TECHNISCHE UNIVERSITÄT DARMSTADT

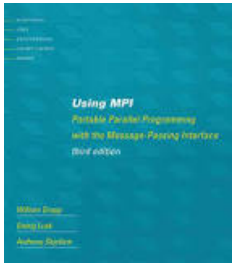
Large-Scale Parallel Computing

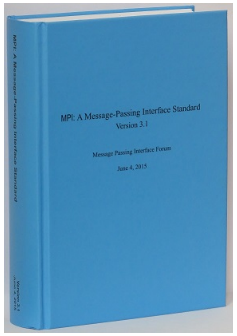
Prof. Dr. Felix Wolf

MESSAGE PASSING INTERFACE PART 1

Literature

- William Gropp, Ewing Lusk, Anthony Skjellum: Using MPI, 3rd edition, MIT Press, 2014
- Message Passing Interface Forum: MPI: A Message-Passing Interface Standard Version 3.1
  - http://www.mpi-forum.org

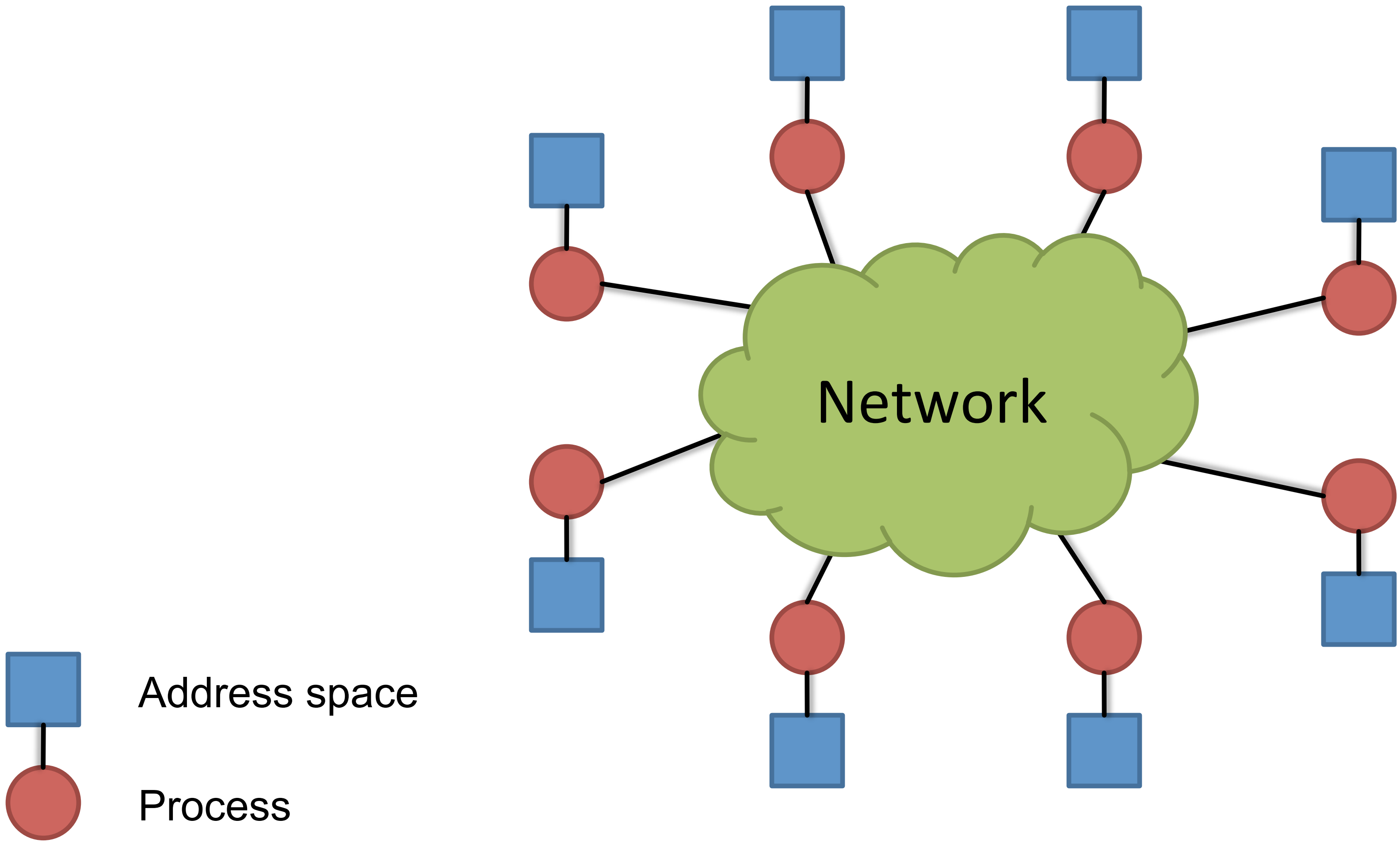




Outline

- Message-passing model
- Basic MPI concepts
- Essential MPI functions
- Simple MPI programs

Message passing



Message passing

- Suitable for distributed memory
- Multiple processes each having their own private address space
- Access to (remote) data of other processes via sending and receiving messages (explicit communication)

  - Sender invokes send routine
  - Receiver invokes receive routine

```
if (my_id == SENDER)
    send_to(RECEIVER, data);

if (my_id == RECEIVER)
    recv_from(SENDER, data)
```

- De-facto standard MPI: www.mpi-forum.org

Advantages

- Universality - Works with both distributed and shared memory
- Expressivity - Intuitive (anthropomorphic) and complete model to express parallelism
- Ease of debugging - Debugging message-passing programs easier than debugging shared-memory programs
  - Although writing debuggers for message-passing might be harder
- Performance & scalability - Better control of data locality. Distributed memory machines provide more memory and cache
  - Can enable super-linear speedups

Disadvantages

- Incremental parallelization hard - Parallelizing a sequential program using MPI often requires a complete redesign
- Message-passing primitives relatively low-level - Much programmer attention is diverted from the application problem to an efficient parallel implementation
- Communication and synchronization overhead - Communication and synchronizations costs can become bottleneck at large scales (especially group communication)
- MPI standard quite complex - The basics are simple, but using MPI effectively requires substantial knowledge

What is MPI?

- MPI stands for Message Passing Interface
- MPI specifies a library, not a language
  - Specifies names, calling sequences, and results of functions / subroutines needed to communicate via message passing
