Recap of MPI programming

MPI (message passing interface)

MPI is a library standard for programming distributed memory

- MPI implementation(s) available on almost every major parallel platform (also on shared-memory machines)
- Portability, good performance & functionality
- Collaborative computing by a group of individual processes
- Each process has its own local memory
- Explicit message passing enables information exchange and collaboration between processes

The message-passing model

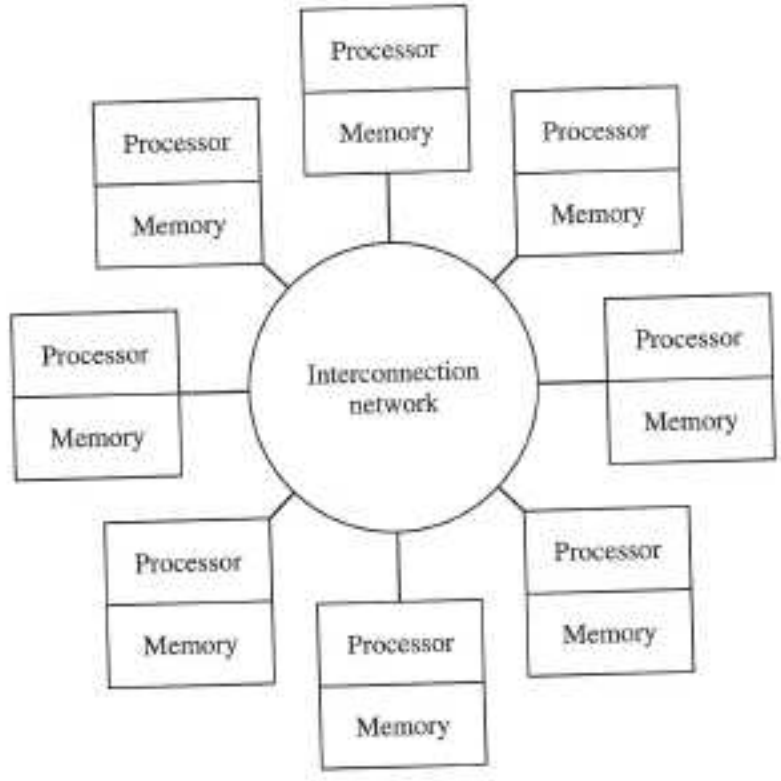


Figure 4.1 The message-passing model assumes that the underlying hardware is a collection of processors, each with its own local memory, and an interconnection network supporting message-passing between processors.

MPI language bindings

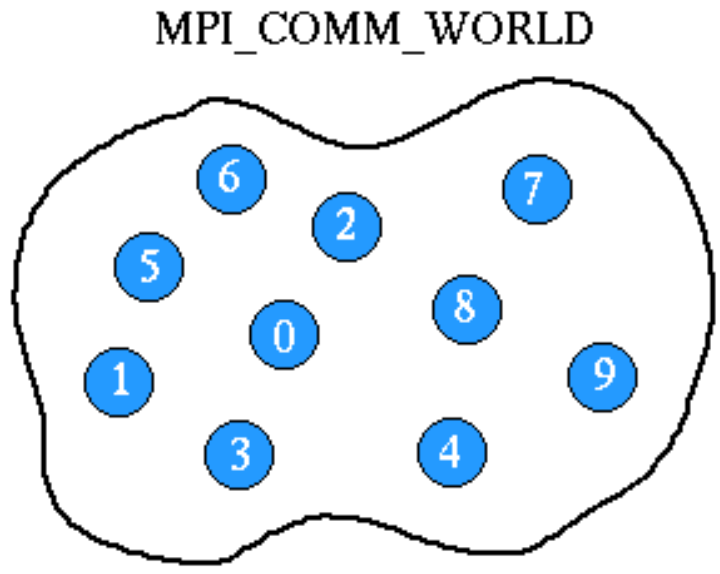
C binding

```
#include <mpi.h>

rc = MPI_Xxxxx(parameter, ... )
```

