Hybrid MPI/OpenMP Programming

Yun (Helen) He
NERSC User Services Group

NUG 2013, February 15, 2013
Outline

• Architecture Trend
• Brief Review of OpenMP Basics
• Benefits of Hybrid MPI/OpenMP
• Hybrid MPI/OpenMP Programming Model
• Hybrid MPI/OpenMP Issues
• Compile and Run hybrid MPI/OpenMP at NERSC
Common Architectures

• **Shared Memory Architecture**
  – Multiple CPUs share global memory, could have local cache
  – Uniform Memory Access (UMA)
  – Typical Shared Memory Programming Model: OpenMP, Pthreads, ...

• **Distributed Memory Architecture**
  – Each CPU has own memory
  – Non-Uniform Memory Access (NUMA)
  – Typical Message Passing Programming Model: MPI, ...

• **Hybrid Architecture**
  – UMA within one SMP node or socket
  – NUMA across nodes or sockets
  – Typical Hybrid Programming Model: hybrid MPI/OpenMP, ...
Technology Trends

- Multi-socket nodes with rapidly increasing core counts.
- Memory per core decreases.
- Memory bandwidth per core decreases.
- Network bandwidth per core decreases.
- Deeper memory hierarchy.
Hopper Compute Nodes

- 2 twelve-core AMD 'MagnyCours' 2.1-GHz processors per node (2 sockets)
- 2 dies per socket
- 6 cores per die
- Each core has own L1 and L2 caches
- Each die (NUMA node) shares an L3 cache
- Each core has shared access to memory on all NUMA nodes
- But memory access to the remote NUMA nodes are slower
Edison and Carver Compute Nodes

• **Edison:**
  
  – Each compute node consists of two 8-core Intel Sandy Bridge 2.6 GHz processors (2 sockets)
  – 16 physical cores per node.
  – 32 logical cores when Hyper Threading (HT) is used.

• **Carver:**
  
  – Each compute node consists of two quad-core Intel Nehalem 2.67 GHz processors (2 sockets)
Hopper Memory Bandwidth

% qsub -l
% setenv CRAY_ROOTFS DSL
% aprun -n 1 numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5
node 0 size: 8191 MB
node 0 free: 7837 MB
node 1 cpus: 6 7 8 9 10 11
node 1 size: 8192 MB
node 1 free: 7883 MB
node 2 cpus: 12 13 14 15 16 17
node 2 size: 8192 MB
node 2 free: 7803 MB
node 3 cpus: 18 19 20 21 22 23
node 3 size: 8192 MB
node 3 free: 7844 MB

32 GB per node

node distances:
node 0 1 2 3
0: 10 16 16 16
1: 16 10 16 16
2: 16 16 10 16
3: 16 16 16 10

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>21.3</td>
<td>19.2</td>
<td>12.8</td>
<td>6.4</td>
</tr>
<tr>
<td>1</td>
<td>19.2</td>
<td>21.3</td>
<td>6.4</td>
<td>12.8</td>
</tr>
<tr>
<td>2</td>
<td>12.8</td>
<td>6.4</td>
<td>21.3</td>
<td>19.2</td>
</tr>
<tr>
<td>3</td>
<td>6.4</td>
<td>12.8</td>
<td>19.2</td>
<td>21.3</td>
</tr>
</tbody>
</table>

GB/sec
Edison Memory Bandwidth

% qsub -l
% setenv CRAY_ROOTFS DSL
% aprun -n 1 numactl --hardware

available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 16 17 18 19 20 21 22 23
node 0 size: 32744 MB
node 0 free: 31693 MB
node 1 cpus: 8 9 10 11 12 13 14 15 24 25 26 27 28 29 30 31
node 1 size: 32768 MB
node 1 free: 31335 MB

64 GB per node

Reports 32 cores with HT enabled

Streams Triad Numbers:
Full node: 78 GB/sec.
All cores within same NUMA node: 39 GB/sec
1 core between NUMA node 0 and 1: 9 GB/sec

node distances:
node 0 1
0: 10 20
1: 20 10

2 NUMA domains
What is OpenMP

- OpenMP is an industry standard API of C/C++ and Fortran for shared memory parallel programming.

- OpenMP components:
  - Compiler Directives and Clauses
    - Interpreted when OpenMP compiler option is turned on.
    - Each directive applies to the succeeding structured block.
  - Runtime Libraries
  - Environment Variables
OpenMP Programming Model

• **Fork and Join Model**
  – Master thread forks new threads at the beginning of parallel regions.
  – Multiple threads share work in parallel.
  – Threads join at the end of the parallel regions.

• Each thread works on **global shared** and its own **private** variables.

