Benefits and Costs of Coarse-Grained Multithreading for HARMONIE

Enda O’Brien
Irish Centre for High-End Computing
Executive Summary

In the future (present?) ever more parallelism will be needed:
1. **MPI** for inter-node parallelism
2. **OpenMP** multithreading for intra-node parallelism
3. **Vectorization** for intra-core parallelism

There will be no escape from multithreading:
• MPI-only will not scale to \((\#\text{nodes} \times \#\text{cores/node})\)
  – MPI\_alltoall would excessively dominate run-time.
• HARMONIE already has quite “coarse-grained” multithreading...
  – But it is still not “good enough”.
  – Could it be even more fully “coarse-grained”?
  – Is full OpenMP domain decomp. (like MPI) possible, or desirable?
• Cycle 40 looks like it makes much better use of OpenMP.
Motivation

• The mission: make HARMONIE work on Xeon Phi coprocessors.
• HARMONIE now runs on Xeon Phi, in “native”, “symmetric”, and even “offload” mode – but only very slowly.
• The problems are mostly with Xeon Phi*.
• But one issue we can address is the *multithreading in HARMONIE.*

*(See 2014 ASM presentation)
“Fine-Grained” vs. “Coarse-Grained” Multithreading

**Fine-grained:**
- Parallel regions created and destroyed multiple times during run.
- Each thread does relatively little work.
- Parallel regions interspersed with sequential ones.
- Often loop-level parallelism, low in call-tree.

**Coarse-grained:**
- Parallel regions survive for long periods – even entire run.
- Each thread does a lot of work.
- Parallel regions interrupted by “critical sections”
- Domain-level parallelism, high in call tree.

Currently, HARMONIE has “partial” coarse-grained multithreading: it’s at a high level in call-tree, but avoids true “subdomain decomposition”.
Towards Full “Coarse-Grained” Multithreading

Consider a “global” domain of grid-points:

\[(\text{NX\_glob}, \text{NY\_glob}, \text{NZ\_glob})\]

After MPI decomposition, each MPI process sees:

\[(\text{NX\_loc}, \text{NY\_loc}, \text{NZ\_loc})\]

After more decomposition for coarse-grained multithreading:

\[(\text{NX\_loc}, \text{NY\_loc}, \text{NZ\_tloc})\]

(“Thread-local”)
Fine-Grained Multithreading Code

!$OMP PARALLEL DO PRIVATE(i,j,k)
    do k=2,nz_loc-1
        do j=2,ny_loc-1
            do i=2,nx_loc-1
                arr_out(i,j,k) = wght1*arr_in(i,j,k) + wght2*(
                    arr_in(i-1,j,k) + arr_in(i+1,j,k) +
                    arr_in(i,j-1,k) + arr_in(i,j+1,k) +
                    arr_in(i,j,k-1) + arr_in(i,j,k+1) )
            enddo
        enddo
    enddo
!$OMP END PARALLEL DO
Partial Coarse-Grained Multithreading

```fortran
!$OMP PARALLEL PRIVATE(<long list of private variables here...>)
...
!$OMP SINGLE  ! Only 1 thread does any I/O needed
...
!$OMP END SINGLE
...
do icount=1,ncount  ! Main outer loop
call halo_exchange(array,myrank,nx_loc,...)  ! Critical sections?
...
!$OMP DO
do k=2,nz_loc+1  ! Central nested loops as in fine-grain case
...
end do
!$OMP END DO
end do  ! End of main outer loop
!$OMP END PARALLEL
```

Parallel region spans entire code, including critical sections
Full Coarse-Grained Multithreading

!!!real shado_E(ny_loc,nz_loc)  ! MPI-local
    real shado_E(ny_loc,nz_tloc)  ! Thread-local
...
! Thread-wise decomposition needs extra book-keeping:
    kmin = mythread*nz_tloc  ! For thread-local point range
...
! Extract the “shadow-zone” part of main array:
!!!!$OMP DO
!!!   do k=2,nz_loc +1  ! MPI-only
     do k=2,nz_tloc+1  ! Thread decomposition
       do j=2,ny_loc+1
         shado_E(j-1,k-1) = ext_in(nx_loc+1,j, k)
         shado_E(j-1,k-1) = ext_in(nx_loc+1,j,kmin+k)
       enddo
     enddo
   enddo
!!!!$OMP END DO
Full Coarse-Grained Multithreading, contd.

```c
!!!!$OMP SINGLE
!Use if MPI not thread-safe: !$OMP ORDERED
!!! call MPI_Recv(shado_W(1,1),ny_loc*nz_loc ,MPI_REAL, &
                call MPI_Recv(shado_W(1,1),ny_loc*nz_tloc,MPI_REAL, &
                &      idest_e,itag_w,MPI_COMM_WORLD,istatus,ierror)
!
!!! call MPI_Send(shado_E(1,1),ny_loc*nz_loc, MPI_REAL, &
                call MPI_Send(shado_E(1,1),ny_loc*nz_tloc,MPI_REAL, &
                &      idest_e,itag_e,MPI_COMM_WORLD,ierror)
!Use this if MPI not thread-safe:  !$OMP END ORDERED
!!!!$OMP END SINGLE
```

Each thread now sends & receives its own messages - is this wise?
Coarse vs. Fine-grained OpenMP Parallelism

- **Fine-grained**: each loop is a separate OMP parallel region
- **Partial Coarse-grained**: Single high-level OpenMP parallel region
- **Full Coarse-grained**: Data fully decomposed to be thread-aware (as well as MPI-aware). Requires more, smaller MPI messages!
HARMONIE Parallel “Mix”: OpenMP or MPI?

