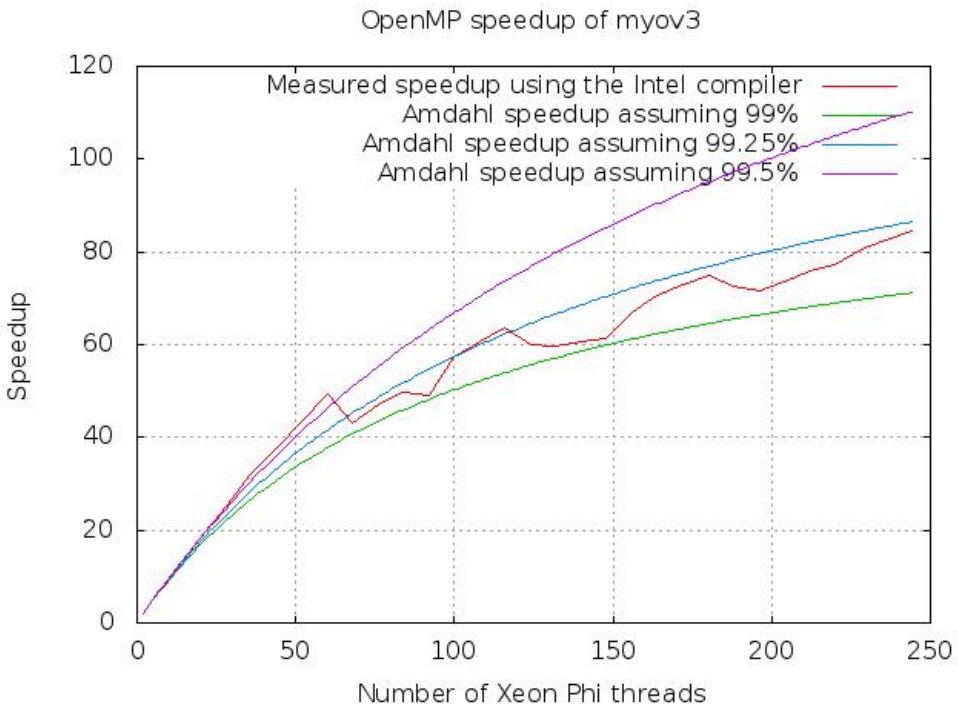
Danish Meteorological Institute
Ministry for Climate and Energy

Technical Report 12-20

Thread scaling with HBM

Jacob Weismann Poulsen and Per Berg



www.dmi.dk/dmi/tr12-20.pdf **Copenhagen 2012**

Danish Meteorological Institute
Technical Report 12-20

Colophone

Serial title:
Technical Report 12-20

Title:
Thread scaling with HBM

Subtitle:

Authors:
Jacob Weismann Poulsen and Per Berg

Other Contributers:

Responsible Institution:
Danish Meteorological Institute

Language:
English

Keywords:
Xeon Phi, parallelisation, code optimization, thread scaling, openMP, load balancing, vectorization, alignment, memory tuning, cache tuning, NUMA, many-core architecture, profiling, MKL, tridiagonal solver, Fortran, code quality, ocean model

Url:
www.dmi.dk/dmi/tr12-20.pdf

ISSN:
1399-1388

ISBN:

Version:
1.0

Website:
www.dmi.dk

Copyright:
Danish Meteorological Institute

DMI
Technical Report 12-20

Thread scaling with HBM

Jacob Weismann Poulsen Per Berg

1 Introduction

This note describes findings from applying the 2.7 release of the HBM code on the testcase formally known as myov3. The sole focus will be on the thread scaling of this test case. The purpose of the note is firstly to establish a foundation for an evaluation of the potential use of many-core architectures such as the Intel Xeon Phi or the NVIDIA Kepler for the HBM model, secondly to identity initiatives towards improvements of the thread scaling and vectorization of the code. Thus, this note is aimed at readers who wish to work on those subjects. We present our findings as more or less self-explaining tables and figures. The interested reader should consult our previous reports [1] and [2] for more in-depth details of the implementation. We will, however, draw some conclusions on the lessons learned from this experiment and we will briefly present ideas for performance improvements.

We begin with a summary of important characteristics of the compute nodes and of the test case that we use throughout this note. Then we describe the results we obtained on these compute nodes in sections 4 - 7, and we present a summary of the results in section 8. In section 9 we present results of running the HBM 2.7 release and the myov3 test case *without* any modifications at all on a many-core architecture. In section 10-11, based on our findings, we discuss how the performance of the HBM code can be improved. Finally, we draw some conclusions in section 12.

Acknowledgement: We wish to express our gratitude to Michael Greenfield, Larry Meadows and Karthik Raman from Intel for investigating the performance of the code on the Xeon Phi coprocessor and for the fruitful discussion that we have had with them during these investigations. Thanks! We are grateful to our language lawyer Bill Long from Cray for excellent clarifications on the Fortran language details. Moreover, thanks are due to Peter Thejl from DMI for helping with the regression analysis using IDL. Fi-

nally, we would like to thank Maria Grazia Giuffreda from CSCS for granting us access to some of their systems.

2 Benchmark systems

Throughout this note we will present results from runs conducted on various systems. Table 1 summarizes the characteristics of the individual CPU-based nodes that we used. The systems are either local at DMI (Cray XT5 and a standalone server) or present at CSCS[1] where we have used the Cray XE6 system known as the Monte Rosa system and the Intel Xeon based system called Piz Julier. We have used as many different compilers as we had access to on the various systems. The suite of compilers include some well-known brands as well as a few less known. The names of the compilers and specific version number of each compiler on each system are shown in table 2. As can be seen from table 2, the pool of available compilers varies from system to system, so we can unfortunately not perform a full-scale cross-system, cross-compiler analysis of the code and testcase, but we are still able to report some pretty interesting findings for some popular compilers which are available on the tested systems. The specific compiler flags we used are listed in table 3 for the IEEE builds and in table 4 for the TUNE builds. Note that TUNE in this context means nothing but adding some generally reasonable optimization flags and IEEE means nothing but adding generally reasonable non-optimization flags.

3 The MyOV3 testcase

In the present paper, all tests are performed using one single testcase. This testcase constitutes the upcoming MyOcean Version 3 setup which we plan to run fully operational at DMI from 1st April 2013. The setup is modified slightly from the case presented as Variant0 in [2] in the following ways: In the Baltic Sea, the horizontal resolution is now 1 n.m.[2], there is 122 layers with a top-layer thickness of 2 meter and vertical resolution of 1 meter down to 100 meters depth, and we have also tried to improve both initial field and the bathymetry. Moreover, in Øresund in the Inner Danish Water domain, the bathymetry has been modified and the bottom friction has been increased in order to improve the prediction of storm surges in Copenhagen.

[1] http://www.cscs.ch/service_compute_resources/index.html
[2] 1 nautical mile = 1852 meters.

	XT5 DMI	Xeon DMI	XE6 CSCS	Xeon CSCS
Nickname			*Monte Rosa*	*Piz Julier*
CPU vendor	AMD	Intel	AMD	Intel
CPU Microarchitecture	K10 (10h)	Nehalem	Bulldozer (15h)	Westmere
CPU model	Istanbul 2431	Xeon X7550	Interlagos 6272	Xeon E5649
GHz	2.4 GHz	2.0 GHz	2.1 GHz	2.53 GHz
Sockets	2	4	2	2
HyperThreading/Module	no	yes	yes	yes
Cores/Modules per socket	6/	8/	/8	6/
Threads total	12	64	32	24
Memory	16 Gb	128 Gb	32 Gb	48 Gb
L3 cache (shared)	6 Mb	18 Mb	16 Mb	12 Mb
L2 cache (per core)	512 Kb	256 Kb	2/2 Mb	256 Kb
D1 cache	64 Kb	32 Kb	16 Kb	32 Kb
Technology	45 nm	45 nm	32 nm	32 nm
TDP	75 W	130 W	115 W	80 W
SIMD size	128 bits	128 bits	256 bits	128 bits

Table 1: Node specifications for the systems used throughout this note. TDP: thermal design power, the maximum amount of power the cooling system is required to dissipate. Note that for AMD, the L2 is exclusive of D1 and the L3 is non-inclusive. For Intel, the L2 is non-inclusive and the L3 is inclusive of L1 and L2. The coherency protocol used in AMD is MOESI whereas Intel uses MESIF.

	XT5 DMI	Xeon DMI	XE6 CSCS	Xeon CSCS
gfortran	4.5.3	4.6.3	4.5.3	4.3.4
intel	12.0.4.191	12.1	12.1.2.273	
pgi	12.6.0	12.9-0	12.5.0	11.10.0
cray	7.4.1.112		8.0.6	
pathscale	3.2.99			
sun	12.2	12.3		
open64	4.2.4	5.0		
openuh		3.0.26		
nag	5.3.1(907)			
lahey	8.10b			

Table 2: List of compilers and compiler versions used in this study. Note that the gfortran version (4.6.3) present on the Intel Xeon system at DMI is buggy and the code was patched to allow a work-around for the bug reported here: `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=55314`.

Compiler	IEEE flags
gfortran	-fsignaling-nans -fno-reciprocal-math -ftrapping-math -fno-associative-math -fno-unsafe-math-optimizations
intel	-O0 -traceback -fp-model precise -fp-stack-check -fpe0
pathscale	-fno-unsafe-math-optimizations -OPT:IEEE_arithmetic,IEEE_arith=1 -TENV:simd_imask=OFF
pgi	-O0 -Kieee -Ktrap=fp -Mchkstk -Mchkfpstk -Mnoflushz -Mnofpapprox -Mnofprelaxed
cray	-O1 -Ofp0 -K trap=fp
open64	-fno-unsafe-math-optimizations -TENV:simd_imask=OFF -g -Wl,-z,muldefs
openuh	-fno-unsafe-math-optimizations -TENV:simd_imask=OFF -g -Wl,-z,muldefs
sun	-O0 -ftrap=%all,no%inexact -fsimple=0 -fns=no
nag	-O0 -ieee=stop -nan
lahey	-O0 -Knofsimple -Knofp_relaxed --trap

Table 3: List of compiler flags chosen for the IEEE builds in this study.

Compiler	TUNE flags
gfortran	-O3 -funroll-loops -ffast-math -fdump-ipa-inline -finline-functions -finline-limit=5000
intel	-O2
pathscale	-O3
pgi	-fastsse -Mipa=fast,inline
cray	-O2 -Oipa5
open64	-O3 -LNO:simd_verbose=on -LNO:vintr_verbose=on
openuh	-O3 -LNO:simd_verbose=on -LNO:vintr_verbose=on
sun	-O3 -vpara
nag	-O4 -mismatch_all -ieee=full -Bstatic -time
lahey	--O3 --sse2

Table 4: List of compiler flags chosen for the TUNE builds in this study.

Another difference is that we needed to increase the update frequency for tracer advection, tracer diffusion, turbulence and thermodynamics from every third to every second main time step. Therefore care must be taken when comparing scaling and timing: The myov3 case presented here will appear to run slower than the Variant0 case in [2], and scaling is expected to be poorer due to more frequent entries into notoriously bad-scaling routines of the `tflow` module. Table 5 summarizes the setup and figure 1 shows how the four sub-domains nest to each other. The computational intensity for this setup is $I_r = 60.6$ (cf. section 3.5 in [2]).

4 Serial references

First, we have conducted serial runs on our local XT5 system to get some proper references on the results. Table 6 summarizes the result of cross-comparing the statistics on prognostic model variables found in the logfiles produced by these 6 hour simulations. Moreover, figure 2 shows the serial performance obtained using the different compilers with the two (IEEE and TUNE) classes of compiler flags and figure 3 shows the timings with TUNE flags only making it easier to see which compiler generated the fastest binary on the same system.

	NS	IDW	WS	BS
approximate resolution [n.m.]	3.0	0.5	1.0	1.0
mmx [N/S]	348	482	149	720
nmx [W/E]	194	396	156	567
kmx [layers]	50	77	24	122
gridpoints [mmx*nmx*kmx]	3375600	14697144	557856	49805280
iw2 [surface wetpoints]	18908	80884	11581	119334
iw3 [wetpoints]	479081	1583786	103441	6113599
wetpoint ratio [iw3/gridpoints]	14.2%	10.8%	18.5%	12.3%
φ [latitude]	65° 52′ 30″N	57° 35′ 45″N	55° 41′ 30″N	65° 53′ 30″N
λ [longitude]	04° 07′ 30″W	09° 20′ 25″E	06° 10′ 50″E	14° 35′ 50″E
$\Delta\varphi$	0° 3′ 00″	0° 0′ 30″	0° 1′ 00″	0° 1′ 00″
$\Delta\lambda$	0° 5′ 00″	0° 0′ 50″	0° 1′ 40″	0° 1′ 40″
dt [sec]	25	12.5	25	12.5
maxdepth [m]	696.25	78.00	53.60	398.00
min Δx [m]	3787.40	827.62	1740.97	1261.65
CFL	0.797	0.790	0.626	0.910
I_r	1.8	12.0	0.4	46.3

Table 5: The testcase termed myov3. The I_r number for the setup is 60.6.

	NS (ε/δ)	**IDW** (ε/δ)	**WS** (ε/δ)	**BS** (ε/δ)
Avg salinity	1.17e-06 / 3.36e-08	8.35e-07 / 4.78e-08	7.58e-08 / 2.26e-09	6.36e-08 / 9.55e-09
RMS for salinity	1.11e-06 / 3.19e-08	1.33e-06 / 6.74e-08	7.44e-08 / 2.22e-09	6.08e-08 / 8.67e-09
STD for salinity	1.77e-06 / 1.62e-06	1.27e-06 / 1.37e-07	2.50e-08 / 1.18e-08	8.06e-08 / 3.67e-08
Avn temp	3.87e-06 / 3.37e-07	1.23e-06 / 7.32e-08	1.55e-07 / 8.78e-09	9.98e-08 / 1.19e-08
RMS for temp	5.36e-06 / 4.29e-07	1.22e-06 / 6.87e-08	1.75e-07 / 9.71e-09	2.26e-07 / 2.10e-08
STD for temp	4.64e-06 / 9.46e-07	9.74e-07 / 1.63e-07	3.05e-07 / 8.50e-08	2.36e-07 / 3.53e-08
Min salinity	2.80e-12 / 5.47e-11	0.00e+00 / 0.00e+00	0.00e+00 / 0.00e+00	0.00e+00 / 0.00e+00
Max salinity	3.93e-10 / 1.11e-11	9.49e-06 / 2.73e-07	5.34e-10 / 1.52e-11	1.95e-07 / 1.04e-08
Min temp	1.68e-10 / 2.85e-11	2.35e-06 / 9.50e-07	3.99e-07 / 4.03e-08	3.00e-13 / 1.40e-12
Max temp	3.09e-07 / 1.20e-08	9.01e-09 / 3.51e-10	1.20e-12 / 4.55e-14	1.74e-08 / 6.59e-10
Min u	1.06e-03 / 4.01e-04	1.16e-05 / 1.44e-05	2.52e-07 / 1.59e-07	2.22e-06 / 2.96e-06
Max u	5.77e-03 / 3.20e-03	1.65e-05 / 2.46e-05	3.58e-11 / 5.22e-11	8.97e-05 / 1.85e-04
Min v	1.68e-03 / 6.54e-04	1.13e-05 / 1.13e-05	2.01e-10 / 2.33e-10	3.25e-06 / 6.33e-06
Max v	5.46e-04 / 3.25e-04	1.33e-04 / 1.98e-04	4.38e-07 / 2.22e-07	2.41e-06 / 2.98e-06
Avg z	2.83e-06 / 2.31e-05	5.31e-07 / 2.90e-06	1.36e-06 / 5.05e-06	4.83e-08 / 1.82e-07
RMS for z	5.64e-05 / 1.02e-04	9.66e-07 / 4.86e-06	1.09e-06 / 1.91e-06	7.92e-08 / 2.92e-07
STD for z	5.79e-05 / 1.07e-04	1.61e-06 / 2.07e-05	1.94e-06 / 3.85e-06	1.72e-07 / 3.08e-06
Min z	9.04e-05 / 3.08e-05	4.27e-06 / 2.51e-04	1.03e-05 / 9.88e-06	8.69e-07 / 6.84e-06
Max z	4.96e-04 / 2.04e-04	1.15e-05 / 2.60e-05	1.07e-10 / 5.09e-11	1.05e-07 / 1.36e-07

Table 6: Worst case differences on statistics on prognostic model variables between the 18 serial runs (the 9 compilers on the Cray XT5 system, 2 classes of compiler flags) of the myov3 setup. Simulation length is 6 hours.

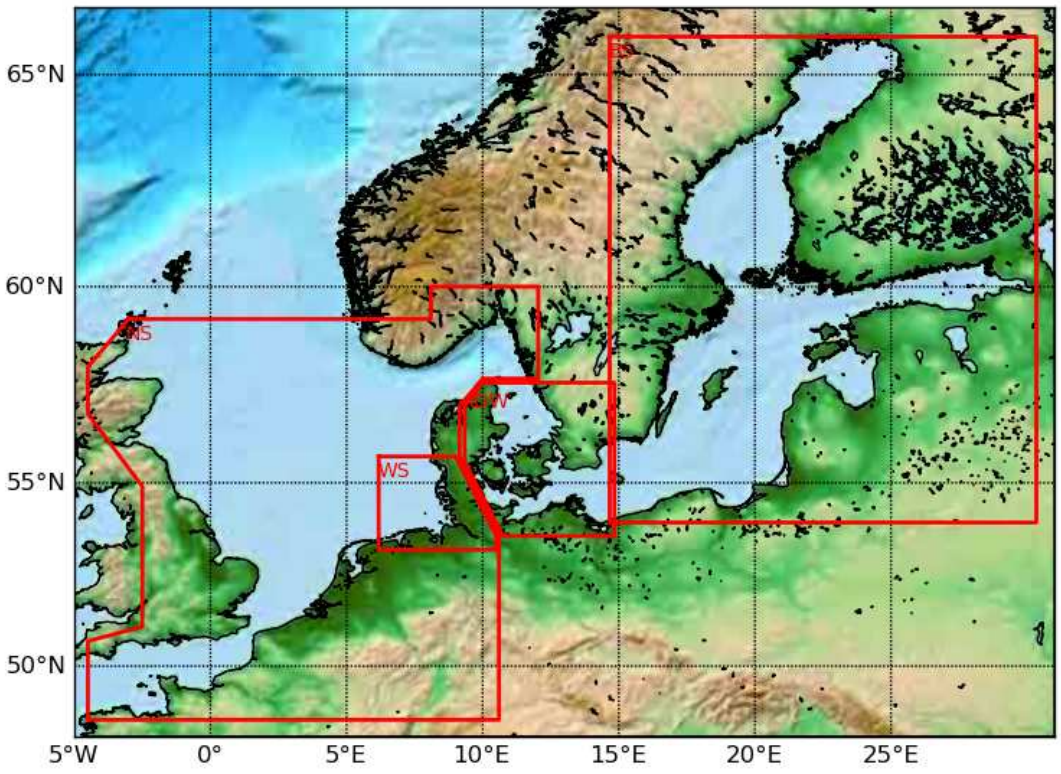


Figure 1: Nesting of the four domains in the MyO V3.0 case.

5 Memory footprint

It is important that the testcase will fit into the relatively (compared to CPUs) small memory size present on accelerators today. That is, we should ensure that our testcase will fit into say 6 Gb of memory. Table 7 shows the memory footprint of the myov3 testcase using various compilers and various configure options. Hence, we conclude that from a memory capacity point of view this testcase is indeed suitable for accelerator experiments.

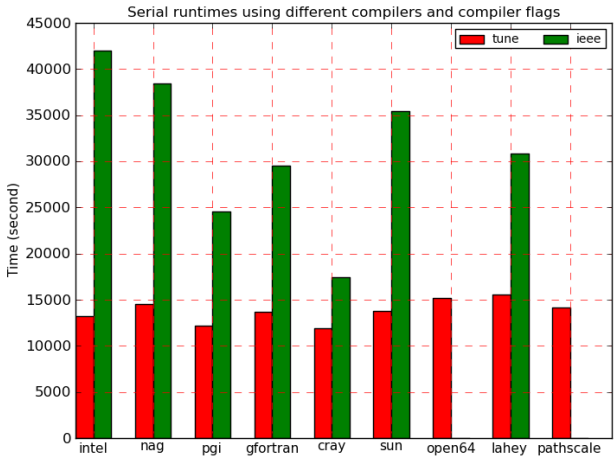


Figure 2: This plot shows the serial timings from IEEE and TUNE runs for the various compilers present at our local XT5 system. Note that there are some IEEE runs that did not make it within the walltime specified. Simulation length is 6 hours.

6 openMP

Table 8 shows the deviations in statistics of prognostic model variables that we see when we cross compare all logfiles generated across compilers, compiler flags and the four systems used in this note. Note that the differences across all runs does not differ much from the differences that we saw across the 18 serial runs on the XT5 system.

Figure 4 shows the openMP scaling on the Cray XE6 at CSCS when the code was build with configure option `--enable-openmp` and configure option `--enable-openmp --enable-mpi`, respectively. The differences in runtimes between binaries built with these two sets of configure options are shown in figure 5, i.e. deviations are mostly relatively small though there are some peaks. Taking into account that configure option `--enable-mpi` is memory neutral (see table 7) as well as the fact that our previous studies on our local XT5 shows that binaries build with this option runs faster due to the more NUMA friendly initializations (not shown here) tells us that nothing is lost by building with our default configuration `--enable-openmp`

	Tune	openMP_Tune	openMP_MPI_Tune
gfortran	3700 Mb	4199 Mb	4199 Mb
pgi	3702 Mb	4191 Mb	4191 Mb
intel	3871 Mb	4388 Mb	4388 Mb
cray	4698 Mb	5199 Mb	5199 Mb

Table 7: Memory footprint on XE6 for various compilers and various configure options. Note that for these compilers the memory requirement increases when openMP is included but it does not increase further when MPI is included in the build.

	NS (ε/δ)	IDW (ε/δ)	WS (ε/δ)	BS (ε/δ)
Avg salinity	1.34e-06 / 3.85e-08	1.19e-06 / 6.78e-08	1.76e-07 / 5.26e-09	8.15e-08 / 1.22e-08
RMS for salinity	1.28e-06 / 3.68e-08	1.67e-06 / 8.46e-08	1.74e-07 / 5.18e-09	8.99e-08 / 1.28e-08
STD for salinity	1.78e-06 / 1.63e-06	1.43e-06 / 1.55e-07	3.38e-08 / 1.60e-08	1.37e-07 / 6.24e-08
Avg temp	3.87e-06 / 3.37e-07	1.82e-06 / 1.09e-07	3.73e-07 / 2.11e-08	1.48e-07 / 1.76e-08
RMS for temp	5.36e-06 / 4.29e-07	1.65e-06 / 9.31e-08	3.52e-07 / 1.95e-08	2.71e-07 / 2.52e-08
STD for temp	4.64e-06 / 9.46e-07	1.28e-06 / 2.15e-07	3.05e-07 / 8.50e-08	2.66e-07 / 3.98e-08
Min salinity	3.20e-12 / 6.25e-11	0.00e+00 / 0.00e+00	0.00e+00 / 0.00e+00	0.00e+00 / 0.00e+00
Max salinity	3.93e-10 / 1.11e-11	9.49e-06 / 2.73e-07	5.37e-10 / 1.53e-11	1.95e-07 / 1.04e-08
Min temp	2.24e-10 / 3.80e-11	4.75e-06 / 1.92e-06	3.99e-07 / 4.03e-08	4.00e-13 / 1.86e-12
Max temp	5.25e-07 / 2.05e-08	1.36e-08 / 5.30e-10	3.10e-12 / 1.18e-13	1.74e-08 / 6.59e-10
Min u	1.14e-03 / 4.30e-04	1.47e-05 / 1.82e-05	2.91e-07 / 1.84e-07	3.51e-06 / 4.69e-06
Max u	7.65e-03 / 4.25e-03	1.65e-05 / 2.46e-05	4.62e-11 / 6.73e-11	9.31e-05 / 1.92e-04
Min v	1.78e-03 / 6.96e-04	1.47e-05 / 1.46e-05	1.30e-09 / 1.50e-09	3.37e-06 / 6.57e-06
Max v	6.69e-04 / 3.98e-04	1.52e-04 / 2.25e-04	4.43e-07 / 2.25e-07	3.51e-06 / 4.34e-06
Avg z	8.95e-06 / 7.31e-05	5.71e-07 / 3.12e-06	1.36e-06 / 5.05e-06	6.08e-08 / 2.29e-07
RMS for z	7.16e-05 / 1.29e-04	9.66e-07 / 4.86e-06	1.09e-06 / 1.91e-06	8.80e-08 / 3.25e-07
STD for z	7.20e-05 / 1.33e-04	1.61e-06 / 2.07e-05	1.94e-06 / 3.85e-06	1.72e-07 / 3.08e-06
Min z	9.14e-05 / 3.11e-05	1.04e-05 / 6.14e-04	1.03e-05 / 9.88e-06	9.63e-07 / 7.58e-06
Max z	9.99e-04 / 4.11e-04	1.67e-05 / 3.76e-05	4.76e-10 / 2.26e-10	1.72e-07 / 2.22e-07

Table 8: Worst case differences on statistics on prognostic model variables between all the runs in this note. That is, between 10 compilers, serial build (TUNE+IEEE), openMP build (TUNE), openMP+MPI build (TUNE) of the myov3 setup on the 4 systems. imulation length is 6 hours.

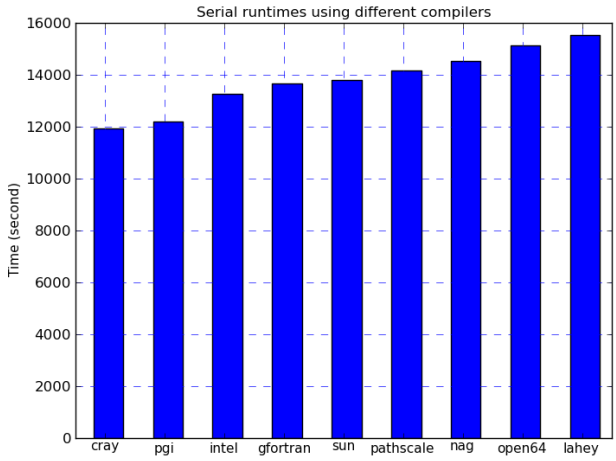


Figure 3: This plot shows the serial timings from the TUNE runs for the various compilers present at our local XT5 system. Simulation length is 6 hours.

`--enable-mpi`. On the XE6 system we obtained a consistent Amdahl scaling with threads of approximately 97% across compilers as shown in figure 4 and 6.

On the 24 cores Intel Xeon system at CSCS we obtained consistent scaling using the two compilers present, cf. figure 7. For the pgi compiler this corresponds to an Amdahl scaling with threads of approximately 98% as shown in figure 8.

We also ran the application on our local Intel Xeon system using the six compilers listed in table 2. The performance obtained using 64 hyperthreads is listed in table 9, from where the major conclusion is that not all compilers can generate code that exploit the HyperThreading feature. Taking the fastest of the compilers from this table (intel), we demonstrate in figure 9 that with our current decomposition heuristics we are indeed able to partition the workload sufficiently well so that we can feed every 64 threads with a well-balanced computational workload.

Compiler	Timing [sec]
intel	614.5
pgi	697.2
gfortran	709.2
open64	1102.8
openuh	1281.2
sun	1474.9

Table 9: The obtained timings for doing a 6 hour simulation on the local 4 socket Intel Xeon system using all 64 hyperthreads. Note that the timings obtained on this system are the fastest that we have obtained in this study but the standalone server is also the one with most threads.

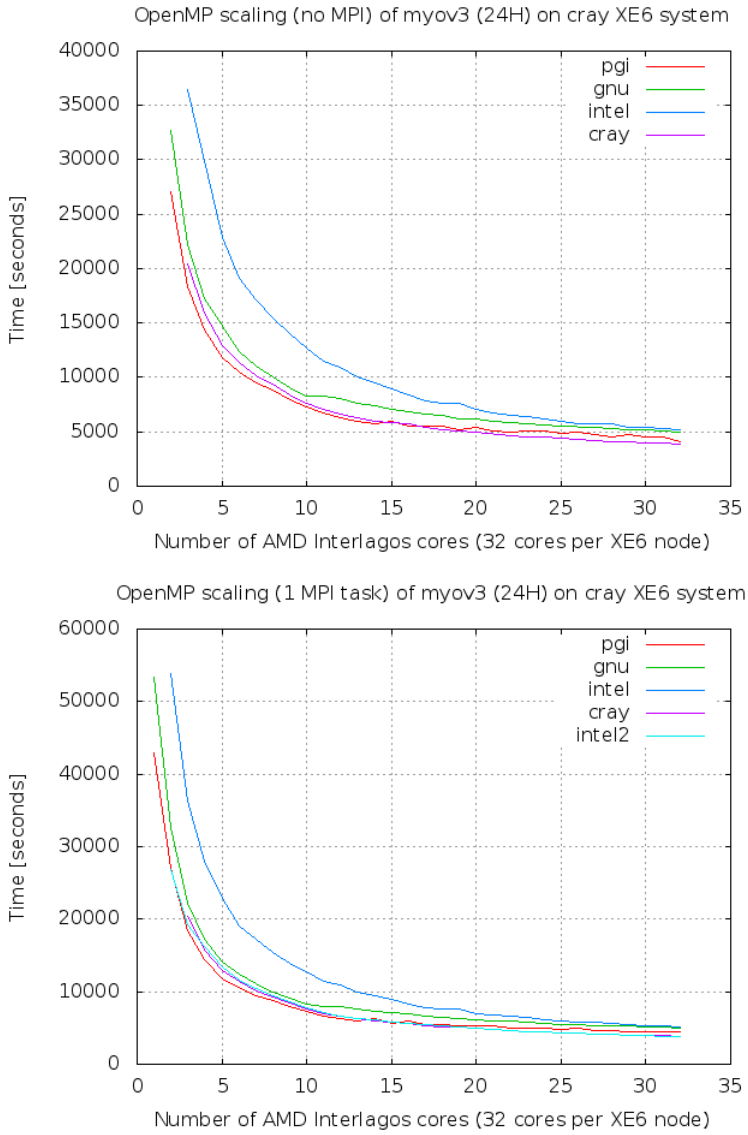


Figure 4: openMP scaling of myov3 using different compilers at the XE6 system. Simulation length is 24 hours Upper: The code was build with configure option `--enable-openmp`. Lower: The code was build with configure option `--enable-openmp --enable-mpi` and 1 MPI task was used when running it. We have used `-cc cpu -ss` for all but intel2 where we have used `-cc numa_node -ss`.

