Complete HBM model runs on Intel® Xeon® processors and Intel® Xeon Phi™ processors - part I

Per Berg[*] Karthik Raman[†] Jacob Weismann Poulsen[*]

September 14, 2016

1 Preface

In this paper, we provide a survey of recent node/card performance results attained by the HBM model, cf. [1], [2], [3] and [4].

Previously, when we have discussed HBM performance results we have focused mainly on performance of kernels related to HBM. Studying these kernels have revealed parts that are highly bound on memory bandwidth and parts that tend to be more compute bound and the focused studies have allowed us to improve the individual performance of these kernels. In this paper, we will turn our attention to **complete** model runs. Moreover, we intend to present the performance results in the context that matters most to the end users of the model, namely as *time-to-solution* results and as *nodes-needed-for-production-runs* and *kilowatts-required-for-production-runs*. A run is classified as a production run if the run will complete a given forecast length (say a 24 hour simulation) in less than a specified wall-time (say 10 minutes).

Traditionally, software have been developed under the assumption that performance comes from hardware[1]. Increased performance came from increased processor clock speed. The increased processor clock speed came

[*]DMI
[†]Intel

[1]see e.g. Tim Mattson, `https://www.youtube.com/watch?v=cMWGeJyrc9w`

at a price, namely increased power consumption. Thus, increasing performance was achieved at the cost of increasing power at a rate faster than linear. This unsustainable power model eventually hits the so-called power wall.

This evolution of processor clock speed stalled quite some time ago and the only way to increase performance, without breaking the power budget, is to add parallelism and improve data locality[2]. Performance really has become a question about readiness for parallel computations. Thus, performance must come from the software. A keyword for the required work on the software (or so-called code modernization) is scalability. By scalability we here mean

- **fabric scalability**: the ability to scale across nodes, distributed memory (MPI, CAF)
- **thread scalability**: the ability to scale with increasing number of cores/threads on each node/card, shared memory (OpenMP)
- **vector scalability**: the ability to utilize vectorization for data parallelism, SIMD

Please note, scalability does not necessarily imply efficiency, nor does it imply portable performance. Thus, as a typical example, one could think of an application which possesses near-perfect scaling across a couple of nodes on one particular system but for which the performance on the individual multi-core nodes is hopeless and/or for which performance is not portable.

Ensuring the needed *efficient scalability* is an on-going process and code modernization work generally improves performance on any processor. The need for such modernization is not temporary. It is therefore necessary not only to code applications for present days hardware, but to ensure *portable scalability* that also incorporates the evolution trends and ensure scalability towards future hardware.

The present paper is a snapshot, demonstrating where we are and how far we have come with respect to code modernization of HBM when a single

[2]see e.g. James Reinders, "Moving Down the Path Toward Code Modernization", `http://www.hpcwire.com/2015/08/19/moving-down-the-path-toward-code-modernization/`

node is considered. The source code tag used for the present paper is `16049`.

The first section introduces the testcase and the benchmark systems that we will use for this study. Moreover, it surveys the main components within the timeloop from a computational perspective. The second section presents performance results attained on three generations of Intel® Xeon processors. The third section presents out-of-the-box performance attained on 1st generation Intel® Xeon Phi™ coprocessors (codenamed Knights Corner™) and 2nd generation Intel® Xeon Phi™ processors (codenamed Knights Landing™) and it cross-compares this with the performance attained on the Intel® Xeon™ processors. Finally, we present our conclusions in section four.

This paper is the first in a planned series of three papers and this one is meant to serve as baseline reference report. It will present out-of-the-box node performance results on Intel Xeon Phi and Intel Xeon processor nodes. The second will present out-of-the-box cluster performance results. (aiming at Q4, 2016). These first two papers constitutes the baseline performance as is, and the results revealed therein will be used as guidelines for prioritizing further optimization efforts. The final paper will present cluster performance results after we have had some time to tune the code for modern processors (aiming at Q3, 2017). This will complete our modernization efforts on HBM towards the new HPC cluster that DMI will receive towards the end of 2017.

2 Introduction

This section will introduce the test-cases that we will deal with in this and the following papers. It will also summarize relevant runtime information such as memory footprints and IO initialization requirements. Finally, we will summarize the hardware that we have used during this study.

2.1 The Baffin Bay setups

To make our points and not to complicate things more than necessary, we will neither use meteo-forcing fields nor other forcing data in this paper, which may even be protected by proprietary rights, yet we would like to demonstrate our findings on realistic data sets that we can share freely with anyone interested. Therefore, we have supplied with the source code relevant

data from a Baffin Bay setup generated from the freely available ETOPO2[3] data set.

Figure 1 shows the geographical location of the setup and the bathymetry and table 1 summarizes the set of Baffin Bay setups. The setups distinguish themselves from our usual setups because they do not apply our two-way nesting feature, cf. [1]. We hope that this will ease the interpretation of the out-of-the-box performance results. We have chosen to present the set of setups in this first paper to reveal the obvious challenges that one faces when the resolution is increased but also to demonstrate that our compact index representation allow us to run relatively large setups on a relatively small footprint.

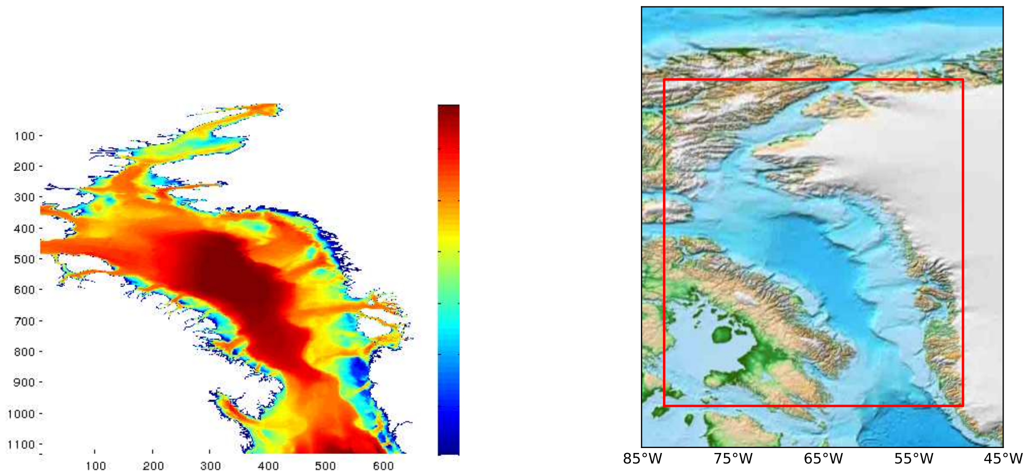

Figure 1: Map of the 1nm bathymetry for the Baffin Bay setup (left) and map of Baffin Bay area (right).

In the present paper we focus solely on node performance, i.e. the performance related to threading and vectorization in a shared memory context. We acknowledge the fact that other technical issues, such as performance related to I/O and distributed memory, as well as other model features, such as nesting and open boundary forcing, do indeed play huge roles for

[3]ETOPO2 is a 2 arc-minute global relief model of Earth's surface that integrates land topography and ocean bathymetry,
http://www.ngdc.noaa.gov/mgg/global/global.html

	BB_2	**BB_1**	**BB_05**	**BB_025**	**BB_0125**
resolution [n.m.]	2.0	1.0	0.5	0.25	0.125
resolution [m]	3704	1852	926	463	232
mmx [N/S]	565	1130	2260	4520	9040
nmx [W/E]	331	661	1322	2644	5288
kmx [layers]	137	137	137	137	137
3D gridpoints	25621055	102329410	409317640	1637270560	6549082240
3D wetpoints	7136949	27838287	111353148	445412592	1781650368
ratio3 [%]	27.9	27.2	27.2	27.2	27.2
2D points	187015	746930	2987720	11950880	47803520
2D wetpoints	77285	300822	1203288	4813152	19252608
ratio2 [%]	41.3	40.3	40.3	40.3	40.3
φ [latitude]	81° 50′0″ N	81° 50′0″ N	81° 50′0″ N	81° 50′0″ N	81° 50′0″ N
λ [longitude]	82° 30′0″ W	82° 30′0″ W	82° 30′0″ W	82° 30′0″ W	82° 30′0″ W
$\Delta\varphi$	0° 2′00″	0° 1′00″	0° 0′30″	0° 0′15″	0° 0′7.5″
$\Delta\lambda$	0° 6′00″	0° 3′00″	0° 1′30″	0° 0′45″	0° 0′22.5″
dt [sec]	10	5	2.5	1.25	0.625
maxdepth [m]	2387.00	2387.00	2387.00	2387.00	2387.00
min Δx [m]	1579.57	789.78	394.89	197.35	98.72
CFL	0.9	0.9	0.9	0.9	0.9
Complexity (I_r)	68	528	4221	33768	270144
IO (hz) [Mb]	55	213	850	3399	13593

Table 1: The Baffin Bay testcases. The color yellow in the cells is used to denote challenges that the implementation has to deal with as the setup increases. Yellow indicates data set sizes where it may not be a trivial task to read it in; reading these data may contribute a significant portion to the timings. Please consult [1] and [2] for a definition and discussion of the computational complexity index I_r.

any real operational model application, but we will here liberate ourselves from burdens like those.

Let us try to make some very crude extrapolations of the setups in table 1 into corresponding global setups. Size-wise, seen from the number of grid-points, the BB_025 testcase in table 1 coincides with global ocean setups that are in $\approx$ 4 km range and BB_0125 coincides with a global setup in the $\approx$ 2 km range, but remember that the timesteps for the BB_025 and BB_0125 setups are significantly smaller than they would be for the corresponding global testcases due to the very fine resolution. Now, time-wise, the time it takes to solve say BB_2 coincides with the time it would take to solve a global setup with around 2260x1322x137 points in a 15 km resolution, or the time it takes to solve BB_05 corresponds to the time it would take to solve a global setup with around 9040x5288x137 points in 4 km resolution. Thus, we trust that we span the whole range of sizes and resolutions that are used in production today onto the sizes and resolutions that are beyond what we would imagine to appear within the next 5 years.

Note that the BB_05 and onward are artificial extensions of BB_1. The one following BB_0125, namely BB_00625, requires that the number of 3D wetpoints is represented in 64-bit integers. Thus, asynchronous IO-servers constitute a mandatory component once we get to such setups. Hence, we will confine our initial studies to the first 4-5 cases. The smallest case, BB_2, is small enough to fit into the memory of a single node or accelerator whereas the larger one, BB_025, requires several nodes (assuming the usual capacity of 64 GB) just to run, at least if we do not use dedicated IO servers to handle the global MPI variables. In this paper, where we focus on node performance, we will confine the treatment to the BB_2 case.

2.2 Imbalance inherited in the setup

There are two important aspects related to imbalance. Firstly, we need to split the work properly among the threads and this is a non-trivial problem, cf. [4] and figure 2. Secondly, we need to ensure proper CPU and memory affinity.

NUMA layout. All the 2D and 3D arrays are first-touched according to threads that will operate on these arrays, cf. [4]. This does give rise to a reasonable NUMA layout with a few exceptions. First, the land-point index

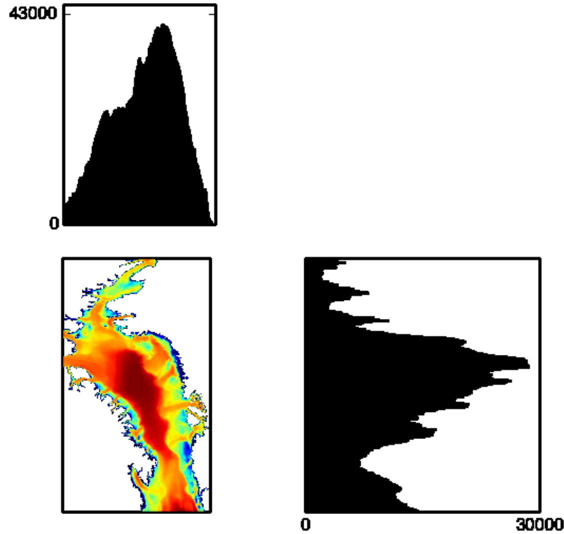

Figure 2: Irregularity of the problem. The color coding shows the number of wet-points below each surface point, white is on land, and the color scale runs from dark blue for 1 point to dark red for 137 points. The histogram to the right shows the distribution of number of wet-points along each zonal (i.e. constant latitude) grid line. The upper histogram shows the distribution of number of wet-points along each meridional (constant longitude) grid line. This figure is meant to convey the challenges that one faces when it comes to split the work evenly among the threads.

0 which is placed on the `MASTER` thread which again will be placed on the first socket. Second, the tiny fraction of the indices that belong to the page that hold indices belonging to several threads that could be spread across sockets. With the current implementation there is no guarantee that these threads are placed on the same socket. The decomposition algorithm could ensure that the split between threads that live on different sockets always happens at page boundaries and this would be a simple improvement to the current algorithm which we will consider in our future work on improving the model.

2.3 Memory footprint

It is important to understand the memory requirements as we scale the setups so table 2 is an attempt to summarize this.

The largest case can run on a single node assuming that this node has 256 GB of memory available but requires at least 7 nodes if they have the "usual" capacity of 64 GB. It should be mentioned that different compilers and different compiler options do change these numbers a bit so the numbers in the table should be read as ballpark numbers. The numbers table 2 were attained from a build with the Cray compiler using optimization flags (see section 2.5). Note that the single node fingerprint (which includes both data and code) translate roughly into an equivalent of 62 `real(8)` fields.

