MPI

Course “Parallel Computing”

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Wolfgang.Schreiner@risc.jku.at
http://www.risc.jku.at
Message Passing Interface (MPI)

- An API for portable distributed memory programming.
  - Set of library routines, no compiler support needed.
- Official bindings for C and Fortran.
  - Inofficial bindings exist for various other languages.
- Various implementations.
  - MPICH: initial implementation by Argonne National Lab.
  - MVAPICH: MPICH derivative by Ohio State university.
  - Open MPI: merger of various previous implementations.
  - Commercial implementations by HP, Intel, Microsoft.
  - Hardware support: SGI MPT with MPI offload engine.
- Maintained by the MPI Forum.

MPI Execution Model

SPMD: Single Program, Multiple Data.
Compiling and Executing MPI

Paths (Default):

CPATH=...:/opt/sgi/mpt/mpt-2.04/include
LIBRARY_PATH=...:/opt/sgi/mpt/mpt-2.04/lib
LD_LIBRARY_PATH=...:/opt/sgi/mpt/mpt-2.04/lib

Source:

#include <mpi.h>

Intel Compiler:

module load intelcompiler
icc -Wall -O3 -lmpi matmult.c -o matmult

GCC:

module load GnuCC
gcc -Wall -O3 -lmpi matmult.c -o matmult

Execution:

export MPI_DSM_CPULIST=512-1023:2
mpirun -np 16 ./matmult
#include <mpi.h>

int main(int argc, char *argv[]) {
    char message[20];
    int size, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("processor %d among %d processors\n", rank, size);
    if (rank == 0) { /* code for process zero */
        strncpy(message,"Hello, there", 19);
        MPI_Send(message, 20, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
    }
    else if (rank == 1) { /* code for process one */
        MPI_Status status;
        MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
        printf("received :\%s:\n", message);
    }
    MPI_Finalize();
    return 0;
}
Basic Operations

int MPI_Init(int *argc, char ***argv)

int MPI_Comm_size(MPI_Comm comm, int *size)
IN  comm  communicator (handle)
OUT size  number of processes in the group of comm (integer)

int MPI_Comm_rank(MPI_Comm comm, int *rank)
IN  comm  communicator (handle)
OUT rank  rank of the calling process in group of comm (integer)

int MPI_Finalize(void)

int MPI_Abort(MPI_Comm comm, int errorcode)
IN  comm  communicator of tasks to abort
IN  errorcode  error code to return to invoking environment

Starting a computation, determining its scope, terminating it.
int MPI_Send(const void* buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm)

IN buf initial address of send buffer (choice)
IN count number of elements in send buffer (non-negative integer)
IN datatype datatype of each send buffer element (handle)
IN dest rank of destination (integer)
IN tag message tag (integer)
IN comm communicator (handle)

Returns when message buffer may be used again; may (but need not) block, if no matching receive statement was issued.
Blocking Receive

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

**OUT** `buf` initial address of receive buffer (choice)

**IN** `count` number of elements in receive buffer (non-negative integer)

**IN** `datatype` datatype of each receive buffer element (handle)

**IN** `source` rank of source or MPI_ANY_SOURCE (integer)

**IN** `tag` message tag or MPI_ANY_TAG (integer)

**IN** `comm` communicator (handle)

**OUT** `status` status object (Status)

Blocks until a matching message could be received; if more than one message matches, the first one sent is received.
# Example: Computing Pi by Throwing Darts

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

double dboard (int darts);
#define DARTS 50000 /* number of throws */
#define ROUNDS 100 /* number of iterations */
#define MASTER 0 /* task ID of master task */

int main (int argc, char *argv[]) {
    double homepi, pi, avepi, pirecv, pisum;
    int taskid, numtasks, source, mtype, i, n;
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
    printf("MPI task %d has started...
", taskid);
    srandom (taskid);
    avepi = 0;
    for (i = 0; i < ROUNDS; i++) {
        homepi = dboard(DARTS);
        if (taskid != MASTER) {
            mtype = i;
            MPI_Send(&homepi, 1, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);
        } else {
            /* Master receives messages from all workers */
            /* - Message type set to the iteration count */
            /* - Message source set to wildcard DONTCARE: */
            mtype = i;
            pisum = 0;
            for (n = 1; n < numtasks; n++) {
                /* keep running total of pi */
                MPI_Recv(&pirecv, 1, MPI_DOUBLE, MPI_ANY_SOURCE, mtype, MPI_COMM_WORLD, &status);
                pisum = pisum + pirecv;
            }
            /* Average value of pi for this iteration */
            pi = (pisum + homepi)/numtasks;
            /* Average value of pi over all iterations */
            avepi = ((avepi * i) + pi)/(i + 1);
        }
        printf(" After %8d throws, pi = %10.8f
", (DARTS * (i + 1)), avepi);
    }
    if (taskid == MASTER)
        printf("Real value of PI: 3.1415926535897 
");
    MPI_Finalize();
    return 0;
}
```

