DTS202TC Fundamentals of Parallel Computing

Week 2 - 2 Advanced MPI Programming

Hongbin Liu

2023

Xi'an Jiaotong-Liverpool University
西交利物浦大学

Recall from last lecture

- Hardware architecture:
  - SISD, SIMD, MIMD
- Pointer to pointer communication
  - MPI_Send and MPI_Receive
- Collective communication
  - MPI_Reduce, MPI_Allreduce
- Debugging MPI parallel application

Review MPI_Send MPI_Recv

```c
#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];
   int        comm_sz;  /* Number of processes */
   int        my_rank;  /* My process rank     */

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

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

   MPI_Finalize();
   return 0;
}  /* main */
```

All MPI use must be between init and finalise

What if MPI_Recv is called before MPI_Send?

Collective vs. Point-to-Point Communications

- All the processes in the communicator must call the same collective function. For example, a program that attempts to match a call to MPI Reduce on one process with a call to MPI Recv on another process is erroneous, and, in all likelihood, the program will hang or crash.
- Point-to-point communications are matched on the basis of tags and communicators. Collective communications don’t use tags, so they’re matched solely on the basis of the communicator and the order in which they’re called.

Time	Process 0	Process 1	Process 2
0	`a = 1; c = 2`	`a = 1; c = 2`	`a = 1; c = 2`
1	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`
2	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`

Today’s Goals

- Review last lecture
- More collective MPI
  - MPI_Scatter
  - MPI_Gather
- MPI matrix-vector multiplication
- MPI Performance measurement
- Live Demo

Broadcast

- Data belonging to a single process is sent to all of the processes in the communicator

MPI_BCAST(buffer, count, datatype, root, comm)

INOUT	buffer	starting address of buffer (choice)
IN	count	number of entries in buffer (non-negative integer)
IN	datatype	datatype of buffer (handle)
IN	root	rank of broadcast root (integer)
IN	comm	communicator (handle)

C binding

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root,
              MPI_Comm comm)
```

A Version of Get_input that Uses MPI_Bcast

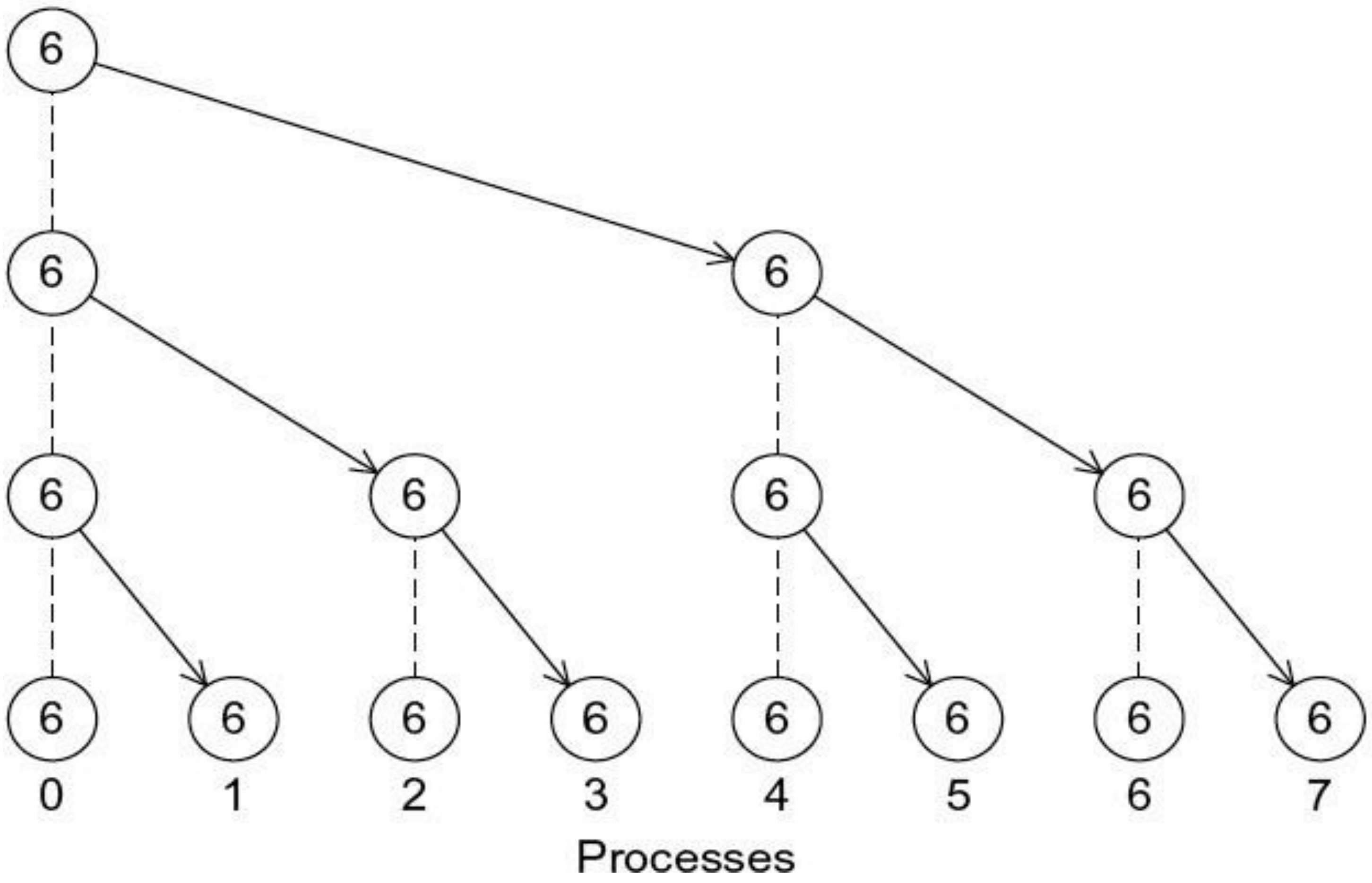
```c
void Get_input(int my_rank, int comm_sz, double* a_p, double* b_p,
      int* n_p) {

