU version there is a very long way to go but again the performance is not our main concern at this stage. We trust that there is room for improvement, cf. next subsection.

4.6.3 Reducing memory transfers between CPU and GPU

The subroutine `momeqs` has two `intent(out)` variables, `un` and `vn`, that we must transfer back from the GPU after each call but not all the `intent(in)` variables need to be communicated from the CPU onto the GPU prior to each execution of `momeqs`. If we consider the `intent(in)` variables relative to the sub-domain at hand[20] then we can split them into:

- static parameters (should be transferred only once)
- semi-static parameters (should be transferred only when required)

[20]Note that we will need to add extra logic on top of the control provided by the openACC directives in order to implement this.

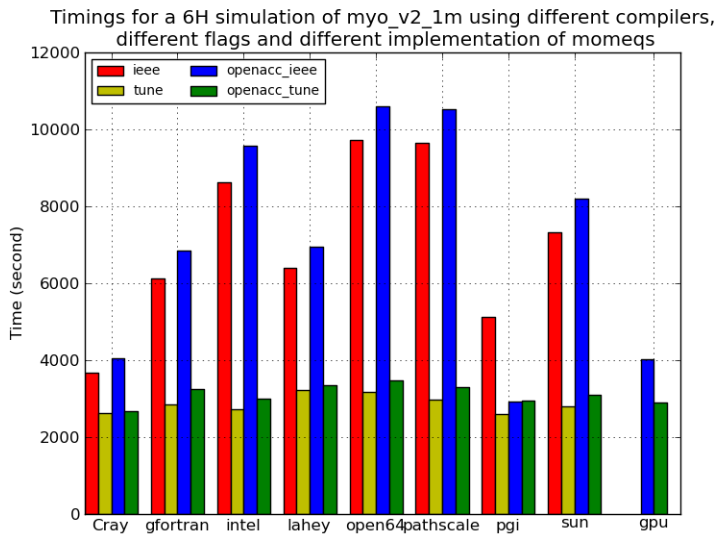


Figure 14: Cross-comparing timings

- dynamic (must be transferred prior to each call)

The semi-static parameters do change every main time step but do not change within the sub timesteps, e.g. the IDW domain is calling `momeqs` twice as many times as the NS domain but the meteorological forcing and the ice variables only change in the main time step, so for IDW these variables will only change prior to call number $0, 2, 4, \ldots$ of `momeqs`.

The static variables are:
`m, iwet2, nud, nvd, nweir, dt, rt, ty, gty,`
`lud, lvd, widx, mm, dx, ft, gtx, tx, khu, khv, ind, kh`.

The semi-static variables are:
`taux, tauy, pl, casus, ice, ueis, veis, rho, avv, z0srf`.

Finally, the dynamic variables are:
`z, u, v, hz, hx, hy, press, eddyh, shear, stretch, div, eddyd`.

After this classification, we can make a rough estimate of the number of bytes that we need to communicate from the CPU onto the GPU, cf. table 12.

Domain	Static	Semi-static	Dynamic
NS	13.3 Mb	8.3 Mb	40.4 Mb
IDW	56.2 Mb	27.7 Mb	133.5 Mb
WS	2.3 Mb	2.2 Mb	8.7 Mb
BS	19.8 Mb	11.0 Mb	56.4 Mb

Table 12: Estimating the data required to communicate the different classes of variables passed onto the subroutine `momeqs`.

Glancing the profile below we find that we can make improvement to a significant portion of total time, i.e. the portion that uses 34.6% of the time, by reducing the cost of the communication from the CPU onto the GPU but the profile also reveals that we need better GPU code generation in order to be able to compete with the CPU. The time spent on computations on the GPU is, however, very small, almost negligible. In conclusion, we need to investigate this technology further before we are ready to deal with a real design for the entire application.

```
Table 2:  Time and Bytes Transferred for Accelerator Regions
  Host  |    Host  |     Acc  |   Acc Copy  |  Acc Copy  | Calls  |Function
 Time%  |    Time  |    Time  |         In  |       Out  |        |
        |          |          |   (MBytes)  |  (MBytes)  |        |

 100.0% | 2371.446 | 2370.175 | 1932474.397 | 194309.121 |  72000 |Total
|---------------------------------------------------------------------------------------------
|  56.8% | 1347.973 |       -- |          -- |         -- |  14400 |momeqs_.ACC_SYNC_WAIT@li.606
|  34.6% |  820.709 |  820.709 | 1932474.397 |         -- |  14400 |momeqs_.ACC_COPY@li.145
|   8.5% |  201.994 |  201.994 |          -- | 194309.121 |  14400 |momeqs_.ACC_COPY@li.606
|   0.0% |    0.510 | 1347.472 |          -- |         -- |  14400 |momeqs_.ACC_KERNEL@li.145
|   0.0% |    0.259 |       -- |          -- |         -- |  14400 |momeqs_.ACC_REGION@li.145
|=============================================================================================
```

5 Validation and verification

The presence of bugs in programs can be regarded as a fundamental phenomenon; the bug-free program is an abstract theoretical concept like the absolute zero of thermodynamics, which can be envisaged but never attained.

Jacob T. Schwartz, professor in math and computer science

The aim of this chapter is to describe our focus on implementation quality. It is our experience that many scientific codes put more focus on adding new features than to ensure correctness. We have focused more on the correctness than on the features implying that we test the software using techniques that are common in the field of computer science but less used in the fields of scientific computing that we are aware of. For instance, in the weather and ocean modelling community it is quite common to measure the quality of a given model by cross-comparing model results with observations. But, unfortunately, we have seen far too many examples where a model gets impressive verification scores yet it is amazingly easy to prove that the implementation is totally off in points where we do not have observations. We have even seen quite a few examples of apparently nice verifications against observations but where it turned out that the model results were solely dictated by the - in this respect - rather arbitrary error handling by the chosen compiler on the system where the buggy model implementation was run; fixing the implementation bugs one by one eventually made the card house crash, resulting in unacceptable verification scores.

Having said that, we really do believe that cross-comparing model results with observations is a reasonable thing to do. At the end of the day it is also this kind of quality assurance that the end-user is interested in - but if it stands alone then it might be quite risky. Both in the sense that results may be totally off in some points/sub-regions that could be interesting to users, and in the sense that the results are not the outcome of solving complicated systems of equations but rather the outcome of a failed attempt to do so mixed with the artifacts of the system's response to this failed attempt. If the sole purpose was to get impressive verification scores in the observation points then we would probably be much better off using a simple statistical model that we could run on a laptop. We feel obliged to justify the computational resources that we use to run the complicated models by putting much more focus on verifying the outcome of the computations performed.

Think about it - would you trust a model where the sole **fact** about its quality was that results in general appear within the expected range, e.g. all parameters has a physical meaningful value in all the grid points, i.e. appear within the expected range, and that it has reasonable results in say 0.02%[21] of the grid points where we have observations for a few (relative to the total number of prognostic variables per grid point) parameters. It becomes even more questionable if the model uses assimilation and thus at frequent intervals tries to approach or even nudge the values of the observed data in the grid points. Consistent buggy observations will lead to consistent buggy results in the neighbourhood and at remote points but the verification scores might still be very impressive and better than if we ran without assimilation, so what does a good verification score give us here? The most positive answer to this question we can think of is: ”Very little”.

When developers are doing code revisions, i.e. modifying existing code or introducing new features into the code, it happens that only the intended outcome of the developer's own code segments is verified. It would be better if one could assure that

- the bug fix or the improvement was successfully done based on the found flaw,
- the new feature was successfully implemented based on expected outcome,

while providing a general assurance that no new errors were introduced because ...

... as a consequence of the introduction of new bugs, program maintenance requires far more system testing per statement written than any other programming. Theoretically, after each fix one must run the entire batch of test cases previously run against the system, to ensure that it has not been damaged in an obscure way. In practice, such regression testing must indeed approximate this theoretical idea, and it is very costly.

Fred Brooks, The Mythical Man Month

[21] The usual test-case has a total of 479081+1583550+103441+671985=2838057 3D wet-points and has observations of some parameters in 106 points. The number of observation levels are certainly less than 5 on average and the total number of observations is thus certainly less than 5*106 and 530/2838057 $<$ 0.0002.

In other words, we find it important to verify after each code revision that what worked before still works and that the code revisions work as intended. A suite of test-cases is a useful help for this, serving as reference cases, but also for guidance in daily work, general documentation of and confidence in the model capabilities, training of and inspiration for modellers, etc. Different, updated test-cases explore different corners of the applicability range (realistic cases, actual operational and project model setups, smaller toy examples, different features are addressed, ...).

In the following subsections we summarize the things we do to ensure correctness and to prevent bugs from sneaking in as we move on with the development. It should be stressed that most of these techniques are not used just when we are up for releases of the source code or when we put a new version of the model into operational mode. Many are actually used during our daily work with the model code as we try to improve things, and selected items are implemented into a test system that is executed as a crontab job every weekday's night.

These nightly tests

- should help us catch errors, inconsistencies, improper designs, non-conforming implementations, etc, early in the development phase,
- should do it at a minimal cost without too much human involvement,
- are intended for smaller, not too much time-consuming regression testing that are performed often at regular intervals,
- should cover a sufficiently broad range of builds and runs such that important aspects are treated,

but we must, however, emphasize that they

- are not a substitute for developers' own more thorough testing and users' model calibration and validation,
- will not cover every corner needed to be tested,
- are not intended for blame-storming but should be considered as a valuable help for the developer community as a whole.

Actually, none of the verification methods that we describe here will relieve us from the original burden (or joy) of doing a careful numerical analysis prior to implementing a new feature. We expect that one has done a careful forward-backward analysis and thus that one ensures that the problem solved runtime is indeed well-conditioned. However, we realize that there are many aspects that one cannot analyse on a piece of paper. For instance, the latest IEEE standard for floating points (IEEE 754-2008) does not specify mandatory requirements for any elementary function such as e^x, $\sin(x)$, x^n, $\ln(x)$ and so forth. The standard only requires that the 5 operations: addition, subtraction, multiplication, division and square root are correctly rounded (using any of the four rounding modes that the standard defines). Also different architectures will have different floating point capabilities, e.g. the SPARC64, IBM POWER and the latest GPUs from NVIDIA have FMA[22] operations that allows for more accurate results than the split operation consisting of a multiply operation followed by an add operation, cf. [10]. Note that the most recent AMD Interlagos have FMA operations too but FMAs are brand-new on the x86-64 platforms so how will our code react to all these implementation-dependent system aspects? Different compilers will allow one to use different capabilities of the processor, e.g. some may not have flags for enabling handling of denormalized numbers whereas others have[23] and different compilers may analyze the code differently and eventually generate quite different machine code operations which eventually may lead to slightly different results. Classical compiler books such as e.g. the so-called *dragon book* [1] even has doubtful rewrites from a numerical point of view so one has to study the numerical side-effects resulting from the different compilers to ensure that these side-effects are inferior to the final results. In addition, we do not believe that numerical analysis on the entire code base is feasible, i.e. the numerical analysis is most often confined to smaller programming units. Thus, what do we do to evaluate the numerical quality of the entire code eventually? This is what this chapter is all about.

5.1 Phrases commonly used in model development

Before we head off describing the various tests, we give a brief definitions of the various phrases that are often used in this context.

[22]fuzed-multiply-add.

[23]This is the case even on systems where the hardware does support denormalized numbers.

In scientific model development generally and also in this chapter, the following words are often used: *calibration, validation, authentication, comparison, evaluation, verification.* These words seem to be subject to diverse interpretation in the modelling community, especially with respect to *validation* and *verification* which for one person are synonyms but for others mean something different. This may lead to some confusion. Let us try to elaborate a little on the above words and the expected action behind them.

Traditionally, a modelling project - given that you work with a released model source code, i.e. a code that has passed the series of technical tests sufficiently well - has three phases, namely a *calibration phase* and a *validation phase* and finally a *production phase.*

Model calibration:
Model calibration is to tune the model until it reproduces the simulated physical (or chemical, biological, etc) phenomena sufficiently well. Model calibration is the process where you "turn the buttons", adjust the free coefficients, modify the bathymetry, the boundary data, etc. Preferably, you have beforehand chosen a (some) suitable calibration period(s), often of relatively long duration so that "all" important situations are covered, and you have agreed-upon measures for how well the model must reproduce the physics of the true nature for this and that parameter - otherwise the calibration task will never end; one can always claim that something should be better.

Model validation:
Model validation is the phase that follows the calibration. The objective of the validation phase is to demonstrate *universality* in the sense that one must be able to reproduce the calibration results, i.e. that the model for a given region which has been calibrated to simulate the physics sufficiently well for one (or more) time period(s) also performs sufficiently well during other periods, keeping all coefficients etc untouched. In other words, hold the model up against data sets which are independent of those used for the calibration.

The traditional cal/val project phases are often followed by a model authentication which takes place during the production phase.

Model authentication:
Model authentication is to demonstrate that the calibrated and validated

model performs sufficiently well during production runs by verification against independent (i.e. not used during cal/val phases) data sets. Authentication can be an integrated part of the production system (e.g. online validation/verification) monitoring the "correctness" more or less on the fly.

Comparison:
A comparison is simply the process of comparing (at least) two data sets, e.g. measured and calculated salinity at a station, raw or statistically processed. There is no interpretation behind, just plain collation of the data material in e.g. tables or in a graphical form. The data material can be processed (statistically, graphically color coded, averaged, or whatever ...), and there can have been different targeted analyses performed.

Evaluation:
Evaluation includes some sort of interpretation; one should relate the comparisons to the objectives: Is this sufficiently well, or what should be better and how do we make it better? Evaluation can be performed more or less automatically, among other things dependent on the measures you (hopefully) agreed upon. Examples: If you have a goal that the STD of the error of modelled sea surface elevation at a given station must not exceed 10 cm, then it is easy to evaluate this from the statistical comparison. But, if you find from a graphical comparison that the pycnocline in the Inner Danish Waters eventually erodes too much, then it requires some human interpretation and analysis to guide the road to follow in attempt to make it better.

Verification:
Verification is the process where you compare and evaluate something. It can be that during the model calibration you test "Is the effect of turning this button as expected, or what should be improved?". It can also be that during production you will like to know if the water level is being forecasted with sufficient precision according to the pre-defined measures. It could be that during code development you do a technical verification "is this ANSI code?", "are the results identical across different configure incarnations as they should be?", ...
Verification with respect to modelling can have two objectives:

- either you would like to show how good the model is, how much confidence the end-user can have in the results,
- or you try to find priorities for further development activities, where

should we focus?

There is a tendency that the first item takes place during production, while the second is during development, but they should ideally accompany each other.

5.2 Technical bugs - serial focus

Debugging is an extremely time-consuming process and we have consequently tried to invent procedures that allow us to catch issues with as little effort as possible. There is a number of technical bugs that we often see and that we can catch by proper usage of the capabilities provided by our compilers. It is a well-known fact that different compilers have different diagnostic capabilities and we therefore recommend to use as many as possible to help catching technical bugs as early as possible. If there was one single compiler superior to the rest we would for sure use that one but, unfortunately, there is not - each have their own pros and cons.

Each paragraph in this section describes a test category and in each paragraph we will mention the scripts we use to conduct the tests. However, we must emphasize that the mechanics inside the scripts is not important. It is the procedure behind, i.e. the *concept*, that we wish to focus on. By showing the scripts we just prove how easy it is to conduct these tests once the framework is established.

The scripts mentioned in this section are handling flags for the following compilers: *pgi, ifort, pathf90, gfortran, crayftn, sun, nagwaref95, lahey, open64, xlf90, sxf90, mipspro* and *absoft*. We perform tests with the first 7 compilers on a nightly basis and the others mentioned in the list occasionally, i.e. when time and access permits it. The scripts will build the code with relevant flags using all the compilers present on the system that one launches it on and then run all the tests in the list of test-cases with the binaries generated. That is, instead of building one binary and test it with one test-case we will typically build 100+ binaries[24] and launch each of these binaries using a couple of test-cases, each run with several different openMP and MPI decompositions.