  - Language bindings for C/C++ and Fortran
  - MPI programs are linked with the MPI library and compiled with ordinary compilers

What is MPI? (2)

- MPI is a specification, not a particular implementation
  - De-facto standard for message passing
  - Defined by the MPI Forum – open group with representatives from academia and industry
  - www.mpi-forum.org
- Correct MPI program should be able to run on all MPI implementations without change
- Both proprietary and portable open-source implementations

History

- Version 1.0 (1994)
  - Fortran77 and C language bindings
  - 129 functions
- Version 1.1 (1995)
  - Corrections and clarifications, minor changes
  - No additional functions
- Version 1.2 (1997)
  - Further corrections and clarifications for MPI-1
  - 1 additional function

History (2)

- Version 2.0 (1997)
  - MPI-2 with new functionality (193 additional functions)
  - Parallel I/O
  - Remote memory access
  - Dynamic process management
  - Multithreaded MPI
  - C++ binding
- Version 2.1 (2008)
  - Corrections and clarifications
  - Unification of MPI 1.2 and 2.0 into a single document
- Version 2.2 (2009)
  - Further corrections and clarifications
  - New functionality with minor impact on implementations

History (3)

TECHNISCHE UNIVERSITÄT DARMSTADT

- Version 3.0 (2012)
  - New functionality with major impact on implementations
  - Non-blocking collectives
  - Neighborhood collectives
  - New one-sided communication operations
  - Fortran 2008 bindings
  - Tool information interface
- Version 3.1 (2015)
  - Mostly corrections and clarifications
  - Few new functions added

Minimal message interface

send(address, length, destination, tag)

Message buffer (length in bytes)

Used for matching

Actual message can be shorter than buffer

receive(address, length, source, tag, actlen)

Message buffer

(address, length) not really adequate

- Often message is non-contiguous
  - Example: row of matrix that is stored column-wise
  - In general, dispersed collection of structures of different sizes
  - Programmer wants to avoid “packing” messages
- Data types may have different sizes in heterogeneous systems
  - Length in bytes not an adequate specification of the semantic content of the message

