GWDG – Kurs
Parallel Programming with MPI

The Message Passing Interface (MPI): An Introduction

Oswald Haan
ohaan@gwdg.de

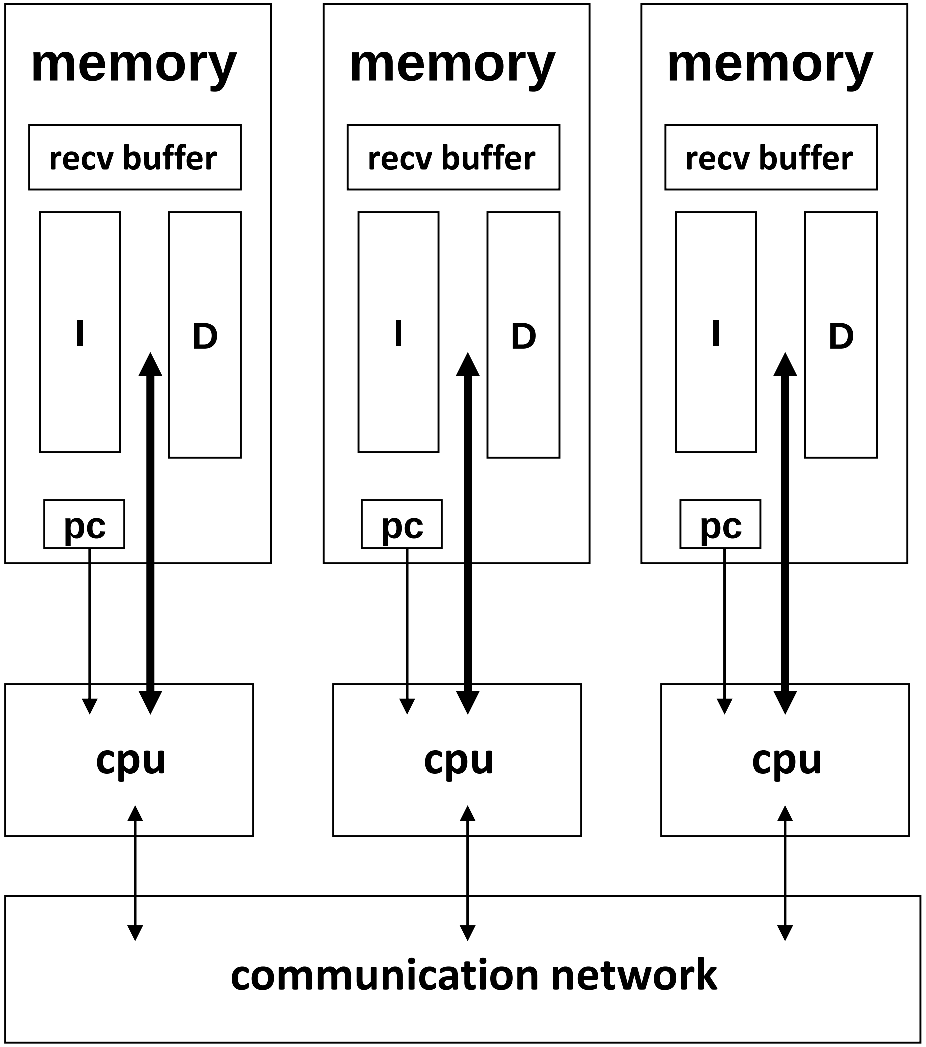
Programming Model: Message Passing

Multiple processors connected
to a communication network

objects:
local data + instructions,
local program counters (pc)
unique task identification (tid)