Nice side-effects of this study:

- we may find that certain compilers produce wrong results

[24]typically 8 compilers, 5 groups of compiler flags (ieee, tune, bound, stack, tune), 4 configure incarnations (serial, openMP, MPI, openMP+MPI)

- we may find that certain compiler options produce wrong results
- we know which compiler is producing the most reliable results
- we know which compiler is producing the fastest executable
- we know which compiler to use if the build time is important

ANSI compliance
If the code does not comply with the language standard, then the behaviour is unpredictable and consequently useless for a further study. As an example, there is a tendency in the modelling community to blend different language dialects, i.e. older Fortran 77 with newer Fortran 90; we must here emphasize that such a coding practise is not always compliant with the ANSI standard and consequently the outcome may be very different from the developer's intentions. The script

```
./build_ansi.sh
```

will add relevant compiler options to warn about ANSI violations in the code.

Many compilers are capable of producing nice warnings about ANSI violations, some are better than others, and often different compilers warn about different issues, and there is *no* compiler that is capable of catching all violations. Therefore, it is recommended to test for ANSI compliance using as many compilers as possible. Needless to say, the developer will have to study the messages emerging from these builds to figure out if and how issues need to be addressed.

It is our experience that not all ANSI violations can be caught automatically by the compilers, not even using many compilers, unfortunately. The developers need to review the code manually. A good coding style is very helpful in this respect. We recommend that a certain code style is followed routinely through all phases, right from writing the first source code line through source code maintenance, see appendix B for our styling rules.

Violating array boundaries
Most compilers have the capability for out-of-bounds testings for arrays and character strings. Many annoying bugs can be caught at an early stage of the development phase using boundary checking. The scripts

```
./build_bound.sh && ./run_bound.sh
```

will add relevant compiler options to force the compiler to generate code with boundary checking for all the compilers present on the system at hand and then run all tests in the list of test-cases.

If your compiler does not support boundary checking you may use your debugger to find some of these issues. However, it is important to realize that debuggers cannot compete with compilers in these areas. Firstly, they can only catch these issues for variables stored on the heap. Secondly, instead of keeping track of individual variables as compilers do they keep track of allocated chunks and consequently they can only catch violations that goes beyond the allocated chunk. Global variables are sometimes allocated together and boundary issues within such chunks will in general not be caught by debuggers.

For the sake of completeness, we provide a totalview^{TM} script for automatic tracking of boundary issues based on *guard blocks* as well as on *red-zones*[25].

Uninitialized local variables
Not all compilers have the capability of initializing real variables with SNAN[26] but some have and in that case `build_stack.sh` will add the relevant compiler options. Some even have the possibility of pre-initializing integers on the stack too to say the largest possible value which is likely to lead to a crash if used. If this option is present it will be added by this script too.

```
./build_stack.sh && ./run_stack.sh
```

We encourage you to read your compiler manual carefully with respect to these options. For instance, some compilers may place large local variables in the heap segment or local variables with the `SAVE` attribute may be placed on the BSS segment and it is not all compilers that allow you to initialize variables in the heap segment or BSS segment with SNAN.

Uninitialized variables in the heap segment
A few compilers have the capability of initializing real variables with SNAN and in that case `build_heap.sh` will add the relevant compiler options. Some are capable of catching uninitialized reals on the heap too and there

[25] Please consult [9] for the details.

[26] IEEE 754 comprises two kinds of NaN (Not a Number) formats, a quiet NaN (qNaN) and a signaling NaN (sNaN) and the latter will generate a floating point exception (if one has asked the compiler to generate code that enables FPEs) when it is used in a computation.

are some that have the possibility of pre-initializing integers to say largest possible value, hopefully leading to a crash when running a test-case.

```
./build_heap.sh && ./run_heap.sh
```

Again, we encourage you to read the compiler manual carefully with respect to these options (not all are capable of initializing e.g. `COMMON` block and module variables - not that `COMMON` blocks are used in this project, though, cf. appendix B).

You can also use the debugger to catch uninitialized variables on the heap by painting allocated memory with say 64bit SNAN (it is not possible to distinguish between data-types when painting via debuggers so you have to choose your patterns based on your source code). We provide a totalview script that you can use to paint memory but you have to rely on the compiler to catch the FPEs[27]. The script `./build_heap.sh` will set the relevant compiler options and `./tv_run_heap.sh` will run it via the debugger and ensure that memory is painted properly.

It is important to understand that the fact that we rely on FPEs to catch these kind of bugs imposes a constraint. If an array has an uninitialized variable at a certain index but this index is not used in any computations with the test-case at hand then we will *not* be able to find it using this method. Actually, we once experienced problems when cross-comparing MPI runs with serial runs and eventually it turned out that the only difference between the two results was the landpoint value (this value is not used in any statistics nor used when we plot differences between two fields).

Other potential technical issues

Many compilers have other good diagnostic capabilities that are less common and we have consequently chosen to collect such more exotic capabilities and gather them into the `diag` group, i.e. you can use the scripts:

```
./build_diag.sh && ./run_diag.sh
```

to add various extra runtime diagnostics. Below are some examples of what these flags could help to detect:

- detect unintended de-referencing of `NULL` pointers,

[27]FPE: floating point exception

- detect dangling pointers,
- detect floating-point overflow in formatted input,
- detect stack overflow at runtime,
- check the stack for available space upon entry to and before the start of a parallel region,

Memory leaks, IO issues, race issues, ...
This is the kind of bugs that the compiler cannot help us detect. We have to rely on debuggers or other external tools to detect these bugs automatically.

Occasionally, we check for memory leaks and IO behaviour. We use valgrind, totalview and truss/strace for these investigations. These investigations are typically done interactively.

In our experience the most difficult problems to debug are race problems with threads and especially for a code like this where thread blocks may span several subroutines. Because of the nature of race issues in openMP implementations where the problem tends to show up randomly, i.e. not in every run but 'only' from time to time and not always with all number of threads, one of the more 'reliable' ways to spot such problems is to run many cases very often and check for binary identical results. The nightly tests serve as a very good servant in this respect.

5.3 Numerical issues - serial focus

The overall aim of this study is to obtain a solid understanding of the results emerging from a simple serial build and a relatively short run, say a 24 hours simulation, before it makes sense to move on with the longer simulations. If we cannot get a 24 hours simulation to produce results across compiler flags, across compilers and across different platforms that are in reasonable agreement then it is very unlikely that we are able to deal with bugs/issues that first become apparent after running year-long simulations. We highly recommend papers like [3], [6] or a textbook like [7] as relevant background information.

5.3.1 Cross-compare statistics and related fingerprints

First, we collect statistics from the runs and try to understand how sensible the current test-case is to the compilers, the compiler flags and the platforms used. A baseline is obtained by gathering logfiles from serial IEEE compliant runs on various platforms with as many compilers as possible using as many relevant test-cases as possible, i.e.

```
./build_ieee.sh && ./run_ieee.sh (on different platforms)
```

The logfiles contain per domain statistics such as *mean*, *minimum*, *maximum*, *rms*, *std* for relevant prognostic variables such as salinity, temperature, sea level and velocities. On top of the statistics, we also cross-compare other run fingerprints such as the number of partially ice-covered grid points and, if a bio-geochemical model is included, the corresponding variables.

As an example, table 13 list the compilers that we usually use on our local cray XT5 installation and the compiler flags we have used to generate IEEE compliant binaries.

Compiler	**IEEE flags**
cray / 7.4.2.106	-O1 -Ofp0 -K trap=fp
intel / 12.0.4.191	-O0 -traceback -fp-model precise -fp-stack-check -fpe0
lahey / 8.10b	-O0 -Knofsimple -Knofp_relaxed
sun / 12.2	-O0 -ftrap=%all,no%inexact -fsimple=0 -fns=no
gfortran / 4.6.0	-fsignaling-nans -ftrapping-math -fno-unsafe-math-optimizations
open64 / 4.2.4-1	-fno-unsafe-math-optimizations -TENV:simd_imask=OFF
pathscale / 3.2.99	-fno-unsafe-math-optimizations -OPT:IEEE_arithmetic,IEEE_arith=1 -TENV:simd_imask=OFF
pgi / 11.10.0	-O0 -Kieee -Ktrap=fp -Mchkstk -Mchkfpstk -Mnoflushz -Mnofpapprox -Mnofprelaxed

Table 13: List of compilers present on the cray XT5 system at DMI and the corresponding compiler flags used for IEEE builds.

Once we have all the logfiles we can cross-compare them and we are usually interested in bounding the differences, i.e. to determine the worst-case differences $\varepsilon_s(f)$ we see across all the runs for each statistical fingerprint s (*mean*, *minimum*, *maximum*, *rms*, *std*) and each prognostic variable f (salinity, temperature, sea level, ...):

$$\varepsilon_s(f) = \max_{c_1,c_2} |s_{c_1}(f) - s_{c_2}(f)|$$

$$\delta_s(f) = \frac{\max_{c1,c2} |s_{c_1}(f) - s_{c_2}(f)|}{\max(|s_{c_1}(f)|, |s_{c_2}(f)|)}$$

with $s_{c_1}(p)$ and $s_{c_1}(p)$ being the statistic s for parameter f obtained by compiler c_1 and c_2 respectively. Sometimes we confine ourselves and fix one compiler c^* as being our reference compiler - typically this is the compiler we used for production builds - and then cross-compare all the other results with that of c^*. We have several scripts that can be used to find $\varepsilon_s(f)$ and $\delta_s(f)$ when cross-comparing a (sub)set of collected results. Needless to mention these fingerprints depend on the test-case at hand and must be established for all the relevant test-cases. If cross-comparing results gives rise to differences higher than expected then we **must** find the reason and fix it before we move on.

In table 14 we show the worst-case differences for a 6 hour simulation of the usual test-case, i.e. the table is a result of cross-comparing logfiles stemming from runs produced by the 8 compilers listed in table 13. One should look carefully through this table, explain apparent odd values and fix unacceptable issues; the table should not rest in itself, it might require action. We may find it necessary to exclude certain compilers whose values may be considered as outliers.

One example of a deviation that is apparent from the table is that the minimum temperature in the Baltic Sea domain seems to fluctuate quite a lot across the 8 compilers compared to all the other statistical values. Looking more carefully at the actual values of minimum temperature produced by each compiler for this domain, we find that gfortran, open64 and sun produces identical results, intel produces a slightly lower value, pathscale and pgi produce slightly higher values, and there are two outliers, cray and lahey. Removing the latter two we find $\varepsilon_{min}(T) = 2.83\ 10^{-5}$ °C, i.e. a factor of 100 is gained on the $\varepsilon_{min}(T)$. A plausible explanation for this very different behavior of the compilers is that the model operates with a cut-off in temperature when it reaches the freezing point at the location which is determined by the local salinity as $T_{freeze} = -0.055S$, and when T drops below T_{freeze} the sea water is treated thermodynamically as sea ice with the excees heat used for ice formation, not for continuous further cooling. The Baltic Sea domain distinguish itself from the other domains by having large pools of relatively fresh water (i.e. low S). Other epsilons are not off for the Baltic Sea. If this difference is unacceptable to the users, then we will have to deal with it numerically.

	NS (ε/δ)	**IDW** (ε/δ)	**WS** (ε/δ)	**BS** (ε/δ)
Mean salinity	1.01e-07 / 2.91e-09	6.71e-07 / 4.14e-08	1.88e-08 / 5.52e-10	1.12e-07 / 1.65e-08
RMS for salinity	9.67e-08 / 2.78e-09	6.60e-07 / 3.63e-08	1.87e-08 / 5.47e-10	7.39e-08 / 1.06e-08
STD for salinity	1.40e-07 / 1.29e-07	4.28e-07 / 5.19e-08	4.40e-09 / 2.25e-09	2.05e-07 / 1.24e-07
Min salinity	0.00e+00 / 0.00e+00	0.00e+00 / 0.00e+00	0.00e+00 / 0.00e+00	0.00e+00 / 0.00e+00
Max salinity	5.57e-10 / 1.57e-11	3.97e-08 / 1.14e-09	9.76e-10 / 2.78e-11	4.35e-11 / 2.76e-12
Mean temp [°C]	2.58e-07 / 2.79e-08	3.06e-07 / 2.74e-08	6.64e-08 / 5.43e-09	9.78e-07 / 1.69e-07
RMS for temp [°C]	2.89e-07 / 3.03e-08	5.24e-07 / 4.53e-08	5.74e-08 / 4.66e-09	2.16e-06 / 3.39e-07
STD for temp [°C]	2.50e-07 / 1.11e-07	1.17e-06 / 3.90e-07	7.51e-08 / 5.34e-08	4.52e-06 / 1.72e-06
Min temp [°C]	7.37e-09 / 1.28e-09	6.53e-10 / 1.63e-10	1.41e-08 / 1.77e-09	2.93e-03 / 1.69e-02
Max temp [°C]	4.01e-13 / 2.16e-14	6.00e-12 / 3.00e-13	9.95e-14 / 5.35e-15	3.69e-10 / 1.99e-11
Mean z [m]	1.19e-06 / 6.39e-05	3.71e-07 / 9.67e-07	6.21e-07 / 1.54e-05	5.82e-08 / 1.31e-07
RMS for z [m]	1.05e-05 / 3.32e-05	3.19e-07 / 8.16e-07	7.63e-07 / 2.03e-06	1.38e-07 / 3.08e-07
STD for z [m]	1.06e-05 / 3.34e-05	5.95e-07 / 8.12e-06	8.35e-07 / 2.24e-06	7.18e-07 / 1.45e-05
Min z [m]	2.21e-10 / 2.09e-10	7.56e-06 / 7.78e-05	1.00e-13 / 6.98e-14	4.23e-07 / 1.21e-06
Max z [m]	2.77e-05 / 1.79e-05	1.08e-06 / 1.93e-06	0.00e+00 / 0.00e+00	7.56e-06 / 1.15e-05
Min u [m/s]	9.99e-14 / 9.04e-14	3.89e-06 / 4.04e-06	0.00e+00 / 0.00e+00	5.50e-08 / 1.48e-07
Max u [m/s]	2.02e-04 / 1.15e-04	8.54e-07 / 1.61e-06	9.15e-09 / 7.65e-09	2.60e-07 / 1.14e-06
Min v [m/s]	2.31e-10 / 1.66e-10	5.82e-07 / 6.99e-07	7.57e-09 / 4.72e-09	2.62e-07 / 5.99e-07
Max v [m/s]	1.04e-04 / 4.95e-05	8.74e-07 / 9.31e-07	9.99e-14 / 9.65e-14	3.75e-05 / 9.55e-05

Table 14: Worst case differences on statistics between the 8 serial IEEE runs.

Second, we need to analyze and understand how sensible the code and the current test-case is to brain-dead tuning flags, c.f. table 15.

Compiler	**TUNE flags**
cray / 7.4.2.106	-O2 -eo -Oipa5
intel / 12.0.4.191	-O3 -fno-alias -ipo -static
lahey / 8.10b	–O3 –sse2
sun / 12.2	-O3
gfortran / 4.6.0	-O3 -funroll-loops -ffast-math -finline-functions -finline-limit=5000
open64 / 4.2.4-1	-O3
pathscale / 3.2.99	-O3
pgi / 11.10.0	-fastsse -Mipa=fast,inline

Table 15: List of compilers present the cray XT5 system at DMI and the corresponding compiler flags used for TUNE builds.

That is, we gather logfiles from serial tune runs on various platforms with as many compilers as possible, i.e.

```
./build_tune.sh && ./run_tune.sh (on different platforms)
```

In this context TUNE does not mean adding music to the work but letting the compiler attempt to do optimization with respect to speed, possibly sacrificing IEEE compliance and possibly allowing unsafe math instructions. We expect to see larger deviations than above with the IEEE runs but the differences must not explode. In table 16 we show the worst-case differences for a 6 hour simulation of the usual test-case, i.e. the table is a result of cross-comparing 16 logfiles stemming from runs produced by 8 different compilers, 2 different classes of compiler flags (ieee and tune) on the same hardware.

Once we have the numbers we can add the test-case to the nightly build and run tests and get an early warning if any of the epsilons are exceeded.

Note that we only see a very small increase in the worst-case differences when moving from IEEE and TUNE implying that we can safely use the TUNE flags listed to build this code. Finally, it should also be mentioned that in all shown cases we have no deviations in the number of partially ice covered grid points.