Message buffer (2)

MPI solution

- (address, count, datatype)
- Example: (a, 300, MPI_REAL) describes array (vector) a of 300 real numbers
- Data type can also be non-contiguous

Separating families of messages

Matching of messages via tag argument

- Wildcard tags match any tag
- Entire program must use tags in a predefined and coherent way
- Problem – libraries should not by accident receive messages from the main program

MPI solution

- Message context
- Allocated at runtime by the system in response to the user and library requests
- No wild-card matching permitted

Naming processes

- Processes belong to groups
- Processes within a given group identified by ranks
  - Integers from 0 to n-1
- Initial group to which all processes belong
  - Ranks from 0 to 1 less than total number of processes

Communicators

- Combines the notions of context and group in a single object called communicator
- Becomes argument to most communication operations
- Destination or source specified in send or receive refers to rank of the process within group identified by communicator
- Two predefined communicators
  - MPI_COMM_WORLD contains all processes
  - MPI_COMM_SELF contains only the local process

Blocking send

TECHNISCHE UNIVERSITÄT DARMSTADT

```
MPI_Send(address, count, datatype, destination, tag, comm)
```

- Sends count occurrences of items of the form datatype starting at address
- Destination specified as rank in the group associated with communicator comm
- Argument tag is an integer used for message matching
- comm identifies a group of processes and a communication context

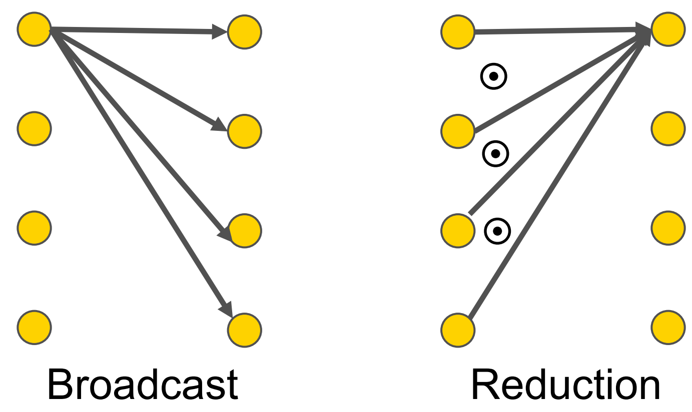
Blocking receive

```
MPI_Recv(address, maxcount, datatype, source,
         tag, comm, status)
```

- Allows maxcount occurrences of items of the form datatype to be received in the buffer starting at address
- Source is rank in comm or wildcard MPI_ANY_SOURCE
- tag is an integer used for message matching or wildcard MPI_ANY_TAG
- comm identifies a group of processes and a communication context
- status holds information about the actual message size, source, and tag

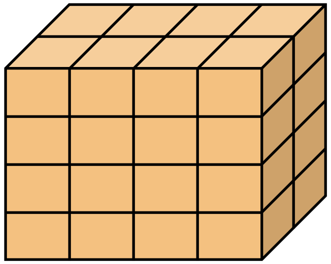
Collective communication & computation

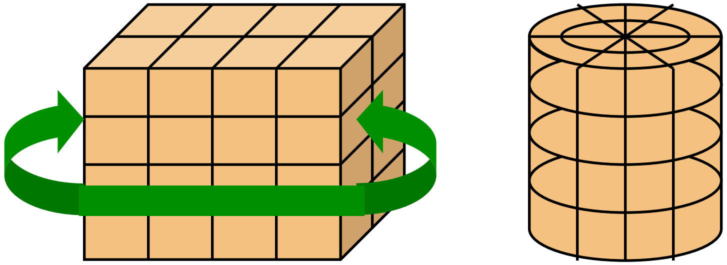
- Recurring parallel group communication patterns (1->n, n->1, n->n)
  - Communication (e.g., broadcast)
  - Computation (e.g., sum reduction)
- Manual implementation via point-to-point messages cumbersome
  - Often suboptimal performance
- MPI offers a range of predefined collective operations
  - Use optimized algorithms
  - May take advantage of hardware-specific features (e.g., network)