#tasks	**BB_2 [GB]**	**BB_1 [GB]**	**BB_05 [GB]**	**BB_025 [GB]**
1	3.3	12.9	51.5	
2	2.0	7.8	30.9	
3	1.2	6.0	24.0	
4	1.4	4.8	20.6	
5	1.2	4.7	18.5	
6	1.1	4.3	17.1	
7	1.1	4.1	16.2	55.0
8	1.0	3.9	15.4	51.9
9	1.0	3.7	14.8	49.5
10	1.0	3.6	14.4	47.7
20	0.8	3.1	12.3	39.2
30	0.8	2.9	11.6	43.0
40	0.8	2.8	11.2	34.9
50	0.7	2.8	11.0	40.7
60	0.1	2.7	10.9	30.5
70	0.7	2.7	10.8	33.1
80	0.1	2.7	10.7	39.5

Table 2: Memory footprint for different setups as function of MPI task count. The table shows how much memory the largest task is consuming and we focus on nodes with 64 GB of memory.

2.4 Initialization

As the setup increases, we have to focus not only on the timeloop but also on the initialization and the IO taking place during initialization. As revealed in table 1, the bathymetry file grows with the setup and reading in several GB does take several seconds and we have to consider how to do it properly as the setups grow. In table 1, it is the BB_025 and BB_0125 that really require some attention but the fact that this is indeed the case just shows that we need to deal with this in order to prepare ourselves for the future setups. Our real solution is to use IO-servers but even in the cases where we run without IO-servers, it is still worth to read in using N tasks (N should be seen in relation to the total number of tasks and to the size of the input file) and `BCAST` to the remaining number of tasks. If we do cold-start runs only, then we can confine ourselves to deal with the bathymetry file at scale but for restart runs, we need to deal with that too. For restart files, the solution at scale is to use asynchronous IO-servers.

2.5 The benchmark system

The systems (IvyBridge, Haswell, Broadwell, KnightsCorner and KnightsLanding) all run `Red Hat Enterprise Linux Server (Santiago)`. The IvyBridge system uses `release 6.5` whereas the rest of the systems use `release 6.6`. We have used the same compiler and the same tools on all the systems in order to ensure consistency. The only exception is the memory footprint information in table 2 which was obtained on a different system[4] The compiler used for all performance tables is the Intel® Fortran compiler, Version 16.0.0.109, Build 20150815. The specifications of the systems are summarized in table 3-6. Moreover all runs have been conducted with the default pagesize which in this case where transparent huge pages (THP) is used become 2Mb. On KnightsLanding, the cluster mode is quad and the memory mode is flat in all runs. All runs but the ones without Intel® HyperThreading technology are done using `KMP_AFFINITY=compact`. The runs without HyperThreading use affinity settings like the particular Haswell example shown below:

```
KMP_AFFINITY=proclist=[0-27:1],granularity=thread,explicit
```

All timings are done using `omp_wtime` and we focus on the timing surrounding the whole **timeloop** including all barriers etc. as well as the worst-

[4] A cray XC system with 10c IVB nodes using the cray compiler and the craypat tools to obtain the information.

case timings, i.e. the slowest thread, done inside the parallel region surrounding the 13 most important components. The 13 components accounts for more than 95% of the total timeloop time. It is important to stress that one should *not* expect that the worst-case timings sum to the timeloop timings simply because they are worst-case timings and also because we have left out some tiny blocks because they do not contribute to the overall picture.

	IVB	**HSW**	**BDW**
Micro-architecture	IvyBridge	Haswell	Broadwell
Model	E5-2697v2	E5-2697v3	E5-2697v4
Released	Q3, 2013	Q3, 2014	Q1, 2016
Sockets/node	2	2	2
Cores/socket	12	14	18
Multi threading	HT	HT	HT
Threads/core	2	2	2
Threads/node	48	56	72
Frequency [GHz]	2.7	2.6	2.3
Lithography [nm]	22	22	14
ISA	AVX	AVX2	AVX2
Memory channels/socket	4	4	4
Memory access	NUMA	NUMA	NUMA
Memory	DDR3 1600	DDR4 2133	DDR4 2400
Memory [GB/node]	64	64	128
L3 cache [MB/socket]	30	35	45
L2 cache [KB/core]	256	256	256
D1 cache [KB/core]	32	32	32
TDP/socket [W]	130	145	145
Ideal [GF/s]	518	1165	1324
HPL [GF/s]	492	949	1236
HPL efficiency [%]	95	81	93
HPL power [W]	700	750	545
HPL [GF/s/W]	0.70	1.26	2.26
HPL/core [GF/s]	20.5	33.9	35.5
Ideal BW [GB/s]	102	136	154
Triad [GB/s]	86	107	129
Triad efficiency [%]	84	78	84
Triad power [W]	620	380	425
Triad [GB/s/W]	0.135	0.287	0.303
Triad/core [GB/s]	3.58	3.82	3.58
Balance [bytes/flops]	0.17	0.11	0.10

Table 3: Specifications for the Intel® Xeon® processor-based systems used throughout this paper. Note that TDP ratings are per core whereas the quoted power numbers and performance/Watt numbers are all system power.

	KNC	**KNL**
Micro-architecture	KnightsCorner	KnightsLanding
Model	7120A	7210
Released	Q2, 2013	Q2, 2016
Cards—Sockets	1	1
Cores/Socket	60	64
Tiles/Socket	N/A	32
Multi threading	SMT	SMT
Threads/core	4	4
Threads/node	240	256
Frequency [GHz]	1.238	1.3
AVX Frequency [GHz]	N/A	1.1
Mesh Frequency [GHz]	N/A	1.6
Lithography [nm]	22	14
ISA	ICMI	AVX-512
Memory	GDDR5	MCDRAM
Memory transerfs	5.5 GT/s	6.4 GT/s
Memory access	UMA	UMA
Memory channels/socket	16	8
Memory [GB]	16	16
Memory mode	N/A	Flat
L3 cache	N/A	N/A
L2 cache [KB]	512/core	1024/tile
D1 cache [KB/core]	32	32
TDP [W]	300	215
Ideal [GF/s]	1188	2662

Table 4: Specifications for the Intel® Xeon Phi$^{\text{TM}}$ processor-based systems (to be continued on next page) used throughout this paper. Note that L2 is shared on KNL. Also note that KNL has access to both fast bandwidth MCDRAM and slower bandwidth DDR4 RAM but we have only specified MDCRAM related numbers in the table since we run the whole model in MCDRAM and do not use the DDR4 RAM attached at all. Finally, note that KNL also come in two other SKUs, namely 7230 (bin2) and 7250 (bin1). The latter has 68 active cores and runs at 1.4Ghz and the uncore frequency is 1.7 Ghz instead of 1.6 Ghz. Moreover, the memory transfers are 7.2 GT/s instead of 6.4 GT/s. Finally, stream triad performance is 490 Gb/s instead of 440 Gb/s.

	KNC	**KNL**
Micro-architecture	KnightsCorner	KnightsLanding
Model	7120A	7210
Ideal [GF/s]	1188	2662
HPL [GF/s]	999	1800
HPL efficiency [%]	84	68
HPL power [W]	312	455
HPL [GF/watt]	3.2	3.95
HPL/core [GF/s]	16.7	28.1
Ideal BW [GB/s]	352	≈600
Triad [GB/s]	177	440
Triad efficiency [%]	50	73
Triad power [W]	528	403
Triad [GB/watt]	0.335	1.09
Triad/core [GB/s]	3.0	6.9
Balance [bytes/flops]	0.18	0.24

Table 5: Specifications for the Intel® Xeon Phi™ processor-based systems used throughout this paper - continued.

	IVB	**HSW**	**BDW**	**KNC**	**KNL**
x86_64	yes	yes	yes	no	yes
x87	yes	yes	yes	yes	yes
SSE	yes	yes	yes	no	yes
AVX	yes	yes	yes	no	yes
AVX2(FMA3)	no	yes	yes	no	yes
AVX512	no	no	no	no	yes
ICMI	no	no	no	yes	no
#cores [%]	100	117	150	250	267
#cores time [%]	100	85.7	66.7	40	37.5
#threads [%]	100	117	150	500	533
#threads time [%	100	85.7	66.7	20	18.75
HPL [GF/s]	492	949	1236	999	1800
HPL ratio [%]	100	192	251	203	366
HPL time [%]	100	52	40	49	27
HPL EER [%]	100	180	323	457	564
Triad [GB/s]	86	107	129	177	440
Triad ratio [%]	100	124	150	206	512
Triad time [%]	100	80	67	49	20
Triad EER [%]	100	213	224	248	807

Table 6: Comparison of system characteristics. This table shows ballpark estimates for the time reductions we expect to see for BW bound parts and it also show an upper bound on the reductions we expect to see for the compute-bound components. Percentages are relative to the IVB which we have chosen as the reference. The HPL EER is the HPL energy efficient ratio, i.e. the HPL performance per Watt, relative to IVB. The Triad EER is the Triad energy efficient ratio, i.e. the Triad performance per Watt, relative to IVB.

2.6 The timeloop components

The characteristics of each of the 13 most time-consuming components within the timeloop are summarized in table 7 and a brief description of the functionality and computational characteristics of each of these is given in appendix A.

Component	SLOC	Bound	BARRIERs	non-SIMD	halo swaps
advection	5400	BW	yes		yes
deform	100	BW	yes		no
uvterm	200	BW	no		no
momeqs	800	BW/flops	no	trisolver	no
turbulence	500	flops	no	trisolver reduction	no
vdiff	100	flops	no	trisolver	no
diffusion	200	BW	yes		no
density	200	flops	(yes)	reduction	yes
sumuvwi	60	BW	no		no
bcldens	300	flops	no	reduction	no
masseqs	100	BW	yes	reduction	no
tflow_up	300	BW	(yes)		yes
smag	100	BW	no		no

Table 7: Summary of component characteristics. See text for explanations.

The `sloccount`[5] for the total HBM source code base is approximately 90.000 so the components we consider in this paper collectively amount to less than 10% of the HBM code lines[6] but still these components are consuming more than 95% of the run-time (cf. section 3.1). The remaining 90% of the code lines are dealing with matters like I/O and nesting and other model features that we do not consider in the present paper here, plus infrastructure components including domain decomposition, data permutation, array allocation and wrappers for OpenMP, MPI and utilities such as the timers which are also important and used here but not accounted for in table 7.

[5]http://www.dwheeler.com/sloccount/

[6]Antoine de Saint-Exupéry: *Perfection is attained, not when there is nothing more to add, but when there is nothing more to remove.*

The **Bound** column in table 7 reflects our immediate gut feeling about how each component is mainly bound. We here characterize components as either memory bandwidth bound (BW) or compute bound (flops).

For `BARRIER`s, "no" means that the component does not contain any `OMP BARRIER`s inside the timed code segment. A "yes" means that it does contain one or more `OMP BARRIER`s and that the last of these `OMP BARRIER` is immediately before exit of the timed chunk, while "(yes)" means that the component contains one or more `OMP BARRIER`s which are buried deep inside the component and not at the end.

Some of the components rely on algorithms that are inherently serial. This is depicted in the **non-SIMD** column. There is the solver for tri-diagonal systems of sets of linear equations (called trisolver), and there are reduction-type loops (e.g. summation accumulation over the water column). Note that the `turbulence` component has both of these, limiting this otherwise stream-friendly component by the scalar performance of the system.

The `halo swaps` column indicates if the component contains any MPI halo swaps inside the timed code segment.

3 Intel® Xeon® processor performance

In this section we present out-of-the box performance on different incarnations of the Intel Xeon processor-based platform. There are many ways that one can present the timings and each approach may give rise to new insights. The goal of this section is to reveal such insights.

3.1 Default decomposition

This subsection will present timings attained when running with one single MPI task and as many OpenMP threads as there are hyperthreads available on the node and when using the default geometric decomposition described in [1] and [4].

Table 8 summarizes these timings and it shows how the hardware improvements from one Intel Xeon processor incarnation to the next affect the various model components differently. It also shows that all the memory BW bound components (see table 7) do attain the 20% improvement that the stream benchmark attains (cf. table 6). Moreover, we observe that `vdiff` on HSW performs under expectation and this ought to be further investigated. Table 9 shows the fraction of time that each of the chosen 13 components account for and based on this table it seems reasonable to explain the timeloop performance by focusing solely on the components that explain 95%-97% of the timeloop performance.

Component	**IVB [s]**	**HSW [s]**	**BDW [s]**	**IVB [%]**	**HSW [%]**	**BDW [%]**
advection	190	151	126	100	80	66
deform	15	12	10	100	81	68
uvterm	19	15	13	100	81	70
momeqs	37	31	23	100	83	62
turbulence	31	23	16	100	74	51
vdiff	8	7	4	100	87	49
diffusion	17	13	11	100	80	68
density	19	12	10	100	63	50
sumuvwi	22	17	14	100	78	65
bcldens	14	8	7	100	59	50
masseqs	11	8	7	100	79	69
tflow_up	31	24	20	100	78	66
smag	11	10	7	100	84	66
timeloop	**441**	**341**	**283**	**100**	**77**	**64**

Table 8: Worst case thread timings attained when running with a single MPI tasks and all available threads on the node. On the right-hand side, we show the timings in percent relative to our reference IVB.