	NS (ε/δ)	**IDW** (ε/δ)	**WS** (ε/δ)	**BS** (ε/δ)
Mean salinity	1.68e-07 / 4.82e-09	6.71e-07 / 4.14e-08	3.16e-08 / 9.26e-10	1.82e-07 / 2.69e-08
RMS for salinity	1.55e-07 / 4.45e-09	6.60e-07 / 3.63e-08	3.12e-08 / 9.13e-10	1.12e-07 / 1.61e-08
STD for salinity	4.15e-07 / 3.83e-07	4.28e-07 / 5.19e-08	5.80e-09 / 2.96e-09	2.72e-07 / 1.64e-07
Min salinity	0.00e+00 / 0.00e+00	0.00e+00 / 0.00e+00	0.00e+00 / 0.00e+00	0.00e+00 / 0.00e+00
Max salinity	7.51e-10 / 2.12e-11	6.90e-08 / 1.98e-09	1.63e-09 / 4.65e-11	5.35e-11 / 3.40e-12
Mean temp [°C]	4.63e-07 / 5.01e-08	7.45e-07 / 6.66e-08	6.64e-08 / 5.43e-09	9.78e-07 / 1.69e-07
RMS for temp [°C]	4.30e-07 / 4.52e-08	9.39e-07 / 8.11e-08	5.74e-08 / 4.66e-09	2.65e-06 / 4.16e-07
STD for temp [°C]	4.46e-07 / 1.97e-07	1.19e-06 / 3.98e-07	7.51e-08 / 5.34e-08	6.27e-06 / 2.38e-06
Min temp [°C]	7.37e-09 / 1.28e-09	1.31e-09 / 3.27e-10	1.41e-08 / 1.77e-09	3.31e-03 / 1.91e-02
Max temp [°C]	6.00e-13 / 3.23e-14	6.60e-12 / 3.30e-13	9.95e-14 / 5.35e-15	5.92e-10 / 3.19e-11
Mean z [m]	1.20e-06 / 6.47e-05	5.12e-07 / 1.33e-06	8.96e-07 / 2.23e-05	5.82e-08 / 1.31e-07
RMS for z [m]	1.32e-05 / 4.16e-05	4.18e-07 / 1.07e-06	7.63e-07 / 2.03e-06	1.72e-07 / 3.85e-07
STD for z [m]	1.32e-05 / 4.19e-05	1.89e-06 / 2.58e-05	8.35e-07 / 2.24e-06	1.03e-06 / 2.07e-05
Min z [m]	2.67e-10 / 2.52e-10	9.24e-06 / 9.52e-05	1.00e-13 / 6.98e-14	7.78e-07 / 2.23e-06
Max z [m]	3.38e-05 / 2.18e-05	1.10e-06 / 1.97e-06	0.00e+00 / 0.00e+00	8.64e-06 / 1.31e-05
Min u [m/s]	9.99e-14 / 9.04e-14	5.16e-06 / 5.37e-06	0.00e+00 / 0.00e+00	5.50e-08 / 1.48e-07
Max u [m/s]	2.23e-04 / 1.27e-04	2.37e-06 / 4.45e-06	1.13e-08 / 9.43e-09	3.24e-07 / 1.42e-06
Min v [m/s]	2.95e-10 / 2.12e-10	1.11e-06 / 1.33e-06	9.55e-09 / 5.95e-09	4.05e-07 / 9.27e-07
Max v [m/s]	1.31e-04 / 6.21e-05	2.22e-06 / 2.37e-06	9.99e-14 / 9.65e-14	3.75e-05 / 9.55e-05

Table 16: Worst case differences on statistics between the 16 serial IEEE+TUNE runs.

5.3.2 Cross-compare pointwise results

Once we have obtained confidence in the expected statistical deviations we need to take a more careful look at the *pointwise* behaviour. Even if $\varepsilon_s(f)$ and $\delta_s(f)$ seem reasonable for all kinds of statistical fingerprints s and all prognostic variables f we may still experience severe problems in say 0.1% of the grid points and who knows if these points are important to one of the end users. That is, for all f we need to analyze the differences in all the $1\colon iw3$ wet-points and see if there is at least one point where the pointwise difference exceeds a given threshold epsilon. That is, we are generally interested in finding the numbers:

$$\varepsilon_p(f) = \max_{iw,c_1,c_2} |f_{c_1}(iw) - f_{c_2}(iw)|$$

Actually, we find it more convenient to restrict ourselves to look at certain more interesting vertical layers, say for salinity and temperature we may look specifically at the surface ($L = 1$) and at the bottom (the bottom layer is attained at different levels $L \in 1, \ldots L_{\max}$ depending on the point at hand and by definition we let $L = 0$ be the bottom layer). That is,

$$\varepsilon_p(f, L) = \max_{iw \in L,c_1,c_2} |f_{c_1}(iw) - f_{c_2}(iw)|$$

with L being either the surface layer or the bottom layer. Hopefully, by studying the differences we will get a solid understanding of how stable or fragile the code is in its computations of each field in the current setup. Note that larger deviations may be due to the fact that the field (e.g. a front) has shifted a grid point or two and it may be necessary to look at more artificial numbers derived from the current value and neighboring values.

For nested setups (and for parallel runs, cf. below) it is important to pay special attention to the geographic locations of the differences. The difference may be due to issues with the nesting code or with the parallelization. Also, the open boundaries may be of special interest. Again, it is important to look very carefully at these results and ensure that there is a valid explanation of all differences found before proceeding. Especially, one should be alert if any kind of systematic differences occur, e.g. confined near the nesting borders or near halo-zones of MPI-tasks. Some examples are shown in figure 15 - 16: If we compare results from two different compilers we typically see a scattered pattern, while implementation bugs in the MPI-communication for short-term simulations show up around the halos, here near 13.5° E and 55.9° N.

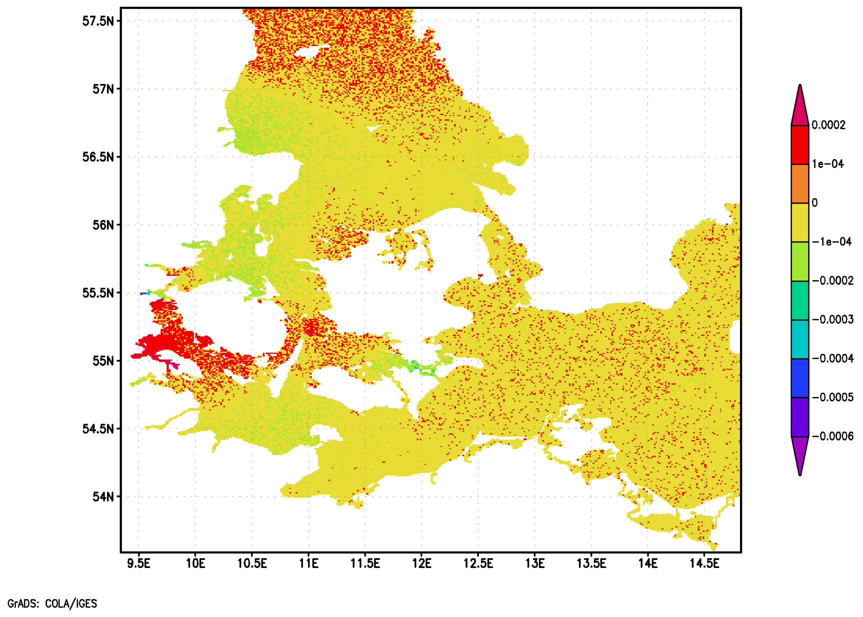


Figure 15: Typical discrepancies in model results between two runs with different compilers.

Once this study is completed we should have a good feeling of how sensible a given field is to the choice of platform, compiler and finally compiler flags.

In table 17 we show the worst-case differences that we find when using different compiler flags. We compare the results per compiler and conclude that the results highly dependents on the choice of the compiler. Note that these differences were completely invisible when we only looked at the statistics.

5.3.3 Cross-compare timings

Now that we have done all the serial builds and runs we might as well compare the time it took to complete these.

Nice side-effects of this study:

	NS (ε/δ)	**IDW** (ε/δ)	**WS** (ε/δ)	**BS** (ε/δ)
sun, lahey				
$\varepsilon_p(*, *)$	0.00e+00/0.00e+00	0.00e+00 /0.00e+00	0.00e+00 /0.00e+00	0.00e+00/0.00e+00
intel				
$\varepsilon_p(zlev, 1)$	5.87e-04/3.78e-04	3.62e-04 /6.48e-04	9.93e-06 /5.12e-06	1.99e-04/3.02e-04
$\varepsilon_p(salt, 0)$	3.03e-02/8.55e-04	4.77e-02 /1.36e-03	3.20e-06 /9.13e-08	9.04e-03/2.58e-04
$\varepsilon_p(salt, 1)$	3.92e-03/1.11e-04	4.22e-02 /1.20e-03	3.20e-06 /9.13e-08	1.53e-02/4.38e-04
$\varepsilon_p(temp, 0)$	1.08e-01/5.80e-03	1.16e-01 /5.80e-03	2.40e-06 /1.29e-07	2.44e-01/1.31e-02
$\varepsilon_p(temp, 1)$	1.47e-02/7.91e-04	4.16e-02 /2.08e-03	2.40e-06 /1.29e-07	6.91e-02/3.72e-03
pathscale				
$\varepsilon_p(zlev, 1)$	2.59e-04/1.67e-04	2.05e-04 /3.67e-04	8.16e-06 /4.21e-06	5.57e-05/8.46e-05
$\varepsilon_p(salt, 0)$	1.37e-02/3.88e-04	9.84e-02 /2.81e-03	3.10e-06 /8.85e-08	1.08e-02/3.08e-04
$\varepsilon_p(salt, 1)$	5.49e-03/1.55e-04	4.44e-02 /1.27e-03	3.30e-06 /9.42e-08	3.56e-03/1.02e-04
$\varepsilon_p(temp, 0)$	4.64e-02/2.49e-03	9.79e-02 /4.90e-03	2.00e-06 /1.07e-07	2.22e-01/1.20e-02
$\varepsilon_p(temp, 1)$	1.07e-02/5.76e-04	3.40e-02 /1.70e-03	2.00e-06 /1.07e-07	4.99e-02/2.69e-03
gfortran				
$\varepsilon_p(zlev, 1)$	3.49e-04/2.25e-04	1.04e-03 /1.86e-03	7.62e-06 /3.93e-06	6.52e-05/9.90e-05
$\varepsilon_p(salt, 0)$	9.75e-03/2.75e-04	9.09e-02 /2.60e-03	1.05e-05 /3.00e-07	7.35e-03/2.10e-04
$\varepsilon_p(salt, 1)$	2.93e-03/8.29e-05	4.22e-02 /1.20e-03	1.05e-05 /3.00e-07	1.01e-02/2.90e-04
$\varepsilon_p(temp, 0)$	1.69e-02/9.07e-04	7.79e-02 /3.89e-03	4.10e-06 /2.20e-07	1.76e-01/9.50e-03
$\varepsilon_p(temp, 1)$	1.25e-02/6.71e-04	2.54e-02 /1.27e-03	4.10e-06 /2.20e-07	4.40e-02/2.37e-03
cray				
$\varepsilon_p(zlev, 1)$	4.16e-04/2.68e-04	2.86e-04 /5.11e-04	5.03e-06 /2.59e-06	7.43e-05/1.13e-04
$\varepsilon_p(salt, 0)$	6.06e-03/1.71e-04	6.21e-02 /1.77e-03	6.30e-06 /1.80e-07	5.37e-03/1.54e-04
$\varepsilon_p(salt, 1)$	4.25e-03/1.20e-04	4.30e-02 /1.23e-03	6.30e-06 /1.80e-07	1.52e-02/4.35e-04
$\varepsilon_p(temp, 0)$	1.45e-02/7.79e-04	1.15e-01 /5.76e-03	3.50e-06 /1.88e-07	6.02e-02/3.24e-03
$\varepsilon_p(temp, 1)$	7.05e-03/3.79e-04	3.81e-02 /1.91e-03	3.50e-06 /1.88e-07	5.89e-02/3.17e-03
open64				
$\varepsilon_p(zlev, 1)$	3.63e-04/2.34e-04	2.67e-04 /4.78e-04	1.23e-05 /6.36e-06	6.52e-05/9.90e-05
$\varepsilon_p(salt, 0)$	9.57e-03/2.70e-04	6.19e-02 /1.77e-03	1.37e-05 /3.91e-07	1.45e-02/4.13e-04
$\varepsilon_p(salt, 1)$	3.93e-03/1.11e-04	3.71e-02 /1.06e-03	1.37e-05 /3.91e-07	1.55e-02/4.42e-04
$\varepsilon_p(temp, 0)$	2.10e-02/1.13e-03	1.68e-01 /8.39e-03	6.90e-06 /3.71e-07	2.42e-01/1.31e-02
$\varepsilon_p(temp, 1)$	1.50e-02/8.07e-04	4.26e-02 /2.13e-03	6.90e-06 /3.71e-07	3.93e-02/2.12e-03
pgi				
$\varepsilon_p(zlev, 1)$	5.10e-04/3.29e-04	4.53e-04 /8.11e-04	1.70e-05 /8.75e-06	8.30e-05/1.26e-04
$\varepsilon_p(salt, 0)$	7.45e-03/2.10e-04	1.09e-01 /3.10e-03	7.40e-06 /2.11e-07	8.27e-03/2.36e-04
$\varepsilon_p(salt, 1)$	2.10e-03/5.94e-05	3.57e-02 /1.02e-03	7.40e-06 /2.11e-07	4.55e-03/1.30e-04
$\varepsilon_p(temp, 0)$	2.31e-02/1.24e-03	9.21e-02 /4.61e-03	6.60e-06 /3.55e-07	2.03e-01/1.10e-02
$\varepsilon_p(temp, 1)$	7.28e-03/3.91e-04	3.48e-02 /1.74e-03	6.60e-06 /3.55e-07	3.86e-02/2.08e-03

Table 17: Worst case pointwise differences when comparing IEEE results with TUNE results.

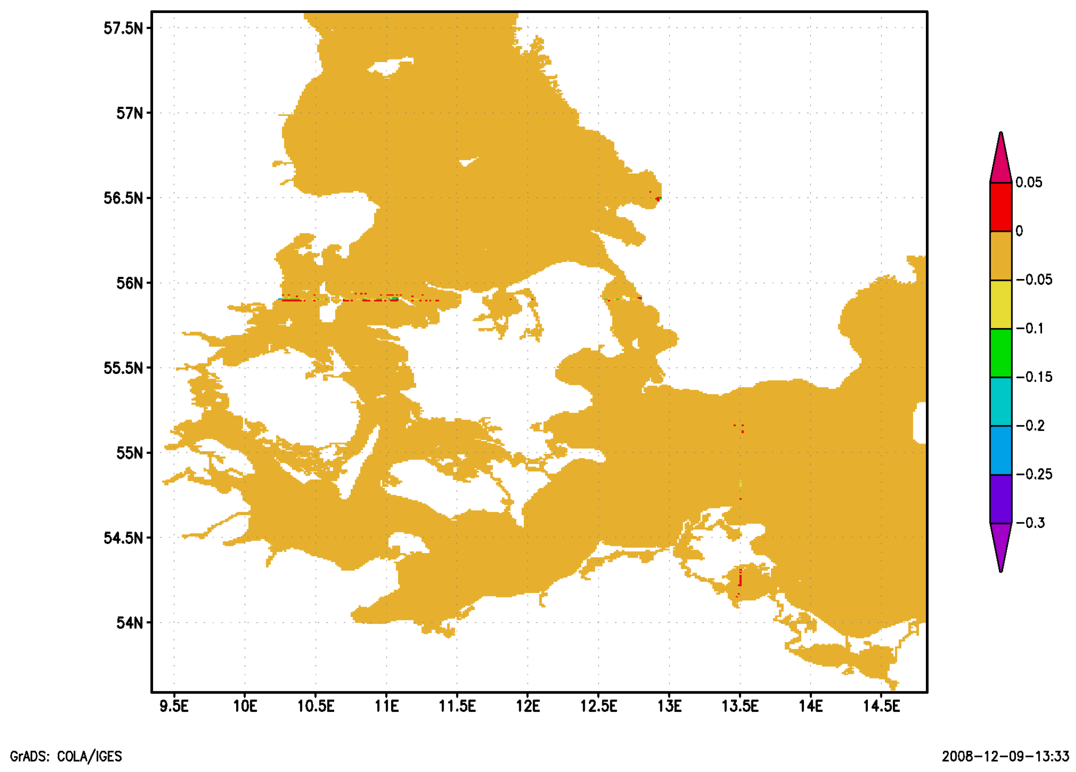


Figure 16: Typical discrepancies in model results between two runs with the same compiler (same build) but run with different numbers of MPI tasks and a flaw in the implemetation of MPI communication.

