Explore Hybrid MPI/OpenMP Scaling on NERSC Systems

Helen He
NERSC User Service

October 28, 2014
Goals and Outline

• Goals
  – Not a tutorial on MPI or OpenMP
  – Practical tips and real case studies of hybrid MPI/OpenMP implementations to prepare applications for Cori

• Outline
  – Introduction
  – Scaling Tips
  – Process and Thread Affinity
  – Tools for OpenMP
  – Case Studies
Introduction
The Big Picture

• The next large NERSC production system “Cori” will be Intel Xeon Phi KNL (Knights Landing) architecture:
  – >60 cores per node, 4 hardware threads per core
  – Total of >240 threads per node

• Your application is very likely to run on KNL with simple port, but high performance is harder to achieve.

• Many applications will not fit into the memory of a KNL node using pure MPI across all HW cores and threads because of the memory overhead for each MPI task.

• Hybrid MPI/OpenMP is the recommended programming model, to achieve scaling capability and code portability.

• Current NERSC systems (Babbage, Edison, and Hopper) can help prepare your codes.
Hybrid MPI/OpenMP Reduces Memory Usage

- Smaller number of MPI processes. Save the memory needed for the executables and process stack copies.

- Larger domain for each MPI process, so fewer ghost cells
  - e.g. Combine 16 10x10 domains to one 40x40. Assume 2 ghost layers.
  - Total grid size: Original: 16x14x14=3136, new: 44x44=1936.

- 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.
Why Scaling is So Important

- Scaling of an application is important to get the performance potential on the Xeon Phi manycore systems.
- Does not imply to scale with “pure MPI” or “pure OpenMP”
- Does not imply the need to scale all the way to 240-way either
- Rather, should explore hybrid MPI/OpenMP, find some sweet spots with combinations, such as: 4 MPI tasks * 15 threads per task, or 8*20, etc.
Babbage

- NERSC Intel Xeon Phi Knights Corner (KNC) testbed.
- 45 compute nodes, each has:
  - Host node: 2 Intel Xeon Sandybridge processors, 8 cores each.
  - 2 MIC cards each has 60 native cores and 4 hardware threads per core.
  - MIC cards attached to host nodes via PCI-express.
  - 8 GB memory on each MIC card
- **Recommend to use at least 2 threads per core to hide latency of in-order execution.**

To best prepare codes on Babbage for Cori:
- Use “native” mode on KNC to mimic KNL, which means ignore the host, just run completely on KNC cards.
- Encourage single node exploration on KNC cards with problem sizes that can fit.
Scaling and Tips
Fine Grain and Coarse Grain Models

Program fine_grain

```fortran
!$OMP PARALLEL DO
  do i=1,n
  ... computation
  enddo
!$OMP END PARALLEL DO

... some serial computation ...

!$OMP PARALLEL DO
  do i=1,n
  ... computation
  enddo
!$OMP END PARALLEL DO
end
```

Program coarse_grain

```fortran
!$OMP PARALLEL
!$OMP DO
  do i=1,n
  ... computation
  enddo
!$OMP END DO

!$OMP DO
  do i=1,n
  ... computation
  enddo
!$OMP END DO
!$OMP END PARALLEL
end
```

- Program is single threaded except when actively using multiple threads, such as loop processing,
- Pro: Easier to adapt to MPI program.
- Con: thread overhead, serial section becomes bottleneck.

- Majority of program run within an OMP parallel region.
- Pro: low overhead of thread creation, consistent thread affinity.
- Con: harder to code, prone to race condition.
Memory Affinity: “First Touch” Memory

• Memory affinity: allocate memory as close as possible to the core on which the task that requested the memory is running.
• Memory affinity is not decided by the memory allocation, but by the initialization. Memory will be local to the thread which initializes it. This is called “first touch” policy.
• Hard to do “perfect touch” for real applications. Instead, use number of threads few than number of cores per NUMA domain.

Initialization

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

Compute

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

Stream NUMA effects - Hopper

Higher is Better

Courtesy of Hongzhang Shan
Cache Coherence and False Sharing

- 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.
Cache Locality

• Use data in cache as much as possible
  – Use a memory stride of 1
    • Fortran: column-major order
    • C: row-major order
  – Access variable elements in the same order as they are stored in memory
  – Interchange loops or index orders if necessary
  – Tips often used in real codes
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

<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>
Programming Tips for Adding OpenMP

• Choose between fine grain or coarse grain parallelism implementation.
• Use profiling tools to find hotspots. Add OpenMP and check correctness incrementally.
• Parallelize outer loop and collapse loops if possible.
• Minimize shared variables, minimize barriers.
• Decide whether to overlap MPI communication with thread computation.
  – Simplest and least error-prone way is to use MPI outside parallel region, and allow only master thread to communicate between MPI tasks.
  – Could use MPI inside parallel region with thread-safe MPI.
• Consider OpenMP TASK.
MPI vs. OpenMP Scaling Analysis

Each line represents multiple runs using fixed total number of cores = #MPI tasks x #OpenMP threads/task.

