High Performance Computing – Programming Paradigms and Scalability Part 5: Distributed-Memory Programming

PD Dr. rer. nat. habil. Ralf-Peter Mundani
Computation in Engineering (CiE)
Scientific Computing (SCCS)

Summer Term 2015
Overview

- message passing paradigm
- collective communication
- programming with MPI
- MPI advanced

At some point…

we must have faith in the intelligence of the end user.

—Anonymous
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 does not complete until message has been received (fax, e.g.)
      - **asynchronous**: send only knows when message has left; 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, \( M \leq P \), \( P \) 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, …
Message Passing Paradigm

- **message buffering**
  - message buffering decouples send and receive operations ➔ a send can complete even if a matching receive hasn’t been posted
  - buffering can be expensive
    - requires the allocation of memory for buffers
    - entails additional memory-to-memory copying

- **types of buffering**
  - *send buffer*: in general allocated by the application program or by the message passing system for temporary usage (➔ system buffer)
  - *receive buffer*: allocated by the message passing system

- problem: buffer space maybe not available on all systems
Message Passing Paradigm

- order of transmission
  - problem: there is no global time in a distributed system
  - hence, wrong send-receive assignments may occur (in case of more than two processes and the usage of wildcards)

```
<table>
<thead>
<tr>
<th></th>
<th>send to P3</th>
<th>recv buf1 from any</th>
<th>recv buf2 from any</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

or

```
<table>
<thead>
<tr>
<th></th>
<th>send to P3</th>
<th>recv buf1 from any</th>
<th>recv buf2 from any</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Overview

- message passing paradigm
- collective communication
- programming with MPI
- MPI advanced
Collective Communication

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

![Diagram showing collective communication](image)
Collective Communication

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

```
A B C D
```

```
A
B
C
D
```
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?

A B C D
E F G H
I J K L
M N O P

A E I M
B F J N
C G K O
D H L P
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

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

- reduce-scatter
  - data from all processes are reduced and distributed
  - example: any ideas?

\[
\begin{align*}
A & \rightarrow B \\
C & \rightarrow D \\
E & \rightarrow F \\
G & \rightarrow H \\
I & \rightarrow J \\
K & \rightarrow L \\
M & \rightarrow N \\
\end{align*}
\]
Collective Communication

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

```
A
B
C
D

R
(S ← A • B)
(T ← A • B • C)
(U ← A • B • C • D)
```
Collective Communication

- parallel prefix (cont’d)
  - problem: finding all (partial) results within $O(\log N)$ steps
  - implementation: two stages (up and down) using binary trees, e.g.
  - example: computing partial sums of $N$ numbers

\[
\begin{align*}
\text{ascend:} & \quad val_P &\leftarrow& \text{val}_{C, \text{left}} + \text{val}_{C, \text{right}} \\
\text{descend (level-wise):} & \quad \begin{align*}
\text{even index} & \quad \rightarrow: \quad \text{val}_C &\leftarrow& \text{val}_P \\
\text{odd index} & \quad \rightarrow: \quad \text{val}_C &\leftarrow& \text{val}_C + \text{val}_{P-1}
\end{align*}
\end{align*}
\]
Overview

- message passing paradigm
- collective communication
- programming with MPI
- MPI advanced
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
  - target platforms: SMPs, clusters, massively parallel processors
- useful links
  - http://www.mpi-forum.org
  - http://www.hlrs.de/mpi/
  - http://www-unix.mcs.anl.gov/mpi/
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

\[ P_1 \quad M \quad P_2 \quad M \quad \ldots \quad M \quad P_N \]

network
Programming with MPI

- types of communication
  - communication hierarchy (≤ MPI-2)

- MPI communication
  - point-to-point
    - blocking
      - synchr.
      - asynchr.
    - non-blocking
      - synchr.
      - asynchr.
  - collective
    - blocking
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

  \[
  \text{MPI\_Init (int *argc, char ***argv)}
  \]

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

  \[
  \text{MPI\_Finalize (void)}
  \]

  - \text{MPI\_Finalize()} does not cancel outstanding communications
  - \text{MPI\_Init()} and \text{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 and further: 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

$ \text{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)

node01
node02:2
lx64ia1.lrz-muenchen.de

- running an MPI program (depends on distribution)

$ \text{mpirun} \ -\text{machinefile} \ <\text{file}> \ -np \ <\text{#procs}> \ my\_prog
Programming with MPI