Component	**IVB** [%]	**HSW** [%]	**BDW** [%]
advection	43	44	45
deform	3	4	4
uvterm	4	4	5
momeqs	8	9	8
turbulence	7	7	6
vdiff	2	2	1
diffusion	4	4	4
density	4	4	3
sumuvwi	5	5	5
bcldens	3	2	2
masseqs	2	2	3
tflow_up	7	7	7
smag	3	3	3
All the above	**96**	**97**	**95**

Table 9: Timings in percent relative to the full **timeloop** timings in table 8. It reveals the fractions of time within the timeloop that we have not accounted for (3%-5%) by focusing solely on the 13 components.

3.2 Hybrid OpenMP+MPI versus single task OpenMP

We will now show how the model performs using hybrid OpenMP+MPI and we will cross-compare this with single-task OpenMP runs with similar decomposition as the OpenMP+MPI hybrid runs. This experiment will indeed show the importance of data locality and of balancing the computations in the sense that the timings from the default decomposition are improved significantly. Figure 3 illustrates how one can partition the domain into sub-slices with an almost equal number of wetpoints in each slice. This is the decomposition strategy used in the hybrid runs and in the sliced OpenMP runs.

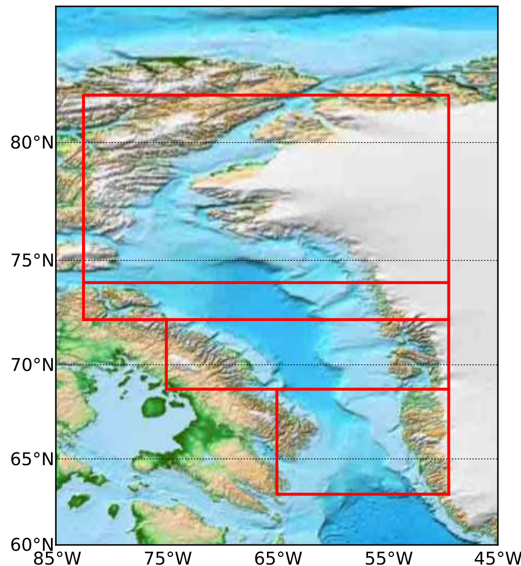

Figure 3: Example showing four slices (zonal split) of the Baffin Bay 1nm setup . With this attempted even-split the number of wetpoints in the 4 subareas become 6941116, 6976306, 6951686 and 6969179, respectively.

It is not uncommon to see that codes are better off using N MPI tasks within the nodes with $N > 1$. Having said that, we are not aware of any technical explanation that would justify this. We trust that a more likely explanation would either be a less efficient OpenMP parallelization strategy and implementation in the code itself or that one uses a compiler that comes

#tasks	#threads	Time [s]	#threads	#omp slices	Time [s]
1	48	436	48	1	436
2	24	439	48	2	428
4	12	424	48	4	415
6	8	410	48	6	400
8	6	403	48	8	396
12	4	397	48	12	386
24	2	420	48	24	384
48	1	451	48	48	398

Table 10: Hybrid runs and pure OpenMP runs with hybrid decomposition on IVB. Red highlights the slowest run whereas green is the fastest run.

#tasks	#threads	Time [s]	#threads	#slices	Time [s]
1	56	341	56	1	340
2	28	346	56	2	337
4	14	326	56	4	325
7	8	314	56	7	312
8	7	314	56	8	310
14	4	314	56	14	299
28	2	341	56	28	300
56	1	388	56	56	334

Table 11: Hybrid runs and pure OpenMP runs with hybrid decomposition on HSW. Red highlights the slowest run whereas green is the fastest run.

#tasks	#threads	Time [s]	#threads	#slices	Time [s]
1	72	278	72	1	279
2	36	277	72	2	273
3	24	271	72	3	267
4	18	266	72	4	262
6	12	255	72	6	256
8	9	252	72	8	251
9	8	252	72	9	248
12	6	251	72	12	242
18	4	253	72	18	239
24	3	260	72	24	240
36	2	272	72	36	260
72	1	318	72	72	242

Table 12: Hybrid runs and pure OpenMP runs with hybrid decomposition on BDW. Red highlights the slowest run whereas green is the fastest run.

with a less efficient implementation of the OpenMP specification. Being less well OpenMP parallelized will eventually prevent the code from scaling well in the sense that the fraction of the memory used to store the halo zones will exceed the number of active compute points much earlier with a pure MPI approach. With a pure MPI approach the fraction will grow with the number of cores whereas with a proper hybrid approach it will grow with the number of nodes and this is an important difference as the number of threads on the nodes seem to grow rapidly. At this point, we have chosen to add another confirmation check on our OpenMP strategy, namely to double check that we cannot beat it by substituting some of the threads with MPI tasks. Tables 10-12 confirm this hypothesis. The slice decomposition improves the performance by 12%-14% alone on all three systems, despite the fact that the cost of join-fork operations for the explicit OpenMP barriers increases with increasing OpenMP thread count. The upcoming paper which deals with cluster runs will hopefully reveal that this result is *not* due to an inefficient MPI implementation which would be one potential explanation but rather due to an efficient use of OpenMP in the code.

The *best* performance is attained using a slice count in the range $\#T/4-\#T/3$. Please also note that the *worst* performance on all three systems is attained when all available hyperthreads are used for MPI tasks; we sim-

ply loose approximately 1/3 in performance. For hybrid parallelised runs, the MPI communication benefits from the OpenMP threading since in the HBM implementation we do threaded encode and decode of send and receive buffers as well as MPI communication on different OpenMP threads.

Tables 13-14 show the timings for each of the computational components obtained for the OpenMP-slice experiments on BDW. Some components are virtually unaffected by varying the decomposition while the `advection` component shows a huge response up to 20%. It is good to confirm the expectation that compute bound components are not really affected by the decomposition and it is interesting to observe that some BW bound components are highly affected while other BW bound components are not affected much. This is a finding that is worth to dive deeper into.

Slices	1	2	3	4	6	8	9	12	18	24	36	72
advection	127	123	122	117	113	109	107	104	103	102	110	102
deform	10	10	9	9	9	9	9	9	9	9	9	9
uvterm	13	13	12	12	11	10	10	9	9	9	10	9
momeqs	23	23	23	23	22	23	22	21	21	21	22	21
turbulence	16	16	16	16	16	17	16	16	16	16	19	16
vdiff	4	4	4	4	4	4	4	4	4	4	4	4
diffusion	12	11	10	10	10	9	9	8	9	8	9	9
density	10	10	9	9	10	10	10	9	9	9	11	10
sumuvwi	15	14	14	15	14	14	15	15	15	15	17	15
bcldens	7	7	7	7	7	7	7	7	7	7	8	7
masseqs	7	7	7	6	6	6	6	6	6	6	7	6
tflow_up	21	20	20	20	20	19	19	19	19	19	20	19
smag	8	8	7	7	8	7	7	7	7	7	8	8
timeloop	279	273	267	262	255	251	248	242	239	240	260	242

Table 13: Timings in seconds for each individual component as well as for the whole timeloop when running on BDW with varying number of slices.

Slices	1	2	3	4	6	8	9	12	18	24	36	72
advection	100	97	96	92	89	86	85	82	81	80	86	81
deform	100	103	91	91	90	88	86	87	86	85	87	86
uvterm	100	99	90	90	83	75	76	73	72	72	76	68
momeqs	100	100	98	97	94	97	95	90	88	91	96	89
turbulence	100	99	100	100	98	102	101	99	97	98	118	100
vdiff	100	95	95	93	95	96	93	94	97	94	111	102
diffusion	100	93	91	87	83	79	76	73	74	73	76	74
density	100	98	95	95	97	98	96	95	95	95	111	100
sumuvwi	100	99	98	100	99	99	100	99	99	100	115	105
bcldens	100	100	100	99	101	101	100	100	100	99	119	106
masseqs	100	95	94	87	83	84	82	82	82	82	89	81
tflow_up	100	94	93	93	93	91	91	91	90	90	96	90
smag	100	101	92	93	98	92	94	92	91	91	98	99
timeloop	**100**	**98**	**96**	**94**	**91**	**90**	**89**	**87**	**86**	**86**	**93**	**87**

Table 14: Timings in percent relative to the default sliced decomposition for each individual components as well as for the whole timeloop when running on BDW with varying number of slices. Green is used to highlight the BW components that are highly improved by the best sliced decomposition.

3.3 Compiler flags

This subsection is devoted to the choice of compiler flags. We have tried many combinations of compiler flags and the conclusion of this study (not shown here) was that the flags do *not* contribute to the performance as long as we ensure to turn on Intel® Advanced Vector Extensions (Intel® `AVX`) and Intel® Advanced Vector Extensions 2 (Intel® `AVX2`), respectively. We have tried all combinations with the flags shown in table 15 and all trials have been repeated 10 times. We found that none of the many combinations showed significant improvements nor degradation over our default flags `-O2 -xAVX` and `-O2 -xCORE-AVX2` in the sense that the effect of substituting compiler flags was no larger than the unavoidable 1% run-to-run variations that we have experienced on the benchmark system. The only really interesting finding from this study was the effect of the SIMD instructions.

Table 16 and table 17 show the improvements attained due to Intel `AVX` and Intel `AVX2`, respectively. We see that the highly BW bound components cannot be improved since they already hit the BW roof but the ones that does not can indeed. Moreover, we see that for flop intensive components Intel `AVX` does indeed improve the performance significantly (up to 38%). The overall improvement gained from Intel `AVX` is around 10% which we is quite reasonable given that the majority of this code is purely BW bound. On the same flop intensive components Intel `AVX2` improves the performance further and `FMA3` certainly contributes to this improvements as shown for density and bcldens where we see impressive improvement beyond 130%. The overall timeloop improvement gained from Intel `AVX2` is around 23% which is quite good given that the majority of this code is mainly BW bound.

-O2
-O3
-xAVX vs -xCORE-AVX2
-no-inline-max-size
-align array64byte
-fimf-precision=low -fimf-domain-exclusion=15
-opt-streaming-stores always
-no-vec
-no-fma

Table 15: List of compiler flags. This table shows the single flags settings that we have used and as part of the investigation we tried all combinations of these flags.

Component	**AVX** [s]	**no-vec** [s]	**AVX** [%]	**no-vec** [%]
advection	157	170	100	108
deform	13	13	100	102
uvterm	13	14	100	101
momeqs	35	39	100	110
turbulence	31	37	100	120
vdiff	8	8	100	102
diffusion	13	14	100	106
density	19	26	100	134
sumuvwi	22	22	100	98
bcldens	14	19	100	138
masseqs	9	9	100	99
tflow_up	28	29	100	103
smag	11	11	100	104
timeloop	**387**	**423**	**100**	**109**

Table 16: Different compiler flags on IVB. This table shows the individual improvements or degradation that are imposed with or without Intel® Advanced Vector Extensions on IVB. On the left-hand side, we show the absolute timings whereas the right-hand side shows the relative effect. Colors are used to highlight significant improvements.

Component	**AVX2** [s]	**no-fma** [s]	**no-vec** [s]	**AVX2** [%]	**no-fma** [%]	**no-vec** [%]
advection	125	125	139	100	100	111
deform	10	11	11	100	105	105
uvterm	11	11	11	100	105	105
momeqs	28	29	39	100	104	142
turbulence	23	23	39	100	102	172
vdiff	7	7	7	100	100	107
diffusion	10	10	12	100	102	114
density	12	15	25	100	127	208
sumuvwi	17	18	17	100	101	98
bcldens	8	11	19	100	136	238
masseqs	7	7	7	100	99	101
tflow_up	22	23	23	100	101	101
smag	8	9	9	100	109	104
timeloop	**299**	**308**	**367**	**100**	**103**	**123**

Table 17: Different compiler flags on HSW. This table shows the individual improvements degradation that are imposed without `FMA3` and without Intel® Advanced Vector Extensions 2 (Intel® `AVX2`), respectively on a 2S HSW. On the left-hand side, we show the absolute timings whereas the right-hand side shows the relative effect. Colors are used to highlight significant improvements.

3.4 Thread imbalance

As mentioned in the introduction, this model deals with high-resolution setups and this fact combined with the fact that it uses z-grid coordinates in the vertical (as opposed to sigma-coordinates) pose some severe challenges to the task of balancing the work load.

We use the balance definitions from DeRose[7], i.e. let max_time be the maximal time among all threads and let avg_time be the average time for the threads and let NT be the total number of threads. Then

$$\begin{aligned} imb_time &= max_time - avg_time \\ imb_time\% &= \frac{NT * imb_time}{(NT-1) * max_time} \end{aligned}$$

That is, an imbalance of 100% for a component means that only one thread spent time in that component.

Tables 18-19 show that the imbalance is changed by the sliced approach. We see individual component-wise improvements stemming from a better balancing that are up to 28% and we see that some components are more neutral to the decomposition. Note also that the best sliced decomposition does not imply the best imbalance% for all components but indeed for the three worst balanced components in the default 1 slice case, namely two heavy `momeqs` and `turbulence` and the `tflow_up`.

In table 20 we have shown the imbalance for all three Intel Xeon processor-based systems using the best sliced decomposition. Note that there is a potential for gaining a couple of seconds if one is capable of improving the balance. It should be noted that there could be additional seconds in the components where we have barriers already. Moreover, we observe that for most components the balance issues do not increase much as we go to higher thread counts with `vdiff` being a notable exception, and perhaps a smaller tendency for the two flop intensive components `density` and `bcldens`.