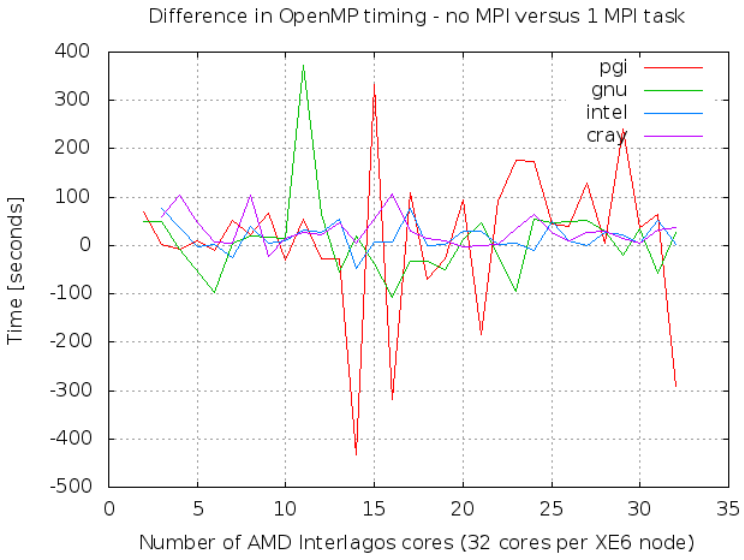


Figure 5: Differences in runtime of myov3 when building with configure option `--enable-openmp` versus when build with configure option `--enable-openmp --enable-mpi` but using only a single MPI task when running it. This is the differences between the upper and lower parts of figure 4 so negative values implies means that the MPI build is faster than the pure openMP build. Note that the overhead introduced when having MPI support build into the binary is sometimes more than compensated for by the more complete NUMA awareness in the MPI build but not always and it seems to be highly compiler dependent. Simulation length is 6 hours.

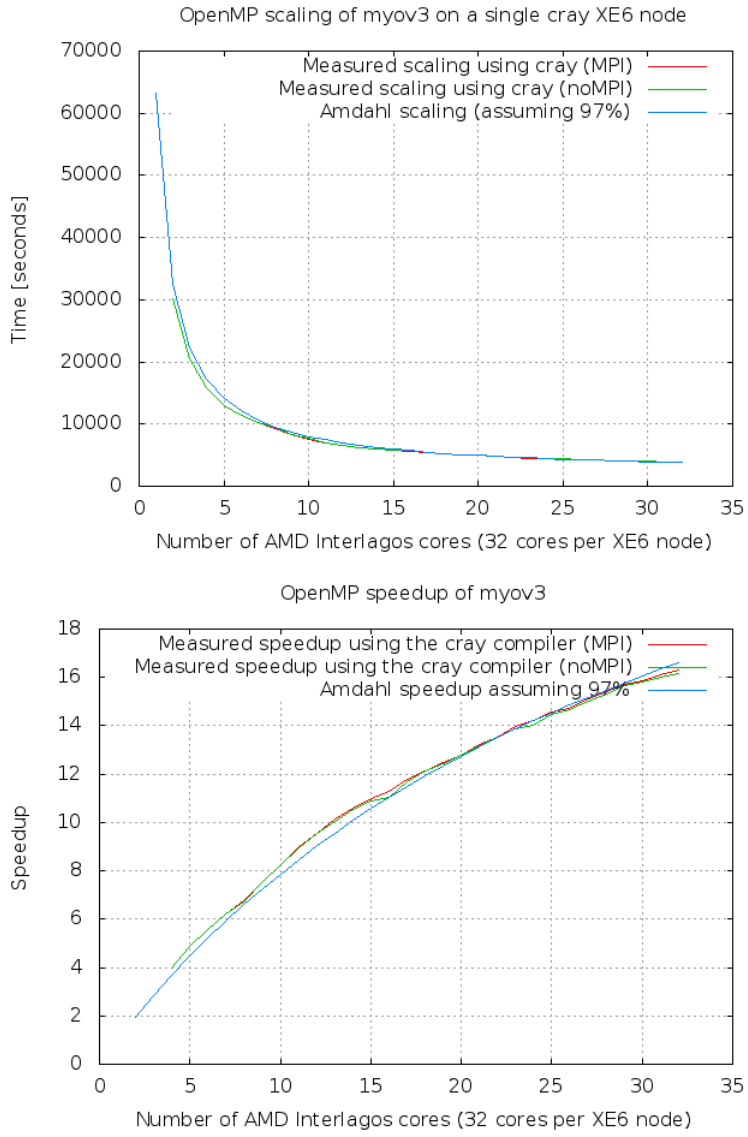


Figure 6: Upper: openMP scaling of myov3 using the cray compiler on the cray XE6 system. Note that the scaling follows Amdahl scaling of 97%. Lower: openMP speedup of myov3 using the Cray compiler on the Cray XE6 system. The code was build with configure option `--enable-openmp` and `--enable-openmp --enable-mpi`, respectively, and the simulation length was 24 hours.

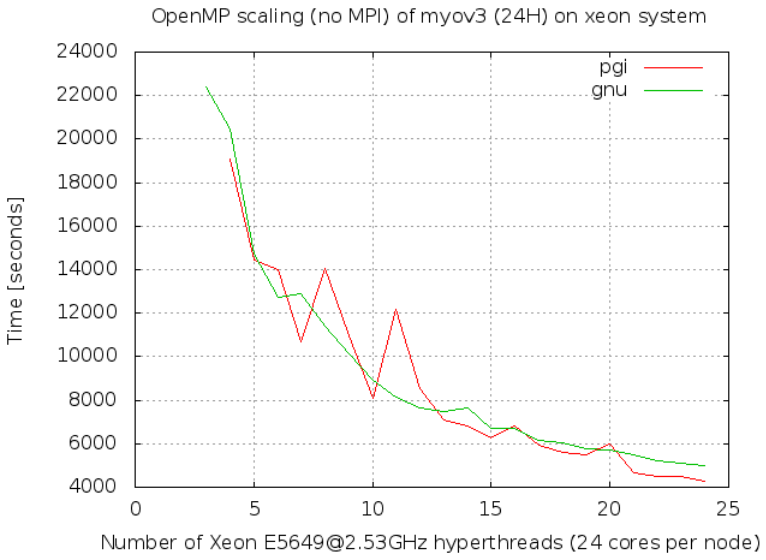


Figure 7: OpenMP scaling of myov3 on the Intel Xeon system at CSCS. Note that the runs for each compiler above gave rise to identical md5sums for the binary output files (`restart` and `tempdat`). Simulation length is 24 hours.

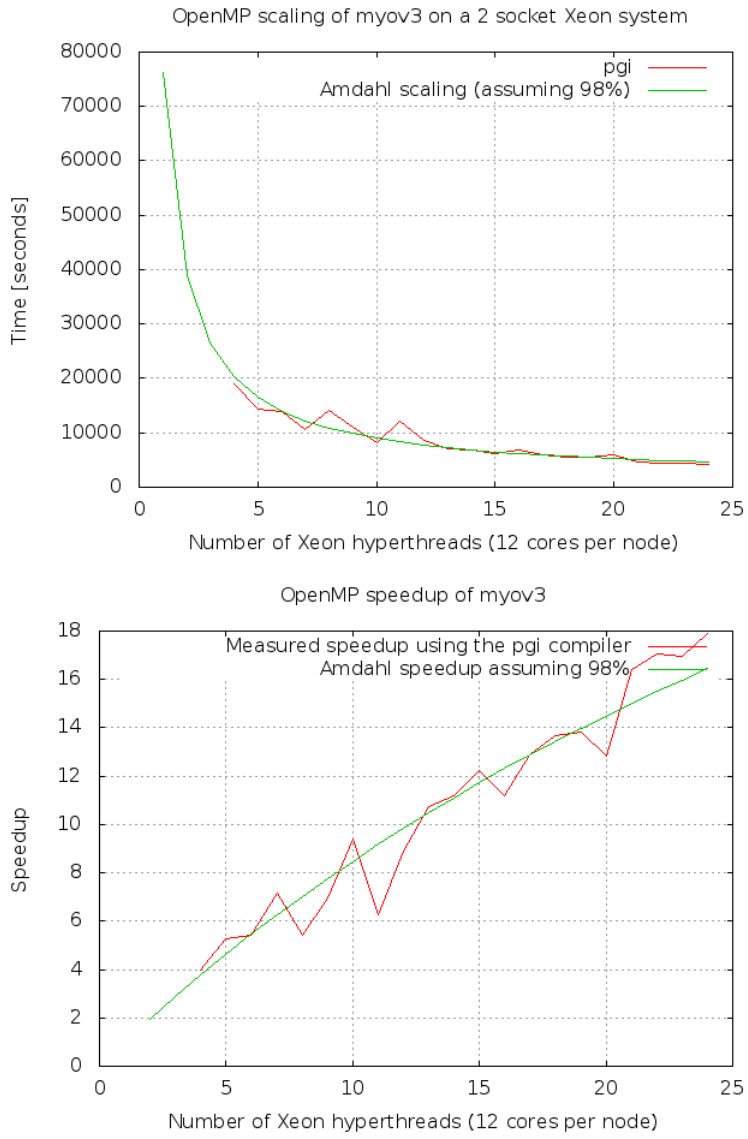


Figure 8: Upper: openMP scaling of myov3 using the PGI compiler on the Intel Xeon system at CSCS. Lower: openMP speedup of myov3 using the PGI compiler on the Intel Xeon system at CSCS. Note that the scaling follows Amdahl scaling of 98%. The code was build with configure option `--enable-openmp`. Simulation length is 24 hours.

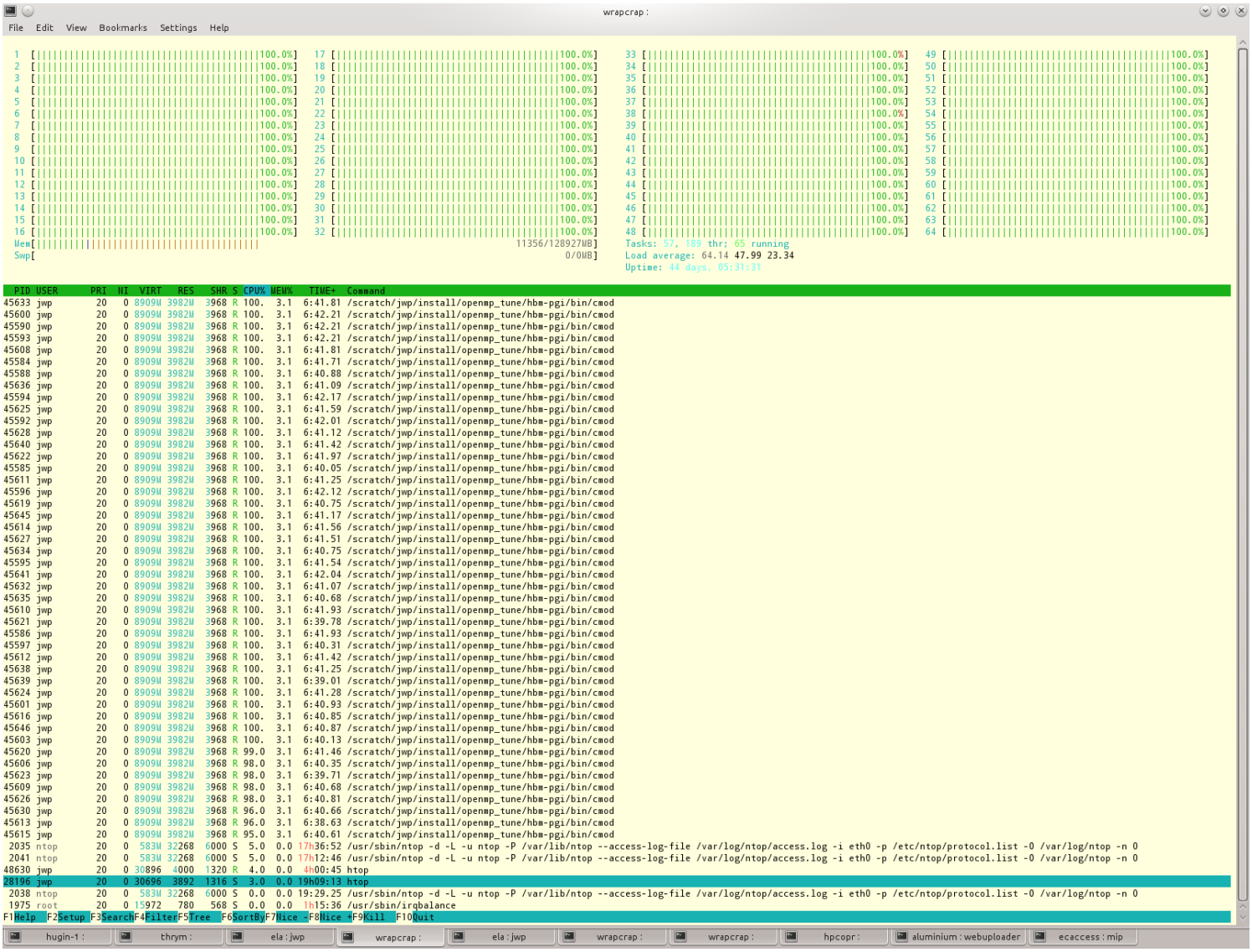


Figure 9: Running the application (build with Intel compiler) with 64 hyperthreads on the local Intel Xeon system. The figure shows that we can keep all 64 threads busy while running this case, e.g. all the green numbers in the upper part of this screen dump (which indicate the actual load) are at 100%.

7 Profiling

In this section we will look at some profiles from running with 32 threads and we will try to explain why we do see some balance issues when running with 32 threads on the AMD Interlagos. Note that appendix C provides detailed information on the decomposition of each nested subdomain in the myov3 setup. Before we head dive into the profiles we better summarize how we thread decompose the workload today: The current decomposition heuristic used for openMP is to give a well-balanced distribution of the number of 3D wetpoints handled by each thread and this strategy has been sufficient in studies done in the past. We do decomposition by running through the surface wetpoints of each nested area one by one in north-south direction, summing up the column lengths, possibly continuing to the next constant longitude grid line until we have reached the the number for an even-split, and then start to decompose for the next thread. This tends to give long, slim thread domains, but it should be remembered that it is meant to work well in our default configuration including openMP threads on MPI tasks in which we apply semi-optimal I-slices for the MPI task decomposition, cf. chapter 4 in [1]. Fortunately, both decompositon heutristics into openMP threads and into MPI tasks works well individually and together, though it is - as always - possible to find room for improvements.

We have seen in real cases that balancing the distribution of the number of 3D wetpoints between threads means a lot to the performance. Below we list some of the issues that in theory matters too but that we do not take into account with the current heuristic:

- The number of 2D wetpoints handled by each thread.
- The size of the invisible halo among the threads, i.e. the water column dependencies and the surface point dependencies.
- The extra work related to points that are located at nesting borders or at open boundaries.
- Irregular dynamics on top of the overall geometry is not taken into account, e.g. sea ice is typically not present in all wetpoints but only in geographically isolated sub-areas.
- The applied decomposition heuristic 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.

- The number of land points adjacent to the wetpoints of the thread is not taken into account.

And please note that some of the issues above may differ in severeness from one nested subdomain to the next.

On NUMA systems it is also important that variables are proper NUMA initialized. The configure option `--enable-openmp --enable-mpi` will ensure that all HEAP variables are properly NUMA initialized ([1], section 4.3.1) except for the temporary work arrays defined and used in the `tflow` module, but needless to mention adding MPI support does add some overhead too. The configure option `--enable-openmp` will ensure that the variables that are permuted due to cache layout ([1], section 2.6) are proper NUMA initialized but the rest are not. Please consult page 43-52 in [1] for the details on the current decomposition heuristic.

Table 10 and table 11 show the statistical properties of the openMP decomposition using 32 threads and this is summarized in balance scores in table 12.

	iw3	**Ideal Mean**	**Attained Mean**	**Min**	**Max**
NS	479081	14971	14971	14494 (32)	15016 (29)
IDW	1583786	49525	49525	49022 (32)	49582 (31)
WS	103441	3232	3232	3027 (32)	3251 (04)
BS	6113599	191022	191022	190019 (32)	191113 (22)

Table 10: Statistics for the openMP decompositions based on attempted even-split of 3D wetpoints (iw3) for testcase myov3 using 32 threads.

In [1] we define a balance score C^* and in table 12 we show the balance attained in each subdomain as well as a weighted balance score for the whole setup for different aspects that may influence the work balance. Note that while the 3D balance indeed looks very good the other quantities do not and our profiles will show whether or not we are bound by the 3D loops only.

	iw2	Ideal Mean	Attained Mean	Min	Max
NS	18908	590	589	458 (31)	982 (30)
IDW	80884	2527	2526	1520 (32)	4336 (02)
WS	11581	361	360	195 (01)	2058 (32)
BS	119334	3725	3724	2324 (17)	7679 (31)

Table 11: Statistics for the resulting 2D decompositions based on attempted even-split of 3D wetpoints (iw3) for testcase myov3 using 32 threads. Note that the span between minimum and maximum is significantly larger than in the statistics in table 10.

	iw3	iw2	halo3d	halo2d
$C^*(NS)$	1.0360	2.144	8.62	10.77
$C^*(IDW)$	1.0114	2.853	8.20	7.30
$C^*(WS)$	1.0740	10.554	12.96	2.23
$C^*(BS)$	1.0058	3.304	65.90	29.54
$\mathbf{C^*}$	**1.0066**	**3.2227**	**52.315**	**24.350**

Table 12: Balance score per area and weighted score for the whole setup when using 32 threads. Note that the iw3 balance is very good but if any of the other ones matter then it will become apparent from the profiles.

```
Time%  |       Time  |     Imb.  |  Imb.  |   Calls  |Group
|             |     Time  | Time%  |          | Function
|             |           |        |          |  Thread=HIDE

100.0% | 3767.695307 |        -- |     -- | 107341.0 |Total
|-------------------------------------------------------------------------------------------------------
|  97.3% | 3667.158358 |        -- |     -- |  54102.0 |USER
||------------------------------------------------------------------------------------------------------
||  24.6% |  927.745279 | 41.856098 |   6.0% |   5184.0 |cmod_hydrodynamics_solvemomeq_.REGION@li.226
||  20.8% |  784.850460 | 52.688824 |   9.0% |   5184.0 |cmod_hydrodynamics_solvemomeq_.REGION@li.180
||  15.6% |  586.732307 |  0.009920 |   0.0% |   1728.0 |tflow_tflow_int_.REGION@li.2628
||   6.5% |  243.956971 |  1.308561 |   0.7% |   1728.0 |tflow_tflow_int_.REGION@li.2668
||   4.5% |  168.541357 |  0.283303 |   0.2% |   1728.0 |tflow_tflow_int_.REGION@li.2682
||   3.7% |  140.791619 |  0.518234 |   0.5% |   1728.0 |tflow_tflow_int_.REGION@li.2799
||   3.5% |  131.634795 |  4.961334 |   5.0% |   5184.0 |cmod_hydrodynamics_solvemasseq_.REGION@li.486
||   2.8% |  105.742824 |  0.490209 |   0.6% |   5184.0 |cmod_hydrodynamics_solvemomeq_.REGION@li.139
||   2.5% |   95.703736 |  0.961210 |   1.3% |   1728.0 |tflow_tflow_int_.REGION@li.2713
||   2.2% |   84.581404 |  0.000000 |   0.0% |      1.0 |main
||   2.0% |   75.494925 |  3.348184 |   5.9% |   5184.0 |cmod_hydrodynamics_solvemasseq_z_.REGION@li.546
||   2.0% |   74.831109 |  0.339240 |   0.6% |   1728.0 |tflow_tflow_int_.REGION@li.2651
||   1.8% |   68.932854 |  1.028196 |   2.0% |   1728.0 |tflow_tflow_int_.REGION@li.2695
||   1.8% |   66.923226 |  1.308783 |   2.6% |    432.0 |MAIN_.REGION@li.1671
||   1.3% |   49.653645 |  3.761034 |  10.1% |    864.0 |MAIN_
||======================================================================================================
|   2.7% |  100.536864 |        -- |     -- |  53236.0 |OMP
||------------------------------------------------------------------------------------------------------
||   1.1% |   42.602801 |  0.000000 |   0.0% |   5184.0 |cmod_hydrodynamics_solvemomeq_.REGION@li.180(ovhd)
||   1.0% |   38.771718 |  0.000000 |   0.0% |   5184.0 |cmod_hydrodynamics_solvemomeq_.REGION@li.226(ovhd)
||======================================================================================================
|   0.0% |    0.000085 |  0.000000 |   0.0% |      3.0 |PTHREAD
|======================================================================================================
```

Figure 10: openMP profile of a 6 hour simulation of myov3 using 4 openMP threads on Cray XE6. The compiler is PGI and the flags are Tune flags.

As mentioned before we are using temporary work arrays in the `tflow` module and none of these are properly NUMA initialized. There are 9 of these temporary work arrays each of size `max(iw3(:))` times the number of components. This is for sure a serious candidate for explaining the balance problems we see in the `tflow` module.

Looking more carefully into the profile whose summaries are found in figures 10 - 12 it seems that the im-balance issue in the `tflow` module splits the threads into three categories: The most expensive, the mid-expensive and the less expensive. This deeper analysis is not shown here but with 32 threads it turns out that if each thread has a number in [1:32], then the most expensive are: $8, 7, 5, 6, 18, 19, 15, 10,$ the mid-expensive are: $27, 22, 29, 13, 20, 32, 26, 24,$ and the rest of the threads fall into the less expensive class. The cost ratio between these three classes is $6 : 5 : 3$. The same applies when building with configure option `--enable-openmp --enable-mpi` but the most expensive are now: $5, 11, 14, 8, 4, 10, 19, 9,$ the mid-expensive are: $29, 22, 30, 20, 27, 25, 31, 17$. Note that thread 6 is in the most expensive class for the configure option `--enable-openmp` and in the least expensive class for the configure option `--enable-openmp --enable-mpi`.

```
Time%  |       Time  |     Imb.  |  Imb.  |   Calls  |Group
|             |     Time  | Time%  |          | Function
|             |           |        |          |  Thread=HIDE
100.0% | 1331.867084 |        -- |     -- | 107369.0 |Total
|-------------------------------------------------------------------------------------------------------
|  83.7% | 1114.233231 |        -- |     -- |  54102.0 |USER
||------------------------------------------------------------------------------------------------------
||  12.0% |  160.021646 |  0.013380 |   0.0% |   1728.0 |tflow_tflow_int_.REGION@li.2628
||  11.8% |  157.350077 | 14.912513 |   9.8% |   5184.0 |cmod_hydrodynamics_solvemomeq_.REGION@li.180
||  11.8% |  156.511866 | 12.410528 |   8.2% |   5184.0 |cmod_hydrodynamics_solvemomeq_.REGION@li.226
||   8.4% |  111.499725 | 27.934333 |  25.9% |   1728.0 |tflow_tflow_int_.REGION@li.2668
||   7.1% |   94.116555 |  0.000000 |   0.0% |      1.0 |main
||   5.4% |   72.519601 | 16.623970 |  23.7% |   1728.0 |tflow_tflow_int_.REGION@li.2682
||   4.8% |   63.515905 | 14.377835 |  23.4% |   1728.0 |tflow_tflow_int_.REGION@li.2695
||   4.1% |   54.786552 |  9.764342 |  18.4% |   5184.0 |cmod_hydrodynamics_solvemasseq_.REGION@li.486
||   4.0% |   53.540016 | 13.009681 |  25.1% |   1728.0 |tflow_tflow_int_.REGION@li.2713
||   3.4% |   45.732918 | 10.138852 |  22.9% |   5184.0 |cmod_hydrodynamics_solvemasseq_z_.REGION@li.546
||   3.1% |   40.837477 |  1.570945 |   4.0% |   1728.0 |tflow_tflow_int_.REGION@li.2651
||   2.8% |   37.846599 |  5.317232 |  14.5% |   1728.0 |tflow_tflow_int_.REGION@li.2799
||   2.1% |   27.332082 |  2.627632 |   9.9% |   5184.0 |cmod_hydrodynamics_solvemomeq_.REGION@li.139
||======================================================================================================
|  16.3% |  217.631541 |        -- |     -- |  53236.0 |OMP
||------------------------------------------------------------------------------------------------------
||   3.9% |   52.144284 |  0.000000 |   0.0% |   1728.0 |tflow_tflow_int_.REGION@li.2668(ovhd)
||   2.3% |   31.235419 |  0.000000 |   0.0% |   1728.0 |tflow_tflow_int_.REGION@li.2682(ovhd)
||   2.3% |   30.109563 |  0.000000 |   0.0% |   1728.0 |tflow_tflow_int_.REGION@li.2695(ovhd)
||   1.8% |   23.924006 |  0.000000 |   0.0% |   1728.0 |tflow_tflow_int_.REGION@li.2713(ovhd)
||   1.3% |   17.287265 |  0.000000 |   0.0% |   5184.0 |cmod_hydrodynamics_solvemasseq_z_.REGION@li.546(ovhd)
||   1.2% |   16.499356 |  0.000000 |   0.0% |   5184.0 |cmod_hydrodynamics_solvemomeq_.REGION@li.180(ovhd)
||   1.2% |   15.662973 |  0.000000 |   0.0% |   5184.0 |cmod_hydrodynamics_solvemomeq_.REGION@li.226(ovhd)
||======================================================================================================
|   0.0% |    0.002313 |  0.000000 |   0.0% |     31.0 |PTHREAD
|======================================================================================================
```

Figure 11: openMP profile of a 6 hour simulation of myov3 using 32 openMP threads on Cray XE6. The compiler is PGI and the flags are Tune flags.

```
Time%  |       Time  |     Imb.  |  Imb.  |   Calls  |Group
|             |     Time  | Time%  |          | Function
|             |           |        |          |  Thread=HIDE
100.0% | 93.175635 |       -- |     -- | 6007.0 |Total
|-------------------------------------------------------------------------------------------------------
|  86.8% | 80.877350 |       -- |     -- | 2987.0 |USER
||------------------------------------------------------------------------------------------------------
||  31.0% | 28.869857 | 0.000000 |   0.0% |    1.0 |cmod_main_
||   8.8% |  8.164287 | 0.205307 |   2.6% |  288.0 |solvemomeq$cmod_hydrodynamics_.REGION@li.232
||   6.8% |  6.317070 | 1.603754 |  26.2% |   96.0 |tflow_int$tflow_.REGION@li.2671
||   6.5% |  6.020506 | 0.001036 |   0.0% |   96.0 |tflow_int$tflow_.REGION@li.2632
||   5.9% |  5.452354 | 0.062519 |   1.2% |  288.0 |solvemomeq$cmod_hydrodynamics_.REGION@li.182
||   4.5% |  4.179901 | 0.932946 |  23.0% |   96.0 |tflow_int$tflow_.REGION@li.2685
||   3.8% |  3.578811 | 0.826012 |  23.8% |   96.0 |tflow_int$tflow_.REGION@li.2698
||   3.4% |  3.189101 | 0.587932 |  19.0% |  288.0 |solvemasseq$cmod_hydrodynamics_.REGION@li.488
||   3.3% |  3.065106 | 0.748358 |  25.2% |   96.0 |tflow_int$tflow_.REGION@li.2720
||   2.7% |  2.553021 | 0.582473 |  23.6% |  288.0 |solvemasseq_z$cmod_hydrodynamics_.REGION@li.557
||   2.5% |  2.371613 | 0.090853 |   4.0% |   96.0 |tflow_int$tflow_.REGION@li.2654
||   2.2% |  2.085787 | 0.298207 |  14.8% |   96.0 |tflow_int$tflow_.REGION@li.2803
||   1.8% |  1.651546 | 0.195964 |  12.2% |  288.0 |solvemomeq$cmod_hydrodynamics_.REGION@li.147
||======================================================================================================
|  13.2% | 12.296385 |       -- |     -- | 2985.0 |OMP
||------------------------------------------------------------------------------------------------------
||   3.3% |  3.032194 | 0.000000 |   0.0% |   96.0 |tflow_int$tflow_.REGION@li.2668(ovhd)
||   1.9% |  1.751958 | 0.000000 |   0.0% |   96.0 |tflow_int$tflow_.REGION@li.2695(ovhd)
||   1.9% |  1.732608 | 0.000000 |   0.0% |   96.0 |tflow_int$tflow_.REGION@li.2682(ovhd)
||   1.5% |  1.390799 | 0.000000 |   0.0% |   96.0 |tflow_int$tflow_.REGION@li.2713(ovhd)
||   1.1% |  0.986007 | 0.000000 |   0.0% |  288.0 |solvemasseq_z$cmod_hydrodynamics_.REGION@li.546(ovhd)
||======================================================================================================
|   0.0% |  0.001892 | 0.000000 |   0.0% |   31.0 |PTHREAD
|   0.0% |  0.000008 | 0.000000 |   0.0% |    4.0 |SYSCALL
|======================================================================================================
```

Figure 12: openMP profile of a 6 hour simulation of myov3 using 32 openMP threads on Cray XE6. The compiler is cray and the flags are Tune flags.

In tables 23 - 26 in appendix C we show how the work of the myov3 testcase is distributed among the threads and how much work there is in each sub-domain, and in figure 44 - 47, also in appendix C, we have tried to illustrate how the thread decomposition has resulted in 32 sub-domains for each of the four nested domains and how the cost classes are distributed when the configure option is `--enable-openmp`.

With these tables and figures we should be able to (partially) explain the observed classification *if* we can assume that the cost class of a thread is mainly determined by how much that particular thread needs data in the halo residing on other threads (since each thread has very nearly equally many 3D wetpoints). Attempting to analyse why a thread falls into a particular cost class we have in figure 13 shown an example of the halosize side-by-side the relative cost (using configure option `--enable-openmp`) of each thread, but it is not easy to see any clear correlation, e.g. it is not immediately evident why thread No. 32 is in the mid-expensive class instead of the less-expensive class, and why say thread Nos. 16 and 17 are not in the

most expensive class. Thus, other factors must play a more important role here. The fact that we see an inconsistent split into the three classes using different configure options also calls for a more detailed profiling before we can complete the analysis.

