TECHNISCHE UNIVERSITÄT DARMSTADT

System and Parallel Programming

Prof. Dr. Felix Wolf

MESSAGE PASSING WITH MPI

Outline

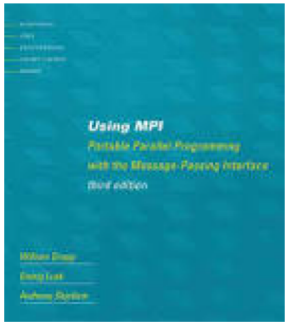
TECHNISCHE UNIVERSITÄT DARMSTADT

- Message-passing model
- Basic MPI concepts
- Essential MPI functions
- Simple MPI programs
- Collective communication
- Hybrid programming

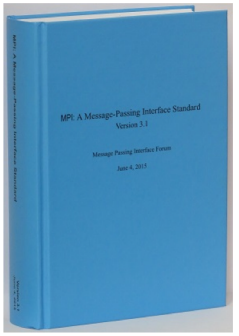
Literature

TECHNISCHE UNIVERSITÄT DARMSTADT

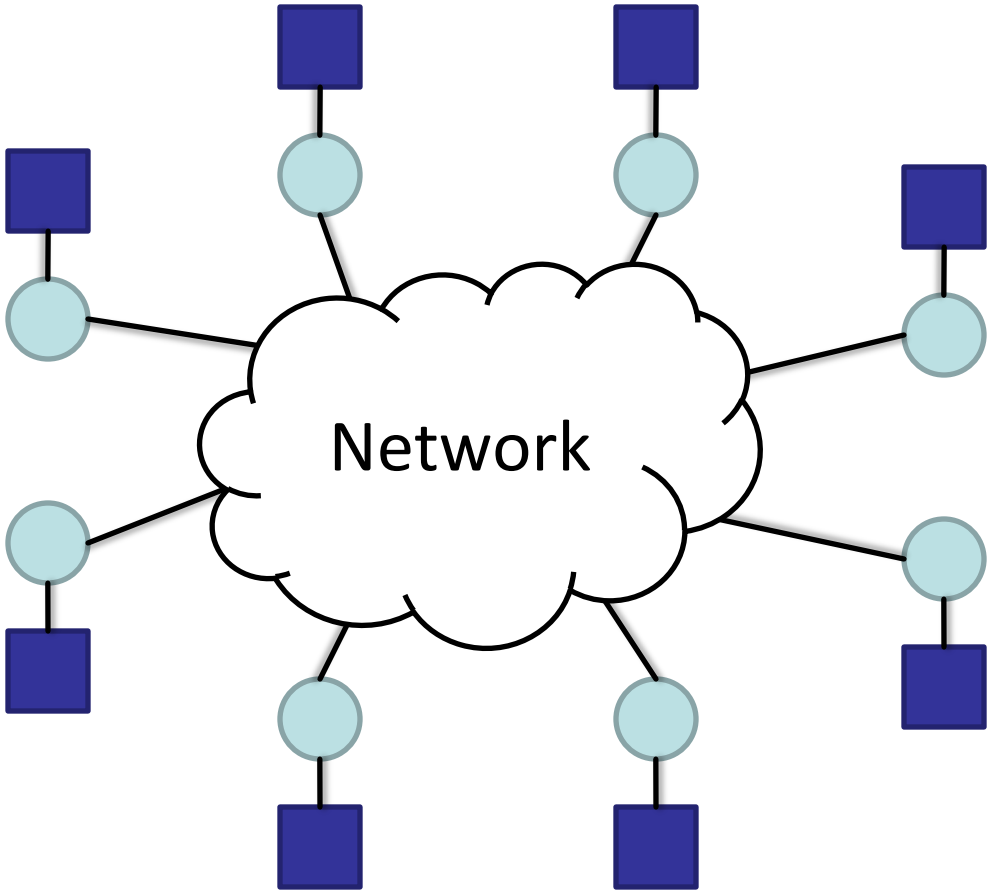
- Using MPI
  by William Gropp, Ewing Lusk, Anthony Skjellum,
  3rd edition, MIT Press, 2014

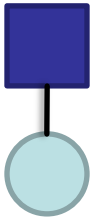


- MPI: A Message-Passing Interface Standard Version 3.1,
  Message Passing Interface Forum,
  http://www.mpi-forum.org



Message passing





Address space

Process

Message passing

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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?

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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)

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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 (e.g., MPICH and Open MPI)

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

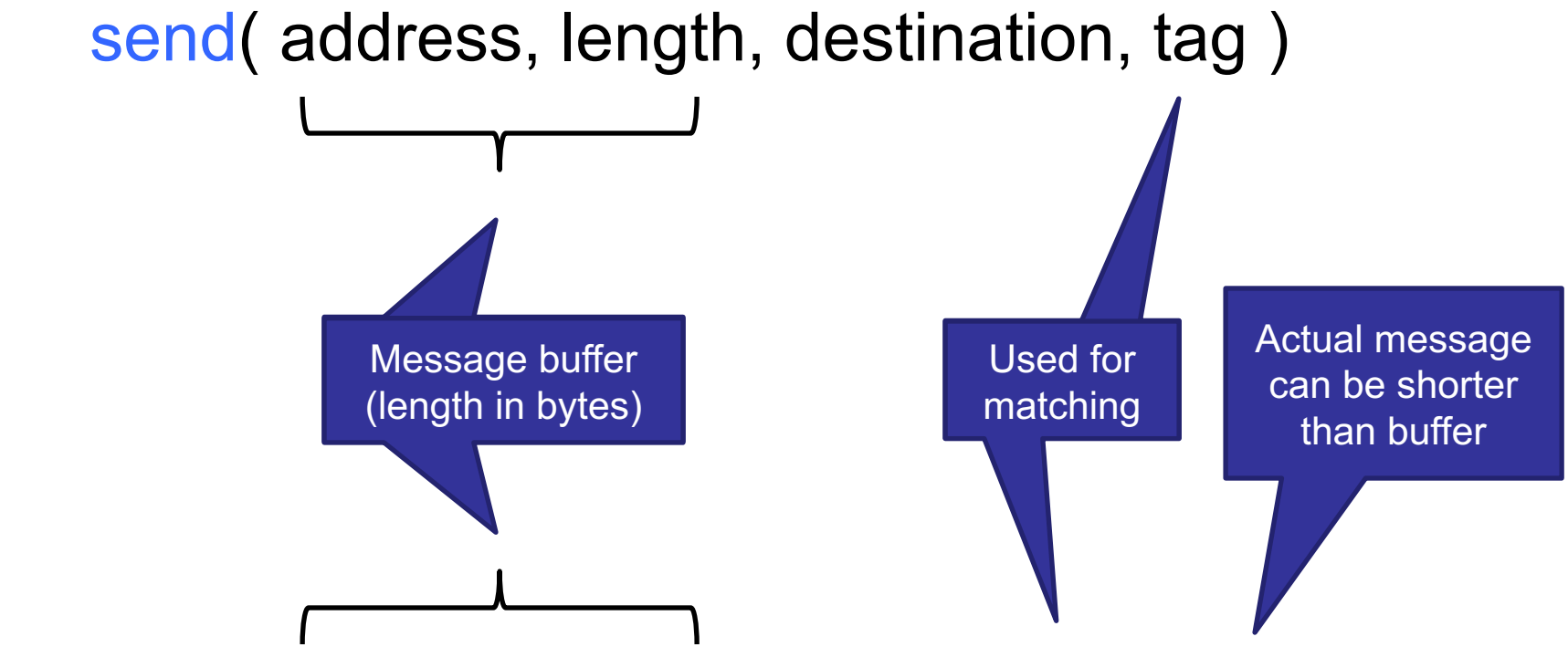
History (2)