- We may find if certain compilers have difficulties in keeping up with the rest. If this is indeed the case and it cannot be fixed by modifying compiler options, then it is time to conduct a more detailed profiling study and eventually get in touch with the compiler vendor.
- One compiler might be super-fast (run-time) but by cross-comparing with the worst-case differences above it might also be very wrong.
- Some compilers might be very slow when building the software.

Some examples: Serial timings (run time) on the XT5 system at DMI for the usual test-case using different compilers with tune flags are shown in figure 17. In figure 18 we have gathered information on the time it takes to build HBM with different configure options for a number of different compilers using different groups of compiler flags: One thing is that a tune variant takes a certain amount of time to build; we can accept that since it will likely pay off when we get to do our production runs. But for our daily-day development and testing, some compilers might seem hopeless as a working tool.

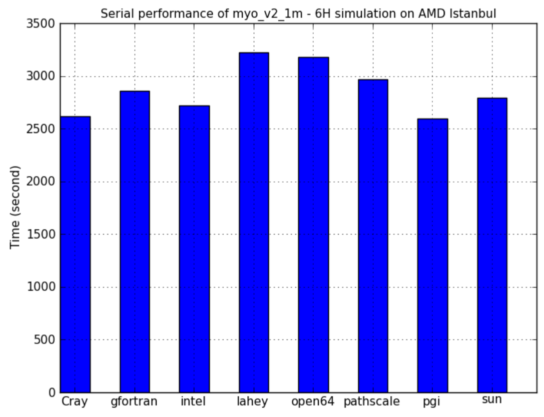


Figure 17: Serial run time using different compilers with tune flags.

5.4 Bugs and numerical issues - parallel focus

Due to the use of `#if defined (MPI)` in the code, compiling with or without MPI actually creates two semantically different codes. MPI-variants of the above ansi, stack and bound tools must also be executed:

- Redo the initial steps with MPI enabled

```
./build_mpi_ansi.sh
./build_mpi_stack.sh && ./run_mpi_stack.sh
./build_mpi_bound.sh && ./run_mpi_bound.sh
```

- Ensure that MPI works correct

```
foreach compiler C
  foreach relevant decomposition D
    md5sum archive tempdat.* restart sponge*
```

must lead to identical `md5sums` for all D when C is fixed.

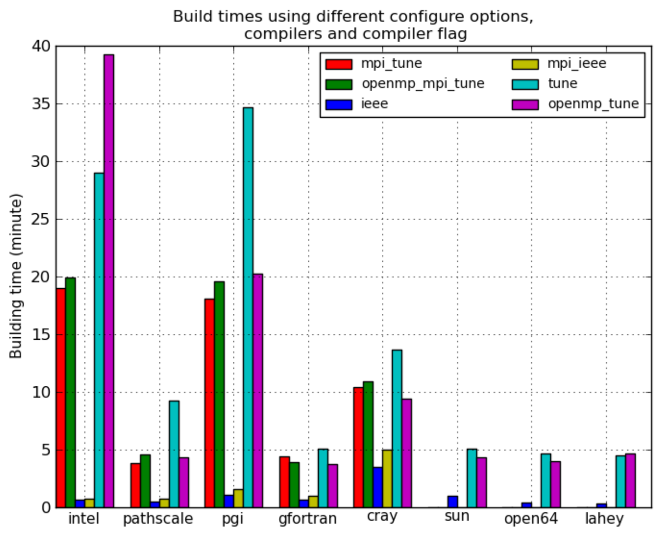


Figure 18: Build timings for different configure options and different compilers with different groups of compiler flags.

- Ensure that openMP works correct

```
foreach compiler C
  foreach number_of_threads N
    md5sum archive tempdat.* restart sponge*
```

  must lead to identical `md5sums` for all N when C is fixed.

- Ensure that MPI+openMP works correct

```
foreach compiler C
  foreach number_of_tasks M
    foreach number_of_threads N
      md5sum archive tempdat.* restart sponge*
```

  must lead to identical `md5sums` for all pairs (M,N) when C is fixed.

The observant reader may wonder why we have `ansi`, `stack` and `bound` for MPI but apparently not for openMP. The reason is that in our experience most compilers cannot generate the latter two tests while having openMP enabled and we have consequently chosen to disregard these tests for the time being.

If the `md5sums` in the experiments above are not identical, then one can try to select a couple of fields, say temperature and salinity, and plot the differences. It is likely that one will see differences around the MPI-tasks' halos or around the nesting borders due to implementation bugs introduced in the corresponding code segments.

Note that the `md5sum` test is a necessary but not a sufficient condition for correct behavior. Not all output files store the fields in full precision, e.g. variables are truncated for formatted output to ascii files or compressed by scaling and conversion to integers for output to disk-space saving binary files. Actually, we could choose to compare restart files only because they contain a snapshot of *all* prognostic variables in full precision and therefore any important discrepancy will show up here; taking other files into account just demonstrates that the output has not been cluttered. If you do not get binary identical restart files but you do get binary identical tempdat files then you know that the differences between the two results are so small that they vanish completely when we scale and store the results as integers.

Table 18 shows 7 runs of 4 different configure options (`default`, `--enable-openmp`, `--enable-mpi`, `--enable openmp --enable-mpi`). The 4 binaries are all generated by the pathscale compiler using the TUNE compiler flag. Please note, that we get binary identical results across these configure options and across different decompositions when we use this particular compiler; this is indeed a sign of a healthy implementation.

If the `md5sums` for the serial runs are identical with those from the parallel runs, then we can skip doing yet another series of epsilon tests and simply conclude that our former tables hold not just for the 16 runs but for *all* the parallel runs as well. On the other hand, if some of the `md5sums` differ then one needs to establish and analyse the new epsilons for both statistics and for the pointwise results.

Table 19 shows that openMP, MPI and serial builds produces binary identical results for different compilers so that the IEEE epsilons computed earlier can be used for openMP and for MPI builds too. That is, our MPI implementation is safe as compared to the pure serial implementation.

md5sum	6 hours restart file
1221f8ed007f97ac68cbab32901a2343	pathscale/0_0_tune/restart
1221f8ed007f97ac68cbab32901a2343	pathscale/0_12_openmp_tune/restart
1221f8ed007f97ac68cbab32901a2343	pathscale/1_0_mpi_tune/restart
1221f8ed007f97ac68cbab32901a2343	pathscale/2_0_mpi_tune/restart
1221f8ed007f97ac68cbab32901a2343	pathscale/12_0_mpi_tune/restart
1221f8ed007f97ac68cbab32901a2343	pathscale/1_12_openmp_mpi_tune/restart
1221f8ed007f97ac68cbab32901a2343	pathscale/2_12_openmp_mpi_tune/restart

Table 18: Comparison of serial TUNE with openMP_TUNE using 12 threads with MPI_TUNE using 1 or 2 MPI tasks with openMP_MPI_TUNE with 12 threads and 1 or 2 MPI tasks. All the results are generated by the PathScale compiler and note that the results are binary identical.

Alas, it is not all the compilers that produce binary identical results across all configure incarnations as pathscale does when we turn on optimizations, so we must do another epsilon test to confirm that the parallel versions do not harm the results in any significant way. In table 20 we see `md5sums` for 27 runs[28], produced across configure options, compilers, compiler flags, decompositions. The table shows that not all configure options produce binary identical results when the TUNE flag is used; care should be taken if one decides to proceed with one of those compilers for production runs.

We encourage the reader to cross-compare results in table 21 with the results presented in table 14 and table 16. Based on these results we can conclude that in terms of statistics it is safe to build the code with optimization flags and it is also safe to run the code in parallel (openMP, MPI or even openMP+MPI). The statistics looks fine for all the variables.

Cross-comparing the results in table 22 with the results in table 21 we see that the pointwise differences are significantly larger than the differences on the per-area statistics. Moreover, one notes that the magnitude of the differences is closely related to the sub-domain, e.g. the differences in WS are significantly smaller than the differences for the other sub-domains. In WS the results in table 22 are (almost) identical for the bottom layer and

[28] 4 compilers, 7 runs each, except for Cray which at the time of writing unfortunately segfaults in openmp_tune runs on our local HPC installation, not on other HPC systems we had access to.

md5sum	**6 hours tempdat file**
0d74fec06448dac8a962bc7d183c5f39	pathscale/0_0_ieee/tempdat.00
0d74fec06448dac8a962bc7d183c5f39	pathscale/0_12_openmp_ieee/tempdat.00
0d74fec06448dac8a962bc7d183c5f39	pathscale/1_0_mpi_ieee/tempdat.00
0d74fec06448dac8a962bc7d183c5f39	pathscale/2_0_mpi_ieee/tempdat.00
0d74fec06448dac8a962bc7d183c5f39	pathscale/2_12_openmp_mpi_ieee/tempdat.00
374bdc91a00e2d746baefa3de940f6d8	pgi/0_0_ieee/tempdat.00
374bdc91a00e2d746baefa3de940f6d8	pgi/0_12_openmp_ieee/tempdat.00
374bdc91a00e2d746baefa3de940f6d8	pgi/1_0_mpi_ieee/tempdat.00
374bdc91a00e2d746baefa3de940f6d8	pgi/2_0_mpi_ieee/tempdat.00
374bdc91a00e2d746baefa3de940f6d8	pgi/2_12_openmp_mpi_ieee/tempdat.00
7ec21b30b046d163f3ec3fa993f7ce65	intel/0_0_ieee/tempdat.00
7ec21b30b046d163f3ec3fa993f7ce65	intel/0_12_openmp_ieee/tempdat.00
7ec21b30b046d163f3ec3fa993f7ce65	intel/1_0_mpi_ieee/tempdat.00
7ec21b30b046d163f3ec3fa993f7ce65	intel/2_0_mpi_ieee/tempdat.00
7ec21b30b046d163f3ec3fa993f7ce65	intel/2_12_openmp_mpi_ieee/tempdat.00
e5571093e928cdab23e556b02bb53c86	cray/0_0_ieee/tempdat.00
e5571093e928cdab23e556b02bb53c86	cray/0_12_openmp_ieee/tempdat.00
e5571093e928cdab23e556b02bb53c86	cray/1_0_mpi_ieee/tempdat.00
e5571093e928cdab23e556b02bb53c86	cray/2_0_mpi_ieee/tempdat.00
e5571093e928cdab23e556b02bb53c86	cray/2_12_openmp_mpi_ieee/tempdat.00
ba64f4822ed208ae22ba24bc2de6e2bc	gfortran/0_0_ieee/tempdat.00
ba64f4822ed208ae22ba24bc2de6e2bc	gfortran/0_12_openmp_ieee/tempdat.00
ba64f4822ed208ae22ba24bc2de6e2bc	gfortran/1_0_mpi_ieee/tempdat.00
ba64f4822ed208ae22ba24bc2de6e2bc	gfortran/2_0_mpi_ieee/tempdat.00
ba64f4822ed208ae22ba24bc2de6e2bc	gfortran/2_12_openmp_mpi_ieee/tempdat.00

Table 19: Comparison of 4 configure options (default, `--enable-openmp`, `--enable-mpi`, `--enable-openmp --enable-mpi`) all using the IEEE compiler flag. That is, IEEE-serial (0_0_ieee), IEEE-openMP (0_12_openmp_ieee), IEEE-MPI (2_0_mpi_ieee) and IEEE-openMP-MPI (2_12_openmp_mpi_ieee) for different compilers. Note that we get binary identical results across the 4 configure options for all compilers when using the IEEE flag category.

md5sum	6 hours restart file
6edde7634af789768134d03eaaf6a309	pgi/0_0_tune/restart
6edde7634af789768134d03eaaf6a309	pgi/1_0_mpi_tune/restart
4f22d22d6ec99ab08832aa3a3abaf801	pgi/2_0_mpi_tune/restart
c176784b3944f81d1b95ea3d7c8a5009	pgi/12_0_mpi_tune/restart
6edde7634af789768134d03eaaf6a309	pgi/0_12_openmp_tune/restart
6edde7634af789768134d03eaaf6a309	pgi/1_12_openmp_mpi_tune/restart
4f22d22d6ec99ab08832aa3a3abaf801	pgi/2_12_openmp_mpi_tune/restart
c13e7c6e2abdc5b9699691067b830190	intel/0_0_tune/restart
bbbff8636db57bc7b6af11b5c4710218	intel/1_0_mpi_tune/restart
bbbff8636db57bc7b6af11b5c4710218	intel/2_0_mpi_tune/restart
bbbff8636db57bc7b6af11b5c4710218	intel/12_0_mpi_tune/restart
626d01aa0f808d13ad5f150e44986bd8	intel/0_12_openmp_tune/restart
c376d67c2ef33f9b7e6a9af8907da4a2	intel/1_12_openmp_mpi_tune/restart
c376d67c2ef33f9b7e6a9af8907da4a2	intel/2_12_openmp_mpi_tune/restart
788935ff7f0c10fb9a5f067aead9f95d	cray/0_0_tune/restart
ffbe5526e7334ab54aa76655de92b476	cray/1_0_mpi_tune/restart
ffbe5526e7334ab54aa76655de92b476	cray/2_0_mpi_tune/restart
ffbe5526e7334ab54aa76655de92b476	cray/12_0_mpi_tune/restart
N/A	cray/0_12_openmp_tune/restart
5c386d6de33c0ef5733de069a9b9626b	cray/1_12_openmp_mpi_tune/restart
5c386d6de33c0ef5733de069a9b9626b	cray/2_12_openmp_mpi_tune/restart
7bae056a42dab838e24f2766b68f401d	gfortran/0_0_tune/restart
7bae056a42dab838e24f2766b68f401d	gfortran/1_0_mpi_tune/restart
7bae056a42dab838e24f2766b68f401d	gfortran/2_0_mpi_tune/restart
7bae056a42dab838e24f2766b68f401d	gfortran/12_0_mpi_tune/restart
66795bd67c36f7f9e1b580845b886a94	gfortran/0_12_openmp_tune/restart
66795bd67c36f7f9e1b580845b886a94	gfortran/1_12_openmp_mpi_tune/restart
66795bd67c36f7f9e1b580845b886a94	gfortran/2_12_openmp_mpi_tune/restart

Table 20: `md5sum` of the restart file using different compilers and different configure options with tune flags.

	NS (ε/δ)	**IDW** (ε/δ)	**WS** (ε/δ)	**BS** (ε/δ)
Mean salinity	1.88e-07 / 5.41e-09	6.71e-07 / 4.14e-08	3.18e-08 / 9.31e-10	2.21e-07 / 3.26e-08
RMS for salinity	1.77e-07 / 5.09e-09	6.60e-07 / 3.63e-08	3.14e-08 / 9.18e-10	1.23e-07 / 1.77e-08
STD for salinity	4.17e-07 / 3.85e-07	4.79e-07 / 5.80e-08	6.01e-09 / 3.07e-09	3.84e-07 / 2.32e-07
Min salinity	0.00e+00 / 0.00e+00	0.00e+00 / 0.00e+00	0.00e+00 / 0.00e+00	0.00e+00 / 0.00e+00
Max salinity	7.51e-10 / 2.12e-11	8.14e-08 / 2.34e-09	1.74e-09 / 4.96e-11	5.35e-11 / 3.40e-12
Mean temp [°C]	7.36e-07 / 7.97e-08	8.48e-07 / 7.58e-08	6.64e-08 / 5.43e-09	9.98e-07 / 1.72e-07
RMS for temp [°C]	5.13e-07 / 5.40e-08	9.93e-07 / 8.57e-08	5.74e-08 / 4.66e-09	2.61e-06 / 4.11e-07
STD for temp [°C]	8.96e-07 / 3.97e-07	1.19e-06 / 3.98e-07	7.51e-08 / 5.34e-08	6.14e-06 / 2.33e-06
Min temp [°C]	7.37e-09 / 1.28e-09	1.31e-09 / 3.27e-10	1.41e-08 / 1.77e-09	3.31e-03 / 1.91e-02
Max temp [°C]	6.00e-13 / 3.23e-14	6.60e-12 / 3.30e-13	9.95e-14 / 5.35e-15	5.98e-10 / 3.22e-11
Mean z [m]	1.33e-06 / 7.17e-05	5.12e-07 / 1.33e-06	9.27e-07 / 2.30e-05	6.31e-08 / 1.42e-07
RMS for z [m]	1.32e-05 / 4.16e-05	4.18e-07 / 1.07e-06	7.63e-07 / 2.03e-06	1.72e-07 / 3.86e-07
STD for z [m]	1.32e-05 / 4.19e-05	2.48e-06 / 3.38e-05	8.35e-07 / 2.24e-06	1.03e-06 / 2.08e-05
Min z [m]	2.86e-10 / 2.70e-10	1.00e-05 / 1.03e-04	1.00e-13 / 6.98e-14	7.85e-07 / 2.25e-06
Max z [m]	5.46e-05 / 3.52e-05	1.30e-06 / 2.33e-06	0.00e+00 / 0.00e+00	9.17e-06 / 1.39e-05
Min u [m/s]	9.99e-14 / 9.04e-14	5.70e-06 / 5.93e-06	0.00e+00 / 0.00e+00	7.30e-08 / 1.97e-07
Max u [m/s]	2.23e-04 / 1.27e-04	2.43e-06 / 4.56e-06	1.20e-08 / 1.01e-08	3.97e-07 / 1.74e-06
Min v [m/s]	2.95e-10 / 2.12e-10	1.42e-06 / 1.70e-06	1.00e-08 / 6.24e-09	3.99e-07 / 9.13e-07
Max v [m/s]	1.42e-04 / 6.72e-05	2.22e-06 / 2.37e-06	9.99e-14 / 9.65e-14	3.75e-05 / 9.55e-05

Table 21: Worst case differences on statistics between 45 runs on XT5 (different configure options, different decompositions, different compilers, different compiler flags).

the surface; this is because the largest differences occur at a position where there is only one layer which is subject to flooding and drying as shown in figure 19, using the pgi compiler as an example, where the ε of 7.40 10^{-6} for salinity is at an isolated point.