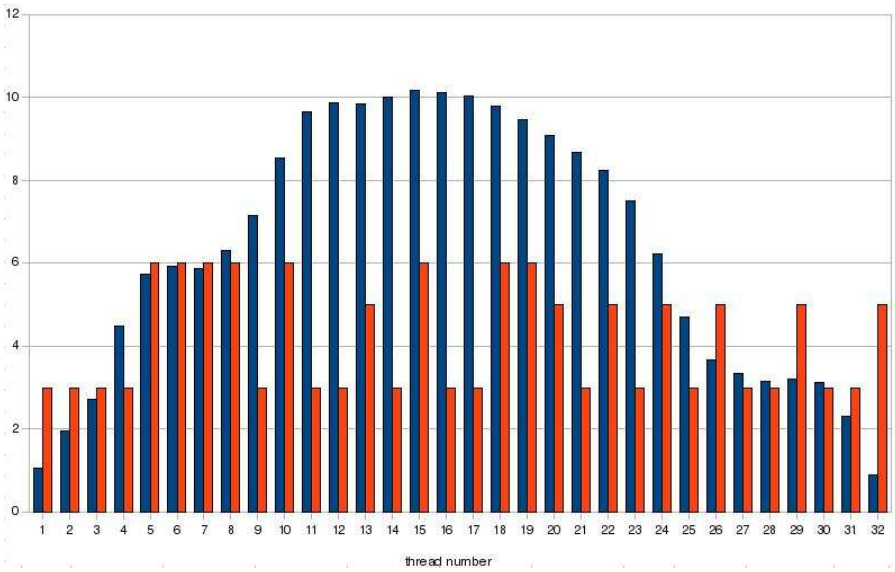


Figure 13: Example of total halosize (blue) and relative cost (red) for each of the 32 threads. Both quantities are in arbitrary units to make them fit on the same graph. The configure option for obtaining the cost is `--enable-openmp`.

It is much more intuitive to find thread No. 32 in the least expensive class as we do with configure option `--enable-openmp --enable-mpi` than in the mid-expensive class as we do with `--enable-openmp` and we have reached the conclusion that the tools used to conduct this profiling may impact the results too much to make it reliable.

Another thing that matters is the column length which in the code appears as the length of the inner-most loops[3] as well as the size of the tri-diagonal system of equations that are used so often throughout the model[4]. In fig-

[3] cf. section 2.3 in [1] for a description of the cache layout, permutation and loop structure of the HBM code.

[4] cf. chapter 3 in [2] for discussions on our use of tri-diagonal linear systems of equations in the HBM code.

```
  USER / solvemomeq$cmod_hydrodynamics_.REGION@li.232$
-----------------------------------------------------------------------------
  Time%                                                 8.8%
  Time                                              8.164287 secs
  Imb. Time                                         0.205307 secs
  Imb. Time%                                            2.6%
  Calls                          35.274 /sec             288.0 calls
  PAPI_L1_DCM                    38.686M/sec          315861606 misses
  PAPI_TLB_DM                     0.183M/sec            1495726 misses
  PAPI_L1_DCA                   897.357M/sec         7326629429 refs
  PAPI_FP_OPS                   401.716M/sec         3279882360 ops
  Average Time per Call                               0.028348 secs
  CrayPat Overhead : Time         0.0%
  User time (approx)              8.165 secs    17145873450 cycles  100.0% Time
  HW FP Ops / User time         401.716M/sec         3279882360 ops    2.4%peak(DP)
  HW FP Ops / WCT               401.716M/sec
  Computational intensity          0.19 ops/cycle       0.45 ops/ref
  MFLOPS (aggregate)            401.72M/sec
  TLB utilization              4898.38 refs/miss     9.567 avg uses
  D1 cache hit,miss ratios       95.7% hits           4.3% misses
  D1 cache utilization (misses)  23.20 refs/miss     2.899 avg hits
```

Figure 14: Zooming in on the momeqs profile from the openMP profile of a 6 hour simulation of myov3 using 32 openMP threads on Cray XE6. The compiler is cray and the flags are Tune flags.

ures 48-51 in appendix D we have shown the distribution of column lengths for the each nested domain. These figures demonstrate clearly - once again - what we already knew, namely that our problem is highly irregular. In tables 13 and 14 we present a statistical summary of those distributions. The results are given for scalar equations, kh, and for $u-$ and $v-$equations, khu and khv, respectively. Note that the most frequent length in the Baltic Sea domain is 101 with more than 3000 occurrences for all three column lengths, and that the second and third most frequent lengths are 1 and 102 for scalar equations while they are 102 and 1 for u and v. The mean and median lengths for the Baltic Sea domain is only approximately half of the most frequent lengths. In the other domains the most frequent lengths are much shorter, e.g. in the second largest domain, the inner Danish Waters, they are 10-12, and the shallow Wadden Sea domain is dominated by one-layer cases which is necessary due to extensive flooding and drying. This diversity makes a general strategy towards vectorization more challenging and we will need to think carefully what we plan to do.

	NS			IDW			WS			BS		
	kh	**khu**	**khv**	**kh**	**khu**	**khv**	**kh**	**khu**	**khv**	**kh**	**khu**	**khv**
mean	25.3	24.7	24.6	19.6	18.9	18.9	8.9	8.6	8.7	51.3	49.9	49.8
median	25	24	24	17	16	16	9	9	9	48	46	46
std	13.3	13.6	13.6	13.9	13.9	13.9	6.0	6.1	6.1	34.7	34.5	34.5

Table 13: Statistical summary of the column lengths of each nested area. The values of mean, median and standard deviation are shown for each domain for scalar equations, kh, and for $u-$ and $v-$equations, khu and khv, respectively.

Domain	**k**	**kh**	**khu**	**khv**
NS	41	1164	1159	1134
	39	949	890	915
	42	745	714	707
IDW	12	3329	3215	3233
	11	3246	3196	3185
	10	3068	3040	3044
WS	1	2386	2338	2371
	16	867	865	856
	9/15	676/	/656	/654
BS	101	3215	3083	3029
	1/102	2637/	/2452	/2468
	102/1	2510/	/2443	/2460

Table 14: The three most frequent column lengths, k, in each nested area and how often these occur. For each value of k we show the corresponding frequency for scalar equations, kh, and for $u-$ and $v-$equations, khu and khv, respectively.

8 Summary

We have tried to summarize the performance numbers attained on the different systems in table 15. Please note that we consider our local XT5 with a maximum of 12 threads too small to enter this comparison in a fair way; the interested reader may consult [1] for scaling studies on the XT5.

DMI Xeon		**CSCS XE6**		**CSCS Xeon**	
Compiler	Timing [sec]	Compiler	Timing [sec]	Compiler	Timing [sec]
intel	614.5	intel	956.3	pgi	1067.5

Table 15: This table summarizes the fastest time obtained by the various compiler and configure options on each of the three system that we used for thread scaling studies.

In the previous reports that we have concluded the intel compiler had problems in keeping up with the performance attained by the other compilers on this code but this time we investigated it further and found that the real problem was due to the pinning done by `aprun` on the AMD-based Cray systems used here (XT5 and XE6). With the intel compiler one needs to use `-cc numa_node` (or explicit pinning) and not the usual `-cc cpu` when launching with `aprun`. As revealed above, the intel compiler is actually the one that generates the fastest code on both AMD hardware and Intel hardware[5].

It is worth mentioning that with the TUNE flags chosen we see binary identical results across the different number of threads $1, \ldots, 32$ on Monte Rosa and across the different number of threads $1, \ldots, 24$ on Piz Julier for all compilers but the cray compiler. If we build with the cray compiler using `-O1 -Oipa5` instead of `-O2 -Oipa5` then we see a slight degradation in performance but then we get binary identical results across the different number of threads with this compiler too; this was also mentioned in [2].

It is interesting to note that we still scale when hitting the limit of 64 threads (as set by the hardware systems available for the present report). Having said that, we also see that the increase in the update frequency for tracer advection, tracer diffusion, turbulence and thermodynamics have strong im-

[5]Please note that the intel compiler was not available on the CSCS Xeon, cf. table 2

plications for the overall performance (compare with e.g. the one-node results from chapter 6 in [2]). It is clear that we should work on improving both the serial performance and the scaling (as revealed in the profiles we do now see balance issues at higher thread counts) of the `tflow` components. This issue has not been so obvious in the previous studies, but we can only imagine that it becomes even more significant when we add more tracers by e.g. including the bi-geo-chem-model.

Finally, we have found a page thrashing issue in `cmod_hydrodynamics.f90` in the openMP block starting at line 486 when using the pgi generated binary, i.e. we do not have a sufficient amount of data references per TLB miss in this region. This is another issue that we ought to deal with for a future release of the HBM code.

We have shown that the application does indeed scale with threads and we have shown that the testcase chosen does indeed fit into a relatively limited amount of memory found on the accelerators and the coprocessors today. From that perspective the code and the testcase is a good candidate for experiments on say some of the emerging many-core architectures.

However, we have also shown that our openMP speedup for this particular testcase on the Intel Xeon architecture is limited to a factor of maximum somewhere between 33 and 50[6]. We have shown that we can attain a speedup of 18 on the Xeon system using 12 cores and 24 hyperthreads. That is if we assume similiar speedup characteristics then in order to be competitive with Xeon each thread on alternative hardware will need to run at a speed that is between 1/2 (assuming attainable speedup on alternative hardware is 36) and 1/3 (assuming attainable speedup on alternative hardware is 54) of the Xeon speed. For the Intel Xeon Phi architecture this is not very likely given that the most expensive subroutine does not vectorize well, cf. appendix G. It should not come as a big surprise by now, but in conclusion we need to work on the HBM code in order to benefit from an architecture like the Xeon Phi architecture and probably even more so to benefit from an architecture like the NVIDIA Kepler GPU (see also our preliminary study on a GPU port in [2]).

Taking the HBM code as a "shelf product", i.e. as is, without preparing the

[6] corresponding to a parallel fraction of the code between 97% and 98% running on a large number of threads.

code, we will make a very crude projection on the performance we could hope for on a many-core architecture: Assume that the cores on the many-core architectures are running at say 1.0GHz and this is to be compared with the Xeon Westmere at 2.53GHz. Each many-core thread then operates 2.53 times slower than each Westmere thread. Moreover, assume that we do not really exploit the longer vector width on the many-core architecture (which is a reasonable assumption since we have not previously worked seriously towards vectorization) nor the higher memory bandwidth. Finally, assume that the thread speedup that we indeed can sustain is more likely 30 (rather than the 36-54 mentioned above) then we should expect that the many-core architecture at most can be $2.53 * 18/30 \approx 1.5$ times slower than the Westmere in order to be competitive. This crude estimate should be taken with a grain of salt and it only states that it would be a big surprise to us if any of the many-core architectures would outperform or even come close to the performance attained at the Xeon Westmere system with this code and this testcase.

Having said that, we strongly believe that we could make some adjustment to the HBM code in order to improve the per-thread performance and thus make it better suited for the emerging many-core architectures.

By conducting the analysis for this report we have learned that decent thread scaling is a necessary but certainly not a sufficient condition for competitive performance on many-core architectures and once again we have reached the conclusion that real performance is gained by doing code improvements that makes the generated binary exploit the hardware that it is supposed to run on.

9 Out-of-the-box run on the Intel Xeon PhiT

Michael Greenfield from Intel kindly offered to run the 2.7 release of the code on a Xeon PhiT system. The system used was a 2 socket server with Intel® Xeon® processor E5-2670 (8C, 2.6GHz. 115W) *pre-production* Intel® Xeon PhiT coprocessor (SE10X B1, 61 core, 1.1GHz, 8GB @ 5.5GT/s) running the gold software release. Thus, it is possible run with up to 240 threads[7] on the coprocessor. The out-of-the-box speedup is shown in figure 15 and note that the attained over-all (i.e. across all thread counts) speedup is close to 99.25% which came as a pleasant surprise to us. For low thread counts the speedup is even higher, close to 99.5%. Moreover, note the linear (i.e. 100%) speedup for thread counts from 60 and up to 240 which is quite unusual. This is a good start but as the analysis in the previous section showed this is not yet sufficient to compete with the CPU performance. One should keep in mind that good scaling may be due to slow performance and the absolute performance is not impressive at all.

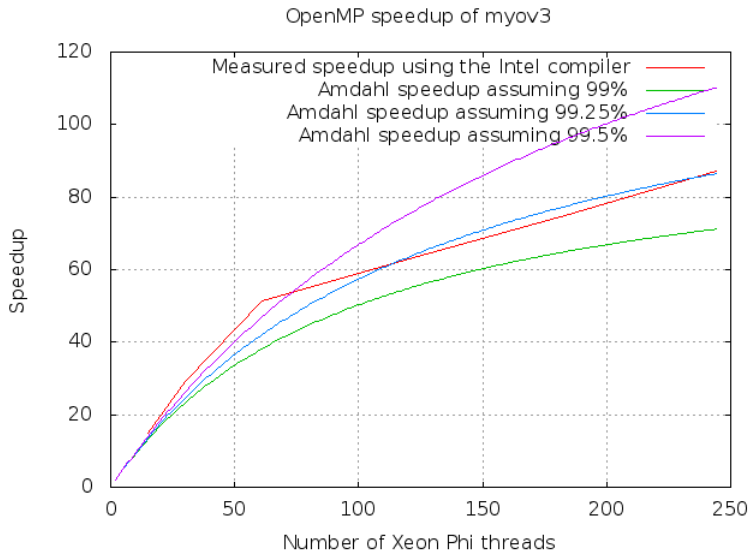


Figure 15: Attained speedup on the Xeon Phi without any changes to the code.

[7] 240 = (61-1)*4. One core is used by the Intel Xeon Phi operating system.

10 Musings on potential improvements

Table 16 and table 17 show the statistical properties of the openMP decomposition using 240 threads and this is summarized in balance scores in table 18. It is evident from those tables that our decomposition is not ideal at this high thread count; the distribution of wetpoints is not uniform across the threads and there are even empty threads in some domains. In order to really gain from the nice scaling demonstrated above we must improve the vectorization of this code on each thread and the load balancing across the threads. This section will outline some ideas on how we could improve these and other issues.

	iw3	Ideal Mean	Attained Mean	Min	Max
NS	479081	1996	2004	0 (240)	2042 (230)
IDW	1583786	6603	6603	3334 (240)	6674 (66)
WS	103441	431	435	0 (240)	448 (1)
BS	6113599	25469	25469	16691 (240)	25572 (129)

Table 16: Statistics for the openMP decompositions based on attempted even-split of 3D wetpoints (iw3) for testcase myov3 using 240 threads.

	iw2	Ideal Mean	Attained Mean	Min	Max
NS	18908	78	78	-1 (240)	164 (219)
IDW	80884	337	336	128 (240)	689 (17)
WS	11581	48	47	-1 (238)	381 (235)
BS	119334	496	495	283 (126)	1468 (229)

Table 17: Statistics for the resulting 2D decompositions based on attempted even-split of 3D wetpoints (iw3) for testcase myov3 using 240 threads. Note that the ratio between minimum and maximum is significantly larger than in the statistics in table 16.

10.1 The tridiagonal solver

As described in [2] a tridiagonal solver is used intensively in this model. The algorithm applied today is the double-sweep algorithm which is inherently serial and thus will not vectorize. On Istanbul this means a theoretical loss

	iw3	**iw2**
$C^*(IDW)$	2.099	5.58
$C^*(BS)$	1.597	5.45
$\mathbf{C^*}$	**1.70**	**5.48**

Table 18: Balance score per area and weighted score for the partial setup consisting of the two larger subdomains amounting to more than 96% of the total number of 3D wetpoints when using 240 threads.

of performance of a factor of 2. On SandyBridge and Interlagos the theoretical loss is a factor of 4. Moreover, on the Xeon Phi the theoretical loss in performance is a factor of 8 and will probably be even higher with future versions of Xeon Phi.

There do exist parallel algorithms for solving a tridiagonal systems and one could hope that some of the lapack libraries (MKL, ACML, libsci, ESSL, pdlib, scsl, ...) would have such an implementation. Having tried to investigate available libraries we must conclude that only MKL[8] and ESSL[9] seem to have a tridiagonal solver that assumes diagonal dominance and thus that have the potential to compete with the simple double-sweep algorithm that we use today.

We better do a back-of-the-envelope estimate of the call frequency of the tridiagonal solver in the code today, that is in:

- The momentum equations, `default_momeqs.f90`
- vertical tracer diffusion, `tflow.f90`
- k- and ω-equations in turbulence, `turbmodels.f90`

For the momentum equations, there is one call per u column and one per v column having more than 2 layers and one time per local timestep. Let N_u and N_v be the number of these columns and let $N_t = \Delta t/\Delta t_i$ with Δt

[8] `http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation`, page 693ff.

[9] `http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp?topic=%2Fcom.ibm.cluster.essl.doc%2Fesslbooks.html`.

being the main timestep and Δt_i being the local timestep for subdomain i[10]. Then the number of calls for a 6 hour simulation will be

$$\frac{6\mathrm{H}}{\Delta t} \sum_{i=1}^{narea} [(N_u + N_v)N_t]_i$$

We can calculate the exact numbers for N_u and N_v from figure 48-51 in appendix D but a fair estimate, i.e. something less than the total number of surface wetpoints, is sufficient here so we can use 90% as a reasonable approximation and do some calculations using the numbers from table 5[11]:

$$777.6 * (18908 * 2 * 1 + 80884 * 2 * 2 + 11581 * 2 * 1 + 119334 * 2 * 2) = 670 \text{ mill}$$

That is, 670 million times for a 6 hour simulation.

The tridiagonal solver is inlined in `tflow_mp_diffusion` for the vertical tracer diffusion (once per water column with 2 or more layers, the inner loop is the number of tracers, `nc`, long). Inlining with the `nc`-loop innermost prevents a lot of unnecessary calls. A conservative assumption is that approximately 95% of all surface point have 2 or more layers, thus the number of calls become[12]:

$$410.4 * (18908 + 80884 + 11581 + 119334) = 94.7 \text{ mill}$$

If we exline the solver then for the myov3 testcase without any extra tracers the frequency of calls will be something like 189 mill times for a 6 hour forecast and it will be 1042 mill times for a myov3 with 9 ergom tracers. To be effectively useful, a library solver should outperform our originally in-lined solver over `nc` components by so much that it compensates for the `nc` calls per water column and the increased book-keeping by not having the `nc`-loop as the innermost loop. It is doubtful if a library solver can do that in the situation with only two tracers, salinity and temperature, which have different diffusion coefficients and therefore a different system matrix. But for say the 9 ergom tracers which have one and the same diffusion coefficient and therefore can share the system matrix, it is very likely that a library

[10] Our nesting implementation requires that Δt is an integer multiple of Δt_i, cf. chapter 3 in [1].

[11] With $(6 * 3600/25) * 0.9 = 777.6$.

[12] With $(6 * 3600/25) * 0.5 * 0.95 = 410.4$.

solver can do a great job.

Finally, the same double-sweep algorithm is also used in `turbmodels` (twice per water column with 2 or more layers). The turbulence model is not called as often as `momeqs`, typically only every second main time step, so the frequency will be like for the vertical tracer diffusion above.

It also seems worth to discuss the size of the system matrix: The distribution of kh, khu, khv on the BS domain is shown in figure 51. Note that a size of 100 seems quite normal in this subdomain.

It is very simple to conduct experiments with library solvers in the momentum equations code (`default_momeqs.f90`) and in the k- and ω-equations in the turbulence code (`turbmodels.f90`) where all we need to do is to substitute the call to the local `TriSolv` with a call to `ddtsvb`[13] and ensure that all resulting `s(:)` references from `TriSolv` are substituted with `d(:)` instead. Moreover, note that `ddtsvb` expects that `dl` and `du` have dimension `N-1` so we need to shift `a` before calling `ddtsvb` like this:

```
do ii=1,keu-1
  a(ii) = a(ii+1)
enddo
```

It is more involved to apply the library function in the vertical tracer diffusion (`tflow.f90`) since the trisolver is inlined in the code today and the vertical tracer diffusion is also intermixed with the horizontal diffusion. One need to do something like outlined in figure 16 to conduct this experiment.

10.2 NUMA

As revealed in section 7 there are temporary work arrays in the `tflow` module that are not probably NUMA initialized. Moreover, there are some places where we have used an inconsistent version of `domp_get_domain`. It is trivial to ensure consistent usage of `domp_get_domain` and we will do that but we do not expect a big gain since the places where we use it inconsistently are not significant in the profiles. As for the temporary arrays in `tflow` we could either introduce domain tailored work arrays and NUMA initialize these accordingly or we could use the most compute intensive subdomain to NUMA initialize the work arrays. We have chosen the latter approach since

[13] This is for MKL. For ESSL, it would be `dgtnpf`.

```
0) split horizontal and vertical diffusion.

1) exline the solver in diffusion in tflow.f90 and use the original trisolver

2) use loop splitting in diffusion so that we solve in three steps:
    s,
    t,
   the nc-2 additional tracers
   for the nc-2 additional tracers we should setup a,b,c and call trisolver
   in a nc-2 loop

3) substitute trisolver call for s and t

   From: call TriSolv(s, a, b, c, d, k)
   To:   call ddtsvb (k, 1, a, b , c , d, k, info)

4) substitute trisolver call for the nc-2 additional tracers

  From:
    do n=3,nc
      call TriSolv(s, a, b, c, d, k)
    enddo
  To:   call ddtsvb (k, nc-2, a, B , c , d, k, info)
```

Figure 16: The plan for preparing `tflow` for the MKL solver.

the first will require a lot of extra memory and we may easily just move the problem from a NUMA problem to a page-thrashing problem.

Another thing that one could consider in this context would be to ensure that all HEAP arrays are aligned at say page boundaries. Some compilers allow one to specify alignments when doing the allocations but this is not part of the standard. Other compiler have flags that allow one to have arrays spanning more pages aligned at page boundaries.

10.3 Vectorization

In [1] we explained how one should express the `k`-loop to allow the compiler to vectorize it, cf. figure 17. This works fine for point-local computations, i.e. for loops that does not need to inspect neighbouring cells, but this is unfortunately not the most common situation. In this section we will explain how one can prepare the innermost `k`-loops so that one can get them vectorized even when neighbouring cells are needed.

For instance, looking at `tflow_mp_c_rin_rout`, we have loops with memory access patterns like shown in figure 18.

That is, from point `mi` we look down (`md`), to the west (`mw`) and towards north (`mn`). In other places/other loops it is `me` (east) and `ms` (south) instead but the argumentation that follows below is the same.

```
! unroll k=1
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
  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
```

Figure 17: Fragment of the present code showing how one can vectorize point-local computation.

```
      do k=2,kb-1
        mi = m(k,  i,  j )
        mw = m(k,  i,  j-1)
        mn = m(k,  i-1,j  )
        md = m(k+1,i,  j  )
        dh   = dt/h_new(mi)
        humi = hx(mi)*u(mi)*facx *dh
        humw = hx(mw)*u(mw)*facx *dh
        hvmi = hy(mi)*v(mi)*facy1*dh
        hvmn = hy(mn)*v(mn)*facy2*dh
        hwmi =         w(mi)*      dh
        hwmd =         w(md)*      dh
        do ic=1,nc
          t8(ic,mi)=   max(humi*t1(ic,mi),zero) - min(humw*t1(ic,mw),zero)   &
                     - min(hvmi*t2(ic,mi),zero) + max(hvmn*t2(ic,mn),zero)   &
                     + max(hwmi*t3(ic,mi),zero) - min(hwmd*t3(ic,md),zero)
          t7(ic,mi)= - min(humi*t1(ic,mi),zero) + max(humw*t1(ic,mw),zero)   &
                     + max(hvmi*t2(ic,mi),zero) - min(hvmn*t2(ic,mn),zero)   &
                     - min(hwmi*t3(ic,mi),zero) + max(hwmd*t3(ic,md),zero)
        enddo
      enddo
```

Figure 18: Fragment of the present code where the compiler cannot see that it can vectorize.

With respect to `mw` and `mn` (or `me` and `ms`) we should just pick zeroes from the temporary work arrays (`t*`) when index goes below the sea bed in the neighbour columns, meaning we could waste a lot of memory accesses besides that the loops obviously do not vectorize as described above. We know that `mi` and `md` are stride-1 *by construction*[14] and we know how to hint the compiler about this stride-1 access, which is done successfully at several but far from all places in the code today, so we should do this everywhere. Moreover, we know that `mw` and `mn` are stride-1 too, and we can easily hint the compiler if we split the `k`-loop into more loops (three in the present case). An example is sketched in figure 19.

```
one loop running over $u$-columns to the west with hints on stride-1:
      mw0 = m(2,i,j-1) - 2
      mi0 = m(2,i,j) - 2
      do k=2,min(kb-1,khu(m(1,i,j-1)))
          mw = mw0 + k
          mi = mi0 + k
          md = mi + 1
          ...
          do ic=1,nc
             t8(ic,mi) = ...
             ...
 a similar loop over v-columns north:
      mn0 = m(2,i-1,j) - 2
      do k=2,min(kb-1,khv(m(1,i-1,j)))
          mn = mn0 + k
           ...
          do ic=1,nc
             ...
 and finally one which collects a possible remainder but which does not
 contain neither mw nor mn:
      do k= ..., kb-1
          mi = mi0 + k
          md = mi + 1
           ...
          do ic=1,nc
             ...
```

Figure 19: Outline of how the loop from figure 18 can be rewritten into three loops with hints on stride-1.

Thus we can turn the indirect addressed loop into a loop that clearly do memory access with stride-1, both in the `k`-loop and in the `ic`-loop, providing the compiler with sufficient information for proper optimization. Actually, this idea is not new, we have already implemented it in `masseqs.f90`. Our plan is now to carry this idea out through the entire code.

Another example is shown in figure 20 where we demonstrate how subloops

[14] For a thorough description of our data structures, indexing and cache layout, please consult chapter 2 in our previous report [1].

in `tflow_c_delta` can be vectorized.

```
Original code:
      call domp_get_domain(1, nbpz, nbpzl, nbpzu)
      do n=nbpzl,nbpzu
        i = krz(1,n)
        j = krz(2,n)
        ...
        kbvs = khv(m(1,i, j))
        do k=1,kbvs
          t5(1:nc,m(k,i,j))=t(1:nc,m(k,i,j))-t(1:nc,m(k,i+1,j))
        enddo
      enddo

Can be rewritten like:
      call domp_get_domain(1, nbpz, nbpzl, nbpzu)
      do n=nbpzl,nbpzu
        i = krz(1,n)
        j = krz(2,n)
        ...
        kbvs  = khv(m(1,i, j))
        kbvsi = khv(m(1,i+1, j))
        ! unroll k=1:
        k = 1
        t5(1:nc,1) =t(1:nc,1)-t(1:nc,m(1,i+1,j))
        ! k > 1
        if (kbvs > 1) then
          koff  = m(2,i,  j) - 2
          koffi = m(2,i+1,j) - 2
          do k=2,min(kbvs,kbvsi)
            t5(1:nc,koff+k) =t(1:nc,koff+k)-t(1:nc,koffi+k)
          enddo
          do k=min(kbvs,kbvsi)+1,kbvs
            t5(1:nc,koff+k) =t(1:nc,koff+k)
          enddo
        endif
      enddo
```

Figure 20: Outline of how the loops in `tflow_c_delta` can be written into more compiler friendly loops.

Finally, we notice that the `tflow` module is not handled very well in terms of vectorization due to the many references to `nc` - the number of tracers - and this number is determined at runtime so the compiler has no chance to make the right choices for the loops (typically array assignments) involving `nc`. In practice `nc` will be either 2 or 11 so we could make this a static choice instead, e.g. during the configure process.

As another example, let us look at the compute-intensive subroutine `momeqs`. Today, this subroutine does not vectorize well, cf. appendix G. Recall that this subroutine sets up and solves the momentum equations which end up with a tridiagonal system of equations:

$$a_i s_{i-1} + b_i s_i + c_i s_{i+1} = d_i,\ i = 1, \ldots, n,\ a_1 = 0,\ c_n = 0$$

We can work on the setup of the system or the solver itself. The solver was dealt with in a previous subsection 10.1, so we now confine ourselves to deal

with the setup of the system which is actually rather involved.

Before we consider to interchange loops and not have `k` as the innermost loop let us see if we could vectorize some of the `k` loops in the setup of the equation system above. The setup of the left-hand side of the equation, i.e. the assignment of `a(:)`, `b(:)` and `c(:)` can be partly vectorized (all but the loop-carried dependency) without much effort. There is at least one compiler that can see this even when expressed as we do it today, cf. figure 21. Alternatively, it can be rewritten as outlined in figure 22 whereby all compilers should be able to see it. Please consult appendix G for the details.

```
a(1) = zero
do k=1,keu-1
  tmp    = one/tm(k)
  a(k+1) = -dt*(avv(mmi+k-1)+avv(mme+k-1))/(tm(k)+tm(k+1))
  a(k)   = a(k)  *tmp
  c(k)   = a(k+1)*tmp
  b(k)   = one - (a(k) + c(k))
enddo
a(keu) = a(keu)/tm(keu)
b(keu) = one - a(keu)
c(keu) = zero
```

Figure 21: The original fragment of the code for setting up `a,b,c` for the u-equation.

```
do k=1,keu-1
  tmp2   = -dt*(avv(mmi+k-1)+avv(mme+k-1))/(tm(k)+tm(k+1))
  a(k+1) = tmp2
  c(k)   = tmp2/tm(k)
enddo
do k=1,keu-1
  tmp  = a(k)/tm(k)
  a(k) = tmp
  b(k) = one - (tmp + c(k))
enddo
a(keu) = a(keu)/tm(keu)
b(keu) = one - a(keu)
c(keu) = zero
```

Figure 22: The simple rewrite of the code for setting up `a,b,c` for the u-equation allowing all compilers to vectorize the two loops.

The setup of the right-hand side of the equation is the more involved part. It takes approximately 80 code lines to reach the definition:

```
d(k) = umi + fti*vst - ucon + uaus - upress - usrf
```

This is all done in one large `k`-loop but we could split it so that the two compute-intensive parts of the loop could vectorize. We would need to

unroll `k=1` and we would need to split the remaining `k=2,keu` loop into a vector candidate loop `k=2,keu_min` and a remainder loop `k=keu_min+1,keu`. Note that we need some extra book-keeping, namely arrays `keu_min(:)` and `kev_min(:)` that give us the upper bound on the `k`-subloop where all the east, west,... neighbours exists. If we know that they exists then we can do stride-1 access in these points too, cf. figure 23.

Alternatively, we can take the same approach as used in `tflow` and split the loop into many thin subloops that all will vectorize. This approach is outlined in figure 24 and in figure 25.

In appendix G we show how different compilers handle the different versions of the code and one should note that with the second approach all compilers are capable of generating vector code for the loops. However, there is no guarantee that the compiler generates *good* vector code nor that the generated vector code is actually chosen at runtime. One of the things that may prevent good vector code is lack of alignment information at build time. Thus, we probed alignment information at runtime and saw that all arguments passed onto `momeqs` were indeed properly aligned when using this compiler option `-align array64byte` so we tried to use the alignment directives below to hint the compiler about this at build time:

```
!DIR$ ASSUME_ALIGNED U: 64
!DIR$ ASSUME_ALIGNED UN: 64
... for all array arguments
```

Alas, we were not able to measure any effects of this and we made the conclusion that the relatively poor performance on the Xeon Phi is not due to lack of vectorization in the computationally expensive subroutines. In subsection 10.5 we discuss what we could do to improve the vector code generated.

