Parallel Programming 1: MPI

9th SimLab Course on Parallel Numerical Simulation
October 4—8, 2010, Belgrade, Serbia

Jérôme Frisch
Technische Universität München

Stabilitätspakt für Südosteuropa
Gefördert durch Deutschland
Stability Pact for South Eastern Europe
Sponsored by Germany

IGSSE TUM
International Graduate School
of Science and Engineering

Overview

- message passing paradigm
- collective communication
- programming with MPI

Message Passing Paradigm

Message passing

- very general principle, applicable to nearly all types of parallel architectures (message-coupled and memory-coupled)
- standard programming paradigm for MesMS, i. e.
  - message-coupled multiprocessors
  - clusters of workstations (homogeneous architecture, dedicated use, high-speed network (InfiniBand, e. g.))
  - networks of workstations (heterogeneous architecture, non-dedicated use, standard network (Ethernet, e. g.))
- several concrete programming environments
  - machine-dependent: MPL (IBM), PSE (nCUBE), …
  - machine-independent: EXPRESS, P4, PARMACS, PVM, …
- machine-independent standards: PVM, MPI

Message Passing Paradigm

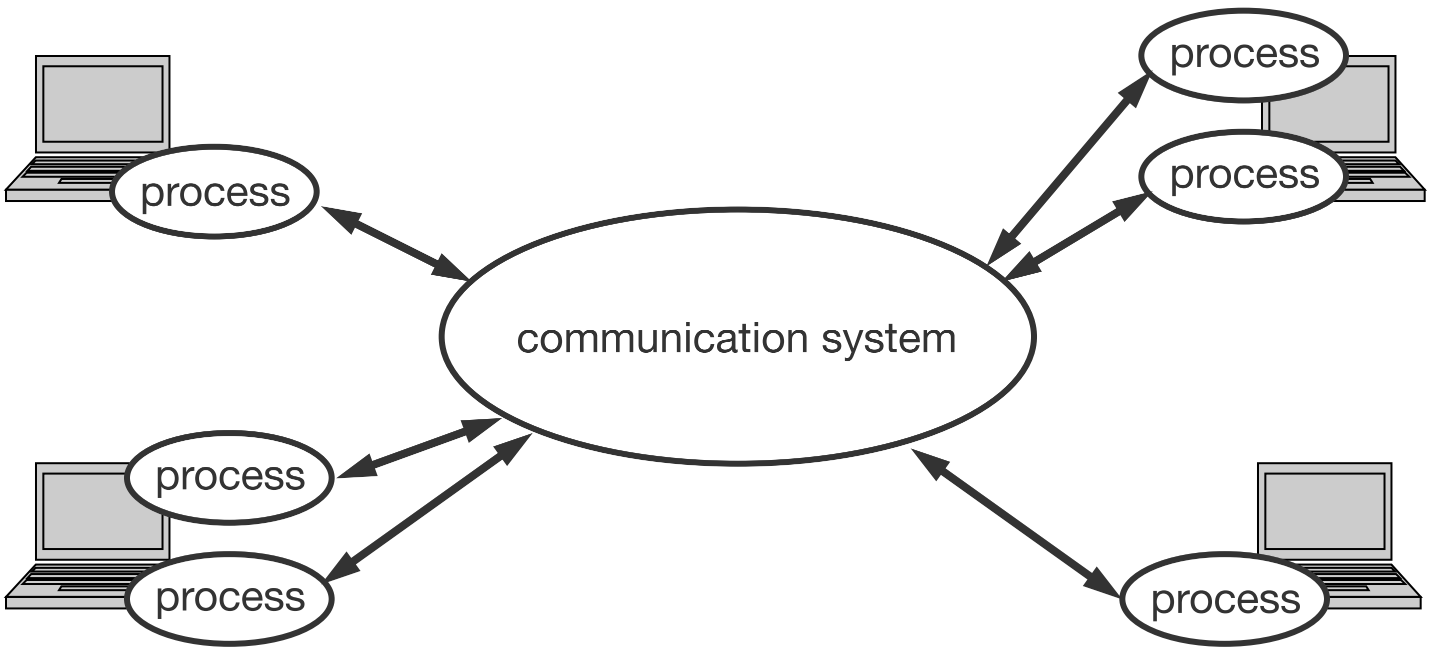
Underlying principle

- parallel program with P processes with different address space
- communication takes place via exchanging messages
  - header: target ID, message information (type of data, …)
  - body: data to be provided
- exchanging messages via library functions that should be
  - designed without dependencies of
    - hardware
    - programming language
  - available for multiprocessors and standard monoprocessors
  - available for standard languages such as C/C++ or Fortran
  - linked to source code during compilation

Message Passing Paradigm

User's view

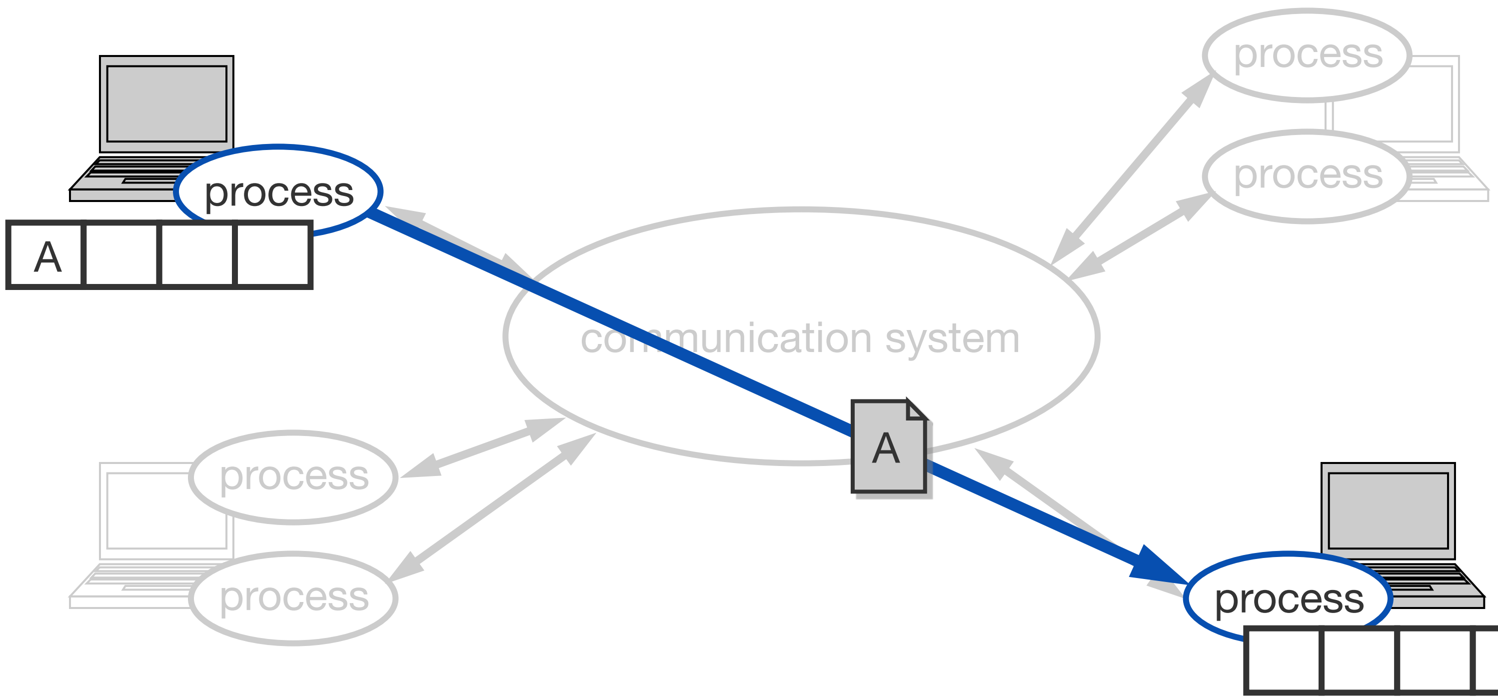
- library functions are the only interface to communication system