- 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; C++ binding deprecated
  - New functionality with minor impact on implementations

History (3)

- 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



Message buffer

TECHNISCHE UNIVERSITÄT DARMSTADT

(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)

TECHNISCHE UNIVERSITÄT DARMSTADT

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

TECHNISCHE UNIVERSITÄT DARMSTADT

Matching of messages via tag

- 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 of context permitted

Naming processes

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

TECHNISCHE UNIVERSITÄT DARMSTADT

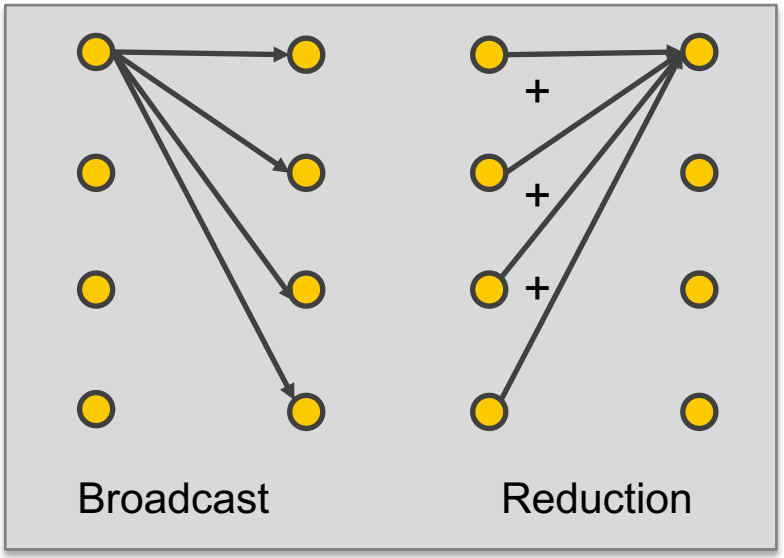
```
MPI_Recv(address, maxcount, datatype, source,
         tag, comm, status)
```

- 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

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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 and often inefficient



- MPI offers a range of predefined collective operations
  - Use optimized algorithms
  - May take advantage of hardware-specific features (e.g., network)

A six-function version of MPI

TECHNISCHE UNIVERSITÄT DARMSTADT

Function	Description
MPI_Init	Initialize MPI
MPI_Comm_size	Find out total number of processes
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

TECHNISCHE UNIVERSITÄT DARMSTADT

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

Contains

- Definition of named constants
- Function prototypes
- Type definitions

Opaque objects

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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

TECHNISCHE UNIVERSITÄT DARMSTADT

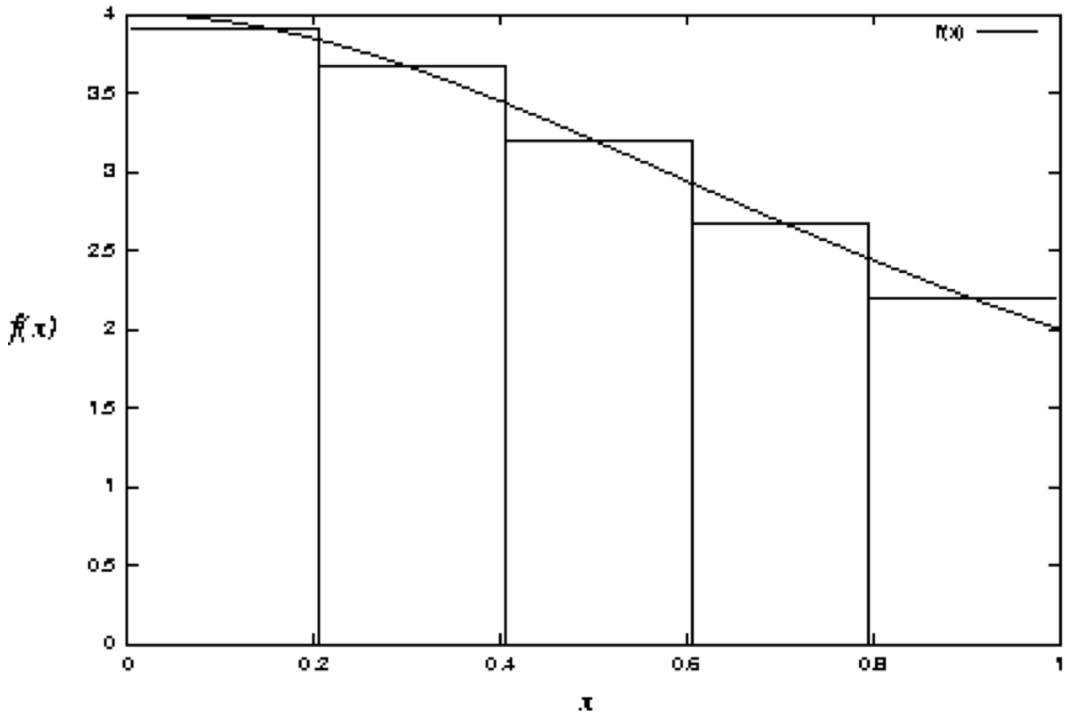
```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[] )
{
    int myid, numprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    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)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
int main( int argc, char *argv[] )
{
    int n, myid, numprocs, i;
    double PI25DT = 3.14159265358979323846264 3;
    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)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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

TECHNISCHE UNIVERSITÄT DARMSTADT

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	(not any C type)
MPI_PACKED	(not any C type)
MPI_LONG_LONG_INT	long long int (64 bit integer)

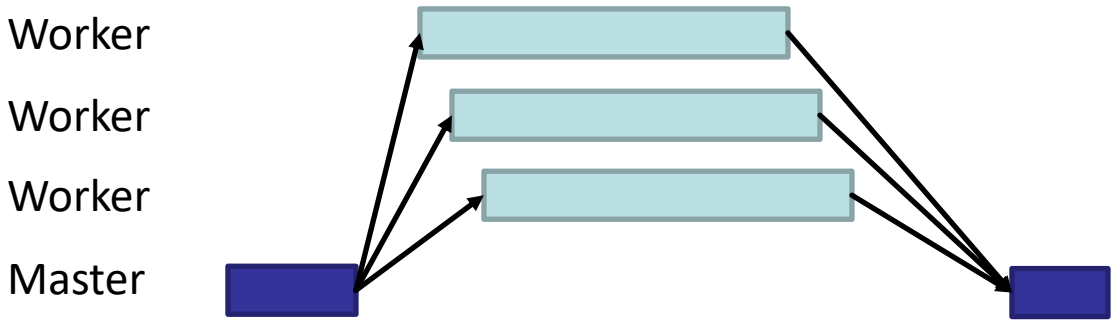
Further datatypes defined in the standard

Predefined reduction operations

TECHNISCHE UNIVERSITÄT DARMSTADT

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



- 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

TECHNISCHE UNIVERSITÄT DARMSTADT

```
#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

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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