10.4 Memory latency in momeqs

We found that the `momeqs` rewrites did indeed improve the performance on the Xeon Phi but we were not able to measure any benefits on the two CPU based systems so in search for an explanation we did a rough estimate of the memory required per outer-loop iteration in `momeqs`. If we have a cache latency issue then we generally expect that a larger number of cores will reduce the problem since a larger number of cores generally mean a larger

```
do k=1,1
  ! unroll k=1
enddo

mik = mm(2,i  ,j  )
nnk = mm(2,i-1,j  )
ssk = mm(2,i+1,j  )
nek = mm(2,i-1,j+1)
eek = mm(2,i  ,j+1)
sek = mm(2,i+1,j+1)

! THE COMPILER SHOULD BE ABLE TO VECTORIZE THIS LOOP
do k=2,khumin
  mi = mik+(k-2)
  nn = nnk+(k-2)
  ss = ssk+(k-2)
  ne = nek+(k-2)
  ee = eek+(k-2)
  se = sek+(k-2)
  umi = u(mi)
  vst  = qrt*(v(nn) + v(ne) + v(mi) + v(ee))
  upress = (press(ee)-press(mi))*ttx/(rho(ee)+rho(mi))
  d(k) = umi + fti*vst - upress - usrf
  urd_v(k)  = umi*txi
  vrd_v(k)  = vst*ty
enddo

do k=2,khumin
   all the iff stuff
   ! compute uaus0_v(k)
enddo

! THE COMPILER SHOULD BE ABLE TO VECTORIZE THIS LOOP
do k=2,khumin
  mi = mik+(k-2)
  nn = nnk+(k-2)
  ss = ssk+(k-2)
  ne = nek+(k-2)
  ee = eek+(k-2)
  se = sek+(k-2)
  ! Horizontal eddy viscosity:
  ehmi = eddyh(mi) + eddyh(ee) + eddyh(ss) + eddyh(se)
  ehnn = eddyh(mi) + eddyh(ee) + eddyh(nn) + eddyh(ne)
  uaus_v(k) = dtd*uaus0_v(k)                                              &
       + ( (eddyh(ee)*stretch(ee)-eddyh(mi)*stretch(mi))*txi              &
          +(ehnn     *shear  (nn)-ehmi     *shear  (mi))*fy              &
          +(eddyd(ee)*div    (ee)-eddyd(mi)*div    (mi))*fxh )
  d(k) = d(k) - ucon_v(k) + uaus_v(k)
enddo

do k=khumin+1,keu
  remaining k's ! as today
enddo
```

Figure 23: Fragment of the code for setting up `d` for the u-equation. First approach with three relatively thick inner loops, two of which should vectorize.

```
do k=1,1
  ! unroll k=1
enddo

mik = mm(2,i  ,j  )
nnk = mm(2,i-1,j  )
ssk = mm(2,i+1,j  )
nek = mm(2,i-1,j+1)
eek = mm(2,i  ,j+1)
sek = mm(2,i+1,j+1)

! THE COMPILER SHOULD BE ABLE TO VECTORIZE ALL LOOPS BELOW
do k=2,keu
  mi = mik+(k-2)
  ee = eek+(k-2)
  d(k) = - (press(ee)-press(mi))*ttx/(rho(ee)+rho(mi)) - usrf
enddo
do k=2,keu
  mi = mik+(k-2)
  umi = u(mi)
  d(k) = d(k) + umi
  urd_v(k) = umi*txi
enddo
do k=2,keu
  mi = mik+(k-2)
  vst  = qrt*v(mi)
  d(k) = d(k) + fti*vst
  vrd_v(k) = vst*ty
enddo
do k=2,keu
  ee = eek+(k-2)
  vst  = qrt*v(ee)
  d(k) = d(k) + fti*vst
  vrd_v(k) = vrd_v(k) + vst*ty
enddo
do k=2,min(keu,kh(mm(1,i-1,j)))
  vst  = qrt*v(nnk+(k-2))
  d(k) = d(k) + fti*vst
  vrd_v(k) = vrd_v(k) + vst*ty
enddo
do k=2,min(keu,kh(mm(1,i-1,j+1)))
  vst  = qrt*v(nek+(k-2))
  d(k) = d(k) + fti*vst
  vrd_v(k) = vrd_v(k) + vst*ty
enddo
.... to be continued
```

Figure 24: Fragment of alternative code for setting up `d` for the u-equation. Second approach with many thin subloops, part I.

```
  do k=2,keu
    mi = mik+(k-2)
    ee = eek+(k-2)
    ehmi = eddyh(mi) + eddyh(ee)
    uaus = ( (eddyh(ee)*stretch(ee)-eddyh(mi)*stretch(mi))*txi             &
           +(                      -ehmi     *shear  (mi))*fy              &
           +(eddyd(ee)*div    (ee)-eddyd(mi)*div    (mi))*fxh )
    d(k) = d(k) + uaus
  enddo
  ! THE COMPILER SHOULD BE ABLE TO VECTORIZE ALL LOOPS BELOW
  do k=2,min(keu,kh(mm(1,i-1,j)))
    nn = nnk+(k-2)
    ehnn = eddyh(mik+(k-2)) + eddyh(eek+(k-2)) + eddyh(nn)
    d(k) = d(k) + ehnn*shear(nn)*fy
  enddo
  do k=2,min(keu,kh(mm(1,i-1,j+1)))
    d(k) = d(k) + eddyh(nek+(k-2))*shear(nnk+(k-2))*fy
  enddo
  do k=2,min(keu,kh(mm(1,i+1,j)))
    d(k) = d(k) - eddyh(ssk+(k-2))*shear(mik+(k-2))*fy
  enddo
  do k=2,min(keu,kh(mm(1,i+1,j+1)))
    d(k) = d(k) - eddyh(sek+(k-2))*shear(mik+(k-2))*fy
  enddo
  do k=2,keu
    d(k) = d(k) + dtd*(u(eek+(k-2)) - two*u(mik+(k-2)))
  enddo
  do k=2,min(keu,kh(mm(1,i-1,j)),kh(mm(1,i-1,j+1)))
    d(k) = d(k) + dtd*(u(nnk+(k-2)) - u(mik+(k-2)))
  enddo
  do k=2,min(keu,kh(mm(1,i+1,j)),kh(mm(1,i+1,j+1)))
    d(k) = d(k) + dtd*(u(ssk+(k-2)) - u(mik+(k-2)))
  enddo
  do k=2,min(keu,kh(mm(1,i,j-1)))
    d(k) = d(k) + dtd*u(wwk+(k-2))
  enddo

  ! THE LOOP BELOW IS THE ONLY ONE THAT WILL NOT BE VECTORIZED
  do k=2,keu
    urd = urd_v(k)
    vrd = vrd_v(k)
    ....
    aurd = abs(urd)
    avrd = abs(vrd)
    if (urd >= zero) then
      j2 = j-1
    else
      j2 = j+1
    endif
    if (vrd >= zero) then
      i4 = i+1
    else
      i4 = i-1
    endif
    if (aurd >= avrd) then
      i1 = i
      j3 = j2
    else
      i1 = i4
      j3 = j
    endif
    d(k) = d(k) - aurd*(u(mm(k,i1,j))-u(mm(k,i1,j2)))                      &
                - avrd*(u(mm(k,i,j3))-u(mm(k,i4,j3)))
  enddo
```

Figure 25: Fragment of alternative code for setting up `d` for the u-equation. Second approach with many thin subloops, part II.

amount of cache memory available to the application. Is this what we see and can we do some rewrites that allow us to improve this part ?

Let us try to estimate the number of bytes we need per outer-loop iteration. Let us confine ourselves to deal with the arrays that are part of the innermost k-loop. That is for u:

avv*2, press*2, rho*2, eddyh*6, eddyd*2, shear*2, div*2, stretch*2, hx*1, u*9, v*4, a*1, b*1, c*1, d*1

and for v:

avv*2, press*2, rho*2, eddyh*6, eddyd*2, shear*2, div*2, stretch*2, hy*1, v*9, u*4, a*1, b*1, c*1, d*1

If we isolate u and v then we have 2*38 arrays in total in the innermost loop so if the k-loop tripcount is 100 then we need: (38 arrays * 8 bytes/element * 100 iterations) $\approx$ 60kb for the arrays in the innermost k-loops per outer iteration. This exceeds the total size of D1 (32kb) on both Xeon X7550 and Sandy Bridge. The size of D1 on the Xeon Phi is 32kb per core so the issue is even more serious here. Thus, momeqs will be very expensive to run on the Xeon Phi, especially when we run with more than one thread per core.

We will now briefly discuss how we could improve this. First, there are a few of the arrays that do not change every timestep, e.g. rho, press and avv and we should consequently be able to reduce the total number of computations where these arrays are involved and only redo them when required. Second, we could consider handling some of the computations more locally either outside the subroutine when the arrays are formed or as a first step in momeqs before entering the other innermost loop. Third, we could mix the handling of u and v so that we ensure that common arrays (most of them are common but not all the directional indices are common) are used just after each other so that the relevant cachelines are not moved back to L2 before we need them again. Finally, we could consider storing some of the coefficient arrays using real(4) instead of real(8).

Note that in the myov3 setup, rho, press and avv are only changed every fourth step for the two largest subdomains BS and IDW. Both rho and press are set in cmod_dens and press is only used in momeqs. Thus, we should be able to pass this information onto momeqs in only one single ar-

ray and we should only redo the computations involving these arrays solely every fourth step and not in every step as we do today.

The arrays `shear`, `div` and `stretch` are set in `deform` and used in `momeqs` and `smagorinsky`. Today they are recomputed in every step, but it is worth analysing whether or not we really need to recompute them so often. Moreover, one could improve locality by doing `deform` and `smagorinsky` just after each other instead of having the turbulence in between them. One could also prepare the momeqs computations with these 5 and store the results in two new arrays, one for u and one for v, and pass these into `momeqs` instead of the 5 original arrays and thereby reduce the D1 pressure in `momeqs` and further improve locality for the 5 arrays.

The array `eddyh` is set in `smagorinsky` and used there and later in `momeqs` and in the horizontal diffusion in `tflow`. The array `eddyd` is set in `smagorinsky` and only used there besides later in `momeqs`. For these two arrays, as for arrays `shear`, `div` and `stretch` above, it also makes good sense to analyze whether or not we really need to re-compute them at every step.

An educated guess is that we would do fine with the two horizontal eddy viscosity terms, `eddyh` and `eddyd`, as well as the three deformation terms, `shear`, `div` and `stretch`, being calculated at the same frequency as the vertical eddy viscosity, `avv`, i.e. only every fourth step for the largest domains in the myov3 case.

We certainly intend to pursue these considerations when time permits it.

10.5 Improving the vector code generated by the compiler

A bird's-eye view on how the subroutines are tied together reveals that the implementation use pointer arrays as actual arguments and assumed-shape arrays as dummy arguments, cf. figure 26. There are two reasons explaining the current organization.

Firstly, it was a desire *not* to have neither the number of sub-domains nor the sizes of the sub-domains hard-coded, and thus we were aiming at a rank-2 data structure supporting a number of entries, determined at run time, each with a different number of elements, also determined at runtime; pointers in derived types as show in figure 26 was at that time the only

```
subroutine foo(a)
  implicit none
  real(8), intent(out) :: a(0:)
  ! ...
end subroutine foo
type cmr1
  real(8), pointer :: p(:)  => null()
end type
type(cmr1), pointer :: a(:)
allocate(a(narea))
do ia=1,narea
  allocate( a(ia)%p(0:ub(ia)) )
enddo
do ia=1,narea
  call foo(a(ia)%p)
enddo
```

Figure 26: A code snapshot illustrating the way the sparse arrays are declared and passed around throughout the implementation.

way to achieve this since the F90/F95 standard did not support allocatables in derived types. Allocatable derived type components were added by a TS (Technical Specification) that was published between f95 and f2003 and that TS was incorporated into F2003. Secondly, one would like to have MPI support added to the code. We tried to kill two birds with one stone, and by using pointers one could easily let MPI global arrays and MPI local arrays be the same thing in serial builds and purely openMP builds and then have them mean different things in the hybrid openMP+MPI build or in the pure MPI build. One could have handled it with allocatables instead as outlined in figure 27 and we could do that without having to change any subroutine calls in the code but that would incur a performance penalty on the serial and the pure openMP builds and that was not an option at the time when this was introduced.

There are at least three problems (as seen from a performance perspective) with the current approach with pointers, namely:

- Compilers will generate code that allows for aliasing.
- Compilers will not be able to deduce that the actual arguments point to contiguous memory. In general, when passing pointers, compilers have to allow for non-unit strides.
- Compilers need to inspect the dope vectors at runtime and see if the actual argument is indeed contiguous and then pass the address of the array. This causes run-time overhead and we know (by definition) that this check is not needed but we do not expose this knowledge to the compiler.

```
subroutine foo(a)
  implicit none
  real(8), intent(out) :: a(0:)
  ! ...
end subroutine foo
type cmr1
  real(8), allocatable :: p(:)
end type
type(cmr1), allocatable :: a(:)
allocate(a(narea))
do ia=1,narea
  allocate(a(ia)%p(0:ub(ia)))
enddo
do ia=1,narea
  call foo(a(ia)%p)
enddo
```

Figure 27: A code snapshot illustrating how one could have dealt with the first desire without using pointers. Note that having allocatable components of derived types is a F2003 feature but most F95-compliant compilers have implemented this extension.

The first problem could be dealt with using compiler flags. There are several compilers that have flags for this.

The second problem could in theory also be dealt with using compiler flags but very few compilers have such flags today. In practice, the second problem is more involved to deal with in a portable fashion. One could try to use the attribute `contiguous` introduced in the F2008 standard but not all compilers support this today, or we could consider switching to explicit bound specification or assumed-size specification of dummy arguments. Both of these ideas would make the code less portable unless we handle it during the build configure process. The F77 specification is tricky to use because we would then rely highly on the compiler implementations. By using this syntax one will force the compiler to ensure that all dummy arguments are indeed contiguous and there are at least two ways of accomplishing this. One is to inspect the dope-vector at runtime and if the actual argument is contiguous, the address of the array is passed, otherwise a compiler temporary is created, the non-contiguous array is copied and the address of the copy is passed. We know that the latter will not happen and the address of the array is passed and things will work fine. Another approach (i.e. another implementation) is to do the copy-in/copy-out constructions at build time but these copy-in/copy-out constructions will fail for `intent(out)` and `intent(inout)` dummy arguments due to the exlined openMP constructs used throughout the code, c.f. figure 28.

```
subroutine foo ...
  ...
  call domp_get_domain(kh, 1, iw2, nl, nu, idx) ! openMP decomposition
  do nsurf=nl,nu
    i = ind(1,nsurf)
    j = ind(2,nsurf)
    ! all threadlocal wet-points (:,:,:) are reached here
    ...
  enddo
  ...
end subroutine foo
...
!$OMP PARALLEL DEFAULT(SHARED)
call foo( ... )
call bar( ... )
!$OMP BARRIER
call baz( ... )
!$OMP END PARALLEL
...
```

Figure 28: F77-syntax for dummy arguments in foo would rely on the usage of dope-vectors at runtime and would break if copy-in/copy-out constructions were generated at build time. It is unlikely that the compiler will keep track of which thread changes what and be able to do the copy-out upon return correctly.

The third problem could be reduced by heavy use of inlining but in summary there are very good reasons to reconsider the current design and get rid of the pointers. The pure openMP builds are no longer our default builds and if a rewrite will potentially improve the performance of the hybrid openMP+MPI build and incur a performance penalty on the pure openMP builds then we can accept it.

10.6 Load balancing

In this subsection we will pose various approaches to improve the thread load balance. We will not repeat the relevant background information on the balance problem already described at page 43-52 in [1] and in section 7 in the present report so please study these before reading this section.

Larry Meadows from Intel has kindly profiled the application and demonstrated that we have load balancing issues at 240 threads, cf. figure 29 where he shows the time spend in `libiomp5`[15] for one of the most expensive subroutines; the large deviations across the threads indicate that we have load balance problems. It can be a little difficult to understand and analyse from these raw measurements, so as a start, a simplified illustration of the overall

[15]The time spend in `libiomp5` is the thread waiting time.

balance issue is given in figure 30 where we have tried to sort the threads according to their total time spend in `libiomp5`.

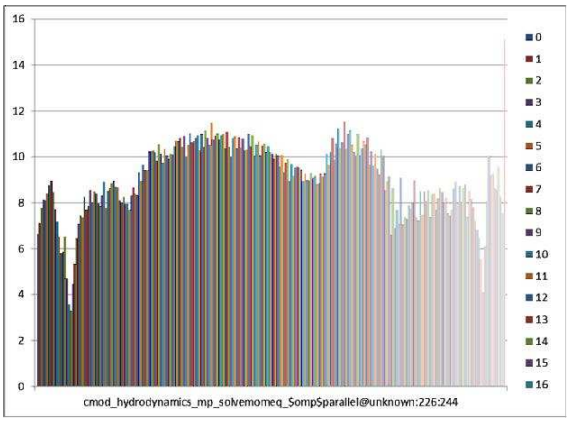


Figure 29: Thread imbalance issue measured on the Xeon Phi. This plot was generated by Larry Meadows and it shows the thread imbalance issue on the Xeon Phi for a part of the most expensive subroutine. The y-axis is the time in `libiomp5`. Along the x-axis are the parallel regions so there is a bar for each of the 240 threads.

We will not be able to improve this situation unless we understand what we observe, i.e. we need to analyse and relate the numbers emerging from the profile with that of the decomposition defined by the current heuristic. Appendix E contains information on the decomposition that emerges when using 240 threads and we have sorted the numbers in various ways and also plotted the numbers from the decomposition as well as from the profile. Moreover, we look at the numbers on a per area basis as well as total numbers. The reason for this is due to the two main kinds of `openMP` constructions that we have in the code today, cf. figure 31.

We may also try to use regression analysis to relate the profile statistics to the decomposition statistics. Thus, let nt be the thread number, and let $iw_3(nt)$ be the number of 3D wetpoints on thread number nt, $h_3(nt)$ be the numbers of 3D halo wetpoints on thread number nt, $iw_2(nt)$ be the number of surface wetpoints on thread number nt and finally $h_2(nt)$ be the number of surface halo wetpoints on thread number nt. Moreover, during the analysis it turned out that it was more handy to use the reciprocal of the

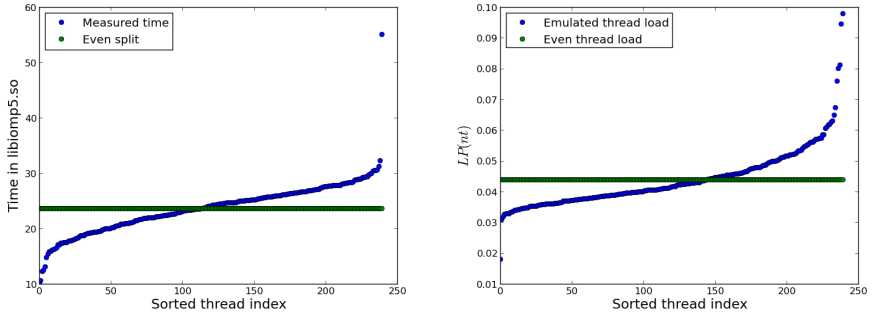


Figure 30: Thread imbalance issue measured on the Xeon Phi. Left: The y-axis is again the time spend in `libiomp5` as in figure 29, but in the present plot the data has been sorted and the x-axis is the index 1:240 from the sorting (i.e. the pool of threads permuted). The green line indicates an even split of the work and represents a perfectly balanced application, so it is clear that there is a lot of room for improvement here. Right: Same as on the left plot but instead of y we have plotted $1/y$ to emulate the thread load based on the thread waiting time, and the x-axis is the index 1:240 from sorting $1/y$.

measured time in `libiomp5` than the raw measurement. We will denote the reciprocal value by $LP(nt)$. If we confine ourselves to the four parameters and apply linear regression we find that the measured load profile $LP(nt)$ can be expressed as

$$LP(nt) = a_0 + a_1 * iw_3(nt) + a_2 * h_3(nt) + a_3 * iw_2(nt) + a_4 * h_2(nt)$$

with $a_0 = -0.079180444$ and with

$a_1 = 2.2999119e-06 \pm 3.8542153e-07$ with significance: 5.9672636
$a_2 = 7.6242388e-08 \pm 6.8580936e-08$ with significance: 1.1117140
$a_3 = 3.9998704e-05 \pm 4.0261109e-06$ with significance: 9.9348243
$a_4 = 5.8200283e-07 \pm 1.1396806e-06$ with significance: 0.51067189

The Pearson correlation coefficient between $LP(nt)$ and the result of the regression analysis $\widehat{LP(nt)}$ is 0.83407898. The significance is defined as the value of a_i divided by its uncertainty, whereby we surprisingly conclude that it seems that we can exclude the halo parameters without loosing significant

```
construction1:
!$OMP PARALLEL DEFAULT (shared) PRIVATE(ia)
  do ia=1,narea
    call foo(iw2(ia), iw3(ia),....)
    call bar(iw2(ia), iw3(ia),....)
    ...
  enddo
!$OMP END PARALLEL

construction2:
do ia=1,narea
  call foo(ia,...)
  call bar(ia,...)
enddo
and within foo and bar we have constructions like
!$OMP PARALLEL DEFAULT (shared)
call baz( ... )
call quux( ... )
!$OMP END PARALLEL
...
```

Figure 31: The two main kinds of `openMP` constructions. With the first kind we need to analyse the numbers across all the domains and with the second kind we need to analyse the numbers on a per domain basis.

information. With only the two most significant parameters (iw_3, iw_2) we get $a_0 = -0.077383095$ and

$$a_1 = 2.5610786e-06 \pm 3.7290310e-07 \text{ with significance: } 6.8679466$$
$$a_3 = 3.4530795e-05 \pm 1.5460429e-06 \text{ with significance: } 22.334953$$

with a Pearson correlation coefficient of 0.82903764. We have compared the linear models and the measured profile in figure 32.

The difference between observation and models does not seem to be noise exclusively, there are indeed some systematic discrepancies, so we should be able to improve the descriptive model further. For instance, the model does not take outliers into account. As an example, the largest peak at threads 17-18 is not caught by the model, neither is the small peak at threads 12-13, while the significant peak at threads 228-230 is. Seeking an explanation we dig into appendix E and find from table 29 and figure 55 that threads 228-230 indeed have high values of $iw_2(nt)$ in the largest BS domain, and our model will weight $LP(nt)$ high here. On the other hand, neither threads 17-18 nor 12-13 are particularly important in BS. But as seen from table 29 and figure 53 these threads have high values of $iw_2(nt)$ in the IDW domain, but not so high as the typical BS $iw_2(nt)$-values so we must dig deeper to explain. From table 29 we see that especially threads 17-18-19 have a very large 2D halo size, $h_2(nt)$, in IDW, both compared to $iw_2(nt)$ at the same threads in IDW and as compared to the $h_2(nt)$ values in BS, so we might

expect a relatively huge cacheline fight with neighbour threads here. Being based on the total numbers, our regression model can of course not describe such per-area behavior. At the moment we will, however, consider the above regression analysis as sufficient for a first shot towards improving load balancing and see how far that will take us.

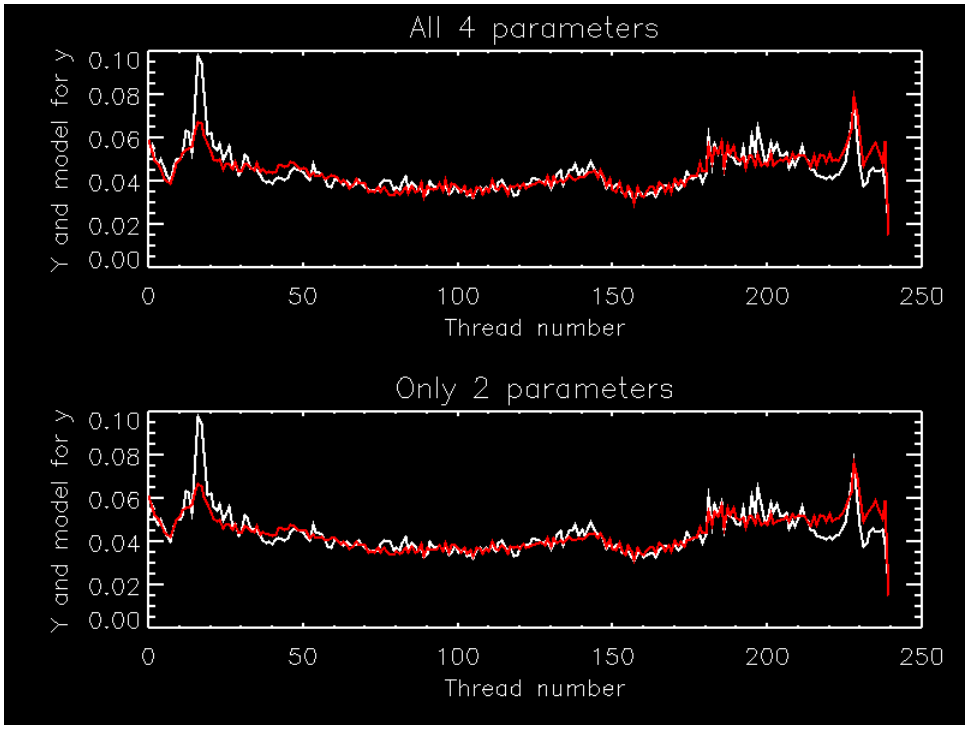


Figure 32: The measured load profile $LP(nt)$ (white curve) and the linear model (red curve) emerging from the regression analysis with all four parameters (top) and with only two parameters (bottom).

Now let us see how we can use this model to design a better thread decomposition heuristic. Let NT denote the total number of threads, say 240. Let nt again be the thread number and let us define a load balance number $LB(nt)$ for each thread number, $nt \in 1, 2, \ldots, NT$, as:

$$LB(nt) = a_0 + a_1 * iw_3(nt) + a_2 * h_3(nt) + a_3 * iw_2(nt) + a_4 * h_2(nt)$$

Assume that the best we can do is to strive for an even-split of $LB(nt)$ across all threads, i.e. $LB(nt) \approx c$, with $c \in \mathbb{R}$ being the constant

$$c = \frac{1}{NT} \sum_{nt=1}^{NT} LB(nt)$$

The original heuristic used $a_1 = 1$ and $a_i = 0$ for $i \neq 1$ and worked quite well for lower thread counts, but as shown in table 31 in appendix E, this is not a fully sufficient approach for 240 threads. Note that for the current testcase we actually see that $h_3(nt)$ becomes larger than $iw_3(nt)$ at many threads. Let us try to incorporate these observations into a refined thread-decomposition.

The complication with the formula above is that we do not know the values of $iw_3(nt)$, $h_3(nt)$, $iw_2(nt)$ and $h_2(nt)$ beforehand. That is, we do not know these numbers before we have constructed the actual decomposition. But when we have decided a decomposition we can calculate all four of them. A practical approach thus seems to be through iterations

$$LB(nt; it) = a_0 + a_1 * iw_3(nt; it-1) + a_2 * h_3(nt; it-1) + \ldots, \;\; it = 1, 2, ...$$

starting from an initial guess which is the distributions that we have from our old heuristics, i.e. where $iw_3(nt; 0)$, $h_3(nt; 0)$, $iw_2(nt; 0)$ and $h_2(nt; 0)$ are the ones obtained from an (nearly) even-split of iw_3. The decomposition of iteration $\#it$ is constructed so that we obtain an (nearly) even distribution of the load across all threads, i.e. we attempt to have

$$c(it) = \frac{1}{NT} \sum_{nt=1}^{NT} LB(nt; it)$$

on each thread.

We can most easily do the decomposition on a per area basis. For the values of coefficients a_i that describe $LB(nt; it)$ for each area we can take the values of the corresponding coefficients obtained from our regression analysis shown above for the total setup. We may then iterate for as long as we want to, or until $LB(nt; it)$ for each area has converged sufficiently well. Remember, finding the perfect solution is NP-complete and the iterative approach described above is nothing but a reasonable heuristic based on LP and LB;

we are merely trying to improve what we already had.

The results from our first attempt with this iterative decomposition generator is shown in figure 33. After just two iterations $LB(nt; it)$ did not change any more in any domain. The figure shows oscillations of much smaller magnitude than the original (i.e. the red curves in figure 32). Our implementation of the decomposition heuristic described above now resulted in a slightly too high $LB(240)$ (before it was much too low). In a second attempt we made the distribution of $LB(nt)$ across the threads a little more uniform and thus we got rid of the peak at 240 threads, see figure 34. Now the range of variations has dropped by a factor of approximately 100 compared to what we started off with (i.e. the red curves in figure 32). We must remember the inherent obstacle for generating a perfectly smooth decomposition across all threads and that is that the two quantities $iw_3(nt)$ and $h_3(nt)$ will always be incremented in chunks of water columns.

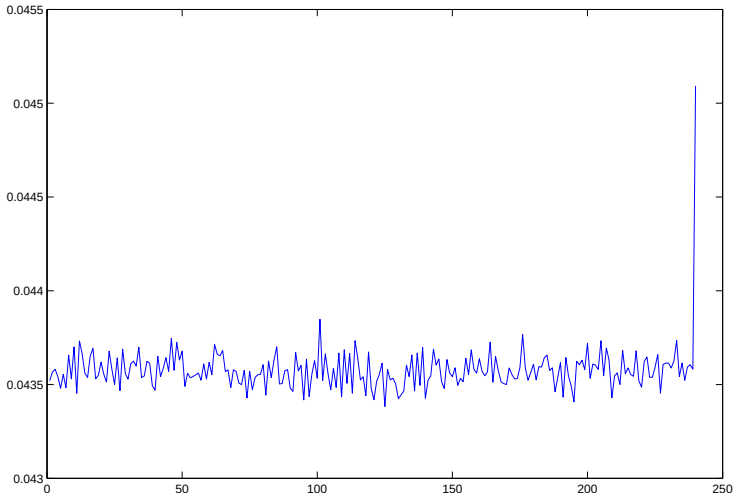


Figure 33: The calculated load balancing distribution $LB(nt)$ from the iterative decomposition generator with coefficients a_i from the linear regression model.

