Hybrid Programming with MPI and OpenMP

B. Estrade
<estrab@lsu.edu>
Objectives

- understand the difference between message passing and shared memory models;
- learn of basic models for utilizing both message passing and shared memory approaches to parallel programming;
- learn how to program a basic hybrid program and execute it on local resources;
- learn of basic issues regarding the *hybridization* of existing serial, all-MPI, or all-OpenMP codes;

What is *Hybridization*?

- the use of inherently different models of programming in a complementary manner, in order to achieve some benefit not possible otherwise;
- a way to use different models of parallelization in a way that takes advantage of the good points of each;
- also known as “mixed mode” programming
How Does Hybridization Help When...

- **introducing MPI into OpenMP:**
  - can help applications scale across multiple SMP nodes;
  - this is not easy because it entails the rethinking of most of the implicit parallelism, which is also what is needed when starting from scratch.

- **introducing OpenMP into MPI:**
  - can help applications make more efficient use of the shared memory on SMP nodes, thus mitigating the need for explicit intra-node communication;
  - straightforward since in many cases, it is similar to introducing OpenMP into serial code.

- **introducing MPI and OpenMP:**
  - will allow most of the effort to go into designing the right balance between shared memory computations and the associated message passing overhead.
When Does Hybridization Make Sense?

• when one wants to scale a shared memory OpenMP application for use on multiple SMP nodes in a cluster;

• when one wants to reduce an MPI application's sensitivity to becoming communication bound;

• when one is designing a parallel program from the very beginning to maximize utilization of a distributed memory machine consisting of individual SMP nodes;
Hybridization Using MPI and OpenMP

- facilitates cooperative shared memory (OpenMP) programming across clustered SMP nodes;
- MPI facilitates communication among SMP nodes, including the efficient packing and sending of complex data structures;
- OpenMP manages the workload on each SMP node;
- MPI and OpenMP are used in tandem to manage the overall concurrency of the application;
MPI

- provides a familiar and explicit means to use message passing on distributed memory clusters;
- has implementations on many architectures and topologies;
- specializes in packing and sending complex data structures over the network;
- is the de facto standard for distributed memory communications;
- requires that program state synchronization must be handled explicitly due to the nature of distributed memory;
- data goes to the process;
- program correctness is an issue, but not big compared to those inherent to OpenMP;
OpenMP

- allows for implicit intra-node communication, which is a *shared memory* paradigm;
- provides for efficient utilization of shared memory SMP systems;
- facilitates relatively easy threaded programming;
- does not incur the overhead of message passing, since communication among threads is implicit;
- is the defacto standard, and is supported by most major compilers (Intel, IBM, gcc, etc);
- the process goes to the data
- program correctness is an issue since all threads can update shared memory locations;
The Best From Both Worlds

- MPI makes inter-node communication relatively easy;
- MPI facilitates efficient inter-node scatters, reductions, and sending of complex data structures;
- Since program state synchronization is done explicitly with messages, correctness issues are relatively easy to avoid;

- OpenMP allows for high performance, and relatively straightforward, intra-node threading;
- OpenMP provides an interface for the concurrent utilization of each SMP's shared memory, which is much more efficient that using message passing;
- Program state synchronization is implicit on each SMP node, which eliminates much of the overhead associated with message passing;
Thread Safety & MPI

- [1] points out that MPI implementations are not required to be “thread safe”
- so what?
- this means that MPI calls made within OpenMP threads must be done so inside of a CRITICAL, MASTER, or SINGLE section
- this also means that a model that has all threads making MPI calls, one should use a thread safe implementation of MPI – otherwise, judicious use of the blocks listed above must be made
- the MPI-2 standard addresses the issue of thread safety, per [2]
- MPI-1 does not address this, by the MPICH1 implementation is supposed to have modes which make it safe, per [3]
- the point: be (thread) safe from top to bottom!
A Common Execution Scenario

1) a single MPI process is launched on each SMP node in the cluster;

2) each process spawns N threads on each SMP node;

3) at some global sync point, the master thread on MPI process 0 communicates with the master thread on all other nodes

4) the threads belonging to each process continue until another sync point or completion;
What Does This Scenario Look Like?