TECHNISCHE UNIVERSITÄT DARMSTADT

```
/* 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)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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

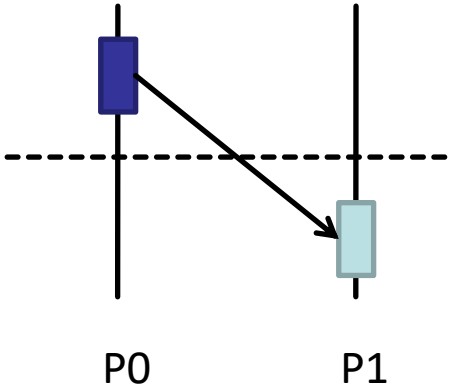
```
int MPI_Get_count(MPI_Status *status,
                  MPI_Datatype datatype, int *count)
```

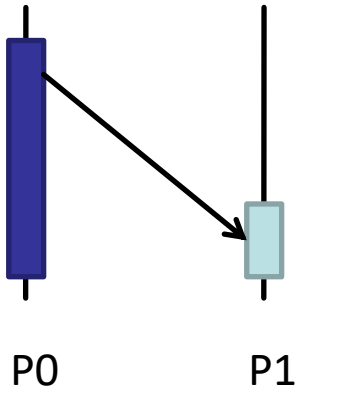
Possible blocking behavior of MPI_Send()

TECHNISCHE UNIVERSITÄT DARMSTADT

- Send can be started whether or not a matching receive has been posted
- Two possible modes for completion
  - Buffered – may complete before a matching receive is posted
  - Synchronous – completes successfully only if a matching receive has been posted and the receive operation has started

Choice usually depends on message size

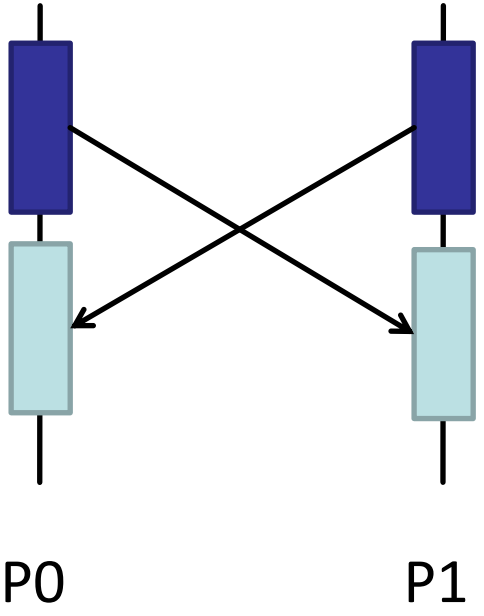




Deadlock

TECHNISCHE UNIVERSITÄT DARMSTADT

- What happens if the two sends do not return until the receivers have posted their receive?
- Definition
  - Two processes are deadlocked if each process is waiting for an event that only the other process can cause
  - If more than two processes are involved in a deadlock then they are waiting in a circular chain
- Analogy: chicken and egg problem



Combined send and receive

TECHNISCHE UNIVERSITÄT DARMSTADT

- Pairing sends and receives without deadlock can be difficult when the arrangement of processes is complex
  - Example: irregular grids
- Alternative: MPI_Sendrecv
  - Allows process to send and receive without worrying about deadlock from lack of buffering

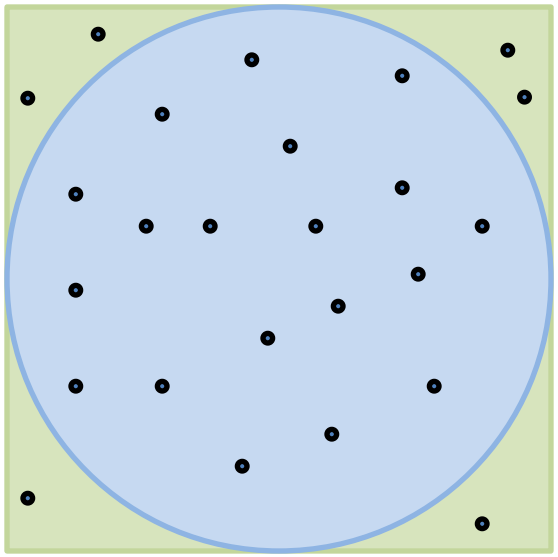
MPI_Sendrecv

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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)
```

- Executes a blocking send and receive operation
- Both send and receive use the same communicator, but possibly different tags
- Semantics as if the caller forked two concurrent threads, one to execute the send, and one to execute the receive, followed by a join of these two threads

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-3.14159265358979323846264 3);

[…]
```

Allreduce

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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_Group world_group, worker_group;
[…]
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_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)

TECHNISCHE UNIVERSITÄT DARMSTADT

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

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

```
int MPI_Comm_free(MPI_Comm *comm)
```

Marks a communicator for deallocation (frees reference)

Further group and communicator operations

TECHNISCHE UNIVERSITÄT DARMSTADT

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

- Split based on “color”
- Comparison
- Duplication

Summary – MPI basics

TECHNISCHE UNIVERSITÄT DARMSTADT

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

Collective operations

TECHNISCHE UNIVERSITÄT DARMSTADT

Solutions for recurring parallel group communication patterns

- Example: broadcast - make a value x residing on one process (P_0 w.l.o.g.) available to all others participating in the operation
  - Often needed during initialization

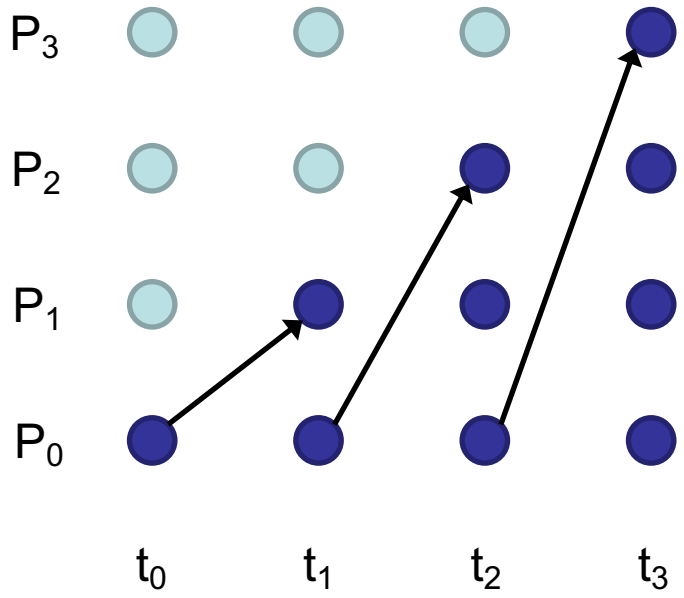
	P_0	P_1	P_2	P_3
Before	x			
After	x	x	x	x

- Typical for many collective operations: one process, the root, plays a special role

Broadcast implementation

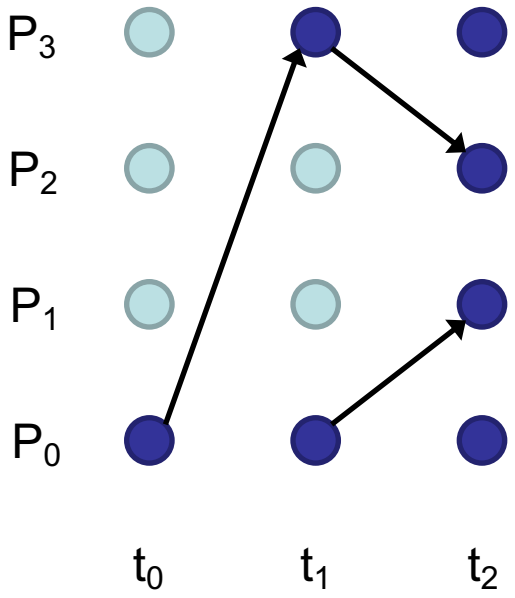
TECHNISCHE UNIVERSITÄT DARMSTADT

Naïve implementation



t =O(p)

Minimum-spanning tree algorithm



t =O(log p)

Broadcast – Minimum-spanning tree algorithm

TECHNISCHE UNIVERSITÄT DARMSTADT