```c
double dboard (int darts);
```
Example: Computing Pi by Throwing Darts

```c
#define sqr(x) ((x)*(x))

double dboard(int darts) {
    /* number of hits */
    int score = 0;
    /* throw darts at board */
    for (int n = 1; n <= darts; n++) {
        /* random coordinates in interval [-1,+1] */
        double r = (double)random()/RAND_MAX;
        double x_coord = (2.0 * r) - 1.0;
        r = (double)random()/RAND_MAX;
        double y_coord = (2.0 * r) - 1.0;

        /* if dart lands in circle, increment score */
        if ((sqr(x_coord) + sqr(y_coord)) <= 1.0) {
            score++;
        }
    }

    /* calculate pi = 4*(pi*r^2)/(4*r^2) */
    double pi = 4.0 * (double)score/(double)darts;
    return(pi);
}
```

From MPI tutorial at https://computing.llnl.gov/tutorials/mpi.
## Datatype Constants

<table>
<thead>
<tr>
<th>MPI_Datatype</th>
<th>Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>char</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
</tbody>
</table>

... Also for all other builtin types.
Derived Datatypes

```c
int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

**IN** count replication count (non-negative integer)
**IN** oldtype old datatype (handle)
**OUT** newtype new datatype (handle)

```c
int MPI_Type_create_struct(int count, const int array_of_blocklengths[],
                            const MPI_Aint array_of_displacements[],
                            const MPI_Datatype array_of_types[], MPI_Datatype *newtype)
```

**IN** count number of blocks (non-negative integer)
**IN** array_of_blocklength number of elements in each block
(array of non-negative integer)
**IN** array_of_displacements byte displacement of each block
(array of integer)
**IN** array_of_types type of elements in each block
(array of handles to datatype objects)
**OUT** newtype new datatype (handle)

```c
int MPI_Type_commit(MPI_Datatype *datatype)
```

**INOUT** datatype datatype that is committed (handle)

Arrays and records; also various other derived types.
double array[1024];

MPI_Datatype array_type;
MPI_Type_contiguous( 1024, MPI_DOUBLE, &array_type );
MPI_Type_commit( &array_type );

MPI_Send(array, 1, array_type, ...);

struct R { int i ; double d; } record;

MPI_Datatype record_type;
int ab[] = { 1, 1 };
MPI_Aint ad[] = { offsetof(struct R, i), offsetof(struct R, d) };
MPI_Datatype at[] = { MPI_INT, MPI_DOUBLE };
MPI_Type_create_struct( 2, ab, ad, at, &record_type );
MPI_Type_commit( &record_type );

MPI_Send(&record, 1, record_type, ...);
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)

IN source rank of source or MPI_ANY_SOURCE (integer)
IN tag message tag or MPI_ANY_TAG (integer)
IN comm communicator (handle)
OUT status status object (Status)

Blocks until a matching message is available (without yet receiving it).
typedef struct _MPI_Status {
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;
    ...
} MPI_Status;

int MPI_Get_count(const MPI_Status *status,
    MPI_Datatype datatype, int *count)

IN status return status of receive operation (Status)
IN datatype datatype of each receive buffer entry (handle)
OUT count number of received entries (integer)

Query the sender, tag, and size of a message (to be) received.
Receiving with Incomplete Information

MPI_Comm comm = ... ; MPI_Status statue = ... ;

MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);

int source = status.MPI_SOURCE;
int tag = status.MPI_TAG;
MPI_Get_count(status, MPI_INT, &count);

int* buf = malloc(count*sizeof(int));
MPI_Recv(buf, count, MPI_INT, source, tag, comm, &status);

Determine the sender, tag, and size of message received.
Controlling Message Buffers

int MPI_Buffer_attach(void* buffer, int size)
IN buffer initial buffer address (choice)
IN size buffer size, in bytes (non-negative integer)

int MPI_Buffer_detach(void* buffer_addr, int* size)
OUT buffer initial buffer address (choice)
OUT size buffer size, in bytes (non-negative integer)

Control buffer to be used for outgoing messages; send operation (typically) only blocks, if buffer is full.
Collective Communication