Virtual topologies

- Applications often define logical adjacency relationships among processes
  - Example: domain decomposition using Cartesian grid
  - Communication occurs between neighbors
- Virtual topologies allow efficient mapping of these adjacency relationships onto the physical topology of the underlying machine
  - Optimization of communication between neighbors
  - Convenient process naming, e.g. using Cartesian coordinates





Debugging and profiling

- Users can intercept MPI calls via “hooks”
  - Allows the definition of custom profiling and debugging mechanisms
- MPI implementations can expose variables that provide insight into internal performance information

Communication modes

- There are also non-blocking versions of send and receive whose completion can be tested and waited for
  - Allows overlap of computation and communication
- Multiple communication modes
  - Standard mode – common practice
  - Synchronous mode – requires send to block until receive is posted
  - Ready mode – way for the programmer to notify the system that receive has already been posted
  - Buffered mode – provides user-controllable buffering

A six-function version of MPI

`MPI_Init`	Initialize MPI
`MPI_Comm_size`	Find out how many processes there are
`MPI_Comm_rank`	Find out which process I am
`MPI_Send`	Send a message
`MPI_Recv`	Receive a message
`MPI_Finalize`	Terminate MPI

Header file

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

Contains

- Definition of named constants
- Function prototypes
- Type definitions

Opaque objects

- Internal representations of various MPI objects such as groups, communicators, datatypes, etc. are stored in MPI-managed system memory
  - Not directly accessible to the user, and objects stored there are opaque
- Their size and shape is not visible to the user
- Accessed via handles, which exist in user space
- MPI procedures that operate on opaque objects are passed handle arguments to access these objects

Generic MPI function format

```
error = MPI_Function(parameter, ...);
```

- Error code is integer return value
- Successful return code will be MPI_SUCCESS
- Failure return codes are implementation dependent

MPI namespace:

The MPI_ and PMPI_ prefixes are reserved for MPI constants and functions (i.e., application variables and functions must not begin with MPI_ or PMPI_).

Initialization and finalization

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

- Must be called as the first MPI function
  - Only exception: MPI_Initialized
- MPI specifies no command-line arguments but does allow an MPI implementation to make use of them

```
int MPI_Finalize()
```

- Must be called as the last MPI function
  - Only exception: MPI_Finalized

Rank in a communicator

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

- Determines the rank of the calling process in the communicator
- The rank uniquely identifies each process in a communicator

```
int myrank;
...
MPI_Init(&argc, &argv);
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

Size of a communicator

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

- Determines the size of the group associated with a communicator

```
int size;
...
MPI_Init(&argc, &argv);
...
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Hello world

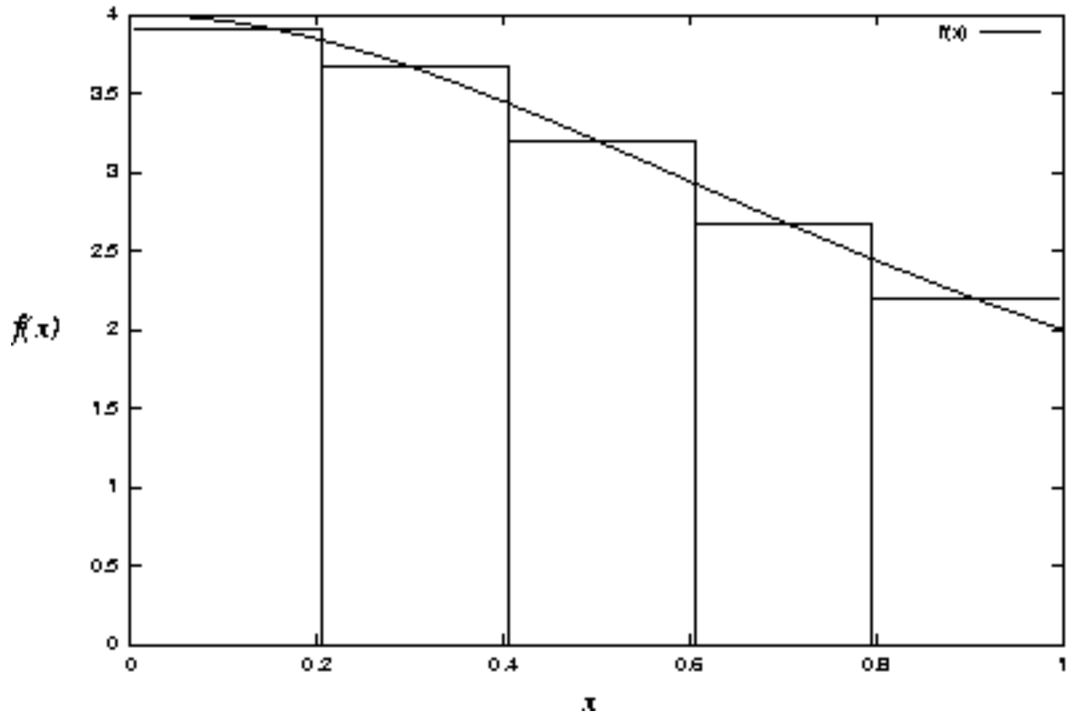
```c
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[] )
{
    int myid, numprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    printf("I am %d out of %d\n", myid, numprocs);
    MPI_Finalize();
    return 0;
}
```

```
> mpiexec -n 4 ./hello
I am 3 out of 4
I am 1 out of 4
I am 0 out of 4
I am 2 out of 4
```