• Threads **synchronize implicitly** by reading and writing shared variables.
Advantages of OpenMP

• **Simple programming model**
  – Data decomposition and communication handled by compiler directives

• **Single source code** for serial and parallel codes

• No major overwrite of the serial code

• **Portable implementation**

• **Progressive parallelization**
  – Start from most critical or time consuming part of the code
Loop-based vs. SPMD

Loop-based:

```c
#$OMP PARALLEL DO PRIVATE(i)
#$OMP& SHARED(a,b,n)
   do I = 1, n
      a(i) = a(i) + b(i)
   enddo
#$OMP END PARALLEL DO
```

SPMD (Single Program Multiple Data):

```c
#$OMP PARALLEL DO PRIVATE(start, end, i)
#$OMP& SHARED(a,b)
   num_thrds = omp_get_num_threads()
   thrd_id = omp_get_thread_num()
   start = n * thrd_id/num_thrds + 1
   end = n * (thrd_num+1)/num_thrds
   do i = start, end
      a(i) = a(i) + b(i)
   enddo
#$OMP END PARALLEL DO
```

SPMD code normally gives better performance than loop-based code, but is more difficult to implement:

- Less thread synchronization.
- Less cache misses.
- More compiler optimizations.
**OMP task and taskwait**

<table>
<thead>
<tr>
<th>Serial:</th>
</tr>
</thead>
<tbody>
<tr>
<td>int fib (int n)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>int x, y;</td>
</tr>
<tr>
<td>if (n &lt; 2) return n;</td>
</tr>
<tr>
<td>x = fib (n - 1);</td>
</tr>
<tr>
<td>y = fib (n - 2);</td>
</tr>
<tr>
<td>return x+y;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OpenMP:</th>
</tr>
</thead>
<tbody>
<tr>
<td>int fib (int n) {</td>
</tr>
<tr>
<td>int x, y;</td>
</tr>
<tr>
<td>if (n &lt; 2) return n;</td>
</tr>
<tr>
<td>#pragma omp task shared (x)</td>
</tr>
<tr>
<td>x = fib (n - 1);</td>
</tr>
<tr>
<td>#pragma omp task shared (y)</td>
</tr>
<tr>
<td>y = fib (n - 2);</td>
</tr>
<tr>
<td>#pragma omp taskwait</td>
</tr>
<tr>
<td>return x+y;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

- Major OpenMP 3.0 addition. Flexible and powerful.
- The **task** directive defines an explicit task. Threads share work from all tasks in the task pool. The **taskwait** directive makes sure all child tasks created for the current task finish.
- Helps to improve load balance.
• **Static**: Loops are divided into \#thrds partitions.
• **Guided**: Loops are divided into progressively smaller chunks until the chunk size is 1.
• **Dynamic, \#chunk**: Loops are divided into chunks containing \#chunk iterations.
• **Auto**: The compiler (or runtime system) decides what to use.
• **Runtime**: Use OMP_SCHEDULE environment variable to determine at run time.
OMP_STACK_SIZE

- OMP_STACK_SIZE defines the private stack space each thread has.
- Default value is implementation dependent, and is usually quite small.
- Behavior is undefined if run out of space, mostly segmentation fault.
- To change, set OMP_STACK_SIZE to \( n \) \((B,K,M,G)\) bytes. `setenv OMP_STACK_SIZE 16M`
Cache Coherence and False Sharing

- ccNUMA node: cache-coherence NUMA node.
- Data from memory are accessed via cache lines.
- Multiple threads hold local copies of the same (global) data in their caches. Cache coherence ensures the local copy to be consistent with the global data.
- Main copy needs to be updated when a thread writes to local copy.
- Writes to same cache line is called false sharing or cache thrashing, since it needs to be done in serial. Use atomic or critical to avoid race condition.
- False sharing hurts parallel performance.
Thread Safety

- In general, IO operations, general OS functionality, common library functions may not be thread safe. They should be performed by one thread only or serialized.

- Avoid race condition in OpenMP program.
  - **Race condition**: Multiple threads are updating the same shared variable simultaneously.
  - Use “critical” directive
  - Use “atomic” directive
  - Use “reduction” directive
Why not perfect speedup with OpenMP?

<table>
<thead>
<tr>
<th>Jacobi OpenMP</th>
<th>Execution Time (sec)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 thread</td>
<td>121</td>
<td>1</td>
</tr>
<tr>
<td>2 threads</td>
<td>63</td>
<td>1.92</td>
</tr>
<tr>
<td>4 threads</td>
<td>36</td>
<td>3.36</td>
</tr>
</tbody>
</table>

Why not perfect speedup?
- Serial code sections not parallelized
- Thread creation and synchronization overhead
- Memory bandwidth
- Memory access with cache coherence
- Load balancing
- Not enough work for each thread
MPI vs. OpenMP

Pure MPI Pro:
- Portable to distributed and shared memory machines.
- Scales beyond one node.
- No data placement problem.