[7] `http://opendl.ifip-tc6.org/db/conf/europar/europar2007/RoseHJ07.pdf`

Slices	1	2	3	4	6	8	9	12	18	24	36	72
advection	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
deform	0.02	0.02	0.04	0.04	0.05	0.06	0.05	0.04	0.05	0.05	0.03	0.05
uvterm	0.71	1.31	1.34	1.82	1.42	0.86	0.94	0.66	0.71	0.71	1.35	0.61
momeqs	2.79	3.14	2.99	3.17	3.16	3.92	3.72	2.95	2.30	2.97	4.31	2.33
turbulence	1.34	1.30	1.35	1.48	1.25	1.83	1.75	1.39	1.13	1.19	4.43	1.62
vdiff	0.32	0.15	0.16	0.09	0.13	0.23	0.12	0.10	0.28	0.15	0.80	0.48
diffusion	0.03	0.00	0.02	0.00	0.02	0.02	0.02	0.01	0.01	0.01	0.01	0.00
density	0.41	0.36	0.36	0.30	0.54	0.56	0.60	0.49	0.44	0.50	2.06	0.99
sumuvwi	0.65	0.50	0.32	0.69	0.38	0.46	0.63	0.48	0.55	0.62	2.86	1.50
bcldens	0.13	0.21	0.24	0.25	0.35	0.41	0.38	0.37	0.35	0.34	1.67	0.83
masseqs	0.00	0.00	0.03	0.03	0.03	0.01	0.02	0.02	0.01	0.01	0.04	0.01
tflow_up	0.79	0.46	0.33	0.64	0.72	0.42	0.50	0.55	0.47	0.57	0.65	0.47
smag	0.31	0.78	0.24	0.38	0.70	0.29	0.46	0.31	0.22	0.23	0.65	0.77

Table 18: Imbalance time for the BDW runs in table 13 and 14. Yellow is used to highlight the components that have internal barriers, green is the slicing that gives rise to the best timeloop timing and red the components with the largest imbalance.

Slices	1	2	3	4	6	8	9	12	18	24	36	72
advection	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
deform	0.18	0.21	0.39	0.47	0.51	0.68	0.60	0.48	0.58	0.57	0.29	0.53
uvterm	5.54	10.34	11.67	15.72	13.47	8.88	9.65	7.02	7.72	7.71	13.97	7.01
momeqs	12.11	13.58	13.23	14.19	14.57	17.55	17.01	14.21	11.31	14.12	19.49	11.42
turbulence	8.38	8.17	8.48	9.25	7.93	11.17	10.80	8.76	7.23	7.57	23.54	10.10
vdiff	8.26	4.25	4.43	2.38	3.53	6.12	3.42	2.79	7.45	4.05	18.70	12.34
diffusion	0.22	0.03	0.17	0.01	0.19	0.19	0.18	0.12	0.14	0.08	0.13	0.00
density	4.13	3.74	3.81	3.20	5.65	5.82	6.35	5.21	4.73	5.31	18.94	10.02
sumuvwi	4.47	3.48	2.29	4.75	2.64	3.24	4.33	3.31	3.83	4.33	17.19	9.89
bcldens	1.91	3.18	3.71	3.76	5.34	6.24	5.75	5.64	5.36	5.26	21.43	11.92
masseqs	0.04	0.01	0.46	0.43	0.46	0.21	0.25	0.39	0.24	0.22	0.59	0.22
tflow_up	3.77	2.32	1.68	3.29	3.72	2.19	2.61	2.88	2.49	3.03	3.22	2.51
smag	4.13	10.10	3.40	5.34	9.33	4.17	6.45	4.47	3.14	3.27	8.67	10.22

Table 19: Imbalance % for the BDW runs in table 13 and 14. Yellow is used to highlight the components that have internal barriers, green is the slicing that gives rise to the best timeloop timing and red the components with the largest imbalance.

Component	IVB [s]	HSW [s]	BDW [s]	IVB [%]	HSW [%]	BDW [%]
advection	0.0	0.0	0.0	0.0	0.0	0.0
deform	0.1	0.0	0.0	0.4	0.4	0.6
uvterm	1.0	0.6	0.7	7.2	5.5	7.7
momeqs	3.3	2.5	2.33	9.7	9.2	11.3
turbulence	2.2	1.2	1.1	7.1	5.4	7.2
vdiff	0.2	0.2	0.3	2.6	3.1	7.4
diffusion	0.0	0.0	0.0	0.1	0.1	0.1
density	0.7	0.5	0.4	3.6	4.2	4.7
sumuvwi	0.7	0.8	0.5	3.4	4.8	3.8
bcldens	0.6	0.3	0.4	4.1	4.0	5.4
masseqs	0.0	0.0	0.0	0.1	0.5	0.2
tflow_up	0.7	0.4	0.5	2.4	1.9	2.5
smag	0.9	0.2	0.2	7.9	2.8	3.1

Table 20: Individual imbalance time in seconds and the imbalance time in imbalance % on different systems. Yellow is used to highlight the components that have internal `BARRIER`s. These barriers will prevent us from studying balancing issues based on timers surrounding these components. They are maintained in the table solely to confirm this fact, i.e. the timers surrounding these components does not reveal balance issues for the yellow marked components. Red is used to mark the components that reveal potential balance issues and dark red is used to mark an unfortunate tendency where the imbalance seem to grow significantly with the number of threads.

3.5 HyperThreading

We will investigate whether or not HyperThreading pays off. Thus, we repeated the fastest sliced decomposition runs with and without HyperThreading and will study the performance of the individual components. This experiment could hint if we have components that are limited by the D1 and L2 cache. The fact that hyperthreads (on the core) share the first two levels of the cache system implies that if one thread is already pushing the cache to its limits then we do not expect to see improvements by adding more threads on that same core that will compete on the same resources.

Tables 21-23 seem to suggest that the importance of HyperThreading decreases (from 13% to 1%) with the newer architectures having more and more cores. Moreover, the benefits seem to be tied to the individual components and the effect varies from one component to the next, and even components that improve on one system degrades on another system. It seems that the highly BW bound components are challenges when it comes to benefit from HyperThreading and this is actually quite reasonable in the sense that the threads do share D1/I1 and L2. On the other hand, as expected, the compute bound components really benefit from HyperThreading.

Component	**48** [s]	**24** [s]	**48** [%]	**24** [%]
advection	157	181	100	115
deform	13	14	100	108
uvterm	13	16	100	123
momeqs	35	40	100	114
turbulence	31	38	100	123
vdiff	8	9	100	113
diffusion	13	16	100	123
density	19	21	100	111
sumuvwi	22	21	100	95
bcldens	14	15	100	107
masseqs	9	10	100	111
tflow_up	29	29	100	100
smag	11	10	100	91
timeloop	**387**	**438**	**100**	**113**

Table 21: Results of running with and without HyperThreading on IVB in second (left) and in percent relative to the 48 threads (right). Affinity settings as described in subsection 2.5. Green highlights the components that improve most from using HyperThreading.

Component	**56** [s]	**28** [s]	**56** [%]	**28** [%]
advection	125	119	100	95
deform	10	11	100	109
uvterm	11	11	100	101
momeqs	28	31	100	112
turbulence	23	28	100	122
vdiff	7	8	100	113
diffusion	10	10	100	98
density	12	14	100	116
sumuvwi	17	18	100	106
bcldens	8	11	100	136
masseqs	7	7	100	98
tflow_up	22	23	100	102
smag	8	9	100	110
timeloop	**299**	**309**	**100**	**103**

Table 22: Results of running with and without HyperThreading on HSW in second (left) and in percent relative to the 56 threads (right). Green highlights the components that improve most from using HyperThreading.

Component	**72** [s]	**36** [s]	**72** [%]	**36** [%]
advection	104	99	100	96
deform	9	8	100	96
uvterm	9	9	100	97
momeqs	21	24	100	117
turbulence	16	20	100	126
vdiff	4	4	100	113
diffusion	9	8	100	94
density	9	11	100	117
sumuvwi	15	15	100	97
bcldens	7	8	100	121
masseqs	6	6	100	93
tflow_up	19	19	100	100
smag	7	7	100	100
timeloop	**244**	**247**	**100**	**101**

Table 23: Results of running with and without HyperThreading on BDW in second (left) and in percent relative to the 72 threads (right). Green highlights the components that improve most from using HyperThreading.

3.6 Frequency variations

Nowadays it is possible to run code on the very same CPU but at different frequencies, the frequency being chosen at runtime. This allows one to run highly BW bound code at lower frequency and thereby save some watts but the timings attained from such experiments also allow one to do rather accurate single node extrapolations to different SKUs and even different incarnations of the Intel Xeon processor architectures. Tables 24-25 and tables 26-27 show the results of running at different frequencies on HSW and on BDW, respectively. Moreover, the BDW tables show the corresponding power measurements. The aim of this experiment is to classify the frequency bound components and also to assess to which extent that they are bound to the frequency. We will also try to reveal if there are cost benefits of running at lower frequencies.

Table 27 reveals that there is a small cost savings in running at a lower frequency: I.e. reducing frequency by 57% from 2.3 GHz to 1.3 GHz results in a 20% reduction in power but also the time required to complete the model run will increase by 20%, thus 4% is gained on the energy budget. The table also shows that the compute bound components gain up to 40% by running at the default frequency over the lower frequencies (or the degradation is up to 70% at the lowest frequency compared to default frequency). The tables re-confirms the classification of the compute bound components and can also be used to order them.

Component	**1.9 GHz**	**2.1 GHz**	**2.5 GHz**	**2.6 GHz**
advection	131	127	126	125
deform	10	10	10	10
uvterm	11	11	11	11
momeqs	34	32	28	28
turbulence	30	28	23	23
vdiff	9	8	7	7
diffusion	11	11	10	10
density	15	14	12	12
sumuvwi	19	17	17	18
bcldens	11	10	9	8
masseqs	7	7	7	7
tflow_up	23	23	22	23
smag	9	8	9	9
timeloop	**331**	**317**	**302**	**299**

Table 24: Timings in seconds attained under different frequencies on HSW.

Component	**1.9 GHz**	**2.1 GHz**	**2.5 GHz**	**2.6 GHz**
advection	105	102	101	100
deform	100	98	99	100
uvterm	100	96	98	100
momeqs	121	114	101	100
turbulence	133	121	103	100
vdiff	136	123	104	100
diffusion	104	104	100	100
density	128	118	103	100
sumuvwi	107	99	98	100
bcldens	134	122	104	100
masseqs	102	98	98	100
tflow_up	101	99	99	100
smag	99	97	107	100
timeloop	**111**	**106**	**101**	**100**

Table 25: Results of running with different frequencies on HSW. Numbers are in percent [%] relative to the 2.6 GHz results. Green highlights the most strongly frequency bound components.

Component	**1.3 GHz**	**1.5 GHz**	**1.7 GHz**	**1.9 GHz**	**2.1 GHz**	**2.2 GHz**	**2.3 GHz**
advection	114	111	108	103	104	104	103
deform	9	9	9	9	9	9	9
uvterm	9	10	10	10	9	10	10
momeqs	32	28	26	21	22	22	21
turbulence	27	24	21	16	17	16	16
vdiff	5	5	4	4	4	4	4
diffusion	9	9	9	9	9	9	9
density	15	14	12	10	10	10	10
sumuvwi	15	15	15	15	15	15	15
bcldens	11	10	9	7	7	7	7
masseqs	6	6	6	6	6	6	6
tflow_up	20	20	19	20	19	19	19
smag	7	7	7	7	7	7	8
timeloop	**291**	**276**	**265**	**242**	**248**	**245**	**244**
power [watt]	**366**	**385**	**432**	**422**	**457**	**467**	**459**

Table 26: Timings in seconds of frequency runs on BDW. The last row of the table is the measured power consumption. This is measured system power and not just CPU and/or memory.

Component	1.3 GHz	1.5 GHz	1.7 GHz	1.9 GHz	2.1 GHz	2.2 GHz	2.3 GHz
advection	111	107	105	100	101	101	100
deform	97	96	97	96	94	96	100
uvterm	98	99	98	100	96	101	100
momeqs	149	132	122	97	103	103	100
turbulence	167	146	130	98	107	101	100
vdiff	137	121	111	102	101	101	100
diffusion	108	104	103	99	102	102	100
density	160	141	127	99	106	103	100
sumuvwi	101	99	102	102	99	98	100
bcldens	171	148	131	99	109	102	100
masseqs	102	101	101	103	103	99	100
tflow_up	103	102	102	102	101	101	100
smag	93	92	94	93	89	94	100
timeloop	**120**	**113**	**109**	**99**	**102**	**101**	**100**
power [%]	**80**	**84**	**94**	**92**	**100**	**102**	**100**

Table 27: Results of running with different frequencies on BDW. Numbers are in percent [%] relative to the 2.3 GHz results. Green highlights the most strongly frequency bound components. The last row of the table is the measured power consumption relative to the 2.3 GHz run. This is measured system power and not just CPU and/or memory.