```
MSTBcast(x, root, left, right)

if left == right return
mid = floor((left + right) / 2))
if root <= mid then dest = right else dest = left

if me == root then Send( x, dest )
if me == dest then Recv( x, root )

if me <= mid and root <= mid
  MSTBcast( x, root, left, mid )
else if me <= mid and root > mid
  MSTBcast( x, root, left, mid )
else if me > mid and root <= mid
  MSTBcast( x, root, left, mid )
else if me > mid and root > mid
  MSTBcast( x, root, left, mid )
```

Source: Ernie Chan, Marcel Heimlich, Avi Purkayastha: Collective communication: theory, practice, and experience. Concurrency Computat.: Pract. Exper. 2007; 19:1749–1783

Collective operations (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

- For many communication patterns exist highly efficient algorithms
- Re-implementing them manually is cumbersome, error prone, and potentially inefficient

```
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);

if (myid == root)
   for (int i=0; i< numprocs; i++)
       if (i != myid) MPI_Send(buf, count, datatype, i, sometag, comm);
else
   MPI_Recv(buf, count, datatype, root, sometag, comm);
```

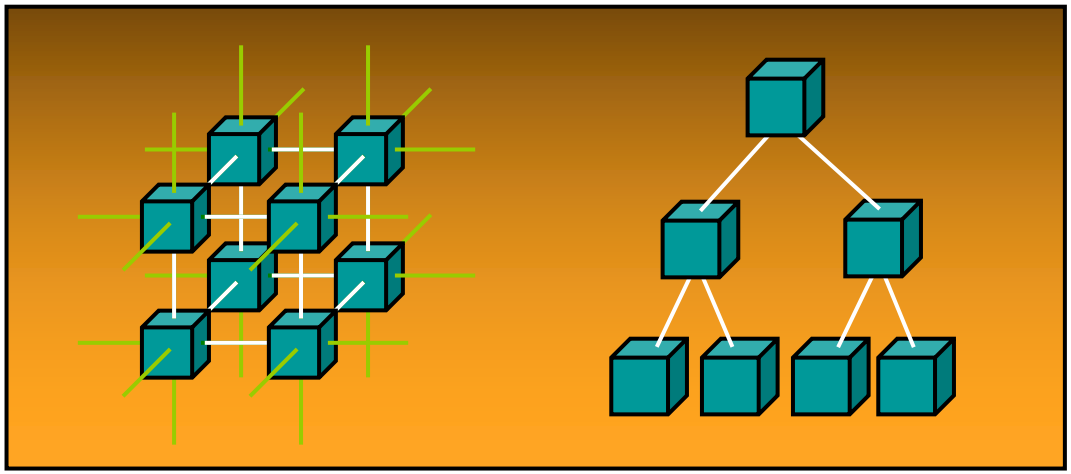
vs.

