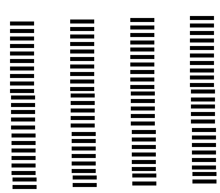
MPI and – Message passing interface
(Chapter 3)

Introduction to MPI

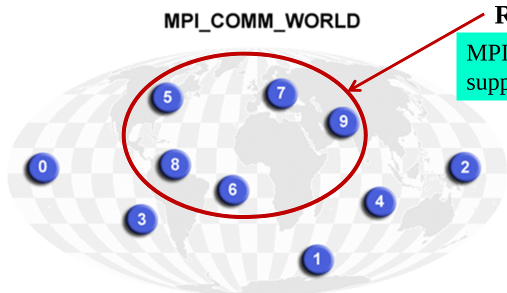
(https://computing.llnl.gov/tutorials/mpi/)

- All large scale multiprocessors have "physically" distributed memory systems.
- A lot of overhead when building a shared address space on top of a physically distributed memory system.
- Some problems can naturally be partitioned into parallel sub-problems (with possible coordination and synchronization)
- MPI (Message Passing Interface) evolved as the standard interface for message passing libraries.
- Note: Sockets is Unix' way of passing messages and many MPI libraries are built using sockets. MPI, however, is much easier to use than sockets.
- An MPI implementation allows a user to start multiple threads (SPMD programming style) and provide functions for the threads to communicate and synchronize.



SPMD Programs

- The user specifies the number of processes and number of processors.
- The same source code is executed by all processes
- One or more process can execute on each processor
- The set of processes is defined as the “MPI_COMM_WORLD”
- Can have different processes do different things by using the process id (rank)
  - MPI_Comm_rank(MPI_COMM_WORLD, &rank)
- Subsets of MPI_COMM_WORLD, called communicators, can be defined by the user.



Rami’s_world

MPI_Comm_rank(Rami’s_world, &rank) supplies the rank within Rami’s_world

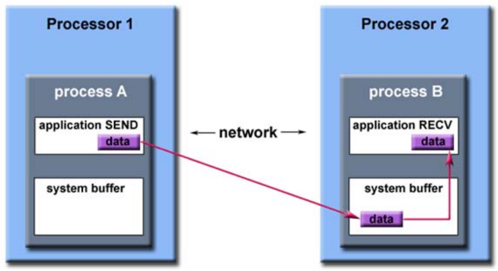
A simple MPI Program

```
#include <mpi.h>
int main(int argc, char *argv[]) {
  int numtasks, my_rank, rc;
  rc = MPI_Init(&argc,&argv);
  if (rc != MPI_SUCCESS) {
     printf ("Error starting MPI program \n");
     MPI_Abort(MPI_COMM_WORLD, rc);
     }
  MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
  MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
  if (my_rank == 0) { /* master */
    printf (“#of tasks= %d, My rank= %d\n",numtasks,rank);
  } else { /* worker */
    printf (“My rank= %d\n", rank);
  }
  MPI_Finalize();
}
```

MPI_Init: Has to be called first, and once

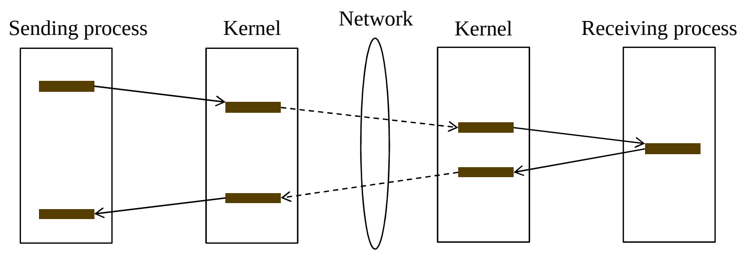
MPI_Finalize: Has to be called last, and once

Point-to-Point Communication



Can be explicitly allocated (in buffered send/receive)

Path of a message buffered at the receiving process



Path of a message across address spaces

Blocking Point-to-Point Communication

Address of data (usually variable name)

For pairing send with receive

```
MPI_Send(x, #_of_items, item_type, dest_rank, tag ,communicator);

MPI_Recv(x, #_of_items, item_type, source_rank, tag, communicator,
         &status);
```

A structure of type MPI_Status

Predefined: MPI_CHAR, MPI_INT, MPI_FLOAT, …

MPI_Send
Source_rank

MPI_Recv
Dest_rank

Blocking: Return after the sender application buffer is free for reuse, or the application buffer received the message, respectively.

