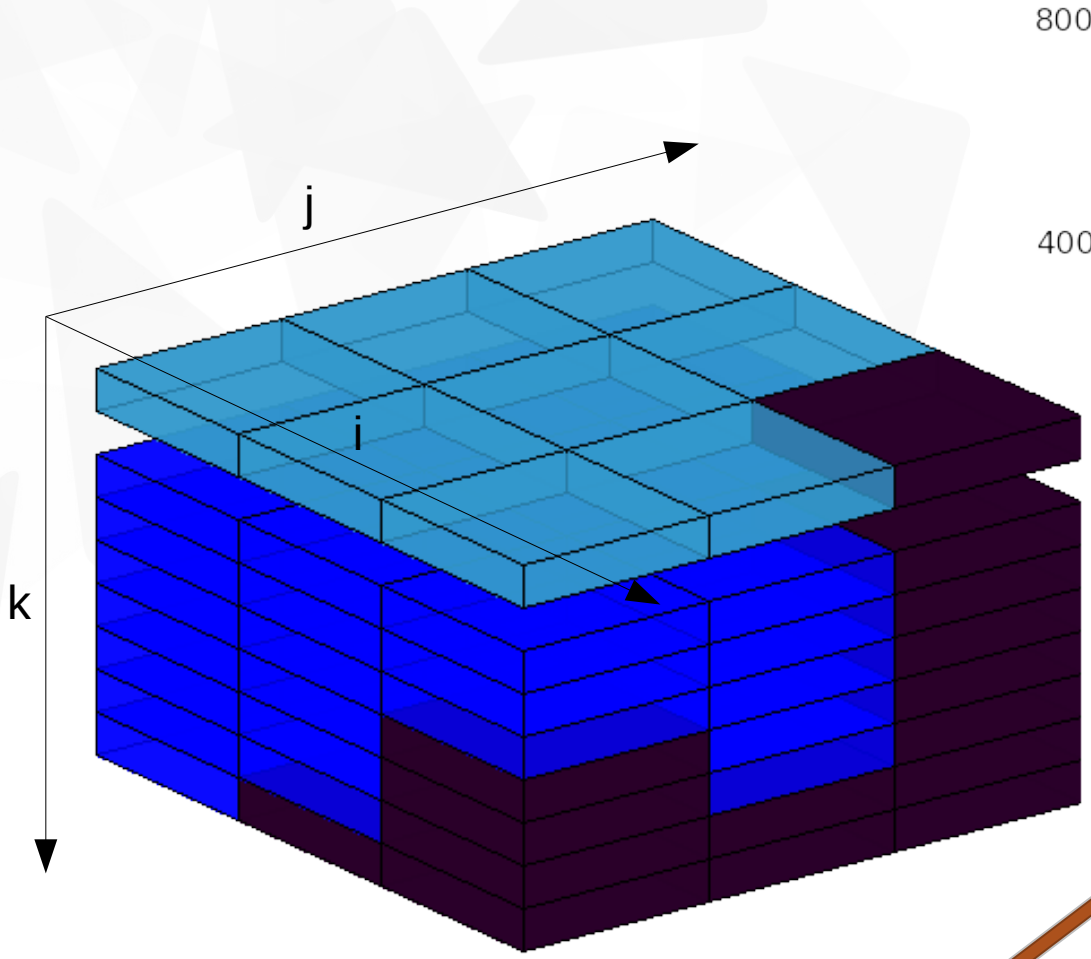
Refactoring for Xeon Phi

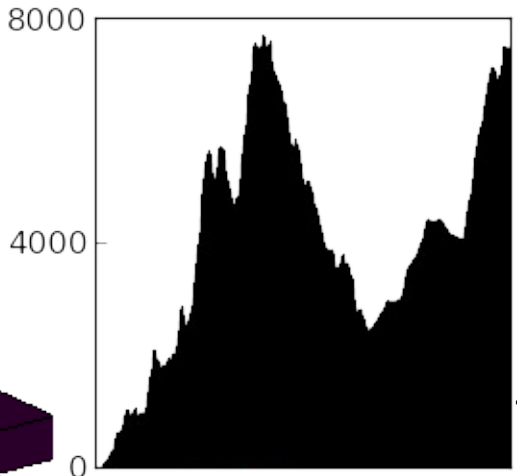
Jacob Weismann Poulsen, DMI, Denmark
Per Berg, DMI, Denmark
Karthik Raman, Intel, USA