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
   }
   MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
}  /* Get_input */
```

```c
if (my_rank == 0) {
   printf("Enter a, b, and n\n");
   scanf("%lf %lf %d", a_p, b_p, n_p);
   for (dest = 1; dest < comm_sz; dest++) {
      MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
      MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
      MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
   }
} else { /* my_rank != 0 */
   MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
         MPI_STATUS_IGNORE);
   MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
         MPI_STATUS_IGNORE);
   MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
         MPI_STATUS_IGNORE);
}
```

What is the problem of this version?

A Tree-structured Broadcast

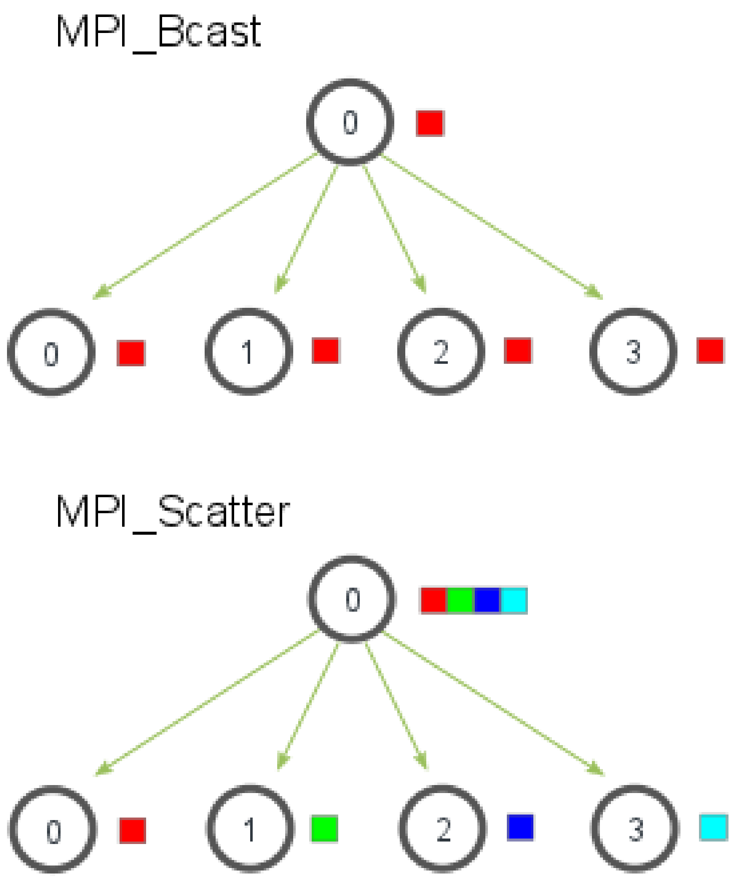


MPI_Scatter

- Similar to MPI_Bcast, however, instead of sending the same piece of data to all processes, MPI_Scatter sends chunks of an array to different processes.

MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	sendbuf	address of send buffer (choice, significant only at root)
IN	sendcount	number of elements sent to each process (non-negative integer, significant only at root)
IN	sendtype	datatype of send buffer elements (handle, significant only at root)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements in receive buffer (non-negative integer)
IN	recvtype	datatype of receive buffer elements (handle)
IN	root	rank of sending process (integer)
IN	comm	communicator (handle)

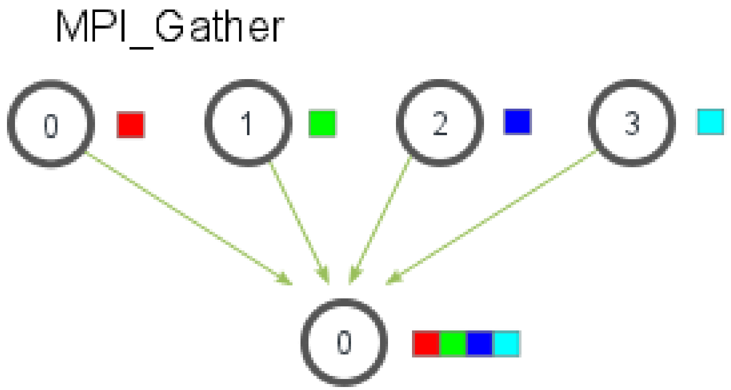


MPI_Scatter example in mpi_vector_add.c