Fortran binding

```
include 'mpif.h'

CALL MPI_XXXXX(parameter,..., ierr)
```

MPI communicator



- An MPI communicator: a "communication universe" for a group of processes
- `MPI_COMM_WORLD` – name of the default MPI communicator, i.e., the collection of all processes
- Each process in a communicator is identified by its rank
- Almost every MPI command needs to provide a communicator as input argument

MPI process rank

- Each process has a unique rank, i.e. an integer identifier, within a communicator
- The rank value is between `0` and `#procs-1`
- The rank value is used to distinguish one process from another
- Commands `MPI_Comm_size` & `MPI_Comm_rank` are very useful
- Example

```
int size, my_rank;
MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

if (my_rank==0) {
   ...
}
```

The 6 most important MPI commands

- `MPI_Init` - initiate an MPI computation
- `MPI_Finalize` - terminate the MPI computation and clean up
- `MPI_Comm_size` - how many processes participate in a given MPI communicator?
- `MPI_Comm_rank` - which one am I? (A number between `0` and `size-1`.)
- `MPI_Send` - send a message to a particular process within an MPI communicator
- `MPI_Recv` - receive a message from a particular process within an MPI communicator

MPI "Hello-world" example

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

int main (int nargs, char** args)
{
  int size, my_rank;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  printf("Hello world, I've rank %d out of %d procs.\n",
         my_rank,size);
  MPI_Finalize ();
  return 0;
}
```

MPI "Hello-world" example (cont'd)

- Compilation example: `mpicc hello.c`
- Parallel execution example: `mpirun -np 4 ./a.out`
- Order of output from the processes is not determined, may vary from execution to execution

```
Hello world, I've rank 2 out of 4 procs.
Hello world, I've rank 1 out of 4 procs.
Hello world, I've rank 3 out of 4 procs.
Hello world, I've rank 0 out of 4 procs.
```

The mental picture of parallel execution

The same MPI program is executed concurrently on each process

Process 0 | Process 1 | · · · | Process P-1

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

int main (int nargs, char** args)
{
  int size, my_rank;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  printf("Hello world, I've rank %d out of %d procs.\n",
         my_rank,size);
  MPI_Finalize ();
  return 0;
}
```

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

int main (int nargs, char** args)
{
  int size, my_rank;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  printf("Hello world, I've rank %d out of %d procs.\n",
         my_rank,size);
  MPI_Finalize ();
  return 0;
}
```

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

int main (int nargs, char** args)
{
  int size, my_rank;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  printf("Hello world, I've rank %d out of %d procs.\n",
         my_rank,size);
  MPI_Finalize ();
  return 0;
}
```

Synchronization

- Many parallel algorithms require that none process proceeds before all the processes have reached the same state at certain points of a program.
- Explicit synchronization