Outline

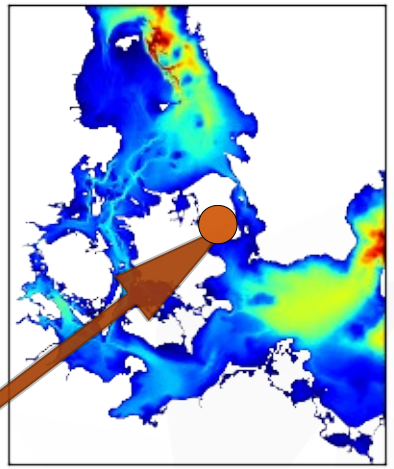
- Data structures
- Node performance
  - Thread parallelization
  - SIMD vectorization
- Performance results

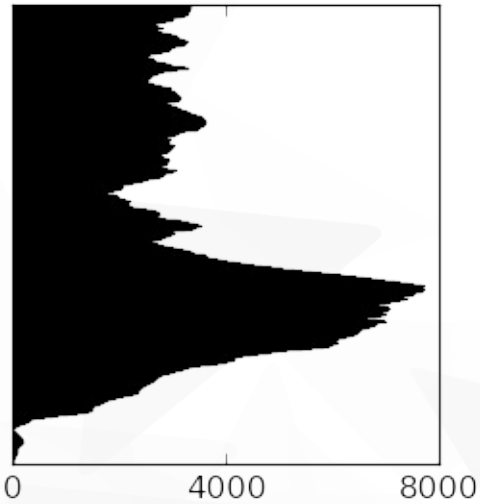
The data is sparse (highly irregular)





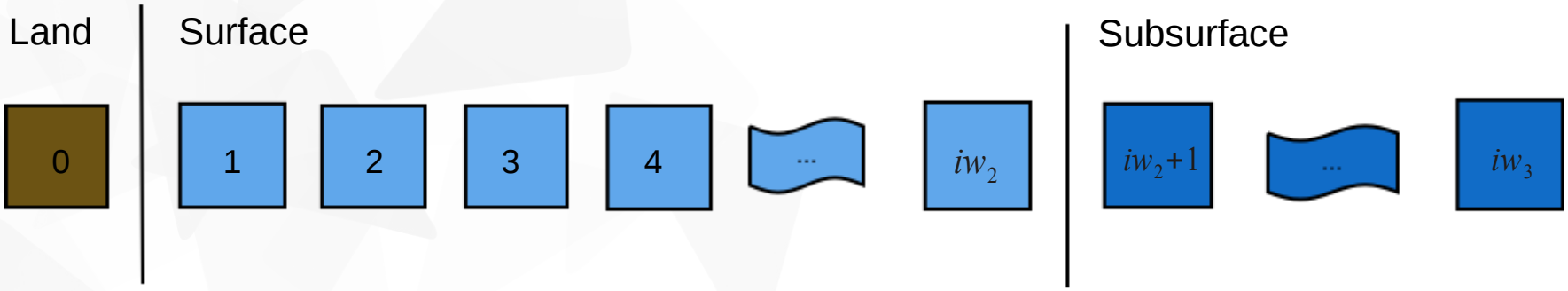
11.1%





YOU ARE HERE

Data layout (serial, indirect addressing)




```
do iw = 1,iw2
  i = ind(1,iw)
  j = ind(2,iw)
  ! all surface wet-points (i,j) reached with stride-1
      ... u(iw) …
enddo
do iw = 1,iw2
  kb = kh(iw)
  if (kb < 2) cycle
  i = ind(1,iw)
  j = ind(2,iw)
  mi0 = mcol(iw)-2
  do k = 2, kb
    ! all subsurface wet-points (k,i,j) are reached with stride-1
    mi = mi0 + k
    ... u(mi) ...
  enddo
enddo
```

Data layout (serial)

- Data layout revisited:
  - Horizontally (unstructured) columns
  - Indirect addressing in the horizontal:

    msrf(0:,0:), ind(1:2,:), mcol(0:), kh(0:)
  - Direct addressing in the vertical
  - (GungHo paper 2013 – similar conclusion for NWP)
- Observation
  - Any enumeration of the surface points and any enumeration of the subsurface points imposes a unique cache pattern (D1, L2, L3, TLB) and some are obviously better than others. Finding the infimum is NP-hard but space-filling-curves could lead to reasonable heuristics. A true challenge to formulate a well-posed problem, though.

Data layout for threads (or tasks + explicit halo)

- Each tread will handle a subinterval of columns:

- Another layout of the columns will impose another threaded layout of data.