Pure MPI Con:
- Difficult to develop and debug.
- High latency, low bandwidth.
- Explicit communication.
- Large granularity.
- Difficult load balancing.

Pure OpenMP Pro:
- Easy to implement parallelism.
- Low latency, high bandwidth.
- Implicit Communication.
- Coarse and fine granularity.
- Dynamic load balancing.

Pure OpenMP Con:
- Only on shared memory machines.
- Scale within one node.
- Possible data placement problem.
- No specific thread order.
Why Hybrid MPI/OpenMP

- Hybrid MPI/OpenMP paradigm is the software trend for clusters of SMP architectures.
- Elegant in concept and architecture: using MPI across nodes and OpenMP within nodes. Good usage of shared memory system resource (memory, latency, and bandwidth).
- Avoids the extra communication overhead with MPI within node. Reduce memory footprint.
- OpenMP adds fine granularity (larger message sizes) and allows increased and/or dynamic load balancing.
- Some problems have two-level parallelism naturally.
- Some problems could only use restricted number of MPI tasks.
- Possible better scalability than both pure MPI and pure OpenMP.
Hybrid MPI/OpenMP Reduces Memory Usage

- Smaller number of MPI processes. Save the memory needed for the executables and process stack copies.
- Save memory for MPI buffers due to smaller number of MPI tasks.
- Fewer messages, larger message sizes, and smaller MPI all-to-all communication sizes improve performance.
- Larger domain for each MPI process, so fewer ghost cells
  - e.g. Combine four 10x10 domains to one 20x20. Assume 2 ghost layers.
  - Total grid size: Original: 4x14x14=784, new: 24x24=576.
A Pseudo Hybrid Code

Program hybrid
call MPI_INIT (ierr)
call MPI_COMM_RANK (…) 
call MPI_COMM_SIZE (…)
... some computation and MPI communication
call OMP_SET_NUM_THREADS(4)
$OMP PARALLEL DO PRIVATE(i)
$OMP& SHARED(n)
do i=1,n
    ... computation
enddo
$OMP END PARALLEL DO
... some computation and MPI communication
call MPI_FINALIZE (ierr)
end
**MPI_INIT_Thread Choices**

- **MPI_INIT_THREAD** *(required, provided, ierr)*
  - **IN**: required, desired level of thread support (integer).
  - **OUT**: provided, provided level of thread support (integer).
  - Returned provided maybe less than required.

- **Thread support levels:**
  - **MPI_THREAD_SINGLE**: Only one thread will execute.
  - **MPI_THREAD_FUNNELED**: Process may be multi-threaded, but only master thread will make MPI calls (all MPI calls are "funneled" to master thread)
  - **MPI_THREAD_SERIALIZED**: Process may be multi-threaded, multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads (all MPI calls are "serialized").
  - **MPI_THREAD_MULTIPLE**: Multiple threads may call MPI, with no restrictions.
## Thread Support Levels

<table>
<thead>
<tr>
<th>environment variable MPICH_MAX_THREAD_SAFETY</th>
<th>Hopper/Edison</th>
<th>Carver</th>
</tr>
</thead>
<tbody>
<tr>
<td>not set</td>
<td>MPI_THREAD_SINGLE</td>
<td>MPI_THREAD_SINGLE</td>
</tr>
<tr>
<td>single</td>
<td>MPI_THREAD_SINGLE</td>
<td>MPI_THREAD_SINGLE</td>
</tr>
<tr>
<td>funneled</td>
<td>MPI_THREAD_FUNNELED</td>
<td>MPI_THREAD_SINGLE</td>
</tr>
<tr>
<td>serialized</td>
<td>MPI_THREAD_SERIALIZED</td>
<td>MPI_THREAD_SINGLE</td>
</tr>
<tr>
<td>multiple</td>
<td>MPI_THREAD_MULTIPLE</td>
<td>MPI_THREAD_SINGLE</td>
</tr>
</tbody>
</table>
MPI Calls Inside OMP MASTER

- **MPI_THREAD_FUNNELED** is required.
- **OMP_BARRIER** is needed since there is no synchronization with OMP_MASTER.
- It implies all other threads are sleeping!

```bash
 !$OMP BARRIER
 !$OMP MASTER
 call MPI_xxx(...)
 !$OMP END MASTER
 !$OMP BARRIER
```
• **MPI_THREAD_SERIALIZED** is required.
• **OMP_BARRIER** is needed since **OMP_SINGLE** only guarantees synchronization at the end.
• It also implies all other threads are sleeping!

```
!$OMP BARRIER
!$OMP SINGLE
  call MPI_xxx(...)
!$OMP END SINGLE
```
• **FUNNELED/SERIALIZED:**
  – All other threads are sleeping while single thread communicating.
  – Only one thread communicating maybe not able to saturate the inter-node bandwidth.