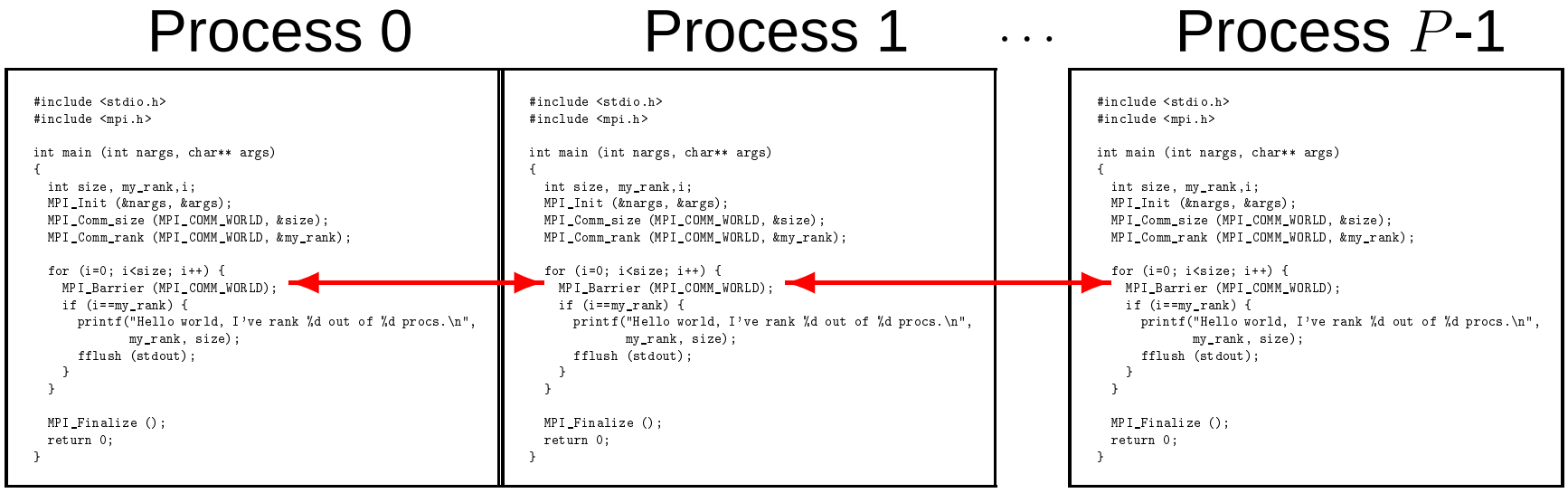
```
int MPI_Barrier (MPI_Comm comm)
```

- Implicit synchronization can be achieved through pairs of `MPI_Send` and `MPI_Recv`.
- When do we need synchronization?
  - Ask yourself the following question: "*If Process 1 progresses 100 times faster than Process 2, will the final result still be correct?*"

Example: ordered output

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

int main (int nargs, char** args)
{
  int size, my_rank,i;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  for (i=0; i<size; i++) {
    MPI_Barrier (MPI_COMM_WORLD);
    if (i==my_rank) {
      printf("Hello world, I've rank %d out of %d procs.\n",
             my_rank, size);
      fflush (stdout);
    }
  }

  MPI_Finalize ();
  return 0;
}
```

Example: ordered output (cont'd)



- The processes synchronize between themselves P times.
- Parallel execution result:

```
Hello world, I've rank 0 out of 4 procs.
Hello world, I've rank 1 out of 4 procs.
Hello world, I've rank 2 out of 4 procs.
Hello world, I've rank 3 out of 4 procs.
```

MPI point-to-point communication

- Involving two MPI processes (sender and receiver)
- Several different types of send and receive commands
  - Blocking/non-blocking send
  - Blocking/non-blocking receive
  - Four modes of send operations
  - Combined send/receive

The simplest MPI send command

```
int MPI_Send(void *buf, int count,
             MPI_Datatype datatype,
             int dest, int tag,
             MPI_Comm comm);
```

This blocking send function returns when the data has been delivered to the system and the buffer can be reused. The message may not have been received by the destination process.

The simplest MPI receive command