As to the magnitudes of the differences in table 22 compared to the corresponding magnitudes of the differences from pure serial runs, cf. table 17, we see that the discrepancies range from none with the PathScale compiler, i.e. no extra uncertainty is added by this compiler when including library support for openMP and MPI (which we already knew from table 18), to approximately a factor of two with the Cray compiler, i.e. the uncertainty is roughly doubled.

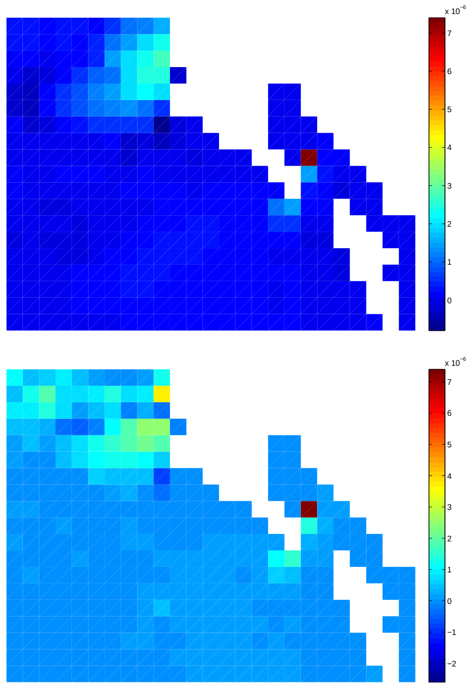


Figure 19: Pointwise differences for salinity in a subregion of the WS domain between IEEE and TUNE runs for the pgi compiler. Upper figure is the surface, lower figure is the bottom.

	NS (ε/δ)	**IDW** (ε/δ)	**WS** (ε/δ)	**BS** (ε/δ)
intel				
$\varepsilon_p(zlev, 1)$	6.80e-04/4.38e-04	4.70e-04 /8.41e-04	9.93e-06 /5.12e-06	2.06e-04/3.13e-04
$\varepsilon_p(salt, 0)$	3.03e-02/8.55e-04	6.25e-02 /1.79e-03	3.60e-06 /1.03e-07	1.45e-02/4.13e-04
$\varepsilon_p(salt, 1)$	3.98e-03/1.12e-04	4.22e-02 /1.20e-03	3.60e-06 /1.03e-07	1.53e-02/4.38e-04
$\varepsilon_p(temp, 0)$	1.08e-01/5.80e-03	1.27e-01 /6.36e-03	2.40e-06 /1.29e-07	2.44e-01/1.31e-02
$\varepsilon_p(temp, 1)$	1.48e-02/7.95e-04	4.33e-02 /2.17e-03	2.40e-06 /1.29e-07	7.24e-02/3.90e-03
pathscale				
$\varepsilon_p(zlev, 1)$	2.59e-04/1.67e-04	2.05e-04 /3.67e-04	8.16e-06 /4.21e-06	5.57e-05/8.46e-05
$\varepsilon_p(salt, 0)$	1.37e-02/3.88e-04	9.84e-02 /2.81e-03	3.10e-06 /8.85e-08	1.08e-02/3.08e-04
$\varepsilon_p(salt, 1)$	5.49e-03/1.55e-04	4.44e-02 /1.27e-03	3.30e-06 /9.42e-08	3.56e-03/1.02e-04
$\varepsilon_p(temp, 0)$	4.64e-02/2.49e-03	9.79e-02 /4.90e-03	2.00e-06 /1.07e-07	2.22e-01/1.20e-02
$\varepsilon_p(temp, 1)$	1.07e-02/5.76e-04	3.40e-02 /1.70e-03	2.00e-06 /1.07e-07	4.99e-02/2.69e-03
gfortran				
$\varepsilon_p(zlev, 1)$	3.49e-04/2.25e-04	1.04e-03 /1.86e-03	1.42e-05 /7.33e-06	6.64e-05/1.01e-04
$\varepsilon_p(salt, 0)$	9.75e-03/2.75e-04	9.09e-02 /2.60e-03	1.11e-05 /3.17e-07	7.35e-03/2.10e-04
$\varepsilon_p(salt, 1)$	2.93e-03/8.29e-05	4.22e-02 /1.20e-03	1.11e-05 /3.17e-07	1.01e-02/2.90e-04
$\varepsilon_p(temp, 0)$	2.01e-02/1.08e-03	7.79e-02 /3.89e-03	4.30e-06 /2.31e-07	2.15e-01/1.16e-02
$\varepsilon_p(temp, 1)$	1.25e-02/6.71e-04	5.16e-02 /2.59e-03	4.30e-06 /2.31e-07	4.40e-02/2.37e-03
cray				
$\varepsilon_p(zlev, 1)$	4.16e-04/2.68e-04	2.86e-04 /5.11e-04	9.05e-06 /4.66e-06	1.82e-04/2.77e-04
$\varepsilon_p(salt, 0)$	4.38e-02/1.24e-03	1.15e-01 /3.29e-03	1.16e-05 /3.31e-07	1.15e-02/3.28e-04
$\varepsilon_p(salt, 1)$	4.25e-03/1.20e-04	4.33e-02 /1.24e-03	1.16e-05 /3.31e-07	1.53e-02/4.36e-04
$\varepsilon_p(temp, 0)$	3.19e-01/1.71e-02	1.69e-01 /8.43e-03	5.60e-06 /3.01e-07	2.17e-01/1.17e-02
$\varepsilon_p(temp, 1)$	7.05e-03/3.79e-04	4.73e-02 /2.37e-03	5.60e-06 /3.01e-07	8.98e-02/4.83e-03
pgi				
$\varepsilon_p(zlev, 1)$	5.10e-04/3.29e-04	4.53e-04 /8.11e-04	1.70e-05 /8.75e-06	8.30e-05/1.26e-04
$\varepsilon_p(salt, 0)$	9.02e-03/2.55e-04	1.09e-01 /3.10e-03	7.40e-06 /2.11e-07	9.68e-03/2.77e-04
$\varepsilon_p(salt, 1)$	2.10e-03/5.94e-05	3.57e-02 /1.02e-03	7.40e-06 /2.11e-07	6.19e-03/1.77e-04
$\varepsilon_p(temp, 0)$	2.31e-02/1.24e-03	9.21e-02 /4.61e-03	6.60e-06 /3.55e-07	2.26e-01/1.22e-02
$\varepsilon_p(temp, 1)$	7.28e-03/3.91e-04	3.48e-02 /1.74e-03	6.60e-06 /3.55e-07	4.84e-02/2.60e-03

Table 22: Worst case pointwise differences per compiler when comparing four files; two serial (IEEE and TUNE) and two parallel (OPENMP_TUNE, OPENMP_MPI_TUNE) for all compilers that supports MPI on our cray XT5 system.

Finally, in table 23 we show the worst-case differences across all compilers. These numbers in combination with the numbers in table 21 are the ones that we should keep in mind when we proceed with the calibration/validation phase. The steps we have taken to reach these numbers provides the required confidence that we need to trust them.

	NS (ε/δ)	**IDW** (ε/δ)	**WS** (ε/δ)	**BS** (ε/δ)
$\varepsilon_p(zlev, 1)$	7.57e-04/4.88e-04	7.45e-04 /1.33e-03	1.57e-05 /8.10e-06	2.36e-04/3.58e-04
$\varepsilon_p(salt, 0)$	4.38e-02/1.24e-03	1.18e-01 /3.37e-03	7.90e-06 /2.25e-07	1.45e-02/4.13e-04
$\varepsilon_p(salt, 1)$	7.47e-03/2.11e-04	4.28e-02 /1.22e-03	7.90e-06 /2.25e-07	1.55e-02/4.44e-04
$\varepsilon_p(temp, 0)$	3.19e-01/1.72e-02	1.76e-01 /8.82e-03	5.10e-06 /2.74e-07	2.42e-01/1.30e-02
$\varepsilon_p(temp, 1)$	1.52e-02/8.16e-04	5.74e-02 /2.87e-03	5.10e-06 /2.74e-07	8.98e-02/4.83e-03

Table 23: Worst case pointwise differences when comparing four files; two serial (IEEE and TUNE) and two parallel (OPENMP_TUNE, OPENMP_MPI_TUNE) across all compilers that supports MPI on our cray XT5 system.

5.4.1 Study scalability across compilers and platforms

As another neat side effect of all these runs, we can easily construct scaling plots and cross-compare compilers, platforms etc. If certain compilers or platforms stick out, then we may try to analyze where the issues pertain to and if possible fix it. If we cannot fix it in the code, then we should get in touch with the compiler vendor and have the issue resolved.

An example with openMP scaling for a number of compilers is shown in figure 20 where the intel compiler performs worse than the rest: It scales pretty well but it has an off-set at 2 cores which it carries throughout.

Note that scale plots only make sense if the results for each point along the curve are identical; if they are not, what you are showing is only that different models run with different computational speed, which is just a trivial and not very interesting statement.

5.4.2 Comparative debugging of numerical problems

Assuming that some of the tests above or some of the nightly tests have revealed some kind of numerical deviation that we can not explain immediately. What do we do to deal with it? In this section we give a very

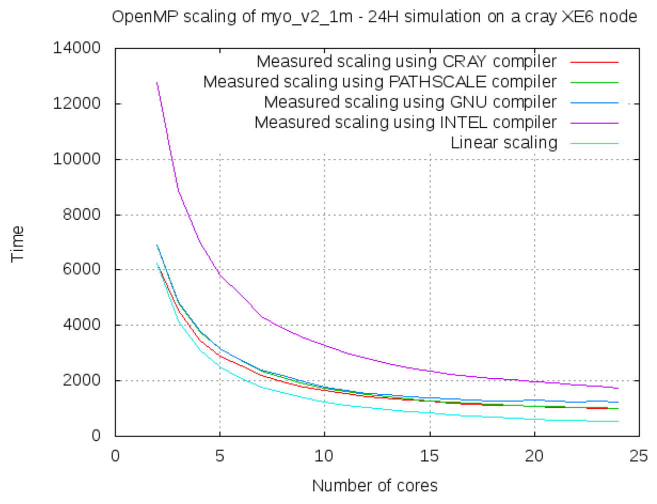


Figure 20: Sustained scaling with openMP for different compilers and compared to linear scaling.

brief overview of some of the techniques we use for more involved debugging purposes.

Beforehand though, it is important to realize that the techniques presented here are not restricted to the HBM code but are suited for debugging any time stepping simulation code. That is, codes that try to simulate the behavior of a complex system over time. We assume that the code keeps track of a set of prognostic variables, say $x_1, \ldots, x_n$ and that we have found that one of our runs produces results that are unexpected. To find the first place where one of these variables is facing an unexpected discrepancy might be a tedious and very time-consuming task. Now, recall why robbers rush to the bank - that is where the money is. Thus, in attempt to work more effectively with the code we have also searched for places that tend to be very time-consuming for us and debugging numerical problems tends to be such a place.

There are several ways to tackle numerical problems that may appear apparent after the analysis above and we here describe some of them:

- Add debug code to the existing code, then rebuild, rerun and cross-compare findings.

- Step through the two codes interactively from within two debug sessions and cross-compare findings as you go along.
- Use our debug framework where one basically instructs the debugger via code comments that one can add incrementally to the code.

We always approach these problems incrementally and let us assume that we have two incarnation of the program say A and B. It could be that A was build with configure option say '`--enable-openmp`' and B was build with configure option '`--enable-openmp --enable-mpi`' or it could be the same options but with different compiler flags, or the same options with different compilers or it could be two runs on two different platforms say A was running on an Intel CPU whereas B ran partly on an AMD CPU and partly on an nvidia GPU.

Assume that we are facing a problem with $\varepsilon_s(x)$. Using the first method above, we would typically add test-code to the application that would print the statistics after each piece of code (say subroutine) that would modify x. Running the new code we would then obtain two series of statistics $A(s(x))_{i\in 1,\ldots,t}$ and $B(s(x))_{i\in 1,\ldots,t}$ that we could cross-compare to pinpoint which point of time and which subroutine was causing the first discrepancy. However, the fact that the extra code will be added incrementally implies that we will need many rebuilds of the code and the manual process of adding code and rebuilding (and rerunning) is rather time-consuming.

The discrepancy is not always obvious to spot when looking only at statistics and another approach that we often use is simply to dump the entire field at relevant places in the code thus producing two sets of files $A(x)_{i\in 1,\ldots,t}$ and $B(x)_{i\in 1,\ldots,t}$. If we expect binary identical results of A and B it is sufficient to use `md5sum` on the two sets of files to find the point of time and the corresponding subroutine. If this is not the case then we need to post-process the two sets of files to find the point of time and the corresponding subroutine.

As revealed above these two very simple operations (adding statistical printouts and storing fields) are crucial to debug numerical problems incrementally. Unfortunately, this approach comes with severe drawbacks: Adding new code lines and rebuild means a new executable being generated by the compiler, and this again may result in different results, especially in situations with a code-bug present; all experienced developers have tried this. Not to mention that this approach is - mildly speaking - not easy to deal with when debugging across multiple tasks in an MPI application. Thus, we need to come up with an easier way that does not involve adding extra code or rebuilding the application to obtain the required information.

We have established a framework for both interactive and non-interactive debugging and validation using the totalview tvscript facility. The framework is based on code comments and thus far, we have implemented to parse the following F90 comments:

```
! TVSCRIPT : statistics [level]
! TVSCRIPT : dprintstat [level] printarg
! TVSCRIPT : cmd [level] cmdarg
! TVSCRIPT : ddump [level] filename field
```

Note that the statistics is meant to be statistics for the most relevant prognostic variables and thus tailored to the application at hand where as the `cmd` instruction is generic and can be used for any fortran application. Note that the optional `level` allows us to build up a hierarchy of debug prints and one will set a global upper bound when running the application and only comments whose `level` does not exceed this upper bound will be activated. The more debug output we produce the more information need to be post-analysed and it is consequently important to have some kind of filter mechanism. Thus far, the hierarchy approach has been sufficient for us.

When one adds the comment '`! TVSCRIPT: statistics`' the debugger will give detailed statistics for all relevant prognostic variables whenever the actionpoint is hit when running the code. This is something that coincides somewhat with the HBM internal `validate` subroutine. The output will be a list of blocks like the one shown below and contains more detailed information than the internal `validate` provide today. However, it is important to realize that the statistics is the statistics that emerges from the internal totalview implementations and their implementations does not coincide completely with ours. We use a *doubly compensated sum algorithm*[29] to ensure as accurate statistics as possible in the HBM implementation itself.