Xeon host node, without HT:

With HT:

Conclusion:
For HARMONIE on Xeon nodes, MPI performs better than OpenMP.

Alternative Conclusion:
OpenMP in HARMONIE is not yet optimized!
HARMONIE on Xeon Phi Coprocessors

• HARMONIE can run on Xeon Phi Coprocessors, in either:
  – “Native” mode (full job run on Phi)
  – “Symmetric” mode (some MPI tasks on Phis, some on hosts)
  – “Offload” mode (some code-sections or “kernels” are transferred to Phi to be run there)
• Performance is poor – partly because HARMONIE runs out of memory before running out of cores (or threads)!
  – Memory use increases almost in proportion to thread-count!
• Cy40 changes should be able to avoid memory blow-up!
  – Still need to be tested, esp. on Xeon Phi...
Harmonie Scalability on MIC (Native & Symmetric modes)

One node; 20-tasks on host, 2 tasks on each of 2 MICs
HARMONIE: Symmetric Mode load (im)balance

Approx. 5 Harmonie timesteps; 20 host & 4 MIC processes
Symmetric Mode Profiles (20 host + 4 MIC MPIS)

<table>
<thead>
<tr>
<th>Host Profile (20 MPI x 2 threads)</th>
<th>Routine</th>
<th>% run-time</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAITRI @thread</td>
<td></td>
<td>4.4</td>
</tr>
<tr>
<td>APL_AROME @thread</td>
<td></td>
<td>4.3</td>
</tr>
<tr>
<td>RRTM_RTRN1A* @thread</td>
<td></td>
<td>3.2</td>
</tr>
<tr>
<td>TRLTOM</td>
<td></td>
<td>3.1</td>
</tr>
<tr>
<td>TRLTOG</td>
<td></td>
<td>3.0</td>
</tr>
<tr>
<td>TRMTOL</td>
<td></td>
<td>2.9</td>
</tr>
<tr>
<td>EGPNORM_TRANS</td>
<td></td>
<td>2.9</td>
</tr>
<tr>
<td>CPG</td>
<td></td>
<td>2.8</td>
</tr>
<tr>
<td>SLCOMM2A</td>
<td></td>
<td>2.7</td>
</tr>
<tr>
<td>INITAPLPAR @thread</td>
<td></td>
<td>2.2</td>
</tr>
<tr>
<td>GATHFLNM&lt;RECV</td>
<td></td>
<td>2.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MIC Prof (2 MIC x 2 MPI x 32 thrd)</th>
<th>Routine</th>
<th>% run-time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLCOMM</td>
<td></td>
<td>26.0</td>
</tr>
<tr>
<td>TRLTOG</td>
<td></td>
<td>7.0</td>
</tr>
<tr>
<td>SLCOMM2A</td>
<td></td>
<td>5.9</td>
</tr>
<tr>
<td>TRGTOL</td>
<td></td>
<td>3.7</td>
</tr>
<tr>
<td>TRMTOL</td>
<td></td>
<td>3.3</td>
</tr>
<tr>
<td>CPG</td>
<td></td>
<td>2.4</td>
</tr>
<tr>
<td>RRTM_RTRN1A* @thread</td>
<td></td>
<td>2.1</td>
</tr>
<tr>
<td>TRMTOS</td>
<td></td>
<td>2.0</td>
</tr>
<tr>
<td>TRLTOM</td>
<td></td>
<td>1.9</td>
</tr>
<tr>
<td>LAITRI @thread</td>
<td></td>
<td>1.7</td>
</tr>
<tr>
<td>APL_AROME @thread</td>
<td></td>
<td>1.7</td>
</tr>
</tbody>
</table>

(SLCOMM dominance on MICs indicates load imbalance; MIC tasks waiting for host messages)
• Harmonie has both MPI and OpenMP parallelism.

• OpenMP directives are at a very high level in source.

• Even so, memory usage is nearly proportional to thread-count – why?