```
int MPI_Recv(void *buf, int count
             MPI_Datatype datatype,
             int source, int tag,
             MPI_Comm comm,
             MPI_Status *status);
```

- This blocking receive function waits until a matching message is received from the system so that the buffer contains the incoming message.
- Match of data type, source process (or `MPI_ANY_SOURCE`), message tag (or `MPI_ANY_TAG`).
- Receiving fewer `datatype` elements than `count` is ok, but receiving more is an error.

MPI message

An MPI message is an array of data elements "inside an envelope"

- *Data*: start address of the message buffer, counter of elements in the buffer, data type
- *Envelope*: source/destination process, message tag, communicator

MPI_Status

The source or tag of a received message may not be known if wildcard values were used in the receive function. In C, `MPI_Status` is a structure that contains further information. It can be queried as follows:

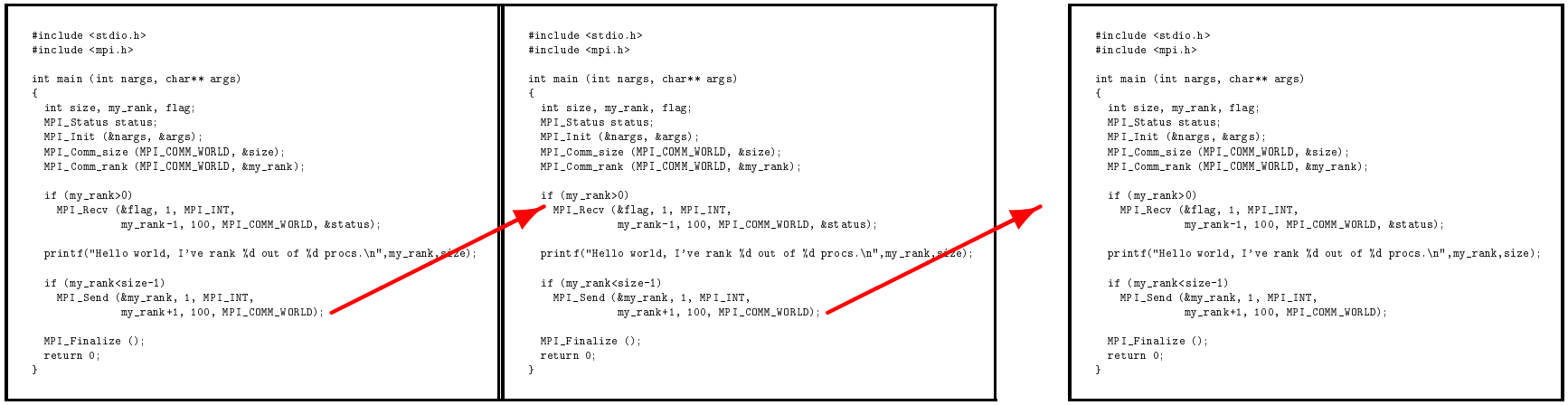
```
status.MPI_SOURCE

status.MPI_TAG

MPI_Get_count (MPI_Status *status,
               MPI_Datatype datatype,
               int *count);
```

Example of MPI_Send and MPI_Recv

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

int main (int nargs, char** args)
{
  int size, my_rank, flag;
  MPI_Status status;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  if (my_rank>0)
    MPI_Recv (&flag, 1, MPI_INT,
              my_rank-1, 100, MPI_COMM_WORLD, &status);

  printf("Hello world, I've rank %d out of %d procs.\n",my_rank,size);

  if (my_rank<size-1)
    MPI_Send (&my_rank, 1, MPI_INT,
              my_rank+1, 100, MPI_COMM_WORLD);

  MPI_Finalize ();
  return 0;
}
```

Example of `MPI_Send`/`MPI_Recv` (cont´d)