Message Passing Paradigm

User's view (cont'd)

- library functions are the only interface to communication system
- message exchange via **`send()`** and **`receive()`**



Message Passing Paradigm

Types of communication

- point-to-point a. k. a. P2P (1:1-communication)
  - two processes involved: sender and receiver
  - way of sending interacts with execution of sub-program
    - *synchronous*: send is provided information about completion of message transfer, i. e. communication not complete until message has been received (fax, e. g.)
    - *asynchronous*: send only knows when message has left; communication completes as soon as message is on its way (postbox, e. g.)
    - *blocking*: operations only finish when communication has completed (fax, e. g.)
    - *non-blocking*: operations return straight away and allow program to continue; at some later point in time program can test for completion (fax with memory, e. g.)

Message Passing Paradigm

Types of communication (cont'd)

- collective (1:M-communication, M ≤ P, P number of processes)
  - all (some) processes involved
  - types of collective communication
    - *barrier*: synchronises processes (no data exchange), i. e. each process is blocked until all have called barrier routine
    - *broadcast*: one process sends same message to all (several) destinations with a single operation
    - *scatter / gather*: one process gives / takes data items to / from all (several) processes
    - *reduce*: one process takes data items from all (several) processes and reduces them to a single data item; typical reduce operations: sum, product, minimum / maximum, …