• **Pure MPI:**
  – Every CPU communicating may over saturate the inter-node bandwidth.

• **Overlap communication with computation!**
Overlap COMM and COMP

- Need at least MPI_THREAD_FUNNELED.
- Many “easy” hybrid programs only need MPI_THREAD_FUNNELED.
- While master or single thread is making MPI calls, other threads are computing.
- Must be able to separate codes that can run before or after halo info is received. Very hard.
- Lose compiler optimizations.

```c
!$OMP PARALLEL
  if (my_thread_rank < 1) then
    call MPI_xxx(…)
  else
    do some computation
  endif
!$OMP END PARALLEL
```
Thread Affinity

• Thread affinity: forces each process or thread to run on a specific subset of processors, to take advantage of local process state.

• OpenMP 3.1 introduces the OMP_PROC_BIND env variable

• On Hopper/Edison, there is aprun command option “-cc”:
  – **-cc cpu (default)**: Each PE’s thread is constrained to the CPU closest to the PE.
  – **-cc numa_node**: Each PE’s thread is constrained to the same NUMA node CPUs.
  – **-cc none**: Each thread is not binded to a specific CPU.

• On Carver, “mpirun” has options:
  – **bind-to-socket**: bind processes to processor sockets
  – **bind-to-core**: bind processes to cores.
  – **bind-to-none (default)**: do not bind processes.
• Memory affinity: allocate memory as close as possible to the core on which the task that requested the memory is running.

• Hopper/Edison: “aprun” option: “-ss”
  – Specifies strict memory containment per NUMA node. A process can only access memory local to its assigned NUMA node.
  – Only makes sense if the thread affinity is accomplished with “–cc cpu” (default) or “–cc numa_node” first.

• No memory affinity option for Carver.
“First Touch” Memory

• Memory affinity is not decided by the memory allocation, but by the initialization. This is called “first touch” policy.
• Hard to do “perfect touch” for real applications.
• On Hopper: NERSC recommends do not use more than 6 threads per node to avoid NUMA effect.

```
#pragma omp parallel for
for (j=0; j<VectorSize; j++) {
    a[j] = 1.0; b[j] = 2.0; c[j] = 0.0;
}
```

```
#pragma omp parallel for
for (j=0; j<VectorSize; j++) {
    a[j]=b[j]+d*c[j];
}
```

Stream NUMA effects - Hopper

Courtesy Hongzhang Shan
## More aprun Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>-n</td>
<td>Number of MPI tasks.</td>
</tr>
<tr>
<td>-N</td>
<td>(Optional) Number of MPI tasks per Node.</td>
</tr>
<tr>
<td>-d</td>
<td>(Optional) Depth, or number of threads, per MPI task. Required in addition to <code>OMP_NUM_THREADS</code> for OpenMP.</td>
</tr>
<tr>
<td>-S</td>
<td>(Optional) Number of MPI tasks per NUMA node.</td>
</tr>
<tr>
<td>-sn</td>
<td>(Optional) Number of NUMA nodes to use per node</td>
</tr>
<tr>
<td>-ss</td>
<td>(Optional) Demands strict memory containment per NUMA node. The default is the opposite - to allow remote NUMA node memory access. Use this for most OpenMP codes.</td>
</tr>
<tr>
<td>-cc</td>
<td>(Optional) Controls how tasks are bound to cores and NUMA nodes. Options are: <code>-cc cpu (default), -cc numa_node, and --cc none.</code></td>
</tr>
</tbody>
</table>
The “-S” option is especially important for hybrid MPI/OpenMP applications, since we would like to spread the MPI tasks onto different NUMA nodes.

**aprun “-S” option**

- `aprun -n 4 -d 6...`
- `aprun -n 4 -S 1 -d 6 ...`
Hopper aprun Command Example

• #PBS -l mppwidth=72 (so 3 nodes!)
• 1 MPI task per NUMA node with 6 threads
  – setenv OMP_NUM_THREADS 6
  – aprun –n 12 –N 4 –S 1 -d 6 -ss ./a.out
• 2 MPI tasks per NUMA node with 3 threads
  – setenv OMP_NUM_THREADS 3
  – aprun –n 24 –N 8 –S 2 -d 3 –ss ./a.out
Hopper Core Affinity

- “xthi.c”: a hybrid MPI/OpenMP code that reports process and thread affinity.
- Source code can be found at (page 95-96): http://docs.cray.com/books/S-2496-4101/S-2496-4101.pdf

