BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI (RAJASTHAN)

CS F422 – Parallel Computing

Lab#7

Note: Please use programs under *Code* directory supplied with this sheet. Do not copy from this sheet.

The lab has the following objectives:
Practice programs for Hybrid MPI OpenMP.

1. Hybrid MPI-OpenMP

Large-scale parallel computers are nowadays exclusively of the distributed-memory type at the overall system level but use shared-memory compute nodes as basic building blocks. Even though these hybrid architectures have been in use for more than a decade, most parallel applications still take no notice of the hardware structure and use pure MPI for parallelization. Application developers confided in the MPI library providers to deliver efficient MPI implementations, which put the full capabilities of a shared-memory system to use for high-performance intranode message passing. It is more than doubtful whether the attitude of running one MPI process per core is appropriate in the era of multicore processors. The parallelism within a single chip will steadily increase, and the shared-memory nodes will have highly parallel, hierarchical, multicore multisocket structures.

The basic idea of a hybrid OpenMP/MPI programming model is to allow any MPI process to spawn a team of OpenMP threads in the same way as the master thread does in a pure OpenMP program. Thus, inserting OpenMP compiler directives into an existing MPI code is a straightforward way to build a first hybrid parallel program. Following the guidelines of good OpenMP programming, compute intensive loop constructs are the primary targets for OpenMP parallelization in a naïve hybrid code. Before launching the MPI processes one has to specify the maximum number of OpenMP threads per MPI process in the same way as for a pure OpenMP program. At execution time each MPI process activates a team of threads (being the master thread itself) whenever it encounters an OpenMP parallel region.

There two basic hybrid programming approaches: Vector mode and task mode

Vector mode implementation

In a vector mode implementation all MPI subroutines are called outside OpenMP parallel regions, i.e., in the “serial” part of the OpenMP code. E.g. Add OpenMP worksharing directives in front of the time-consuming loops.

Consider Jacobi iteration for approximating the solution to a linear system of equations. The following code gives implementation in MPI.

```
#include <stdio.h>
```

```c
#include <math.h>
#include "mpi.h"

/* This example handles a 12 x 12 mesh, on 4 processors only. */
#define maxn 12

int main( argc, argv )
int argc;
char **argv;
{
    int         rank, value, size, errcnt, toterr, i, j, itcnt;
    int         i_first, i_last;
    MPI_Status status;
    double      diffnorm, gdiffnorm;
    double      xlocal[(12/4)+2][12];
    double      xnew[(12/3)+2][12];

    MPI_Init( &argc, &argv );

    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );

    if (size != 4) MPI_Abort( MPI_COMM_WORLD, 1 );

    /* xlocal[][0] is lower ghostpoints, xlocal[][maxn+2] is upper */

    /* Note that top and bottom processes have one less row of interior
       points */
    i_first = 1;
    i_last  = maxn/size;
    if (rank == 0)          i_first++;
    if (rank == size - 1) i_last--;

    /* Fill the data as specified */
    for (i=1; i<=maxn/size; i++)
        for (j=0; j<maxn; j++)
            xlocal[i][j] = rank;
    for (j=0; j<maxn; j++) {
        xlocal[i_first-1][j] = -1;
        xlocal[i_last+1][j] = -1;
    }

    itcnt = 0;
    do {
        /* Send up unless I'm at the top, then receive from below */
        /* Note the use of xlocal[i] for &xlocal[i][0] */
        if (rank < size - 1)
            MPI_Send( xlocal[maxn/size], maxn, MPI_DOUBLE, rank + 1, 0,
                      MPI_COMM_WORLD );
        if (rank > 0)
            MPI_Recv( xlocal[0], maxn, MPI_DOUBLE, rank - 1, 0,
                      MPI_COMM_WORLD, &status );
        /* Send down unless I'm at the bottom */
        if (rank > 0)
            MPI_Send( xlocal[1], maxn, MPI_DOUBLE, rank - 1, 1,
                      MPI_COMM_WORLD );
        if (rank < size - 1)
            MPI_Recv( xlocal[maxn/size+1], maxn, MPI_DOUBLE, rank + 1, 1,
                      MPI_COMM_WORLD, &status );

        /* Compute new values (but not on boundary) */
        itcnt ++;
        diffnorm = 0.0;
        for (i=i_first; i<=i_last; i++)
            for (j=1; j<maxn-1; j++) {
                xnew[i][j] = (xlocal[i][j+1] + xlocal[i][j-1] +
                             xlocal[i+1][j] + xlocal[i-1][j]) / 4.0;
                diffnorm += (xnew[i][j] - xlocal[i][j]) *
                            (xnew[i][j] - xlocal[i][j]);
            }
        /* Only transfer the interior points */
        for (i=i_first; i<=i_last; i++)
            for (j=1; j<maxn-1; j++)
                xlocal[i][j] = xnew[i][j];

        MPI_Allreduce( &diffnorm, &gdiffnorm, 1, MPI_DOUBLE, MPI_SUM,
                       MPI_COMM_WORLD );
        gdiffnorm = sqrt( gdiffnorm );
        if (rank == 0) printf( "At iteration %d, diff is %e\n", itcnt,
                               gdiffnorm );
    } while (gdiffnorm > 1.0e-2 && itcnt < 100);

    MPI_Finalize( );
    return 0;
}
```