```
MPI_Bcast(buf, count, datatype, root, comm);
```

Collective operations (3)

TECHNISCHE UNIVERSITÄT DARMSTADT

- Implementations of predefined collective operations can also exploit special features of the target platform
- Example: collective tree network of IBM Blue Gene/P
  - One-to-all broadcast functionality
  - Support for reduction operations

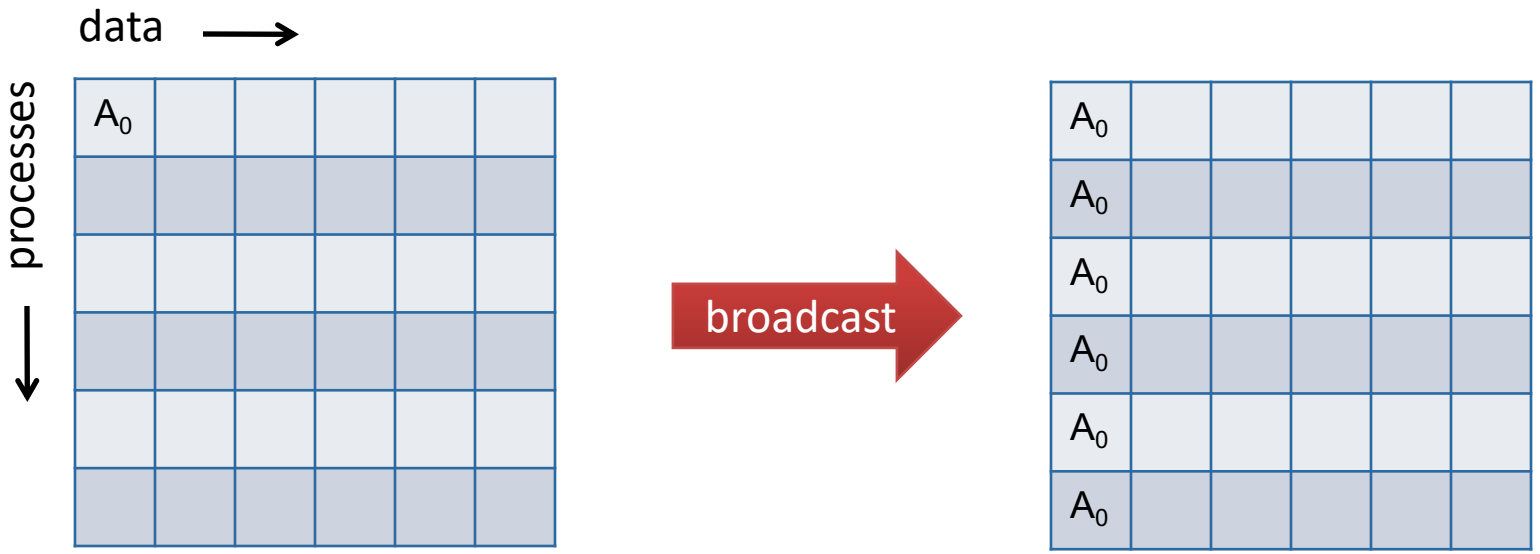


Collective operations - overview

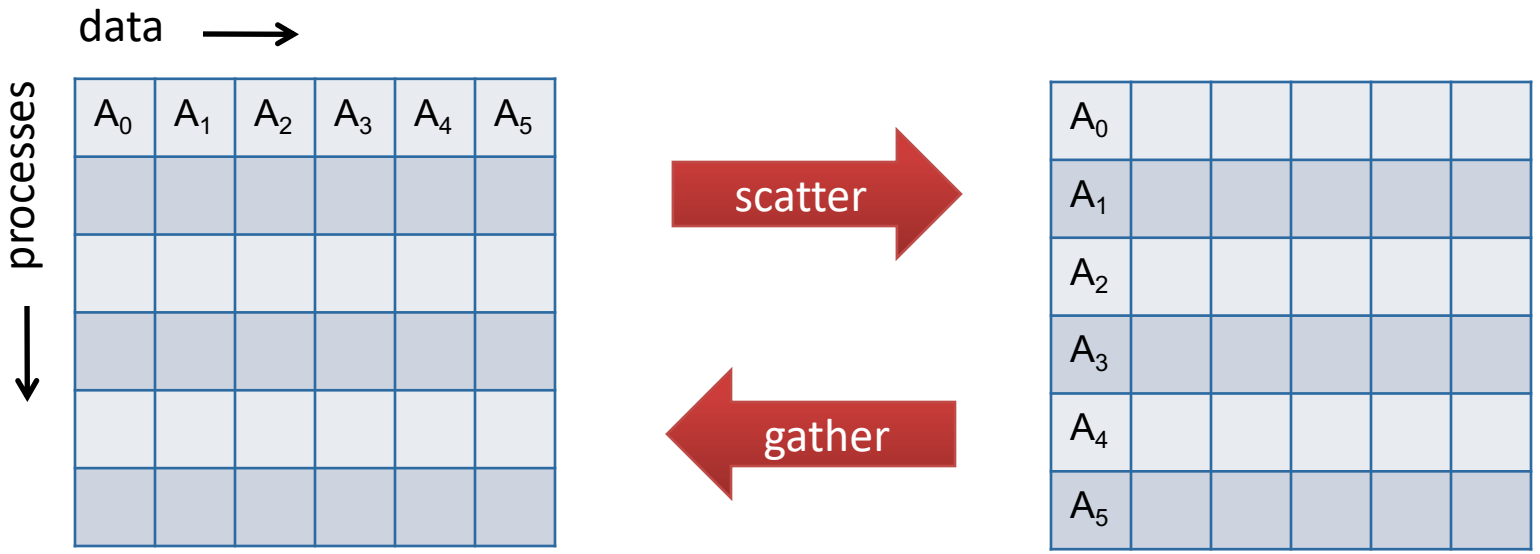
TECHNISCHE UNIVERSITÄT DARMSTADT

Barrier synchronization across all members	MPI_Barrier
Broadcast from one member to all members	MPI_Bcast
Gather data from all members to one member / all members	MPI_Gather MPI_Gatherv MPI_Allgather MPI_Allgatherv
Scatter data from one member to all members	MPI_Scatter MPI_Scatterv
Complete exchange (scatter / gather) from all members to all members	MPI_Alltoall MPI_Alltoallv MPI_Alltoallw
Global reduction operations where the result is returned to one member / all members	MPI_Reduce MPI_Allreduce
Combined reduction and scatter	MPI_Reduce_scatter
Scan across all members of a group	MPI_Scan MPI_Exscan

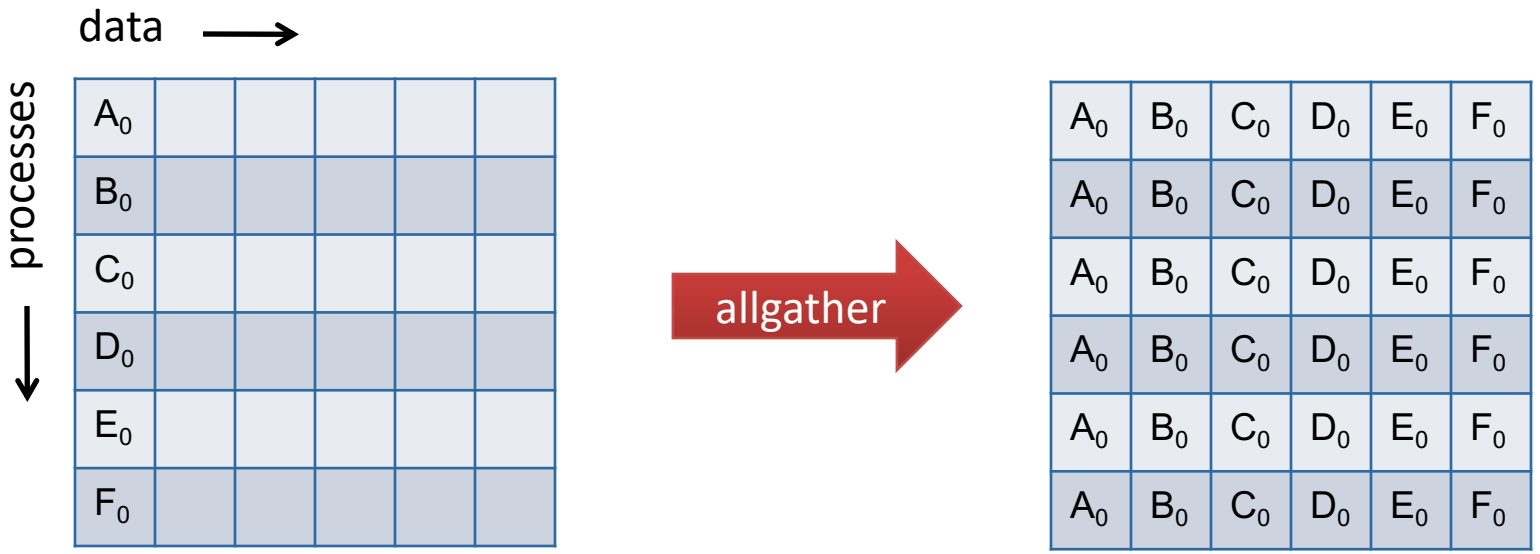
Broadcast



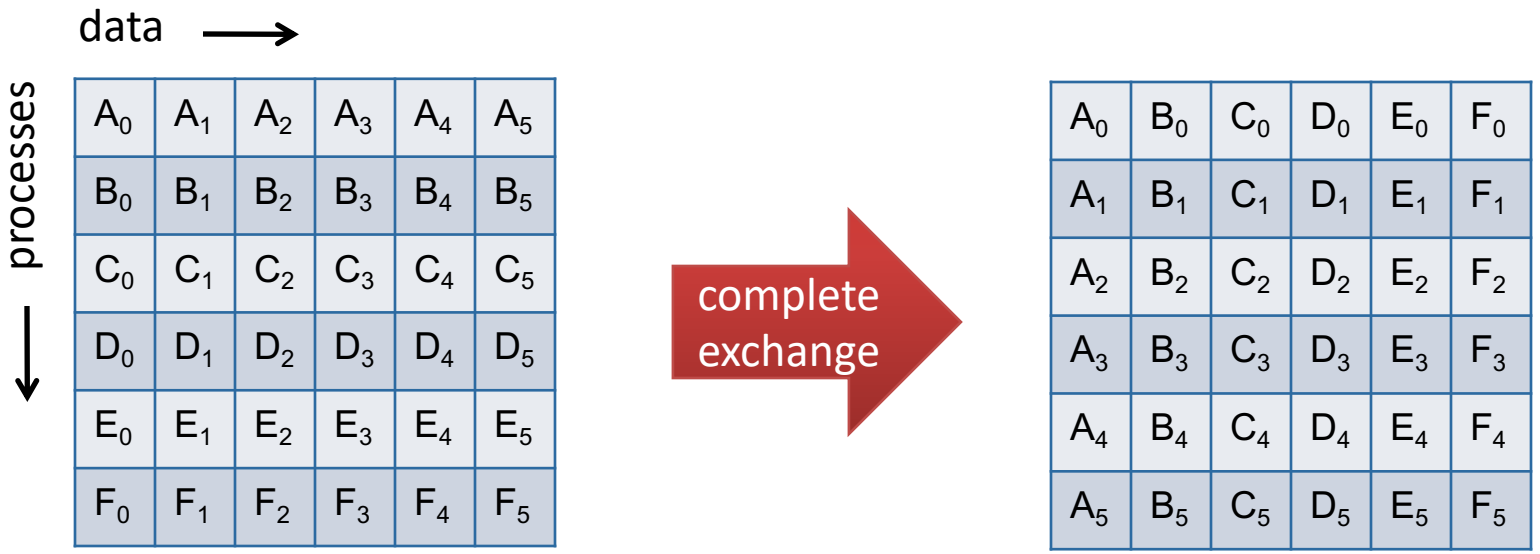
Scatter and gather



Allgather


data
processes
A0
B0
C0
D0
E0
F0
allgather
A0 B0 C0 D0 E0 F0
A0 B0 C0 D0 E0 F0
A0 B0 C0 D0 E0 F0
A0 B0 C0 D0 E0 F0
A0 B0 C0 D0 E0 F0
A0 B0 C0 D0 E0 F0

Complete exchange



Classification

TECHNISCHE UNIVERSITÄT DARMSTADT

All-to-all	
All processes contribute to the result and all processes receive the result	MPI_Allgather, MPI_Allgatherv MPI_Alltoall, MPI_Alltoallv, MPI_Alltoallw MPI_Allreduce, MPI_Reduce_scatter MPI_Barrier
All-to-one	
All processes contribute to the result and one process receives the result	MPI_Gather, MPI_Gatherv MPI_Reduce
One-to-all	
All processes receive the result	MPI_Bcast MPI_Scatter, MPI_Scatterv
Other	
Do not fit into one of the above categories	MPI_Scan, MPI_Exscan

Collective communication rules

TECHNISCHE UNIVERSITÄT DARMSTADT

- Several collective routines such as broadcast or gather have a single originating or receiving process (root)
  - Some arguments are significant only at root and are ignored by all others
- Type matching more strict than in the point-to-point case
  - The amount of data sent must exactly match the amount of data specified by the receiver

Collective communication rules (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

- Blocking
  - Collective calls may return as soon as their participation in the collective communication is complete
  - A collective call may or may not have the effect of synchronizing all calling processes
- All processes in the communicator must call the collective routine
- Often, collective communication can occur “in place” with the output buffer being identical to the input buffer
  - Specified by providing MPI_IN_PLACE instead of send or receive buffer, depending on the operation performed

Barrier synchronization

TECHNISCHE UNIVERSITÄT DARMSTADT

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

- Blocks the caller until all group members have called it. The call returns at any process only after all group members have entered the call
- Applications
  - Correctness (e.g., I/O, ready send)
  - Performance (e.g., limiting the number of messages in transit)

Gather


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

Number of items to be received from each process – not the total number of items

- Each process sends the contents of its send buffer to the root process
- The root process receives the messages and stores them in rank order

Gatherv

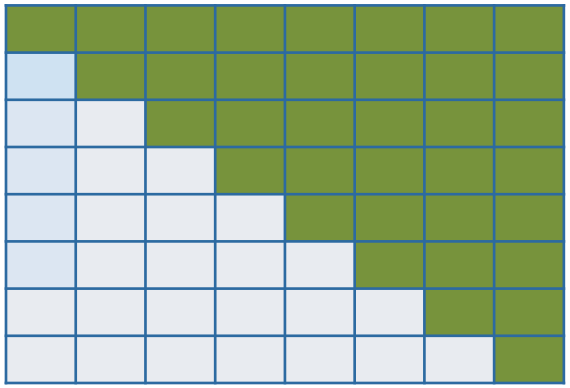

