Communication Optimization for Parallel Processing

Lecture 3

Pangfeng Liu, Department of Computer Science and Information Engineering, National Taiwan University.

MPI

- Message Passing Interface
- A standard message passing library for parallel computers
- MPI was designed for high performance on both massively parallel machines and on workstation clusters.
- SPMD programming model
  - Single Program Multiple Data (SPMD)
  - A single program running on different sets of data.

Programming Model

- SPMD
  - Single Program Multiple Data (SPMD)
  - A single program running on different sets of data.
- Languages
  - C/C++
  - Fortran

MPI Related Links

- MPI home pages
  - http://www-nix.mcs.anl.gov/mpi/homepages.html
- MPI forum
  - http://www.mpi-forum.org/
- MPICH
  - http://www-unix.mcs.anl.gov/mpi/mpich
- MPI/LAM
  - http://www.lam-mpi.org/

Initialization and Clean-up

- MPI_Init
  - Initialize the MPI execution environment.
  - The first MPI routine in your program.
  - The argc and argv parameters are from the standard C command line interface.
- MPI_Finalize
  - Terminate MPI execution environment
  - The last statement in your program

Configuration

- MPI_Comm_size
  - Tells the number of processes in the system.
  - The MPI_COMM_WORLD means all the processor in the system.
- MPI_Comm_rank
  - Tells the rank of the calling process.
- MPI_Get_processor_name
  - Get the name of the processor where this process is running.

Compilation

- mpicc
  - Compile your C source code.
- mpiCC
  - Compile your C++ source code.
  - It does not compile C source code.
- mpif77
  - Compile your Fortran 77 source code.
- mpif90
  - Compile your Fortran 90 source code.

Execution

- mpirun
  - An interface to execute your MPI program
  - You can specify the number of processes by the –np option.
  - You may specify the machines you want your processes to run at by the –machinefile option.
    - Our default machine file is /opt/mpich-1.2.1/share/machine.LINUX

Hello, world!

- Our first MPI program
- This program prints the “hello world” message from all processes.
- The number of processes can be specified by the mpirun options.
- Note that the process rank is assigned according to the machine file.

Hello, World!

- This program prints the “hello world” message from all processes.
- The number of processes can be specified by the mpirun options.
- Note that the process rank is assigned according to the machine file.
- hello.c

Print a Message in Parallel

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

int main( int argc, char *argv[])
{
   int n, myid, numprocs, i;
   int  namelen;
   char processor_name[MPI_MAX_PROCESSOR_NAME];

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &myid);
   MPI_Get_processor_name(processor_name, &namelen);

   fprintf(stderr,"Process %d on %s\n", myid, processor_name);
   MPI_Finalize();

   return 0;
}
```

Broadcast and Reduction

- MPI_Bcast
  - broadcasts a buffer to everyone else.
  - The buffer and count indicate the data that will be broadcast.
  - The data type is essential since MPI can deal with different parallel platforms. See Constant.3.ps for details.
  - The root processor must provide the buffer to be sent. All the other provide buffer for receiving.
- MPI_Reduce
  - Reduces the buffers from everybody, and places result into the same buffer of the root process.
  - You may specify the operation on how to “combine” the data provided by processes. See Constant.3.ps for details.

Time Accounting

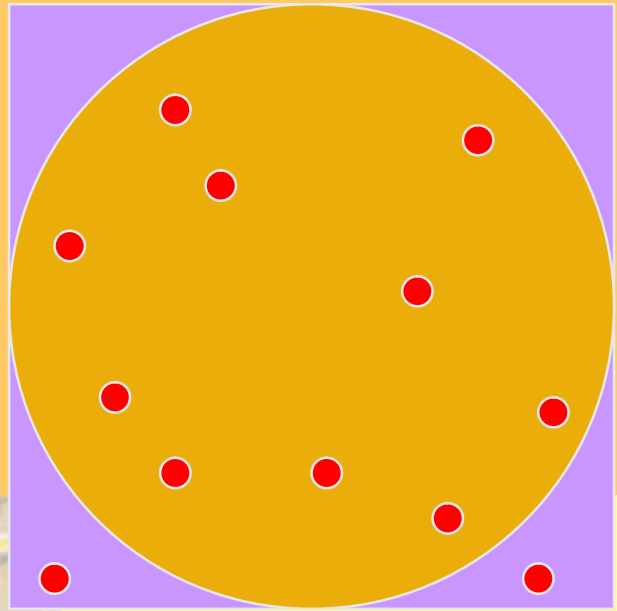
- MPI_Wtime
  - Give the current time in second.
  - Used to compute the elapsed time.

Computing π

- Use MPI_Bcast to send the number of subintervals to all processes.
- Receive and combine the result from processors by MPI_Reduce.
- Uses MPI_Wtime on process 0 to measure the elapsed time.
- The code is cpi.c

Throw a Dart

- Probability = $\pi / 4$
- $\pi = 10 / 12 \times 4 = 3.33$



Variables and Initialization