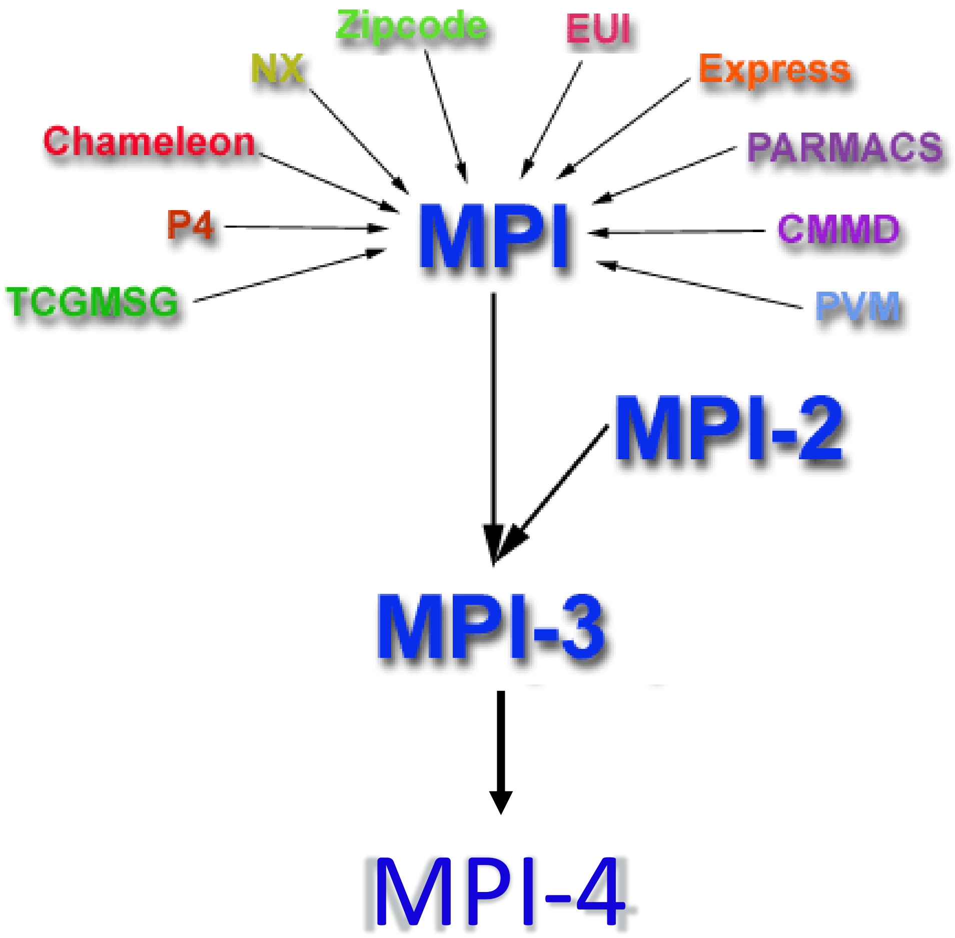
operations:
opcode (op1, op2,...,re1,re2)
send(ad,n,tid), recv(ad,n,tid)

synchronization:
recv is blocking



Programming Interfaces for Message Passing

1980-1990: various hardware and software specific solutions



1994

1996-2009

2012

2021

MPI: Message passing interface

MPI : message passing library interface specification

message passing:

MPI is a realization of the Message Passing Programming Model
(data exchanged between two processes)

With extensions
(collective operations, one-sided communication, process control, parallel I/O, …)

library :

MPI operations are invoked by calls to routines from a library

interface specification:

MPI specifies the calling sequences and the intended results of routine calls in a language independent manner, as well as the binding of this specification to Fortran and C. Bindings to other languages exist.

no implementation:

MPI provides no implementation of the interface specification.
Open source and vendor implementations exist.