• Turns out that main working-array sizes are simply increased in size to accommodate each new thread.
  – Okay for OMP_NUM_THREADS=2 or 4, perhaps
  – Definitely not okay for OMP_NUM_THREADS \geq 10 \text{ (never mind 240!)}
  – Reflects a “partial” implementation of “coarse-grained” multithreading.
High-level OpenMP Parallel Section (CPG)

“Working” arrays are declared (e.g., cpg.F90):

```fortran
REAL*8 :: ARRAY(NPROMA,NLEV,KBLKS)
```

Where:

NPROMA = Sequential “block” size (for optimal cache use), ~30 for scalar processors.
NLEV = no. of vertical levels (~60 or 90)
KBLKS = 1 (MPI-only), or OMP_NUM_THREADS

(Total horizontal grid-points is NGPTOT (~100,000) – not used in array declarations)

```fortran
!$OMP PARALLEL DO PRIVATE(J,L,...)
DO J = 1, NGPTOT, NPROMA
  !(calculate offsets, start & end pts. etc.)
  ...
  L = OML_MY_THREAD
  CALL CPG_GP(J, L, ARRAY(1,1,L),...)
  ...
ENDDO
```

Arrays in CPG_GP are then just 2-D: ARRAY(NPROMA,NLEV).
Final working arrays ultimately saved to large pointer-based state structure.

*Highly parallel, but profligate with memory!*
This is (Roughly...) Equivalent to:

```fortran
REAL*8 :: ARRAY(NPROMA,NLEV) ! Smaller size – but OMP “private”
Where:
NPROMA = Sequential block size, as before.
NLEV = No. of vertical levels, as before

(Total horizontal grid-points is NGPTOT (~100,000) – not used in array declarations)

!$OMP PARALLEL DO PRIVATE(J, ARRAY,...)
DO J = 1, NGPTOT, NPROMA
  !(calculate offsets, start & end pts. etc.)
  ...
  CALL CPG_GP(J, ARRAY,...)
  ...
ENDDO

Using arrays as OpenMP “private” variables multiplies use of memory!
```
How about just reducing NPROMA?

Reducing NPROMA certainly allows use of more threads on Phi.
• However, performance not much better.
• For fixed thread-count, performance is better for larger NPROMA:
Can we re-factor to save memory?

“Working” arrays could be declared:

\[
\text{REAL*8 :: ARRAY(NLEV,NPROMA) ! Switch dimensions}
\]

Where:

NPROMA = is \sim 10 \times \text{larger than before (to fill memory; e.g., \sim 4,000).}

NLEV = no. of vertical levels as before

Parallelize over the (new, larger) NPROMA, as in:

\[
\text{THRED_BLK_SZ = NPROMA/OMP_NUM_THREADS}
\]

\[
\text{DO I = 1, NBLKS ! (NBLKS \sim \text{NGPTOT/NPROMA})}
\]

\[
\text{!(calculate offsets, start & end pts. etc.)}
\]

\[
\text{ !$OMP PARALLEL DO PRIVATE(J,L,...)}
\]

\[
\text{DO J = 1, NPROMA, THRED_BLK_SZ}
\]

\[
\text{L = OML_MY_THREAD()}
\]

\[
\text{CALL CPG_GP(I,J,L, THRED_BLK_SZ, ARRAY(1,L),...)}
\]

\[
\text{...}
\]

\[
\text{ENDDO}
\]

\[
\text{ENDDO}
\]

Vertical dependencies in CPG_GP, etc. are handled naturally (thread-local).

One extra transpose required at end (to save in (x, z) order again).

\textit{Worth further exploration! (?)}
High-level OpenMP in Cycle-40

“Working” arrays are declared (e.g., cpg_drv.F90):

```fortran
REAL*8 :: ARRAY(NPROMA,NFLEVG,NGPBLKS)
```

Where:

- **NPROMA** = Sequential “block” size (for optimal cache use, ~30).
- **NFLEVG** = no. of vertical levels (~60 or 90)
- **NGPBLKS** = No. of “blocks” in NGPTOT; *independent of OMP_NUM_THREADS*

(Total horizontal grid-points is **NGPTOT** (~100,000) – not used in array declarations)

```fortran
!$OMP PARALLEL DO PRIVATE(J,IBL,...)
DO J = 1, NGPTOT, NPROMA
  !(calculate offsets, start & end pts. etc.)
  ...
  IBL = (J-1)/NPROMA+1 ! Block index; not tied to any thread
  CALL CPG(J, IBL, ARRAY(1,1,IBL),...)
  ...
ENDDO
```

Arrays in CPG are then just 2-D: ARRAY(NPROMA,NFLEVG).

Each thread still works on entire NPROMA-sized “block”; still highly parallel.

*This decoupling of NGPBLKS from OMP_NUM_THREADS from might work!*
Characteristics of Cycle-40 Design

✓ Approx. constant memory use as thread-count varies...
  ❌ ... though at a much higher level than in Cy38 (MPI-only).
  ❌ No net improvement if NGPBLKS > OMP_NUM_THREADS.

✓ Any other changes required already made by Tomas Wilhelmsson!
✓ NPROMA can play exactly the same role as before (cache-blocking).

Needs to be tested, especially on Xeon Phi!