Calculating π via numerical integration

$$\int_0^1 \frac{1}{1+x^2}\,dx = \arctan(x)\,|_0^1 = \arctan(1) - \arctan(0) = \arctan(1) = \frac{\pi}{4}$$

$$\Rightarrow \int_0^1 \frac{4}{1+x^2} = \pi$$



Calculating π via numerical integration (2)

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
int main( int argc, char *argv[] )
{
    int n, myid, numprocs, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    if (myid == 0) {
        printf("Enter the number of intervals:");
        scanf("%d",&n);
    }
    [... next slide ...]
    if (myid == 0)
        printf("pi is approximately %.16f, Error is %.16f\n",
                pi, fabs(pi - PI25DT));
    MPI_Finalize();
    return 0;
}
```

Calculating π via numerical integration (3)

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

h   = 1.0 / (double) n;
sum = 0.0;
for (i = myid + 1; i <= n; i += numprocs) {
    x = h * ((double)i - 0.5);
    sum += (4.0 / (1.0 + x*x));
}
mypi = h * sum;
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Broadcast

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

- Broadcasts a message from the process with rank root to all other processes of the group.
  - buf = starting address of buffer
  - count = number of entries in buffer
  - datatype = data type of buffer
  - root = rank of broadcast root
  - comm = communicator

Reduce

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op,
               int root, MPI_Comm comm)
```

- Combines the elements in the input buffer of each process using the operation op and returns the combined value in the output buffer of the process with rank root
  - sendbuf = address of send buffer
  - recvbuf = address of receive buffer
  - count = number of elements in send buffer
  - datatype = data type of elements of send buffer
  - op = reduce operation
  - root = rank of root process
  - comm = communicator

Basic datatypes in C

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MP_LONG_DOUBLE	long double
MPI_BYTE	(any C type)
MPI_PACKED	(any C type)
MPI_LONG_LONG_INT	longlong int (64 bit integer)

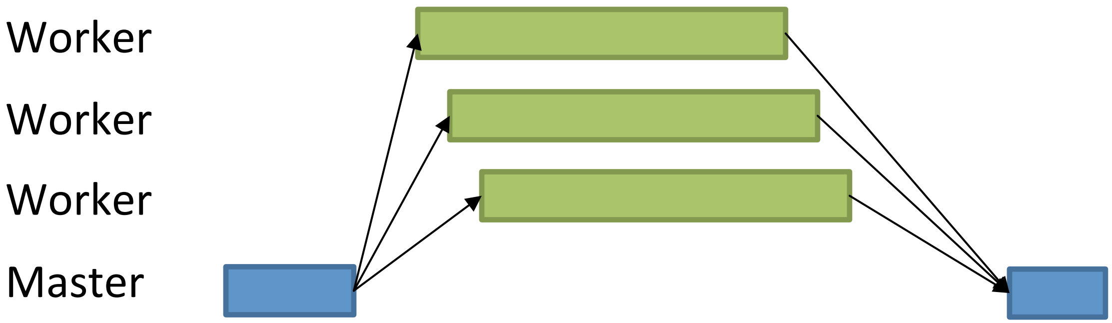
Further datatypes defined in the standard