Out of order receiving

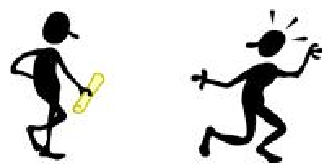
```
MPI_Recv(x, MAX_items, item_type, MPI_ANY_SOURCE, MPI_ANY_TAG,
         communicator, &status);
```

Larger or equal to expected size

Get actual values using

MPI_Status*
{ MPI_SOURCE
MPI_TAG
MPI_ERROR }

Allows message reception from any source



```
MPI_Get_count(MPI_Status status /*in*/ , MPI_Datatype type /*in*/ ,
              int*  count /*out*/);
```


Non-blocking Point-to-Point Communication

```
MPI_Isend(x, #_of_items, item_type, dest_rank, tag ,communicator,
          &request);
```

A request number returned by MPI. Of type MPI_Request

```
MPI_Irecv(x, #_of_items, item_type, source_rank, tag, communicator,
         &request);

MPI_Wait(&request, &status);
```

Blocks until the operation corresponding to "request" is completed

```
MPI_Waitall(count, array of requests, array of statuses);

MPI_Test(&request, &flag, &status);
MPI_Testall();
MPI_Testsome();
MPI_Testany();
```

non-blocking

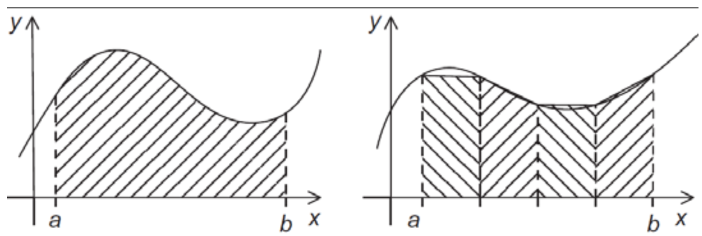
Returns "true" (1) if operation had completed and "false (0), otherwise

Types of send/receive

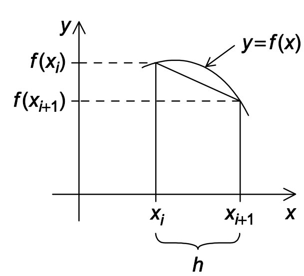
- **Blocking:** MPI_Send() and MPI_Recv()
  - o Return after the sender application buffer is free for reuse, or the application buffer received the message, respectively.
- **Synchronous blocking:** MPI_Ssend()
  - o Returns after the destination process received the message
- **Non-blocking:** MPI_Isend() and MPI_Irecv()
  - o Returns immediately. MPI_wait and MPI_Test indicate that the non-blocking send or receive has completed locally
- **Synchronous non-blocking:** MPI_Issend()
  - o Returns immediately. MPI_wait and MPI_Test indicate that the destination process has received the message
- **Buffered**: allows the programmer to explicitly control system buffers.

There are other send/receive routines with different blocking properties

Example – The trapezoidal rule for integration




```
/* Input:   a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
   x_i = a + i*h;
   approx += f(x_i);
}
approx = h*approx;
```



```c
int main(void) {
   int my_rank, comm_sz, n = 1024, local_n;      // n, a and b are the input to the program
   double a = 0.0, b = 3.0, h, local_a, local_b;
   double local_int, total_int;
   int source;

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

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

   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
   local_int = Trap(local_a, local_b, local_n, h); // apply trapezoidal rule from local_a  to local_b

   if (my_rank != 0) {
      MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,
            MPI_COMM_WORLD);
   } else {
      total_int = local_int;
      for (source = 1; source < comm_sz; source++) {
         MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         total_int += local_int;
      }
   }

   if (my_rank == 0) {
      printf("With n = %d trapezoids, our estimate\n", n);
      printf("of the integral from %f to %f = %.15e\n",
          a, b, total_int);
   }
   MPI_Finalize();
   return 0;
} /*  main  */
```