```c
if (my_rank == 0) {
   a = malloc( size: n*sizeof(double));
   if (a == NULL) local_ok = 0;
   Check_for_error(local_ok, fname, "Can't allocate temporary vector",
         comm);
   printf("Enter the vector %s\n", vec_name);
   for (i = 0; i < n; i++)
      scanf("%lf", &a[i]);
   MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0,
      comm);
   free(a);
```

MPI_Gather

- MPI_Gather is the inverse of MPI_Scatter



MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative integer)
IN	sendtype	datatype of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice, significant only at root)
IN	recvcount	number of elements for any single receive (non-negative integer, significant only at root)
IN	recvtype	datatype of recv buffer elements (handle, significant only at root)
IN	root	rank of receiving process (integer)
IN	comm	communicator (handle)

MPI_Gather Example in mpi_vector_add.c

```c
if (my_rank == 0) {
    b = malloc( size: n*sizeof(double));
    if (b == NULL) local_ok = 0;
    Check_for_error(local_ok, fname, "Can't allocate temporary vector",
          comm);
    MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
          0, comm);
    printf("%s\n", title);
    for (i = 0; i < n; i++)
       printf("%f ", b[i]);
    printf("\n");
    free(b);
} else {
    Check_for_error(local_ok, fname, "Can't allocate temporary vector",
          comm);
    MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE, 0,
       comm);
}
```

MPI_Allgather

- Concatenates the contents of each process' send_buf_p and stores this in each process' recv_buf_p.

```
int MPI_Allgather(
      void*          send_buf_p   /* in  */,
      int            send_count   /* in  */,
      MPI_Datatype   send_type    /* in  */,
      void*          recv_buf_p   /* out */,
      int            recv_count   /* in  */,
      MPI_Datatype   recv_type    /* in  */,
      MPI_Comm       comm         /* in  */);
```

Matrix-vector multiplication

$$
\begin{bmatrix}
a_{00} & a_{01} & \cdots & a_{0,n-1} \\
a_{10} & a_{11} & \cdots & a_{1,n-1} \\
\vdots & \vdots & & \vdots \\
a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
\vdots & \vdots & & \vdots \\
a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1}
\end{bmatrix}
\begin{bmatrix}
x_0 \\ x_1 \\ \vdots \\ x_{n-1}
\end{bmatrix}
=
\begin{bmatrix}
y_0 \\ y_1 \\ \vdots \\ y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\ \vdots \\ y_{m-1}
\end{bmatrix}
$$

Multiply a matrix by a vector – pseudo-code

```
/* For each row of A */
for (i = 0; i < m; i++) {
   /* Form dot product of ith row with x */
   y[i] = 0.0;
   for (j = 0; j < n; j++)
      y[i] += A[i][j]*x[j];
}
```

Serial matrix-vector multiplication

```
void Mat_vect_mult(
        double  A[]  /* in  */,
        double  x[]  /* in  */,
        double  y[]  /* out */,
        int     m    /* in  */,
        int     n    /* in  */) {
    int i, j;

    for (i = 0; i < m; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++)
            y[i] += A[i*n+j]*x[j];
    }
}  /* Mat_vect_mult */
```

An MPI matrix-vector multiplication function (1)