% aprun -n 4 ./xthi
Hello from rank 0, thread 0, on nid01085. (core affinity = 0)
Hello from rank 1, thread 0, on nid01085. (core affinity = 1)
Hello from rank 3, thread 0, on nid01085. (core affinity = 3)
Hello from rank 2, thread 0, on nid01085. (core affinity = 2)

% aprun -n 4 -S 1 ./xthi
Hello from rank 3, thread 0, on nid01085. (core affinity = 18)
Hello from rank 0, thread 0, on nid01085. (core affinity = 0)
Hello from rank 2, thread 0, on nid01085. (core affinity = 12)
Hello from rank 1, thread 0, on nid01085. (core affinity = 6)
Carver Core Affinity

2 nodes, 2 MPI tasks per node, OMP_NUM_THREADS=4

% mpirun -np 4 -bysocket -bind-to-socket ./xthi

Hello from rank 1, thread 0, on c0803. (core affinity = 4-7)
Hello from rank 1, thread 3, on c0803. (core affinity = 4-7)
Hello from rank 1, thread 1, on c0803. (core affinity = 4-7)
Hello from rank 1, thread 2, on c0803. (core affinity = 4-7)
Hello from rank 3, thread 1, on c0540. (core affinity = 4-7)
Hello from rank 3, thread 3, on c0540. (core affinity = 4-7)
Hello from rank 3, thread 0, on c0540. (core affinity = 4-7)
Hello from rank 3, thread 2, on c0540. (core affinity = 4-7)
Hello from rank 0, thread 0, on c0803. (core affinity = 0-3)
Hello from rank 0, thread 2, on c0803. (core affinity = 0-3)
Hello from rank 2, thread 0, on c0540. (core affinity = 0-3)
Hello from rank 2, thread 1, on c0540. (core affinity = 0-3)
Hello from rank 0, thread 1, on c0803. (core affinity = 0-3)
Hello from rank 0, thread 3, on c0803. (core affinity = 0-3)
Hello from rank 2, thread 2, on c0540. (core affinity = 0-3)

% mpirun -np 4 -bynode ./xthi

Hello from rank 1, thread 0, on c0540. (core affinity = 0-7)
Hello from rank 1, thread 1, on c0540. (core affinity = 0-7)
Hello from rank 1, thread 2, on c0540. (core affinity = 0-7)
Hello from rank 1, thread 3, on c0540. (core affinity = 0-7)
Hello from rank 0, thread 0, on c0803. (core affinity = 0-7)
Hello from rank 0, thread 2, on c0803. (core affinity = 0-7)
Hello from rank 2, thread 0, on c0803. (core affinity = 0-7)
Hello from rank 2, thread 2, on c0803. (core affinity = 0-7)
Hello from rank 2, thread 1, on c0803. (core affinity = 0-7)
Hello from rank 2, thread 3, on c0803. (core affinity = 0-7)
Hello from rank 0, thread 2, on c0803. (core affinity = 0-7)
Hello from rank 0, thread 3, on c0803. (core affinity = 0-7)
Hello from rank 0, thread 1, on c0803. (core affinity = 0-7)
Hello from rank 3, thread 2, on c0540. (core affinity = 0-7)
Hello from rank 3, thread 1, on c0540. (core affinity = 0-7)
Hello from rank 3, thread 3, on c0540. (core affinity = 0-7)
Hello from rank 3, thread 0, on c0540. (core affinity = 0-7)
Compile Hybrid MPI/OpenMP

- **Always use the compiler wrappers:**
  - Hopper/Edison: ftn, cc, C++
  - Carver: mpif90, mpicc, mpiCC

- **Need to use the programming environment for each compiler**

- **Portland Group Compilers (Hopper/Carver)**
  - Add compiler option “-mp”
  - For example: `% ftn –mp mycode.f90` (Hopper)
  - `% mpif90 –mp mycode.f90` (Carver)
  - Fully support OpenMP 3.0 from pgi/8.0.
  - Plan to fully support OpenMP 3.1 in early 2013.
  - Partial OpenMP 3.1 support in between.
Compile Hybrid MPI/OpenMP (2)

**Cray Compilers (Hopper/Edison)**
- Hopper: `% module swap PrgEnv-pgi PrgEnv-cray`
  Edison: `% module swap PrgEnv-intel PrgEnv-cray`
- No additional compiler option needed.
  - Use “-h noomp” to disable OpenMP
- For example: `% ftn mycode.f90`
- Fully support OpenMP 3.1 from cce/8.1.0 (released 9/20/2012).