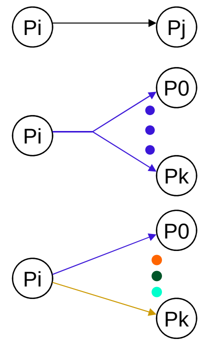

Dealing with input

Most MPI implementations only allow process 0 in MPI_COMM_WORLD access to stdin. Hence, it must read the data and send to the other processes.

```c
void Get_input(
      int      my_rank  /* in  */,
      int      comm_sz  /* in  */,
      double*  a_p      /* out */,
      double*  b_p      /* out */,
      int*     n_p      /* out */) {
   int dest;

   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);
   }
}  /* Get_input */
```

Bad practice to depend on in-order message delivery. Should use tags

Type of messages



- **Point-to-point**: one processor sends a message to another processor
- **One-to-all**: one processor broadcasts a message to all other processors
- **One-to-all personalized**: one processor sends a different message to each other processor
- **All-to-all**: each processor broadcasts a message to all other processors
- **All-to-all personalized**: each processor sends a different message to each other processors

Collective communication

- Can be built using point-to-point communications, but typical MPI implementations have optimized them
- All processes place the same call, although depending on the process, some arguments may not be used

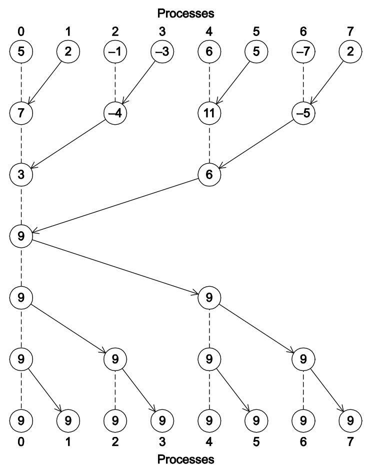
```
MPI_Bcast(x, n_items, type, root, MPI_COMM_WORLD)

MPI_Barrier(MPI_COMM_WORLD)

MPI_Reduce(x,r,n_items,type,op,root,MPI_COMM_WORLD)
```

- x: Private data to be reduced
- r: Location of reduced data
- op: Operator used in reduction: MPI_MAX, MPI_SUM, MPI_PROD, …

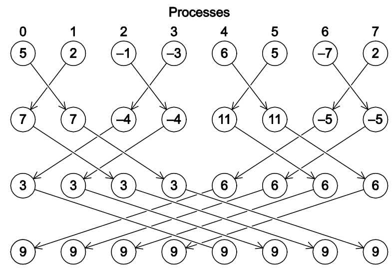
```
MPI_Allreduce(x,r,n_items,type,op,MPI_COMM_WORLD)
```

Same as MPI_Reduce() except that every thread gets the result, not only "root" (equivalent to MPI_Reduce followed by MPI_Bcast)

Efficiency of MPI_Allreduce



A global sum followed by distribution of the result.



A butterfly-structured (hypercube)global sum.

Order of collective Communication

- Collective communications do not use tags – they are matched purely on the basis of the order in which they are called
- The names of the memory locations are irrelevant to the matching
- Example: Assume three processes with calling MPI_Reduce with operator MPI_SUM, and destination process 0.