Process 0 Process 1 ··· Process P-1

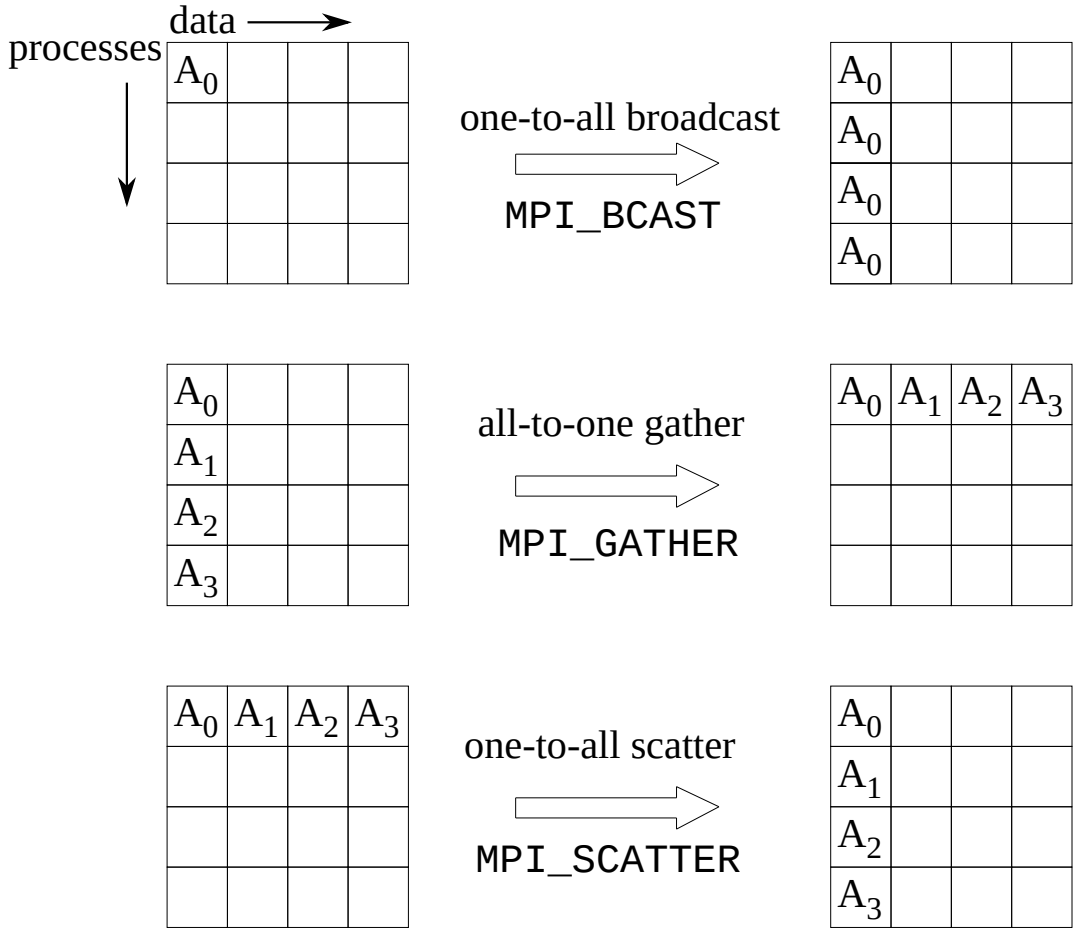


```
#include <stdio.h>
#include <mpi.h>

int main (int nargs, char** args)
{
  int size, my_rank, flag;
  MPI_Status status;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  if (my_rank>0)
    MPI_Recv (&flag, 1, MPI_INT,
              my_rank-1, 100, MPI_COMM_WORLD, &status);

  printf("Hello world, I've rank %d out of %d procs.\n",my_rank,size);

  if (my_rank<size-1)
    MPI_Send (&my_rank, 1, MPI_INT,
              my_rank+1, 100, MPI_COMM_WORLD);

  MPI_Finalize ();
  return 0;
}
```

