HBM code modernization

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

- HBM
  - Introduction
  - Data structures and parallelization
  - Performance

- ESCAPE
  - Initial study of a radiation kernel (if time permits)
Introduction

- A 3D ocean circulation model (solves IBVP)
- Basis of DMI's regional production forecasts:
  - storm surge warning in DK
  - MyOcean & Copernicus Baltic MFC
- 2-way dynamical nesting
  - horizontal, vertical, time
  - any number of nesting levels
  - high resolution in regional seas
  - very high resolution in straits
- HBM community: DMI, BSH, MSI, FM1 + third parties (MOU)

<table>
<thead>
<tr>
<th></th>
<th>mmx [N/S]</th>
<th>nmx [E/W]</th>
<th>kmx [layers]</th>
<th>#3d</th>
<th>iw3/#3d [%]</th>
<th>iw2/#2d [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>baltic</td>
<td>720</td>
<td>567</td>
<td>122</td>
<td>49805280</td>
<td>12.5</td>
<td>12.3</td>
</tr>
<tr>
<td>idw</td>
<td>482</td>
<td>396</td>
<td>77</td>
<td>14697144</td>
<td>12.5</td>
<td>10.8</td>
</tr>
<tr>
<td>ns</td>
<td>348</td>
<td>194</td>
<td>50</td>
<td>3375600</td>
<td>25</td>
<td>14.2</td>
</tr>
<tr>
<td>ws</td>
<td>149</td>
<td>154</td>
<td>24</td>
<td>550704</td>
<td>25</td>
<td>18.8</td>
</tr>
</tbody>
</table>
Introduction

- Homogenised production forecast on pan-European scale
- Regional climate setups covering a small domain is less interesting but with this they can do pan-European climate runs at rather high resolution
- Reference: 1nm~1.85 km~1.15 mi

Resolution in this setup:
- Vertical resolution from 1 meter
- Horizontal resolution from ~200 meter to 5.5km

10 two-way nested domains:
- Bosphorus / Dardanelles Straits: ~0.1 n.m.
- Inner Danish waters: ~0.5 n.m.
- Marmara Sea / Gibraltar / Baltic Sea: ~1 n.m.
- North Sea / Shelf / Med.Sea / Black Sea: ~3 n.m.

Points [mill]: #3d:110, iw3:19
Memory [Gb]: 7.2
The data is sparse and highly irregular
Data structures and parallelization
SIMD (no interactions between columns)