- writing and running MPI programs (cont’d)
  - example

```c
#include <mpi.h>

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
    - ranks of processes sending / receiving the message
    - memory location (send buffer) of data to be transmitted
    - type of data to be transmitted
    - amount of data to be transmitted
    - memory location (receive buffer) for data to be stored
  - 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 \(\rightarrow\) type conversion on heterogeneous parallel architectures done by the system (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.)

<table>
<thead>
<tr>
<th>MPI data type</th>
<th>C / C++ data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
</tbody>
</table>
## Programming with MPI

- messages (cont’d)
  - MPI data types (2)

<table>
<thead>
<tr>
<th>MPI data type</th>
<th>C / C++ data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>represents eight binary digits</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>for matching any other type</td>
</tr>
</tbody>
</table>
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 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)

\[\text{MPI\_Send (buf, count, data type, dest, tag, comm)}\]

- receiving a message

\[\text{MPI\_Recv (buf, count, data type, src, tag, comm, status)}\]

- \textit{tag}: marker to distinguish between different sorts of messages (i.e. communication context)
- \textit{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 (one buffer per process); size should be at least the sum of all outstanding sends
    - 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
  - 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

```c
int rank, buf;
MPI_Status status;

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, &status);
} else {
    MPI_Recv (&buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    MPI_Send (&rank, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```
Programming with MPI

- blocking P2P communication (cont’d)
  - example: buffered send

```c
int intsize, charsize, buffersize;
void *buffer;

MPI_Pack_size (MAX, MPI_INT, MPI_COMM_WORLD, &intsize);
MPI_Pack_size (MAX, MPI_CHAR, MPI_COMM_WORLD, &charsize);

buffersize = intsize + charsize + 2*MPI_BSEND_OVERHEAD;
buffer = (void *)malloc (buffersize*sizeof (void *));
MPI_Buffer_attach (buffer, buffersize);

if (rank == 0) {
    MPI_Bsend (msg1, MAX, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Bsend (msg2, MAX, MPI_CHAR, 2, 0, MPI_COMM_WORLD);
}
```
Programming with MPI

- blocking P2P communication (cont’d)
  - example: communication in a ring – does this work?

```c
int rank, size, buf, next, prev;
MPI_Status status;

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

next = (rank+1)%size;
prev = (rank-1+size)%size;
MPI_Recv (&buf, 1, MPI_INT, prev, 0, MPI_COMM_WORLD, &status);
MPI_Send (&rank, 1, MPI_INT, next, 0, MPI_COMM_WORLD);

MPI_Finalize();
```
Programming with MPI

- non-blocking P2P communication
  - problem: blocking communication does not return until it 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 \textit{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)
    
    \[
    \text{MPI\_Isend (buf, count, data type, dest, tag, comm, request)}
    \]
  
  - receiving a message
    
    \[
    \text{MPI\_Irecv (buf, count, data type, src, tag, comm, request)}
    \]

- communication modes
  
  - synchronous send: \text{MPI\_Issend( arguments )}
  
  - buffered send: \text{MPI\_Ibsend( arguments )}
  
  - standard send: \text{MPI\_Isend( arguments )}
  
  - ready send: \text{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
    - \textit{wait}: blocks until communication has been completed
    - \textit{test}: returns TRUE or FALSE depending whether or not
      communication has been completed; it does not block

\begin{verbatim}
MPI_Wait \langle request, status \rangle
\end{verbatim}

\begin{verbatim}
MPI_Test \langle request, flag, status \rangle
\end{verbatim}

- what’s an \texttt{MPI_Isend()} with an immediate \texttt{MPI_Wait()}
Programming with MPI

- non-blocking P2P communication (cont’d)
  - waiting / testing for completion of multiple communications

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Waitall()</td>
<td>blocks until all have been completed</td>
</tr>
<tr>
<td>MPI_Testall()</td>
<td>TRUE if all, otherwise FALSE</td>
</tr>
<tr>
<td>MPI_Waitany()</td>
<td>blocks until one or more have been completed, returns (arbitrary) index</td>
</tr>
<tr>
<td>MPI_Testany()</td>
<td>returns flag and (arbitrary) index</td>
</tr>
<tr>
<td>MPI_Waitsome()</td>
<td>blocks until one or more have been completed, returns index of all completed ones</td>
</tr>
<tr>
<td>MPITestsome()</td>
<td>returns flag and index of all completed ones</td>
</tr>
</tbody>
</table>

