Threading on the Node

OpenMP works great if it is implemented correctly
Looking at KNL

- **Lots of cores**
  - While MPI can run across all the cores, there are situations where MPI hits bottlenecks due to the number of MPI tasks on the node
  - We ALWAYS want to run multiple MPI tasks on the node
  - Want to identify a MPI sweet spot
  - A good guide is to start with a MPI task on each NUMA region

- **OpenMP**
  - Traditional approach has many short-comings
    - Requires a lot of code modifications
      - Lots of Comment Line directives
      - Does not deal with locality
      - Difficult to load balance
  - Is there a better way?
    - SPMD OpenMP
      - Fewer code modifications
      - Requires a better understanding of threads

- **KNL has NUMA issues**
Cluster Mode: Quadrant

Chip divided into four virtual Quadrants

Address hashed to a Directory in the same quadrant as the Memory

Affinity between the Directory and Memory

Lower latency and higher BW than all-to-all. SW Transparent.

1) L2 miss, 2) Directory access, 3) Memory access, 4) Data return
Why does all-MPI work well on multi/many core architectures?

- **All MPI forces locality**
  - Each MPI task allocated/utilizes memory within the NUMA region that it is running in.

- **All MPI allows tasks to run asynchronously**
  - This allows very good sharing of the memory bandwidth available on the node
  - We have found that KNL Quadrant mode is very good for all-MPI due to the memory bandwidth sharing to all memories

- **One MPI disadvantage is that re-distributing work is difficult and inefficient**
  - Have to move a lot of data
Can we take clues from all-MPI advantages and disadvantages to design a good OpenMP code?

● Can we force locality like MPI does?
  ● MPI forces each MPI task to allocate the data that it uses.
  ● Tradition OpenMP has no notion of locality

● Can we allow threads to work asynchronously?
  ● MPI only barriers when messages are exchanged
  ● Tradition OpenMP implies barriers after a parallel region
  ● Loop level parallelism forces too much synchronization

● Can we somehow control scheduling of the threads to enable more dynamic re-distribution of work
Can we force locality like MPI does?

- **Introduce a high level !$OMP PARALLEL region**
  - Down the call chain the user is responsible for managing threads
    - Initialize shared data within the !$OMP PARALLEL region, each thread allocates the data it will be using
    - Application developer must assure that shared data is shared as in the Fortran/C/C++ convention
      - This can be an issue down the call chain, when a shared local variable is required; that is, a reduction variable
    - Application developer must assure that private data is allocated on stack as with the Fortran and C conventions
What if you need a shared variable down the call chain

Subroutine within_a_parallel(

Real, pointer :: shared_p(:)

 !$omp single
allocate(shared_p(0:100))
 !$omp end single copyprivate(shared_p)

...

End subroutine
Can we allow threads to work asynchronously?

- **Must minimize synchronization**
  - Calling un-treaded routines
    - Must extend concept to all computational kernels
    - You can have replicated computation across the threads
  - Calling library routines
    - Can each thread call a un-threaded library routine
    - If library routine is threaded – must barrier prior to and after call
  - Calling MPI
    - Consider having each thread do it own message passing
      - Example coming
High-level OpenMP and Thread Scalable MPI-RMA:

Application Study with the Wombat Astrophysical MHD Code

Dr. Peter Mendygral
Cray Inc.
Wombat Driver and Parallel Region

Setup and object constructors

Thread parallel region
Array allocation and initialization

Time step loop
- DomainSolver%solve(mhd)
- DomainSolver%solve(ct)
- DomainSolver%solve(passive)
- DomainSolver%solve(particle kick+drift)
- FMG Solver%solve(all levels)
- DomainSolver%solve(particle kick)

I/O data dump(s)
Update time step

Array cleanup

Object destructors

Simulation complete
Communication Concerns

- If a rank is made much wider with threads, serialization around MPI will limit thread scaling and overall performance
  - Nearly all MPI libraries implement thread safety with a global lock
  - Cray is addressing this issue
    - Released per-object lock library
    - Threading enhancements under design now for two-sided (released per-object lock library a first step)

- Wide OpenMP also means more communication to process
  - Every Patch now has its own smaller boundaries to communicate
  - Starts tipping the behavior towards the message rate limit
  - Two-sided tag matching cannot be done in parallel and will limit thread scaling
    - May start hitting tag limit

- Slower serial performance of KNL => maybe look for the lightest weight MPI layer available
  - MPI-RMA over DMAPP on Cray systems is a thin software layer that achieves similar performance to SHMEM
RMA Boundary Communication Cycle