More compact and more efficient programs by the use of collective communication operations.
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, 
    int root, MPI_Comm comm)

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

Sender (root) and receivers perform the same operation.
Gather

```c
int MPI_Gather(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
                void* recvbuf, int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```

**IN** sendbuf  
starting address of send buffer (choice)

**IN** sendcount  
number of elements in send buffer (non-negative integer)

**IN** sendtype  
data type 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  
data type of recv buffer elements
(significant only at root) (handle)

**IN** root  
rank of receiving process (integer)

**IN** comm  
communicator (handle)

Receiver (root) and senders perform the same operation; also the root is one of the senders.
**Gather-to-all**

```c
int MPI_Allgather(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
                  void* recvbuf, int recvcount, MPI_Datatype recvtype,
                  MPI_Comm comm)
```

- **IN** `sendbuf` starting address of send buffer (choice)
- **IN** `sendcount` number of elements in send buffer (non-negative integer)
- **IN** `sendtype` data type of send buffer elements (handle)
- **OUT** `recvbuf` address of receive buffer (choice)
- **IN** `recvcount` number of elements received from any process (non-negative integer)
- **IN** `recvtype` data type of receive buffer elements (handle)
- **IN** `comm` communicator (handle)

Every process serves both as a sender and a receiver.
Scatter

```c
int MPI_Scatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
                 void* recvbuf, int recvcount, MPI_Datatype recvtype,
                 int root, MPI_Comm 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  
data type of send buffer elements  
  (significant only at root) (handle)

**OUT** recvbuf  
address of receive buffer (choice)

**IN** recvcount  
number of elements in receive buffer (non-negative integer)

**IN** recvtype  
data type of receive buffer elements (handle)

**IN** root  
rank of sending process (integer)

**IN** comm  
communicator (handle)

**Sender (root) and receivers perform the same operation; also the root is one of the receivers.**
Example: Matrix Multiplication

```c
int main(int argc, char* argv[]) {
    MPI_Comm comm = MPI_COMM_WORLD;
    int size, rank;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(comm, &size);
    MPI_Comm_rank(comm, &rank);

    int n;
    if (rank == 0) {
        if (argc != 2) MPI_Abort(comm, -1);
        n = atoi(argv[1]);
        if (n == 0) MPI_Abort(comm, -1);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, comm);

    // row number n of A is extended to size*n0
    int n0 = n%size == 0 ? n/size : 1+n/size;
    double* A;
    if (rank == 0) {
        A = malloc(size*n0*n*sizeof(double));
        for (int i=0; i<n; i++)
            for (int j=0; j<n; j++)
                A[i*n+j] = rand()%10;
    }
    double* A0 = malloc(n0*n*sizeof(double));
    MPI_Scatter(A, n0*n, MPI_DOUBLE,
                A0, n0*n, MPI_DOUBLE, 0, comm);

    double* B = malloc(n*n*sizeof(double));
    if (rank == 0) {
        for (int i=0; i<n; i++)
            for (int j=0; j<n; j++)
                B[i*n+j] = rand()%10;
    }
    MPI_Bcast(B, n*n, MPI_DOUBLE, 0, comm);

    double* C0 = malloc(n0*n*sizeof(double));
    for (int i=0; i<n0; i++) {
        for (int j=0; j<n; j++) {
            C0[i*n+j] = 0;
            for (int k=0; k<n; k++)
                C0[i*n+j] += A0[i*n+k]*B[k*n+j];
        }
    }
    double* C;
    if (rank == 0) {
        C = malloc(size*n0*n*sizeof(double));
        MPI_Gather(C0, n0*n, MPI_DOUBLE,
                    C, n0*n, MPI_DOUBLE, 0, comm);
        if (rank == 0) { print(C, n, n); }
    }
    MPI_Finalize();
}
```
Reduction Operations

More compact and more efficient programs by the use of reduction operations.
int MPI_Reduce(const void* sendbuf,
    void* recvbuf, int count, MPI_Datatype datatype,
    MPI_Op op, int root, MPI_Comm comm)

IN  sendbuf  address of send buffer (choice)
OUT recvbuf  address of receive buffer (choice, significant only at root)
IN  count    number of elements in send buffer (non-negative integer)
IN  datatype  data type of elements of send buffer (handle)
IN  op        reduce operation (handle)
IN  root      rank of root process (integer)
IN  comm      communicator (handle)

Receiver (root) and senders perform the same operation; also the root is one of the senders.
Predefined Reduction Operations