```
double f( double a )
{
    return (4.0 / (1.0 + a*a));
}

int main( int argc, char *argv[])
{
    int done = 0, n, myid, numprocs, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    double startwtime = 0.0, endwtime;
    int  namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);

    fprintf(stderr,"Process %d on %s\n",  myid, processor_name);
```

Collect the Results

```
n = 0;
 while (!done) {
    if (myid == 0) {
       if (n==0) n=100; else n=0;
       startwtime = MPI_Wtime();
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n == 0)
       done = 1;
    else {
       h   = 1.0 / (double) n;
       sum = 0.0;
       for (i = myid + 1; i <= n; i += numprocs) {
          x = h * ((double)i - 0.5);
          sum += f(x);
       }
       mypi = h * sum;
       MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
       if (myid == 0) {
          printf("pi is approximately %.16f, Error is %.16f\n",  pi, fabs(pi - PI25DT));
          endwtime = MPI_Wtime();
          printf("wall clock time = %f\n", endwtime-startwtime);
       }
    }
 }
 MPI_Finalize();
 return 0;
}
```

Send a Message

- MPI_Send
  - Sends a message to a specified process.
  - The sender can attach a tag to specify the purpose of this message.

Receive a Message

- MPI_Receive
  - Receives a message from a process.
  - The receiver can specify the tag for incoming messages, or use MPI_ANY_TAG to receive any message.
  - The receiver can specify the sender with the source argument, or use MPI_ANY_SOURCE to receive messages from any process sending it the message.
  - The return status contains information for this receiving operation.

Another Hello, World!

- This example is taken from http://www-rcd.cc.purdue.edu/~dseaman/Courses/IntroPar/hello.html
- hello2.c

Another Hello

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

#define MSG_LENGTH 15
main (int argc, char *argv[])
{
   int i, tag=1, tasks, iam;
   char message[MSG_LENGTH];
   MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &tasks);
   MPI_Comm_rank(MPI_COMM_WORLD, &iam);
   if (iam == 0) {
      strcpy(message, "Hello, world!");
      for (i=1; i<tasks; i++)
         MPI_Send(message, MSG_LENGTH, MPI_CHAR, i, tag, MPI_COMM_WORLD);
      } else {
         MPI_Recv(message, MSG_LENGTH, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
      }
      printf("node %d: %s\n", iam, message);
      MPI_Finalize();
}
```

Numerial Integration

- Taken from pp. 57 of Pacheco.
- Each processor computes a particular interval.
- All the results are collected and reported.

Variables

```
main(int argc, char** argv)
{
   int my_rank; /* My process rank */
   int p; /* The number of processes */
   float a = 0.0; /* Left endpoint */
   float b = 1.0; /* Right endpoint */
   int n = 1024; /* Number of trapezoids */
   float h; /* Trapezoid base length */
   float local_a; /* Left endpoint my process */
   float local_b; /* Right endpoint my process */
   int local_n; /* Number of trapezoids for my calculation */
   float integral; /* Integral over my interval */
   float total; /* Total integral */
   int source; /* Process sending integral */
   int dest = 0; /* All messages go to 0 */
   int tag = 0;
   MPI_Status  status;

   MPI_Init(&argc, &argv); /* Let the system do what it needs to start up MPI */
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); /* Get my process rank */
   MPI_Comm_size(MPI_COMM_WORLD, &p); /* Find out how many processes */
```

Integration

```
h = (b-a)/n; /* h is the same for all processes */
local_n = n/p; /* So is the number of trapezoids */

local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
integral = Trap(local_a, local_b, local_n, h);

if (my_rank == 0) {
   total = integral;
   for (source = 1; source < p; source++) {
      MPI_Recv(&integral, 1, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &status);
      total = total + integral;
   }
} else
   MPI_Send(&integral, 1, MPI_FLOAT, dest, tag, MPI_COMM_WORLD);

if (my_rank == 0) {
   printf("With n = %d trapezoids, our estimate\n", n);
   printf("of the integral from %f to %f = %f\n", a, b, total);
}
MPI_Finalize();
} /*  main */
```