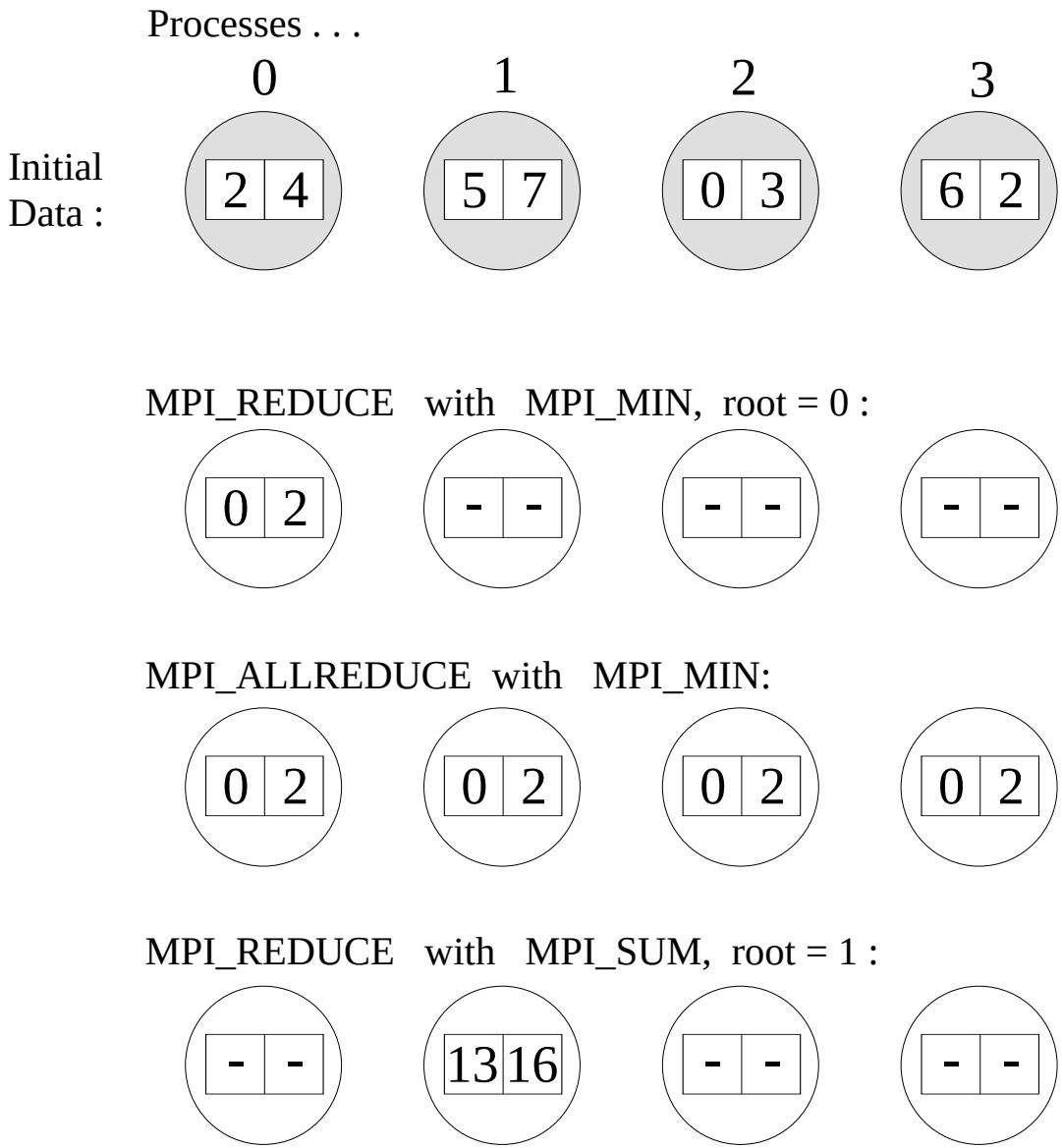
- Enforcement of ordered output by passing around a "semaphore", using `MPI_Send` and `MPI_Recv`
- Successful message passover requires a matching pair of `MPI_Send` and `MPI_Recv`

MPI collective communication

A collective operation involves *all* the processes in a communicator: (1) synchronization (2) data movement (3) collective computation



Collective communication (cont´d)



Computing inner-product in parallel

- Let us write an MPI program that calculates inner-product between two vectors $\vec{u} = (u_1, u_2, \ldots, u_M)$ and $\vec{v} = (v_1, v_2, \ldots, v_M)$,

$$c = \sum_{i=1}^{M} u_i v_i = u_1 v_1 + u_2 v_2 + \ldots + u_M v_M$$

- Partition $\vec{u}$ and $\vec{v}$ into P segments each of sub-length

$$m = \frac{M}{P}$$

- Each process first concurrently computes its local result
- Then the global result can be computed