P_0

SMP 0

t_0

P_1

SMP 1

t_1

P_2

SMP 2

t_2

P_3

SMP 3

t_3
Basic Hybrid “Stub”

```c
#include <omp.h>
#include "mpi.h"
#include <stdio.h>
#define _NUM_THREADS 4

/* Each MPI process spawns a distinct OpenMP
 * master thread; so limit the number of MPI
 * processes to one per node
 */

int main (int argc, char *argv[]) {
    int p,my_rank,c;
    /* set number of threads to spawn */
    omp_set_num_threads(_NUM_THREADS);

    /* initialize MPI stuff */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

    /* the following is a trivial parallel OpenMP
     * executed by each MPI process
     */

    #pragma omp parallel reduction(+:c)
    { c = omp_get_num_threads();
    }

    /* expect a number to be printed for each MPI process */
    printf("%d\n",c);
    /* finalize MPI */
    MPI_Finalize();
    return 0;
}
```

It is recommended by [1] that the number of threads be set with `omp_set_num_threads` because `OMP_NUM_THREADS` is considered non-portable.
Compiling

- IBM p5 575s:
  - mpcc_r, mpCC_r, mpxlf_r, mpxlf90_r, mpxlf95_r

```
bash
%mpcc_r -qsmp=omp test.c
%OMP_NUM_THREADS=4 poe ./a.out -rmpool 1 -nodes 1 -procs 2
```

- x86 Clusters:
  - mpicc, mpiCC, mpicxx, mpif77, mpif90

```
bash
%mpicc -openmp test.c
%OMP_NUM_THREADS=4 mpirun -np 2 -machinefile $PBS_NODFILE ./a.out
```
Special Considerations

- processor “affinity” is becoming increasingly important in multi-core and virtualized SMP environments
- this is accounted for with the MVAPICH implementation of MPI used at LSU/LONI
- the following environmental variables must be set accordingly to make sure that the threads on each compute node are not bound to a single core – this will devastate performance
- these variables, for MVAPICH1 and MVAPICH2, respectively are:
  
  ```bash
  VIADEV_USE_AFFINITY=0  # for MVAPICH1
  MV2_ENABLE_AFFINITY=0  # for MVAPICH2
  ```
#PBS (Linux)

```bash
#!/bin/bash
#PBS -q checkpt
#PBS -A your_allocation
#PBS -l nodes=4:ppn=8
#PBS -l cput=2:00:00
#PBS -l walltime=2:00:00
#PBS -o /work/yourdir/myoutput2
#PBS -j oe # merge stdout and stderr
#PBS -N myhybridapp
export WORK_DIR=/work/yourdir
# create a new machinefile file which only contains unique nodes
cat $PBS_NODEFILE | uniq > hostfile
# get number of MPI processes and create proper machinefile
export NPROCS=`wc -l hostfile | gawk '//{print $1}'`
ulimit -s hard
# setting number of OpenMP threads
cd $WORK_DIR
# OpenMP/MPI settingsd
export VIADEV_USE_AFFINITY=0 # affinity control for mvapich1
export MV2_ENABLE_AFFINITY=0 # affinity control for mvapich2
export OMP_NUM_THREADS=8 # should use omp_set_num_threads be used internally
mpirun -machinefile ./hostfile -np $NPROCS ./hybrid.x
```

*Shangli Ou, https://docs.loni.org/wiki/Running_a_MPI/OpenMP_hybrid_Job*
LoadLeveler (AIX)

```
#!/usr/bin/ksh
# @ job_type = parallel
# @ input = /dev/null
# @ output = /work/default/ou/flower/output/out.std
# @ error = /work/default/ou/flower/output/out.err
# @ initialdir = /work/default/ou/flower/run
# @ notify_user = ou@baton.phys.lsu.edu
# @ class = checkpoint
# @ notification = always
# @ checkpoint = no
# @ restart = no
# @ wall_clock_limit = 10:00:00
# @ node = 4,4
# @ network.MPI = sn_single,shared,US
# @ requirements = ( Arch == "Power5" )
# @ node_usage = not_shared
# @ tasks_per_node = 1
# @ environment=MP_SHARED_MEMORY=yes; COPY_ALL
# @ queue
# the following is run as a shell script
export OMP_NUM_THREADS=8
mpirun -NP 4 ./hybrid.x
```

*Shangli Ou, https://docs.ioni.org/wiki/Running_a_MPI/OpenMP_hybrid_Job*
Retro-fitting MPI Apps With OpenMP