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

Thread (and task) load balancing

- Formal definition:

Let $I = \{1, \ldots, m\}$ be the column index set and let $\{w_1, \ldots, w_m\}$ be the weights associated with the individual columns. Let n denote the number of threads/tasks. A disjoint subinterval $I_i = \{[l_i; u_i]\}_{i=1,\ldots,n}$ covering of I induces a cost vector $(c_1, \ldots, c_n)$ with $c_i = \sum_{j=l_i}^{u_i} w_j$. The cost c of the covering is defined as $\max_i c_i$. The balance problem is to find a covering that minimizes c.

- Observation: The NP-hard problem is reduced to the integer partition problem which provides an exact solution within time complexity: $O(m^2n)$.
- Heuristics: Greedy approach or alternating greedy approach with runtime complexity: O(n).
- The weights can be a sum of sub weights while retaining problem complexity!

Thread parallelism - insights

- SPMD based (like MPI) and *not* loop based in order to minimize synchronization. A single openMP block with orphaned barriers surrounding synchronization points such as MPI haloswaps will do (nice side-effect: No explicit scoping).
- On NUMA architectures proper NUMA-layout for all variables is important.
- Consistent loop structures and consistent data layout and usage throughout the whole code.
- Proper balancing is very important at scale (Amdahl). It can be done either offline (exact) or online (heuristic).
- Tuning options for balancing: Linear regression based on profiles, cf. DMI technical report tr12-20.

Refactoring for SIMD

Actually not as simple as it may sound....

SIMD target loops

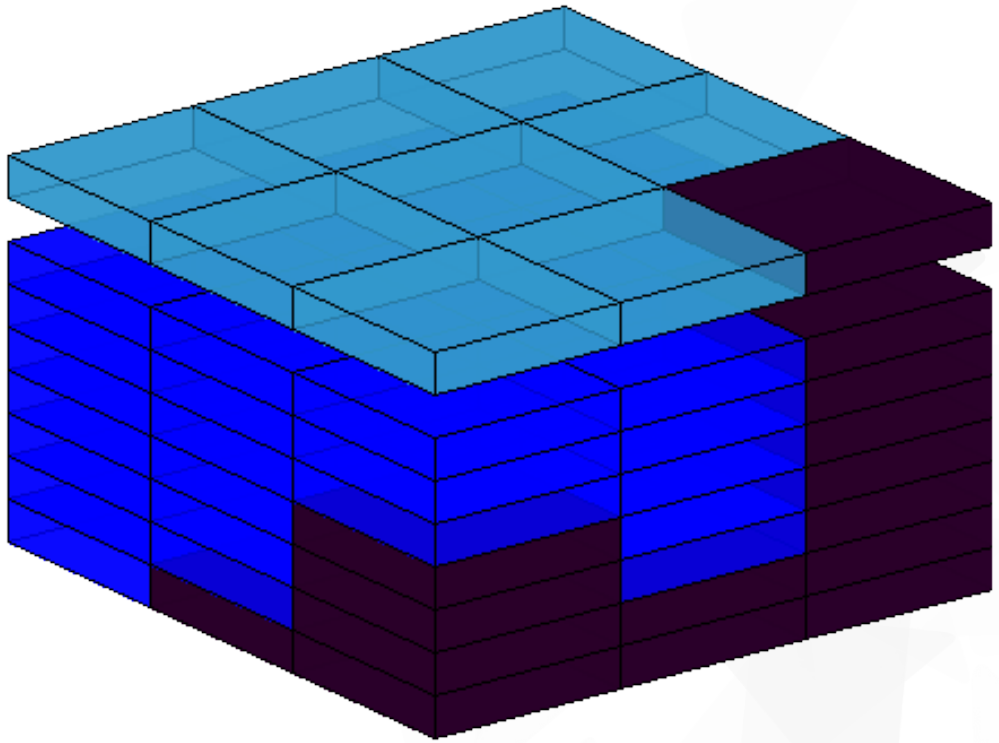
- All loops are structured like this:

```
do iw=       ! horizontal - mpi/openmp parallelization
  do k=      ! vertical   - vectorization
    do ic=   ! innermost loop (in advection) with number of tracers
      …
    enddo
  enddo
enddo
```

- Could vectorize at the iw-level but hardware is not ready. Thus, the aim is to vectorize all the k-loops
- Trivial obstacles to vectorization
  - Indirections
  - Assumed-shape (F2008 contiguous attribute)
  - Branches (min/max/sign)

SIMD target loops

- Design choice for stencil codes: columns one-by-one using work arrays (tune for tripcount) or whole stencil in one go (tune for intensity) plus required remainder loops.
- Refactor strategy using computational intensity (CI) and D1 pressure as the guide lines. High CI is good
- … but not too high; use blocking to reduce pressure on D1, L2,...