```
int MPI_Gatherv(void *sendbuf, int sendcount,
                MPI_Datatype sendtype,
                void *recvbuf, int *recvcounts,
                int *displs, MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```

- Extends the functionality of gather by allowing
  - A varying count of data from each process
  - More flexibility as to where the data is placed on the root via displacements
- The data received from process j is placed into recvbuf of the root process beginning at displs[j] elements (in terms of the recvtype)

Example

TECHNISCHE UNIVERSITÄT DARMSTADT

- The root process gathers 8-i doubles from each rank i and places them into the rows of an 8x8 array, filling the upper triangular matrix



Example (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int myrank;
double recvcounts[] = {8,7,6,5,4,3,2,1};
double displs[] =     {0,9,18,27,36,45,54,63};

MPI_Comm_rank(comm, &myrank);
MPI_Gatherv(sendbuf,
            8-myrank,              /* send count */
            MPI_DOUBLE,            /* send datatype */
            8x8_array,             /* receive buffer */
            recvcounts,            /* receive counts */
            displs,                /* displacements */
            MPI_DOUBLE,            /* receive datatype */
            root,
            comm);
```

Scatter and scatterv

TECHNISCHE UNIVERSITÄT DARMSTADT

Scatter is the inverse operation to gather

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

int MPI_Scatterv(void *sendbuf, int *sendcounts,
                 int *displs, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount,
                 MPI_Datatype recvtype,
                 int root, MPI_Comm comm
```

All-to-all exchange

- MPI_Allgather / MPI_Allgatherv
  - Can be thought of as a gather / gatherv, but where all processes receive the result, instead of just the root
- MPI_Alltoall / MPI_Alltoallv
  - An extension of allgather to the case where each process sends distinct data to each of the receivers
  - Alltoallv allows displacements to be specified for senders and receivers
- MPI_Alltoallw
  - Most general form of complete exchange
  - Allows separate specification of count, displacement, and datatype; displacement is specified in bytes

All-to-all exchange (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

- MPI_Allreduce
  - Like a normal reduce, except that result appears in the receive buffer of all group members
- MPI_Reduce_scatter_block and MPI_Reduce_scatter
  - The result of the reduce operation is scattered to all group members

Reduction

TECHNISCHE UNIVERSITÄT DARMSTADT

- Combine every element of a collection into a single element using an associative combiner function
  - Many different orderings possible
  - If the combiner function is commutative, even more orderings possible

Example: sum reduction

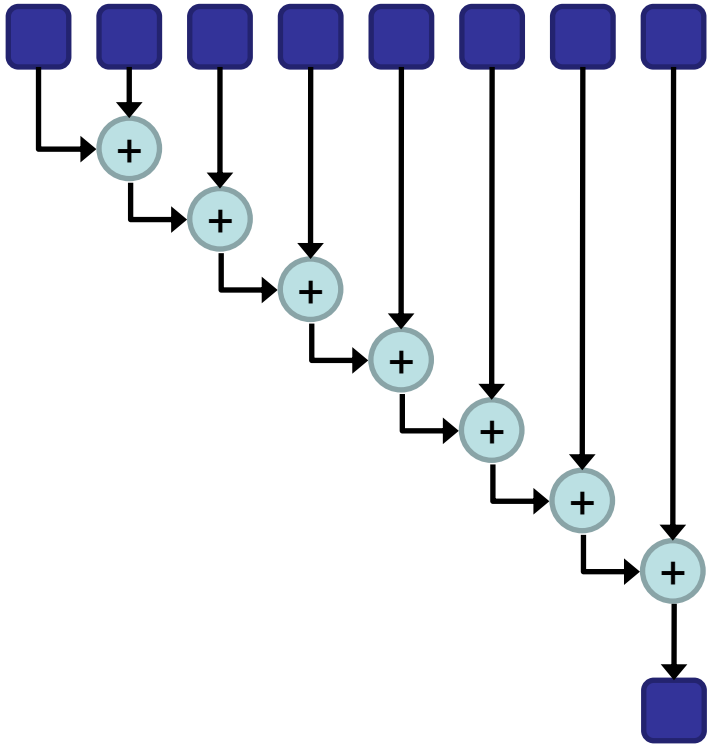
```
double sum = 0;
for ( int i = 0; i < n; i++ )
    sum = sum + a[i];
```