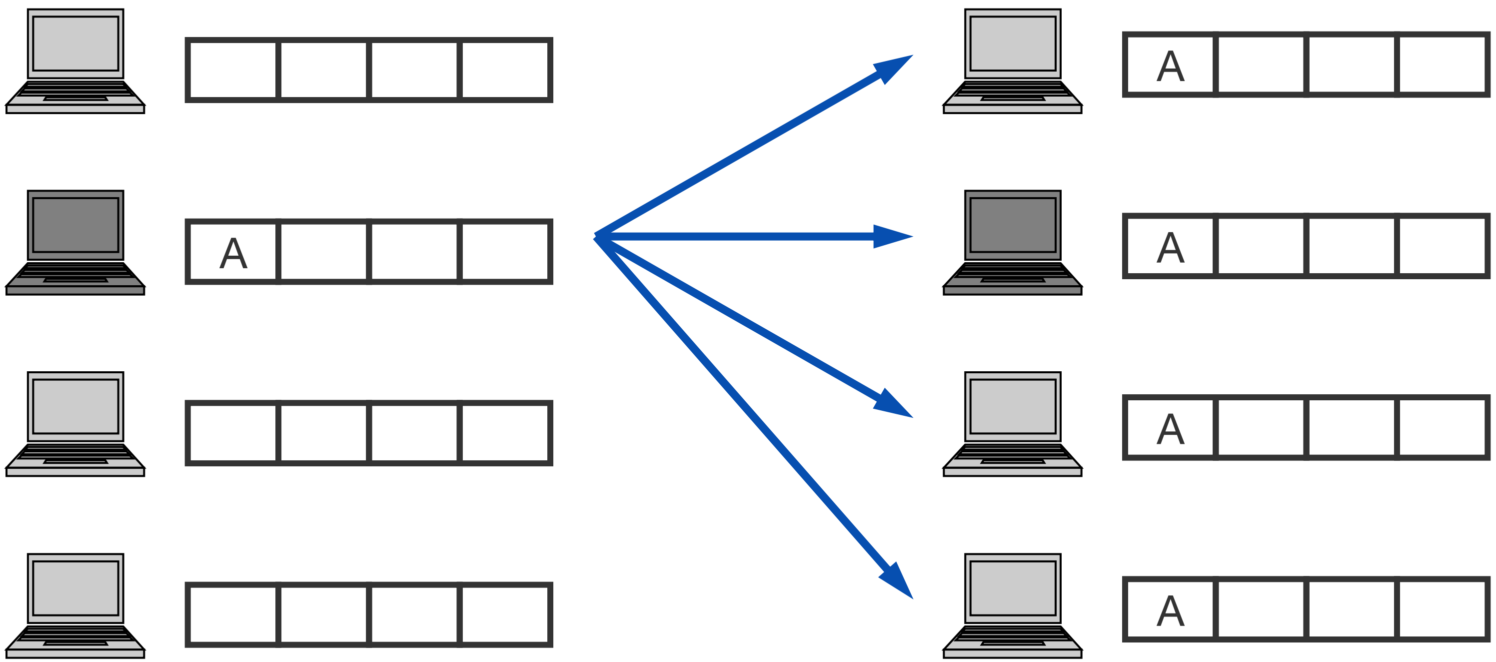
Overview

- message passing paradigm
- collective communication
- programming with MPI

Collective Communication

Broadcast

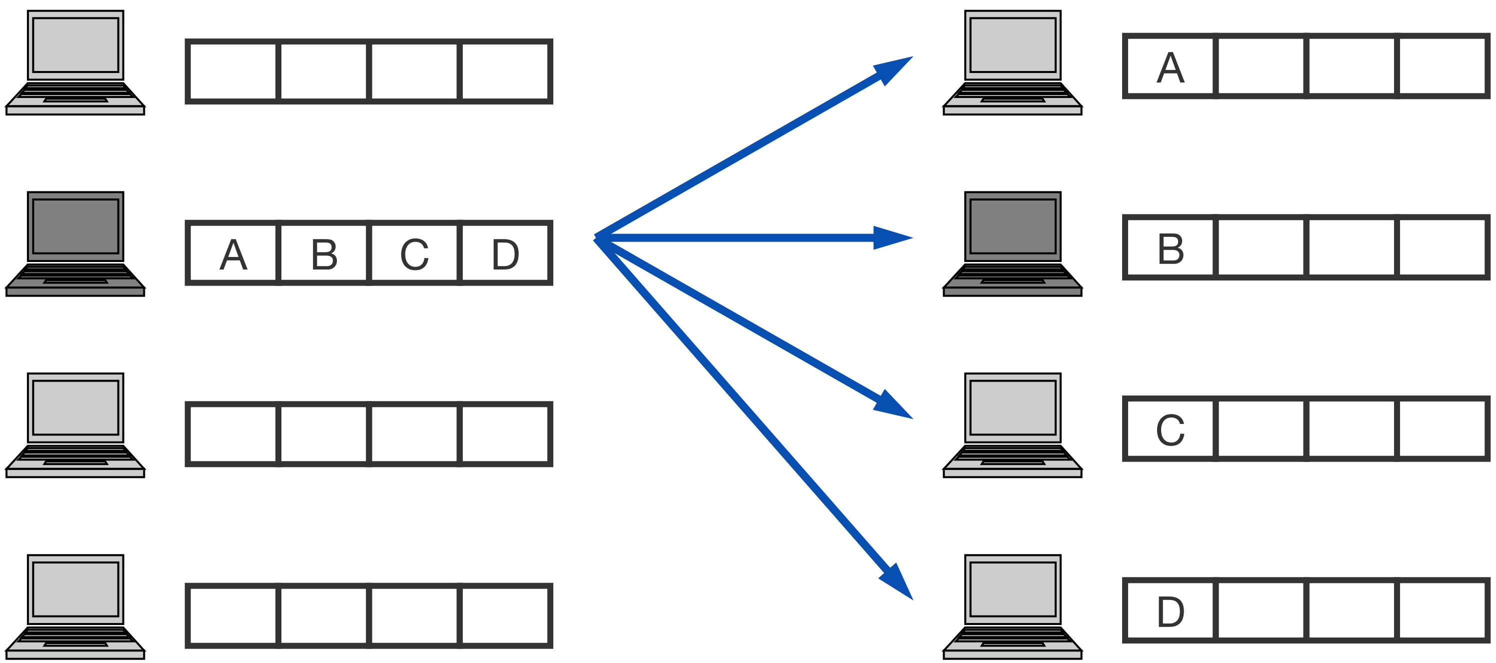
- sends same message to all participating processes
- example: first process in competition informs others to stop



Collective Communication

Scatter

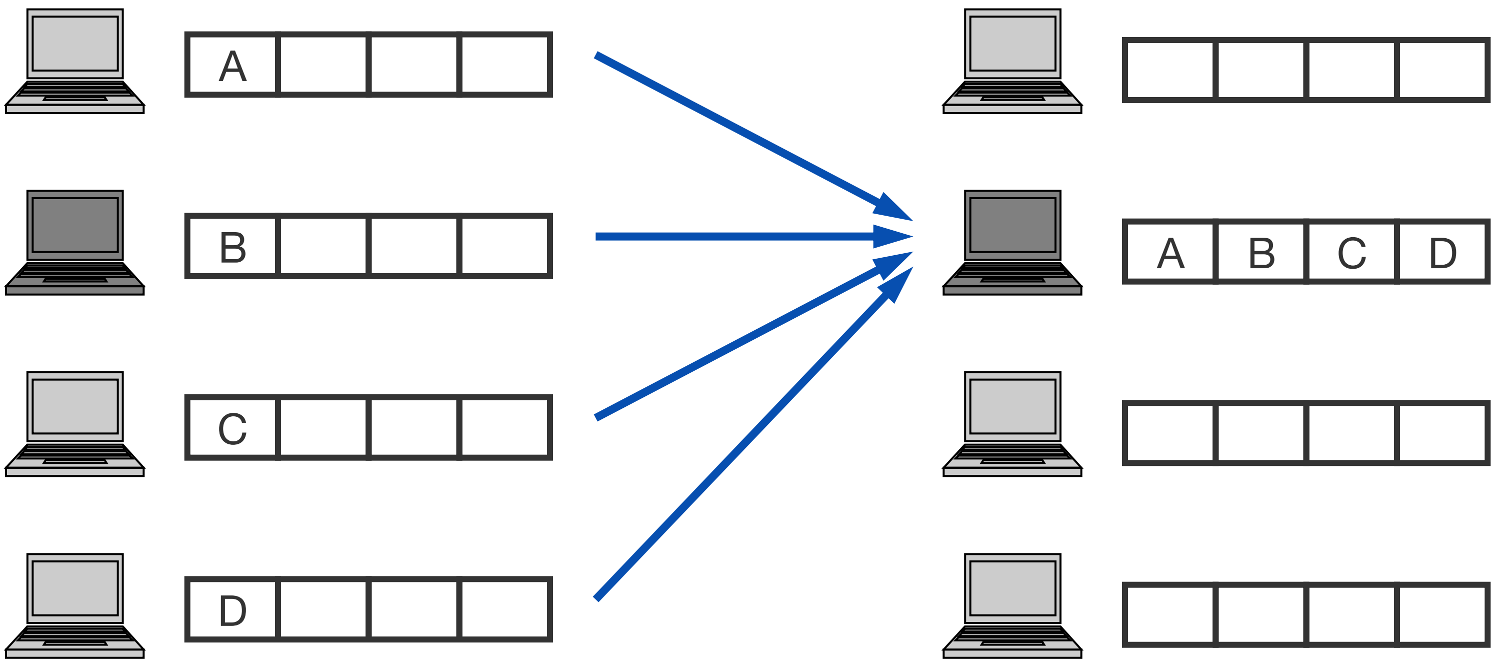
- data from one process are distributed among all processes
- example: rows of a matrix for a parallel solution of SLE



Collective Communication

Gather

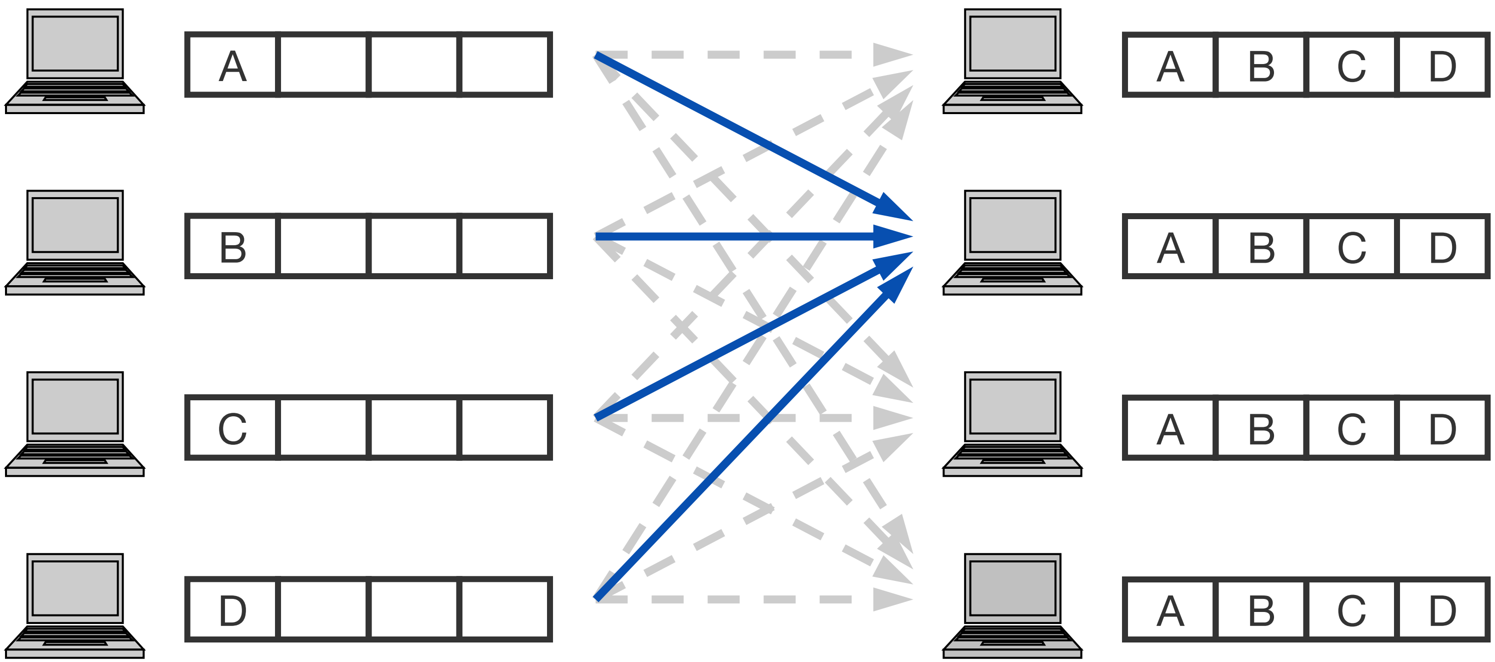
- data from all processes are collected by a single process
- example: assembly of solution vector from parted solutions