```
void Mat_vect_mult(
      double    local_A[]  /* in  */,
      double    local_x[]  /* in  */,
      double    local_y[]  /* out */,
      int       local_m    /* in  */,
      int       n          /* in  */,
      int       local_n    /* in  */,
      MPI_Comm  comm       /* in  */) {
   double* x;
   int local_i, j;
   int local_ok = 1;
```

An MPI matrix-vector multiplication function (2)

```
   x = malloc(n*sizeof(double));
   MPI_Allgather(local_x, local_n, MPI_DOUBLE,
         x, local_n, MPI_DOUBLE, comm);

   for (local_i = 0; local_i < local_m; local_i++) {
      local_y[local_i] = 0.0;
      for (j = 0; j < n; j++)
         local_y[local_i] += local_A[local_i*n+j]*x[j];
   }
   free(x);
}  /* Mat_vect_mult */
```

Timing Serial Program

```
#include “time.h”
double start, finish;
...

GET_TIME(start);

/* Code to be timed */
...
GET_TIME(finish);
printf("Elapsed time = %e seconds \n", finish - start);
```

Elapsed MPI Time

```
double start, finish;
...
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();

/* Code to be timed */
...
finish = MPI_Wtime();
printf("Proc %d > Elapsed time = %e seconds \n", my_rank, finish - start);
```

MPI_Wtime does not include the idol time, e.g. time of waiting for a message to come.

Make sure only print out in Master process

Speedup and Efficiency

- Speedup: Ratio of execution time on one process to that on *p* processes

$$Speedup = \frac{t_1}{t_p}$$

- Efficiency: Speedup per process

$$Efficiency = \frac{t_1}{t_p \times p}$$

Speedups of a Parallel Application

comm_sz	Order of Matrix				
	1024	*2048*	*4096*	*8192*	*16,384*
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

Efficiencies of a Parallel Application

comm_sz	Order of Matrix				
	1024	*2048*	*4096*	*8192*	*16,384*
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

Scalability

- A program is scalable if the problem size can be increased at a rate so that the efficiency doesn’t decrease as the number of processes increase

Scalability (Cont.)

- Programs that can maintain a constant efficiency without increasing the problem size are sometimes said to be strongly scalable
- Programs that can maintain a constant efficiency if the problem size increase at the same rate as the number of processes are sometimes said to be weakly scalable

Safety in MPI Programs

- The MPI standard allows MPI_Send to behave in two different ways:
  - it can simply copy the message into an MPI managed buffer and return,
  - or it can block until the matching call to MPI_Recv starts.

Safety in MPI Programs

- Many implementations of MPI set a threshold at which the system switches from buffering to blocking.
- Relatively small messages will be buffered by MPI_Send.
- Larger messages, will cause it to block.

Safety in MPI Programs

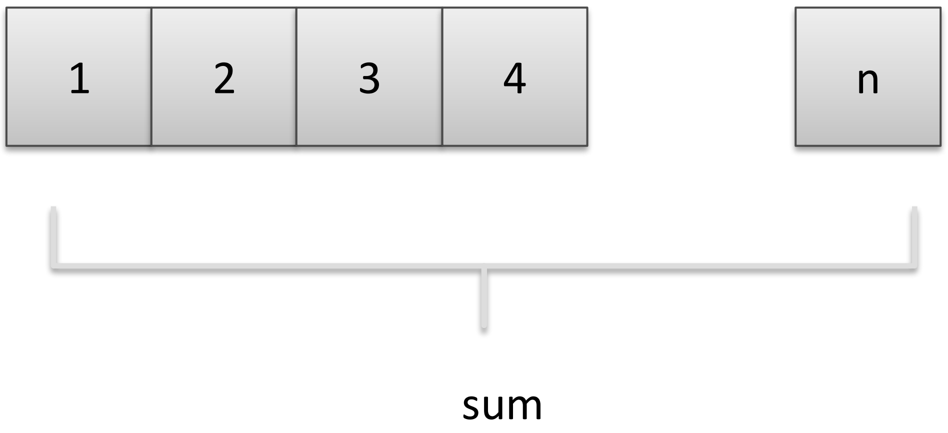
- If the MPI_Send executed by each process blocks, no process will be able to start executing a call to MPI_Recv, and the program will hang or deadlock.
- Each process is blocked waiting for an event that will never happen.

MPI_Sendrecv

- An alternative to scheduling the communications ourselves.
- Carries out a blocking send and a receive in a single call.
- The dest and the source can be the same or different.
- Especially useful because MPI schedules the communications so that the program won’t hang or crash.

Live coding demo

- Summing up an array



Wrap up