Predefined reduction operations

Name	Meaning
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
MPI_BXOR	bit-wise exclusive or
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

Master worker

TECHNISCHE
UNIVERSITÄT
DARMSTADT



- Self-scheduling algorithm - master coordinates processing of tasks by providing input data to workers and collecting results
- Suitable if
  - Workers need not communicate with one another
  - Amount of work each worker must perform is difficult to predict
- Example: matrix-vector multiplication

Matrix-vector multiplication

TECHNISCHE UNIVERSITÄT DARMSTADT

$$A * \vec{b} = \vec{c}$$

Unit of work = dot product of one row of matrix A with vector b

Master

- Broadcasts b to each worker
- Sends one row to each worker
- Loop
  - Receives dot product from whichever worker sends one
  - Sends next task to that worker
  - Termination if all tasks are handed out

Worker

- Receives broadcast value of b
- Loop
  - Receives row from A
  - Forms dot product
  - Returns answer back to master

Macros

```
#include "mpi.h"
#define MAX_ROWS 1000
#define MAX_COLS 1000
#define MIN(a, b) ((a) > (b) ? (b) : (a))
#define DONE MAX_ROWS+1
```

Matrix-vector multiplication: common part

```
int main(int argc, char **argv) {
  double A[MAX_ROWS][MAX_COLS], b[MAX_COLS], c[MAX_ROWS];
  double buffer[MAX_COLS], ans;
  int myid, master, numprocs;
  int i, j, numsent, sender, done;
  int anstype, row;
  int rows, cols;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  master = 0;
  rows   = 100;
  cols   = 100;

  if (myid == master) { /* master code */
  } else { /* worker code */
  }
  MPI_Finalize();
  return 0;
}
```

Matrix-vector multiplication: master

```
/* Initialize A and b (arbitrary) */
[…]
numsent = 0;

/* Send b to each worker process */
MPI_Bcast(b, cols, MPI_DOUBLE, master, MPI_COMM_WORLD);

/* Send a row to each worker process; tag with row number */
for (i = 0; i < MIN(numprocs - 1, rows); i++) {
  MPI_Send(&A[i][0], cols, MPI_DOUBLE, i+1, i, MPI_COMM_WORLD);
  numsent++;
}
```

Matrix-vector multiplication: master (2)

```
for (i = 0; i < rows; i++) {
  MPI_Recv(&ans, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG,
           MPI_COMM_WORLD, &status);
  sender  = status.MPI_SOURCE;

  /* row is tag value */
  anstype = status.MPI_TAG;
  c[anstype] = ans;

  /* send another row */
  if (numsent < rows) {
    MPI_Send(&A[numsent][0], cols, MPI_DOUBLE, sender,
             numsent, MPI_COMM_WORLD);
    numsent++;
  } else {
    /* Tell sender that there is no more work */
    MPI_Send(MPI_BOTTOM, 0, MPI_DOUBLE, sender, DONE, MPI_COMM_WORLD);
  }
}
```

Matrix-vector multiplication: worker

```
MPI_Bcast(b, cols, MPI_DOUBLE, master, MPI_COMM_WORLD);

/* Skip if more processes than work */
done = myid > rows;

while (!done) {
  MPI_Recv(buffer, cols, MPI_DOUBLE, master, MPI_ANY_TAG, MPI_COMM_WORLD,
           &status);
  done = status.MPI_TAG == DONE;
  if (!done) {
    row = status.MPI_TAG;
    ans = 0.0;
    for (i = 0; i < cols; i++) {
      ans += buffer[i] * b[i];
    MPI_Send(&ans, 1, MPI_DOUBLE, master, row, MPI_COMM_WORLD);

  }
}
```