[29]The actual algorithm is the one invented by Priest which is an improvement of the famous summation algorithms by Donald Knuth and William Kahan.

```
Count:                479081
Zero Count:           0
Sum:                  4422764.84702221
Minimum:              5.73619261069032
Maximum:              18.7505818671698
Median:               8.02156057004163
Mean:                 9.23176842125279
Standard Deviation:   2.26401041287361
First Quartile:       7.41735188698377
Third Quartile:       11.1451235690212
Lower Adjacent:       5.73619261069032
Upper Adjacent:       16.7245665976456

NaN Count:            0
Infinity Count:       0
Denormalized Count:   0

Checksum:             55131
```

When one adds the comment '`!  TVSCRIPT: dprintstats <arg>`' the debugger will allow one to print statistics for the variable(s) whenever the actionpoint is hit, e.g.

```
! TVSCRIPT: dprintstat cmod_arrays‘cmp(1)%p(2,2:);
```

will print statistics for the salinity from all subsurface layers in area 1.

When one adds the comment '`!  TVSCRIPT: cmd <arg>`' the debugger will allow one to do execute commands whenever the actionpoint is hit, e.g.

```
! TVSCRIPT: cmd dprint cmod_arrays‘cmp(1)%p(1,924:927)
! TVSCRIPT: cmd dprint i; dprint j ; dprint k
! TVSCRIPT: cmd (if i==11) $stop
```

Finally, one can use the comment '`!  TVSCRIPT : ddump [level] filename field`' to get a binary dump of the field whenever the actionpoint is hit.

5.5 Analysing longer simulations

All the tests performed above are of a relatively short duration, i.e. simulating 6 or 24 hours. From these tests, we now know how sensible our implementation is with respect to platforms/compilers/flags/parallelizations and even some source rewrites[30], and we have fixed all the known bugs that

[30]For instance, the rewrite of `momeqs` required for the openACC support.

our modelling projects requires. Especially, we should now have a good understanding of how our operational setup performs, and we should be in a very good position to make a qualified choice of the compiler and run-configuration we wish to use for our production runs. Moreover, we have established lower bounds on calibration/validation efforts in the sense that it does not make sense to strive for deviations less than those presented in the worst-case studies.

This is important prior to the calibration and validation phases because then we can have confidence in the model implementation, we have a reference and a solid foundation; it would make little sense to continue long, time consuming model simulations if we had no evidence whatsoever of a sane implementation or if we knew that we carried important or even critical bugs. At this point, it is important to stress that we do not intend to reinvent the wheel and describe all the different kinds of analysis that one can think of and that is traditionally performed during model calibration-validation studies and studies of sensitivities to e.g. model forcing, choice of model parameters, etc. Our intention here is merely to reveal some of the pitfalls that one should keep in mind before going into these kinds of heavily man-time consuming studies.

It is therefore recommended to do one or more multi-year hindcast runs as part of the model testing prior to release of the software, the purpose being to reveal possible issues that were not covered by the short runs. And if issues are found, have them treated appropriately. Then we are ready to release the model code for production and the user can start the calibration-validation work which should of course never be finalized with experimental, unreleased code.

Issues we particularly look for with these longer simulations are:

- Run-time fluctuations from the reference; might indicate that a part of the implementation is not so healthy after all.
- During storms, both high and low water situations, is the model on its way to a blow-up due to e.g. strong currents or drying grid cells?
- During cooling or heating, do we see issues? E.g. freezing to soon, accumulation of heat.
- Volume balance; does an area loose water?

- Salinity balance; is salt mass preserved?

If we discover an issue, we need to uncover its origin; if it is a problem with the implementation it must be fixed in the code; if it is a problem with the forcing it should be dealt with there; is it a problem in the setup, it is the setup that should be corrected. It is important that we at this stage are capable of distinguishing between these different issues and deal with it at appropriate places, else the risk is that we will delay the present project and all other projects that rely on it.

A simple approach is to study statistics of model variables and this will often bring us a long way forward, but more sophisticated analyses can be performed as well. Here follows a brief description of what we do.

Let the model run for a longer period of time, e.g. one or more years and collect spatial minimum, maximum and average of relevant forcing variables and relevant prognostic variables over the entire period in say yearly chunks. If all statistical values are within expected ranges, fine. If not, locate and fix the problem before proceeding. Hopefully, we will be able to catch silly issues that might otherwise build up over time. A typical issue revealed in this way include drift in a prognostic variable which cannot be explained from physics and which is due to improper implementations of some parts of the code.

We have prepared a script, `long_stats.sh`, that extracts min and max and average for all the variables from the logfiles and thus provide ranges. It also plots min, max and average for all variables in all areas. It is important to study these plots and ensure that everything works as expected on a per area basis. If e.g. the forcing is off at certain timepoints then a refined calibration-validation study later on may be a waste of time. Moreover, if the area statistics of the prognostic variables is off at certain timepoints, then the pointwise behaviour (the one that we track with observations) will likely be off too and the calibration-validation study will be a waste of time.

One thing we would like to do in the future is to analyse how our epsilons generated from short-term model runs relate to uncertainties on multi-decade simulations. I.e. can we say anything about how large the expected error on a prognostic variable, say temperature, will be when you have epsilons of the shown magnitudes? Or can we say that in order to have a certain accuracy in the results from long simulations we must ensure epsilons smaller than a given order of magnitude? This is - to the best of our knowledge - an uncovered subject in the ocean modelling community in general, but it ought to have a high priority on the agenda due to the increasing interest in climate scenario modelling with ocean circulation models of the type like the HBM.

A Appendix: Survey of global variables

Here we give a brief survey of the most important global variables.

In table 24 we list the prognostic variables, and the major diagnostic variables are listed in table 25.

In table 26 we have listed parameters of the meteorological forcing which are read in from a file. Also shown in this table are derived meteorological forcing parameters, i.e. first the time-interpolated meteorological parameters and then the forcing parameters which are obtained from model diagnostics.

The most important index arrays are listed in table 27, and table 28 shows the variables that are related to description of the grid.

We have listed the major variables used for open boundary conditions and for nesting conditions in table 29.

Table 24: Prognostic global variables.

Internal name	Description	Located at
`un, u`	zonal component of current at new and old time steps	east face of grid cell
`vn, v`	meridional component of current at new and old time steps	south face of grid cell
`zn, z`	sea surface elevation at new and old time steps	surface grid points
`cmp`	component array to store sea temperature, salinity, and any optional passive tracers (e.g. biogeochemical tracers)	grid points
`tke`	turbulent kinetic energy	grid points
`diss`	dissipation of turbulent kinetic energy	grid points
`avv`	eddy diffusivity for momentum	upper face of grid cells
`t_soil`	temperature of a 3-layer soil model of the sea bed	grid points below sea bed
`ice`	component array to store ice thickness,	surface grid points

continues ...

Table 24: *... continued*

	ice concentration and snow cover	
`casus`	mask for ice coverage (no ice = .true.)	surface grid points
`tsnei`	ice/snow surface temperature	surface grid points
`ueis`	zonal component of ice velocity	east face of surface grid cell
`veis`	meridional component of ice velocity	south face of surface grid cell

Table 25: Diagnostic global variables.

Internal name	Description	Located at
`w`	vertical component of velocity	upper face of grid cell
`ui, vi`	zonal and meridional components of current, time-averaged for tracer advection	east and south cell faces, resp.
`dispv`	component array to store the eddy diffusivities for sea temperature, salinity, and any possible passive tracers	upper face of grid cells
`shear`	horizontal shear of horizontal velocity	south-east corner of grid cell
`stretch`	horizontal stretch of horizontal velocity	grid points
`div`	horizontal divergence of horizontal velocity	grid points
`eddyh`	horizontal Smagorinsky eddy viscosity	grid points
`eddyd`	horizontal divergence eddy viscosity	grid points
`rho`	local density of sea water	grid points
`press`	hydrostatic pressure	grid points
`geopot`	geopotential	grid points

Table 26: Global meteorological forcing parameters and derived quatities.

Internal name	Description
`wua, wun`	zonal component of 10 m wind at the surface grid points, at old and new time step of the meteo file (input)
`wva, wvn`	meridional component of 10 m wind at the surface grid points, at old and new time step of the meteo file (input)
`pla, pln`	mean sea level air pressure at the surface grid points, at old and new time step of the meteo file (input)
`atempa, atempn`	2m air temperature at the surface grid points, at old and new time step of the meteo file (input)
`humida, humidn`	relative humidity at the surface grid points, at old and new time step of the meteo file (input)
`clouda, cloudn`	cloud cover at the surface grid points, at old and new time step of the meteo file (input)
`precipo, precipn`	precipitation at the surface grid points, at old and new time step of the meteo file (input)
`wu, wv`	zonal and meridional components of 10 m wind at the surface grid points, interpolated to the main HBM time step
`pl, atemp, humid, cloud precip`	pressure, temperature, humidity, cloud cover, and precipitation at surface grid points, interpolated to the main HBM time step
`taux, tauy`	zonal and meridional components of wind stress over sea, available at the eastward and southward surface grid cell faces, resp., interpolated to the HBM time step
`hflux`	buyoancy-affecting part of the surface heat flux at grid points, interpolated to the main time step
`evapo`	evaporation at surface grid points, interpolated to the HBM time step
`tpe`	precipitation/evaporation temperature at surface grid points

Table 27: Major global index arrays used.

Internal name	Description
`mm1k`	permuted wet-point index look-up table
`ind`	i,j index pair corresponding to each wet-point in the surface
`kh`	number of k-levels at each horizontal location, i.e. defining the water column
`khu, khv`	number of active transport levels at the east and south face, resp., at each location
`khbnd`	number of k-levels at each location along the open boundary
`me1`	surface wet-point index look-up table
`mm1`	un-permuted wet-point index look-up table, should not be used or used with care
`krz`	i,j index pair and type (W,E,S,N) of each point on an z boundary of a domain
`kru, krv`	i,j index pair and type (W,E,S,N) of each point on an u or v nesting border, resp.
`idx`	lower/upper index pair of the openMP thread decomposition for easy look-up
`krq`	i,j index position of each point source (rivers)
`muvse1`	index of the 8 quantities of ice mechanics associated with each grid cell
`induvse`	position i,j along with type of each component of the ice mechanics

Table 28: Global variables describing the grid.

Internal name	Description
`hz`	static height of each grid cell (input)
`h`	dynamic height of each grid cell, `hz + z`
`hu, hv`	dynamic height of the eastward and southward faces, resp.,

continues ...

Table 28: ... *continued*

	of each grid cell
`h_old`, `h_new`	height of each grid cell at old and new time step for use in tracer advection
`ti`	topography at each horizontal grid point

Table 29: Global boundary and nesting variables.

Internal name	Description
`bndzk`	prescribed boundary conditions for S and T (and possible passive tracers) at the open, lateral boundaries (model input)
`bndz`	array to store boundary conditions for S and T (and possible passive tracers). That is, sponges at open lateral boundaries, and nesting conditions at nesting borders
`rwz`	sea level boundary values at open boundaries for main area and for nesting conditions at nesting borders
`zetac`	actual baroclinic correction to z at open boundaries, either as calculated from the density field or prescribed or combo
`zetac2`	prescribed baroclinic correction to z at open boundaries, read from file
`iceobc`	prescribed boundary conditions for ice variables at the open model boundaries (input)
`bndice`	array to store boundary conditions for ice variables. That is, That is, sponges at open lateral boundaries, and nesting conditions at nesting borders

B Appendix: Code styling

Following the ansi Fortran standard when you code gives you some assurance of the quality of your work in the sense that the syntax can be verified and that different compilers are expected to be able to understand each code statement, cf. chapter 5. But it does not guarantee that the semantics is such that the code behaves as intended when executed. There are lots of pitfalls in coding; we have seen far too many examples of code taken from textbooks that simply is not well suited for practical applications. Technical bugs and unfortunate technical approaches make it difficult or impossible to do satisfactory validation of model results, to obtain portability, robustness across platforms/compilers, and also to do further development and to add new features, incrementally. Most of these troubles can be avoided if one applies rules and styling throughout, and performs testing and technical validation as outlined in chapter 5 frequently.

It is inevitable that there will be 'silly' ways to use smart features of the Fortran language. Clearly, we want to avoid that and to recommend certain practices over others. The general ethos is to write portable code that is easily readable and maintainable. Code should be written in as general a way as possible to allow for unforeseen modifications. In practice this means that coding might take a little longer. This small extra effort is well spent, however, as maintenance costs - and headaches of your fellow developers - will largely be reduced over the lifetime of the software.

The rather lengthy description of our programming rules is available for those persons with affiliation to the relevant modelling and development projects[31] but a comprehensive except will be given in the present chapter. That is, we will here describe the main concepts behind our styling rules.

The above-mentioned document contains a collection of concepts/rules/styles for documentation of the code, for version management, as well as defining standards for writing code and guide on practical implementation issues. The coding standards are designed to improve code readability and maintainability as well as to ensure, as far as possible, its portability and the efficient use of computer resources as well as of manpower.

Our code styling is based somewhat on literature survey and previous prac-

[31] http://hbmtrac.dmi.dk/browser/trunk/cmod/doc/web/f90style_dmi.txt

tice but most of all on experience. The rules are designed to encourage good programming practice, to simplify maintenance, and to ease readability of the code by establishing some basic common style rules. It must be emphasized that the style rules have for sure not been prepared to put restrictions into the developer's working routines. The developers are not expected to know every single detail by heart before they can contribute to the development. If some part of our styling rules for one reason or the other doesn't make sense, e.g. is outdated, it should be changed.

Some of the most important aspects of our styling rules dictate, in summary, that we

- use standard ansi Fortran95
- use no obsolete constructs (e.g. no `GOTO`)
- always use `IMPLICIT NONE` and explicit declaration of all variables in all source files
- assign explicit scope, `PRIVATE` or `PUBLIC`, on all module variables
- use explicit `INTENT`ion on all arguments in all subroutines
- always explicitly apply the `ONLY` clause with the `USE` statement
- have automatic interface blocks provided through `MODULE`s
- allow no namespace cluttering
- allow no implicit `KIND`s in declaration or in intrinsics
- allow no unused variables, no unused arguments
- allow no unnecessary hardcoding

and that the source code must

- pass building/running with checks for uninitialized variables on the stack/heap and boundary checking on all the platforms and with all compilers available to us (see chapter 5)
- be able to run parallel, all configure incarnations, i.e. serial, OMP, MPI, and combinations hereof, and produce the same results (see chapter 4)

So far, no enforcement strategy has been agreed. It is up to the individual developer to behave nicely. It is obviously important, however, that standards are adhered to - particularly that the documentation is kept up to date with the software; and that the software is written in as portable a manner as possible. If standards are not adhered to the code will not be exchangeable.

Automatic tools (e.g. a script) has been devised to test for compliance with the standards. This has been included into the nightly test suite (see chapter 5) as a pre-processing to the compilations.

B.1 Documentation and version management

Documentation may be split into two categories: external documentation outside the code, and internal documentation inside the code. In order for the documentation to be useful it needs to be both up to date and readable. For readability, all documentation, both internal and external, must be in English.

The source code including ”everything”, that is build/run scripts, various reports and test-cases, documentations (including this paper), is managed with a version control system, gluing it all together and keeping track of all revisions.

External documentation:
In most cases this will be provided at the project level or in a separate doc folder, rather than for each individual routine. Preferably, it includes the following:

- Top Level Scientific documentation: This sets out the problem being solved by the project and the scientific rationale for the solution method adopted. This documentation should be independent of (i.e. not refer to) the code itself.

- Implementation documentation describing the particular implementation of the solution method described in the scientific documentation; a road map of the project; how to compile, link and run the project; the use of the project, preferably - if appropriate - advising test-cases with description of sensible results; the modification history for larger chunks of code revisions in a separate Changelog (while individual files' history is kept track of by the version management system); ongoing considerations, discussions, design issues, feature implementation notes, etc, concerning the code together with descriptions on postponed tasks and planned activities.

Internal documentation:
This is to be applied at the individual routine level. There are four types of internal documentation, all of which must be present.

- Procedure headers: Every subroutine, function, module etc should have a header. The purpose of the header is to describe the function of the routine, probably by referring to external documentation, and to document the variables used within the routine. All variables used within a routine must be declared in the header and, preferably, commented as to their purpose. The intention, type, kind, and dimension of all arguments must be explicitly stated in the attribute list of the argument declaration.