- blocking and non-blocking forms can be combined
Programming with MPI

- non-blocking P2P communication (cont’d)
  - example: communication in a ring

```c
int rank, size, buf, next, prev;
MPI_Request request;

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

next = (rank+1)%size;
prev = (rank-1+size)%size;
MPI_Irecv (&buf, 1, MPI_INT, prev, 0, MPI_COMM_WORLD, &request);
MPI_Send (&rank, 1, MPI_INT, next, 0, MPI_COMM_WORLD);
MPI_Wait (&request, MPI_STATUS_IGNORE);

MPI_Finalize ();
```
Programming with MPI

- collective communication
  - characteristics (< MPI-3)
    - all processes (within communicator) communicate
    - synchronisation may or may not occur
    - all collective operations are blocking operations
    - 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

  \[ \text{MPI\_Gather} \left(sbuf, scount, \text{data type send}, rbuf, rcount, \text{data type recv}, root, \text{comm}\right) \]

  \[ \text{MPI\_Scatter} \left(sbuf, scount, \text{data type send}, rbuf, rcount, \text{data type recv}, root, \text{comm}\right) \]

- variants
  - \text{MPI\_Allgather()}: all processes collect data from all others
  - \text{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

\[
\text{MPI\_Reduce (sbuf, rbuf, count, data type, op, root, comm)}
\]

- variants (no specified root)
  - \text{MPI\_Allreduce()}: all processes receive result
  - \text{MPI\_Reduce\_Scatter()}: resulting vector is distributed to all
  - \text{MPI\_Scan()}: processes receive partial result (parallel prefix)
Programming with MPI

- collective communication (cont’d)
  - possible reduction operations (1)

<table>
<thead>
<tr>
<th>operator</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>find global maximum</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>find global minimum</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>calculate global sum</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>calculate global product</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>make logical AND</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>make bitwise AND</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>make logical OR</td>
</tr>
</tbody>
</table>
Programming with MPI

- collective communication (cont’d)
  - possible reduction operations (2)

<table>
<thead>
<tr>
<th>operator</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_BOR</td>
<td>make bitwise OR</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>make logical XOR</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>make bitwise XOR</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>find global minimum and its position</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>find global maximum and its position</td>
</tr>
</tbody>
</table>
Programming with MPI

- example
  - finding prime numbers with the “Sieve of ERATOSTHENES” (*)
    - given: set of (integer) numbers $A$ ranging from 2 to $N$
      1) find minimum value $a_{MIN}$ of $A$ $\rightarrow$ next prime number
      2) delete all multiples of $a_{MIN}$ within $A$
      3) continue with step 1) until $a_{MIN} > \left\lfloor \sqrt{N} \right\rfloor$
      4) hence, $A$ contains only prime numbers

- parallel approach
  - distribute $A$ among all processes ($\rightarrow$ data parallelism)
  - find local minimum and compute global minimum
  - delete all multiples of global minimum in parallel

(*) Greek mathematician, born 276 BC in Cyrene (in modern-day Libya), died 194 BC in Alexandria
Programming with MPI

- example (cont’d)
  - finding prime numbers with the “Sieve of ERATOSTHENES”

```c
min ← 0
A[] ← 2 ... MAX

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

divide A into size 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();
```
Overview

- message passing paradigm
- collective communication
- programming with MPI
- MPI advanced (*)

(*) not to be covered within this lecture
MPI Advanced

- persistent communication
  - overhead through repeated communication calls (several `send()` or `receive()` within a loop, e.g.)
  - idea of re-casting the communication
  - persistent communication requests may reduce the overhead
  - freely compatible with normal point-to-point communication

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

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

- one routine for each send mode: Ssend, Bsend, Send, Rsend
- each routine returns immediately, creating a request handle
MPI Advanced

- persistent communication (cont’d)
  - request handle to execute communication as often as required

**MPI_Start** (request)

- MPI_Start() initiates respective non-blocking communication
- completion to be tested with known routines (test / wait)
- request handle must be de-allocated explicitly when finished

**MPI_Request_free** (request)

- variant: MPI_Startall() to activate multiple request
MPI Advanced

- persistent communication (cont’d)
  - example: column-wise data distribution
    - communication among direct neighbours
    - several communication stages

```c
while (...) do
  update boundary cells
  call MPI_Start() for sending updates left / right
  call MPI_Start() for receiving updates left / right
  update non-boundary cells
  wait for completion of send / receive operation
od
```

call MPI_Request_free() to de-allocate request handles
MPI Advanced

- **shift**
  - passes data among processes in a “chain-like” fashion
  - each process sends and receives a maximum of one message