Scaling may depend on the kernel algorithms and problem sizes.

In this test case, 15 MPI tasks with 8 OpenMP threads per task is optimal.

Understand your code by creating the MPI vs. OpenMP scaling plot, find the sweet spot for hybrid MPI/OpenMP.

It can be the base setup for further tuning and optimizing on Xeon Phi.
If a Routine Does Not Scale Well

• Examine code for serial/critical sections, eliminate if possible.
• Reduce number of OpenMP parallel regions to reduce overhead costs.
• Perhaps loop collapse, loop fusion or loop permutation is required to give all threads enough work, and to optimize thread cache locality. Use NOWAIT clause if possible.
• Pay attention to load imbalance. If needed, try dynamic scheduling or implement own load balance scheme.
• Experiment with different combinations of MPI tasks and number of threads per task. Less MPI tasks may not saturate inter-node bandwidth.
• **Test different process and thread affinity options.**
• Leave some cores idle on purpose, for memory capacity or bandwidth capacity.
Process and Thread Affinity for Hopper/Edison
• Hopper: NERSC Cray XE6, 6,384 nodes, 153,126 cores.
  • 4 NUMA domains per node, 6 cores per NUMA domain.
• Edison: NERSC Cray XC30, 5,576 nodes, 133,824 cores.
  • 2 NUMA domains per node, 12 cores per NUMA domain.
    2 hardware threads per core.
• Memory bandwidth is non-homogeneous among NUMA domains.
MPI Process Affinity: aprun “-S” Option

- Process affinity: or CPU pinning, binds MPI process to a CPU or a ranges of CPUs on the node.
- Important to spread MPI ranks evenly onto different NUMA nodes.
- Use the “-S” option for Hopper/Edison.

GTC Hybrid MPI/OpenMP on Hopper, 24,576 cores

Run Time (sec)
Thread Affinity: aprun “-cc” Option

- Thread locality is important since it impacts both memory and intra-node performance.
- Thread affinity: forces each process or thread to run on a specific subset of processors, to take advantage of local process state.
- On Hopper/Edison:
  - The default option is -cc cpu (use for non-Intel compilers)
  - Pay attention to Intel compiler, which uses an extra thread.
    - Use “-cc none” if 1 MPI process per node
    - Use “-cc numa_node” (Hopper) or “-cc depth” (Edison) if multiple MPI processes per node
Process and Thread Affinity for Babbage
Babbage MIC Card

Babbage: NERSC Intel Xeon Phi testbed, 45 nodes.

- 1 NUMA domain per MIC card: 60 physical cores, 240 logical cores.
- Process affinity: spread MPI process onto different physical cores.
- Logical core 0 is on physical core 60.
Thread Affinity: KMP_AFFINITY

- Run Time Environment Variable.
- **none**: no affinity setting. Default setting on the host.
- **compact**: default option on MIC. Bind threads as close to each other as possible

<table>
<thead>
<tr>
<th>Node</th>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HT1</td>
<td>HT2</td>
<td>HT3</td>
</tr>
<tr>
<td>Thread</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

- **scatter**: bind threads as far apart as possible. Default setting on MIC.

<table>
<thead>
<tr>
<th>Node</th>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HT1</td>
<td>HT2</td>
<td>HT3</td>
</tr>
<tr>
<td>Thread</td>
<td>0</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

- **balanced**: only available on MIC. Spread to each core first, then set thread numbers using different HT of same core close to each other.

<table>
<thead>
<tr>
<th>Node</th>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HT1</td>
<td>HT2</td>
<td>HT3</td>
</tr>
<tr>
<td>Thread</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

- **explicit**: example: setenv KMP_AFFINITY “explicit, granularity=fine, proclist=[1:236:1]”
- New env on coprocessors: KMP_PLACE_THREADS, for exact thread placement
Thread Affinity: KMP_PLACE_THREADS

• **New setting on MIC only.** In addition to KMP_AFFINITY, can set exact but still generic thread placement.

• **KMP_PLACE_THREADS** = 
  $$<n>Cx<m>T,<o>O$$
  
  – $$<n>$$ Cores times $$<m>$$ Threads with $$<o>$$ of cores Offset
  
  – e.g. 40Cx3T,1O means using 40 cores, and 3 threads (HT2,3,4) per core

• **OS runs on logical proc 0, which lives on physical core 60**
  
  – OS procs on core 60: 0,237,238,239.
  