Making use of collective communication

```
MPI_Comm_size (MPI_COMM_WORLD, &num_procs);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

my_start = M*my_rank/num_procs;
my_stop = M*(my_rank+1)/num_procs;

my_c = 0.;
for (i=my_start; i<my_stop; i++)
  my_c = my_c + (u[i] * v[i]);

MPI_Allreduce (&my_c, &c, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

Calculation of π

Want to numerically approximate the value of π

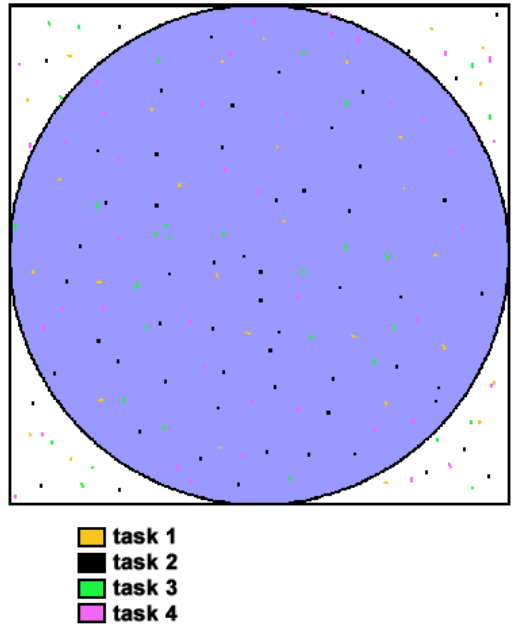
- Area of a circle: $A = \pi R^2$
- Area of the largest circle that fits into the unit square: $\frac{\pi}{4}$, because $R = \frac{1}{2}$
- Estimate of the area of the circle $\Rightarrow$ estimate of π
- How?
  - Throw a number of random points into the unit square
  - Count the percentage of points that lie in the circle by

$$\left((x - \frac{1}{2})^2 + (y - \frac{1}{2})^2 \right) \leq \frac{1}{4}$$

  - The percentage is an estimate of the area of the circle
- $\pi \approx 4\,A$

Parallel calculation of π

```
num = npoints/P;
my_circle_pts = 0;

for (j=1; j<=num; j++) {
  /* pseudo-code: generate random 0<=x,y<=1 */
  if (((x-0.5)*(x-0.5)+(y-0.5)*(y-0.5))<0.25)
    my_circle_pts += 1
}

MPI_Allreduce(&my_circle_pts,&total_count,1,MPI_INT,MPI_SUM,MPI_COMM_WORLD

pi = 4.0*total_count/npoints;
```



The issue of load balancing

What if `npoints` is not divisible by `P`?

- Simple solution of load balancing

```
num = npoints/P;
if (my_rank < (npoints%P))
  num += 1;
```

- Load balancing is very important for performance
- Homogeneous processes should have as even distribution of work load as possible

Probing in MPI

- It is possible in MPI to only read the envelope of a message before choosing whether or not to read the actual message.

```
int MPI_Probe(int source, int tag, MPI_Comm comm,
              MPI_Status *status)
```

- The `MPI_Probe` function blocks until a message matching the given `source` and/or `tag` is available
- The result of probing is returned in an `MPI_Status` data structure

Example: sum of random numbers

```
int main (int nargs, char** args)
{
  int size, my_rank, i, a, sum;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  srand (7654321*(my_rank+1));
  a = rand()%100;

  if (my_rank==0) {
    MPI_Status status;
    sum = a;
    for (i=1; i<size; i++) {
      MPI_Probe (MPI_ANY_SOURCE,500,MPI_COMM_WORLD,&status);
      MPI_Recv (&a, 1, MPI_INT,
                status.MPI_SOURCE,500,MPI_COMM_WORLD,&status);
      sum += a;
    }

    printf("<%02d> sum=%d\n",my_rank,sum);
  }
  else
    MPI_Send (&a, 1, MPI_INT, 0, 500, MPI_COMM_WORLD);