We trust that the method described above can solve the balance problems but we need to tune the parameters and in order to do so we probably need a few iterations, i.e. we need to:

`REPEAT:`

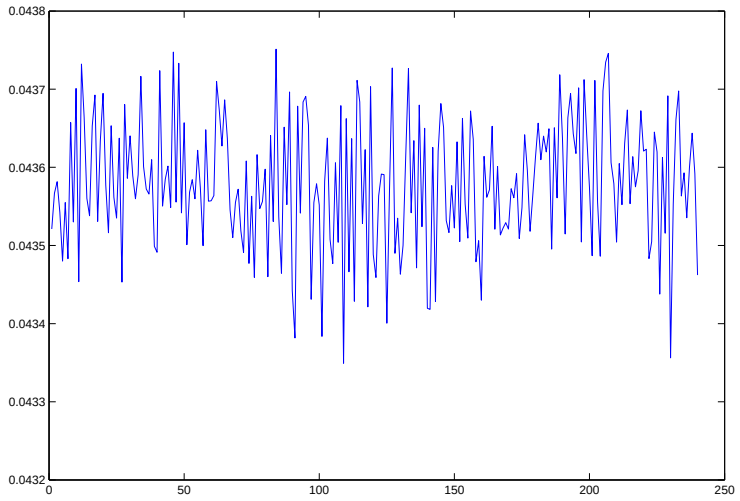


Figure 34: The calculated load balancing distribution $LB(nt)$ from the slightly modified iterative decomposition generator with coefficients a_i from the linear regression model.

```
run with new heuristic
get profile numbers LP(nt)
use LP(nt) to adjust the parameters used in the heuristic
```

The first run using the new heuristic improved the overall performance by 7% so it seems worth to take a couple of iterations to refine this.

It should be mentioned that the whole analysis above builds on two assumptions that we need to elaborate on, namely *i)* that the thread load numbers are reproducible and rather accurate, and *ii)* that the total load can also be used as a measure for the per-domain load. If *i)* we can not obtain reliable load profiles, or *ii)* we cannot use the estimated coefficients from the total profile on a per-domain basis, the outcome will likely fail to perform better. The latter assumption will not always hold if we turn towards other setups, cf. appendix H and this is something we need to address eventually too. Maybe we should aim at doing constructions like the first shown in figure 31 so that we can use heuristics that works across all subdomains and thus allows an even split also in cases where the size of the individual subdomains differs a lot.

11 Profiling using the new decomposition

In this section we will analyse the profile emerging from the run of the experiment termed `ex49-ni` which was the first experiment using the iterative decomposition strategy described in section 10.6. Appendix F provides some detailed information for the decomposition used in this experiment. Please note that while the profile measurements in the previous section was the time spend in `libiomp5` we here measure the time spend by each thread; this should make it more easy to understand what is going on and to relate observations to model quantities. The profile is summarized in figures 35 and 36 and in table 19. The whole run took 108 seconds of which 100 seconds were in parallel blocks and the remaining 8 seconds was spend in serial blocks. The serial blocks amounts to approximately 7% so this is an obvious candidate for improvement but this is not the topic of this section.

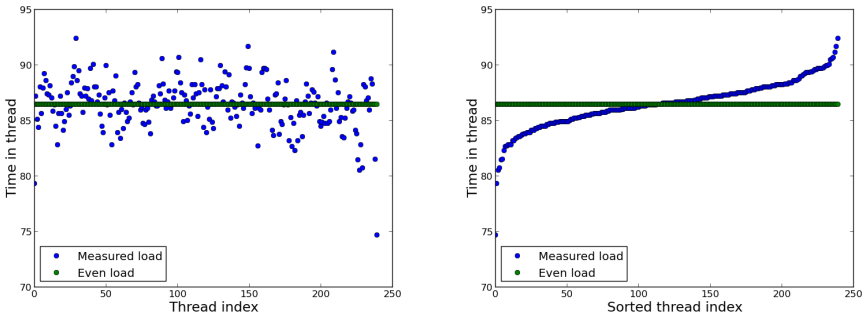


Figure 35: Thread imbalance issue measured on the Xeon Phi using experiment `ex49-ni`. Note that the difference between the most expensive thread and the mean load is close to 6 seconds which amounts to approximately 5% (total runtime for `ex49-ni` was 108 seconds).

From table 20 and from figure 37 we conclude that the 4 parameters that we have focused on thus far are not sufficient to describe the measurements done this time. We do *not* know if the measurements are indeed reproducible nor do we know if the explanation of these measurements should be found in the hardware and/or in the system software layer or within the model code and the current setup itself.

Figure 36 shows that the balance issues are present in subroutine `momeqs`

OMP block	Sum	Min	Max	Mean	STD
solvemomeq@226:244	5837.57	20.932358	26.234004	24.3232	0.739329
solvemomeq@180:199	3424.5	13.162706	15.447899	14.2687	0.498905
tflow_int@3131:3145	2438.94	5.941498	11.700183	10.1622	0.773837
tflow_int@3169:3173	2197.71	6.855576	11.700183	9.15715	0.750696
tflow_int@3183:3187	1249.91	3.290676	6.215722	5.20795	0.541285
solvemomeq@139:169	762.34	2.376599	4.113345	3.17642	0.338692
tflow_int@3322:3350	689.58	2.010968	3.656307	2.87325	0.341147
tflow_int@3214:3313	653.931	1.919561	3.290677	2.72471	0.265658
solvemasseq@489:510	590.128	1.736746	3.016454	2.45887	0.245811
tflow_int@3154:3159	525.686	1.279707	2.833638	2.19036	0.308214
solvemasseq_z@550:564	502.834	1.462523	2.833639	2.09514	0.297923
tflow_int@3196:3200	463.528	1.371115	3.199269	1.93137	0.307214
MAIN@1230:1237	459.232	1.1883	2.74223	1.91347	0.284378
MAIN@1720:1732	363.62	0.731261	2.193784	1.51508	0.255605
MAIN@1609:1634	245.795	0.548446	1.553931	1.02415	0.211014
MAIN@1068:1112	200.64	0.457038	1.279708	0.835999	0.170794
MAIN@993:1014	57.0384	0	0.548446	0.23766	0.108733
solvemomeq@297:307	23.5832	0	0.457038	0.0982634	0.0922889
MAIN@946:958	18.5558	0	0.365631	0.0773158	0.0785645
MAIN@1457:1466	17.8245	0	0.639854	0.0742688	0.125798
MAIN@1672:1691	17.0018	0	0.731262	0.070841	0.111928
solvemasseq_z@570:573	5.66729	0.0	0.274223	0.0236137	0.0471967
massnest_z@703:709	3.01646	0	0.182816	0.0125686	0.0336178

Table 19: A summary of the profile information. Individual statistics for the time used by each of the 240 openMP threads for the 23 most time consuming openMP blocks.

	$n2d$	$h3d$	$h2d$
$n3d$	-0.98750126	0.81115859	0.39682246
$n2d$		-0.89272082	-0.51815027
$h3d$			0.77500587

Coefficient	**Value**	$\pm$	**Significance**
a_0	176.14361		
a_1	-0.0013136499	0.0032610463	-0.40283080
a_2	-0.041150995	0.056902219	-0.72318788
a_3	-0.00018938709	0.00011383040	-1.6637655
a_4	0.0036603346	0.0010738286	3.4086768
a_0	27.180970		
a_1	0.0013686821	0.00035052085	3.9047095
a_2	0.012544247	0.0047947962	2.6162211
a_0	106.06904		
a_2	-0.018243411	0.0020118670	-9.0679014
a_3	-0.00014441753	2.2214415e-05	-6.5010726
a_4	0.0039319393	0.00083428334	4.7129544

Table 20: Upper part: The correlation between the four input parameters. There is a strong anti-correlation between $n2d$ and $n3d$ and we can probably exclude one of them. Lower part: Results of three linear regression analyses using 4, 2 and 3 parameters. The Pearson correlation coefficient was $R = 0.56710102$, $R = 0.49547223$ and $R = 0.56668785$, respectively. The maximal absolute difference between the measurements and the result of the regression is 5.5, 7.8 and 5.3, respectively.

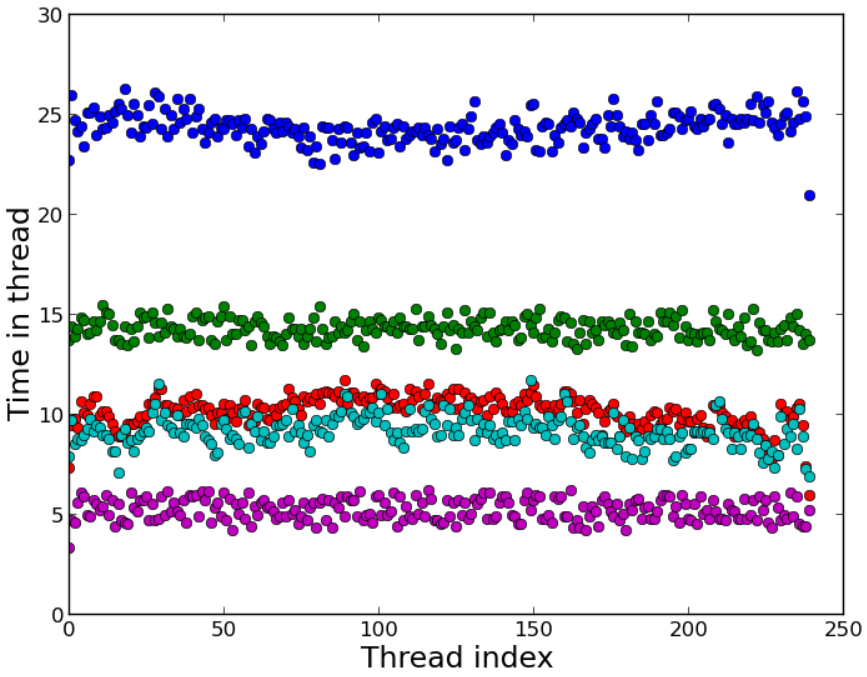


Figure 36: This figure focuses on the top5 individual chunks in table 19 and shows the time each thread spend.

(top 1,2,6 and 18) so we only need to inspect the data structures and control flow of that subroutine in order to seek an explanation of the overall balance issues. Note that `momeqs` uses `khu` and `khv` and that each outer loop need to look at neighbour columns in 8 directions. Moreover, recall that we have 4 different domains and not one single domain so the 4 parameters we have worked with thus far are really 4*4 parameters. Let us substitute the $n3d$ parameter with `kh`, `khu` and `khv` and let us count the number of times each of the eight neighbour columns from current center and from u and v placement have a shorter column length than at the current point and let us do it on a per-domain basis. These numbers express the number of remainder loops in the subroutine and also their trip-counts so we introduce parameters that will tell how many branches the code will take in the given trip round. In total we now have (4-1)*4+3*4+3*8*4=120 parameters in-

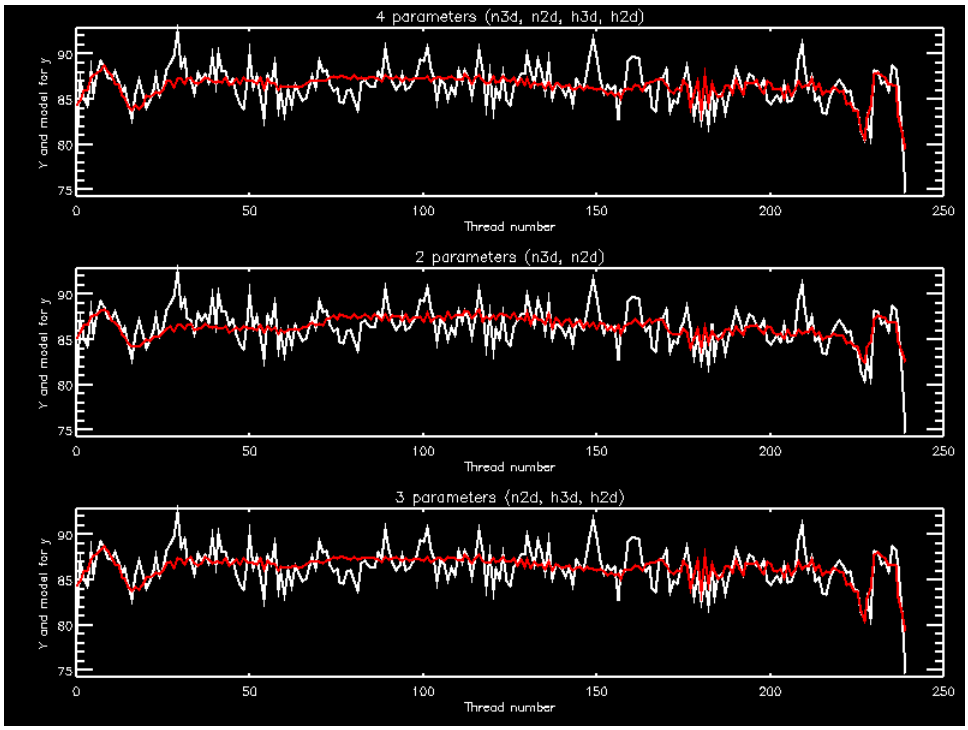


Figure 37: Regression analysis using the same 4 parameters as we used in the first round, namely $iw_3(nt)$, $iw_2(nt)$, $h_3(nt)$ and $h_2(nt)$. Upper figure shows results of using all four, middle figure is with $iw_3(nt)$ and $iw_2(nt)$ only, and bottom figure is with the three $iw_3(nt)$, $iw_2(nt)$ and $h_2(nt)$. The Pearson correlation coefficient is 0.56710102, 0.49547223 and 0.56668785, respectively. Hence, the poor regression.

stead of the 4 used in the last section. If the 120 parameters cannot describe our load then we could try to incorporate hardware and/or runtime environment parameters too. Luckily, we see that the new Pearson correlation coefficient become 0.85597656 and as shown in figure 38 we now also have a much better regression. However, it is important to stress that none of the 120 parameters are very significant and trying to reduce the numbers also reduced the correlation. This implies that we cannot use the regression to improve the balance but our focus at this point of time was merely to describe approaches and not so much to pose final solutions.

The next step would be to do several profiles with the same thread placement. The aim being to see if the spikes were reproducible and whether or not they would move around. Moreover, we would like to do several profiles using different placements strategies and then do the analysis on a core basis instead of a thread basis. These experiments should allow us to conclude whether the spikes are due to OS jitter or hardware exhaustion. Hopefully, this insight would allow us to improve the balance.

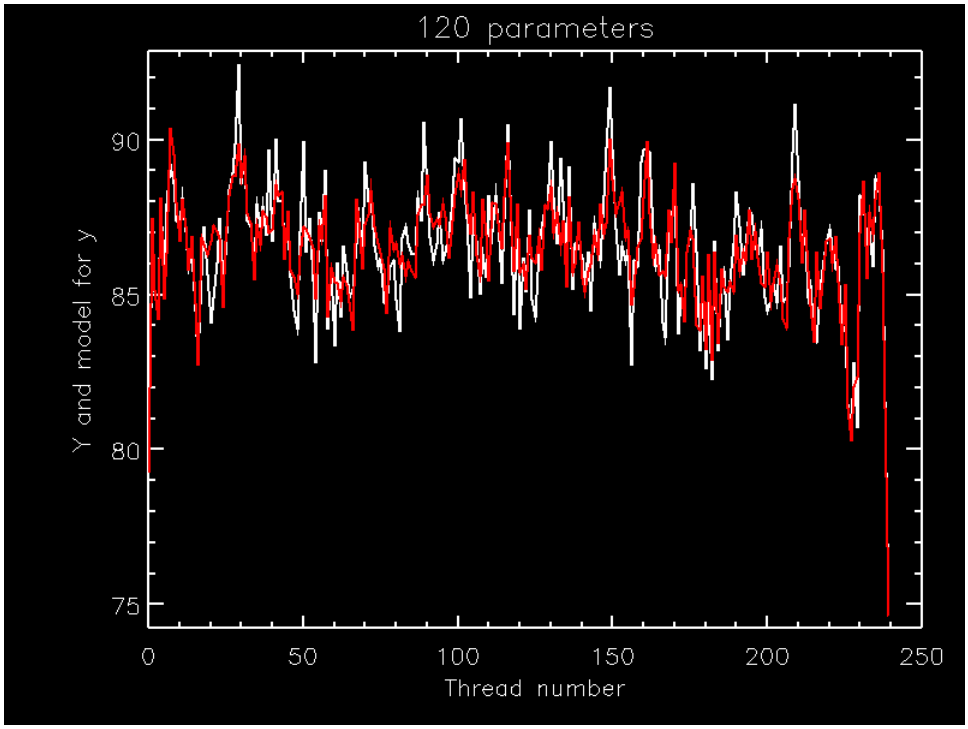


Figure 38: Regression analysis using 120 parameters. Note that $R = 0.85597656$ and the maximal absolute difference between the measurements and the result of the regression is 2.9123180 whereas the mean difference now is 0.86086447.

Based on the findings above we plan to optionally read in (or write out) openMP decompositions (just as we do for MPI). This will allow us to fine tune the decompositions (by hand or by an offline tool) and based on the say 120 parameter regression we will be able to evaluate each decomposition

within having to run the model. This should allow us to get at least a 50% improvement and thus another 3 seconds. Figure 39 shows the result of a simple manual attempt to remove some of the high spikes but again note that we are using far too many parameters to make this useful in practice.

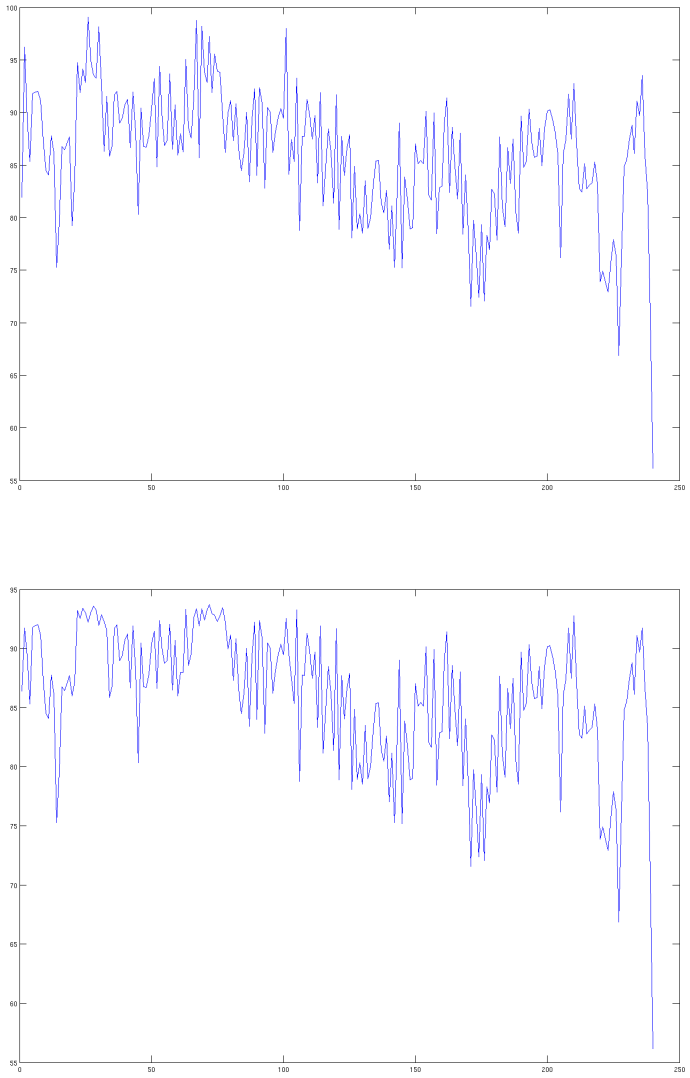


Figure 39: Top: The thread imbalance found using the decomposition for experiment `ex49-ni`. Bottom: The thread imbalance found after a few manual changes to the decomposition. Note that the highest peak has dropped from 99 to 93.

12 Conclusion

This paper was never meant to be more than initial reporting so we will try to confine ourselves and wrap up the findings.

For the use of external tridiagonal solvers we have tried to use the MKL solver and we can summarize our findings on the Xeon X7550 as:

- We were not able to find a Fortran95 interface for the MKL solver.
- We were not able to link statically with the MKL library.
- We were not able to force the MKL solver to get inlined.
- We were not able to measure real improvements when using the MKL solver in the momentum equations.
- We were not able to measure real improvements when using the MKL solver in `tflow` nor in `turbmodel`.

We were not able to measure any real runtime improvements from the vectorization efforts in `momeqs` on the Xeon X7550.

The vectorization improvements in `tflow` and `turbmodels`, on the other hand, gave significant improvements and we should continue this effort throughout the remaining code. The overall improvement of these preliminary initiatives on the Xeon X7550 was 35% which is quite good for scattered efforts conducted during our holiday.

Michael Greenfield from Intel has conducted runs with our code experiments both on an Intel Sandy Bridge system (SNB-EP running at 2.6Ghz, 1600 memory and 64 GB of DDR3 with 16 cores) and on a Xeon PhiT coprocessor as explained in section 9. In summary:

- measured no improvements when using the MKL solver over the simple double-sweep trisolver implementation neither on the Sandy Bridge system nor on the Xeon Phi system in any of the three cases above (`momeqs`, `tflow` and `turbmodel`).
- measured improvements from the vectorization efforts on `tflow` on both the Xeon Phi and the Sandy Bridge system.

- measured improvements from the vectorization efforts in `momeqs` but only on the Xeon Phi system, not on the Sandy Bridge system.
- measured improvements from the vectorization efforts in `turbmodels` on the Xeon Phi system.
- measured improvements from the new decomposition heuristic on the Xeon Phi system.
- measured improvements from the efforts on parallelising the serial components (`mom_c_f`, `rand_z`, `mom_f_c`, `w_c_f`, `bndstz`) on the Xeon Phi system.

The findings on the MKL solver coincide with our findings on the Xeon 7550. The size of our equation systems could be too small to benefit from the MKL BABE implementation or the call-overhead may exceed the potential gain. It would be nice if we would be able to inline it in the future but this is very unlikely, unfortunately.

The findings on `momeqs` on the Sandy Bridge system coincide with our findings on the Xeon X7550. It is interesting and encouraging to note that the `momeqs` rewrites did indeed improve the performance on the Xeon Phi system.

On top of running with the rewrites outlined in section 9 Michael Greenfield has also performed investigations on the choices of compiler flags. The best timings obtained on the three systems are summarized in table 21 and the corresponding compiler flags as well as the flags used for the initial timings of the 2.7 release are shown in table 22.

On the Xeon Phi, we have extremely good speedup with a parallel portion in the 99.5%-99.75% range up to the 60 cores, cf. the upper part of figure 40, and then from 60-240 threads the speedup is still good but the parallel portion gets down to something between 99.0% and 99.25%, cf. the lower part of figure 40. Note that both the D1 (size 32kb) and the L2 (size 512kb) is shared among the 4 threads on the core so this tendency coincides with the theory that we could be limited by cache latency to some extend.

The pointer rewrites mentioned in section 10.5 came with a cost that amounts to almost 5% in performance for the pure openMP build on our XT5-system but with the default, hybrid openMP+MPI configuration we gained a few

Xeon X7550			Sandy Bridge			Xeon PhiT		
Threads	Timing	Gain	Threads	Timing	Gain	Threads	Timing	Gain
64	396	35%	16	528	20%	240	586	60%

Table 21: This table summarizes the fastest time for a 6 hour simulation without IO obtained by the various versions of the Intel compiler (version 12.1.0 on the Xeon X7550 and version 13.0, rev177 on Sandy Bridge and Xeon PhiT) across various compiler flags on each of the three system that we used for thread scaling studies. The flags used is shown in table 22. On the Xeon Phi the timings with hourly IO is 612 seconds and the main reason for this added overhead is that we have an expensive serial chunk that does permutation of all arrays before writing out. We have not tried to improve this since the IO code used here is obsolete. It seems most relevant to compare the Sandy Bridge performance with that of the Xeon PhiT and we note that the difference between the two is approximately 10% which we can explain by the balance issues which have a more significant impact on the performance on the Xeon Phi. Thus, we also expect that this difference will vanish when we have done a bit more tuning of the code.

Platform	**Reference TUNE flags**	**Fastest TUNE flags**
Xeon X7550	`-O3 -fno-alias`	`-O3 -fno-alias`
Sandy Bridge	`-O3`	`-O3 -fno-alias -ipo`
Xeon PhiT	`-O3 -fno-alias -ipo -traceback -fimf-precision=low -fimf-domain-exclusion=15 -opt-assume-safe-padding -opt-streaming-stores always -opt-streaming-cache-evict=0`	`-O3 -fno-alias -ipo -traceback -fimf-precision=low -fimf-domain-exclusion=15 -opt-assume-safe-padding -opt-streaming-stores always -opt-streaming-cache-evict=0`

Table 22: List of compiler flags chosen for the 2.7 release builds versus the fastest builds obtained with the code rewrites described in section 9.

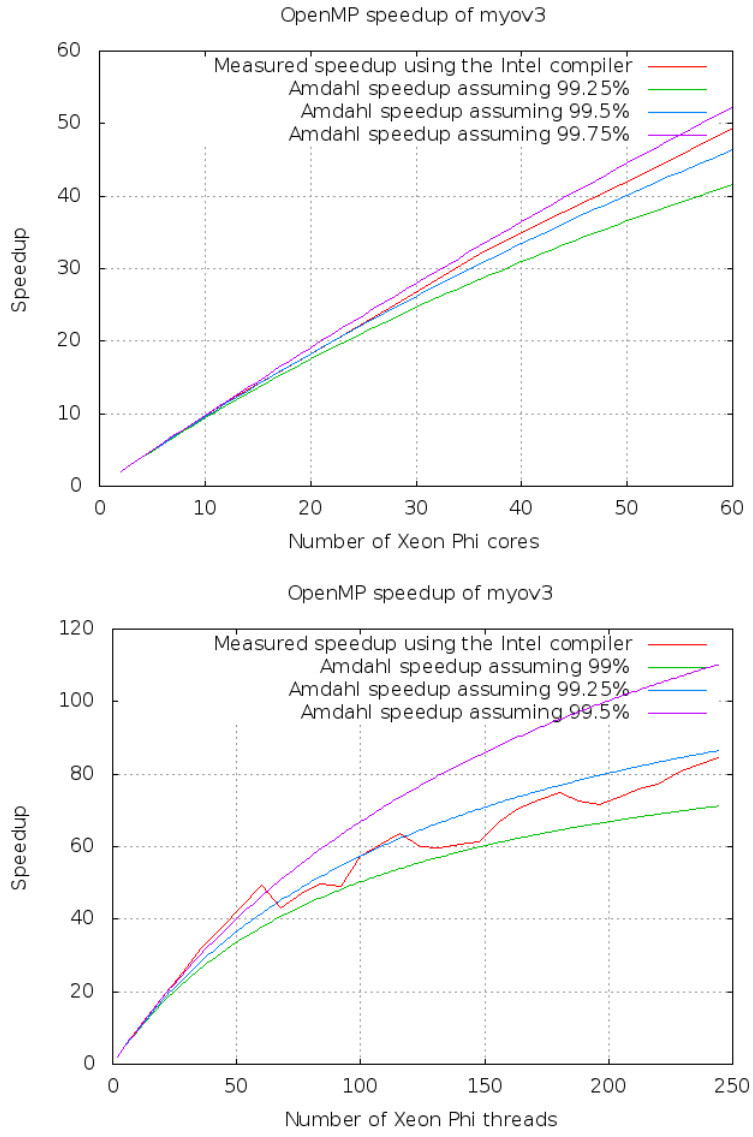


Figure 40: Attained speedup on the Xeon Phi with modifications to the code. Top: Using 1 to 60 threads. Bottom: Using 1 to 240 threads.

percent in performance.

Finally, we have developed a new method for generating the decomposition in a way that takes the actual measured load on each thread into account. With this, we have shown that we can improve the load balancing on the many-core Xeon Phi system using 240 threads to gain 7% in performance and we explained how we can optimize the load balancing even more and potentially gain yet another few percent. The method is, however, general in the sense that it is beneficial to apply it for lower thread counts too, and it can be adapted for NUMA and other asymmetric architectures as well. Furthermore, it can be applied for optimization of our MPI domain decomposition and especially our mixed openMP+MPI decomposition. We will explore these opportunities in the future.

A Appendix: Build instructions

The model uses `autoconf` and the corresponding configure process is meant to set mandatory compiler settings and to distinguish between different build incarnations. If one does not specify any configure options then one will get a serial build. Note that configure generates a file with the compiler flag settings called `Makefile.include`. In case one wishes to change compiler flags one can either redo the configure or adjust the generated `Makefile.include` file or one can pass the new flags onto make itself like make FCFLAGS="-O3". In figure 41 we show some build examples in `BASH`-syntax. All the build examples will generate a single binary called `cmod`.

```
tar -zxvf hbm-2.7.tar.gz
cd hbm-2.7

serial binary
FCFLAGS='-O1' FC=ftn ./configure && make -j <twice_nr_cores_on_build_host>

openmp binary
FCFLAGS='-O1' FC=ftn ./configure --enable-openmp && make -j <twice_nr_cores_on_build_host>

mpi binary
FCFLAGS='-O1' FC=ftn ./configure --enable-mpi && make -j <twice_nr_cores_on_build_host>

openmp+mpi binary
FCFLAGS='-O1' FC=ftn ./configure --enable-openmp --enable-mpi && make -j <twice_nr_cores_on_build_host>

openmp+openacc binary
FCFLAGS='-Minfo=accel -fast' FC=ftn ./configure --enable-openmp --enable-openacc && make -j

a silent make
time make -j 64 > logfile_for_build.txt 2>&1
```

Figure 41: Build samples.

```
In case one wishes to use special compiler flags for a subset of the
objects then one could do something like

rm cmod

and then adjust Makefile.manual. For instance assume that we need
to build some files with O1 instead of O2. Then we would need to
add these lines:

$(ROOT)/src/foo.o: FCFLAGS := $(subst -O2,,$(FCFLAGS) -O1)
$(ROOT)/src/bar.o: FCFLAGS := $(subst -O2,,$(FCFLAGS) -O3)
```

Figure 42: Building with file based compiler flags.

B Appendix: Run instructions

It should be noted that the model uses `big-endian` input files so for those compilers that do not have compiler switches for this (and thus does not allow the configure process to handle it) one will have to specify this as run-time environment. One will also need to specify `ulimit -s unlimited` in `BASH`-syntax or `limit stacksize unlimited` in `CSH`-syntax. One may also need to adjust environment variables pertaining to the system and chosen configure options such as e.g. `OMP_NUM_THREADS`. An example is shown in figure 43.

```
tar -zxvf testcase.tar.gz
cd testcase
ulimit -s unlimited
export OMP_NUM_THREADS=64
<builddir>/cmod
grep -A57 'Validation prints:' logfile.000
grep -i 'took' logfile.000
md5sum restart tempdat.0* sponge_[ts]
```

Figure 43: Run instructions.

There might also be compiler specific environment settings needed, e.g. when using `aprun` placement one have to set `PSC_OMP_AFFINITY` for pathscale binaries to false.