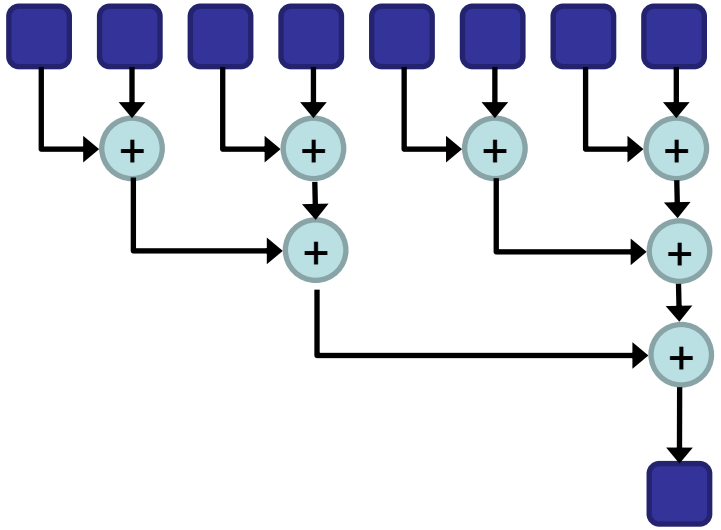
Applications of reduction

TECHNISCHE UNIVERSITÄT DARMSTADT

- Averaging of random Monte Carlo (random) samples
- Convergence testing in iterative solvers for systems of linear equations (e.g., conjugate gradient)
- Image comparison metrics (e.g., in video encoding)
- Row-column (dot) products in matrix multiplication

Serial vs. parallel implementation





Some typical combiner functions (e.g., floating-point add) not truly associative – precise result may depend on ordering

Reduce (+) – Minimum-spanning tree algorithm

TECHNISCHE UNIVERSITÄT DARMSTADT

```
MSTReduce(x, root, left, right)

if left == right return
mid = floor((left + right) / 2))
if root <= mid then src = right else src = left

if me <= mid and root <= mid
  MSTReduce( x, root, left, mid )
else if me <= mid and root > mid
  MSTReduce( x, root, left, mid )
else if me > mid and root <= mid
  MSTReduce( x, root, left, mid )
else if me > mid and root > mid
  MSTReduce( x, root, left, mid )

if me == src then Send( x, root )
if me == root then Recv( tmp, src ) and x = x + tmp
```

Source: Ernie Chan, Marcel Heimlich, Avi Purkayastha: Collective communication: theory, practice, and experience. Concurrency Computat.: Pract. Exper. 2007; 19:1749–1783

Reduction operations in MPI

TECHNISCHE UNIVERSITÄT DARMSTADT

- MPI_Reduce
- MPI_Allreduce
- MPI_Reduce_scatter_block
- MPI_Reduce_scatter
- MPI_Scan
- MPI_Exscan

Predefined reduction operations

TECHNISCHE UNIVERSITÄT DARMSTADT

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

Also user-defined operators possible

MINLOC and MAXLOC

TECHNISCHE UNIVERSITÄT DARMSTADT

- Operators that can be used to compute an extreme value (min/max) and the rank of the process containing this value

MPI_MAXLOC

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$

where

$$w = \max(u,v)$$

and

$$k = \begin{cases} i & if \quad u > v \\ \min(i,j) & if \quad u = v \\ j & if \quad u < v \end{cases}$$

MPI_MINLOC

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$

where

$$w = \min(u,v)$$

and

$$k = \begin{cases} i & if \quad u < v \\ \min(i,j) & if \quad u = v \\ j & if \quad u > v \end{cases}$$

MINLOC and MAXLOC (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

- These reduction operations are defined to operate on arguments that consist of a pair
- The C binding of MPI provides suitable pair types

Name	Description
MPI_FLOAT_INT	float and integer
MPI_DOUBLE_INT	double and integer
MPI_LONG_INT	long and integer
MPI_2INT	pair of integers
MPI_SHORT_INT	short integer and integer
MPI_LONG_DOUBLE_INT	long double and integer

Example

- Each process has an array of 30 doubles. For each of the 30 entries, compute the value and rank of the process containing the largest value

```
/* each process has an array of 30 double: ain[30] */
double ain[30], aout[30];
int ind[30];
struct {
  double val;
  int rank;
} in[30], out[30];
int i, myrank, root = 0;
```

Example (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
MPI_Comm_rank(comm, &myrank);

for (i=0; i<30; ++i) {
  in[i].val = ain[i];
  in[i].rank = myrank;
}

MPI_Reduce( in, out, 30, MPI_DOUBLE_INT, MPI_MAXLOC, root, comm );

/* At this point, the answer resides on process root */
if (myrank == root) {
  /* read ranks out */
  for (i=0; i<30; ++i) {
    aout[i] = out[i].val;
    ind[i] = out[i].rank;
  }
}
```

Scan

TECHNISCHE UNIVERSITÄT DARMSTADT

- Compute all partial reductions in a collection

Example: inclusive scan