  MPI_Finalize ();
  return 0;
}
```

Example of deadlock

- When a large message is sent from one process to another
  - and if there is insufficient OS storage at the destination, the send command must wait for the user to provide the memory space (through a receive command)
  - the following code is unsafe because progress depends on the availability of system buffers

Process 0	Process 1
`Send(1)`	`Send(0)`
`Recv(1)`	`Recv(0)`

Solutions to deadlocks

- Order the send/receive calls more carefully
- Use `MPI_Sendrecv`
- Use `MPI_Bsend`
- Use non-blocking operations

Overlap communication with computation

- Performance may be improved on many systems by overlapping communication with computation. This is especially true on systems where communication can be executed autonomously by an intelligent communication controller.
- Use of non-blocking and completion routines allow computation and communication to be overlapped. (Not guaranteed, though.)

Non-blocking send

```
int MPI_Isend(void* buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

- The command returns “immediately”
- The message buffer should not be rewritten when the command returns
- Must check for local completion

Non-blocking receive

```
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm, MPI_Request *request)
```

- The command returns “immediately”
- The message buffer should not be read yet
- Must check for local completion
- The use of nonblocking receives may also avoid system buffering and memory-to-memory copying, as information is provided early on the location of the receive buffer.

MPI_Request

- A request object identifies various properties of a communication operation
- A request object also stores information about the status of the pending communication operation

Local completion

- Two ways of checking on non-blocking sends and receives
  - `MPI_Wait` blocks until the communication is complete

    ```
    MPI_Wait(MPI_Request *request, MPI_Status *status)
    ```

  - `MPI_Test` returns “immediately”, and sets flag to true is the communication is complete

    ```
    MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
    ```

Example of non-blocking send/receive

```
int main(int argc, char *argv[])
{
int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
MPI_Request reqs[4];
MPI_Status stats[4];

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

prev = rank-1; next = rank+1;
if (rank == 0)  prev = numtasks - 1;
if (rank == (numtasks - 1))  next = 0;

MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

      {  /* do some work */  }

MPI_Waitall(4, reqs, stats);

MPI_Finalize();
}
```

More about point-to-point communication

- When a standard mode blocking send call returns, the message data and envelope have been “safely stored away”. The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer.
- MPI decides whether outgoing messages will be buffered. If MPI buffers outgoing messages, the send call may complete before a matching receive is invoked. On the other hand, buffer space may be unavailable, or MPI may choose not to buffer outgoing messages, for performance reasons. Then the send call will not complete until a matching receive has been posted, and the data has been moved to the receiver.

Four modes of MPI's send

- standard mode – a send may be initiated even if a matching receive has not been initiated
- buffered mode – similar to standard mode, but completion is always independent of matching receive, and message may be buffered to ensure this
- synchronous mode – a send will not complete until message delivery is guaranteed
- ready mode – a send may be initiated only if a matching receive has been initiated

Send-receive operations

- MPI send-receive operations combine in one call the sending of a message to one destination and the receiving of another message, from another process
- A send-receive operation is very useful for executing a shift operation across a chain of processes

```
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 int dest, int sendtag,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 int source, int recvtag,
                 MPI_Comm comm, MPI_Status *status)
```

Null process

- The special value `MPI_PROC_NULL` can be used to specify a “dummy” source or destination for communication
- This may simplify the code that is needed for dealing with boundaries, e.g., a non-circular shift done with calls to send-receive
- A communication with process `MPI_PROC_NULL` has no effect

Example:

```
int left_rank = my_rank-1, right_rank = my_rank+1;
if (my_rank==0)
  left_rank = MPI_PROC_NULL;
if (my_rank==size-1)
  right_rank = MPI_PROC_NULL;
```

MPI timer

```
double MPI_Wtime(void)
```

This function returns a number representing the number of wall-clock seconds elapsed since some time in the past.

Example usage:

```
double starttime, endtime;
starttime = MPI_Wtime();
/* ....  work to be timed  ... */
endtime   = MPI_Wtime();
printf("That took %f seconds\n",endtime-starttime);
```