Collective Communication

Gather-to-all

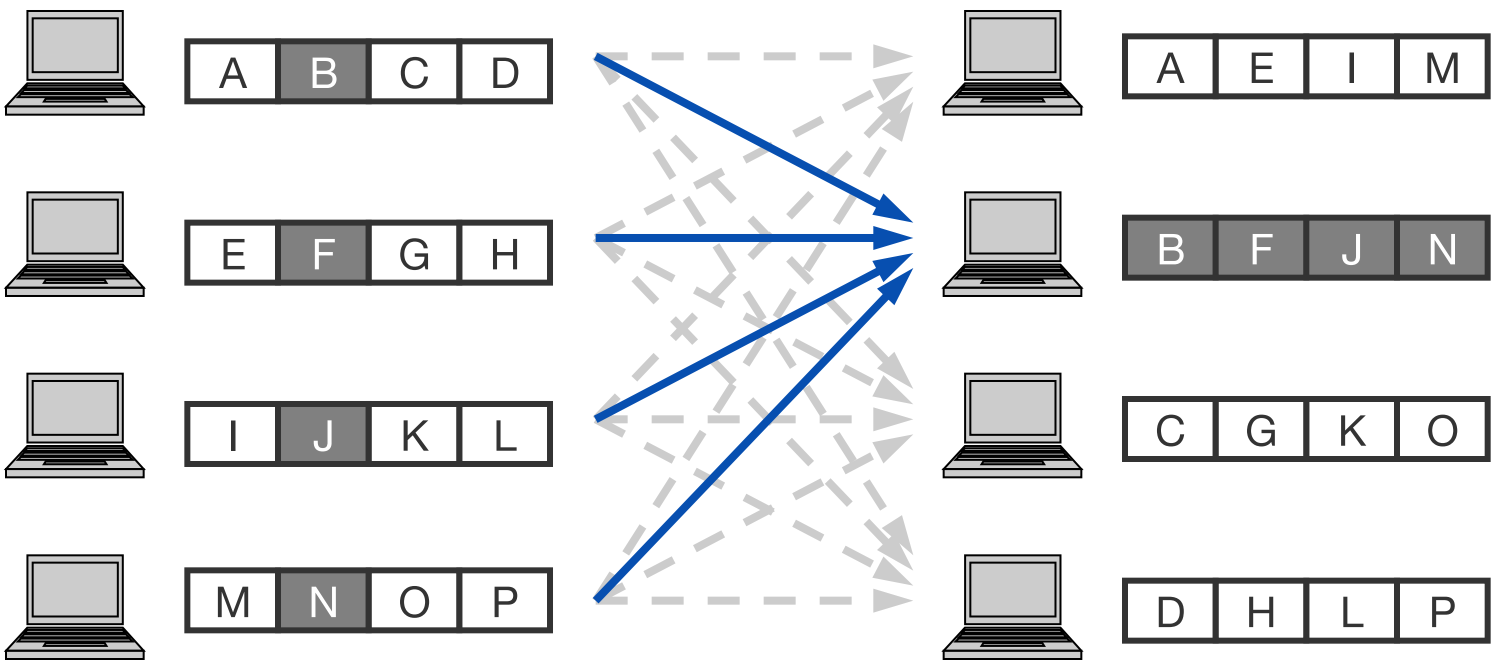
- all processes collect distributed data from all others
- example: as before, but now all processes need global solution for continuation



Collective Communication

All-to-all

- data from all processes are distributed among all others
- example: any ideas?



Collective Communication

All-to-all (cont'd)

- also referred to as *total exchange*
- example: transposition of matrix A (stored row-wise in memory)
  - total exchange of blocks B_{ij}
  - afterwards, each process computes transposition of its blocks

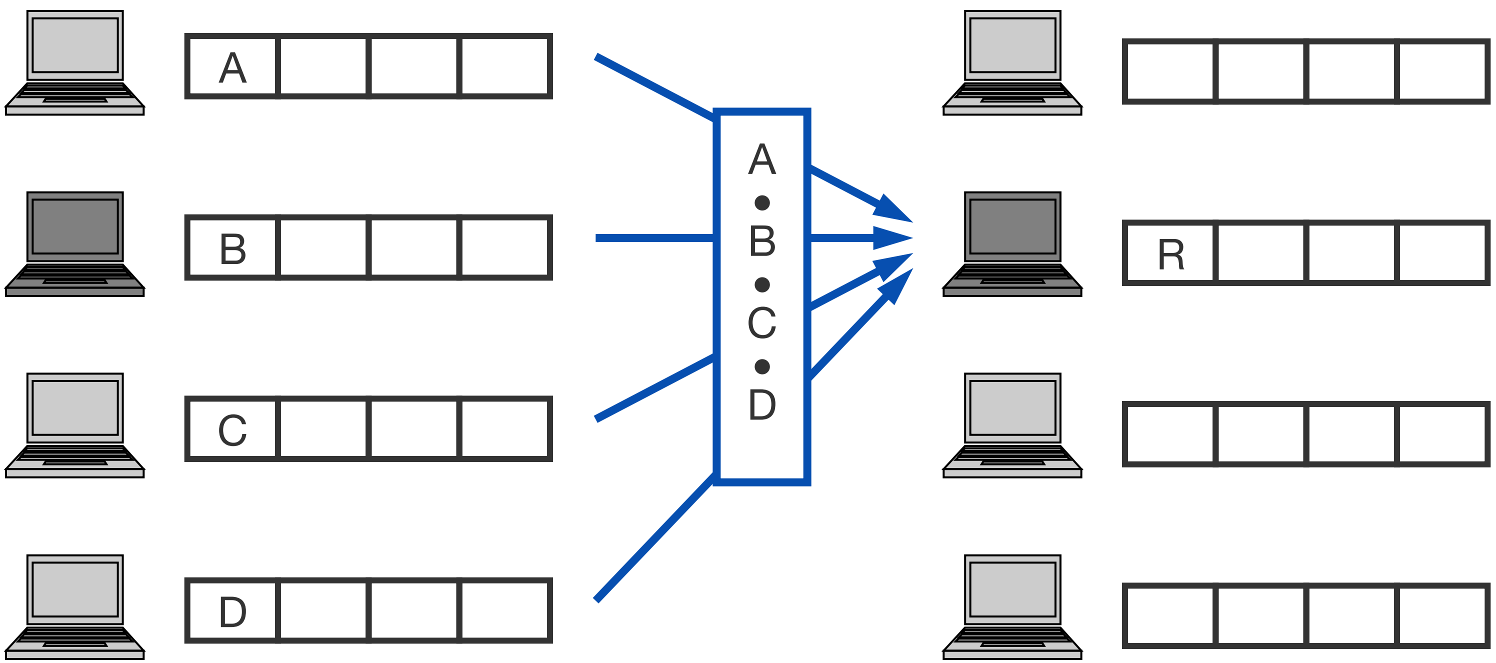
$$A = \begin{pmatrix} B_{11} & B_{12} & B_{13} & B_{14} \\ B_{21} & B_{22} & B_{23} & B_{24} \\ B_{31} & B_{32} & B_{33} & B_{34} \\ B_{41} & B_{42} & B_{43} & B_{44} \end{pmatrix} \quad \rightarrow \quad A^T = \begin{pmatrix} B_{11} & B_{21} & B_{31} & B_{41} \\ B_{12} & B_{22} & B_{32} & B_{42} \\ B_{13} & B_{23} & B_{33} & B_{43} \\ B_{14} & \boxed{B_{24}} & B_{34} & B_{44} \end{pmatrix}$$