- one routine for sending / receiving, i.e. atomic communication

  ```c
  MPI_Sendrecv (sbuf, scount, send data type, dest, stag,
               rbuf, rcount, recv data type, src, rtag,
               comm, status)
  ```

- hence, blocking communication, but no risk of deadlocks
- usage of MPI_NULL_PROC for more “symmetric” code
MPI Advanced

- shift (cont’d)
  - example

<table>
<thead>
<tr>
<th>process</th>
<th>source</th>
<th>destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MPI_NULL_PROC</td>
<td>MPI_NULL_PROC</td>
</tr>
<tr>
<td>2</td>
<td>MPI_NULL_PROC</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>MPI_NULL_PROC</td>
</tr>
</tbody>
</table>

- variant: \texttt{MPI\_Sendrecv\_replace()} to use same buffer for sending and receiving
MPI Advanced

- timers
  - useful routine for timing programs

```c
double MPI_Wtime (void)
```

- returns elapsed wall-clock time in seconds
- timer has no defined starting point ➔ two calls are necessary for computing difference (in general within master process)

```c
double time1, time2;

MPI_Init (&argc, &argv);

time1 = MPI_Wtime ();

MPI_Finalize ();
```
MPI Advanced

- derived data types
  - basic types only consist of (arrays of) variables of same type
  - not sufficient for sending mixed and / or non-contiguous data
  - hence, creation of derived data types

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Type_contiguous()</td>
<td>elements of same type stored in contiguous memory</td>
</tr>
<tr>
<td>MPI_Type_vector()</td>
<td>blocks of elements of same type with displacement (number of elements) between blocks</td>
</tr>
<tr>
<td>MPI_Type_hvector()</td>
<td>same as above; displacement in bytes</td>
</tr>
<tr>
<td>MPI_Type_indexed()</td>
<td>different sized blocks of elements of same type with different displacements (number of elements)</td>
</tr>
<tr>
<td>MPI_Type_hindexed()</td>
<td>same as above; displacements in bytes</td>
</tr>
<tr>
<td>MPI_Type_struct()</td>
<td>different sized blocks of elements of different type with different displacements (bytes)</td>
</tr>
</tbody>
</table>
MPI Advanced

- derived data types (cont’d)
  - derived data types are created at runtime
  - creation is done in two stages
    - construction of new data type definition from existing ones (either derived or basic)
    - commitment of new data type definition to be used in any number of communications

  **MPI**\_**Type**\_**commit** (data type)

  - complementary routine to **MPI**\_**Type**\_**commit**() for de-allocation

  **MPI**\_**Type**\_**free** (data type)
MPI Advanced

- derived data types (cont’d)
  - MPI_Type_vector()

MPI_Type_vector (count, blocklength, stride, oldtype, newtype)

oldtype:

newtype:

5 elements displacement between blocks (i.e. stride)

3 elements per block (i.e. blocklength)

2 blocks (i.e. count)
MPI Advanced

- derived data types (cont’d)
  - example: matrix A stored row-wise in memory
    - sending a row is no problem, but sending a column
    - hence, definition of new data type via `MPI_Type_vector()`

```c
MPI_Datatype newtype;
MPI_Type_vector (4, 1, 10, MPI_DOUBLE, &newtype);
MPI_Type_commit (&newtype);
MPI_Send (&(A[0][8]), 1, newtype, dest, 0, comm);
```
MPI Advanced

- virtual topologies
  - allows for a convenient process naming
  - naming scheme to fit the communication pattern
  - simplifies writing of code
  - example: communication only with nearest neighbours
    - virtual topology to reflect this fact (2D grid, e.g.)
    - hence, simplified communication based on grid coordinates

```
  (0,0)  (0,1)  (0,2)
  (1,0)  (1,1)  (1,2)
```

MPI Advanced

- virtual topologies (cont’d)
  - creating a topology produces a new communicator
  - MPI allows generation of
    - Cartesian topologies
      - each process is “connected” to its neighbours
      - boundaries can be cyclic
      - processes are identified by Cartesian coordinates
    - graph topologies
      - arbitrary connections between processes
      - see MPI document for more details
MPI Advanced