Land | Surface | Subsurface
--- | --- | ---
0 | 1 2 3 4 ... \(i_{w_2}\) | \(i_{w_2} + 1\) ... \(i_{w_3}\)

```

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

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

This shows only, the most simple case where there is no interactions between columns.
SIMD (interactions between columns)

\[\text{do } iw = 1,iw2\]
\[kb = kh(iw)\]
\[\text{if }(kb < 2) \text{ cycle}\]
\[i = ind(1,iw)\]
\[j = \text{ind}(2,iw)\]
\[mi0 = mcol(iw) - 2\]
\[me0 = mcol(msrf(i, j+1)) - 2\]
\[mw0 = mcol(msrf(i, j-1)) - 2\]
\[mn0 = mcol(msrf(i-1,j )) - 2\]
\[ms0 = mcol(msrf(i+1,j )) - 2\]

\[kmx = \min(kb,kh(msrf(i,j+1)),kh(msrf(i,j-1)),kh(msrf(i-1,j)),kh(msrf(i+1,j))))\]
\[! \text{ The FAT loop}\]
\[\text{do } k = 2, \ kmx\]
\[\text{mi} = mi0 + k\]
\[\text{me} = me0 + k\]
\[\text{mw} = mw0 + k\]
\[\text{mn} = mn0 + k\]
\[\text{ms} = ms0 + k\]
\[\ldots t(mi) \ldots t(me) \ldots\]
\[\text{enddo}\]

\[! \text{ and a bunch of SKINNY remainder loops}\]
\[\text{do } k=\max(2,kmx+1),\min(kb,kh(msrf(i,j+1))) \! \text{ only mi, me}\]
\[\text{mi} = mi0 + k\]
\[\text{me} = me0 + k\]
\[\ldots\]
\[\text{enddo}\]
\[\ldots\]
\[\text{enddo}\]
Outer parallelization (birds eye)

! initialization
...
call hbm_mpi_init() ! decompose the problem (1,...,iw2 Cs) such that each MPI task
  ! deals with a subset of C locally enumerated by index: 1...iw2_l

call hbm_omp_init() ! decompose the task-local problem (1,...,iw2_l Cs) into threadlocal
  ! sub-chunks and NUMA first-touch according to this decomposition.
...
! Timeloop (set of calls followed by halo-swaps, followed by set of calls, followed...)
$OMP PARALLEL DEFAULT(SHARED)
call foo(...) ; call bar(...) ; ...
call halo_update(...) ! deals with MPI and openMP
call baz(...) ; call quux(...) ; ...
$OMP END PARALLEL

subroutine foo(...)
  ...
call domp_get_domain(kh, 1, iw2_l, n1, nu, idx)
do iw=n1,nu
  i = ind(1,iw)
  j = ind(2,iw)
  ! all task and threadlocal wet-points (:,:, :) are reached here
  ...
endo
end subroutine foo

No more synchronization for threads than for MPI tasks
Thread load balancing

- Each thread will handle a sub **interval** of columns! Another set of sub **sets** will impose another split of the threads.

- 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 $c$.

- The **NP-hard problem** is reduced to the integer partition problem which provides an exact solution within time complexity: $O(m^2n)$. The weights can be a sum of sub weights while retaining problem complexity!

- Heuristics: Greedy approach with runtime complexity: $O(n)$. 
Thread parallelism - summary

- 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*).

- Consistent loop structures and consistent data layout and usage throughout the whole code implying that it is easy to ensure a proper NUMA first-touch.

- The OpenMP standard does not provide us with a clause that allows for advanced balance control so we have to wrap our own. It can be done either offline (exact) or online (heuristic).
Halo swaps (birds eye)

!Timeloop (set of calls followed by halo-swaps, followed by set of calls, followed...)
!$OMP PARALLEL DEFAULT(SHARED)
call foo( ... ); call bar( ... ); ...
call halo_update( ... ) ! deals with MPI and openMP
call baz( ... ); call quux( ... ); ...
!$OMP END PARALLEL

subroutine halo_update(ia,a,...)
  ...
  if (sloppy_halo_omp) then
    !$OMP BARRIER
  else
    ! more involved tracking and swap via !$OMP FLUSH
  endif
  if (mpi_swap2d) then
    call hbm_mpi_halo_update_2d(ia,a)
  else
    call hbm_mpi_halo_update_3d(ia,a)
  endif
end subroutine halo_update

subroutine hbm_mpi_halo_update_2d(ia,a)
  integer(4), intent(in) :: ia
  integer(4), intent(in) :: a(:)
call MPI_neighbor_alltoallv(a,sendc2d(ia)%p,sdispls2d(ia)%p,desttype2d(ia)%p,&
  a,recvc2d(ia)%p,rdispls2d(ia)%p,srctype2d(ia)%p, &
  halo_comm(ia),ierr)
end hbm_mpi_halo_update_2d
Numa initialization

```fortran
... !$OMP PARALLEL DEFAULT(SHARED) ...
call numa_init( ... ) ...
!$OMP END PARALLEL