Premature abstraction is the root of all evil
(a hands on experience)

Premature abstraction is the root of all evil

- This topic may not coincide with your expectations:
  - I will not talk about how one can loose a leg with OOD (google it, e.g. Mike Acton).
  - I will not talk about how one looses performance by using the HW abstraction that cores within a node have distributed memory (do the math on a piece of paper).
  - …
- Instead I will describe how the most simple HW abstraction (a 2D-array) will result in more than 2x performance loss on Xeon Phi and this should serve as a warning against using even the most simple abstractions without a prior analysis of consequences.

Premature abstraction is the root of all evil

- The design idea was to hold all tracers in one 2D-array and treat all tracers in a similar fashion in one go like this (simplified illustration of the obstacle):

```
1 do k=2,kmax
2   k1 = k+off1
3   k2 = k+off2
4   t(1:nc,k) = t(1:nc,k) + A(k)*(B(1:nc,k1)-B(1:nc,k2))
5 enddo
```

- With dynamic nc the compiler vectorizes nc-loop:

  (4): (col. 7) remark: LOOP WAS VECTORIZED

- With static nc, the compiler vectorizes the k-loop:

  (1): (col 7) remark: LOOP WAS VECTORIZED

Premature abstraction is the root of all evil

- Alas, this is the code generated:
  - AVX (essentially a software gather operation):

```
...
vmovsd  (%r10,%rcx,2), %xmm6
vmovhpd 16(%r10,%rcx,2), %xmm6, %xmm6
vmovsd  32(%r10,%rcx,2), %xmm7
vmovhpd 48(%r10,%rcx,2), %xmm7, %xmm7
vinsertf128 $1, %xmm7, %ymm6, %ymm7
...
```

  - MIC (a hardware gather):

```
...
vgatherdpd (%r13,%zmm2,8), %zmm6{%k5}
...
```

- Especially for the MIC target not what we aimed at (issues on SNB/IVB with 256-bit unaligned load/store so a software gather may not be as bad as it looks to me)

Premature abstraction is the root of all evil

- The obstacle in a nutshell: A static nc (2) implies unrolling:

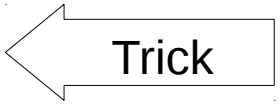
```
do k=1,kmax
  k1 = k+off1
  k2 = k+off2
  t(1,k) = t(1,k) + A(k)*(B(1,k1)-B(1,k2))
  t(2,k) = t(2,k) + A(k)*(B(2,k1)-B(2,k2))
enddo
```

- And the unrolling implies that the optimizer sees the loop as a stride-2 loop but we know better so let's state what the compiler should have done (next slide)
- And no.... interchanging loops is not the solution since it implies a 2x cost on BW and VL is reduced by 1/nc :

The compiler transformation that we hoped for:

- Proper handling of a mix of 2D and 1D (load with nc=2):

zmm1←	t(1,1)	t(2,1)	t(1,2)	t(2,2)	t(1,3)	t(2,3)	t(1,4)	t(2,4)
zmm2←	t(1,5)	t(2,5)	t(1,6)	t(2,6)	t(1,7)	t(2,7)	t(1,8)	t(2,8)
zmm3←	B(1,1+k1)	B(2,1+k1)	B(1,2+k1)	B(2,2+k1)	B(1,3+k1)	B(2,3+k1)	B(1,4+k1)	B(2,4+k1)
zmm4←	B(1,5+k1)	B(2,5+k1)	B(1,6+k1)	B(2,6+k1)	B(1,7+k1)	B(2,7+k1)	B(1,8+k1)	B(2,8+k1)
zmm5←	B(1,1+k2)	B(2,1+k2)	B(1,2+k2)	B(2,2+k2)	B(1,3+k2)	B(2,3+k2)	B(1,4+k2)	B(2,4+k2)
zmm6←	B(1,5+k2)	B(2,5+k2)	B(1,6+k2)	B(2,6+k2)	B(1,7+k2)	B(2,7+k2)	B(1,8+k2)	B(2,8+k2)
zmm7←	A(1)	A(1)	A(2)	A(2)	A(3)	A(3)	A(4)	A(4)
zmm8←	A(5)	A(5)	A(6)	A(6)	A(7)	A(7)	A(8)	A(8)



- Proper handling of a mix of 2D and 1D (arithmetic):

```
zmm9  = zmm1 + zmm7*(zmm3-zmm5)!k=1,4;nc=1,2
zmm10 = zmm2 + zmm8*(zmm4-zmm6)!k=5,8;nc=1,2
```

- But did not get so we need to drop the 2D abstraction if performance matters to us.