C-binding of blocking send and receive

TECHNISCHE UNIVERSITÄT DARMSTADT

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

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

Return status

TECHNISCHE UNIVERSITÄT DARMSTADT

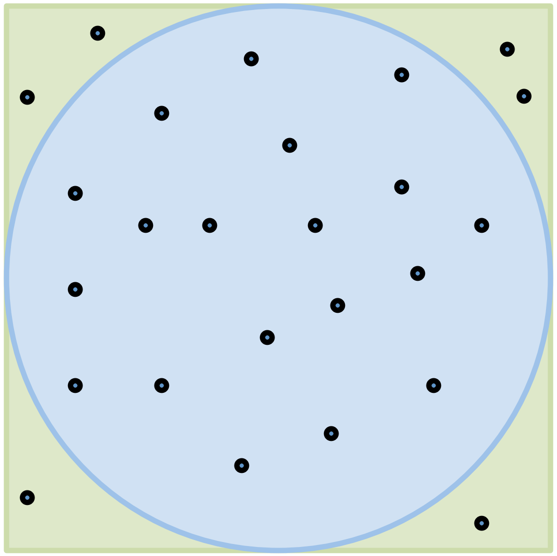
- In C, status is structure with three fields

MPI_SOURCE	rank of source process
MPI_TAG	tag of message
MPI_ERROR	error code

- The three fields provide information on the message actually received
- The number of entries received can be obtained using the function

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

Monte Carlo computation of π



Radius r = 1

Area of circle = π

Area of square = 4

Ratio of areas q = π / 4

=> π = 4q

Compute ratio q

- Generate random points (x,y) in the square
- Count how many turn out to be in the circle

Monte Carlo computation of π (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

Master

- Loop
  - Receives request from any worker
  - Generates pair of random numbers
  - Sends them to requesting worker

Worker

- Sends initial request to master
- Loop
  - Receives pair of random numbers and computes coordinates
  - Decides whether point sits inside circle
  - Synchronizes with all other workers to determine progress (i.e., accuracy of approximation)
  - Either terminates loop or requests new pair of random numbers

Monte Carlo computation of π (3)

TECHNISCHE UNIVERSITÄT DARMSTADT

- Every worker synchronizes with all other workers to determine progress
  - All workers calculate collectively current value of approximation
  - Then they compare it to the exact value of π

```
/* worker code */
[…]

MPI_Allreduce(&in, &totalin, 1 , MPI_INT, MPI_SUM,
              workers);
MPI_Allreduce(&out, &totalout, 1 , MPI_INT, MPI_SUM,
              workers);
Pi = (4.0*totalin)/(totalin + totalout);
error = fabs( Pi-3.141592653589793238462643);

[…]
```

Allreduce

```
int MPI_Allreduce(void *sendbuf, void *recvbuf,
                  int count, MPI_Datatype datatype,
                  MPI_Op op, MPI_Comm comm)
```

- Combines values from all processes and distributes the result back to all processes
  - sendbuf = starting address of send buffer
  - count = number of elements in send buffer
  - datatype = data type of elements of send buffer
  - op = reduce operation
  - comm = communicator

Using communicators

Two communicators

- World: all processes
- Workers: all processes except random number server

```
[…]
MPI_Comm_size(world,&numprocs);
MPI_Comm_rank(world,&myid);
server = numprocs-1;       /* last proc is server */
MPI_Comm_group( world, &world_group );
ranks[0] = server;
MPI_Group_excl( world_group, 1, ranks, &worker_group );
MPI_Comm_create( world, worker_group, &workers );
MPI_Group_free(&worker_group);
MPI_Group_free(&world_group);
[…]
MPI_Comm_free(&workers);
```

Frees only reference - not necessarily the entire object

Using communicators and groups

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

Accesses the group associated with given communicator

```
int MPI_Group_excl(MPI_Group group, int n, int *ranks,
                   MPI_Group *newgroup)
```

Produces a group by deleting those processes with ranks rank[0], …,rank[n-1]

```
int MPI_Group_free(MPI_Group *group)
```

Marks a group object for deallocation (frees reference)

Using communicators and groups (2)

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group,
                    MPI_Comm *newcomm)
```

Creates a new communicator from the specified group
(subset of parent communicator's group)

```
int MPI_Comm_free(MPI_Group *group)
```

Marks a communicator for deallocation (frees reference)