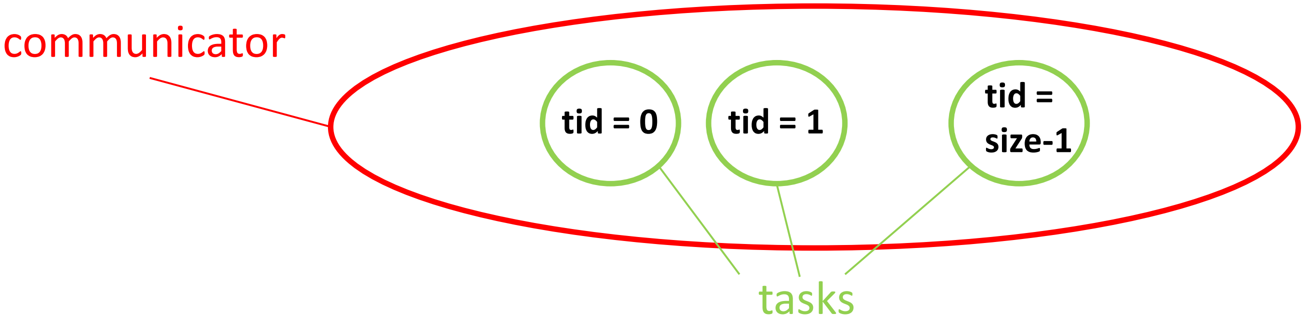
The MPI setup for Parallel Processing

An MPI program consists of autonomous processes (tasks), executing their own code, in an MIMD style.
Each process executes in its own address space.
The processes communicate via calls to MPI communication primitives.

A ***communicator*** is a collection of MPI tasks communicating with each other.

The ***size*** of a communicator is its number of tasks.

The ***rank*** of a task within a communicator is its unique identification number (tid) between 0 and size-1.



Every task in an MPI program belongs to one or more communicators and is aware of the communicators it belongs to, of their sizes and of its own tid relative to each communicator.

MPI for SPMD Parallel Programs

A special case of the MIMD programming model arises if every process in a parallel run executes the same program:

SPMD = Single Program Multiple Data

Instructions and Data for each process can be chosen individually from the common program, guided by its unique process identification number.

With no loss of generality the MPI program will be of SPMD type, i.e. there is a **single program** , and **every process** in the MPI application will execute this program in its **own execution context**.

MPI Components

Predefined data types and constants in header files (C and Fortran 77),
in modules (Fortran90/95/03/08):

Routines for

environmental management:	MPI-1
point to point communications:	MPI-1
collective communications:	MPI-1
derived data types	MPI-1
Communicator	MPI-1
Process topologies	MPI-1
Dynamic process management	MPI-2
One sided communication	MPI-2
Parallel file I/O	MPI-2
Shared memory windows	MPI-3
Partitioned communications	MPI-4
Big counts	MPI-4

MPI-Dokumentation

Official documentation of the standards(MPI-3.1) by MPI-Forum:
https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf
https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

Open MPI documentation (man pages)
https://www.open-mpi.org/doc/current/

MPI- The Complete Reference (Only versions 1, 1.1)
http://www.cslab.ntua.gr/courses/common/mpi-book.pdf

Parallel Programming for Science and Engineering
Using MPI, OpenMP, and the PETSc library
https://web.corral.tacc.utexas.edu/CompEdu/pdf/pcse/EijkhoutParallelProgramming.pdf

Parallel Programming with MPI
https://www.cs.usfca.edu/~peter/ppmpi/

Using MPI
https://wgropp.cs.illinois.edu/usingmpiweb

MPI-Forum

The MPI-Forum is an international group of hardware manufacturers, software developers and users, which started the standardization and steers the further development of the MPI standard.

http://www.mpi-forum.org/

MPI Implementations

The MPI standard is a specification of calling sequences and intended results of routines in a language-independent manner, as well as the binding of this specification to Fortran and C.

MPI implementions provide :

- Translation of the MPI program into machine specific code for using the underlying hardware of cpus and communication system efficiently.
- Interface for starting, controlling and stopping of multiple processes on the physical processors.

Examples of implementions (available at GWDG)

OpenMPI:	OpenSource implemention of MPI-3.1
Intel-mpi:	Vendor implemention of MPI-3.1

MPI : Pros and Cons

- MPI programs run on **all parallel systems**:
  from multicore desktop to largest supercomputer
- Separation of specification and implementation:
  - A standard for the library interface provides **portability**
  - Hardware specific implementations allow to **optimize performance** using the underlying hardware properties
- Application oriented routines **simplify programming** und allow **efficient implementations**

- Parallel processing with Message Passing requires **distribution and exchange of data** in addition to distribution of work.
- Support of a **large number of library routines** for special use cases (more than 250 MPI routines: https://www.open-mpi.org/doc/current/)