**Intel Compilers**
- Hopper: `% module swap PrgEnv-pgi PrgEnv-intel`
  Edison: *no need to swap PrgEnv*
  Carver: `% module unload pgi openmpi`
  `% module load intel openmpi-intel`
- Add compiler option “-openmp”
- For example: `% ftn -openmp mycode.f90` (Hopper/Edison)
  `% mpif90 -openmp mycode.f90` (Carver)
- Fully support OpenMP 3.1 from intel/12.1 (released 9/5/2011).
Compile Hybrid MPI/OpenMP (3)

• **GNU Compilers**
  – Hopper: `% module swap PrgEnv-pgi PrgEnv-gnu`
  – Edison: `% module swap PrgEnv-intel PrgEnv-gnu`
  – Carver: `% module unload pgi openmpi`
    % module load gcc openmpi-gcc
  – Add compiler option “-fopenmp”
  – For example: `% ftn -fopenmp mycode.f90` (Hopper/Edison)
    % mpif90 -fopenmp mycode.f90 (Carver)
  – Fully support OpenMP 3.1 from gcc/4.7.0 (released 3/22/2012).

• **Pathscale Compilers (Hopper only)**
  – `% module swap PrgEnv-pgi PrgEnv-pathscale`
  – Add compiler option “-mp”
  – For example: `% ftn -mp mycode.f90`
  – Fully support OpenMP 2.5
Run Hybrid MPI/OpenMP on Hopper

- Each Hopper node has 4 NUMA nodes, each with 6 UMA cores.
- Recommend to use max 6 OpenMP threads per node, and MPI across NUMA nodes. (although up to 24 OpenMP threads per Hopper node possible).

Interactive batch jobs:

2 Hopper nodes, 8 MPI tasks, 6 OpenMP threads per MPI task:

- % qsub -l V -q interactive -lmppwidth=48
- wait for a new shell
- % cd $PBS_O_WORKDIR
- % setenv OMP_NUM_THREADS 6
- % setenv PSC_OMP_AFFINITY FALSE (note: for Pathscale only)
- % aprun -n 8 -N 4 -S 1 -ss -d 6 ./mycode.exe
  (for Intel: add “-cc numa_node” in the aprun line).
Run Hybrid MPI/OpenMP on Hopper (2)

Sample batch script:
(pure OpenMP example, Using 24 OpenMP threads)

#PBS -q debug
#PBS -l mppwidth=24
#PBS -l walltime=00:10:00
#PBS -j eo
#PBS –V

cd $PBS_O_WORKDIR
setenv OMP_NUM_THREADS 24

#uncomment this line for pathscale
#setenv PSC_OMP_AFFINITY FALSE

# add “-cc numa_node” for Intel
aprun –n 1 -N 1 –d 24 ./mycode.exe

• Run batch jobs:
  – Prepare a batch script first
  – % qsub myscript

• Hybrid MPI/OpenMP
  – 1 Hopper node, 4 MPI tasks, 6 OpenMP threads per MPI task:
    • % aprun –n 4 –N 4 –S 1 –ss
      –d 6 ./mycode.exe
  – 2 Hopper nodes, 8 MPI tasks, 6 threads per MPI task:
    • #PBS -l mppwidth=48
      – 24 cores/node * 2 nodes
    • % aprun –n 8 –N 4 –S 1 –ss 
      –d 6 ./mycode.exe
Special Considerations for Pathscale and Intel Compilers on Hopper

• For Pathscale compilers, need to set environment variable PSC_OMP_AFFINITY to FALSE at the run time.
  – This is to turn off the Pathscale internal control of cpu affinity.

• For Intel compilers, need to use “-cc numa_node” or “-cc none” instead of the default “-cc cpu” option for aprun.
  – This is due to Intel starts an extra thread with OpenMP.
Run Hybrid MPI/OpenMP on Edison

- Each Edison node has 2 NUMA nodes, each with 8 UMA cores (or 16 UMA logical cores with HT).
- Recommend to use max 8 OpenMP threads per node (max 16 threads with HT), and MPI across NUMA nodes.
- Interactive batch jobs:

  2 Edison nodes, 4 MPI tasks, 8 OpenMP threads per MPI task:
  - `% qsub -l -V -q interactive -lmppwidth=32`
  - `wait for a new shell`
  - `% cd $PBS_O_WORKDIR`
  - `% setenv OMP_NUM_THREADS 8`
  - `% aprun -n 4 -N 2 -S 1 -ss -d 8 ./mycode.exe`

  (for Intel: add “setenv KMP_AFFINITY compact” before the aprun line, also add “–cc numa_node” in the aprun line).
Sample batch script: *(pure OpenMP example, Using 16 OpenMP threads)*

```
#PBS -q debug
#PBS -l mppwidth=16
#PBS -l walltime=00:10:00
#PBS -j eo
#PBS -V

cd $PBS_O_WORKDIR
setenv OMP_NUM_THREADS 16

# uncomment this line for Intel
#setenv KMP_AFFINITY scatter

# need to add "-cc none" for Intel
aprun -n 1 -N 1 -d 16 ./mycode.exe
```