```
int MPI_Comm_create_group(MPI_Comm comm, MPI_Group group,
                          int tag, MPI_Comm *newcomm)
```

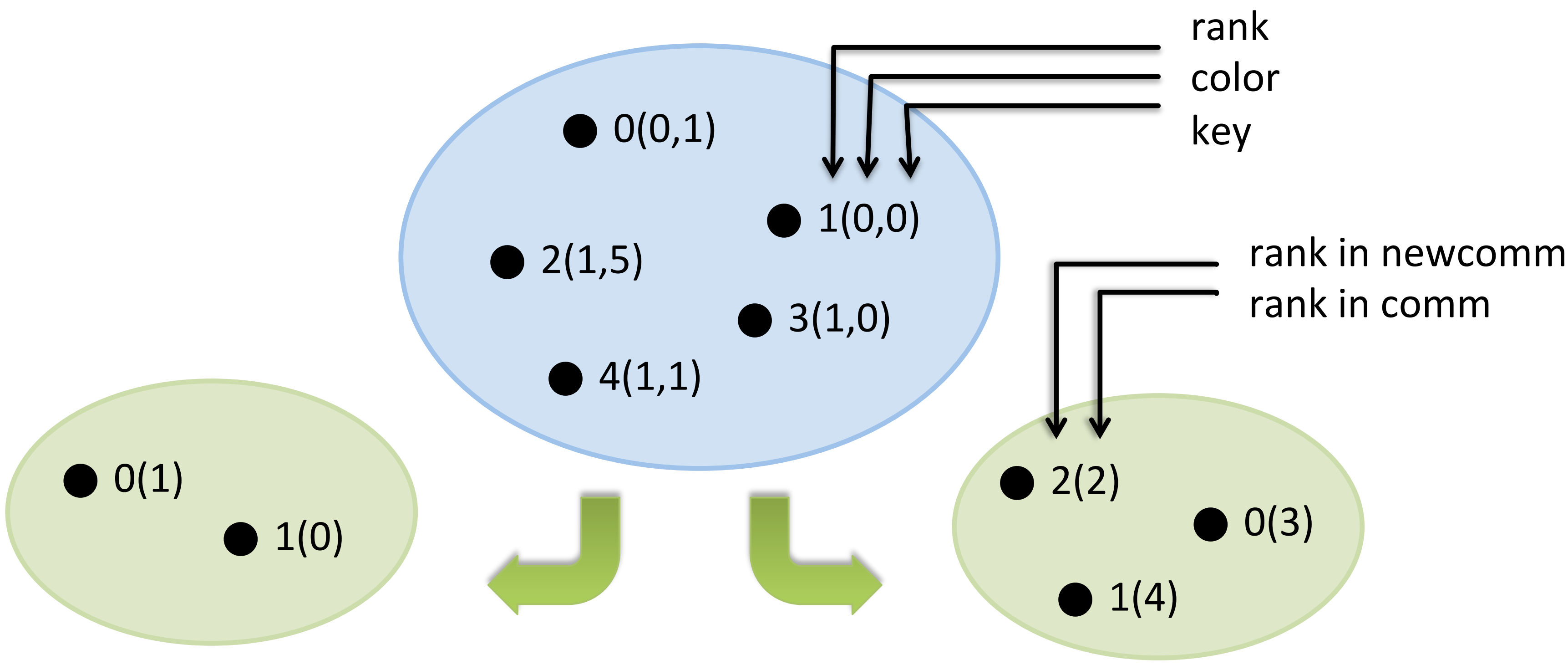
Similar to MPI_Comm_create except that MPI_Comm_create must be called by all processes in the group of comm, whereas MPI_Comm_create_group must be called by all processes in group, which is a subgroup of the group of comm (needed for fault tolerance)

Splitting communicators

```
int MPI_Comm_split(MPI_Comm comm,  int color, int key,
                   MPI_Comm *newcomm)
```

- Partitions the group associated with comm into disjoint subgroups, one for each value of color. Within each subgroup, the processes are ranked in the order defined by the value of key

Splitting communicators (2)



Further group and communicator operations

Groups

- Size and rank
- Translate ranks between groups
- Compare two groups
- Union, intersection, and difference
- New group from existing group via explicit inclusion
- Inclusion and exclusion of ranges with stride

Communicators

- Comparison
- Duplication

Summary

- Programming model – multiple processes with private address spaces communicate by exchanging messages
  - Between two processes via point-to-point communication
  - Between groups of processes via collective communication because more convenient & efficient
- Advantage – easy to understand
- Disadvantage – results in complex programs
- Central concept – communicators
  - Define group of processes
  - Define communication context (similar to wave length)