Please inspect the logfile `logfile.000` upon completion. While running it one can inspect the file `zeitschritt` that keeps track of the progress within the timeloop. The timings can be found by grepping after took as shown in figure 43. Moreover, one can check that results are consistent with previous results by grepping for statistics in the logfile, cf. figure 43 and by inspecting the binary output files. `md5sum` is not supposed to change when neither compiler nor compiler flags have changes.

C Appendix: The work balance with 32 threads

The aim of this appendix is to document the work balance emerging from `dmi_omp.F90` when using 32 threads providing us with background information when trying to explain the balance issue revealed in the profiles.

For each of the four nested domains, tables 23, 24, 25 and 26 show the subparts that each of the 32 threads is handling together with the associated halo-sizes. This is intended as background information when studying the related profiles.

The figures 44, 45, 46 and 47 show the actual decomposition into the 32 thread for each of the four nested domains. The obtained `tflow` cost classes are also dispalyed in the figures.

D Appendix: Column-wise view on computational work.

Figure 48-51 show histograms of the distribution of column lengths in each nested domain.

T	surface	iw2	iw3	halo2d	halo3d
1	[1;552]	551	14977	61	1522
2	[553;1204]	651	14976	147	3536
3	[1205;1772]	567	14983	194	4870
4	[1773;2292]	519	14987	208	6029
5	[2293;2769]	476	14994	223	7046
6	[2770;3233]	463	14978	244	7730
7	[3234;3703]	469	14978	250	7976
8	[3704;4183]	479	14983	262	8326
9	[4184;4656]	472	15004	274	8676
10	[4657;5131]	474	14971	286	8902
11	[5132;5635]	503	15001	294	9117
12	[5636;6123]	487	14984	298	8883
13	[6124;6700]	576	14988	309	8531
14	[6701;7252]	551	14999	324	8339
15	[7253;7857]	604	14984	333	8340
16	[7858;8426]	568	14978	334	8281
17	[8427;9029]	602	14998	333	8301
18	[9030;9661]	631	14984	329	7855
19	[9662;10312]	650	14990	324	7369
20	[10313;10972]	659	14995	319	7061
21	[10973;11637]	664	14980	312	6942
22	[11638;12309]	671	14995	300	7051
23	[12310;12906]	596	14988	285	6911
24	[12907;13503]	596	14979	263	6594
25	[13504;14104]	600	14986	256	6300
26	[14105;14746]	641	14975	242	5691
27	[14747;15398]	651	14988	206	4858
28	[15399;16078]	679	14989	192	4068
29	[16079;16954]	875	15016	181	3219
30	[16955;17937]	982	14976	118	2331
31	[17938;18396]	458	14983	60	2000
32	[18397;18908]	511	14494	31	1057

Table 23: openMP decomposition of NS subdomain using 32 threads.

T	surface	iw2	iw3	halo2d	halo3d
1	[1;3372]	3371	49535	102	1808
2	[3373;7709]	4336	49532	339	4284
3	[7710;11889]	4179	49533	511	6638
4	[11890;14989]	3099	49527	632	9418
5	[14990;18457]	3467	49532	745	10766
6	[18458;21868]	3410	49534	743	10183
7	[21869;25394]	3525	49528	686	9695
8	[25395;28180]	2785	49549	653	11476
9	[28181;30352]	2171	49528	642	13859
10	[30353;32430]	2077	49577	618	14833
11	[32431;34467]	2036	49530	596	14610
12	[34468;36520]	2052	49532	582	13989
13	[36521;38775]	2254	49566	579	13090
14	[38776;40892]	2116	49541	506	11955
15	[40893;43124]	2231	49562	456	10619
16	[43125;45481]	2356	49544	468	9557
17	[45482;48109]	2627	49525	455	8409
18	[48110;51147]	3037	49532	447	7267
19	[51148;54417]	3269	49529	428	6327
20	[54418;57713]	3295	49535	332	5436
21	[57714;59756]	2042	49559	215	5462
22	[59757;61367]	1610	49536	222	6440
23	[61368;63093]	1725	49537	266	7588
24	[63094;64786]	1692	49536	289	8420
25	[64787;66779]	1992	49530	333	8509
26	[66780;68949]	2169	49530	362	8271
27	[68950;71381]	2431	49553	418	8971
28	[71382;73705]	2323	49554	505	10836
29	[73706;75847]	2141	49529	537	12754
30	[75848;77645]	1797	49563	508	13692
31	[77646;79364]	1718	49582	472	13899
32	[79365;80885]	1520	49022	233	6845

Table 24: openMP decomposition of IDW using 32 threads.

T	surface	iw2	iw3	halo2d	halo3d
1	[1;196]	195	3249	141	2166
2	[197;422]	225	3247	275	4215
3	[423;622]	199	3247	275	4165
4	[623;851]	228	3251	276	4092
5	[852;1061]	209	3239	274	3951
6	[1062;1291]	229	3245	274	3902
7	[1292;1531]	239	3247	273	3829
8	[1532;1750]	218	3237	272	3759
9	[1751;1987]	236	3241	273	3712
10	[1988;2233]	245	3236	272	3641
11	[2234;2481]	247	3246	269	3607
12	[2482;2724]	242	3246	266	3561
13	[2725;2953]	228	3234	265	3403
14	[2954;3217]	263	3242	267	3240
15	[3218;3493]	275	3232	270	3120
16	[3494;3804]	310	3233	278	3075
17	[3805;4102]	297	3233	282	3020
18	[4103;4393]	290	3233	273	2997
19	[4394;4681]	287	3236	267	2938
20	[4682;4969]	287	3239	264	2866
21	[4970;5272]	302	3237	252	2721
22	[5273;5572]	299	3239	243	2607
23	[5573;5874]	301	3239	240	2508
24	[5875;6195]	320	3242	243	2427
25	[6196;6531]	335	3233	244	2290
26	[6532;6884]	352	3233	243	2157
27	[6885;7266]	381	3236	240	2020
28	[7267;7656]	389	3243	239	1860
29	[7657;8126]	469	3232	239	1627
30	[8127;8713]	586	3232	239	1284
31	[8714;9522]	808	3235	246	869
32	[9523;11581]	2058	3027	126	325

Table 25: openMP decomposition of WS subdomain using 32 threads.

T	surface	iw2	iw3	halo2d	halo3d
1	[1;4327]	4326	191032	120	6959
2	[4328;8085]	3757	191060	238	11395
3	[8086;13572]	5486	191035	401	15949
4	[13573;17576]	4003	191085	629	30394
5	[17577;20840]	3263	191057	703	41042
6	[20841;23972]	3131	191024	739	43333
7	[23973;27241]	3268	191022	748	43134
8	[27242;30407]	3165	191038	762	45520
9	[30408;33641]	3233	191045	848	51534
10	[33642;36470]	2828	191041	941	64219
11	[36471;38979]	2508	191088	1007	75435
12	[38980;41446]	2466	191036	1045	78536
13	[41447;44008]	2561	191065	1076	79468
14	[44009;46559]	2550	191084	1093	82245
15	[46560;48940]	2380	191032	1087	85296
16	[48941;51304]	2363	191029	1076	85935
17	[51305;53629]	2324	191037	1060	86198
18	[53630;55980]	2350	191022	1042	85196
19	[55981;58315]	2334	191061	1033	83169
20	[58316;60771]	2455	191100	1027	80368
21	[60772;63270]	2498	191029	1030	76478
22	[63271;66059]	2788	191113	1042	71055
23	[66060;69211]	3151	191112	1049	62637
24	[69212;73248]	4036	191034	1054	49359
25	[73249;79027]	5778	191031	961	34251
26	[79028;84575]	5547	191029	723	24412
27	[84576;89483]	4907	191089	556	20998
28	[89484;94571]	5087	191089	476	17746
29	[94572;99582]	5010	191068	458	16870
30	[99583;104941]	5358	191046	457	15640
31	[104942;112621]	7679	191065	246	7699
32	[112622;119206]	6584	190019	37	1308

Table 26: openMP decomposition of BS subdomain using 32 threads.

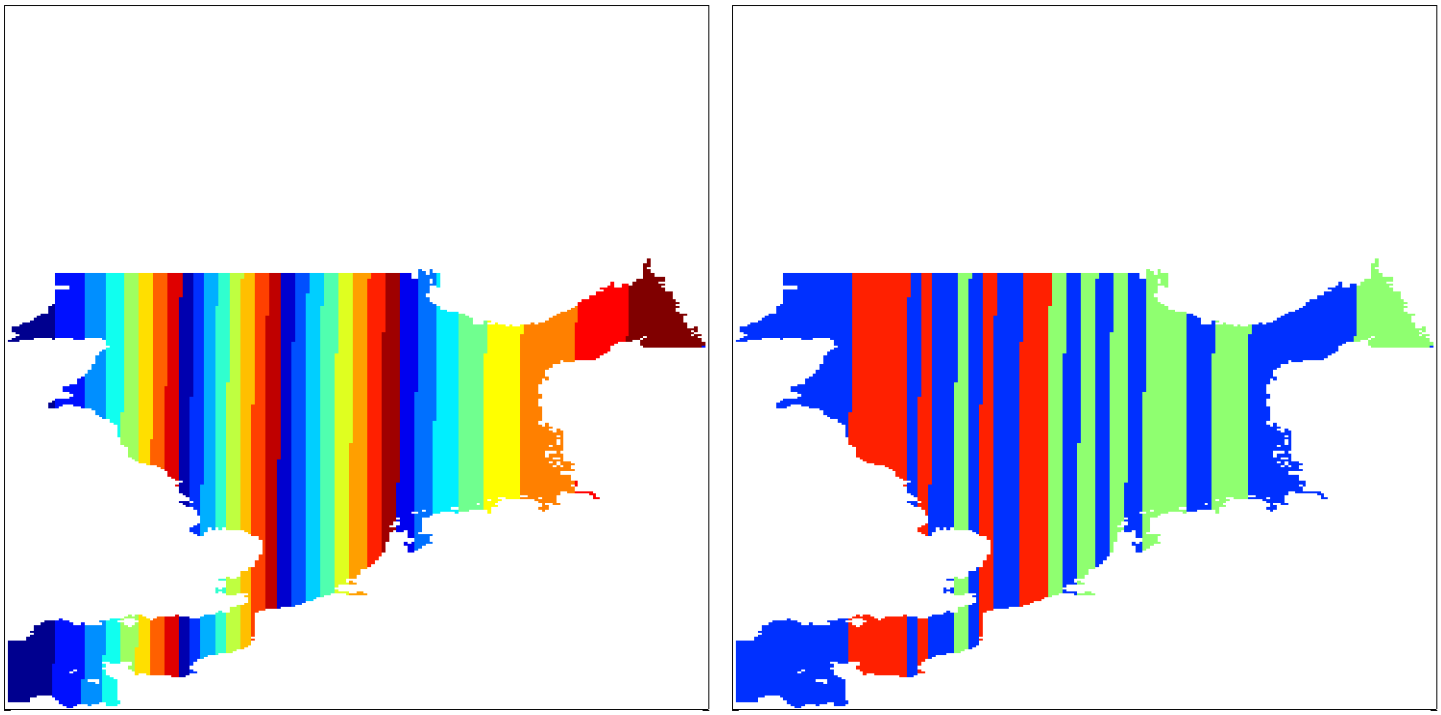


Figure 44: Illustration of the 32 threads openMP decomposition for the North Sea.
Right: Each thread's domain is displayed using pseudo-colours. Threads are numbered from 1 to 32 in left to right order with thread No. 1 handling the deep blue domain to the far left and thread No. 32 the brown domain to the far right.
Left: Each thread is coloured according to its cost class for the `tflow` module. Threads in the most expensive class are red, mid-expensive are green, and the least expensive threads are blue. Configure option is `--enable-openmp`.

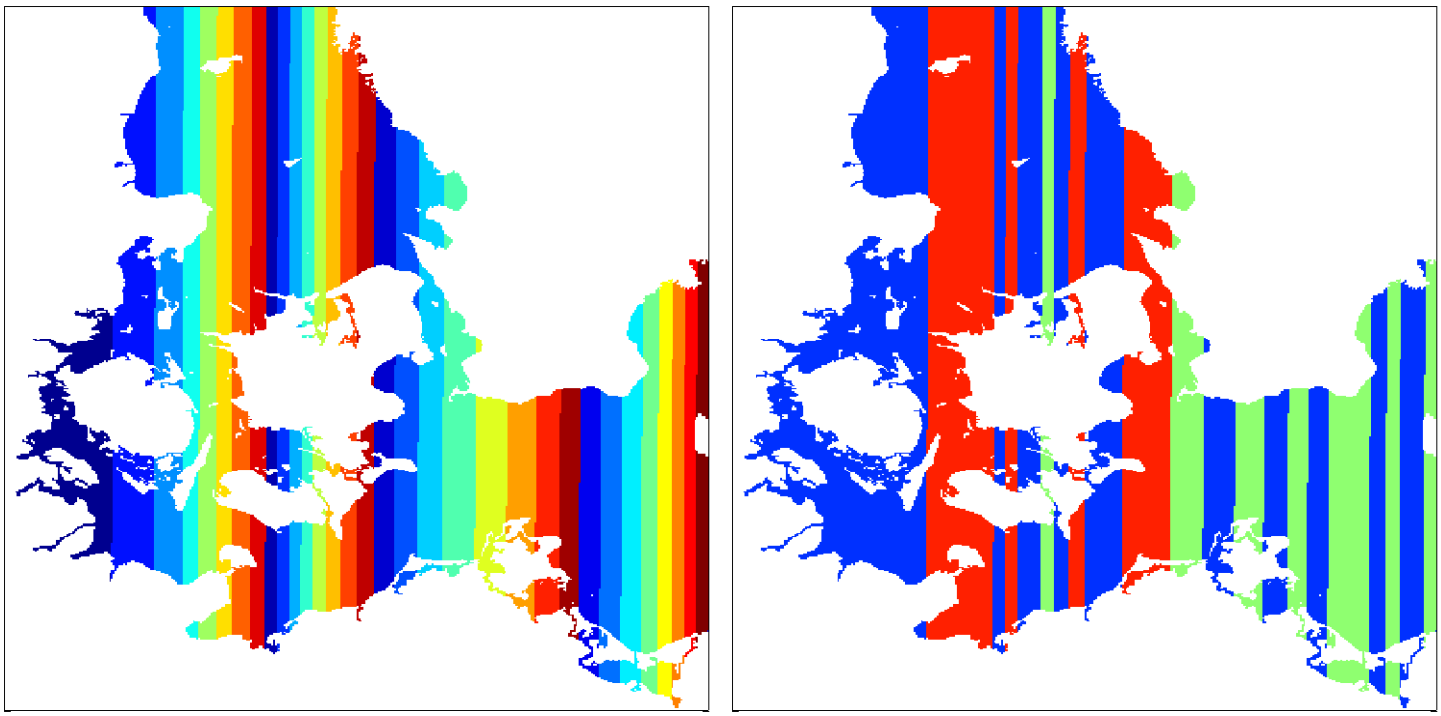


Figure 45: Illustration of the 32 threads openMP decomposition for the Inner Danish Waters.
Right: Each thread's domain is displayed using pseudo-colours. Threads are numbered from 1 to 32 in left to right order with thread No. 1 handling the deep blue domain to the far left and thread No. 32 the brown domain to the far right.
Left: Each thread is coloured according to its cost class for the `tflow` module. Threads in the most expensive class are red, mid-expensive are green, and the least expensive threads are blue. Configure option is `--enable-openmp`.

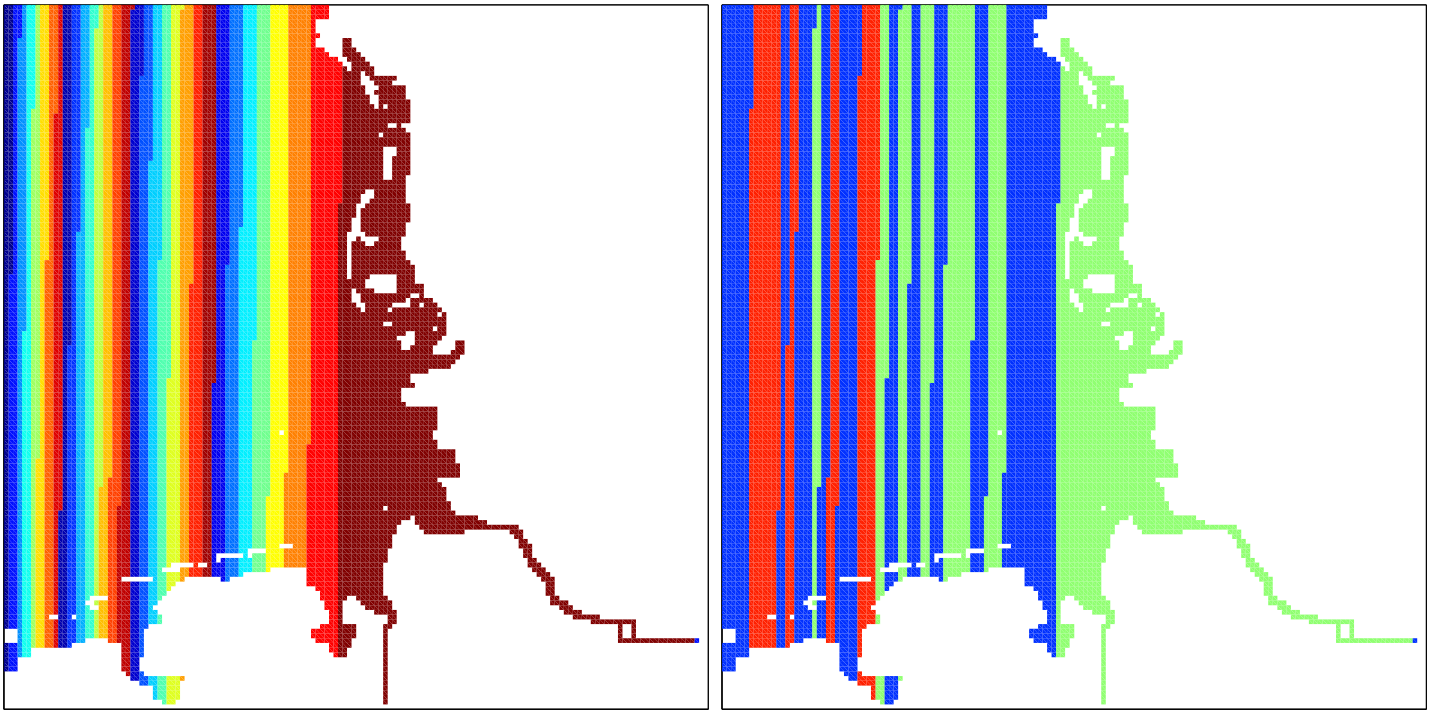


Figure 46: Illustration of the 32 threads openMP decomposition for the Wadden Sea.
Right: Each thread's domain is displayed using pseudo-colours. Threads are numbered from 1 to 32 in left to right order with thread No. 1 handling the deep blue domain to the far left and thread No. 32 the brown domain to the far right.
Left: Each thread is coloured according to its cost class for the `tflow` module. Threads in the most expensive class are red, mid-expensive are green, and the least expensive threads are blue. Configure option is `--enable-openmp`.

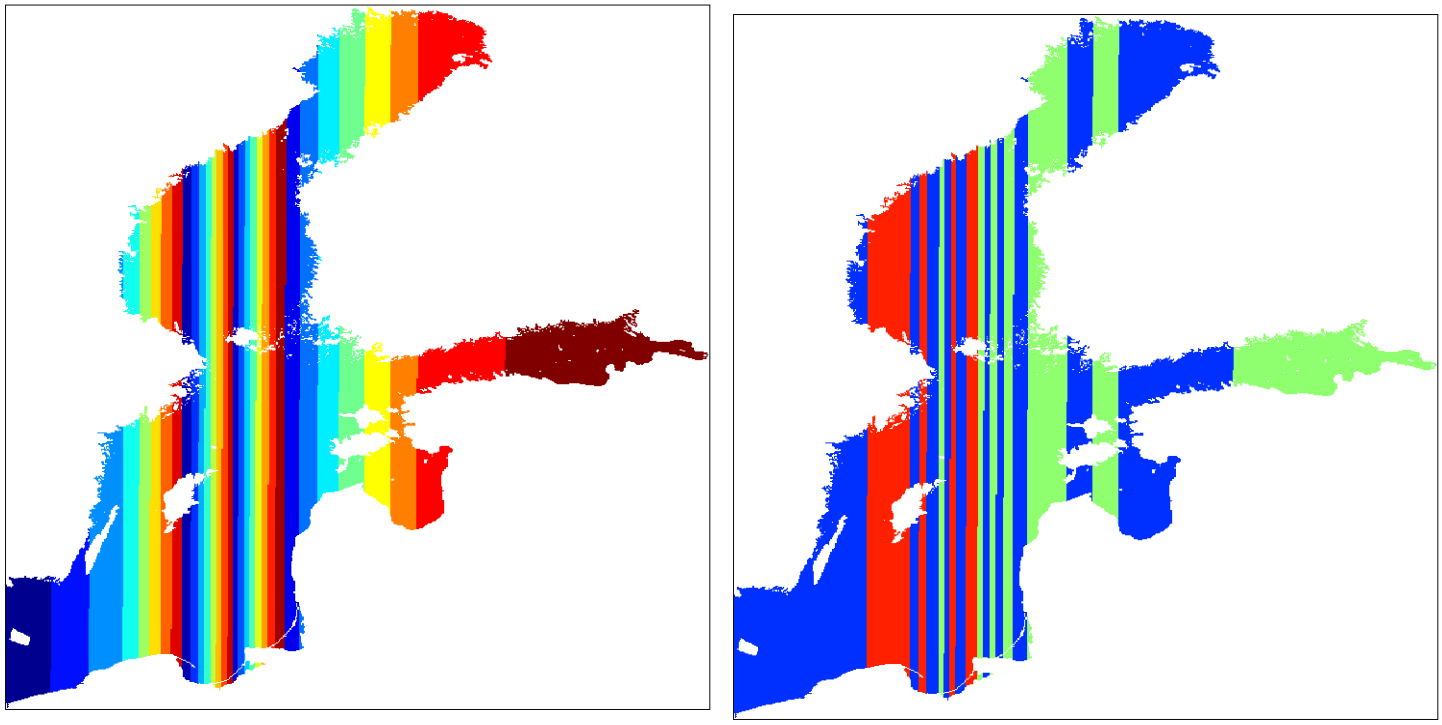


Figure 47: Illustration of the 32 threads openMP decomposition for the Baltic Sea.
Right: Each thread's domain is displayed using pseudo-colours. Threads are numbered from 1 to 32 in left to right order with thread No. 1 handling the deep blue domain to the far left and thread No. 32 the brown domain to the far right.
Left: Each thread is coloured according to its cost class for the `tflow` module. Threads in the most expensive class are red, mid-expensive are green, and the least expensive threads are blue. Configure option is `--enable-openmp`.

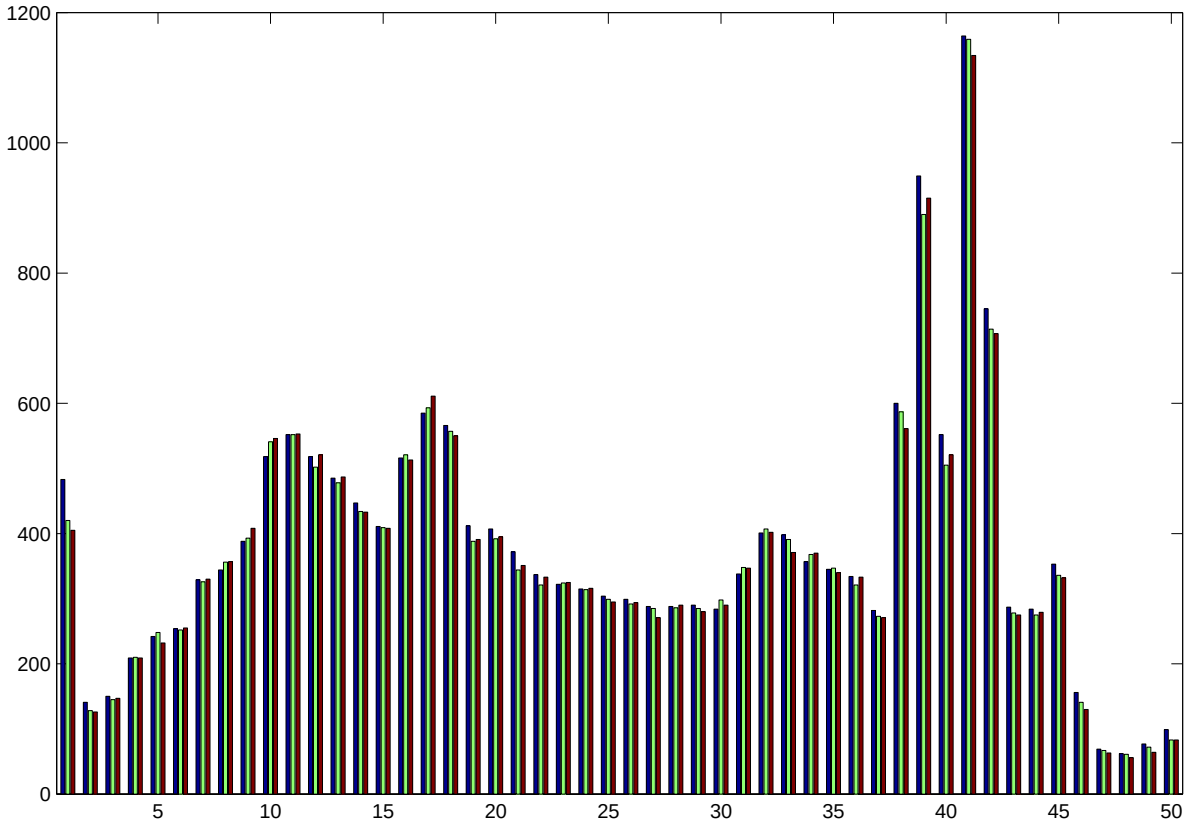


Figure 48: Illustration of the distribution of column lengths $[1:50]$ for the North Sea domain. Blue bars are the lengths of scalar equations. Green and brown are the $u-$ and $v-$ momentum equations.

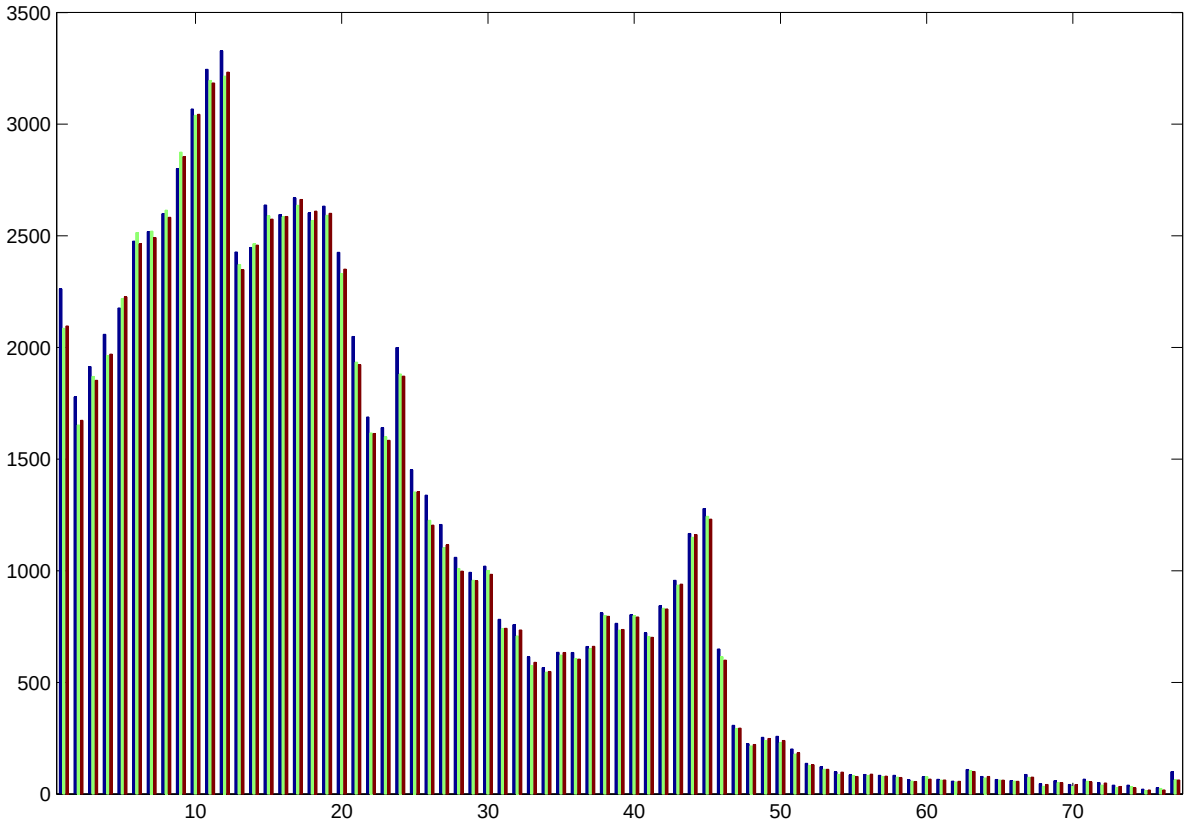


Figure 49: Illustration of the distribution of column lengths $[1:77]$ for the Inner Danish Water domain. Blue bars are the lengths of scalar equations. Green and brown are the $u-$ and $v-$ momentum equations.