subroutine numa_init(...) ...
call domp_get_domain(kh, 1, iw2, nl, nu, idx) 
! surface init 
a(nl:nu) = 0.0_8 ...
! depth init 
do iw=nl,nu 
  kb = kh(iw)
  if (kb < 2) cycle
  ml = mcol(iw)
  mu = ml + kb - 2
  a(ml:mu) = 0.0_8 
  ... 
enddo 
end subroutine numa_init
```
Thread parallelism – premature optimization

- Could improve the BW usage by splitting the surface and depth handling. This will improve the temporal locality but may require additional barriers.

- **OMP BARRIERS** may be done with p2p via **OMP FLUSH**. Why don't openMP have an **OMP UPDATE** similar to openACC to reduce cache coherency overhead?

- Manual padding to deal with missing system support at the cacheline level (false sharing) and at the page level (perfect NUMA locality).

- Several heuristics to improve the balancing as mentioned earlier. HOWEVER, we need more performance insights before considering pursuing these ideas further.
Performance
Testcase (timeloop focus)

- Input files provided by:
  Lars Johanson & Jens Murawsky
- No nesting, no IO, no met forcing

Summary:

<table>
<thead>
<tr>
<th>res [m]</th>
<th>[N/S]</th>
<th>[E/W]</th>
<th>[layers]</th>
<th>3d [mill]</th>
<th>iw3 [mill]</th>
<th>dt [sec]</th>
<th>Mem [mb]</th>
</tr>
</thead>
<tbody>
<tr>
<td>3600</td>
<td>565</td>
<td>331</td>
<td>137</td>
<td>25.6</td>
<td>7.1</td>
<td>10</td>
<td>3355</td>
</tr>
<tr>
<td>1800</td>
<td>1130</td>
<td>661</td>
<td>137</td>
<td>102.3</td>
<td>27.8</td>
<td>5</td>
<td>13068</td>
</tr>
<tr>
<td>900</td>
<td>2260</td>
<td>1322</td>
<td>137</td>
<td>409.3</td>
<td>111.4</td>
<td>2.5</td>
<td>52225</td>
</tr>
<tr>
<td>450</td>
<td>4520</td>
<td>2644</td>
<td>137</td>
<td>1637.3</td>
<td>445.4</td>
<td>1.25</td>
<td>206030</td>
</tr>
</tbody>
</table>

#gridpoints is 1.5xN640, but the model still have a “small” memory footprint
https://software.ecmwf.int/wiki/display/FCST/Increase+in+data+volume
Intermezzo: Musings on performance

- Performance = FCT( method , implementation )
- Evaluation of performance: Use generic measures like FCdays/hour and Energy2solution at different node counts and under the use of different nodes.
- Given good scientific performance - all “we” really care about is that time2solution (T2S) and energy2solution (E2S) comply with our requirements
  \[ E2S \sim (\text{Power Draw/Node}) \times (\text{Node Hours}) \times \text{PUE} \]
Performance results (1S KNL vs 2S BDW)

- Are these universal results?
  - 1nm (1800m) setup: T2S is not 1.7x but 1.9x faster
  - Nested production setup (p. 3): T2S is not 1.7x but 1.4x
  - A single KNL node is sufficient to complete 1 forecast day within 10 minutes for both the 2nm BaffinBay setup (p. 16) and our current nested production setup (p. 3).
## Performance summary (1nm setup, single node)

<table>
<thead>
<tr>
<th>Fraction of time</th>
<th>Normalized time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>E5-2697v4@2.3</td>
</tr>
<tr>
<td>advection</td>
<td>43%</td>
</tr>
<tr>
<td>deform</td>
<td>3%</td>
</tr>
<tr>
<td>uvterm</td>
<td>3%</td>
</tr>
<tr>
<td>smag</td>
<td>3%</td>
</tr>
<tr>
<td>momeqs</td>
<td>7%</td>
</tr>
<tr>
<td>turbulence</td>
<td>6%</td>
</tr>
<tr>
<td>vdiff</td>
<td>1%</td>