- virtual topologies (cont’d)
  - Cartesian topology

  MPI_Cart_create (old_comm, ndims, dims[], periods[],
                   reorder, cart_comm)

  - \textit{ndims}: number of dimensions
  - \textit{dims}: number of processes in each dimension
  - \textit{periods}: dimension has cyclic boundaries (TRUE or FALSE)
  - \textit{reorder}: choose dependent if data is yet distributed or not
    - FALSE: process ranks remain the same
    - TRUE: MPI may renumber (to match physical topology)
MPI Advanced

- virtual topologies (cont’d)
  - mapping functions to convert between rank and grid coordinates
  - converting given grid coordinates to process rank (returns MPI_NULL_PROC for rank if coordinates are off-grid in case of non-periodic boundaries)

  \[
  \text{MPI\_Cart\_rank}\ (\text{cart\_comm}, \text{coords}[\], \text{rank})
  \]

  - converting given process rank to grid coordinates

  \[
  \text{MPI\_Cart\_coords}\ (\text{cart\_comm}, \text{rank}, \text{ndims}, \text{coords}[\])
  \]
MPI Advanced

- virtual topologies (cont’d)
  - computing correct ranks for a shift

\[
\text{MPI\_Cart\_shift (cart\_comm, direction, disp, src, dest)}
\]

- \(\text{direction} \in [0, \text{ndims}-1]\): dimension to perform the shift
- \(\text{disp}\): displacement in that direction (positive or negative)
- returns two results
  - \(\text{src}\): rank of process from which to receive a message
  - \(\text{dest}\): rank of process to which to send a message
  - otherwise: MPI\_NULL\_PROC if coordinates are off-grid

- \text{MPI\_Cart\_shift()} does not perform the shift itself \(\Rightarrow\) to be done separately via \text{MPI\_Send()} or \text{MPI\_Sendrecv()}
MPI Advanced

- virtual topologies (cont’d)
  - example

\[\text{direction} = 0\]
\[\text{disp} = 2\]

\[\text{direction} = 1\]
\[\text{disp} = -1\]

- process calling \texttt{MPI\_Cart\_shift()}
- source
- destination
case study
- task: two-dimensional smoothing of grayscale pictures
- pictures stored as (quadratic) matrix $P$ of type integer
- elements $p(i, j) \in [0, 255]$ of $P$ stored row-wise in memory
- linear smoothing of each pixel (i.e. matrix element) via

$$p(i, j) = \frac{1}{5}(p(i+1, j) + p(i-1, j) + p(i, j+1) + p(i, j-1) - 4\cdot p(i, j))$$

- several smoothing stages to be applied on $P$
MPI Advanced

- case study (cont’d)
  - data parallelism → domain decomposition, i.e. subdivision of $P$ into equal parts (stripes, blocks, …)
  - hence, processes organised via virtual Cartesian topology (grid)
  - boundary values of direct neighbours needed by each process for its local computations (simplified data exchange via shifts)

MPI_Cart_create()

- $P_1$, $P_4$, $P_7$
- $P_2$, $P_5$, $P_8$
- $P_3$, $P_6$, $P_9$

- $0x0$, $(0,0)$, $(1,0)$, $(2,0)$
- $0x1$, $(0,1)$, $(1,1)$, $(2,1)$
- $0x2$, $(0,2)$, $(1,2)$, $(2,2)$
MPI Advanced

- case study (cont’d)
  - communication
    - exchange of updated boundaries with neighbours in each iteration $\Rightarrow$ `MPI_Cart_shift()` and `MPI_Sendrecv()` due to virtual topology (2D grid)
    - usage of `MPI_NULL_PROC` for nodes at the borders of domain
    - problem for vertical boundaries (data stored row-wise in memory) $\Rightarrow$ definition of derived data type (vector)

```
MPI_Type_vector()
MPI_Type_commit()
```
MPI Advanced

- case study (cont’d)

  MPI_Comm_rank ();
  MPI_Comm_size ();

  MPI_Cart_Create ();
  distribute data among processes (MPI_Scatter, e.g.)

  MPI_Type_vector ();
  MPI_Type_commit ();

  while (condition) do
      compute new values for interior p(i,j) in subdomain
      exchange interface values with all neighbours
      MPI_Cart_shift ();
      MPI_Sendrecv ();
      update interface values
  od

  gather data and assemble result