Integral Functions

```
float Trap(float local_a, float local_b, int local_n, float h)
{
   float integral; /* Store result in integral */
   float x;
   int i;

   integral = (f(local_a) + f(local_b))/2.0;
   x = local_a;
   for (i = 1; i <= local_n-1; i++) {
      x = x + h;
      integral = integral + f(x);
   }
   integral = integral*h;
   return integral;
} /*  Trap */

float f(float x) {
   float return_val;
   /* Calculate f(x). */
   /* Store calculation in return_val. */
   return_val = x*x;
   return return_val;
} /* f */
```

A Parallel Inner Product

- Taken from Chap 5, pp. 75 & ff in PPMPI.
- The routine Read_vector reads the vector from the user.
  - Process 0 gets the data and sends it to others.
  - All the others receive the data from process 0.
- Each processor uses a serial inner product algorithm to compute the results.

Broadcast the Parameter

```
#define MAX_LOCAL_ORDER 100

main(int argc, char* argv[]) {
   float  local_x[MAX_LOCAL_ORDER];
   float  local_y[MAX_LOCAL_ORDER];
   int    n, n_bar;  /* = n/p */
   float  dot;
   int    p, my_rank;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &p);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   if (my_rank == 0)  {
      printf("Enter the order of the vectors\n");
      scanf("%d", &n);
   }
   MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
   n_bar = n/p;
   Read_vector("the first vector", local_x, n_bar, p, my_rank);
   Read_vector("the second vector", local_y, n_bar, p, my_rank);
   dot = Parallel_dot(local_x, local_y, n_bar);
   if (my_rank == 0)
      printf("The dot product is %f\n", dot);
   MPI_Finalize();
}  /* main */
```

Read the Vector

```
void Read_vector (char*  prompt , float  local_v[], int n_bar, int p, int    my_rank)
{
   int i, q;
   float temp[MAX_LOCAL_ORDER];
   MPI_Status status;

   if (my_rank == 0) {
      printf("Enter %s\n", prompt);
      for (i = 0; i < n_bar; i++)
         scanf("%f", &local_v[i]);
      for (q = 1; q < p; q++) {
         for (i = 0; i < n_bar; i++)
            scanf("%f", &temp[i]);
         MPI_Send(temp, n_bar, MPI_FLOAT, q, 0, MPI_COMM_WORLD);
      }
   } else {
      MPI_Recv(local_v, n_bar, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
         &status);
   }
}  /* Read_vector */
```

Inner Product

```
float Serial_dot(float  x[], float  y[], int n)
{
   int    i;
   float  sum = 0.0;

   for (i = 0; i < n; i++)
      sum = sum + x[i]*y[i];
   return sum;
} /* Serial_dot */

float Parallel_dot(float  local_x[],  float  local_y[], int  n_bar )
{
   float  local_dot;
   float  dot = 0.0;

   local_dot = Serial_dot(local_x, local_y, n_bar);
   MPI_Reduce(&local_dot, &dot, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
   return dot;
} /* Parallel_dot */
```

Another Parallel Inner Product

- Same as the previous one, but the reduction is done on all processors.
- MPI_Allreduce
  - Similar to MPI_Reduce, but all processes get the result.

All-reduction Application

```
float Parallel_dot(float  local_x[],  float  local_y[],  int  n_bar)
{
   float  local_dot;
   float  dot = 0.0;

   local_dot = Serial_dot(local_x, local_y, n_bar);
   MPI_Allreduce(&local_dot, &dot, 1, MPI_FLOAT, MPI_SUM,
      MPI_COMM_WORLD);

   return dot;
} /* Parallel_dot */
```

Scatter the Data

- MPI_Scatter
  - This routine distributes data to all processors.
  - The root process sends the data via a send buffer.
  - All the others provide a receive buffer for the incoming data from the root.

Gather the Data

- MPI_Gather
  - This routine works the opposite way to MPI_Scatter. It collects data from all processes to the root process.
- MPI_Allgather
  - Similar to MPI_Gather but all the processes receive the result.

Matrix-Vector Multiplication

- Use MPI_Scatter to distribute the rows of the matrix and the segments of the input vector to all processes.
- Use MPI_Allgather to collect the entire vector in order to perform multiplication.
- Finally the process 0 uses MPI_Gather to collect all the fragments from other processes and prints the result.