- MPI_MAX: maximum
- MPI_MIN: minimum
- MPI_SUM: sum
- MPI_PROD: product
- MPI_LAND: logical and
- MPI_BAND: bit-wise and
- MPI_LOR: logical or
- MPI_BOR: bit-wise or
- MPI_LXOR: logical exclusive or (xor)
- MPI_BXOR: bit-wise exclusive or (xor)
- MPI_MAXLOC: max value and location
- MPI_MINLOC: min value and location

Commutative and associative operations; thus the order of reduction does not matter.
User-Defined Reduction Operations

int MPI_Op_create(MPI_User_function* user_fn, int commute, MPI_Op* op)

IN user_fn user defined function (function)
IN commute true if commutative; false otherwise.
OUT op operation (handle)

Turn user-defined function to a reduction operation; must be associative but not necessarily commutative.
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

double dboard (int darts);
#define DARTS 50000 /* number of throws */
#define ROUNDS 100 /* number of iterations */
#define MASTER 0 /* task ID of master task */

int main (int argc, char *argv[])
{
    double homepi, pisum, pi, avepi;
    int taskid, numtasks, i;
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
    printf("MPI task %d has started...
", taskid);

    /* Set seed for random number generator */
    srand48(taskid);

    avepi = 0;
    for (i = 0; i < ROUNDS; i++) {
        /* All tasks calculate pi */
        homepi = dboard(DARTS);
        /* sum values of homepi across all tasks */
        MPI_Reduce(&homepi, &pisum, 1, MPI_DOUBLE,
                    MPI_SUM, MASTER, MPI_COMM_WORLD);
        if (taskid == MASTER) {
            pi = pisum/numtasks;
            avepi = ((avepi * i) + pi)/(i + 1);
            printf(" After %8d throws, pi = %10.8f
", 
                     (DARTS * (i + 1)), avepi);
        }
    }
    if (taskid == MASTER)
        printf("Real PI: 3.1415926535897 
");
    MPI_Finalize();
    return 0;
}
int MPI_Allreduce(const void* sendbuf,
                   void* recvbuf, int count, MPI_Datatype datatype,
                   MPI_Op op, MPI_Comm comm)

IN sendbuf  address of send buffer (choice)
OUT recvbuf address of receive buffer (choice)
IN count    number of elements in send buffer (non-negative integer)
IN datatype data type of elements of send buffer (handle)
IN op       reduce operation (handle)
IN comm     communicator (handle)

Every process serves both as a sender and a receiver.
int main(int argc, char *argv[]) {
    ...
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_WORLD, &np);
    MPI_Comm_rank(MPI_WORLD, &me);

    // read work size and work at process 0
    int size; float* work;
    if (me == 0) {
        size = read_work_size();
        work = malloc(size*sizeof(float));
        read_array(work, size);
    }

    // distribute work size to every process
    MPI_Bcast(&size, 1, MPI_INT, 0, MPI_WORLD);

    // allocate space for local work in every process
    if (size%np != 0) MPI_Abort(MPI_WORLD, -1);
    int lsize = size/np;
    float* lwork = malloc((lsize+2)*sizeof(float));

    // distribute work to all processes
    MPI_Scatter(work, lsize, MPI_FLOAT,
                lwork+1, lsize, MPI_FLOAT, 0, MPI_WORLD);

    // determine neighbors in ring
    int lnbr = (me+np-1)%np;
    int rnbr = (me+1)%np;

    // iterate until convergence
    float globalerr = 99999.0;
    while (globalerr > 0.1) {
        // exchange boundary values with neighbors
        MPI_Send(lwork+1, 1, MPI_FLOAT,
                 lnbr, 10, MPI_WORLD);
        MPI_Recv(lwork+lsize+1, 1, MPI_FLOAT,
                  rnbr, 10, MPI_WORLD, &status);
        MPI_Send(lwork+lsize, 1, MPI_FLOAT,
                 rnbr, 20, MPI_WORLD);
        MPI_Recv(lwork, 1, MPI_FLOAT,
                 lnbr, 20, MPI_WORLD, &status);

        // perform local work
        compute(lwork, lsize);

        // determine maximum error among all processes
        float localerr = error(lwork, lsize);
        MPI_Allreduce(&localerr, &globalerr,
                      1, MPI_FLOAT, MPI_MAX, MPI_WORLD);
    }

    // collect results at process 0
    MPI_Gather(local+1, lsize, MPI_FLOAT,
               work, lsize, MPI_Float, 0, MPI_WORLD);
    if (me == 0) { write_array(work, size); }
    MPI_Finalize();
}
Example: Finite Difference Problem

Handle boundaries by “ghost cells”.