3.7 Individual scaling within the node

The computational components are split into those that never attain peak performance and those that actually do. The ones that attain peak performance are all memory BW bound components. The components that are memory BW bound show a classic scaling curve with a ramp tapering off when we move towards the end of socket 1 and then again rising almost linearly from the start of socket 2, see e.g. figure 6. The ones that never attain peak performance will scale all the way, see e.g. figure 5. That is, when we look at the scaling/speedup aspect we need to be careful in stating the reference and the placement strategy, since these both have a high impact on the shape of the curves and we should not be unsatisfied by an apparent poor scaling graph since it may simply reveal that we reach peak performance without using all threads available to us, nor should we be satisfied with an apparently ideal linear scaling unless we actually get close to the peak performance when using all threads available. Figure 4 summarizes the scaling of the whole timeloop on the different incarnations of the Intel Xeon processor architectures and we notice the overall BW bound shape of the curves.

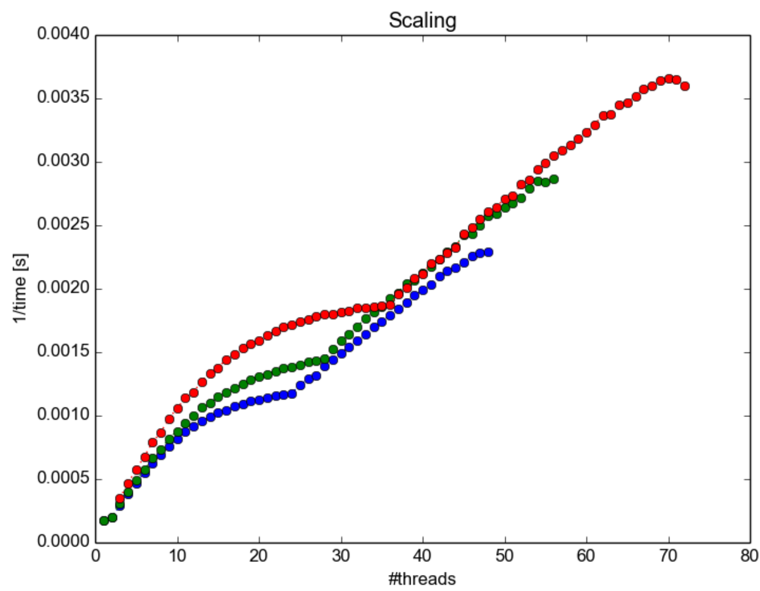

Figure 4: Scaling of the whole timeloop on the different generations of Intel® Xeon® processors. IVB is shown in blue, HSW in green, and BDW in red.

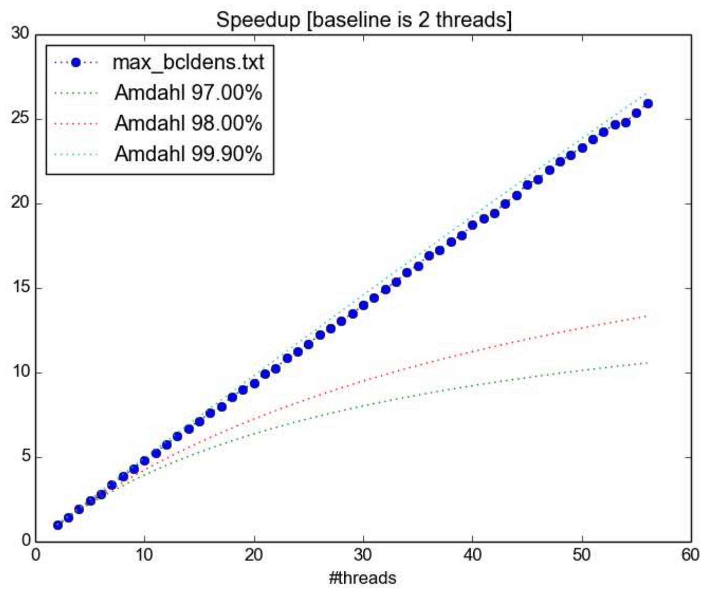

Figure 5: Example showing scaling of one of the FLOP bound components (`bcldens`).

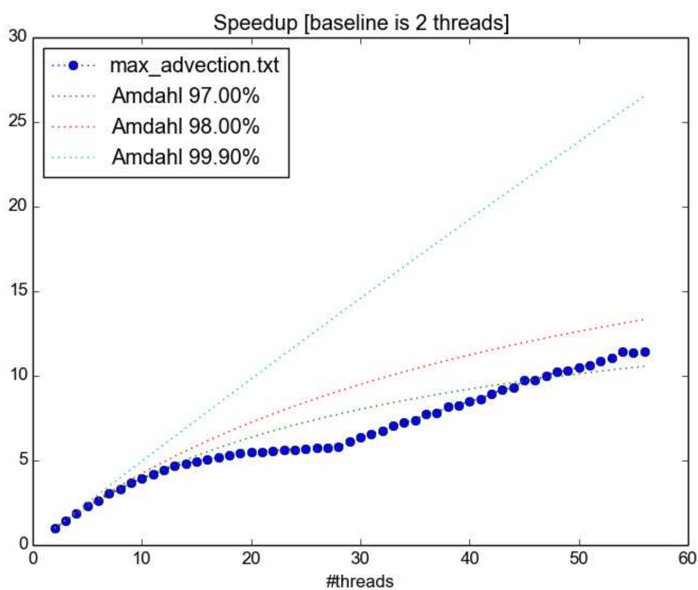

Figure 6: Example showing a typical scaling image of one of the BW bound components (`advection`). The slope of the scaling plot clearly shows how the BW is exhausted on one socket and then when the second socket comes into play the performance is improved again until this become exhausted too.

4 Intel® Xeon Phi™ coprocessor

In this section we present out-of-the box performance on KNC and we will cross-compare KNC performance with IVB performance.

4.1 Default decomposition

In the previous section, we started off with the default decomposition that is used in HBM. Table 28 shows the as-is performance attained on KNC using 240 threads versus the performance that we attained using the 48 threads available on the 2S IVB system. The table shows that the overall performance on KNC is slightly better than what we can attain on 2S IVB using the same default decomposition on both *but* when we look at the individual components we see large differences in attained performance. The BW bound components perform well on KNC, but the components with the non-SIMD friendly trisolvers and reduction (marked in red) behave poorly on KNC as compared to IVB. This is no surprise and it totally confirms our expectations. The table also shows that KNC with default decomposion is on par with the best performance on IVB that we can attain when we allow for sliced decompositions.

It is, however, slightly disappointing that we are far from the extrapolated performance of 49% suggested by the IVB/KNC ratio for both the HPL and the stream Triad. A similar behavior is also found for KNL versus BDW, and we therefore have postponed further discussion on this issue to the KNL section, see section 5.1.

Table 29 extends table 9 with a column for KNC, and one notes that the 13 chosen components are still representative for explaining the overall timeloop performance. It is interesting but not surprising to observe that the static trend seen for the Intel Xeon processor line is not carried over to KNC, i.e. the fraction that the different components accounts for is slightly different.

It should also be mentioned that the sliced decomposition does not perform so well on KNC as we revealed on the Intel Xeon processor line. Actually, running without slices outperforms all the sliced runs that we tried. The fact that this pattern repeats itself on KNL made us postpone the treatment of this issue to the KNL section, see section 5.4.

Component	**IVB** [s]	**IVB**$_S$ [s]	**KNC** [s]	**IVB** [%]	**IVB**$_S$ [%]	**KNC** [%]
advection	191	157	156	100	83	82
deform	15	13	8	100	85	55
uvterm	19	13	12	100	71	65
momeqs	37	35	51	100	96	139
turbulence	32	31	53	100	99	169
vdiff	8	8	16	100	102	212
diffusion	17	13	11	100	76	68
density	19	19	18	100	101	97
sumuvwi	22	22	12	100	102	54
bcldens	14	14	14	100	102	101
masseqs	11	9	8	100	84	73
tflow_up	30	28	18	100	94	60
smag	11	11	7	100	94	64
timeloop	**437**	**387**	**401**	**100**	**89**	**92**

Table 28: Worst case thread timing attained when running with a single MPI tasks and all available threads on the node, 2S IVB versus 1 KNC card. The first column is IVB run with the same decomposition as the KNC run whereas the second column IVB$_S$ is the best attainable timing we could get on IVB by using a sliced decomposition. The bold timeloop line is the timing attained around a serial block including all barriers and smaller components within the timeloop whereas the remaining timers are all from within a threaded context and represents the worst-case timing among all the threads. It is important to stress that one should *not* expect that the worst-case timings sum to the timeloop timings simply because they are worst-case timings and also because we have left out some tiny blocks because they do not contribute to the overall picture. Red marks the components that perform poorly relatively speaking.

Component	**IVB**	**HSW**	**BDW**	**KNC**
advection	43	44	45	39
deform	3	4	4	2
uvterm	4	4	5	3
momeqs	8	9	8	13
turbulence	7	7	6	13
vdiff	2	2	1	4
diffusion	4	4	4	3
density	4	4	3	5
sumuvwi	5	5	5	3
bcldens	3	2	2	3
masseqs	2	2	3	2
tflow_up	7	7	7	5
smag	3	3	3	2
All	**96**	**97**	**95**	**96**

Table 29: Timings in percent relative to the **timeloop** timings in table 8 and 28. It reveals the fractions of time within the timeloop that we have not accounted for (3%-5%) by focusing solely on the 13 components. Red marks the components that perform poorly relatively speaking.

4.2 Compiler flags

This subsection is devoted to the choice of compiler flags. We have tried many combinations of compiler flags and the conclusion of this study was that a couple of flags do contribute to the performance. We found that `-O2 -mmic` served as a better baseline than `-O3 -mmic` and tried several combinations of flags from table 15. It turned out that the streaming flags

```
-opt-streaming-stores always -opt-streaming-cache-evict=0
```

did improve the performance slightly for a couple of the memory BW bound components, notably `sumuvwi` and `deform`. Finally, it was also found that for a couple of compenents the flags

```
-fimf-precision=low -fimf-domain-exclusion=15
```

had significant impact, notably `momeqs`, `turbulence` and `vdiff`, i.e. those that are relatively heavy on e.g. `DIV` operations through the trisolverr. Table 30 summarizes the findings of this study and it should be mentioned that the best combination of flags additionally combined with `-no-vec` implies that the timeloop increases to 1050 seconds (not shown), so SIMD is, as expected, far more important on KNC than it was on the Intel Xeon processors in the sense that SIMD alone accounts for an overall performance factor of more than 2.6.

Component	**cf1** [s]	**cf2** [s]	**cf3** [s]	**cf1** [%]	**cf2** [%]	**cf3** [%]
advection	162	163	156	100	101	96
deform	10	8	8	100	81	98
uvterm	13	12	12	100	95	99
momeqs	69	68	51	100	99	76
turbulence	68	64	53	100	94	83
vdiff	22	22	16	100	98	75
diffusion	13	13	11	100	100	89
density	21	21	18	100	97	89
sumuvwi	15	12	12	100	78	98
bcldens	16	16	14	100	97	88
masseqs	8	8	8	100	97	100
tflow_up	20	19	18	100	96	96
smag	8	8	7	100	96	97
timeloop	**463**	**451**	**401**	**100**	**97**	**89**

Table 30: Compiler flags study on KNC. The table summarizes the compiler flags that had most impact on the performance. The flag cf1 is shorthand notation for `-O2 -mmic` whereas cf2 is shorthand notation for cf1 plus `-opt-streaming-stores always -opt-streaming-cache-evict=0` and cf3 is shorthand notation for cf2 plus `-fimf-precision=low -fimf-domain-exclusion=15`. Green is used to highlight the components that are most affected by the additional compiler flags.

4.3 Imbalance

This section will reveal the imbalance numbers for the default decomposition run on KNC described above. Moreover, it will cross-compare these numbers with the ones attained for the fastest sliced IVB run from the Intel Xeon processor section, cf. IVB numbers from table 20. We have repeated these numbers in table 31 that also shows the KNC numbers.

Component	**IVB** [s]	**KNC** [s]	**IVB** [%]	**KNC** [%]
advection	0.0	0.0	0.0	0.0
deform	0.1	0.0	0.4	0.5
uvterm	1.0	2.3	7.2	18.3
momeqs	3.3	4.5	9.7	8.8
turbulence	2.2	7.7	7.1	14.5
vdiff	0.2	1.3	2.6	7.7
diffusion	0.0	0.0	0.1	0.1
density	0.7	0.7	3.6	3.9
sumuvwi	0.7	0.6	3.4	4.7
bcldens	0.6	0.7	4.1	5.4
masseqs	0.0	0.0	0.1	0.3
tflow_up	0.7	0.5	2.4	2.9
smag	0.9	0.4	7.9	4.9

Table 31: Imbalance numbers for the KNC run with respect to the default decomposition and from the fastest sliced version attainable on IVB, cf. table 20. Yellow is used to highlight the components that have internal barriers. Red is used to highlight the components for which the imbalance grows significantly from IVB to KNC.

5 Intel Xeon phi, KNL

In this section we present out-of-the box performance on KNL and we will cross-compare KNL performance with BDW performance and with KNC performance.

5.1 Default decomposition

Table 32 cross-compares BDW performance with KNL performance. Assuming that BDW performance is reasonable then we conclude that our KNL performance could be better. That is, if we compare the attained 65% relative KNL/BDW performance to the similar relative HPL and Triad performance of 69% and 29%, respectively, we observe that performance gain is much closer to the HPL gain than to the Triad gain by moving from BDW to KNL (a similar feature was observed from IVB to KNC). This is primarily due to a few components performing relatively poor. Thus, if we focus on top-5 (marked in red in table 32) we have: `vdiff`, `turbulence`, `momeqs`, `bcldens` and `density`, that is, three components that have the non-SIMD friendly trisolver as a major component and two that are really flop intensive but that also have reductions.