$$\boxed{\begin{matrix} b_{11}^{(24)} & \cdots & b_{1N}^{(24)} \\ \vdots & \ddots & \vdots \\ b_{N1}^{(24)} & \cdots & b_{NN}^{(24)} \end{matrix}}$$

Collective Communication

Reduce

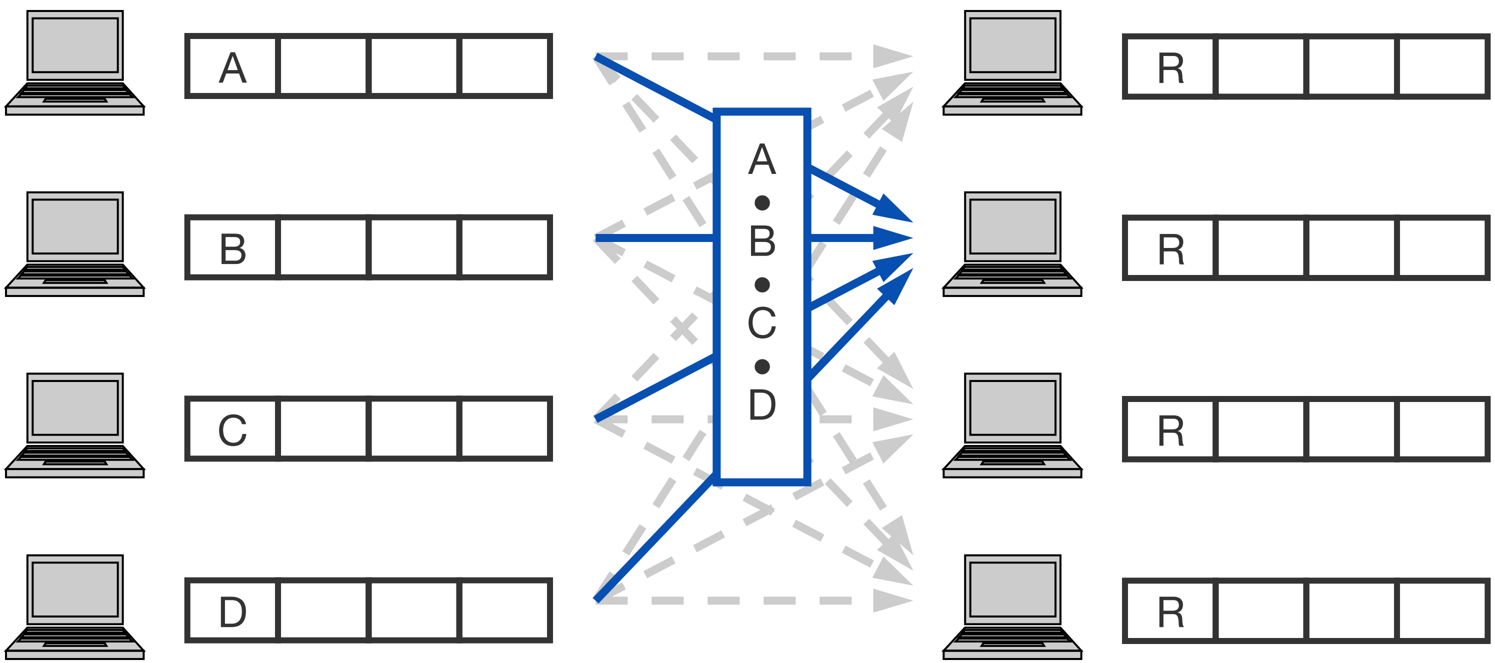
- data from all processes are reduced to single data item(s)
- example: global minimum / maximum / sum / product / …



Collective Communication

All-reduce

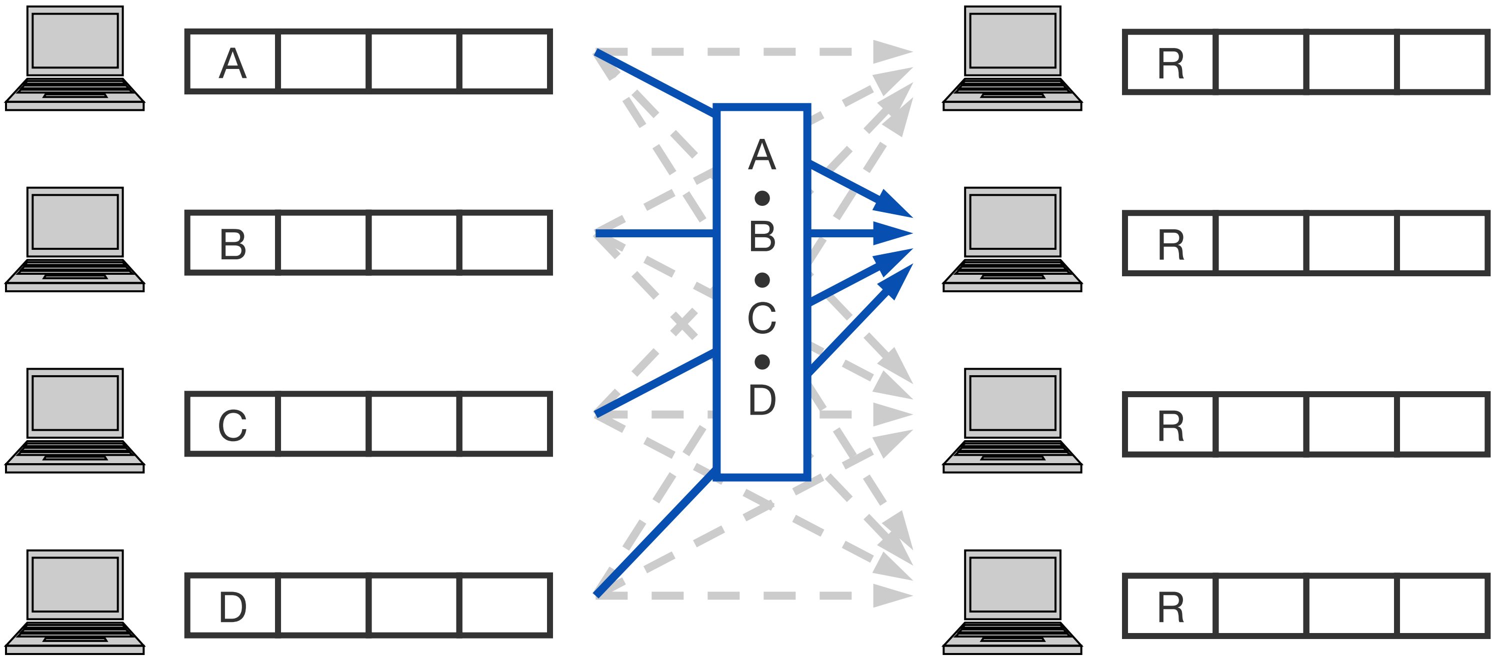
– all processes are provided reduced data item(s)
– example: finding prime numbers with "Sieve of ERATOSTHENES" ➔ processes need global minimum for deleting multiples of it