- involves most commonly the work-sharing of simple loops;
- is the easiest of the two “retro-fit” options because the program state synchronization is already handled in an explicit way; adding OpenMP directives admits the need for implicit state synchronization, which is easier;
- each MPI process is already encapsulated wrt other processes, so threading it does not necessarily run the risk of producing side effects in other MPI processes;
- benefits depend on how many simple loops may be work-shared; otherwise, the effects tend towards using fewer MPI processes;
- the number of MPI processes per SMP node will depend on how many threads one wants to use per process;
- most beneficial for communication bound applications, since it reduces the number of MPI processes needing to communicate; however, CPU processor utilization on each node becomes an issue;
Retro-fitting OpenMP Apps With MPI

- not as straightforward as retro-fitting an MPI application with OpenMP because global program state must be explicitly handled with MPI;
- OpenMP applications make use of *side effects* across threads, so lumping subsets of tightly coupled threads into MPI process is a hard – and often times, not even possible
- requires careful thought about how each process will communicate amongst one another;
- may require a complete reformulation of the parallelization, with a need to possibly redesign it from the ground up;
- successful retro-fitting of OpenMP applications with MPI will usually yield greater improvement in performance and scaling, presumably because the original shared memory program takes great advantage of the entire SMP node;
General Retro-fitting Guidelines

- adding OpenMP to MPI applications is fairly straightforward because the distributed memory of multiple SMP nodes has already been handled;
- MPI applications that are communication bound and have many simple loops that may be work-shared will benefit greatly due to the reduction in need for communication among SMP nodes;
- adding MPI to OpenMP applications is not very straightforward, but will yield better scaling and higher performing application in many cases;
- OpenMP applications handle program state implicitly, thus introducing MPI requires the explicit handling of program state - which is not easy do “bolt on” after the fact;
- in general, adding MPI to OpenMP applications should initiate a redesign of the application from the ground up in order to handle the need for explicit synchronizations across distribute memory;
- fortunately, much of the old OpenMP application code may be reused;
Designing Hybrid Apps From Scratch

• redesigning an application, whether originally using OpenMP or MPI, is the ideal situation; although, this is not always possible or desired;

• benefits are greatest when considering the introduction of MPI into shared memory programs;

• great care should be taken to find the right balance of MPI computation and OpenMP “work”; it is the shared memory parts that do the work; MPI is used to simply keep everyone on the same page;

• Prioritized list of some considerations
  
i. the ratio of communication among nodes and time spend keeping the processors on a single node should be minimized in order to maximize scaling;

  ii. the shared memory computations on each node should utilize as many threads as possible during the computation parts;

  iii. MPI is most efficient at communicating a small number of larger data structures; therefore, many small messages will introduce a communication overhead unnecessarily;
Example Concept 1

ROOT MPI Process Controls All Communications

- most straightforward paradigm;
- maps one MPI process to one SMP node;
- each MPI process spawns a fixed number of shared memory threads;
- communication among MPI processes is handled by the main MPI process only, at fixed predetermined intervals;
- allows for tight control of all communications;

```
// do only if master thread, else wait
#pragma omp master
{ if (0 == my_rank)
    // some MPI_ call as ROOT process
  else
    // some MPI_ call as non-ROOT process
}
// end of omp master
```
What Does Example 1 Look Like?
#include <omp.h>
#include "mpi.h"
#include <stdio.h>
#define _NUM_THREADS 4

int main (int argc, char *argv[]) {
    int p, my_rank, c;

    /* set number of threads to spawn */
   omp_set_num_threads(_NUM_THREADS);

    /* initialize MPI stuff */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* the following is a parallel OpenMP */
    /* executed by each MPI process */
    #pragma omp parallel reduction(+:c)
    {
        #pragma omp master
        {
            if ( 0 == my_rank)
                // some MPI_ call as ROOT process
                c = 1;
            else
                // some MPI_ call as non-ROOT process
                c = 2
        }
    }

    /* expect a number to get printed for each MPI process */
    printf("%d\n", c);

    /* finalize MPI */
    MPI_Finalize();
    return 0;
}
Example Concept 2

Master OpenMP Thread Controls All Communications

- each MPI process uses its own OpenMP master thread (1 per SMP node) to communicate;
- allows for more asynchronous communications;
- not nearly as rigid as example 1;
- more care needs to be taken to ensure efficient communications, but the flexibility may yield efficiencies elsewhere;

```c
// do only if master thread, else wait
#pragma omp master
{
    // some MPI_call as an MPI process
}
// end of omp master
```
What Does a Example 2 Look Like?