Q?

1. Compile the program given in jacobi.c. Measure time.

2. Compile the program given in Jacobi_vector_openmp.c. Measure time. Do you find any difference?
3. Modify the program in (2), by adding more openMP directives.

Task mode implementation

The task mode is most general and allows any kind of MPI communication within OpenMP parallel regions. Based on the thread safety requirements for the message passing library, the MPI standard defines three different levels of interference between OpenMP and MPI. Functional task decomposition and decoupling of communication and computation are two areas where task mode can be useful.

- MPI_THREAD_SINGLE: Only one thread will execute.
- MPI_THREAD_FUNNELED: The process may be multithreaded, but only the main thread will make MPI calls.
- MPI_THREAD_SERIALIZED: The process may be multithreaded, and multiple threads may make MPI calls, but only one at a time; MPI calls are not made concurrently from two distinct threads.
- MPI_THREAD_MULTIPLE: Multiple threads may call MPI anytime, with no restrictions

The MPI_Init_thread (mpich.org) can be used to set or get the required level supported by the MPI library.

Consider the following program given in mpi_hello.c. Here MPI_Recv is being called in a loop.

```

/* File:
 *     mpi_hello.c
 *
 * Purpose:
 *     A "hello,world" program that uses MPI
 *
 * Compile:
 *     mpicc -g -Wall -std=C99 -o mpi_hello mpi_hello.c
 * Usage:
 *     mpiexec -n<number of processes> ./mpi_hello
 *
 * Input:
 *     None
 * Output:
 *     A greeting from each process
 *
 * Algorithm:
 *     Each process sends a message to process 0, which prints
 *    the messages it has received, as well as its own message.
 *
 * IPP:  Section 3.1 (pp. 84 and ff.)
 */
#include <stdio.h>
#include <string.h>  /* For strlen             */
#include <mpi.h>     /* For MPI functions, etc */

const int MAX_STRING = 100;

int main(void) {
   char       greeting[MAX_STRING];  /* String storing message */
   int        comm_sz;               /* Number of processes    */
   int        my_rank;               /* My process rank        */

   /* Start up MPI */
   MPI_Init(NULL, NULL);

   /* Get the number of processes */
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

   /* Get my rank among all the processes */
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank != 0) {
      /* Create message */
      sprintf(greeting, "Greetings from process %d of %d!",
            my_rank, comm_sz);
      /* Send message to process 0 */
      MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
            MPI_COMM_WORLD);
   } else {
      /* Print my message */
      printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
      for (int q = 1; q < comm_sz; q++) {
         /* Receive message from process q */
         MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
            0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         /* Print message from process q */
         printf("%s\n", greeting);
      }
   }

   /* Shut down MPI */
   MPI_Finalize();
65.
66.    return 0;
67.}  /* main */
```

Q?

1. Can the given loop (line 54) be parallelized using opneMP directives? Which level of support would work?
2. Consider using OpenMP directivbes in point2point.c. Check any performance improvement.

End of lab