Hybrid Programming with MPI and OpenMP

B. Estrade
<estrabd@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 complimentary 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;
How Does Hybridization Help?

- introducing MPI into OpenMP applications can help scale across multiple SMP nodes;
- introducing OpenMP into MPI applications can help make more efficient use of the shared memory on SMP nodes, thus mitigating the need for explicit intra-node communication;
- introducing MPI and OpenMP during the design/coding of a new application can help maximize efficiency, performance, and scaling;
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;
Hybridization Using MPI and OpenMP

- facilitates cooperative shared memory (OpenMP) programming across clustered SMP nodes;
- MPI facilitates communication among SMP nodes;
- 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;
- is the defacto 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 allows for inter-node communication;
- MPI facilitates efficient inter-node reductions and sending of complex data structures;
- Program state synchronization is explicit;

- OpenMP allows for high performance intra-node threading;
- OpenMP provides an interface for the concurrent utilization of each SMP's shared memory;
- Program state synchronization is implicit;
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 each SMP communicate with one another;
4) the threads belonging to each process continue until another sync point or completion;
What Does This Scenario Look Like?

\[ P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow P_3 \]

\[ t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow t_3 \]

SMP 0 \rightarrow SMP 1 \rightarrow SMP 2 \rightarrow SMP 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 parallel OpenMP
     * executed by each MPI process
     */

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

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

cat $PBS_NODEFILE | uniq > hostfile

export NPROCS=`wc -l hostfile | gawk '//{print $1}'`
ulimit -s hard

export OMP_NUM_THREADS=8
mpirun -machinefile ./hostfile -np $NPROCS ./hybrid.x
```

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

```bash
#!/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 = checkpt
# @ 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.loni.org/wiki/Running_a_MPI/OpenMP_hybrid_Job*
Retro-fitting MPI Apps With OpenMP

- involves most commonly the work-sharing of simple looks;
- 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;
- 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;
- 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 to 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;

```c
// 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?

Diagram showing the execution of processes on different SMPs over time.

- **P₀**: Process 0 at different time steps (t₀, t₁, t₂, t₃).
- **SMP 0, 1, 2, 3**: Separate SMPs with processes executing in parallel.

Diagram illustrates the parallel execution of processes across SMPs.
Example 1 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,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?
Example 2 Code Stub

#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 be utilized for some level of control and correctness;*
What Does Example 3 Look Like?

P₀  P₁  P₂  P₃

SMP 0  SMP 1  SMP 2  SMP 3

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

<table>
<thead>
<tr>
<th>Example</th>
<th>Code Explanation</th>
</tr>
</thead>
</table>
| 1.      | `#pragma omp master`<br>```c
{ if (0 == my_rank)
   // some MPI_ call as ROOT process
   else
       // some MPI_ call as non-ROOT process
}
// end of omp master
``` |
| 2.      | `#pragma omp master`<br>```c
{ // some MPI_ call as an MPI process
}
// end of omp master
``` |
| 3.      | `#pragma omp critical`<br>```c
{ // 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;
Additional Resources

- http://docs.loni.org
  - https://docs.loni.org/wiki/Running_a_MPI/OpenMP_hybrid_Job
- sys-help@loni.org
- otrs@loni.org