```
if (n > 0) b[0] = a[0];
for ( int i = 1; i < n; i++ )
    b[i] = b[i-1] + a[i];
```

- Not obviously parallelizable – only after reordering operations
  - Total work increased in comparison to serial version

Scan in MPI

TECHNISCHE UNIVERSITÄT DARMSTADT

- Inclusive scan
  - Performs a prefix reduction on data distributed across the group

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

  - The operation returns, in the receive buffer of the process with rank i, the reduction of the values in the send buffers of the processes with ranks 0,…,i (inclusive)
- Exclusive scan (same arguments)
  - For processes with rank i > 0, the operation returns, in the receive buffer of the process with rank i, the reduction of the values in the send buffers of processes with ranks 0,…,i-1 (inclusive)

Summary – collective operations

TECHNISCHE UNIVERSITÄT DARMSTADT

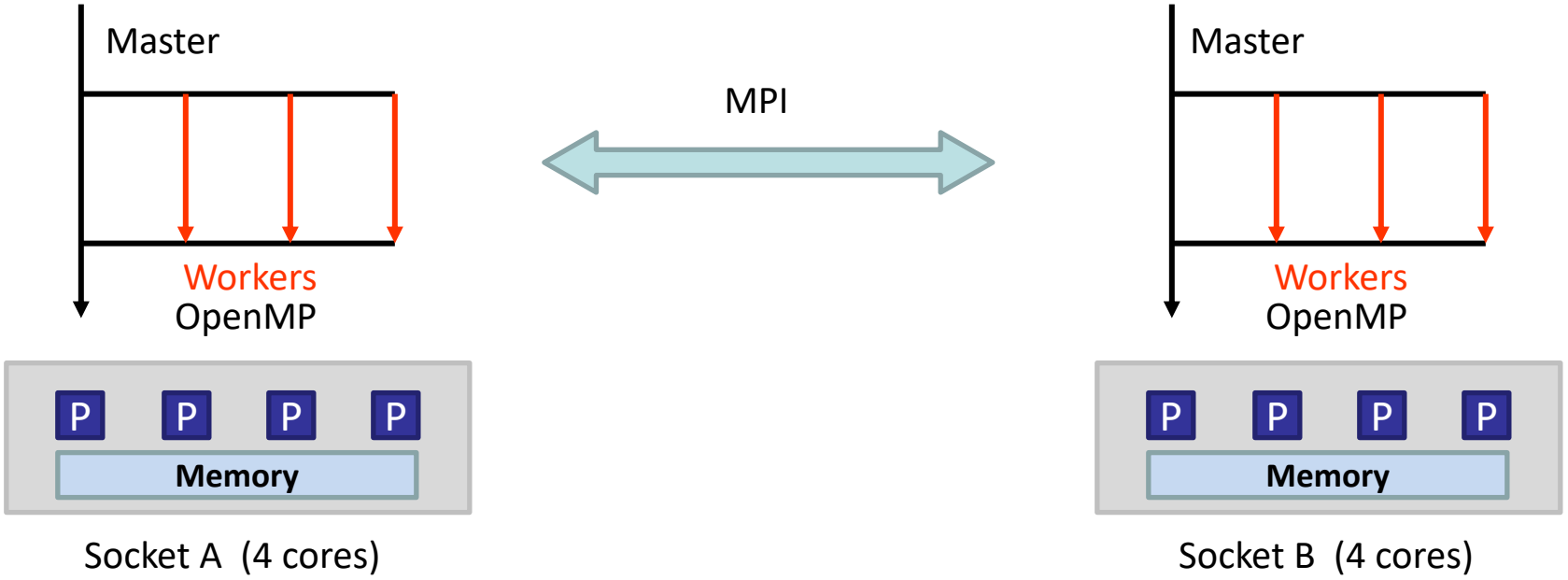
- Motivation – convenience and efficiency
- Classification
  - All-to-all, all-to-one, one-to-all, other
- Rules
  - Some operations have distinct root process
  - All processes of a communicator must call the operation
- Synchronization
  - Explicit synchronization via barrier
  - Some operations may involve implicit synchronization
- NOTE: So far, we have covered only blocking operations

Hybrid programming

TECHNISCHE UNIVERSITÄT DARMSTADT

- Most clusters are composed of shared-memory nodes
  - Rising numbers of cores per chip make them continuously wider
- Using one MPI process per core suffers from scalability and efficiency limitations
  - Extra memory needed to maintain separate private address spaces (e.g., for communication buffers)
  - Effort to copy data between these address spaces
  - Too many external MPI links per node
- Solution – combine MPI and OpenMP to reduce the number of MPI processes
  - Often one per socket instead of one per core

Hybrid programming



- Increased programming complexity
- Tuning of threads-per-process ratio difficult

MPI and threads

TECHNISCHE UNIVERSITÄT DARMSTADT

- In a thread-compliant implementation, an MPI process is a process that may be multi-threaded
  - Each thread can issue MPI calls
- However, threads are not separately addressable
  - A rank in a send or receive call identifies a process, not a thread
  - A message sent to a process can be received by any thread in this process

Initialization


```
int MPI_Init_thread(int *argc, char ***argv),
                    int required, int *provided)
```

- Initializes MPI in the same way as MPI_Init and initializes the thread environment
  - required = desired level of thread support
  - provided = level of thread support provided by the implementation
- Different processes in MPI_COMM_WORLD may require different levels of thread support

Thread support levels

TECHNISCHE UNIVERSITÄT DARMSTADT

MPI_THREAD_SINGLE

- Only one thread will execute

MPI_THREAD_FUNNELED

- Process may be multi-threaded, but the application must ensure that only the main thread makes MPI calls

MPI_THREAD_SERIALIZED

- Process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time (all MPI calls are serialized)

MPI_THREAD_MULTIPLE

- Multiple threads may call MPI, with no restrictions

These values are monotonic

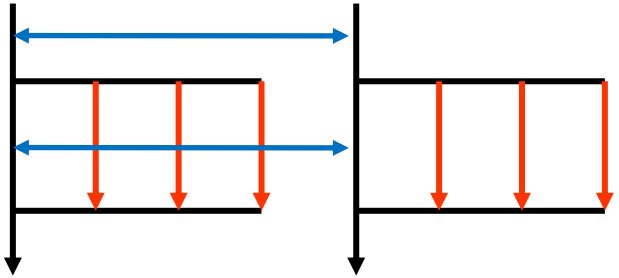
- MPI_THREAD_SINGLE < MPI_THREAD_FUNNELED < MPI_THREAD_SERIALIZED < MPI_THREAD_MULTIPLE

Thread support levels (2)

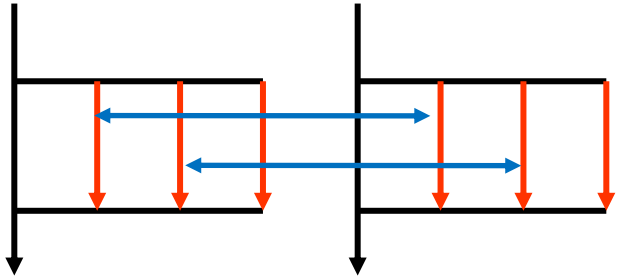
TECHNISCHE UNIVERSITÄT DARMSTADT

MPI_THREAD_SINGLE

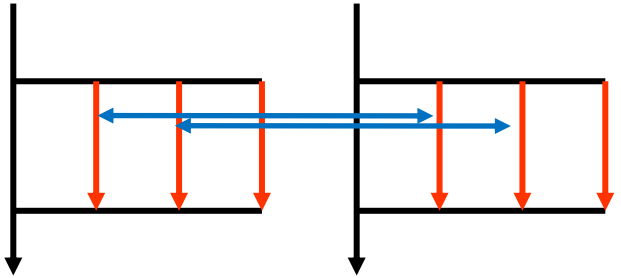
MPI_THREAD_FUNNELED



MPI_THREAD_SERIALIZED



MPI_THREAD_MULTIPLE



Thread support levels (3)

TECHNISCHE UNIVERSITÄT DARMSTADT

- If possible, the call will return provided = required
- Failing this, the call will return the least supported level such that provided > required
- If the user requirement cannot be satisfied, then the call will return in provided the highest supported level

- Advice to the programmer: check whether required <= provided

Thread-compliant MPI implementation

- Will be able to return provided = MPI_THREAD_MULTIPLE
- All MPI calls are thread-safe, i.e., two concurrently running threads may make MPI calls and the outcome will be as if the calls executed in some order
- MPI calls will block the calling thread only, allowing another thread to execute, if available.

Conflicting communication calls

TECHNISCHE UNIVERSITÄT DARMSTADT

- The user must prevent races when threads within the same application post conflicting communication calls
- The user can make sure that two threads in the same process will not issue conflicting communication calls by using distinct communicators at each thread
- Matching of collective calls on a communicator is done according to the order in which the calls are issued at each process
- If concurrent threads issue such calls on the same communicator inter-thread synchronization can be used to order them

Summary – hybrid programming

- Widening shared-memory nodes call for hybridization
  - OpenMP for work sharing within a node (or less)
  - MPI for parallelism among different nodes
- Four thread support levels in MPI
  - Single
  - Funneled
  - Serialized
  - Multiple

Features not covered

- Non-blocking communication
- Virtual topologies
- Intercommunicators
- Profiling interface
- Error handling
- Dynamic process management
- One-sided communication
- File I/O
- Neighborhood collectives