Collective Communication

All-reduce

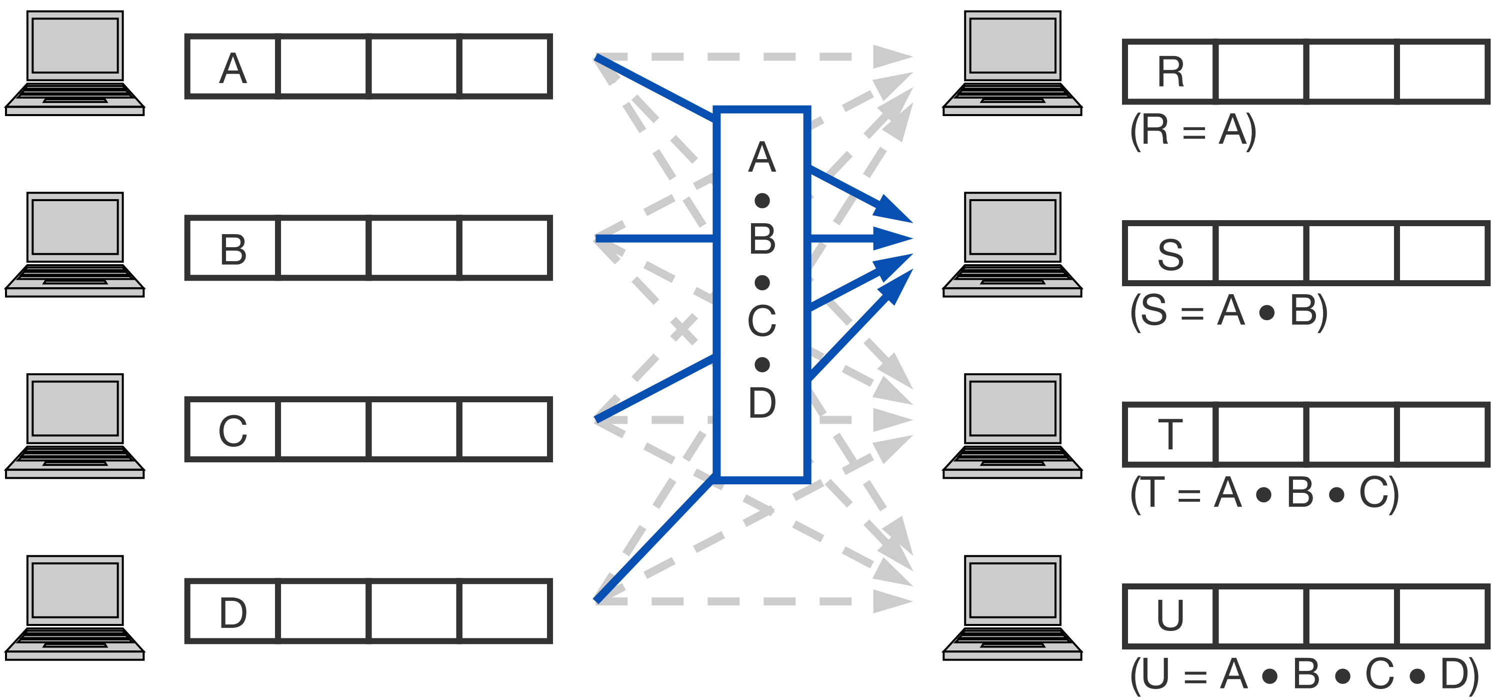
- all processes are provided reduced data item(s)
- example: finding prime numbers with "Sieve of ERATOSTHENES" ➔ processes need global minimum for deleting multiples of it



Collective Communication

Parallel prefix

- processes receive partial result of reduce operation
- example: matrix multiplication in quantum chemistry



Overview

- message passing paradigm
- collective communication
- programming with MPI

Programming with MPI

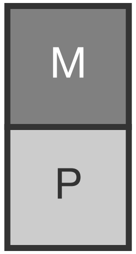
Brief overview

- de facto standard for writing parallel programs
- both free available and vendor-supplied implementations
- supports most interconnects
- available for C / C++, Fortran 77, and Fortran 90
  (C++ functionality deprecated in MPI 3.0)
- target platforms: SMPs, clusters, massively parallel processors

Programming with MPI

Programming model

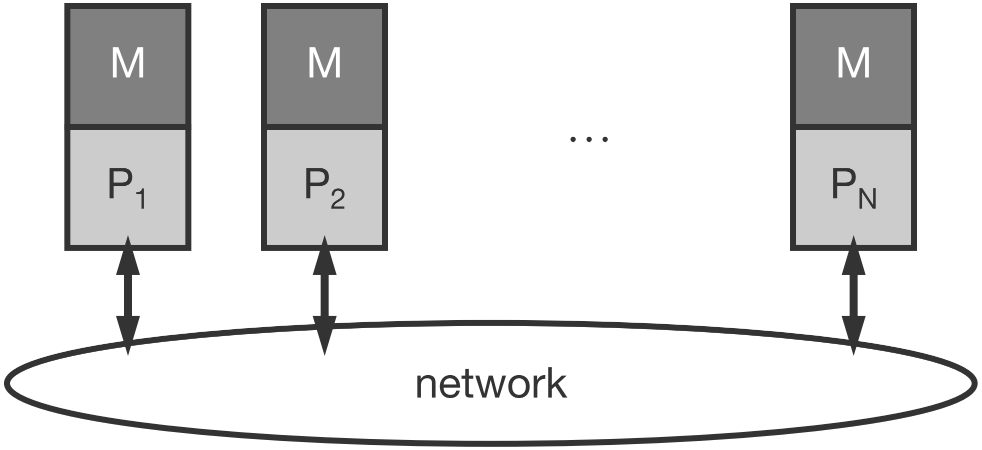
- sequential programming paradigm
  - one processor (P)
  - one memory (M)