The first three account for 26% of the total time on KNL and this finding is no real surprise since non-SIMD friendly components do not match an architecture with 2 VPU units capable of doing 512-bit SIMD instructions. If we in the future manage to find a proper solution for the trisolver issue and thereby reduce time by (say) a factor of 2 for these three components, then the timeloop would take 158 seconds and BDW versus KNL performance ratio would then be 57% which would better match our expected target than the attained 65% in table 32.

The latter two that account for 7% of the total time on KNL, on the other hand, surprises us. The performance attained in `bcldens` and `density` does not follow the HPL patterns from table 6 at all, so these two components do not show the expected performance. This needs to be investigated further; we cannot explain this issue at present.

Table 33 extends table 29 for KNL too, and one notes that KNL differs slightly from the other 4 in the sense that there is a significant contribution besides the 13 usual components that needs to be investigated in order to

Component	**BDW [s]**	**BDW$_S$ [s]**	**KNL [s]**	**BDW [%]**	**BDW$_S$ [%]**	**KNL [%]**
advection	127	103	66	100	81	52
deform	10	9	3	100	86	33
uvterm	13	9	5	100	72	38
momeqs	23	21	22	100	88	94
turbulence	16	16	20	100	97	123
vdiff	4	4	6	100	97	151
diffusion	12	9	5	100	74	47
density	10	9	8	100	95	78
sumuvwi	15	15	5	100	99	35
bcldens	7	7	6	100	100	91
masseqs	7	6	4	100	82	53
tflow_up	21	19	8	100	90	38
smag	8	7	4	100	91	46
timeloop	**279**	**239**	**182**	**100**	**86**	**65**

Table 32: Worst case thread timings attained when running with a single MPI tasks and all available threads on the node, 2S BDW versus 1 KNL socket. The BDW and the KNL columns are from the default decomposition, whereas the BDW$_S$ columns are from the best attainable slice decomposition. The bold timeloop line is the timing attained around a serial block including all barriers and smaller components within the timeloop whereas the remaining timers are all from within a threaded context and represent the worst-case timing among all the threads. It is important to stress that one should *not* expect that the worst-case timings sum to the timeloop timings simply because they are worst-case timings and also because we have left out some tiny blocks because they do not contribute to the overall picture. Red is used to mark the components that does not perform so well on KNL, and green is used to mark the fastest platform when looking at the whole timeloop.

explain the overall timeloop performance. It appears that the additional component is the barriers in the hydrodynamics module which accounts for 11 seconds, and including these 11 seconds into the pool we again account for 95% of the total timeloop time.

Component	**IVB**	**HSW**	**BDW**	**KNC**	**KNL**
advection	43	44	45	39	36
deform	3	4	4	2	2
uvterm	4	4	5	3	3
momeqs	8	9	8	13	12
turbulence	7	7	6	13	11
vdiff	2	2	1	4	3
diffusion	4	4	4	3	3
density	4	4	3	5	4
sumuvwi	5	5	5	3	3
bcldens	3	2	2	3	3
masseqs	2	2	3	2	2
tflow_up	7	7	7	5	4
smag	3	3	3	2	2
All	**96**	**97**	**95**	**96**	**89**

Table 33: Timings in percent relative to the **timeloop** timings in table 8, 28 and 32. It reveals the fraction of time within the timeloop that we have not accounted for (3%-5%) in the first four columns and 11% in the KNL column by focusing solely on the 13 components. Red is used to highlight the fact that the 13 components no longer give a proper explanation of the timeloop.

5.2 Compiler flags

This subsection is devoted to the choice of compiler flags on KNL. We have tried many combinations of compiler flags and the conclusion of this study was that a couple of flags do contribute to the performance. We found that `-O2 -xMIC-AVX512` served as a better baseline than `-O3 -xMIC-AVX512` and tried several combinations of flags from table 15. It turned out that the only flag that had a significant impact for a couple of subroutines was

```
-fimf-precision=low -fimf-domain-exclusion=15
```

Table 34 summarizes the findings of this study. One observes that SIMD is still important (i.e. a factor of 1.8) but not quite as important as it was on KNC (factor 2.6). Moreover, default flag `-O2 -xMIC-AVX512` does give quite good performance.

Component	**cf1** [s]	**cf2** [s]	**cf3** [s]	**cf1** [%]	**cf2** [%]	**cf3** [%]
advection	75	78	127	100	103	170
deform	3	3	5	100	98	158
uvterm	5	5	6	100	95	120
momeqs	22	19	38	100	84	174
turbulence	20	18	38	100	90	191
vdiff	6	5	6	100	78	96
diffusion	5	6	9	100	110	163
density	8	8	26	100	99	333
sumuvwi	5	5	6	100	102	122
bcldens	6	6	22	100	94	350
masseqs	5	4	5	100	76	102
tflow_up	8	8	12	100	103	144
smag	3	4	6	100	106	179
timeloop	**190**	**183**	**342**	**100**	**96**	**180**

Table 34: Compiler flags study on KNL. The flag cf1 is shorthand notation for `-O2 -xMIC-AVX512` whereas cf2 is shorthand notation for cf1 plus `-fimf-precision=low -fimf-domain-exclusion=15`. Finally, cf3 is shorthand notation for cf2 plus `-no-vec`.

5.3 KNC versus KNL

In this section we will cross-compare KNC performance with KNL performance and see what we can learn from this.

Table 35 cross-compares KNC performance with KNL performance and it shows that KNC is an excellent proxy for KNL performance in the sense that all components show consistent timings in the range from $\frac{1}{3}$ to $\frac{1}{2}$. The KNL/KNC ratio for stream triad and HPL is 41% and 55%, respectively. From table 35 we see that most components, as expected, are close to the triad ratio, and we see that some even have a lower percentage. We even see that flop bound components which have a high fraction of serial computations (i.e. the trisolver) perform way better than the HPL ratio predicts, which must be due to the scalar performance is a factor of 3 faster on KNL than on KNC. Thus, KNL performance coincides with our expectations gained from our KNC runs.

We may use the study given in table 35 to derive a naive, empirical model for estimation of KNL performance when the KNC performance is known. The timings for BW bound components are simply weighted by the triad KNL/KNC ratio of 0.40. The timings for flop bound components are weighted by the HPL KNL/KNC ratio of 0.56 except for those flop bound components that contain a significant amount of serial compute work; for these the serial fraction is divided by 3. From our experience with HBM we have learned (not shown) that the fraction of serial work in `momeqs`, `turbulence`, and `vdiff` may very roughly be estimated to $\frac{1}{3}$. Thus, from the KNC component timings (in seconds) we find:

```
    0.40*(156+8+12+11+12+8+18+7)
+   0.56*((51+53+16)*2/3 + (18+14)) + (51+53+16)*1/3/3 = 169
```

which is pretty close to the measured KNL timings for the components:

```
    66+3+5+22+20+6+5+8+5+6+4+8+4 = 162
```

To account for the remainder, i.e. the timeloop time minus sum of component times, we know from section 5.1 that these are primarily due to explicit OpenMP barriers, and assuming that the cost of these barriers is roughly proportional to the actual number of OpenMP threads we may extrapolate the remainder from KNC to KNL as

```
    (401 - (156+8+12+11+12+8+18+7+51+53+16+18+14))*256/240 = 18
```

which is in good agreement with

```
182 - (66+3+5+22+20+6+5+8+5+6+4+8+4) = 20
```

that one gets from the KNL-seconds in table 35. The complete estimate from KNC to KNL is thus

```
169 + 18 = 187
```

Component	**KNC [s]**	**KNL [s]**	**KNC [%]**	**KNL [%]**
advection	156	66	100	42
deform	8	3	100	40
uvterm	12	5	100	40
momeqs	51	22	100	43
turbulence	53	20	100	37
vdiff	16	6	100	36
diffusion	11	5	100	48
density	18	8	100	42
sumuvwi	12	5	100	44
bcldens	14	6	100	43
masseqs	8	4	100	50
tflow_up	18	8	100	44
smag	7	4	100	48
timeloop	**401**	**182**	**100**	**45**

Table 35: Worst case thread timing attained when running with a single MPI tasks and all available threads on a KNC card versus a KNL socket. Times are in seconds in the left part of the table. The right part of the table lists the timings in percent relative to KNC. The bold timeloop line is the timing attained around a serial block including all barriers and smaller components within the timeloop whereas the remaining timers are all from within a threaded context and represents the worst-case timing among all the threads. It is important to stress that one should *not* expect that the worst-case timings sum to the timeloop timings simply because they are worst-case timings and also because we have left out some tiny blocks because they do not contribute to the overall picture.

5.4 Sliced decomposition

Table 36 shows that the sliced approach that gave a 12%-14% improvement on the Intel Xeon processors does not improve the performance on KNL at all. This is consistent with similar findings on KNC (not shown). This is an interesting finding that will provide new insights. Note that time even increases by more than a factor 12 at the largest slice count compared to the default decomposition.

Component	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**
advection	66	69	72	84	70	79	76	79	721
deform	3	4	4	4	4	4	3	4	33
uvterm	5	5	5	5	5	6	5	6	56
momeqs	22	24	23	24	24	26	24	32	456
turbulence	20	22	22	22	21	23	21	28	342
vdiff	6	6	6	6	6	6	6	9	134
diffusion	5	5	6	5	5	5	5	5	51
density	8	9	8	8	8	9	9	11	148
sumuvwi	5	5	5	5	5	5	5	7	36
bcldens	6	6	6	7	7	7	7	9	122
masseqs	4	4	4	4	4	4	3	5	37
tflow_up	8	8	8	9	8	9	8	8	51
smag	4	4	4	4	3	4	4	5	50
timeloop	**182**	**189**	**194**	**201**	**194**	**207**	**196**	**227**	**2339**

Table 36: Timings attained using a varying number of slices on KNL. The heading of the column denote the number of slices used. Green indicates the best timing and red the worst.

According to table 1 BB_2 has 331 lines in the W/E direction implying that the 256 threads will roughly get a line each in the default decomposition. Moreover, 4 consecutive threads will (with compact placement) roughly loop through 4 W/E lines and one can imagine how the 4 hardware threads will process these lines in a relatively aligned fashion. This will impose maximum D1/L2 overlap between the threads and since these cache layers are shared this will be good for performance. Once we add the slices we expect that the overlap will decrease since data for threads on the same core will be more separated, and this will show as less good performance. This is our working hypothesis in trying to explain the findings in table 36.

This hypothesis can be justified by conducting the experiments with less threads per core. We do not expect to see the sliced decomposition decrease the performance with 1 thread/core. The results of this study are shown in tables 37-39. With 1 thread/core we do indeed see improved performance by using slices. With 2 and 3 threads/core the variation in performance with number of slices is relatively modest and not at all so wild as with 4 threads/core. This confirms of hypothesis excellently.

Component	**1**	**4**	**8**	**16**	**32**	**64**
advection	235	230	217	212	213	211
deform	9	10	10	10	9	9
uvterm	17	19	20	19	18	17
momeqs	72	74	78	76	75	75
turbulence	66	68	70	65	66	71
vdiff	17	17	18	18	18	19
diffusion	16	15	15	14	14	14
density	26	27	28	27	27	27
sumuvwi	12	13	13	12	12	13
bcldens	21	22	23	22	22	22
masseqs	10	11	11	10	10	10
tflow_up	24	23	23	22	22	22
smag	10	10	11	10	10	10
timeloop	**546**	**547**	**546**	**526**	**527**	**528**

Table 37: Timings attained using a varying number of slices with 1 hardware thread per core on KNL. The heading of the column denotes the number of slices used. Green indicates the best timing.

Component	**1**	**2**	**4**	**8**	**32**	**64**	**128**
advection	108	124	115	112	123	109	130
deform	5	5	5	5	5	5	6
uvterm	9	10	10	10	10	9	11
momeqs	38	39	39	40	40	38	51
turbulence	33	38	37	36	37	37	50
vdiff	9	9	9	9	9	9	14
diffusion	7	8	7	8	8	7	8
density	14	14	14	14	15	14	19
sumuvwi	6	7	7	7	7	7	10
bcldens	11	12	11	12	12	11	15
masseqs	5	6	6	6	5	5	7
tflow_up	11	12	12	11	12	12	13
smag	5	6	6	5	5	5	7
timeloop	**267**	**295**	**283**	**281**	**296**	**274**	**347**

Table 38: Timings attained using a varying number of slices with 2 hardware threads per core on KNL. The heading of the column denote the number of slices used. Green indicates the best timing.

Component	**1**	**2**	**4**	**8**	**16**	**32**	**64**
advection	83	79	78	77	79	79	84
deform	4	4	4	4	4	4	3
uvterm	6	7	7	7	7	8	7
momeqs	25	27	27	28	27	29	26
turbulence	24	26	25	25	25	26	25
vdiff	6	6	6	6	6	6	6
diffusion	6	5	5	5	5	5	5
density	9	10	10	10	10	11	10
sumuvwi	5	5	5	5	5	5	5
bcldens	8	8	8	8	8	8	8
masseqs	4	4	4	4	4	4	4
tflow_up	9	8	8	8	8	8	9
smag	4	4	4	4	4	4	4
timeloop	**197**	**200**	**197**	**198**	**199**	**204**	**202**

Table 39: Timings attained using a varying number of slices with 3 hardware threads per core on KNL. The heading of the column denote the number of slices used. Green indicates the best timing.