Performance numbers

The module for the advection was chosen as a candidate for tunings. A single node run on both IVB and KNC showed that ≈44% of the time was spent here. The time spent on KNC was 3x the time on IVB when we started to investigate this.

Benchmark systems

- Intel Xeon E5-2697 v2 (30Mb cache, 2.70 GHz)
  - Launched Q3, 2013
  - Number of cores/threads on 2 sockets: 24/48
  - DDR3-1600 MHz, 8*8 GB
  - Peak flops (HPL: 543 GF/s, 450 Watt)
  - Peak BW (Stream: 84 GB/s, 408 Watt)

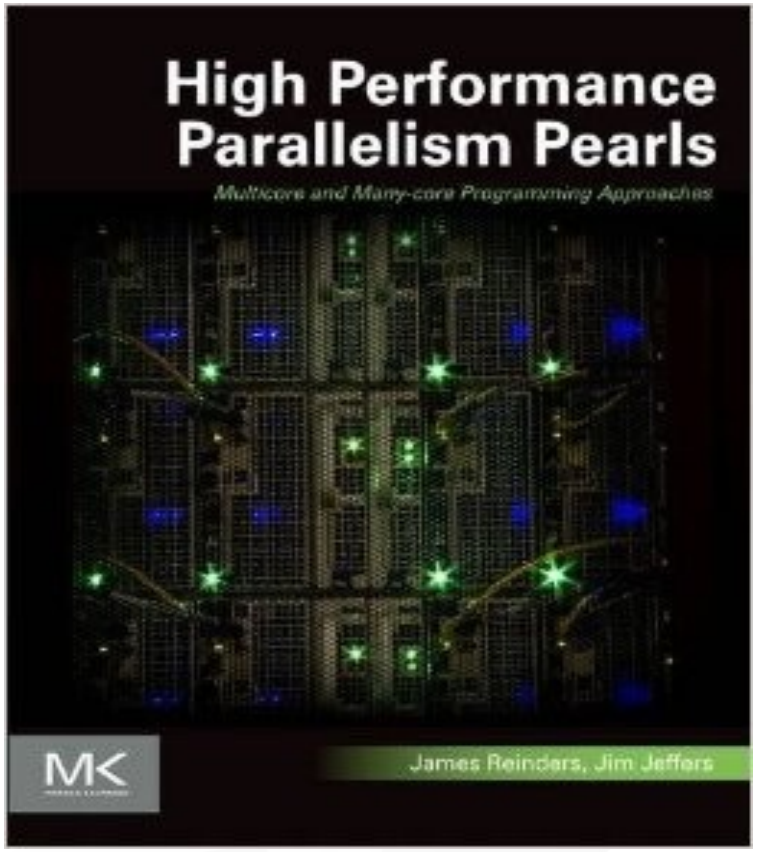
- Intel Xeon Phi 7120A (30.5Mb cache, 1.238 GHz)
  - Launched Q2, 2013
  - Number of cores/threads: 60/240
  - GDDR5, 5.5 GT/s, 16 GB
  - Peak flops (HPL: 999 GF/s, 313 Watt)
  - Peak BW (Stream: 181 GB/s, 283 Watt)

Performance (memory bandwidth bound)

Advection (same algorithm)	2S-IVB 2697v2	KNC C0 7120A
Threads	48	240
Timing [sec]	81	72
Relative timing [%]	100	89
Stream Triad [GB/s]	86	177
Stream Triad [GB/s/watt]	0.21	0.63
BW sustained [GB/sec]	86	155
BW sustained [% peak]	100	88
BW sustained [GB/sec/watt]	0.21	0.55
Vector intensity sustained		7

More information

- The foray is documented in chapter 3 in High Performance Parallelism Pearls (Morgan Kaufmann; ISBN: 978-0128021187).
- Code and testcase is available online: http://lotsofcores.com



- The preparation work is documented in a technical report: http://www.dmi.dk/fileadmin/user_upload/Rapporter/tr12-20.pdf

Acknowledgement

- Michael Greenfield, Intel
- Larry Meadows, Intel
- John Levesque, Cray
- Bill Long, Cray