Sample batch script: *(hybrid MPI/OpenMP example, Using 8 OpenMP threads per task)*

```
#PBS -q debug
#PBS -l mppwidth=32
#PBS -l walltime=00:10:00
#PBS -j eo
#PBS -V

cd $PBS_O_WORKDIR
setenv OMP_NUM_THREADS 8

# uncomment this line for Intel
#setenv KMP_AFFINITY compact

# need to add "-cc numa_node" for Intel
aprun -n 4 -N 2 -S 1 -ss -d 8 ./mycode.exe
```
Run Hybrid MPI/OpenMP on Edison with HT (3)

**Sample batch script:**
(pure OpenMP example with HT, Using 32 OpenMP threads)

```
#PBS -q debug
#PBS -l mppwidth=32
#PBS -l mppnppn=32
#PBS -l walltime=00:10:00
#PBS -j eo
#PBS -V

cd $PBS_O_WORKDIR
setenv OMP_NUM_THREADS 32

# uncomment this line for Intel
#setenv KMP_AFFINITY scatter

# need to add “-cc none” for Intel
aprun -n 1 -N 1 -j 2 -d 32 ./mycode.exe
```

**Sample batch script:**
(hybrid MPI/OpenMP example with HT, Using 16 OpenMP threads per task)

```
#PBS -q debug
#PBS -l mppwidth=64
#PBS -l mppnppn=32
#PBS -l walltime=00:10:00
#PBS -j eo
#PBS -V


cd $PBS_O_WORKDIR
setenv OMP_NUM_THREADS 16

# uncomment this line for Intel
#setenv KMP_AFFINITY compact

# need to add “-cc numa_node” for Intel
aprun -n 4 -N 2 -S 1 -j 2 -ss -d 16 ./mycode.exe
```
Special Considerations for Intel Compilers on Edison

• For Intel compilers, need to use “-cc numa_node” or “-cc none” instead of the default “-cc cpu” option for aprun. Also set a run time env **KMP_AFFINITY** to compact or scatter.
  – This is due to the conflict of the internal Intel thread affinity and the aprun thread affinity.

• when OMP_NUM_THREADS <= 8 (or <= 16 with HT)
  – setenv KMP_AFFINITY compact
  – add “-cc numa_node” in the aprun line

• when OMP_NUM_THREADS > 8 (or > 16 with HT)
  – setenv KMP_AFFINITY scatter
  – add “-cc none” in the aprun line
Run Hybrid MPI/OpenMP on Carver

• Each Carver node has 8 cores, 2 sockets with 4 cores each.

• Use max 8 OpenMP threads per node.

• Interactive batch jobs:
  – Pure OpenMP example, using 8 OpenMP threads:
  – % qsub –l –V –q interactive –l nodes=1:ppn=1,pvmem=20GB
  – wait for a new shell
  – % cd $PBS_O_WORKDIR
  – setenv OMP_NUM_THREADS 8
  – % mpirun –np 1 ./mycode.exe

• Change nodes:ppn, pvmem and mpirun –np options for hybrid MPI/OpenMP jobs.
Run batch jobs:
- Prepare a batch script first
- % qsub myscript

Hybrid MPI/OpenMP
- 2 Carver nodes, 1 MPI task per node, 8 OpenMP threads per MPI task:
  - #PBS -l nodes=2:ppn=1
  - #PBS -l pvmem=20GB
  - setenv OMP_NUM_THREADS 8
  - % mpirun --np 2 ./mycode.exe

Notice the setting for pvmem
- Default is 20GB per MPI process per node.
- Set to 10GB for 2 MPI tasks per node
- Set to 5 GB for 4 MPI tasks per node
On a single node, hybrid MPI/OpenMP NAS Parallel Benchmarks:
• Reduced memory footprint with increased OpenMP threads.
• Hybrid MPI/OpenMP can be faster or comparable to pure MPI.
• Try different compilers.
• Sweet spot: BT: 1-3 threads; LU: 6 threads.
Hopper: Hybrid MPI/OpenMP fvCAM

Community Atmospheric Model:
• Memory reduces to 50% with 3 threads but only 6% performance drop.
• OpenMP time starts to grow from 6 threads.
• Load imbalance in “Dynamics” OpenMP

Courtesy of Nick Wright, et. al, NERSC/Cray Center of Excellence
3d Gyrokinetic Toroidal Code:
• Memory reduces to 50% with 3 threads, also 15% better performance
• NUMA effects seen with 12 threads
• Mixed results in different kernels

Courtesy of Nick Wright, et. al, NERSC/Cray Center of Excellence
Hybrid Parallelization Strategies

- From sequential code, decompose with MPI first, then add OpenMP.
- From OpenMP code, treat as serial code.
- From MPI code, add OpenMP.
- Simplest and least error-prone way is to use MPI outside parallel region, and allow only master thread to communicate between MPI tasks. MPI_THREAD_FUNNELED is usually the best choice.
- Could use MPI inside parallel region with thread-safe MPI.
- Avoid MPI_THREAD_MULTIPLE if you can. It slows down performance due to the usage of global locks for thread safety.
Why Mixed OpenMP/MPI Code is Sometimes Slower?