6 Summary

In this section, we will briefly summarize the initial findings and present the performance results.

6.1 Findings on Intel Xeon processor

SIMD. The SIMD instruction sets Intel `AVX` and Intel `AVX2` will certainly improve flop intensive kernels, but what kind of gains can one expect on larger legacy codes? We set out to investigate the effects of Intel `AVX` and Intel `AVX2` when it comes to HBM. As shown in table 16-17 we do see a huge improvement for some individual components (e.g. `density`, `bcldens`, `turbulence`, `momeqs`) that are not already attaining the peak BW and thus completely bound by that. We even see improvements for the components that are close to attaining the peak BW without the additional pressure that one can put on the memory system by using Intel `AVX` and Intel `AVX2` (`advection`). All in all we see almost 20% performance boost on HSW due to SIMD alone.

Fused multiply and add. As expected, `FMA3` certainly improves performance for the most flop intensive components `density` and `bcldens`. It is, however, somewhat disappointing to us to see the relatively small improvement for `turbulence` when using `FMA3` since it contains some quite flop intensive parts, including similar computations as those in `density` and `bcldens`; this is likely due to this component being quite bound by the serial trisolver and reduction loops, and we will need to investigate this further at a later stage.

Hybrid versus single task. We found that we were *not* able to improve performance by using MPI instead of OpenMP on the node, i.e. we showed that whatever we could attain with MPI was attainable by OpenMP solely and actually single-task performance was always slightly faster than the cases where we traded some OpenMP threads with MPI tasks. It should be stressed that this finding does not prove that OpenMP is better than MPI. All it shows is that with identical decomposition we have not been able to beat the single-task performance and among all decompositions, single-task performance was also the fastest. The MPI halo swaps in HBM are done using `isend` and `irecv` and all threads participate in the communication as well as the wrapping and unwrapping of packages. The abstraction that we use a distributed memory model (MPI) on a single node does add overhead

and so does the additional thread barrier needed for the MPI halo swaps. Thus, it definitely does *not* come as a huge surprise to us that we can indeed measure this overhead. Having said that, we are pleased to find the overhead is relatively small ... as long as we do not use all available hardware threads for MPI tasks but leave some for OpenMP threads; using all the hardware threads for MPI is close to a catastrophe for performance.

HyperThreading. We observed that the importance of hyperthreads decreased going from IVB to BDW and our hypothesis is that the shared cache components (D1, I1, L2) are already pushed to their limits with a single thread per core. Actually, table 23 shows that the components that are highly BW bound are hurt by running with HyperThreading and this confirms our hypothesis. The components that are highly bound on frequency and flops, on the other hand, will benefit from HyperThreading until they become BW bound too. It will be interesting to follow this trend on the Intel Xeon Phi processor-based systems.

Decomposition strategy. We are not surprised to find that decreasing the perimeter of water columns belonging to each thread will improve the performance since decreasing the perimeter means less communication (cache coherence) between the threads. It was interesting to observe that we can gain 12%-14% performance improvement just by introducing the sliced approach.

Imbalance. The overall idea behind the imbalance study was to ensure that we did not loose too many seconds due to a poor balance of the problem and to find the components that would need extra attention with respect to balancing. We found that indeed there is room for improvements here. The most striking finding is that imbalance is (as seen over all components) most severe where we have significant scalar portions, i.e. in components that contains the tri - solver and reduction - type loops and those that are very flop intensive. This means, that the implementation is more sensitive to the distribution of column lengths in the decomposition rather than to the distribution of number of wet points. Imbalance does not seem to become worse in components that have a lot of neighbor to neighbor communication.

Frequency and power. The study showed that some components are very sensitive to changing the frequency while others are virtually unaffected. The most sensitive components are also those which *a priori* were classified as compute bound components, and for these the degradation by

running at the lowest frequency compared to the default frequency was up 70%. Maybe this can be used as an automatic tool to discriminate compute bound computational components from the pool of all components?

However academically interesting it is to study the performance of HBM at different clock frequencies, table 27 reveals that there is only little gained on the energy budget: Running the full HBM at lower frequency just prolongs the execution time (in seconds) at almost the same rate as the power consumption (in Watt) goes down, such that the energy consumption (in kWh) stays virtually unaffected.

Performance. First off, we will assume that the cores across the different Intel Xeon processor incarnations are identical and we will confirm that we at least get the performance that one should attain given the more cores available on the node. Thus, from a core count perspective we should at least reduce the time by 14% from IVB to HSW, and from IVB to BDW we should at least reduce the time by 33%. The result is shown in table 40.

	IVB	**HSW**	**BDW**
#core	100	86	67
Triad	100	80	67
HPL	100	52	38
HBM Measured	**100**	**77**	**63**

Table 40: Relative performance numbers. The row with measured values refers to the best timings attained for the whole timeloop in HBM. All numbers are in percent [%] relative our reference IVB.

Note that the additional cores indeed contribute to the performance, i.e. the implementation appears to scale with the number of cores, or actually better. The measured HBM scaling with hardware is somewhere between that of triad and HPL, closer to the triad scaling. The fact that the implementation is not solely BW bound but has flop bounds components too allows us to improve performance beyond what one could expect from the stream triad numbers. Still, the majority of the code is mostly BW bound so we do not get close to the improvements that one achieves with HPL. These results coincide with our intuitive expectations.

If we focus on what matters most to the end users, namely the total runtime for the application, we see that the performance gains more or less follows the performance gains that are attainable by the stream triad benchmark, i.e. 20% from IVB to HSW and yet another 20% from HSW to BDW. Thus, with a well-established reference for the most time-consuming part this is the realistic expectations that one could hope for without any further code optimization efforts.

6.2 Intel Xeon processor performance results

Now it is time to convert the best attained *time-to-solution* (timeloop took) into *nodes-needed-for-production* and *kilowatts-required-for-production* results. This summary is shown in table 41 and numbers refer to this performance study, i.e. refer to revision `16049` of the HBM code.

BB2nm	**IVB**	**HSW**	**BDW**
Initalization [s]	2	2	2
Timeloop [s]	386	299	239
Power CPU [watt]	165	264	208
Power Mem [watt]	7	29	57
Power System [watt]	641	734	447
Energy [kwh]	1.37	1.24	0.28
Nodes	3	2	2
FD [/hour]	7.0	6.0	7.4

Table 41: This is based on the 6H timings and simple extrapolations. The forecast days (FD) per hour is based on the integral part of the nodes required to fulfill the minimum requirement of doing a single forecast day in less than 10 minutes.

Figure 7 summarizes the timings we have attained on the 3 Intel Xeon processor-based architectures that we have studied in this paper, and compares to what it takes to be production ready from a time-to-solution viewpoint. The 8.5 minutes or 510 seconds stems from the Next Generation Global Prediction System (NGGPS)[8] project in the USA whereas the 240 FCdays/day or 6 minutes per FCday is the most strict requirement stated

[8] https://www2.cisl.ucar.edu/sites/default/files/Michalakes_Slides.pdf

at ECMWF[9]. Figure 8 shows the watt-hours required to run the setup using the 4 different architectures.

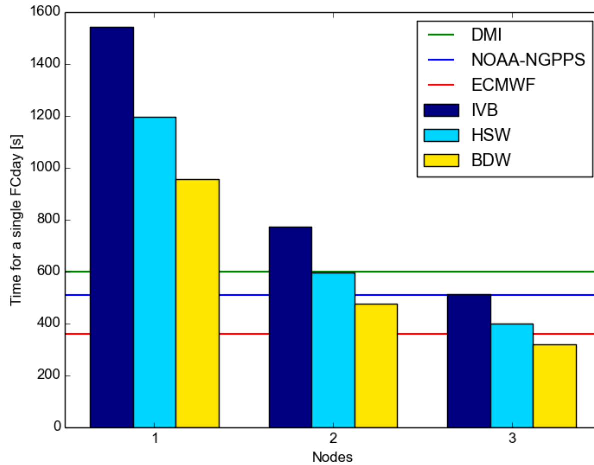

Figure 7: Timings attained on the 3 Intel® Xeon® processor-based architectures. The three horizontal lines mark three different definitions of what it takes to be a production run from a time-to-solution viewpoint.

6.3 Findings on Intel Xeon Phi processor

The main conclusions with respect to issues found may be stated as follows: The three issues described in section 5.1, i.e. trisolver, disappointing performance of flop intensive components, and increased barrier costs, plus the decomposition issue described in section 5.4, should be targets of further investigations in near future since understanding and fixing these holds the key for unlocking the true performance potential of HBM on hardware such as KNL.

Trisolver performance. We understand the issue with the trisolver, and we believe we can fix this by implementing a more SIMD-friendly version that vectorizes across the columns. We do, however, not know yet if this fix is sufficient to bring the KNC and KNL performance into the expected ball

[9] http://www.ecmwf.int/sites/default/files/elibrary/2014/13647-ecmwf-scalability-programme.pdf

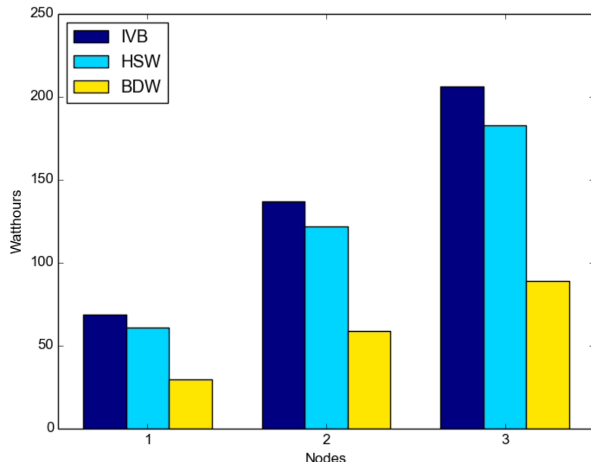

Figure 8: Watt-hours required to run the setup using the 3 different Intel® Xeon® processor-based architectures that we have investigated in this paper.

park.

FLOP performance. The issue on poor performance of flop intensive components is not understood fully yet. Full compute performance requires that all available hardware threads participate at full speed each. We have seen that is this hard to achieve in practice. We expect that this may be one reason for troubles we see, but further investigation should clarify this.

Barrier costs. We observe increased barrier costs on KNL (and KNC). Of course, one would intuitively expect that the involved joins-forks could be more costly the larger the thread count. It seems that this barrier cost is $\mathcal{O}(N_T)$ with N_T being the number of threads, and this cost become significant for large N_T. We should therefore be more careful in our implementation and we should try to arrange the needed barriers more intelligently. One way towards improvement is to replace `OMP BARRIER` which synchronizes everything by a synchronization using `OMP FLUSH` of only the needed data.

Decomposition. Finally, we need to find a better way to do thread decomposition on KNC and KNL (or any platform with more hardware threads

one each core for that matter; the issue is just more pronounced the larger the hardware threads count on each core). The decomposition need to take the core access of data into account, not the individual threads. One potential issue is cache re-usage which seems to be better utilized with the default decomposition than with the sliced decomposition. This might require data permutation (e.g. space filling curves) such that data columns are arranged on the core in a way equally suitable for all threads on that core.

6.4 Updated results

The entire report was done as a performance study of revision `16049` of the trunk. Before completing this report, we re-ran the complete model with the most recent release candidate of the code, namely trunk revision `17154`. Some of the findings from the present paper has been used to improve the performance of the model going from revision `16049` to revision `17154`. Findings of these runs are summarized in the plots 9-12 and table 42. It should be mentioned that all power and performance measurements done in this section were done with `Turbo Disabled` to ensure consistency. It should also be stressed that the KNL runs were done with not only with the 7210 variant that we have used during the present performance study but also using the 7250 variant of KNL.

BB2nm	**BDW**	**KNL**
Timeloop [s]	236	135
Speedup	1.00	1.75
Power System [Watt]	423	372
Speedup (Perf/Watt)	1.00	1.95

Table 42: Note that the **KNL** used in this table is the 7250 variant, and *not 7210* which we used for the performance study in this paper.

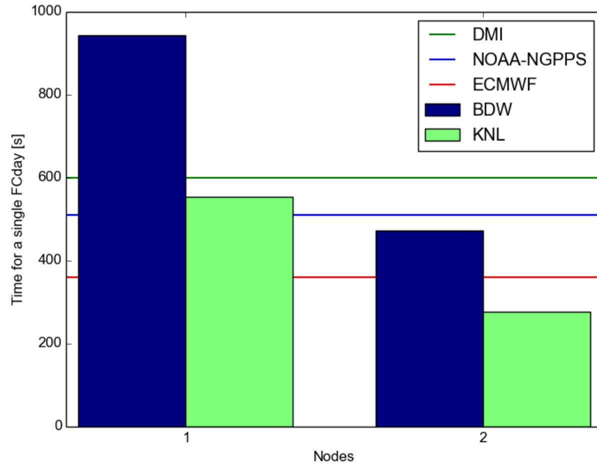

Figure 9: Timings attained on the 2 architectures. The three horizontal lines mark three different definitions of what it takes to be a production run from a time-to-solution viewpoint. A single KNL node is sufficient to meet the DMI local 10 minutes (600 seconds) requirement for a single FD and almost sufficient to meet the NGPPS 8.5 minutes requirement.