Programming with MPI

Programming model (cont'd)

- message-passing programming paradigm
  - several processors / memories
  - each processor runs one or more processes
  - all data are private
  - communication between processes via messages



Programming with MPI

Writing and running MPI programs

- header file to be included: mpi.h
- all names of routines and constants are prefixed with MPI_
- first routine called in any MPI program must be for initialisation

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

- clean-up at the end of program when all communications have been completed

```
MPI_Finalize (void)
```

- `MPI_Finalize()` does not cancel outstanding communications
- `MPI_Init()` and `MPI_Finalize()` are mandatory

Programming with MPI

Writing and running MPI programs (cont'd)

- processes can only communicate if they share a communicator
  - predefined / standard communicator MPI_COMM_WORLD
  - contains list of processes
    - consecutively numbered from 0 (referred to as rank)
    - "rank" identifies each process within communicator
    - "size" identifies amount of all processes within communicator
  - why creating a new communicator
    - restrict collective communication to subset of processes
    - creating a virtual topology (torus, e. g.)
    - …

Programming with MPI

Writing and running MPI programs (cont'd)

– determination of rank

```
MPI_Comm_rank (communicator comm, int &rank)
```

– determination of size

```
MPI_Comm_size (communicator comm, int &size)
```

– remarks
  - rank $\in$ [0, size–1]
  - size has to be specified at program start
    - MPI-1: size cannot be changed during runtime
    - MPI-2: spawning of processes during runtime possible

Programming with MPI

Writing and running MPI programs (cont'd)

- compilation of MPI programs: mpicc, mpicxx, mpif77, or mpif90

```
$ mpicc [ -o my_prog ] my_prog.c
```

- available nodes for running an MPI program have to be stated explicitly via so called machinefile (list of hostnames or FQDNs)
- running an MPI program under MPI-1

```
$ mpirun -machinefile <file> -np <#procs> my_prog
```

- running an MPI program under MPI-2 (mpd is only started once)

```
$ mpdboot -n <#mpds> -f <file>
$ mpiexec -n <#procs> my_prog
```

- clean-up after usage (MPI-2 only): mpdcleanup –f <file>

Programming with MPI

Writing and running MPI programs (cont'd)

– example

```
int main (int argc, char **argv) {
    int rank, size;

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

    if (rank == 0) printf ("%d processes alive\n", size);
    else printf ("Slave %d: Hello world!\n", rank);

    MPI_Finalize ();
    return 0;
}
```

Programming with MPI

Messages

- information that has to be provided for the message transfer
  - rank of process sending the message
  - memory location (send buffer) of data to be transmitted
  - type of data to be transmitted
  - amount of data to be transmitted
  - rank of process receiving the message
  - memory location (receive buffer) for data to be stored
  - amount of data the receiving process is prepared to accept
- in general, message is a (consecutive) array of elements of a particular MPI data type
- data type must be specified both for sender and receiver ➔ no type conversion on heterogeneous parallel architectures (big-endian vs. little-endian, e. g.)

Programming with MPI

Messages (cont'd)

- MPI data types (1)
  - basic types (see tabular)
  - derived types built up from basic types (vector, e. g.)

MPI data type	C / C++ data type
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

Programming with MPI

Messages (cont'd)

– MPI data types (2)

MPI data type	C / C++ data type
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	represents eight binary digits
MPI_PACKED	for matching any other type

Programming with MPI

Point-to-point communication (P2P)

- different communication modes
  - *synchronous send*: completes when receive has been started
  - *buffered send*: always completes (even if receive has not been started); conforms to an asynchronous send
  - *standard send*: either buffered or unbuffered
  - *ready send*: always completes (even if receive has not been started)
  - *receive*: completes when a message has arrived
- all modes exist in both blocking and non-blocking form
  - blocking: return from routine implies completion of message passing stage
  - non-blocking: modes have to be tested (manually) for completion of message passing stage

Programming with MPI

Blocking P2P communication

- neither sender nor receiver are able to continue the program execution during the message passing stage
- sending a message (generic)

```
MPI_Send (buf, count, data type, dest, tag, comm)
```

- receiving a message

```
MPI_Recv (buf, count, data type, src, tag, comm, status)
```

- *tag*: marker to distinguish between different sorts of messages (i. e. communication context)
- *status*: sender and tag can be queried for received messages (in case of wildcard usage)

Programming with MPI

Blocking P2P communication (cont'd)

- synchronous send: `MPI_Ssend( arguments )`
  - start of data reception finishes send routine, hence, sending process is idle until receiving process catches up
  - *non-local operation*: successful completion depends on the occurrence of a matching receive
- buffered send: `MPI_Bsend( arguments )`
  - message is copied to send buffer for later transmission
  - user must attach buffer space first (`MPI_Buffer_Attach()`); size should be at least the sum of all outstanding sends
  - only one buffer can be attached per process at a time
  - buffered send guarantees to complete immediately
    ➔ *local operation*: independent from occurrence of matching receive
  - non-blocking version has no advantage over blocking version

Programming with MPI

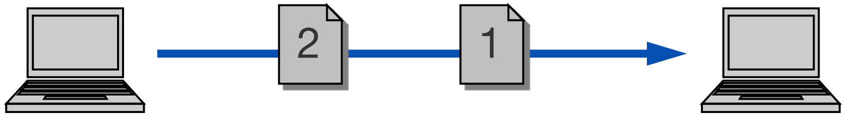
Blocking P2P communication (cont'd)

- standard send: `MPI_Send( arguments )`
  - MPI decides (depending on message size, e. g.) to send
    - *buffered*: completes immediately
    - *unbuffered*: completes when matching receive has been posted
  - completion might depend on occurrence of matching receive
- ready send: `MPI_Rsend( arguments )`
  - completes immediately
  - matching receive must have already been posted, otherwise outcome is undefined
  - performance may be improved by avoiding handshaking and buffering between sender and receiver
  - non-blocking version has no advantage over blocking version

Programming with MPI

Blocking P2P communication (cont'd)

- receive: **`MPI_Recv( arguments )`**
  - completes when message has arrived
  - usage of wildcards possible
    - MPI_ANY_SOURCE: receive from arbitrary source
    - MPI_ANY_TAG: receive with arbitrary tag
    - MPI_STATUS_IGNORE: don't care about state
- general rule: messages from one sender (to one receiver) do not overtake each other, message from different senders (to one receiver) might arrive in different order than being sent



Programming with MPI

Blocking P2P communication (cont'd)

– example: a simple ping-pong

```
int rank, buf;

MPI_Comm_rank (MPI_COMM_WORLD, &rank);

if (rank == 0) {
    MPI_Send (&rank, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Recv (&buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
              MPI_STATUS_IGNORE);
} else {
    MPI_Recv (&buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
              MPI_STATUS_IGNORE);
    MPI_Send (&rank, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```

Programming with MPI

Blocking P2P communication (cont'd)

- example: communication in a ring – does this work?