- Section comments: These divide the code into logical sections and may refer to the external documentation. These comments must be placed on their own lines at the start of and at the indent level of the section they are defining.

- Other comments: These are aimed at a programmer reading the code and are intended to simplify the task of understanding what is going on. These comments must be placed either immediately before or on the same line as the code they are commenting.

- Meaningful names: Code is much more readable if meaningful words are used to construct variable and subprogram names.

Version management:
Subversion (svn) has been chosen for HBM code development project. The agreements made for proper behavior in a code development project are usually specific to each particular development project. There is a document describing the HIROMB-BOOS co-operative guidelines[32] which can be found in the MyOWP6 document library[33]. That document aims towards answering specifically how development is done and what process is followed. These guidelines are internal to that particular project, but the body of the document has been adapted from the very general KDE TechBase SVN

[32]MYO-BAL-HBMDevGuideline_20100204.pdf
[33]`http://intranet.myocean.eu/share/page/site/WP6Room/documentlibrary`

Commit Policy[34] dual licensed under the GNU Free Documentation License 1.2 and the Creative Commons Attribution-ShareAlike License.

B.2 General styling rules

Here we describe some of our general styling rules.

ANSI standard:
The code must be compliant with the ANSI Fortran 95 standard. We use free format syntax and allow a maximum line length of 80 characters. The ANSI standard allows a line length of up to 132 characters, however this could cause problems when viewing, or if print-outs have to be obtained.

If exceptions from the ANSI Fortran standard occur they must be confined to single statements which are then thoroughly documented. At present, two "valid" exceptions from the ANSI Fortran standard are accepted; that is the use of the routines `EXIT([status])` and `FLUSH(unit)` which are non-ANSI Fortran 90 or 95 standard but which are both supported by most compilers. During configuration care is taken as to test for availability and act accordingly, i.e. compilers that do not offer this feature will still build fine without requiring any manual intervention.

Use of CCP flags:
In general, we do not allow `CCP` flags, being not a part of the standard; the standard has no defined preprocessor flags. However, some preprocessor flags like `_OPENMP` and `_OPENACC` are part of the the openMP and openACC standard, respectively. We use two preprocessor flags `_OPENMP` and `MPI` in the current implementation. The reason being that we wish to be able to run in serial, with openMP, with MPI, as well as with MPI and openMP in conjunction and we wish to use the same source code for all these runs.

The compiler **must** define `_OPENMP` according to the openMP standard when one builds with openMP support. Thus, one can safely use constructs like:

```
#if defined (_OPENMP)
  ...
#endif
```

assuming that the compiler understands openMP. The symbol `MPI` on the other hand is not forced by the MPI standard. Just as for openMP, MPI

[34] `http://techbase.kde.org/Policies/SVN_Commit_Policy`

needs library support, i.e. some functions **only** exists if the platform support MPI (assuming that we build and link with the library).

We still wish to be able to build and run on our desktop machines without support of MPI, and this implies that we cannot have code that assumes the existence of MPI.

It should be mentioned that this does not imply that we will accept the `_OPENMP` and `MPI` symbols scattered all over. We should confine this to appear only in the related module, `dmi_omp` and `dmi_mpi`, respectively, and only there.

Fortran source code files that need pre-compilation are identified by their `.F90` extension (compared to the regular `.f90`).

Precision and kind issues:
The application is a 64/32-bit application, i.e. 64bit reals and 32bit integers. Usually this can be specified by compiler flags like e.g. "`-r8 -i4`" but unfortunately this is not standard in any way and different compilers behave differently; some do not have a `-i4` flag, some interpret a `-r8` flag such that variables of type `real` become 64bit precision reals (i.e. of `KIND=8`) but have no influence on constants or on real variables or constants passed to intrinsic functions, etc. In fact, there is no standard for specifying the precision of both variables and constants and their interpretation with respect to intrinsics, in other ways than carefully stating the wanted `KIND` explicitly in the Fortran code.

`KIND` issues relate to maintaining consistency between caller and callee, to type casting and to precision, and it is in the hands of the developer to assure that we don't run into troubles with these. This is extremely important for portability, for performance and for numerical accuracy. Failing to do so will most likely cause a blow-up at runtime, or at least cause inconsistencies between results across platforms, compilers and configure incarnations, or there will be a penalty due to run-time type conversions, if the code will compile at all, that is; the code will for sure have potential for misinterpretations by the reader of the code.

You must always explicitly state the wanted precision through the `(KIND)` specification in the declaration of the variables, using the syntax of appending `_KIND` to constants, and applying the `KIND` argument to intrinsics, i.e.

```
real(8)                :: x, y
integer(4), parameter :: i = 17
x = 42.0_8
y = x + real(i,8)
```

Be explicit, write what you mean:
It is generally recommended that we write what we mean as clear and explicit as possible to avoid "clever" guessing from the compiler and ingenuous speculations from our fellow developers. There exists a bunch of language constructs that can help us with that, some of which we have adapted into our set of styling rules:

`IMPLICIT NONE` must be used in all program units. This ensures that all variables must be explicitly declared, and hence documented.

To avoid namespace cluttering with modules, it is a must to explicitly specify which of the variables, type definitions, etc, defined in a module are to be used from the respective program unit, i.e. always apply the `ONLY` attribute with the `USE` statement and all module variables, type definitions, and subroutines must always be explicitly defined in the module as either `PRIVATE` or `PUBLIC`, depending on their intended scope.

With variable declarations, always use the `::` notation, even if there are no attributes, always explicitly state the `INTENT`ion of arguments in the list of declarations (also if it's `INTENT(INOUT)`), and always declare the length of a character variable using the `(len = )` syntax.

Fortran compilers automatically provide explicit interface blocks between routines following a `CONTAINS` statement (e.g. in a `MODULE`). The interface blocks are also supplied to any routine `USE`ing the module. Thus, it is possible to design a system where no interface blocks are actually coded and yet explicit interface blocks are provided between all routines by the compiler. The way to do this is to 'modularize' the code at the Fortran `MODULE` level, i.e. to place related code together in one `MODULE` after the `CONTAINS` statement.

"Magic numbers" should be avoided; use a variable, possibly a `PARAMETER`, with a meaningful name to store the value.

Recursive:
In general, we do not recommend using recursive routines due to lack of efficiency; they tend to be inefficient in their use of cpu and memory. There is so far one exception, and that is to structure the computational flow of the nested hydrodynamics as shown in section 3.2.

The "plug compatibility" rule:
A HBM code shall refer only to its own modules and subprograms and to those intrinsic routines included in the Fortran standard.

There may, however, be a need to use third-party libraries such as e.g. GRIB and NetCDF libraries, and there is for sure a need to couple externally developed tracer models (e.g. biogeochemical or suspended particulate matter models) to HBM.

We would not like to jeopardize the portability of HBM and we therefore need to establish a framework that will keep the pollution off the building of the HBM core code. We can separate the code into the following subsets:

- `*.f90` files (and a few `*.F90` files) constituting the set of HBM core source files
- a single Fortran file, e.g. `coupler.f90` or `nc-output.f90`, that bridges the gap between HBM and the coupler/output functionality
- `*.f90` files constituting the coupler/output code, either available as part of HBM (one need to select them at configure time to have them build into the executable, though) or available in an external library.

The way we have accomplished this is by using autoconf/automake.

Unused variables:
Generally, just as all used variables much be explicitly declared, we require that all declared variables and arguments are actually used. That is, we do not allow any unused variables anywhere in the code. There are, however, two exceptions to this rule:

The first one is when one compiles without MPI. In that case, by use of

```
#if defined (MPI)
  ...
#else
```

```
    ...
#endif
```

some portions of the dmi_mpi code only constitute subroutine interfaces with no body contents, and thus there will unavoidably be unused arguments. We will allow unused arguments with `intent(in)` to be present in our code at a dedicated segment of the dmi_mpi.F90 file, not scattered all over the source code.

The second exception is with the `coupler.f90` which, in the default case where no tracer model is explicitly attached at configure time, is merely an empty stub consisting of only the interfaces. In that case we allow for unused arguments with proper `INTENT` in all subroutines in the coupler module file `default_coupler.f90` and only there.

No hard coding:
The project should be written so that it is as time-space resolution independent as possible. The resolution and other specific feature choices particular to a given setup must be separated from the general core code, and if needed, must be adjusted or changed at configure time or by reading in from files (e.g. `NAMELIST`s) at run-time.

Aliasing:
There are rules against aliasing in the ansi standard. The rules against aliasing allows for better optimization by the compiler (one of the main reasons that Fortran generally optimizes better than C). For this reason, and because the code will be very difficult to understand and to maintain, aliasing is banned. It can, however, be very difficult or even impossible for the compiler to figure out and warn against aliasing, so it is solely in the hands of the developer to carefully review the code. We can demonstrate it by an example:

A routine has two dummy arguments; one being only referenced, the other being assigned:

```
subroutine foo (a,b)
  real(8), intent(in)  :: a
  real(8), intent(out) :: b

  ... =  ... a ...
  b   =  ...
end subroutine foo
```

At compile time, the compiler will help you check that the intention of the arguments is also fulfilled in the body code of `foo`. Seen from the compiler's point of view, the same effect is implied on the corresponding actual arguments. The caller, however, may not know the intentions, so at compile time you might get through with nasty things like:

```
real(8)          :: c
real(8), target  :: d
real(8), pointer :: e

call foo( c,c )

e => d

call foo( d,e )
```

This is not allowed; if it was, what would the expected behavior be of such a code?

B.3 Dynamic memory

The use of dynamic memory is highly desirable as, in principle, it allows one set of compiled code to work for any specified resolution (or at least within hardware memory limits); and allows efficient reuse of work space memory. Care must be taken, however, as there is a risk for inefficient memory usage, particularly in parallelized code. For example heap fragmentation can occur if space is allocated by a lower level routine and not deallocated before control is passed back up the calling tree. On the other hand, allocating/deallocating in a lower level routine is expensive. So a reasonable solution must be implemented.

There are three ways of obtaining dynamic memory in Fortran 95:

- Automatic arrays: These are arrays initially declared within a subprogram with bounds depending upon variables known at runtime e.g. variables passed into the subprogram via its argument list.
- Pointer arrays: Array variables declared with the `POINTER` attribute may have allocated space at run time by using the `ALLOCATE` command. We recommend using pointers with care because Fortran pointers are just an extra name (an alias) for something else, are not memory addresses as in other languages, are not a specific data type of its own, but should rather be understood as an attribute to the other

data types, and they have both type and rank which must agree with the corresponding target.

- Allocatable arrays: Array variables declared with the `ALLOCATABLE` attribute may have allocated space at run time by using the `ALLOCATE` command. However, unlike pointers, allocatables are not allowed inside derived data types.

Recommendations on dynamic memory allocation:

- Use automatic arrays in preference to the other forms of dynamic memory allocation, locally in subroutines.
- Temporary space allocated locally in a routine using pointer arrays or allocatable arrays as described above should be explicitly freed using the `DEALLOCATE` statement.
- In a given program unit do not repeatedly `ALLOCATE` space. This will almost certainly generate large amounts of unusable memory. Instead, `DEALLOCATE` it and then `ALLOCATE` a larger block of space.
- Preferably, always test the success of a dynamic memory allocation and deallocation. The `ALLOCATE` and `DEALLOCATE` statements have an optional argument to let you do this.

Heap allocation is very expensive compared to stack allocation (**all systems**, **all compilers**). Heap allocation **at runtime** requires a system call (implying a kernel context switch and later interruption once the kernel has the memory for you) and some heuristics dealing with with heap fragmentation. There are even savings during compile time since every decent compiler will try to collect allocations and help the runtime lib doing a better job when it eventually hands off the request to the kernel.

As a rule of thumb, one should only use heap space to store global variables that are allocated **together** at the beginning of the program in as large chunks as possible and are deallocated at the end of the program. The stack can be a limited resource and there may be cases where heap allocation for the global vars is the only feasible approach. Though the stack can be very large it is not unlimited, and it can be very expensive to use the stack repeatedly for large arrays.

Recommendations with respect to contiguous data:
It is always advisable to keep data contiguous for performance. In some situations, we have seen that a couple of compilers fail to handle non-contiguous data correctly (due to bugs in those compilers failing to handle copy-in/copy-out correctly), so if you want to use those compilers you must avoid non-contiguous data for correctness as well.

Looking at the OpenMP 3.0 specification[35] it seems that the specifications have been tightened. A careful look at "2.9.3.2 shared clause", page 88 and the example A.29, page 231 suggests that we must avoid non-contiguous shared data in OpenMP parallel sections:

... *Under certain conditions, passing a shared variable to a non-intrinsic procedure may result in the value of the shared variable being copied into temporary storage before the procedure reference, and back out of the temporary storage into the actual argument storage after the procedure reference. It is implementation defined when this situation occurs.* ...

This gave rise to a complete code review and we found several places where we had constructions that are fully compliant with the Fortran 90 or 95 standard but doubtful with respect to these newer, more strict and, above all, *implementation dependent (!)* OpenMP 3.0 specifications.

So, for correctness as well as for performance and portability, the use of non-contiguous variables should be avoided, and especially, as shared variables in OpenMP parallel sections non-contiguous data is banned.

B.4 Obsolete and banned Fortran features

This section details features deprecated in or made redundant by Fortran 90 or later language dialects. We also ban features whose use is deemed to be bad programming practice as they can degrade maintainability and efficiency of the code. Some old Fortran 77 constructs have died or will most likely die in future Fortran versions, some have better alternatives in the present Fortran version. Here follows a list of banned Fortran features with suggested permitted alternatives.

[35] `http://www.openmp.org/mp-documents/spec30.pdf`

Table 30: Banned Fortran features.

Banned	Alternative
`COMMON` blocks	use a `MODULE` instead
`EQUIVALENCE`	use `POINTER`s or derived data types instead
Assigned and computed `GO TO`	`SELECT CASE` construct
Conditional `GOTO` statement	you may easily find a better way of branching your code, e.g. by use of `IF`, `CASE`, `DO WHILE`, `EXIT` or `CYCLE` statements
Arithmetic `IF` statements	use the block `IF ELSEIF` construct
`PRINT` statements	write to unit `iu06` which is specific to each MPI task, e.g. `WRITE(iu06,*)`
Never use tabs	Use blank space instead; this will ensure that the code looks as intended ... and tabs are not allowed by the Fortran standard
`SYSTEM` calls	better wrap the application into a script that does the necessary system related handling
`STOP`	use `CALL EXIT(status)` (see above under **ANSI Standard**)
`FORMAT` statements	use `character` parameters or explicit format specifiers inside the `read/write` statement
I/O routines' `END` and `ERR`	use `IOSTAT` instead
`DO`-loops that terminate in some other way than with `ENDDO` or `EXIT`	
Floating point `DO`-loop control variables	always use `INTEGER` control variables
Jump to `ENDIF/ENDDO` or into `IF/DO` blocks from an outer block	
`PAUSE`	
`BLOCK DATA`	
`ENTRY`	a subprogram may only have one entry point; use separate subroutines or possibly overloading (polymorphism)

There are some other banned features that requires some words:

- Reserved keywords as variable names. To avoid namespace cluttering variable names are not allowed that already have a meaning in FORTRAN 90. That is, you cannot introduce a variable called `IF` or `USE`, or which shares its name with intrinsic functions or names from loaded libraries (e.g. `omp_lib` and `mpi`). If you really intend to do overloading of predefined functions, the code documentation should be very particular on its intention and usage!

- Functions with side effects, i.e. functions that alter variables in their argument list or in modules used by the function; or one that performs I/O operations. This is very common in e.g. C programming, but can be confusing. Also, inefficiencies can be avoided if the compiler knows that functions have no side effects.

- Implicitly changing the shape of an array when passing it into a subroutine. Although actually forbidden in the standard it was very common practice in FORTRAN 77 to pass N-dimensional arrays into a subroutine where they would, say, be treated as a 1 dimensional array. This practice, though banned in Fortran 90, is still possible with external routines for which no Interface block has been supplied. This may only work due to assumptions made about how the data is stored: it is therefore unlikely to work on a massively parallel computer. In fact, there is no well-documented standard for mixing F77 argument-passing-style into F90 code; the result being *at best* undetermined. Hence the practice is banned!