  – Avoid use proc 0
A domain is a group of logical cores
- Domains are non-overlapping
- Number of logical cores per domain is a multiple of 4
- 1 MPI process per domain
- OpenMP threads can be pinned inside each domain

\[
\text{I_MPI_PIN_DOMAIN} = \langle \text{size} \rangle[:\langle \text{layout} \rangle]
\]

\[
\begin{align*}
\langle \text{size} \rangle &= \text{omp} \quad \text{adjust to OMP\_NUM\_THREADS} \\
&= \text{auto} \quad \#\text{CPUs}/\#\text{MPI procs} \\
&= \langle n \rangle \quad \text{a number}
\end{align*}
\]

\[
\begin{align*}
\langle \text{layout} \rangle &= \text{platform} \quad \text{according to BIOS numbering} \\
&= \text{compact} \quad \text{close to each other} \\
&= \text{scatter} \quad \text{far away from each other}
\end{align*}
\]
Tools for OpenMP
Adding OpenMP to Your Program

• On Hopper/Edison, under Cray programming environment, Cray Reveal tool helps to perform scope analysis, and suggests OpenMP compiler directives.
  – Based on CrayPat performance analysis
  – Utilizes Cray compiler optimization information

• On Babbage, Intel Advisor tool helps to guide threading design options.
Performance Analysis And Debugging

• **Performance Analysis**
  – Hopper/Edison:
    • Cray Performance Tools
    • IPM
    • Allinea MAP, perf-reports
    • TAU
  – Babbage:
    • Vtune
    • Intel Trace Analyzer and Collector
    • HPCToolkit
    • Allinea MAP

• **Debugging**
  – Hopper/Edison: DDT, Totalview, LGDB, Valgrind
  – Babbage: Intel Inspector, GDB, DDT
Case Studies
Case Studies Introduction

- **OpenMP parallelizing techniques used in real codes.**
- **LBM on TACC Stampede** *(by Carlos Rosales, TACC)*
  - Add OpenMP incrementally
  - Compare OpenMP affinity
- **MFDn on Hopper** *(by H. Metin Aktulga et al., LBNL)*
  - Overlap communication and computation
- **NWChem on Babbage** *(by Hongzhang Shan et al., LBNL)*
  - CCSD(T)
    - Add OpenMP at the outermost loop level
    - Loop permutation, collapse
    - Reduction, remove loop dependency
  - Fock Matrix Construction (FMC)
    - Add OpenMP to most time consuming functions
    - OpenMP Task
    - Find sweet scaling spot with hybrid MPI/OpenMP
Case Study #1: LBM, Add OpenMP Incrementally

- Actual serial run time for Collision > 2500 sec *(plotted above as 200 sec only for better display)*, > 95% of total run time.
- Step 1: Add OpenMP to hotspot Collision. 60X Collision speedup.
- Step 2: Add OpenMP to the new bottleneck, Stream and others. 89X Stream speedup.
- Step 3: Add vectorization. 5X Collision speedup.
- Balanced provides best performance overall.
Case Study #2: MFDn, Overlap Comm and Comp

- MFDn: a nuclear physics code.
- Hopper. Pure MPI: 12,096 MPI tasks.
- Hybrid A: hybrid MPI/OpenMP, 2016 MPI* 6 threads.
- Hybrid B: hybrid A, plus: merge MPI_Reduce and MPI_Scatter into MPI_Reduce_Scatter, and merge MPI_Gather and MPI_Bcast into MPI_Allgatherv.
- Hybrid C: Hybrid B, plus: overlap row-group communications with computation.
- Hybrid D: Hybrid C, plus: overlap (most) column-group communications with computation.

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

- Need at least `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.
Case Study #3: NWChem CCSD(T), Baseline OpenMP

- Due to memory limitation, can only run with 1 MPI process per MIC.
- OpenMP added at the outermost loops of hotspots: Loop Nests. Scales well up to 120 threads.
- GetBlock is not parallelized with OpenMP. Hyper-threading hurts performance.
- Total time has perfect scaling from 1 to 16 threads. Best time at 120 threads.
- Balanced affinity gives best performance.
Case Study #3: NWChem CCSD(T), More OpenMP Optimizations

- GetBlock optimizations: parallelize sort, loop unrolling.
- Reorder array indices to match loop indices.
- Merge adjacent loop indices to increase number of iterations.
- Align arrays to 64 bytes boundary.
- Exploit OpenMP loop control directive, provide compiler hints.
- Total speedup from base is 2.3x.
Case Study #4: NWChem FMC, Add OpenMP to HotSpots (OpenMP #1)