```
int rank, buf;

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

MPI_Recv (&buf, 1, MPI_INT, rank-1, 0, MPI_COMM_WORLD,
          MPI_STATUS_IGNORE);
MPI_Send (&rank, 1, MPI_INT, rank+1, 0, MPI_COMM_WORLD);

MPI_Finalize();
```

Programming with MPI

Non-blocking P2P communication

- problem: blocking communication does not return until communication has been completed ➔ risk of idly waiting and / or deadlocks
- hence, usage of non-blocking communication
- communication is separated into three phases
  1) initiate non-blocking communication
  2) do some work (involving other communications, e. g.)
  3) wait for non-blocking communication to complete
- non-blocking routines have identical arguments to blocking counterparts, except for an extra argument *request*
- request handle is important for testing if communication has been completed

Programming with MPI

Non-blocking P2P communication (cont'd)

- sending a message (generic)

```
MPI_Isend (buf, count, data type, dest, tag, comm, request)
```

- receiving a message

```
MPI_Irecv (buf, count, data type, src, tag, comm, request)
```

- communication modes
  - synchronous send: `MPI_Issend(` ***arguments*** `)`
  - buffered send: `MPI_Ibsend(` ***arguments*** `)`
  - standard send: `MPI_Isend(` ***arguments*** `)`
  - ready send: `MPI_Irsend(` ***arguments*** `)`

Programming with MPI

Non-blocking P2P communication (cont'd)

- testing communication for completion is essential before
  - making use of the transferred data
  - re-using the communication buffer
- tests for completion are available in two different types
  - *wait*: blocks until communication has been completed

```
MPI_Wait (request, status)
```

  - *test*: returns TRUE or FALSE depending whether or not communication has been completed; it does not block

```
MPI_Test (request, flag, status)
```

- what's an `MPI_Isend()` with an immediate `MPI_Wait()`

Programming with MPI

Non-blocking P2P communication (cont'd)

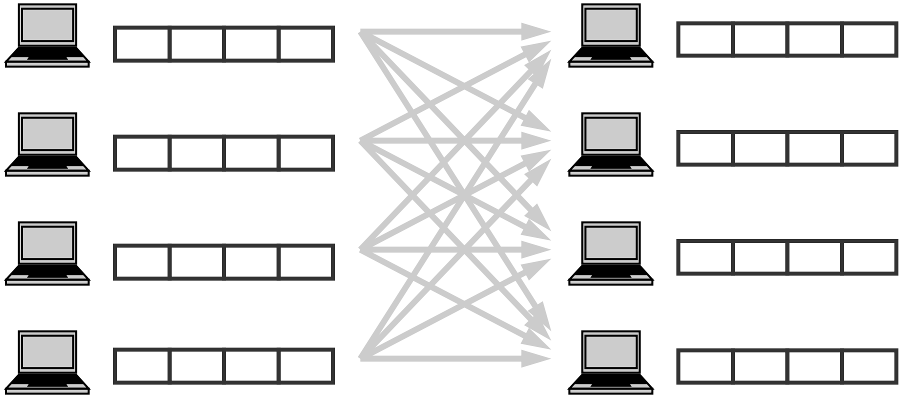
- example: communication in a ring

```
int rank, buf;
MPI_Request request;

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

MPI_Irecv (&buf, 1, MPI_INT, rank-1, 0, MPI_COMM_WORLD,
           &request);
MPI_Send (&rank, 1, MPI_INT, rank+1, 0, MPI_COMM_WORLD);
MPI_Wait (&request, MPI_STATUS_IGNORE);

MPI_Finalize ();
```

Programming with MPI

Collective communication

- characteristics
  - all processes (within communicator) communicate
  - synchronisation may or may not occur
  - all collective operations are blocking operations (non-blocking as well as some new operations since MPI 2.2)
  - no tags allowed
  - all receive buffers must be exactly of the same size



Programming with MPI

Collective communication (cont'd)

- barrier synchronisation
  - blocks calling process until all other processes have called barrier routine
  - hence, `MPI_Barrier()` always synchronises

```
MPI_Barrier (comm)
```

- broadcast
  - has a specified root process
  - every process receives one copy of the message from root
  - all processes must specify the same root

```
MPI_Bcast (buf, count, data type, root, comm)
```

Programming with MPI

Collective communication (cont'd)

- gather and scatter
  - has a specified root process
  - all processes must specify the same root
  - send and receive details must be specified as arguments

```
MPI_Gather (sbuf, scount, data type send, rbuf, rcount,
            data type recv, root, comm)

MPI_Scatter (sbuf, scount, data type send, rbuf, rcount,
             data type recv, root, comm)
```

- variants
  - `MPI_Allgather()`: all processes collect data from all others
  - `MPI_Alltoall()`: total exchange

Programming with MPI

Collective communication (cont'd)

- global reduction
  - has a specified root process
  - all processes must specify the same root
  - all processes must specify the same operation
  - reduction operations can be predefined or user-defined
  - root process ends up with an array of results

```
MPI_Reduce (sbuf, rbuf, count, data type, op, root, comm)
```

- variants (no specified root)
  - `MPI_Allreduce()`: all processes receive result
  - `MPI_Reduce_Scatter()`: resulting vector is distributed among all
  - `MPI_Scan()`: processes receive partial result (➔ parallel prefix)

Programming with MPI

Collective communication (cont'd)

- possible reduction operations (1)

operator	result
MPI_MAX	find global maximum
MPI_MIN	find global minimum
MPI_SUM	calculate global sum
MPI_PROD	calculate global product
MPI_LAND	make logical AND
MPI_BAND	make bitwise AND
MPI_LOR	make logical OR

Programming with MPI

Collective communication (cont'd)

– possible reduction operations (2)

operator	result
MPI_BOR	make bitwise OR
MPI_LXOR	make logical XOR
MPI_BXOR	make bitwise XOR
MPI_MINLOC	find global minimum and its position
MPI_MAXLOC	find global maximum and its position

Programming with MPI

Example

- finding prime numbers with the "Sieve of ERATOSTHENES[1]"
  - given: set of (integer) numbers A ranging from 2 to N
  - algorithm
    1) find minimum value a_{MIN} of A ➔ next prime number
    2) delete all multiples of a_{MIN} within A
    3) continue with step 1) until $a_{MIN} > \lfloor \sqrt{N} \rfloor$
    4) hence, A contains only prime numbers
  - parallel approach
    - distribute A among all processes (➔ data parallelism)
    - find local minimum and compute global minimum
    - delete all multiples of global minimum in parallel

[1] Greek mathematician, born 276 BC in Cyrene (in modern-day Lybia), died 194 BC in Alexandria

Programming with MPI

Example (cont'd)

```
min ← 0
A[] ← 2 … MAX

MPI_Init (&argc, &argv)
MPI_Comm_size (MPI_COMM_WORLD, &size);

divide A into size-1 parts A_i
while ( min <= sqrt(MAX) ) do

    find local minimum min_i from A_i
    MPI_Allreduce (min_i, min, MPI_MIN)
    delete all multiples of min from A_i

od

MPI_Finalize();
```

frisch@tum.de