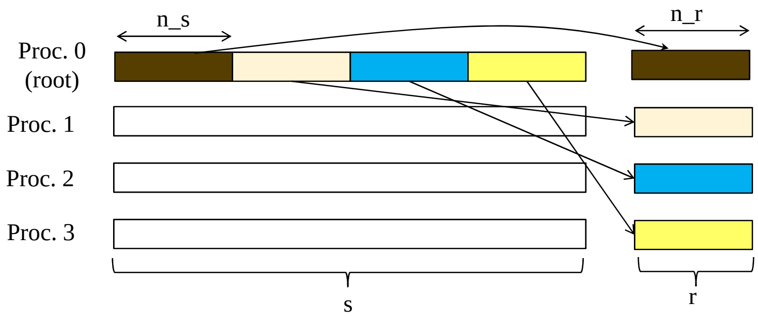
Time	Process 0	Process 1	Process 2
0	`a = 1; c = 2`	`a = 1; c = 2`	`a = 1; c = 2`
1	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`
2	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`

- The order of the calls will determine the matching so, in process 0, the value stored in b will be 1+2+1 = 4, and the value stored in d will be 2+1+2 = 5.

Scatter (personalized broadcast – one to many)

```
MPI_Scatter(s, n_s, s_type, r, n_r, r_type, root, MPI_COMM_WORLD)
```

- s: Data to be scattered, needed only at root
- n_s: # of Items sent to each thread
- r: Location of scattered data
- n_r: # of Items received by each thread



Scatter Example

```
int main(int argc,char **argv) {
    int *a;
    double *recvbuffer;
    ...
    MPI_Comm_size(MPI_COMM_WORLD,&n);
    if (my_rank == 0) { /* master */
        <allocate array a of size N>
        <allocate array recvbuffer of size N/n>
        MPI_Scatter(a, N/n, MPI_INT, recvbuffer, N/n, MPI_INT,
                    0, MPI_COMM_WORLD);
  } else { /* worker */
        <allocate array recvbuffer of size N/n>
        MPI_Scatter(NULL, 0, MPI_INT,  recvbuffer, N/n, MPI_INT,
                    0, MPI_COMM_WORLD);
  }
    ...
}
```

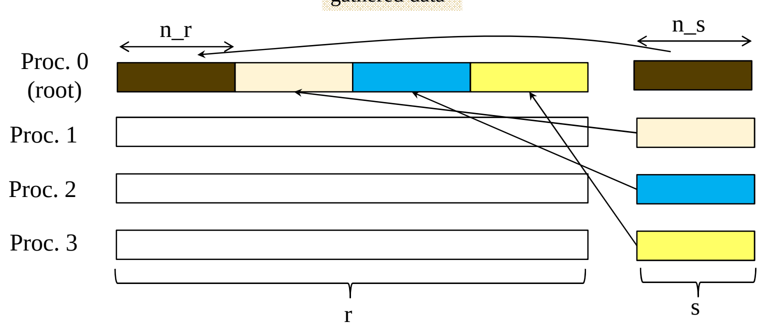
Can use MPI_IN_PLACE

Gather (many to one)

```
MPI_Gather(s, n_s, s_type, r, n_r, r_type, root, MPI_COMM_WORLD)
```

Data to be gathered

Location of gathered data

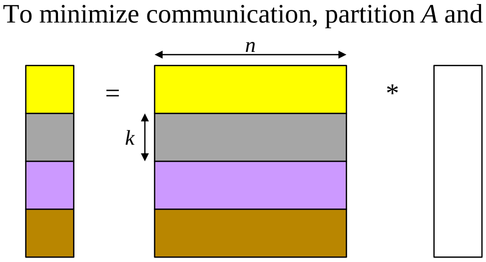



```
MPI_Allgather(s, n_s, s_type, r, n_r, r_type, MPI_COMM_WORLD)
```

No “root”

Examples of global-local data mapping

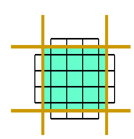
- Consider *nxn* matrix/vector multiplication $y = A * x$ on P processors.
- To minimize communication, partition A and y row wise.



- Each processor, *pid*, will allocate two $k = n/P$ vectors for its shares of x and y and an *kxn* matrix (call it local_A[]) for its share of A.