- All threads are idle except one while MPI communication.
  - Need overlap comp and comm for better performance.
  - Critical Section for shared variables.
- Thread creation overhead
- Cache coherence, false sharing.
- Data placement, NUMA effects.
- Natural one level parallelism problems.
- Pure OpenMP code performs worse than pure MPI within node.
- Lack of optimized OpenMP compilers/libraries.
Debug and Tune Hybrid Codes

• Debugger tools: DDT, Totalview, gdb, Valgrind.
• Profiling: IPM, CrayPat, TAU.
• Decide which loop to parallelize. Better to parallelize outer loop. Decide whether Loop permutation, fusion or exchange is needed. Use NOWAIT clause if possible.
• Choose between loop-based or SPMD.
• Use different OpenMP task scheduling options.
• Experiment with different combinations of MPI tasks and number of threads per MPI task. Fewer MPI tasks may not saturate inter-node bandwidth.
• Adjust MPI and OpenMP runtime environment variables.
• Aggressively investigate different thread initialization options and the possibility of overlapping communication with computation.
• Try OpenMP TASK.
• Leave some cores idle on purpose: memory capacity or bandwidth capacity.
• Try different compilers.
OpenMP Profiling with IPM

- IPM is a light weight profiling tool. OpenMP profiling currently works with PGI and Cray compilers on Hopper. Will be available on Edison soon with Intel and Cray compilers.
- PGI compiler:
  - % module load ipm-openmp/pgi
  - % ftn -mp=trace test_omp.f $IPM
  - % cc -mp=trace test_omp.c $IPM
- Cray compiler:
  - % module swap PrgEnv-pgi PrgEnv-cray
  - % module load ipm-openmp/cray
  - % ftn -h omp_trace test_omp.f $IPM
  - % cc -h omp_trace test_omp.c $IPM
- Run the code as usual on the compute nodes.
- OMP_PARALLEL: Total time spent in OMP regions.
- OMP_IDLE: total time from each thread waiting for others. This shows load imbalance.

---

```bash
# command : ./jacobi_mpiomp
# start : Thu Feb 02 10:04:21 2012 host : nid01840
# stop : Thu Feb 02 10:04:22 2012 wallclock : 0.77
# mpi_tasks : 4 on 1 nodes %comm : 12.50
# omp_thrds : 6 %omp : 85.05
# mem [GB] : 0.03 g flop/sec : 1.52
#
# % : [total] <avg> min max
# wallclock :  3.09  0.77  0.77  0.77
# MPI : 0.39 0.10 0.01 0.13
# OMP : 2.63 0.66 0.64 0.71
# OMP idle : 0.10 0.03 0.01 0.07
# %wall :
# MPI : 12.50 1.02 16.38
# OMP : 85.05 82.60 92.30
# #calls :
# MPI : 14056 3514 3514 3514
# mem [GB] : 0.03 0.01 0.01 0.01
# #
# @OMP_PARALLEL 2.63 9010 85.05
# @OMP_IDLE 0.62 54060 19.91
# MPI_Allreduce 0.22 2000 7.14
# MPI_Bcast 0.12 16 3.84
# MPI_Sendrecv 0.05 4000 1.49
# MPI_Comm_size 0.00 4016 0.02
# MPI_Comm_rank 0.00 4016 0.01
# MPI_Init 0.00 4 0.00
# MPI_Finalize 0.00 4 0.00
```

---

Profiling with IPM

```bash
$IPM PrgEnv-pgi
$IPM PrgEnv-cray
$IPM test_omp.c
$IPM test_omp.f
```
Conclusions

• Flat MPI is still the dominant parallel programming model today. But it is time to consider adding thread parallelism to MPI.

• Hybrid MPI/OpenMP is suited for the multi-core architecture trend.

• Whether hybrid MPI/OpenMP performs better than MPI depends on whether the communication advantage outcomes the thread overhead, etc. or not.

• A great benefit for using hybrid MPI/OpenMP is the reduced memory footprint per node.
Further References

- Sample Codes and Scripts
  - module load training/2013
  - cd $EXAMPLES/NUG/hybrid
- MPI: http://www.mcs.anl.gov/research/projects/mpi/
- OpenMP: http://openmp.org
National Energy Research Scientific Computing Center