- Total number of MPI ranks=60; OMP=N means N threads per MPI rank.
- Original code uses a shared global task counter to deal with dynamic load balancing with MPI ranks.
- Loop parallelize top 10 routines in TEXAS package (75% of total CPU time) with OpenMP. Has load-imbalance.
- OMP=1 has overhead over pure MPI.
- OMP=2 has overall best performance in many routines.
Case Study #4: NWChem FMC, OpenMP Task Implementation (OpenMP #3)

**Fock Matrix Construction — OpenMP Task Implementation**

```c
c$OMP parallel
myfock() = 0

; $OMP master
current_task_id = 0
mytid = omp_get_thread_num()
My_task = global_task_counter(task_block_size)
for ijk = 2*ntype to 2 step -1 do
  for ij = min(ntype, ijk - 1) to max(1, ijk - ntype) step -1 do
    kl = ijk - ij
    if (my_task.eq. current_task_id) then
      c$OMP task firstprivate(ij,kl) default(shared)
      create_task(ij,kl, ...)
      c$OMP end task
      my_task=global_task_counter(task_block_size)
    end if
    current_task_id = current_task_id + 1
  end for
end for

; $OMP end master
; $OMP taskwait
; $OMP end parallel
Perform Reduction on myfock to Fock matrix
```

- OpenMP task model is flexible and powerful.
- The task directive defines an explicit task.
- Threads share work from all tasks in the task pool.
- Master thread creates tasks.
- The taskwait directive makes sure all child tasks created for the current task finish.
- Helps to improve load balance.

**Use OpenMP tasks.**
- To avoid two threads updating Fock matrix simultaneously, a local copy is used per thread. Reduction at the end.
Case Study #4: NWChem FMC, Run Time

**Fock Matrix Construction Time**

- **Flat MPI** is limited to a total of 60 ranks due to memory limitation.
- **OpenMP #1** uses flat MPI up to 60 MPI processes, then uses 2, 3, and 4 threads per MPI rank.
- **OpenMP #2** and #3 are pure OpenMP.
- **OpenMP #2** module-level parallelism saturates at 8 threads (critical and reduction related). Then when over 60 threads, hyper-threading helps.
- **OpenMP #3** Task implementation continues to scale over 60 cores. 1.33x faster (with 180 threads) than pure MPI.
- The OpenMP Task implementation benefits both MIC and Host.
Case Study #4: NWChem FMC, MPI/OpenMP Scaling and Tuning

- Another way of showing scaling analysis result.
- Sweet spot is either 4 MPI tasks with 60 OpenMP threads per task, or 6 MPI tasks with 40 OpenMP threads per task.
- 1.64x faster than original flat MPI.
- 22% faster than 60 MPI tasks with 4 OpenMP threads per task.
Summary

• Use Edison/Babbage to help you to prepare for Cori regarding thread scalability (hybrid MPI/OpenMP implementation).
  – MPI performance across nodes or MIC cards on Babbage is not optimal.
  – Concentrate on optimization on single MIC card.

• Case studies showed effectiveness of OpenMP
  – Add OpenMP incrementally. Conquer one hotspot at a time.
  – Experiment with thread affinity choices. Balanced is optimal for most applications. Low hanging fruit.
  – Pay attention to cache locality and load balancing. Adopt loop collapse, loop permutation, etc.
  – Find sweet spot with MPI/OpenMP scaling analysis.
  – Consider OpenMP TASK. Major code rewrite.
  – Consider overlap communication with computation. Very hard to do.

• Optimizations targeted for one architecture (XE6, XC30, KNC) can help performance for other architectures (Xeon, XC30, KNL).
References

• NERSC Hopper/Edison/Babbage web pages:
  – https://www.nersc.gov/users/computational-systems/hopper
  – https://www.nersc.gov/users/computational-systems/edison
  – https://www.nersc.gov/users/computational-systems/testbeds/babbage

• OpenMP Resources:
  – https://www.nersc.gov/users/computational-systems/edison/programming/using-openmp/openmp-resources/

• Cray Reveal at NERSC:


• Jim Jeffers and James Reinders, “Intel Xeon Phi Coprocessor High-Performance Programming”. Published by Elsevier Inc. 2013.

• Intel Xeon Phi Coprocessor Developer Zone:
  – http://software.intel.com/mic-developer

• Programming and Compiling for Intel MIC Architecture

• Interoperability with OpenMP API

• Intel Cluster Studio XE 2015