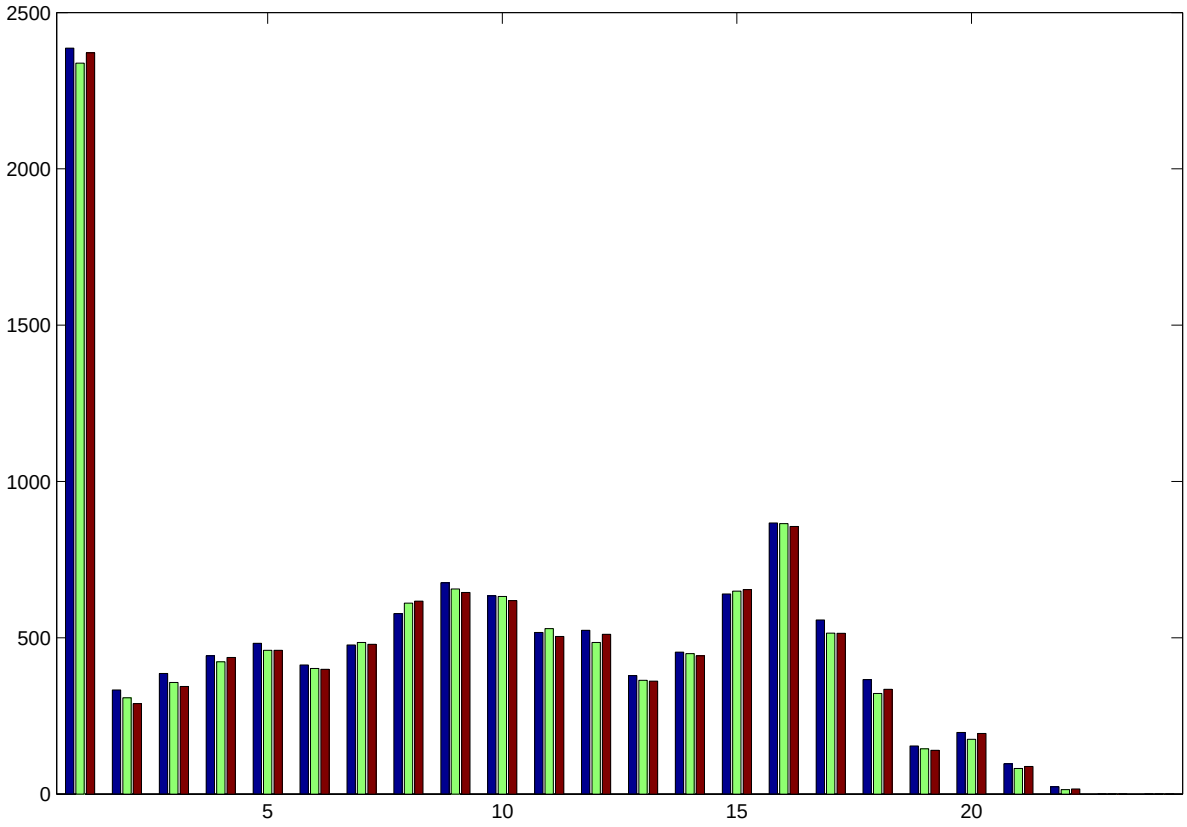


Figure 50: Illustration of the distribution of column lengths $[1:24]$ for the Wadden Sea domain. Blue bars are the lengths of scalar equations. Green and brown are the $u-$ and $v-$ momentum equations.

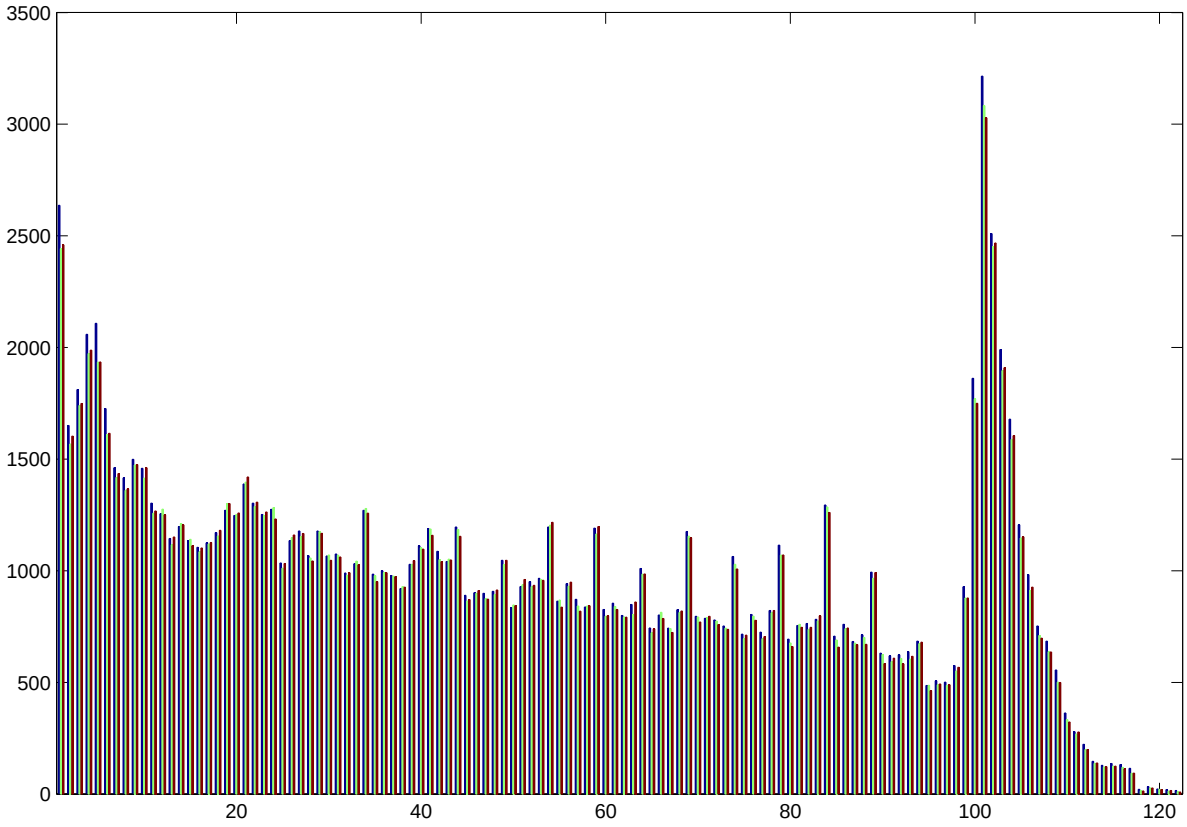


Figure 51: Illustration of the distribution of column lengths $[1:122]$ for the Baltic Sea domain. Blue bars are the lengths of scalar equations. Green and brown are the $u-$ and $v-$ momentum equations.

E Appendix: The work balance with 240 threads

The aim of this appendix is to document the work balance emerging from `dmi_omp.F90` when using 240 threads providing us with background information when trying to explain the balance issue revealed in the profiles.

Larry Meadows from Intel kindly profiled the application for us. Table 27 and table 28 shows the ten threads with the most and the least work, respectively, according to this profile.

Tables 29–34 show top-five statistics with respect to distribution of most and least 2D points, 3D points and halo points for the current decomposition for each nested area as well as for the total setup.

Figures 52–55 show the actual 2D decomposition into the 240 thread for each of the four nested domains.

Figures 56–59 show the halo decompositions (2D and 3D) when the problem is decomposed using 240 threads for each of the four nested domains.

#T	libiomp5.so time
17	10.219385
18	10.585014
19	12.294345
16	12.477155
229	13.162713
228	14.826331
198	15.393062
13	15.84096
14	15.950647
182	16.517377

Table 27: The 10 threads with most work.

#T	libiomp5.so time
240	55.127981
158	32.3309
106	31.234013
161	30.594161
90	30.502751
165	30.484465
157	30.374783
119	30.045714
162	29.332732
98	29.314448

Table 28: The 10 threads with least work.

Area	#T	Interval	2D	avg 2D	3D	avg 3D	2D halo	3D halo
BS	229	[108156;109624]	1468	496	25473	25469	556	5222
BS	239	[116816;118139]	1323	496	25487	25469	192	2028
BS	230	[109625;110851]	1226	496	25484	25469	354	3243
BS	228	[107060;108155]	1095	496	25483	25469	756	8516
BS	240	[118140;119206]	1066	496	16691	25469	62	641
ID	17	[8357;9046]	689	337	6612	6603	1020	4871
ID	12	[5246;5898]	652	337	6605	6603	802	3942
ID	13	[5899;6546]	647	337	6609	6603	874	4306
ID	16	[7719;8356]	637	337	6607	6603	964	4770
ID	18	[9047;9668]	621	337	6615	6603	1044	5178
NS	219	[17214;17378]	164	78	1997	1996	330	2082
NS	220	[17379;17535]	156	78	2019	1996	282	1913
NS	218	[17058;17213]	155	78	2015	1996	350	2578
NS	216	[16801;16950]	149	78	2038	1996	348	2789
NS	221	[17536;17668]	132	78	2040	1996	238	1763
WS	235	[10512;10893]	381	48	431	431	372	200
WS	236	[10894;11270]	376	48	431	431	256	135
WS	234	[10196;10511]	315	48	433	431	398	255
WS	237	[11271;11581]	310	48	351	431	86	46
WS	233	[9929;10195]	266	48	431	431	410	304
ALL	229		1884					
ALL	230		1677					
ALL	17		1609					
ALL	18		1576					
ALL	239		1539					

Table 29: The 5 threads with most 2D points for each of the 4 subdomains and the 5 threads with the largest sum of 2D points. Note that 3 of the 5 threads in the ALL top5 is in the top10 profile.

Area	#T	Interval	2D	avg 2D	3D	avg 3D	2D halo	3D halo
BS	126	[52935;53218]	283	496	25515	25469	1140	50714
BS	93	[42220;42505]	285	496	25509	25469	1148	50607
BS	129	[53862;54152]	290	496	25572	25469	1164	50796
BS	134	[55423;55713]	290	496	25532	25469	1166	50727
BS	90	[41202;41492]	290	496	25470	25469	1166	50714
ID	240	[80757;80885]	128	337	3334	6603	262	3350
ID	237	[80206;80371]	165	337	6619	6603	664	13244
ID	239	[80572;80756]	184	337	6606	6603	546	8694
ID	236	[80018;80205]	187	337	6613	6603	752	12948
ID	168	[61921;62111]	190	337	6610	6603	486	7402
NS	240	[18909;18908]	-1	78	0	1996	0	0
NS	239	[18875;18908]	33	78	270	1996	24	148
NS	85	[5776;5824]	48	78	2008	1996	200	4091
NS	100	[6864;6913]	49	78	2010	1996	204	4086
NS	102	[7023;7072]	49	78	2015	1996	204	4104
WS	238	[11582;11581]	-1	48	0	431	0	0
WS	239	[11582;11581]	-1	48	0	431	0	0
WS	240	[11582;11581]	-1	48	0	431	0	0
WS	11	[286;306]	20	48	441	431	88	915
WS	16	[428;448]	20	48	432	431	88	904
ALL	158		593					
ALL	90		641					
ALL	106		643					
ALL	155		660					
ALL	79		660					

Table 30: The 5 threads with least 2D points for each of the 4 subdomains and the 5 threads with the smallest sum of 2D points. Note that 3 of the threads in the ALL top5 are in the top10 of the threads with least work.

Area	#T	Interval	2D	avg 2D	3D	avg 3D	2D halo	3D halo
BS	129	[53862;54152]	290	496	25572	25469	1164	50796
BS	89	[40819;41201]	382	496	25569	25469	1514	49781
BS	104	[45950;46266]	316	496	25568	25469	1262	50377
BS	151	[60830;61149]	319	496	25567	25469	1284	50809
BS	41	[21870;22258]	388	496	25565	25469	1440	44965
ID	66	[29668;29930]	262	337	6674	6603	1030	13078
ID	230	[78637;78883]	246	337	6671	6603	962	13200
ID	76	[32442;32731]	289	337	6662	6603	1148	12793
ID	232	[79127;79354]	227	337	6654	6603	902	13207
ID	68	[30234;30539]	305	337	6653	6603	1210	12957
NS	230	[18295;18348]	53	78	2042	1996	118	2245
NS	227	[18100;18170]	70	78	2041	1996	136	2113
NS	228	[18171;18235]	64	78	2040	1996	136	2212
NS	221	[17536;17668]	132	78	2040	1996	238	1763
NS	216	[16801;16950]	149	78	2038	1996	348	2789
WS	207	[7584;7643]	59	48	448	431	244	896
WS	1	[1;22]	21	48	448	431	48	490
WS	56	[1622;1663]	41	48	447	431	172	911
WS	47	[1338;1385]	47	48	447	431	194	914
WS	3	[45;69]	24	48	447	431	54	483
ALL	159				34655			
ALL	66				34655			
ALL	89				34650			
ALL	120				34645			
ALL	104				34643			

Table 31: The 5 threads with most 3D points for each of the 4 subdomains and the 5 threads with the largest sum of 3D points. Note that the profile showed that the most expensive threads were: 17, 18, 16, 19, 229, 228, 13, 14, 198, 182 and none of these threads appear in the table above so a decomposition strategy that aims at an even split in the number of 3D points is obviously not good enough at these thread counts.

Area	#T	Interval	2D	avg 2D	3D	avg 3D	2D halo	3D halo
BS	240	[118140;119206]	1066	496	16691	25469	62	641
BS	112	[48515;48840]	325	496	25469	25469	1304	50291
BS	119	[50734;51039]	305	496	25469	25469	1222	50065
BS	17	[8894;9720]	826	496	25469	25469	682	10237
BS	72	[35071;35453]	382	496	25469	25469	1504	49309
ID	240	[80757;80885]	128	337	3334	6603	262	3350
ID	117	[44301;44567]	266	337	6603	6603	892	9762
ID	133	[50009;50406]	397	337	6603	6603	888	7174
ID	138	[52104;52520]	416	337	6603	6603	954	7091
ID	146	[55709;56166]	457	337	6603	6603	794	5403
NS	240	[18909;18908]	-1	78	0	1996	0	0
NS	239	[18875;18908]	33	78	270	1996	24	148
NS	155	[11428;11518]	90	78	1996	1996	368	3965
NS	161	[11953;12041]	88	78	1996	1996	352	4012
NS	18	[1367;1440]	73	78	1996	1996	290	3876
WS	238	[11582;11581]	-1	48	0	431	0	0
WS	239	[11582;11581]	-1	48	0	431	0	0
WS	240	[11582;11581]	-1	48	0	431	0	0
WS	237	[11271;11581]	310	48	351	431	86	46
WS	111	[3464;3497]	33	48	431	431	140	872
ALL	240				20025			
ALL	239				32363			
ALL	238				34094			
ALL	237				34472			
ALL	212				34511			

Table 32: The 5 threads with least 3D points for each of the 4 subdomains and the 5 threads with the smallest sum of 3D points.

Area	#T	Interval	2D	avg 2D	3D	avg 3D	2D halo	3D halo
BS	144	[58563;58857]	294	496	25543	25469	1172	50897
BS	98	[43908;44211]	303	496	25537	25469	1216	50896
BS	139	[56987;57280]	293	496	25546	25469	1176	50855
BS	106	[46614;46910]	296	496	25519	25469	1176	50840
BS	138	[56683;56986]	303	496	25509	25469	1210	50839
ID	237	[80206;80371]	165	337	6619	6603	664	13244
ID	232	[79127;79354]	227	337	6654	6603	902	13207
ID	230	[78637;78883]	246	337	6671	6603	962	13200
ID	234	[79584;79801]	217	337	6634	6603	868	13191
ID	235	[79802;80017]	215	337	6608	6603	858	13159
NS	96	[6555;6605]	50	78	2028	1996	208	4127
NS	166	[12374;12470]	96	78	2037	1996	390	4120
NS	104	[7185;7236]	51	78	2023	1996	208	4115
NS	56	[3891;3948]	57	78	2027	1996	236	4113
NS	31	[2312;2367]	55	78	2027	1996	228	4110
WS	24	[644;692]	48	48	443	431	200	924
WS	31	[860;883]	23	48	445	431	100	923
WS	40	[1133;1156]	23	48	445	431	100	922
WS	12	[307;330]	23	48	444	431	100	922
WS	82	[2472;2498]	26	48	446	431	112	920
ALL	79							68503
ALL	98							68383
ALL	90							68338
ALL	82							68214
ALL	85							68212

Table 33: The 5 threads with most 3D halo points for each of the 4 sub-domains and the 5 threads with the largest sum of 3D halo points. Note that none of the top5 threads in ALL are in the top10 of the most expensive threads.

Area	#T	Interval	2D	avg 2D	3D	avg 3D	2D halo	3D halo
BS	240	[118140;119206]	1066	496	16691	25469	62	641
BS	239	[116816;118139]	1323	496	25487	25469	192	2028
BS	233	[112480;113141]	661	496	25478	25469	170	3231
BS	230	[109625;110851]	1226	496	25484	25469	354	3243
BS	238	[115942;116815]	873	496	25476	25469	258	3422
ID	1	[1;500]	499	337	6605	6603	108	613
ID	2	[501;1011]	510	337	6617	6603	236	1421
ID	3	[1012;1490]	478	337	6606	6603	260	1605
ID	4	[1491;1956]	465	337	6607	6603	250	1692
ID	5	[1957;2439]	482	337	6630	6603	332	2284
NS	240	[18909;18908]	-1	78	0	1996	0	0
NS	239	[18875;18908]	33	78	270	1996	24	148
NS	238	[18771;18874]	103	78	1996	1996	76	779
NS	1	[1;64]	63	78	2012	1996	76	1096
NS	237	[18695;18770]	75	78	2008	1996	124	1457
WS	238	[11582;11581]	-1	48	0	431	0	0
WS	239	[11582;11581]	-1	48	0	431	0	0
WS	240	[11582;11581]	-1	48	0	431	0	0
WS	237	[11271;11581]	310	48	351	431	86	46
WS	236	[10894;11270]	376	48	431	431	256	135
ALL	240							3991
ALL	1							5969
ALL	239							10870
ALL	2							12099
ALL	3							13711

Table 34: The 5 threads with least 3D halo points for each of the 4 subdomains and the 5 threads with the smallest sum of 3D halo points.

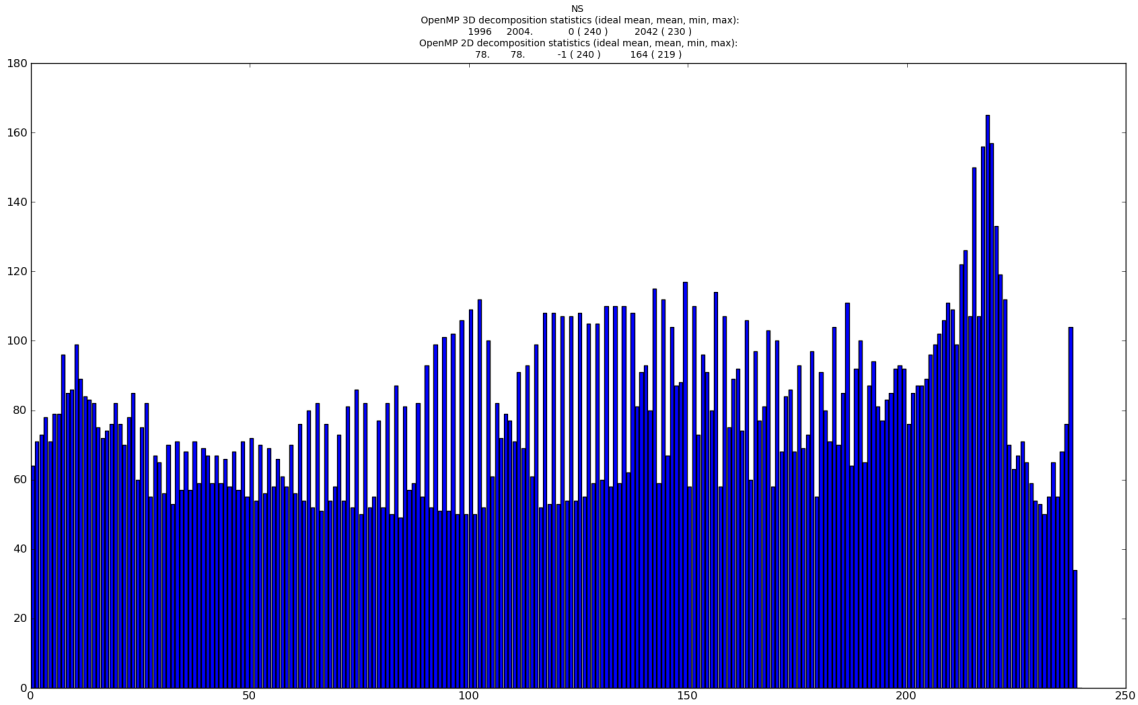


Figure 52: Illustration of the 240 threads 2D openMP decomposition for the North Sea. Configure option is `--enable-openmp`.

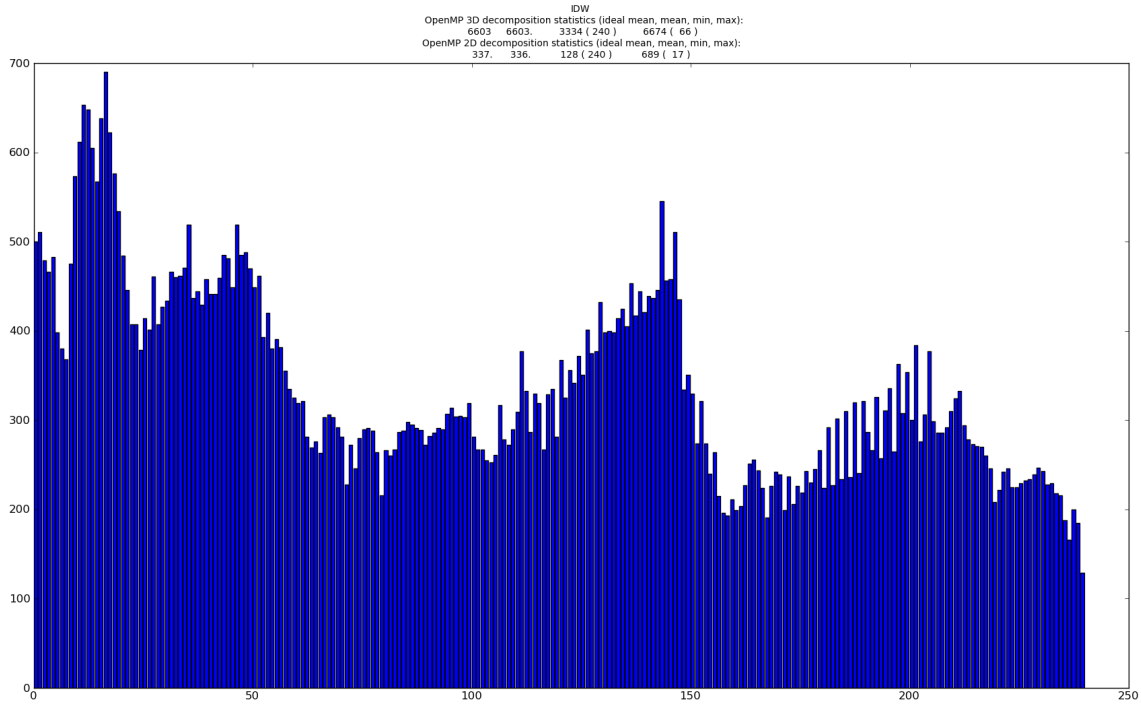


Figure 53: Illustration of the 240 threads 2D openMP decomposition for the Inner Danish Water. Configure option is `--enable-openmp`.

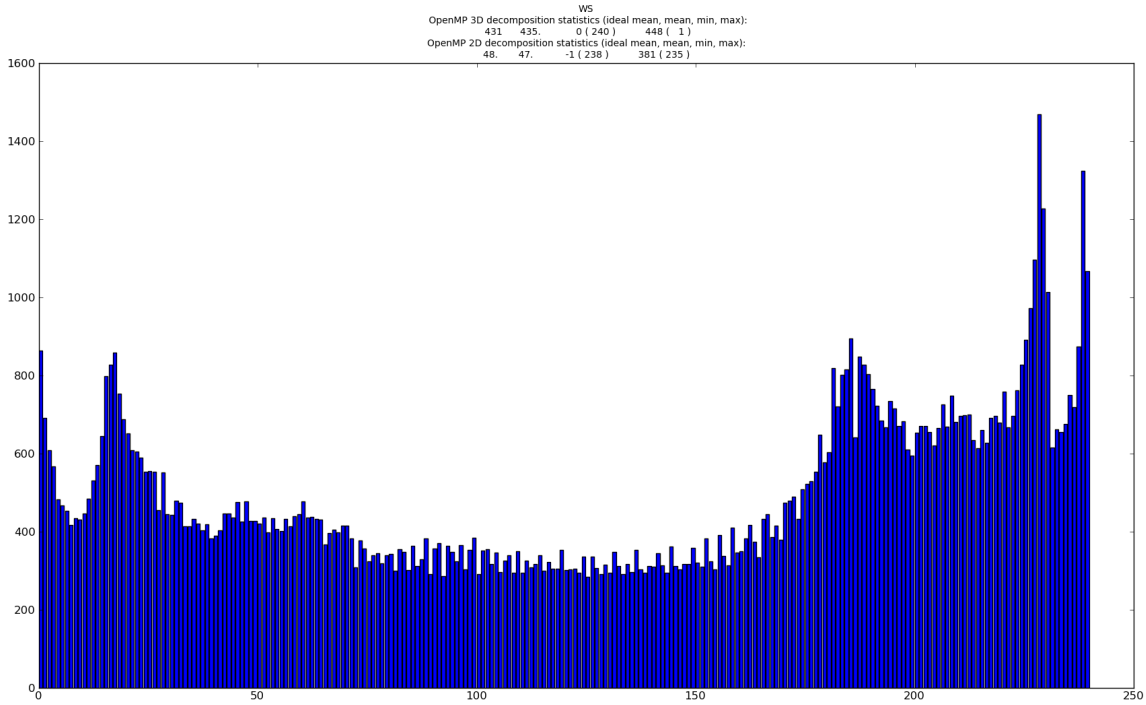


Figure 54: Illustration of the 240 threads 2D openMP decomposition for the Wadden Sea. Configure option is `--enable-openmp`.

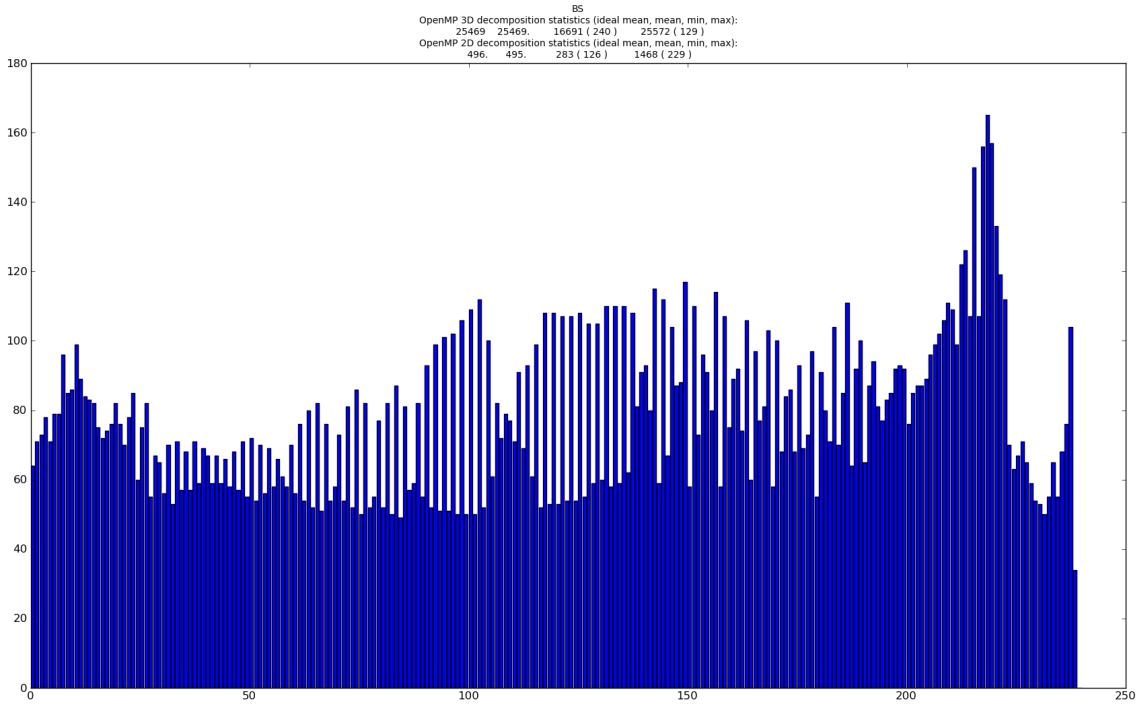


Figure 55: Illustration of the 240 threads 2D openMP decomposition for the Baltic Sea. Configure option is `--enable-openmp`.

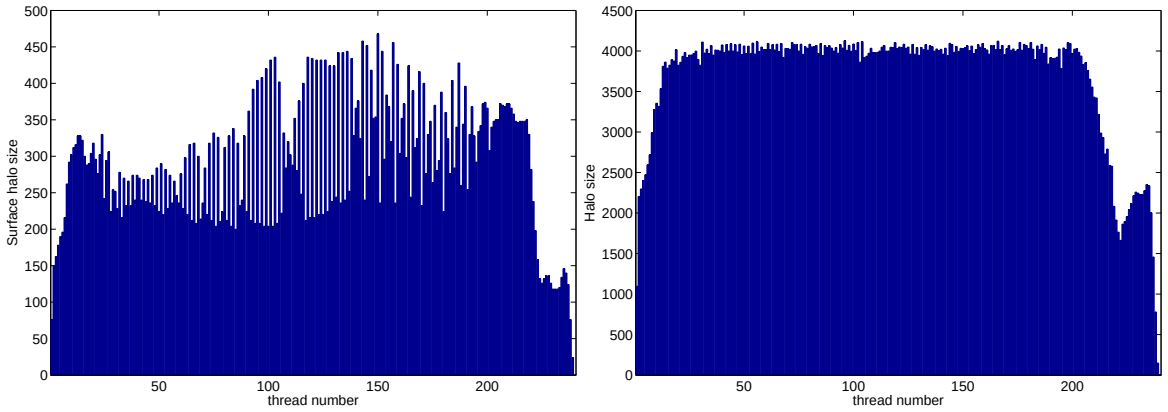


Figure 56: Illustration of the 240 threads halo decomposition for the North Sea. Configure option is `--enable-openmp`. This size of the invisible halo can regarded as the size of potential cacheline fight that the thread may have with other threads.
Right: The size of the 3D halo.
Left: The size of the 2D halo.

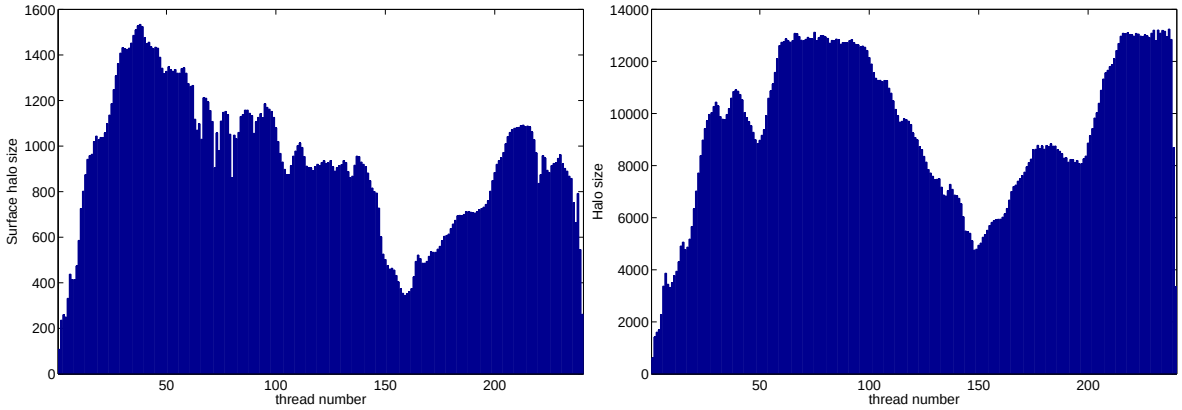


Figure 57: Illustration of the 240 threads halo decomposition for the Inner Danish Water. Configure option is `--enable-openmp`. This size of the invisible halo can regarded as the size of potential cacheline fight that the thread may have with other threads.
Right: The size of the 3D halo.
Left: The size of the 2D halo.