- The `CONVERT` in `OPEN` statements to specify the endianess of the binary file, e.g.

  ```
  open(unit=lun,file=trim(filename),convert='big_endian',iostat=ios)
  ```

  since this is not supported by the standard. Instead, you need to check with your compiler of choice how to select endianess though compiler flags, like e.g. `-convert big_endian` for use with `ifort` at compile time, or like e.g. `export GFORTRAN_CONVERT_UNIT="big_endian"` for use with older versions of `gfortran` at run time. Compilers that have flags for handling endianess are handled automatically by the autoconf setup.

- Labelled `DO` constructs like

  ```
  DO 77 i=1,n
    ...
  77 CONTINUE
  ```

Instead you should use `DO-ENDDO`, possibly labelled, like

```
iloop: DO i=1,n
   ...
END DO ! iloop
```

C Appendix: Other ocean models

C.1 HBM at DMI (future experiments)

We have described the test-case that we are using today, i.e. MyO V2, in details in the paper but the experiments with the somewhat artificial DMI large test-case have shown that we can use the proof-of-concept MPI implementation to run real cases too. In table 31 we describe the next version, MyO V2.1, of our setup which is planned for production during the third quarter of 2012. The I_r number for this setup is 60.6. This case has increased horizontal resolution in the BS domain and it can be seen as an intermediate step towards finer resolution in the entire region. As figure 21 indicates, we can run this somewhat larger case within the same operational time window as we do today with the current MyO V2 setup; it is only a matter of how many computer resources we want to spend. For comparison, we have included timing and resources as obtained on our local Cray XT5 for a beta version of the MyO V2.1 setup (using 5 MPI tasks) as well as for the current MyO V2 setup (using 1 MPI task) in table 40.

Domain	**Res. [n.m.]**	**mmx**	**nmx**	**kmx**	**iw2**	**iw3**	**dt [sec]**	I_r
IDW	0.5	482	396	77	80884	1583786	12.5	12.0
NS	3	348	194	50	18908	479081	25	1.8
WS	1	149	156	24	11581	103441	25	0.4
BS	1	720	567	122	119334	6113599	12.5	46.3

Table 31: The next version of the MyO operational model, MyO V2.1. The I_r number for this setup is 60.6.

In table 32 and table 33 we describe two variants of finer resolution setups that we at the time of writing are working on, aiming at future operational applications. We have tried to run the first variant and the scaling of this on our local system is illustrated in figure 22. Moreover, figure 23 shows how the four sub-domains nest to each other. The I_r numbers for these setups are 89.6 and 227.4, respectively.

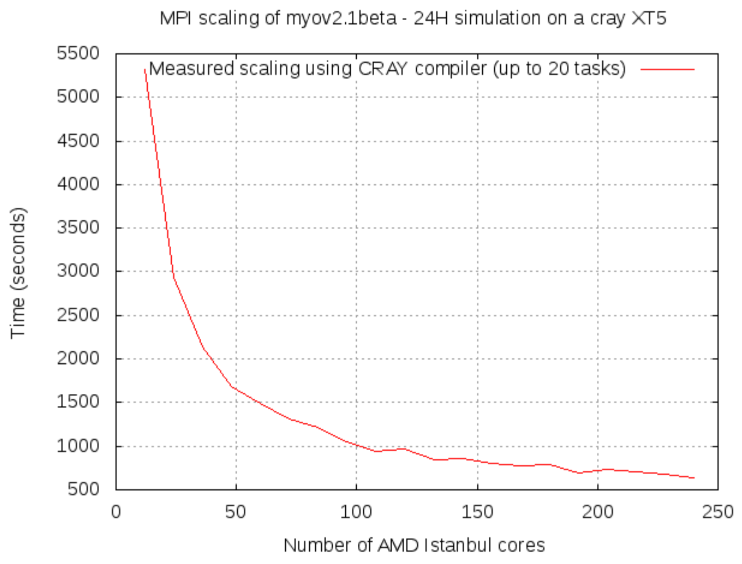


Figure 21: Scaling of the MyO V2.1 test case using automatically generated I-slices performed on our local Cray XT5 with 12 openMP threads on each MPI task and one MPI task on each node.

C.2 HIROMB at SMHI

Lars Axell, SMHI, reports that their official HIROMB consists of two domains, a 3 n.m. North Sea and a 1 n.m. Inner Danish Waters and Baltic Sea, cf. table 34. The two domains are run separately, i.e. it is not one nested setup.

Moreover, it takes roughly 15 minutes to run a 24 hour forecast using 32 MPI tasks during summer and roughly 20 minutes to run a 24 hour forecast using 32 MPI during winter.

C.3 GETM at FCOO

Jesper Larsen, The Danish Maritime Safety Administration, reports that their official GETM setup consists of two separate domains, a 1 n.m. setup for the North Sea/Baltic Sea and a 600 meter setup for the Inner Danish waters, cf. table 35.

Domain	Res. [n.m.]	mmx	nmx	kmx	iw2	iw3	dt [sec]	I_r
IDW	0.5	482	396	77	80884	1583786	10	15.0
rNS	3	220	127	46	15593	409049	20	1.9
WSNS	1	389	208	110	32515	1560598	10	14.8
BS	1	720	567	122	119334	6113599	10	57.9

Table 32: The next HBM experiment at DMI - variant 1. The I_r number for the setup is 89.6.

Domain	Res. [n.m.]	mmx	nmx	kmx	iw2	iw3	dt [sec]	I_r
IDW	0.25	964	792	77	323536	6335114	5	120.1
rNS	1	660	381	46	140337	3687912	10	34.9
WSNS	1	389	208	110	32515	1560598	10	14.8
BS	1	720	567	122	119334	6113599	10	57.9

Table 33: The next HBM experiments at DMI - variant 2. The I_r number for the setup is 227.4.

Domain	mmx	nmx	kmx	iw2	iw3	dt [sec]	I_r
BS01	735	752	47	144237	1918522	25	7.3
NS03	350	415	50	34410	506074	50	0.96

Table 34: HIROMB production at SMHI.

Domain	mmx	nmx	kmx	iw2	iw3	dt [sec]	I_r
NS1C	1371	1203	60	335823	20149380	180	10.6
DK600	857	820	60	177806	10668360	90	11.2

Table 35: GETM production at FCOO

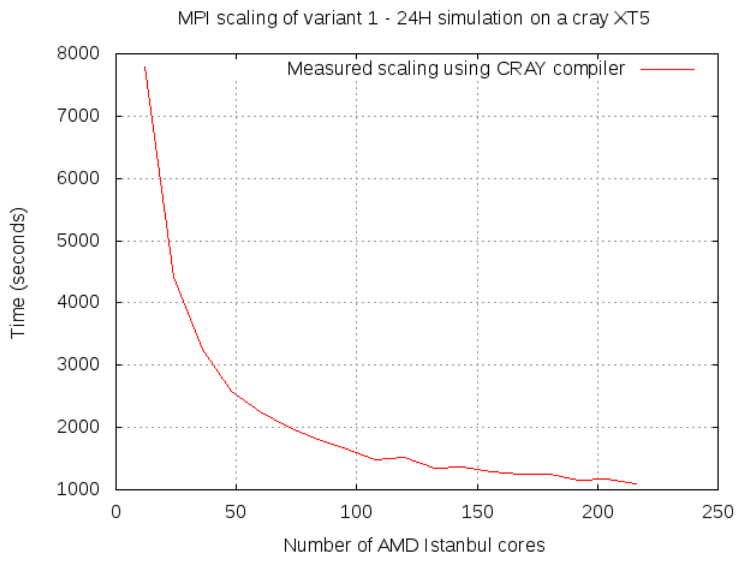


Figure 22: Scaling of the the next HBM experiment at DMI - variant 1 using automatically generated I-slices performed on our local Cray XT5 with 12 openMP threads on each MPI task and one MPI task on each node.

Moreover, they are using 207 tasks to run a 54 hour forecast for NS1C and it takes roughly 23 minutes and 126 tasks to run a 54 hour forecast the DK600 setup and that takes roughly 25 minutes.

C.4 Nemo at ECMWF

Kristian Mogensen, ECMWF, reports that ECMWF in 2012 is running nemo with the global ORCA1 grid but they have started to look into the larger ORCA025 grid, cf. table 36. They are coupling NEMO to IFS in an EPS setup with 51 members.

With 96 MPI tasks running on the Power6 platform at ECMWF it takes 150 minutes to complete an ORCA025 run without ice and approximately 300 minutes to complete a run with ice. The forecast length for these runs is 744 hours (a month).

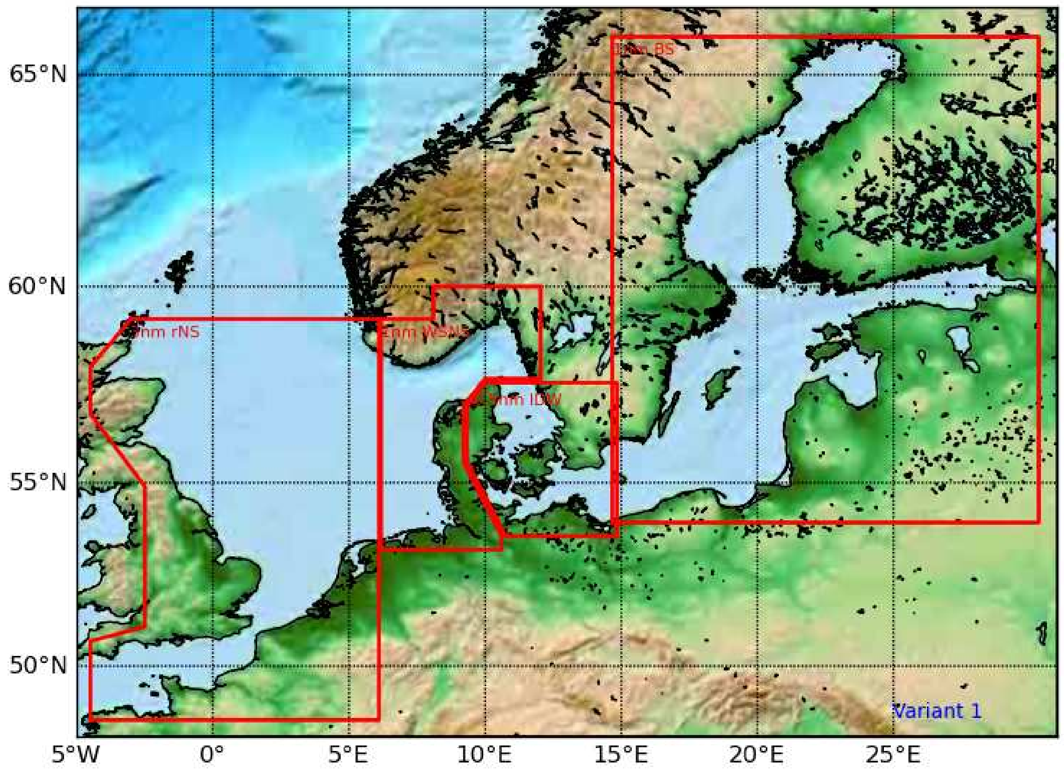


Figure 23: Illustration of new test domain.

C.5 Nemo at Mercator

Fabrice Hernandez, Mercator, reports that the largest test-case that they run is a global 1/12 degree setup, cf. table 37. It takes 5 hour on 64 NEC sx9 CPUs to run a 1-week forecast and somewhat surprising they only run this once a week. The total amount of output for 1-week is 75 Gb. The model runs every week on the MeteoFrance machine SX9 called Yuki. The run needs 63 CPUs for the model and the coupling scheme (PALM) needs 1 CPU which means 64 tasks in total. An NEC sx9 node has 16 CPUs, which means that the forecasting system run uses 4 NEC sx9 nodes.

Domain	**mmx**	**nmx**	**kmx**	**iw2**	**iw3**	**dt [sec]**	I_r
ORCA1	362	292	42	$\approx$ 66414	$\approx$ 2789388	3600	0.07
ORCA025	1442	1021	75	$\approx$ 809755	$\approx$ 60731632	1200	4.8

Table 36: Nemo production at ECMWF (ORCA1) and Nemo tests at ECMWF (ORCA025).

Domain	**mmx**	**nmx**	**kmx**	**iw2**	**iw3**	**dt [sec]**	I_r
Global 1/12°	4320	3058	50	660528000	342299000	360	90.1

Table 37: Nemo production at Mercator.

C.6 HYCOM at DMI

Till Andreas Rasmussen, DMI, reports that the regional NAA setup for HYCOM-CICE that they run in production in 2012 is defined as shown in table 38.

Domain	**mmx**	**nmx**	**kmx**	**iw2**	**iw3**	**dt [sec]**	I_r
NAA	1228	1213	29	606551	$\approx$ 11099883	300	3.5

Table 38: HYCOM production at DMI

They are using 144 MPI tasks to complete a 90 hour forecast within 45 minutes.

C.7 HYCOM at NRL

Joe Metzger, United States Naval Research Laboratory, reports that the largest HYCOM setups that they have tried to run are the global 1/12 degree and 1/25 degree setups, cf. table 39.

On a cray XE6, they do a 72 hour forecast of the 1/12 degree setup in 64.28 minutes using 503 tasks or in 31.85 minutes using 1001 tasks or 17.13 minutes using 2034 tasks or 9.55 minutes using 4074 tasks.

On a cray XE6, they do a 24 hour forecast of the 1/25 degree setup in

Domain	**mmx**	**nmx**	**kmx**	**iw2**	**iw3**	**dt [sec]**	I_r
NRL 1/25°	9000	6595	32	35900000	≈ 861600000	100	816.5
NRL 1/12°	4500	3298	32	9100000	≈ 218400000	240	86.2

Table 39: HYCOM setups at NRL.

72.38 minutes using 1001 tasks or 36.88 minutes using 2045 tasks or 19.95 minutes using 4043 tasks.

C.8 Summary

In table 40 we have tried to include all the setups that we have heard about and sorted them according to their I_r numbers. We have not tried to cross-compare the NEC vector cpus with conventional CPUs since that would not make much sense.

Model	**Domain**	**cores**	**Time** [sec]	**CM**	I_r	I_r/CM [x10^{-3}]
HYCOM	NRL 1/25°	4043	1235	83218	816.5	9.8
HBM	DMI variant 2	N/A	N/A	N/A	227.4	N/A
Nemo	Mercator 1/12°	N/A	2571	N/A	90.1	N/A
HBM	DMI variant 1	108	1484	2671	89.6	33.5
HYCOM	NRL 1/12°	4074	198	13467	86.2	6.4
HBM	DMI MyO V2.1beta	60	1477	1477	60.6	41.0
HBM	DMI MyO V2*	16	808	215	14.0	65.0
HBM	DMI MyO V2	12	1428	286	14.0	49.0
GETM	FCOO dk600	126	667	1401	11.2	8.0
GETM	FCOO ns1c	207	613	2115	10.6	5.0
HIROMB	SMHI	32	1050	560	7.3	13.0
Nemo	ECMWF experiment	96	435	696	4.8	6.9
HYCOM	DMI naa	144	720	1728	3.5	2.0
Nemo	ECMWF production	N/A	N/A	N/A	0.07	N/A

Table 40: Summary of the reports that we have received where we have normalized the time so that we report what it takes to do a 24 hour simulation. The number **CM** is shorthand for the compute minutes it takes to complete the 24 hour simulation, i.e. the power used to solve the problem. The number I_r/CM indicates the resource efficiency, the larger the better. The domain DMI MyO V2* refers to a test made on a standalone intel Xeon which seems more efficient than the usual AMDs available on the Cray systems.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Princiles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[3] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, March 1991.

[4] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack problems.* Springer, 2004.

[5] David Blair Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors - A Hands-on Approach.* Morgan Kaufmann, 2010.

[6] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3):12:1–12:41, May 2008.

[7] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic.* Birkhäuser Boston, 2010. ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-0-8176-4704-9.

[8] David A. Patterson and John L. Hennessy. *Computer Organization and Design - The Hardware / Software Interface (Revised 4th Edition).* The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press, 2012.

[9] Rogue Wave software. Debugging memory problems with memoryscape, version 3.2.1. Technical report, 2011.

[10] Nathan Whitehead and Alex Fit-Florea. Precision and performance: Floating point and ieee 754 compliance for nvidia gpus. Technical report, 2011.

Previous reports
Previous reports from the Danish Meteorological Institute can be found on:
http://www.dmi.dk/dmi/dmi-publikationer.htm