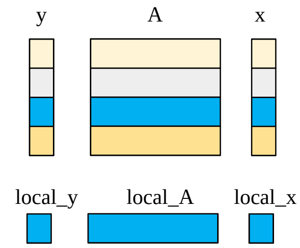
local_A[*i*,*j*] = A[*k***pid* + *i* , *j*]
local_y[*i*] = A[*k***pid* + *i*]

- In SOR (Laplace iterative solver), we may simplify programming by augmenting the local domains by a stripe to accommodate boundary data received from other processors.



Example: Matrix-vector multiplication

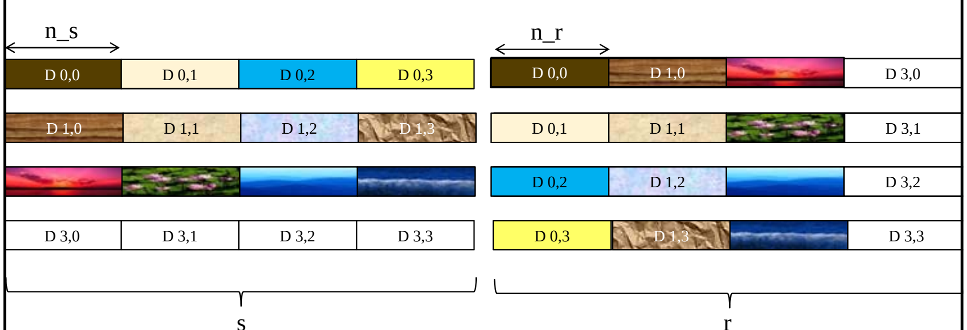
```
void Mat_vect_mult(
        double    local_A[]  /* in  */,
        double    local_x[]  /* in  */,
        double    local_y[]  /* out */,
        int       local_m    /* in  */,
        int       n          /* in  */,
        int       local_n    /* in  */,
        MPI_Comm  comm       /* in  */)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



21

All to all personalized

```
MPI_Alltoall(s, n_s, s_type, r, n_r, r_type, MPI_COMM_WORLD)
```



Example: Matrix transpose

22

Derived data types

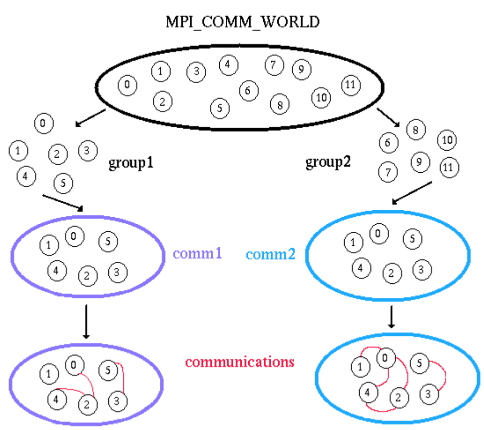
- Used to represent any collection of data items in memory by storing both the types of the items and their relative locations in memory.
- This allows the use of these data types in the send and receive calls.
- Formally, consists of a sequence of basic MPI data types together with a displacement for each of the data types.
- Trapezoidal Rule example:

Variable	Address
a	24
b	40
n	48

$$\{(\texttt{MPI\_DOUBLE}, 0), (\texttt{MPI\_DOUBLE}, 16), (\texttt{MPI\_INT}, 24)\}$$

What more can you do?

- Build virtual topologies
- Define new communicators from MPI_COMM_WORLD



- Extract handle of old group
  - MPI_Comm_group ()
- Form new group as a subset of old group
  - MPI_Group_incl ()
- Create new communicator for new group
  - MPI_Comm_create ()
- Determine new rank in new communicator
  - MPI_Comm_rank ()
- Communicate in new group
- Free up new communicator and group
  - MPI_Comm_free ()
  - MPI_Group_free ()

Overlapping communication and computation

- Example in SOR can
  - Isend
  - Ireceive
  - Do computation that do not depend on received message
  - Wait for receive to complete
  - Complete the computation.