![Diagram of example](image)
Example 2 Code Stub

```c
#include <omp.h>
#include "mpi.h"
#include <stdio.h>
#define _NUM_THREADS 4

int main (int argc, char *argv[]) {
    int p, my_rank;
    int c = 0;
    /* set number of threads to spawn */
    omp_set_num_threads(_NUM_THREADS);

    /* initialize MPI stuff */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* the following is a parallel OpenMP */
    /* executed by each MPI process */
    #pragma omp parallel
    {
        #pragma omp master
        {
            // some MPI_call as an MPI process
            c = 1;
        }
    }

    /* expect a number to get printed for each MPI process */
    printf("%d\n", c);
    /* finalize MPI */
    MPI_Finalize();
    return 0;
}
```
Example Concept 3
All OpenMP Threads May Use MPI Calls

• this is by far the most flexible communication scheme;
• enables true distributed behavior similar to that which is possible using pure MPI;
• the greatest risk of inefficiencies are contained using this approach;
• great care must be made in explicitly accounting for which thread of which MPI process is communication;
• requires a addressing scheme that denotes the tuple of which MPI processes participating in communication and which thread of the MPI process is involved; e.g., <my_rank,omp_thread_id>;
• neither MPI nor OpenMP have built-in facilities for tracking this;
• critical sections, potentially named, may have to be utilized for some level of control and correctness since MPI implementations are not assured to be thread safe [1]!
What Does Example 3 Look Like?

---

**Diagram Description:**

- The diagram illustrates a four-node system with P0, P1, P2, and P3 nodes connected at each time step t0, t1, t2, and t3.
- SMP 0 through SMP 3 are connected to specific nodes at each time step, indicating parallel processing.

---

B. Estrade <estrabd@lsu.edu>, HPC @ LSU - High Performance Computing Workshop
Example 3 Code Stub

```c
#include <omp.h>
#include "mpi.h"
#include <stdio.h>
#define _NUM_THREADS 4

int main (int argc, char *argv[]) {
    int p, my_rank;
    int c = 0;
    /* set number of threads to spawn */
    omp_set_num_threads(_NUM_THREADS);

    /* initialize MPI stuff */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* the following is a parallel OpenMP
     * executed by each MPI process
     */
    #pragma omp parallel
    {
        #pragma omp critical /* not required */
        {
            // some MPI_ call as an MPI process
            c = 1;
        }
    }

    /* expect a number to get printed for each MPI process */
    printf("%d\n", c);
    /* finalize MPI */
    MPI_Finalize();
    return 0;
}
```
Comparison of Examples

1. // do only if master thread, else wait
   #pragma omp master
   {
      if (0 == my_rank)
         // some MPI_ call as ROOT process
      else
         // some MPI_ call as non-ROOT process
   }
   // end of omp master

2. // do only if master thread, else wait
   #pragma omp master
   {
         // some MPI_ call as an MPI process
   }
   // end of omp master

3. // each thread makes a call; can utilize
   // critical sections for some control
   #pragma omp critical
   {
         // some MPI_ call as an MPI process
   }
General Design Guidelines

• the ratio of communications to time spent computing on each SMP node should be minimized in order to improve the scaling characteristics of the hybrid code;

• introducing OpenMP into MPI is much easier, but the benefits are not as great or likely as vice-versa;

• the greatest benefits are seen when an application is redesigned from scratch; fortunately, much of the existing code is salvageable;

• there are many, many communication paradigms that may be employed; we covered just 3; it is prudent to investigate all options;

• great care must be taken to ensure program correctness and efficient communications;
Summary

- simply compiling MPI and OpenMP into the same program is easy;
- adding OpenMP to an MPI app is easy, but the benefits may not be *that* great (but give it a shot!);
- adding MPI to an OpenMP app is hard, but usually worth it;
- designing a hybrid application from scratch is ideal, and allows one to best balance the strengths of both MPI and OpenMP to create an optimal performing and scaling application;
- there are a lot of schemes that incorporate both shared and distributed memory, so it is worth the time to investigate them wrt the intended applications;
References/Additional Resources

- http://docs.loni.org
- sys-help@loni.org