Variables and Types

```
#define MAX_ORDER 100

typedef float LOCAL_MATRIX_T[MAX_ORDER][MAX_ORDER];

main(int argc, char* argv[]) {
   int my_rank;
   int  p;
   LOCAL_MATRIX_T  local_A;
   float  global_x[MAX_ORDER];
   float ocal_x[MAX_ORDER];
   float local_y[MAX_ORDER];
   int m, n;
   int local_m, local_n;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &p);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank == 0) {
      printf("Enter the order of the matrix (m x n)\n");
      scanf("%d %d", &m, &n);
   }
```

Matrix-Vector Multiplication

```
MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

local_m = m/p;
local_n = n/p;

Read_matrix("Enter the matrix", local_A, local_m, n, my_rank, p);
Print_matrix("We read", local_A, local_m, n, my_rank, p);

Read_vector("Enter the vector", local_x, local_n, my_rank, p);
Print_vector("We read", local_x, local_n, my_rank, p);

Parallel_matrix_vector_prod(local_A, m, n, local_x, global_x,
   local_y, local_m, local_n);
Print_vector("The product is", local_y, local_m, my_rank, p);

MPI_Finalize();

}  /* main */
```

Read the Matrix

```
void Read_matrix(char* prompt, LOCAL_MATRIX_T  local_A, int  local_m,
    int n, int my_rank, int p)
{
    int i, j;
    LOCAL_MATRIX_T  temp;
    for (i = 0; i < p*local_m; i++)
        for (j = n; j < MAX_ORDER; j++)
            temp[i][j] = 0.0;

    if (my_rank == 0) {
        printf("%s\n", prompt);
        for (i = 0; i < p*local_m; i++)
            for (j = 0; j < n; j++)
                scanf("%f",&temp[i][j]);
    }
    MPI_Scatter(temp, local_m*MAX_ORDER, MPI_FLOAT, local_A, local_m*MAX_ORDER,
        MPI_FLOAT, 0, MPI_COMM_WORLD);
}  /* Read_matrix */
```

Read the Vector

```
void Read_vector(char*  prompt, float  local_x[], int local_n, int my_rank, int p)
{
   int   i;
   float temp[MAX_ORDER];
   if (my_rank == 0) {
      printf("%s\n", prompt);
      for (i = 0; i < p*local_n; i++)
         scanf("%f", &temp[i]);
   }
   MPI_Scatter(temp, local_n, MPI_FLOAT, local_x, local_n, MPI_FLOAT, 0,
      MPI_COMM_WORLD);
}  /* Read_vector */
```

Product Computation

```
void Parallel_matrix_vector_prod(LOCAL_MATRIX_T local_A, int m, int n, float local_x[],
    float global_x[], float local_y[], int local_m, int local_n)
{
    /* local_m = m/p, local_n = n/p */

    int i, j;

    MPI_Allgather(local_x, local_n, MPI_FLOAT, global_x, local_n, MPI_FLOAT,
        MPI_COMM_WORLD);
    for (i = 0; i < local_m; i++) {
        local_y[i] = 0.0;
        for (j = 0; j < n; j++)
            local_y[i] = local_y[i] + local_A[i][j]*global_x[j];
    }
}  /* Parallel_matrix_vector_prod */
```

Print the Matrix

```
void Print_matrix(char* title, LOCAL_MATRIX_T  local_A, int local_m, int n , int my_rank, int p)
{
   int   i, j;
   float temp[MAX_ORDER][MAX_ORDER];

   MPI_Gather(local_A, local_m*MAX_ORDER, MPI_FLOAT, temp,
      local_m*MAX_ORDER, MPI_FLOAT, 0, MPI_COMM_WORLD);

   if (my_rank == 0) {
      printf("%s\n", title);
      for (i = 0; i < p*local_m; i++) {
         for (j = 0; j < n; j++)
            printf("%4.1f ", temp[i][j]);
         printf("\n");
      }
   }
}  /* Print_matrix */
```

Print the Vector

```
void Print_vector(char*  title, float  local_y[], int local_m, int my_rank, int p)
{
   int   i;
   float temp[MAX_ORDER];
   MPI_Gather(local_y, local_m, MPI_FLOAT, temp, local_m, MPI_FLOAT, 0,
      MPI_COMM_WORLD);
   if (my_rank == 0) {
      printf("%s\n", title);
      for (i = 0; i < p*local_m; i++)
         printf("%4.1f ", temp[i]);
      printf("\n");
   }
}  /* Print_vector */
```