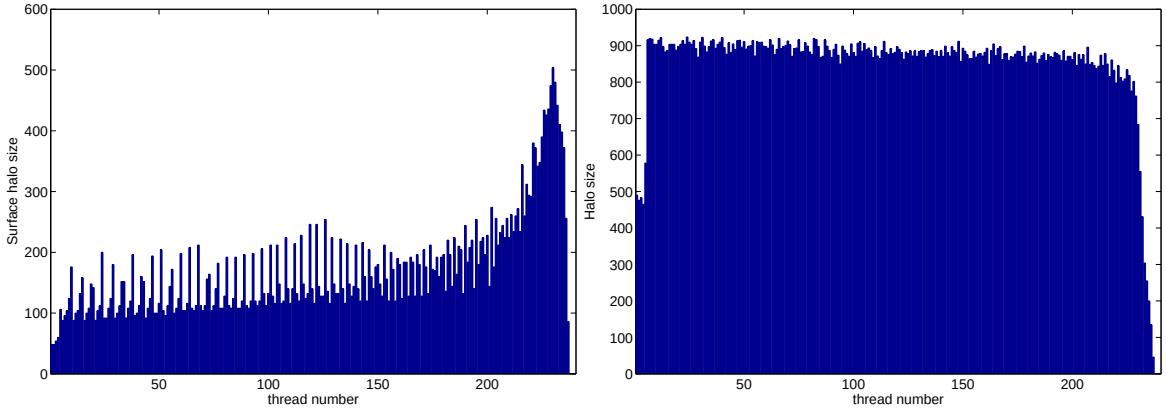


Figure 58: Illustration of the 240 threads halo decomposition for the Wadden Sea. Configure option is `--enable-openmp`. This size of the invisible halo can regarded as the size of potential cacheline fight that the thread may have with other threads.
Right: The size of the 3D halo.
Left: The size of the 2D halo.

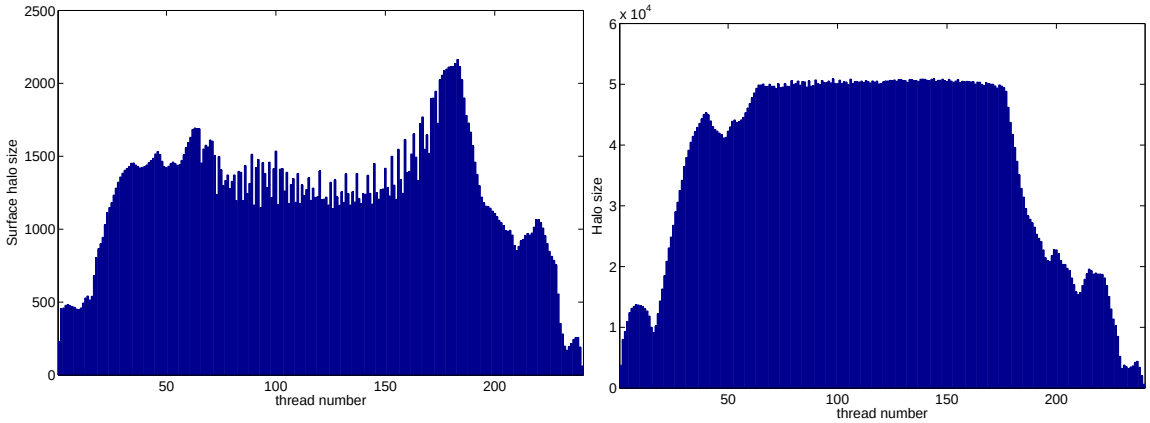


Figure 59: Illustration of the 240 threads halo decomposition for the Baltic Sea. Configure option is `--enable-openmp`. This size of the invisible halo can regarded as the size of potential cacheline fight that the thread may have with other threads.
Right: The size of the 3D halo.
Left: The size of the 2D halo.

F Appendix: The work balance with 240 threads - second round

This appendix contains information on the decomposition used in the runs in section 11.

Area	#T	n2db	n2de	n2h
BS	177	75016	75657	1097
BS	176	74419	75015	1091
BS	178	75658	76440	1079
BS	175	73771	74418	1076
BS	174	73188	73770	1072
IDW	44	17571	17961	780
IDW	45	17962	18346	777
IDW	46	18347	18732	776
IDW	43	17176	17570	773
IDW	47	18733	19115	768
NS	99	7092	7194	207
NS	151	11304	11404	204
NS	136	10037	10137	204
NS	138	10203	10302	203
NS	97	6935	7032	201
WS	230	10813	10881	141
WS	227	10605	10673	141
WS	222	10262	10330	141
WS	224	10399	10466	140
WS	223	10331	10398	140
ALL	65			1796
ALL	46			1779
ALL	61			1774
ALL	45			1773
ALL	67			1770

Table 35: The 5 threads with most 2D halo points for each of the 4 subdomains and the 5 threads with the largest sum of 2D halo points.

Area	#T	n2db	n2de	n3h
BS	134	56987	57302	56363
BS	128	54945	55257	56240
BS	137	58011	58328	56219
BS	131	55969	56281	56202
BS	143	60082	60400	56070
IDW	239	80399	80640	15320
IDW	238	80143	80398	15245
IDW	91	33918	34204	14652
IDW	237	79897	80142	14556
IDW	92	34205	34490	14515
NS	98	7033	7091	4744
NS	102	7354	7411	4744
NS	77	5485	5542	4733
NS	79	5625	5681	4728
NS	100	7195	7252	4721
WS	12	426	459	1407
WS	19	700	733	1395
WS	5	155	188	1389
WS	26	976	1009	1362
WS	33	1253	1287	1360
ALL	90			75545
ALL	102			75295
ALL	81			75258
ALL	96			75226
ALL	87			75124

Table 36: The 5 threads with most 3D halo points for each of the 4 subdomains and the 5 threads with the largest sum of 3D halo points.

Area	#T	n2db	n2de	n2d
BS	240	118116	119206	1091
BS	228	108981	110009	1029
BS	227	107992	108980	989
BS	239	117148	118115	968
BS	229	110010	110896	887
IDW	14	5301	5777	477
IDW	20	8012	8476	465
IDW	21	8477	8939	463
IDW	15	5778	6238	461
IDW	16	6239	6695	457
NS	223	17485	17605	121
NS	221	17265	17384	120
NS	220	17150	17264	115
NS	224	17606	17718	113
NS	216	16742	16846	105
WS	238	11372	11442	71
WS	237	11301	11371	71
WS	239	11443	11512	70
WS	236	11231	11300	70
WS	234	11092	11161	70
ALL	240			1504
ALL	228			1451
ALL	227			1393
ALL	239			1363
ALL	229			1314

Table 37: The 5 threads with most 2D points for each of the 4 subdomains and the 5 threads with the largest sum of 2D points.

Area	#T	n2db	n2de	n3d
BS	131	55969	56281	28046
BS	128	54945	55257	28043
BS	137	58011	58328	28030
BS	134	56987	57302	28017
BS	108	48181	48500	28000
IDW	169	60056	60298	8449
IDW	170	60299	60548	8310
IDW	168	59805	60055	8280
IDW	240	80641	80885	8108
IDW	239	80399	80640	8090
NS	232	18266	18326	2405
NS	233	18327	18388	2381
NS	234	18389	18450	2342
NS	235	18451	18513	2340
NS	98	7033	7091	2336
WS	1	1	34	701
WS	19	700	733	687
WS	12	426	459	683
WS	5	155	188	675
WS	26	976	1009	666
ALL	117			38024
ALL	113			37975
ALL	108			37695
ALL	102			37676
ALL	90			37621

Table 38: The 5 threads with most 3D points for each of the 4 subdomains and the 5 threads with the largest sum of 3D points.

Area	#T	n2db	n2de	n2h
BS	232	112363	112976	135
BS	233	112977	113608	144
BS	234	113609	114254	146
BS	236	114930	115607	151
BS	237	115608	116289	155
IDW	169	60056	60298	190
IDW	168	59805	60055	192
IDW	170	60299	60548	194
IDW	4	1255	1669	195
IDW	167	59539	59804	205
NS	240	18810	18908	39
NS	239	18727	18809	53
NS	232	18266	18326	60
NS	233	18327	18388	61
NS	234	18389	18450	63
WS	239	11443	11512	50
WS	238	11372	11442	53
WS	240	11513	11581	59
WS	1	1	34	70
WS	12	426	459	70
ALL	240			518
ALL	5			630
ALL	2			631
ALL	3			638
ALL	4			643

Table 39: The 5 threads with least 2d halo points for each of the 4 subdomains and the 5 threads with the least sum of 2D halo points.

Area	#T	n2db	n2de	n3h
BS	240	118116	119206	774
BS	239	117148	118115	2017
BS	238	116290	117147	3182
BS	229	110010	110896	3338
BS	233	112977	113608	3411
IDW	1	1	420	662
IDW	2	421	835	1560
IDW	4	1255	1669	1868
IDW	3	836	1254	1929
IDW	5	1670	2068	1988
NS	240	18810	18908	446
NS	1	1	75	1092
NS	239	18727	18809	1234
NS	225	17719	17819	1801
NS	223	17485	17605	1810
WS	240	11513	11581	2
WS	239	11443	11512	16
WS	238	11372	11442	44
WS	237	11301	11371	72
WS	235	11162	11230	93
ALL	1			6103
ALL	240			9128
ALL	2			12457
ALL	3			14176
ALL	4			16035

Table 40: The 5 threads with least 3D halo points for each of the 4 subdomains and the 5 threads with the least sum of 3D halo points.

Area	#T	n2db	n2de	n2d
BS	128	54945	55257	313
BS	131	55969	56281	313
BS	134	56987	57302	316
BS	140	59045	59360	316
BS	137	58011	58328	318
IDW	239	80399	80640	242
IDW	169	60056	60298	243
IDW	240	80641	80885	245
IDW	237	79897	80142	246
IDW	236	79650	79896	247
NS	79	5625	5681	57
NS	100	7195	7252	58
NS	102	7354	7411	58
NS	77	5485	5542	58
NS	104	7507	7565	59
WS	1	1	34	34
WS	12	426	459	34
WS	19	700	733	34
WS	26	976	1009	34
WS	5	155	188	34
ALL	113			720
ALL	117			721
ALL	90			733
ALL	102			734
ALL	108			738

Table 41: The 5 threads with least 2D points for each of the 4 subdomains and the 5 threads with the least sum of 2D points.

Area	#T	n2db	n2de	n3d
BS	240	118116	119206	17032
BS	228	108981	110009	17419
BS	227	107992	108980	18038
BS	239	117148	118115	18606
BS	229	110010	110896	19951
IDW	14	5301	5777	4376
IDW	20	8012	8476	4529
IDW	21	8477	8939	4536
IDW	15	5778	6238	4649
IDW	16	6239	6695	4689
NS	221	17265	17384	1313
NS	223	17485	17605	1316
NS	220	17150	17264	1405
NS	240	18810	18908	1456
NS	224	17606	17718	1492
WS	240	11513	11581	69
WS	228	10674	10743	70
WS	231	10882	10951	72
WS	232	10952	11021	74
WS	238	11372	11442	75
ALL	240			26665
ALL	228			27119
ALL	227			28067
ALL	239			28803
ALL	229			29618

Table 42: The 5 threads with least 3D points for each of the 4 subdomains and the 5 threads with the least sum of 3D points.

G Appendix: Compiler handling of the momeqs subroutine

The aim of this appendix is to summarize how different compilers handle the most expensive subroutine `momeqs` present in `default_momeqs.f90`. Being able to vectorize is quite important to both the Intel Xeon Phi and NVIDIA GPUs and as revealed below this subroutine was not vectorized when we began this study, cf. [2](12-18) and [1](42-43) for a proper treatment. Figure 60, 61 and 62 shows how different compilers handled the original code. Figure 63, 64 and 65 shows how different compilers handle the rewrite, first approach whereas figure 66, 67 and 68 show how they handle the rewrites from the second approach. It should be stressed that the fact that the compiler can generate vector code does *not* imply that it generates good vector code. All we show here is that each rewrite reveals more information to the compiler allowing it to generate vector code. The compiler will typically generate code for different alignments of arrays but there might also be code for short vector counts and for long vector counts.

```
momeqs:
    138, Invariant assignments hoisted out of loop
    166, Memory copy idiom, loop replaced by call to __c_mcopy8
    171, Loop not vectorized: data dependency
         Loop unrolled 2 times
    202, Loop not vectorized: may not be beneficial
    266, Unrolled inner loop 4 times
```

Figure 60: The static analysis done by the PGI compiler, original code.

```
momeqs_standalone_v0.f90(166): (col. 9) remark: loop was not vectorized: vectorization possible
                                          but seems inefficient.
momeqs_standalone_v0.f90(171): (col. 7) remark: loop was not vectorized: existence of vector dependence.
momeqs_standalone_v0.f90(212): (col. 9) remark: loop was not vectorized: subscript too complex.
momeqs_standalone_v0.f90(269): (col. 26) remark: loop was not vectorized: subscript too complex.
momeqs_standalone_v0.f90(138): (col. 3) remark: loop was not vectorized: not inner loop.
```

Figure 61: The static analysis done by the Intel compiler, original code.

```
  do nsurf=nl,nu
ftn-6286 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v0.f90, Line = 138
  A loop starting at line 138 was not vectorized because it contains input/output operations at line 182.
ftn-6213 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v0.f90, Line = 166
  A loop starting at line 166 was conditionally vectorized.
ftn-6382 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v0.f90, Line = 171
  A loop starting at line 171 was partially vector pipelined.
ftn-6209 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v0.f90, Line = 171
  A loop starting at line 171 was partially vectorized.
ftn-6270 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v0.f90, Line = 204
  A loop starting at line 204 was not vectorized because it contains conditional code which is more
                  efficient if executed in scalar mode.
ftn-6332 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v0.f90, Line = 270
  A loop starting at line 270 was not vectorized because it does not map well onto the target architecture.
```

Figure 62: The static analysis done by the Cray compiler, original code.

```
momeqs:
    146, Invariant assignments hoisted out of loop
    173, Memory copy idiom, loop replaced by call to __c_mcopy8
    182, Loop not vectorized: data dependency
         Loop unrolled 2 times
    215, Loop not vectorized/parallelized: loop count too small
    289, Generated 4 alternate versions of the loop
         Generated vector sse code for the loop
    358, Generated 4 alternate versions of the loop
         Generated vector sse code for the loop
    377, Loop not vectorized: may not be beneficial
```

Figure 63: The static analysis done by the PGI compiler, rewrite first approach.

```
momeqs_standalone_v1.f90(173): (col. 9) remark: loop was not vectorized: vectorization possible
                                                but seems inefficient.
momeqs_standalone_v1.f90(182): (col. 7) remark: loop was not vectorized: existence of vector dependence.
momeqs_standalone_v1.f90(289): (col. 7) remark: LOOP WAS VECTORIZED.
momeqs_standalone_v1.f90(345): (col. 27) remark: loop was not vectorized: subscript too complex.
momeqs_standalone_v1.f90(358): (col. 7) remark: PARTIAL LOOP WAS VECTORIZED.
momeqs_standalone_v1.f90(358): (col. 7) remark: PARTIAL LOOP WAS VECTORIZED.
momeqs_standalone_v1.f90(387): (col. 9) remark: loop was not vectorized: subscript too complex.
momeqs_standalone_v1.f90(146): (col. 3) remark: loop was not vectorized: not inner loop.
```

Figure 64: The static analysis done by the Intel compiler, rewrite first approach.

```
ftn-6286 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v1.f90, Line = 146
  A loop starting at line 146 was not vectorized because it contains input/output operations at line 193.
ftn-6213 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v1.f90, Line = 173
  A loop starting at line 173 was conditionally vectorized.
ftn-6382 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v1.f90, Line = 182
  A loop starting at line 182 was partially vector pipelined.
ftn-6209 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v1.f90, Line = 182
  A loop starting at line 182 was partially vectorized.
ftn-6209 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v1.f90, Line = 289
  A loop starting at line 289 was partially vectorized.
ftn-6270 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v1.f90, Line = 313
  A loop starting at line 313 was not vectorized because it contains conditional code which is more
                              efficient if executed in scalar mode.
ftn-6209 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v1.f90, Line = 358
  A loop starting at line 358 was partially vectorized.
ftn-6270 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v1.f90, Line = 377
  A loop starting at line 377 was not vectorized because it contains conditional code which is more
                              efficient if executed in scalar mode.
```

Figure 65: The static analysis done by the Cray compiler, rewrite first approach.

```
momeqs:
    142, Invariant assignments hoisted out of loop
    170, Memory copy idiom, loop replaced by call to __c_mcopy8
    182, Generated 4 alternate versions of the loop
         Generated vector sse code for the loop
         Generated 2 shuffle instructions for the loop
    188, Generated 4 alternate versions of the loop
         Generated vector sse code for the loop
    219, Loop not vectorized/parallelized: loop count too small
    282, Generated 4 alternate versions of the loop
         Generated vector sse code for the loop
    287, Generated 4 alternate versions of the loop
         Generated vector sse code for the loop
    293, Generated 4 alternate versions of the loop
         Generated vector sse code for the loop
    299, Generated 3 alternate versions of the loop
         Generated vector sse code for the loop
    305, Generated 3 alternate versions of the loop
         Generated vector sse code for the loop
    310, Generated 3 alternate versions of the loop
         Generated vector sse code for the loop
    317, Generated 3 alternate versions of the loop
         Generated vector sse code for the loop
    326, Generated 3 alternate versions of the loop
         Generated vector sse code for the loop
    331, Generated 3 alternate versions of the loop
         Generated vector sse code for the loop
    334, Generated 3 alternate versions of the loop
         Generated vector sse code for the loop
    337, Generated 3 alternate versions of the loop
         Generated vector sse code for the loop
    342, Generated 3 alternate versions of the loop
         Generated vector sse code for the loop
    345, Generated 3 alternate versions of the loop
         Generated vector sse code for the loop
    348, Generated 3 alternate versions of the loop
         Generated vector sse code for the loop
    351, Generated 3 alternate versions of the loop
         Generated vector sse code for the loop
    388, Generated 3 alternate versions of the loop
         Generated vector sse code for the loop
```

Figure 66: The static analysis done by the PGI compiler, rewrite second approach.

```
momeqs_standalone_v3.f90(170): (col. 9) remark: loop was not vectorized: vectorization possible
                                          but seems inefficient.
momeqs_standalone_v3.f90(182): (col. 7) remark: FUSED LOOP WAS VECTORIZED.
momeqs_standalone_v3.f90(282): (col. 7) remark: LOOP WAS VECTORIZED.
momeqs_standalone_v3.f90(287): (col. 7) remark: LOOP WAS VECTORIZED.
momeqs_standalone_v3.f90(293): (col. 7) remark: LOOP WAS VECTORIZED.
momeqs_standalone_v3.f90(299): (col. 7) remark: LOOP WAS VECTORIZED.
momeqs_standalone_v3.f90(305): (col. 7) remark: LOOP WAS VECTORIZED.
momeqs_standalone_v3.f90(310): (col. 7) remark: LOOP WAS VECTORIZED.
momeqs_standalone_v3.f90(317): (col. 7) remark: LOOP WAS VECTORIZED.
momeqs_standalone_v3.f90(326): (col. 7) remark: LOOP WAS VECTORIZED.
momeqs_standalone_v3.f90(331): (col. 7) remark: LOOP WAS VECTORIZED.
momeqs_standalone_v3.f90(334): (col. 7) remark: LOOP WAS VECTORIZED.
momeqs_standalone_v3.f90(337): (col. 7) remark: LOOP WAS VECTORIZED.
momeqs_standalone_v3.f90(342): (col. 7) remark: LOOP WAS VECTORIZED.
momeqs_standalone_v3.f90(345): (col. 7) remark: LOOP WAS VECTORIZED.
momeqs_standalone_v3.f90(348): (col. 7) remark: LOOP WAS VECTORIZED.
momeqs_standalone_v3.f90(351): (col. 7) remark: LOOP WAS VECTORIZED.
momeqs_standalone_v3.f90(381): (col. 29) remark: loop was not vectorized: subscript too complex.
momeqs_standalone_v3.f90(388): (col. 9) remark: LOOP WAS VECTORIZED.
momeqs_standalone_v3.f90(142): (col. 3) remark: loop was not vectorized: not inner loop.
```

Figure 67: The static analysis done by the Intel compiler, rewrite second approach.

```
ftn-6286 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v3.f90, Line = 142
  A loop starting at line 142 was not vectorized because it contains input/output operations at line 197.
ftn-6213 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v3.f90, Line = 170
  A loop starting at line 170 was conditionally vectorized.
ftn-6382 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v3.f90, Line = 182
  A loop starting at line 182 was partially vector pipelined.
ftn-6209 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v3.f90, Line = 182
  A loop starting at line 182 was partially vectorized.
ftn-6209 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v3.f90, Line = 282
  A loop starting at line 282 was partially vectorized.
ftn-6004 crayftn: SCALAR MOMEQS, File = momeqs_standalone_v3.f90, Line = 287
  A loop starting at line 287 was fused with the loop starting at line 282.
ftn-6004 crayftn: SCALAR MOMEQS, File = momeqs_standalone_v3.f90, Line = 293
  A loop starting at line 293 was fused with the loop starting at line 282.
ftn-6004 crayftn: SCALAR MOMEQS, File = momeqs_standalone_v3.f90, Line = 299
  A loop starting at line 299 was fused with the loop starting at line 282.
ftn-6209 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v3.f90, Line = 305
  A loop starting at line 305 was partially vectorized.
ftn-6209 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v3.f90, Line = 310
  A loop starting at line 310 was partially vectorized.
ftn-6209 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v3.f90, Line = 317
  A loop starting at line 317 was partially vectorized.
ftn-6209 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v3.f90, Line = 326
  A loop starting at line 326 was partially vectorized.
ftn-6382 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v3.f90, Line = 331
  A loop starting at line 331 was partially vector pipelined.
ftn-6209 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v3.f90, Line = 331
  A loop starting at line 331 was partially vectorized.
ftn-6382 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v3.f90, Line = 334
  A loop starting at line 334 was partially vector pipelined.
ftn-6209 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v3.f90, Line = 334
  A loop starting at line 334 was partially vectorized.
ftn-6382 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v3.f90, Line = 337
  A loop starting at line 337 was partially vector pipelined.
ftn-6209 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v3.f90, Line = 337
  A loop starting at line 337 was partially vectorized.
ftn-6382 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v3.f90, Line = 342
  A loop starting at line 342 was partially vector pipelined.
ftn-6209 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v3.f90, Line = 342
  A loop starting at line 342 was partially vectorized.
ftn-6382 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v3.f90, Line = 345
  A loop starting at line 345 was partially vector pipelined.
ftn-6209 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v3.f90, Line = 345
  A loop starting at line 345 was partially vectorized.
ftn-6382 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v3.f90, Line = 348
  A loop starting at line 348 was partially vector pipelined.
ftn-6209 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v3.f90, Line = 348
  A loop starting at line 348 was partially vectorized.
ftn-6382 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v3.f90, Line = 351
  A loop starting at line 351 was partially vector pipelined.
ftn-6209 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v3.f90, Line = 351
  A loop starting at line 351 was partially vectorized.
ftn-6332 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v3.f90, Line = 356
  A loop starting at line 356 was not vectorized because it does not map well onto the target architecture.
ftn-6382 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v3.f90, Line = 388
  A loop starting at line 388 was partially vector pipelined.
ftn-6209 crayftn: VECTOR MOMEQS, File = momeqs_standalone_v3.f90, Line = 388
  A loop starting at line 388 was partially vectorized.
```

Figure 68: The static analysis done by the Cray compiler, rewrite second approach.

H Appendix: The PanEU setup

This appendix presents the largest experimental setup that we run today, namely the pan-eu setup. This setup shows that we need to refine our current decomposition heuristics (both openMP and MPI) since the size of the different domains differ a lot and without improved heuristic the smaller domains will eventually prevent scaling. Table 43 and table 44 summarizes the setup shown in figure 69. The I_r number for this setup is 182.1. After the pointer rewrites and the following cleanup in global arrays this case can now run in approximately 9 Gb of memory. With the asynchronous IO server one should then be able to fit the whole setup into a system with say a Sandy Bridge CPU handling the IO and a Xeon Phi coprocessor handling the computations.

	NA	**NS**	**MS**	**WS**	**IDW**	**BS**
resolution [n.m.]	3.0	3.0	3.0	1.0	0.5	1.0
mmx [N/S]	859	348	341	149	482	720
nmx [W/E]	341	194	567	156	396	567
kmx	78	50	84	24	77	122
gridpoints	22847682	3375600	16241148	557856	14697144	49805280
iw2	104527	18908	73746	11581	80884	119206
iw3	5647632	479083	4214803	103441	1583786	6112717
f_{iw3}	25%	14.2%	26.0%	18.5%	10.8%	12.3%
φ [latitude]	65 52 30N	65 52 30N	47 16 30N	55 41 30N	57 35 45N	65 53 30N
λ [longitude]	16 22 28W	04 07 30W	5 27 30W	06 10 50E	09 20 25E	14 35 50E
$\Delta\varphi$	0 3 0	0 3 00	0 3 0	0 1 00	0 0 30	0 1 00
$\Delta\lambda$	0 5 0	0 5 00	0 5 0	0 1 40	0 0 50	0 1 40
dt [sec]	10.0	10.0	10.0	10.0	10.0	10.0
maxdepth [m]	6087.69	696.25	5066.49	53.60	78.00	398.00
min Δx	4029.34	3787.40	5559.78	1740.97	827.62	1261.65
CFL	0.798	0.319	0.708	0.250	0.632	0.728
I_r	58.6	1.8	39.9	0.4	12.0	46.3

Table 43: The first 6 sub-domains of the testcase termed paneu.

	GIBR	DRE	BOSE
resolution [n.m.]	1.0	0.3	0.1
mmx [N/S]	119	90	108
nmx [W/E]	145	120	150
kmx	66	35	34
gridpoints	1138830	378000	550800
iw2	8656	1415	2535
iw3	390316	27765	59123
f_{iw3}	34%	13.6%	10.1%
φ [latitude]	36 44 30N	40 29 50N	41 14 55N
λ [longitude]	06 24 28W	26 10 10E	28 00 03E
$\Delta\varphi$	0 1 00	00 00 20	00 00 10
$\Delta\lambda$	0 1 00	00 00 20	00 00 06
dt [sec]	5	5	2.5
maxdepth [m]	1531.97	81.69	77.70
min Δx	1485.09	469.76	139.34
CFL	0.742	0.533	0.767

Table 44: The MS subdmain have another 3 subdomains enclosed amounting to a total of 9 subdomains.

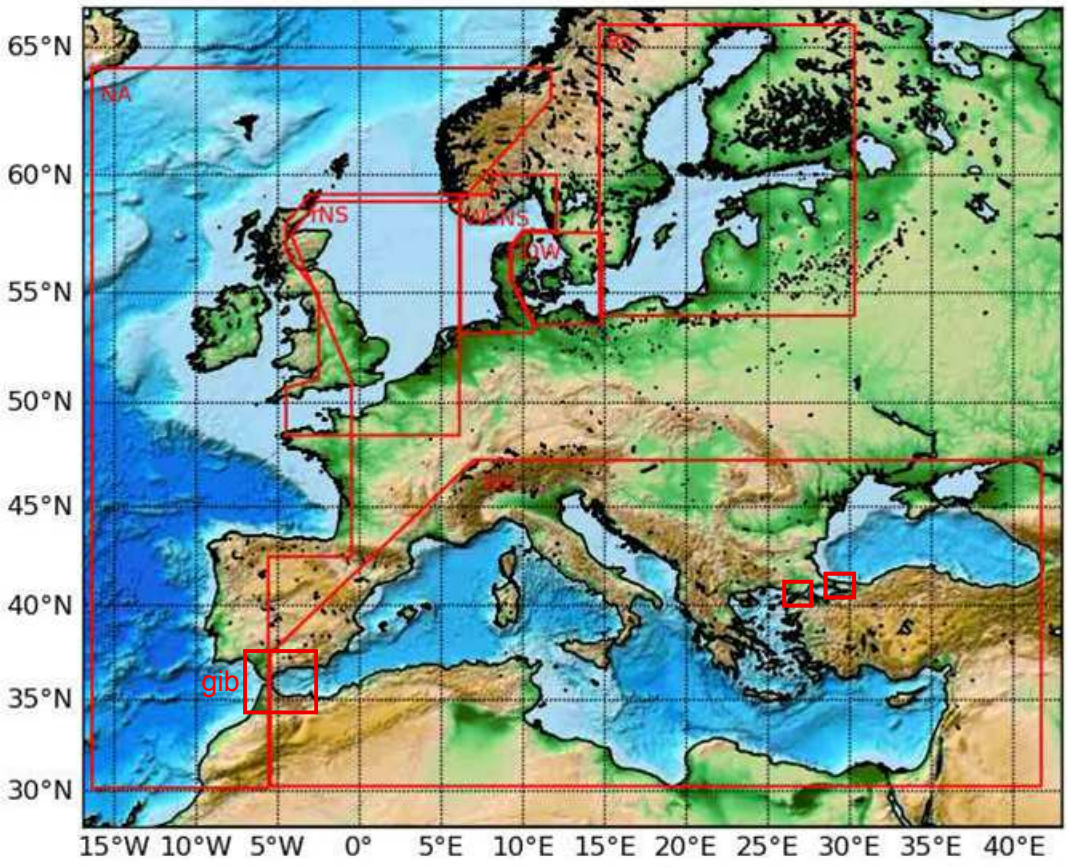


Figure 69: The 9 subdomains that make up the pan-eu testcase.

References

[1] Per Berg and Jacob Weismann Poulsen. Implementation details for HBM. DMI Technical Report No. 12-11. Technical report, DMI, Copenhagen, 2012.

[2] Jacob Weismann Poulsen and Per Berg. More details on HBM - general modelling theory and survey of recent studies. DMI Technical Report No. 12-16. Technical report, DMI, Copenhagen, 2012.

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