</tr>
<tr>
<td>diffusion</td>
<td>4%</td>
</tr>
<tr>
<td>density</td>
<td>2%</td>
</tr>
<tr>
<td>sumuvwi</td>
<td>6%</td>
</tr>
<tr>
<td>bcldens</td>
<td>2%</td>
</tr>
<tr>
<td>masseqs</td>
<td>2%</td>
</tr>
<tr>
<td>tflow_up</td>
<td>8%</td>
</tr>
<tr>
<td>timeloop</td>
<td>100%</td>
</tr>
</tbody>
</table>
# Hardware

<table>
<thead>
<tr>
<th>Micro-architecture</th>
<th>IvyBridge</th>
<th>Haswell</th>
<th>Broadwell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>E5-2697v2</td>
<td>E5-2697v3</td>
<td>E5-2697v4</td>
</tr>
<tr>
<td>Released</td>
<td>Q3, 2013</td>
<td>Q3, 2014</td>
<td>Q1, 2016</td>
</tr>
<tr>
<td>Cores/node</td>
<td>24</td>
<td>28</td>
<td>36</td>
</tr>
<tr>
<td>Frequency [GHz]</td>
<td>2.7</td>
<td>2.6</td>
<td>2.3</td>
</tr>
<tr>
<td>#cores [%]</td>
<td>100</td>
<td>117</td>
<td>150</td>
</tr>
<tr>
<td>#cores time [%]</td>
<td>100</td>
<td>85.7</td>
<td>66.7</td>
</tr>
<tr>
<td>HPL [GF/s]</td>
<td>492</td>
<td>949</td>
<td>1236</td>
</tr>
<tr>
<td>HPL efficiency [%]</td>
<td>95</td>
<td>81</td>
<td>93</td>
</tr>
<tr>
<td>HPL time [%]</td>
<td>100</td>
<td>52</td>
<td>40</td>
</tr>
<tr>
<td>HPL power [W]</td>
<td>700</td>
<td>750</td>
<td>545</td>
</tr>
<tr>
<td>HPL [GF/s/W]</td>
<td>0.7</td>
<td>1.26</td>
<td>2.26</td>
</tr>
<tr>
<td>Triad [GB/s]</td>
<td>86</td>
<td>107</td>
<td>129</td>
</tr>
<tr>
<td>Triad efficiency [%]</td>
<td>84</td>
<td>78</td>
<td>84</td>
</tr>
<tr>
<td>Triad time [%]</td>
<td>100</td>
<td>80</td>
<td>67</td>
</tr>
<tr>
<td>Triad power [W]</td>
<td>620</td>
<td>380</td>
<td>425</td>
</tr>
<tr>
<td>Triad [GB/s/W]</td>
<td>0.135</td>
<td>0.287</td>
<td>0.303</td>
</tr>
</tbody>
</table>
Acknowledgement

- Michael Greenfield, Intel
- Larry Meadows, Intel
- John Levesque, Cray
- Bill Long, Cray
- Kevin Thomas, Cray
- Brent Leback, Nvidia
ESCAPE – The acraneb* dwarf


Initial study

<table>
<thead>
<tr>
<th>Baseline</th>
<th>case [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLOC [lines]</td>
<td>5687</td>
</tr>
<tr>
<td>Language state</td>
<td>F77 fixed-size, cast</td>
</tr>
<tr>
<td>Technical state</td>
<td>OK</td>
</tr>
<tr>
<td>Numerical state</td>
<td>4 digits</td>
</tr>
<tr>
<td>Largest psize on 16Gb</td>
<td>200x200x80</td>
</tr>
<tr>
<td>Largest psize on 64Gb</td>
<td>400x400x80</td>
</tr>
<tr>
<td>Largest psize on 128Gb</td>
<td>600x600x80</td>
</tr>
<tr>
<td>Target psize</td>
<td>1200x1080x65</td>
</tr>
</tbody>
</table>

Memory footprint [Gb] in baseline

Memory footprint [Gb] after refactoring

Now, < 64Gb
Performance improvement

- Current refactorization improvement:
  - Memory footprint reduction: 6.6x
  - Time (E5-2680v1@2.7 reference timing provided: 1.600)

<table>
<thead>
<tr>
<th></th>
<th>E5-2680v1@2.7</th>
<th>E5-2697v4@2.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>Refactoring</td>
<td>0.045</td>
<td>0.010</td>
</tr>
</tbody>
</table>

Investment in hardware vs software (matters more today!)

400x400x80

Intel Compiler
The sustained performance is not impressive neither on BDW nor on KNL but promising that KNL already outperforms BDW after our initial refactorization steps (memory trimming, SPMD threading and SIMD vectorization of the most expensive loop).