- Single passive RMA exposure epoch used for the duration of the application
  - No explicit synchronization between ranks
  - RMA semantics make computation/communication overlap simpler to achieve
Thread Hot MPI-RMA

- DMAPP library was enhanced to be “thread hot” for SHMEM
  - “thread hot” is more than “thread safe”
  - “thread hot” implies concurrency and performance across threads was central to the design

- MPI-RMA over DMAPP leverages this feature as of MPT 7.3.2
  - No locks used in DMAPP layer
  - Very light weight locking in MPI layer
  - Design makes it very likely that locks are uncontended
  - Network resources efficiently managed among threads
  - Performance approaches that of N independent processes when using N threads

- Example on HSW with 16 threads each on 2 nodes
  - OSU passive MPI_Put bandwidth for 8 B message
  - MPT 7.3.1 = 5.27 MB/s
  - MPT 7.3.2 = 399.9 MB/s
  - 75X improvement
● Tunable Patch size very important to performance
- Rank reordering
  - Cartesian domain optimization for XC topology/placement improves largest run wall time by additional 2.3%
- Less than 8% difference between 1 and 36 threads per rank
- Ideal for application like Wombat is 0%
Can we somehow control scheduling of the threads to enable more dynamic re-distribution of work

- With SPMD OpenMP, the user can take the responsibility for allocating the work to the threads
  - Can be simple chunking
  - Can understand the scarcity of the problem and allocate work accordingly
  - Can dynamically use runtime statistics to address load-imbalance
HBM code modernization -
insights from a Xeon Phi experiment

Jacob Weismann Poulsen, DMI, Denmark
Per Berg, DMI, Denmark
Karthik Raman, Intel, USA
The data is sparse and highly irregular
Data layout for threads (or tasks + explicit halo)

- Each thread will handle a subinterval of columns:

- Another layout of the columns will impose another decomposition for the threads (and the tasks).

```plaintext
!$OMP PARALLEL DEFAULT(SHARED)
call foo(...);call bar(...); ... 
!$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
   ...
endo
```

...
What about performance portability

- When running S3D on Titan we used Cuda Streams extensively
  - Consider using the 16 Cuda streams for the 16 threads within the SPMD unit.
Example of Cuda Streams (1)

```fortran
!$ACC DATA PRESENT(ids_lgl1,ids_ptr,ug,u)
!$ACC& CREATE(ug2)
    call rzero_acc(ug, stride*n)
    call rzero_acc(ug2, stride*n_nonlocal)

do k = 0, stride-1
!$ACC PARALLEL LOOP GANG VECTOR ASYNC(k+1)
!$ACC& PRIVATE(il,sil,sig)
    do i=1,nglobl       ! local Q^T
!$ACC LOOP SEQ
        do j = ids_ptr(i),ids_ptr(i+1)-1
            il = ids_lgl1(j)
            sil = k*n+il
            if (i.le.n_nonlocal) then   ! MPI
                sig = k*n_nonlocal+i
                ug2(sig) = ug2(sig)+u(sil)
            else
                sig = k*n+i
                ug(sig) = ug(sig)+u(sil)
            endif
        enddo
    enddo
!$ACC UPDATE HOST(ug2(k*n_nonlocal+1:(k+1)*n_nonlocal)) ASYNC(k+1)
enddo
!$ACC WAIT
```
Example of Cuda Streams (2)

do k = 0, stride-1
!$ACC UPDATE DEVICE(ug2(k*n_nonlocal+1:(k+1)*n_nonlocal)) ASYNC(k+1)

!$ACC PARALLEL LOOP GANG VECTOR ASYNC(k+1)
!$ACC& PRIVATE(il,sil,sig)
do i=1,nglobl ! local Q
!$ACC LOOP SEQ
  do j = ids_ptr(i),ids_ptr(i+1)-1
    il = ids_lgl1(j)
sil = k*n+il
    if (i.le.n_nonlocal ) then ! MPI
      sig = k*n_nonlocal+i
      u(sil) = ug2(sig)
    else
      sig = k*n+i
      u(sil) = ug(sig)
    endif
  enddo
endo
endo
!$ACC WAIT
!$ACC END DATA
Conclusion

- To scale well on many/multi-core systems, application developers must develop efficient threading
  - Must pay attention to NUMA regions
  - Must avoid overhead caused by
    - Too much synchronization
    - Load imbalance
- On some systems all-MPI will perform very well and will out-perform poorly implemented OpenMP
  - Not performance portable to hosted accelerators
- SPMD or Wide OpenMP is an alternative
  - Application developers have all the power to generate difficult to find race conditions, must understand the implications of high level threading