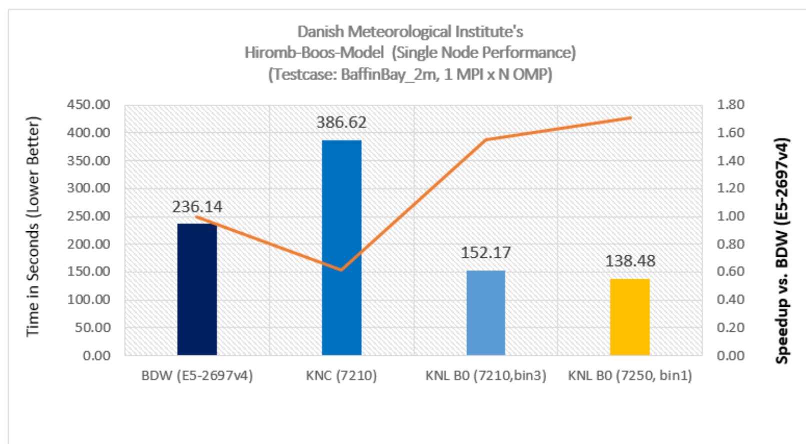

Figure 10: Updated timings attained using the most recent version of the HBM code on BDW, KNC and two KNLs.

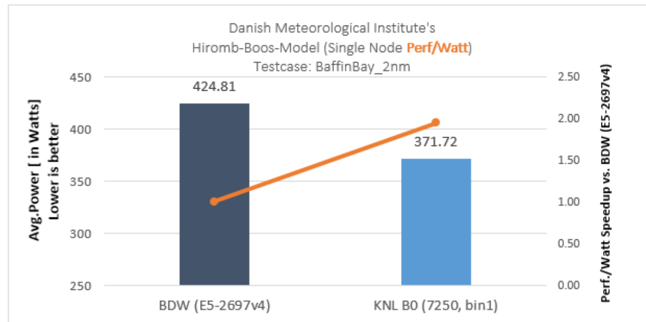

Figure 11: Cross-comparing performance efficiency.

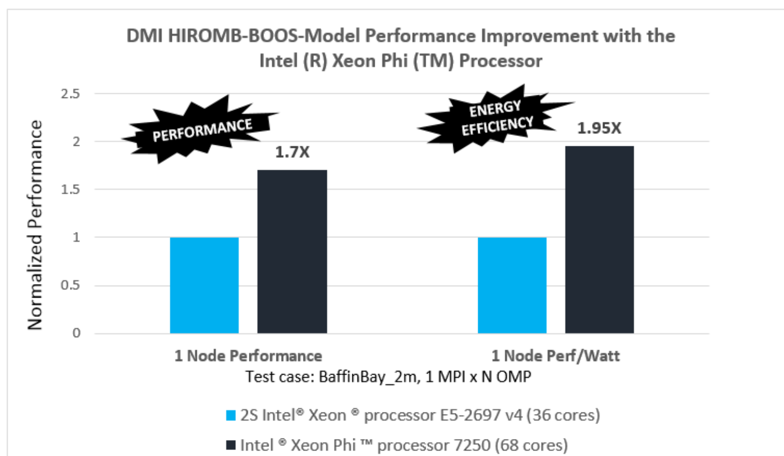

Figure 12: Overall performance summary.

A Appendix: Description of the timeloop components

In this appendix we give a brief summary of the functionality and compuational characteristics of each of the 13 components.

The short name given in italic for each component is the individual identifier used throughout this paper.

A.1 Tracer Advection (*advection*)

The advection is a rather involved component which implement a TVD scheme and it is the *only* truly full 3D component of the entire model, thus a need to look at neighbors in all directions though not at the same time. It computes flux on and transport through the 6 faces on the grid cell with limiters. There are lots of conditional operations based on the flow direction, i.e. control dependency in the vertical innermost `k`-loops. Composed of linear equations, short operator length ($\max \pm 1$), explicit FD scheme. The math in the `k`-loops is relatively simple, almost entirely short-latency flop operations. Short operator length was maintained through split of successive operator into separate components with appropriate `OMP BARRIER`s between. A typical loop in this part of the code will need to do read-only on two neighbors grid-cells (east-west, north-south, up-down) when dealing with actual grid point.

This component constitutes a self-contained collection of a dozen separated operator components with appropriate `OMP BARRIER`s and we therefore expect no imbalance issues for this component taken as a whole though this apparently well-balanced component is really just hiding balancing problems of the individual components.

Accounting for more than 40% of the total run time, this component is a natural candidate for doing optimization and it was heavily studied in [4]. Up to three nested levels of IF-branches (branching on flow direction) were replaced by arithmetic selection involving min,max,sgn instead, yielding a flop count up to 8 times more but in a much more SIMD-friendly way allowing for better streaming of data. Later we have also succeed at hiding most of the `div` operations by storing the reciprocal divisor and use `mul` instead.

The flops involved thus include mostly ordinary operations (`add, sub,`

`mul, abs, min, max, sgn`) plus only a few instances of long-latency `div` operations.

A.2 Horizontal Tracer Diffusion (*$diffusion$*)

Extracts coefficients, set up and solve 2D horizontal tracer diffusion equations by an explicit FD scheme with short operator length (± 1), i.e. a plain CSFT scheme with build-in algebraic stabilizer, limiting the diffusion numerical coefficient by relaxation.

Wrt grid-point neighbor communication this component is looking to the side, both in e-w directions and in n-s directions.

The flops involved are simple operations (`add, sub, mul`) and long-latency `div` operations.

A.3 Vertical Diffusion (*$vdiff$*)

Extract coefficients, set up and solve tracer 1D vertical diffusion equation. It uses an implicit FD scheme with operator length equal to the local column length. It solves the equations by using a tri-diagonal solver (BABE algorithm on two simultaneous sets of equations) and thus there are flow dependencies.

Grid-point neighbor communication is only up/down in the column, not even R/O to the sides.

The flops involved are simple operations (`add, sub, mul`) and long-latency `div` operations.

A.4 Density (*$density$*)

First, a sanity check is performed on salinity s and temperature t followed by a `OMP BARRIER`. Then, calculate the temperature, salinity and pressure dependency in the density, ρ, which is found via UNESCO equation of state of sea water, which may be expressed as $P * Q/(Q - p)$. Higher order polynomial $P(s, s', t)$ and $Q(s, s', t, p)$ are in variables s, s', t, p where $s' = s\sqrt{s}$ and p is hydrostatic pressure. There is flow dependency in `k`-loop when calculating p (i.e. reduction type).

Grid-point neighbor communication is only up/down in the column, not even R/O to the sides.

The flops involved include both ordinary operations (`add, sub, mul, max, min`) but also a few long-latency operations (`div, sqrt`).

A.5 Turbulence and Mixing (*turbulence*)

The turbulence model code solves a coupled system of two partial differential equations for vertical transport (vertical diffusion) of turbulent kinetic energy and frequency with non-linear source/sink terms requiring an implicit scheme both with respect to the FD scheme and with respect to the handling of the sink terms. Thus, the operator length equals the column length. The problem is further complicated by compute intense algebraic stability functions which determine the vertical diffusion coefficients and thereby pose some severe non-linear stability and realisability issues. The setup of the two tri-diagonal systems is indeed very flop intensive but once the systems are set up, the solution is found using the non-SIMD friendly Thomas algorithm.

As to the grid-point communication, the turbulence model is a pure vertical process but it uses horizontal centering of the forcing, that is, of wind and current. This implies that there is no explicit nor implicit column-to-column communication of the turbulence variables, not even read-only. The most important components in the current implementation is the computation of $\partial\rho/\partial s$ and $\partial\rho/\partial t$ where ρ is given by the UNESCO equation of state (see above for *density* and below for *bcldens*) but in this case we do the partial differentiation analytically and implement those expressions in the code. There are also some tricky reduction components which we have tried to separate out of the loops leaving the remaining part better suited for SIMD-vectorization.

The flops involved in the turbulence model code include both short-cycle operations but also long-latency operations. One major performance bottleneck in this component is the use of the non-SIMD implementation of the tridiagonal solver.

We expect that this component will scale well and that it will benefit from hyper-threading due to the lack of communication, the high flop intensity and the non-optimal tridiagonal solver.

A.6 Momentum equations (*momeqs*)

It solves equations of motion (or momentum equations), two coupled non-linear PDEs which are suitably linearized and time-centered to enable a split into two non-coupled semi-implicit equations. Setup and solve a tridiagonal system to find `un, vn`. Linear as well as non-linear terms. Implicit FD in the vertical direction, explicit FD in the horizontal direction. Short operator length in horizontal (± 1) but the vertical operator length equals the local column length.

Important components in the current implementation:
Setup of coefficient vertically with flow (and anti ?) loop dependencies.
(Setup of Smagorinsky terms has been pushed to its own component, see below).
The tridiagonal solver (BABE).
Control dependency in `k`-loops on horizontal flow direction in the applied vector-upwind scheme for the convective terms.

Summary of grid-point neighbor communication:
A 9 grid point stencil for the (linearized) convective terms with access of all 9 neighbor columns is required to setup the equation.

The flops involved include both ordinary operations but also a long-latency operation (`div`).

A.7 Baroclinic Pressure Gradients (*bcldens*)

Calculates horizontal baroclinic pressure gradients. Density ρ is found via UNESCO equation of state of sea water, see above for *density*. Then, the two horizontal gradients are found by finite difference between two neighbor columns.

Since the pressure levels must be the same in the two neighbor columns, namely at the level of the velocity component between neighbor grid cells and this level is different from the level of the scalar point (used in *density* above) and different from each other point, we must make separate calculations of ρ for each velocity point *and* its neighbor, meaning we can not re-use any columns when going to the next.

We can, however, split the computations a bit by doing most of the sym-

bolic algebra with pen and paper, not in the Fortran code.

There is flow dependency in `k`-loop when calculating the pressure (i.e. reduction type).

The flops involved include both ordinary operations (`add, sub, mul, min`) but also long-latency operations (`div, sqrt`).

A.8 Upwind Advection Scheme ($tflow_up$)

Advection of s and t for use in the frequent updates of the baroclinic pressure gradients (*bcldens*) when the regular s and t update is not sufficient.

Principally the same as the *advection* component, see above, except that this $tflow_up$ component applies a very simple 3D upwind tracer advection scheme which, considering *all* computational aspects, will considerably simplify everything. It is called more frequent than the *advection* component, though, meaning that it enters the timings with a significant time portion.

There is a `OMP BARRIER` between the calculation cell-face fluxes and the tracer updating.

A.9 Time-averaged Fluxes ($sumuvwi$)

Makes time integration (i.e. a sum) of cell face fluxes ($h_x u, h_y v, w$) at the most resent time step so that we can get the time- and cell-face-averaged transports for the tracer advection and other things. Also makes a simple copy of new, most recently updated `un,vn` to the old `uo,vo` to be ready for the next time loop iteration.

That is, lots of load and stores and few flops. No neighbor communication.

A.10 Smagorinsky Terms for Momentum Equations ($uvterms$)

Calculate u- and v-Smagorinsky terms at correctly space-centered points to be ready for direct use in *momeqs*. Takes the eddy viscosity from *smag*.

Simple math (`add, sub, mul`) and lots of horizontal neighbor communication.

A.11 Horizontal Eddy Viscosity (*smag*)

Calculates the horizontal eddy viscosity terms from the Smagorinsky model based on the deformation terms, see (*deform*) below. Consists mainly of square root of the sum of squares of stretch, divergence and shear, but also has to look to the sides for doing a correct space-centering of the shear term at the scalar point.

Simple math (`add, sub, mul`) and long-latency `SQRT`, plus horizontal neighbor communication of shear.

A.12 Deformation Terms (*deform*)

Calculates the deformation terms for use in the Smagorinsky sub-grid scale model, see (*uvterms*) and (*smag*) above, i.e.

$$\begin{aligned} stretch &= \partial u/\partial x - \partial v/\partial y \\ divergence &= \partial u/\partial x + \partial v/\partial y \\ shear &= \partial u/\partial y + \partial v/\partial x \end{aligned}$$

Simple math (`add,sub,mul`) but lots of horizontal neighbor communication.

A.13 Mass Equation (*masseqs*)

Applies the incomprehensibility assumption (i.e. local mass preservation) to obtain the vertical velocity component w and the surface elevation z.

Most critically, this requires two reduction loops over the vertical.

Simple math (`add, sub, mul`) but lots of horizontal RO neighbor communication to read cell face fluxes.

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.

[3] Jacob Weismann Poulsen and Per Berg. Thread scaling with HBM. DMI Technical Report No. 12-20. Technical report, DMI, Copenhagen, 2012.

[4] Jacob Weismann Poulsen, Per Berg, and Karthik Raman. Chapter 3 - better concurrency and simd on hbm. In James Reinders and Jim Jeffers, editors, *High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches*, volume 1, pages 43 – 67. Morgan Kaufmann, Boston, MA, USA, 2015.

Notices

Intel technologies features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. Check with your system manufacturer or retailer or learn more at intel.com.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting `www.intel.com/design/literature.htm`.

Intel, the Intel logo, Intel Xeon Phi, and Intel Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others. 2016 Intel Corporation