Processes

0 1 2

(1) MPI_BCAST

(2) MPI_SCATTER

(3) MPI_SEND/RECV

(4) MPI_REDUCEALL

(5) MPI_GATHER
Non-Blocking Communication

```c
int MPI_Iprobe(int source, int tag, MPI_Comm comm,
               int *flag, MPI_Status *status)
```

**IN**  
source  rank of source or MPI_ANY_SOURCE (integer)  
tag  message tag or MPI_ANY_TAG (integer)  
comm  communicator (handle)  

**OUT**  
flag  (logical)  
status  status object (Status)

Sets flag to “true” if a matching message is pending to be received; otherwise sets flag to “false”. 
Example: Managing a Shared Data Structure

```c
int READ = 1; int WRITE = 2; int VALUE = 3;
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    int *data = ...;
    while (true) {
        // check for pending request and process it
        int flag, MPI_Status status;
        MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG,
                   MPI_COMM_WORLD, &flag, &status);
        if (flag) {
            flag = process(status, data);
            if (flag) continue;
        }

        // perform own work generating requests
        int server; int address; int value;
        ...
        MPI_Send(&address, 1, MPI_INT,
                 server, READ, MPI_WORLD);
        MPI_Receive(&value, 1, MPI_INT,
                    server, VALUE, MPI_WORLD);
        ...
        int[2] write; write[0] = address; write[1] = value;
        MPI_Send(write, 2, MPI_INT,
                  server, WRITE, MPI_WORLD);
    }
    MPI_Finalize();
}
```

```c
int process(MPI_Status status, int* data) {
    int client = status.MPI_SOURCE;
    int address; int[2] write;
    switch (status.MPI_TAG) {
    case READ:
        MPI_Recv(&address, 1, MPI_INT,
                  client, READ, MPI_WORLD, &status);
        MPI_Send(&data[address], 1, MPI_INT,
                 client, VALUE, MPI_WORLD);
        return true;
    case WRITE:
        MPI_Recv(write, 2, MPI_INT, client,
                  WRITE, MPI_WORLD, &status);
        data[write[0]] = write[1];
        return true;
    default:
        return false;
    }
}
```
Modularity of Program Design

Sequential composition:

- Message intended for a subsequent phase must not be received by a previous phase.

```c
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
IN comm    communicator (handle)
OUT newcomm copy of comm (handle)

MPI_Comm comm;
MPI_Comm_dup(MPI_COMM_WORLD, &comm);
library_fun(comm);
```

New context for specific phase of program.
Modularity of Program Design

- Parallel composition:

Message intended for one parallel computation must not interfere with those of another parallel computation.

```c
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
IN comm communicator (handle)
IN color control of subset assignment (integer)
IN key control of rank assignment (integer)
OUT newcomm new communicator (handle)

MPI_Comm_split(MPI_COMM_WORLD, myid%3, myid, &comm);
switch (myid%3) {
  case 0: fun0(comm); break;
  case 1: fun1(comm); break;
  case 2: fun2(comm); break;
}
```

Multiple new contexts for concurrent program phases.
Communicating Between Groups

```c
int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,
    MPI_Comm peer_comm, int remote_leader, int tag, MPI_Comm *newintercomm)

IN local_comm local intra-communicator (handle)
IN local_leader rank of local group leader in local_comm (integer)
IN peer_comm "peer" communicator;
    significant only at the local_leader (handle)
IN remote_leader rank of remote group leader in peer_comm;
    significant only at the local_leader (integer)
IN tag tag (integer)
OUT newintercomm new inter-communicator (handle)
```

```c
MPI_Comm_split(MPI_COMM_WORLD, myid%2, myid, &comm);
MPI_Comm_rank(comm, &newid);
if (myid%2 == 0) {
    MPI_Intercomm_create(comm, 0, MPI_COMM_WORLD,
        1, 99, &intercomm);
    MPI_Send(message, 1, MPI_INT, newid,
        0, intercomm, &status);
} else {
    MPI_Intercomm_create(comm, 0, MPI_COMM_WORLD,
        0, 99, &intercomm);
    MPI_Receive(message, 1, MPI_INT, newid,
        0, intercomm, &status);
}
```
Further MPI Features

- Additional collective operations.
  - All-to-all communication, gathering/scattering with varying data count.

- Specialized communication operations.
  - Forced synchronous, buffered, and nonblocking communication.

- One-sided communications.

- Virtual topologies.

- Dynamic processes.

- Environmental management.

- Parallel input/output.

Mainly added in MPI versions 2 and 3.