MPI beyond Fortran and C

Other programming languages, for which bindings to the MPI functionality exist:

- C++
- Python
- R
- Matlab
- Java
- Scala
- Haskell
- OpenGL
- Ada
- Caml
- Lisp
- C#

Available on GWDG-clusters

mpi4py: open source python bindings for MPI

Documentation for mpi4py

Manual
https://mpi4py.readthedocs.io/en/stable

Application Programming Interface (API) Reference
https://mpi4py.github.io/apiref

A Python Introduction to Parallel Programming with MPI
https://materials.jeremybejarano.com/MPIwithPython

Article **Mpi4py** in GWDG‘s web site
https://info.gwdg.de/wiki/doku.php?id=wiki:hpc:mpi4py

MPI Header Files and Modules

C:

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

Fortran77:

```
        include ´mpif.h´
```

Fortran90/95/03

```
        use mpi or include ´mpif.h´
```

Fortran 2008

```
        use mpi_f08
```

Header files contain declarations (and in some cases initializations) of MPI specific data types and objects. The most important object is:

```
        MPI_COMM_WORLD
```

MPI_COMM_WORLD is the name of the communicator object which is created, when an MPI program is launched and consists of all processes, which begin executing the MPI program.

MPI Handle

Handles identify special types of MPI objects.
Handles refer to internal MPI data structures.

For the programmer, handles are

- predefined constants in header files mpi.h , mpif.h and in modules mpi, mpi_f08
  - Example: MPI_COMM_WORLD
  - Can be used in initialization expressions or assignments.
  - The object accessed by the predefined constant handle exists and does not change between **MPI_Init** and **MPI_Finalize.**
- values returned by some MPI routines,

to be stored in variables, that are defined as

- in Fortran:
  - INTEGER
- in Fortran 2008:
  - Special types, e.g. TYPE(MPI_Comm)
- in C:
  - special MPI typedefs, e.g., MPI_Comm

Language-Independent Specification

- Example: MPI_RECV for receiving data:

MPI_RECV (buf, count, datatype, source, tag, comm, status)

OUT	buf	initial address of receive buffer (choice)
IN	count	number of elements in receive buffer (non-negative integer)
IN	datatype	datatype of each receive buffer element (handle)
IN	source	rank of source or MPI_ANY_SOURCE (integer)
IN	tag	message tag or MPI_ANY_TAG (integer)
IN	comm	communicator (handle)
OUT	status	status object (status)

IN, OUT, INOUT: intended use of argument is input, output or both:

MPI Conventions for C and Fortran

All names for MPI routines and predefined objects begin with **`MPI_`**
therefore:

MPI_...... namespace must be reserved for MPI constants and routines,
i.e. application routines and variable names must not begin with **`MPI_`**

All MPI routines provide error codes, as return value in C, as parameter in Fortran.
Many MPI routines require as parameter a communicator object:

C: (names are case sensitive, first letter of routine name must be upper case, all following letters must be lower case)

```
int error;
error = MPI_Xxxx(parameter,...);
```

Fortran: (names are case insensitive)

```
INTEGER ERROR
CALL MPI_XXXX(parameter,...,ERROR)
```

Syntax of MPI functions

FORTRAN :

```
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM,
STATUS, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM,
                 STATUS(MPI_STATUS_SIZE),IERROR
```

C :

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

Syntax of MPI functions

mpi4py

```
obj = comm.recv(buf=None, source= ANY_SOURCE,
                          tag= ANY_TAG, status=None)

where comm:       Communicator, e.g. MPI.COMM_WORLD
      obj:        Any type python object,
      buf:        buffer or None
      source:     int or ANY_SOURCE
      tag:        int or ANY_TAG,
      status:     MPI.status() or None
```

Error handling implicit via python module   **`Exception`**

Syntax of MPI functions

mpi4py

```
comm.Recv(buf, source= ANY_SOURCE,
              tag= ANY_TAG, status=None)

where comm: Communicator, e.g. MPI.COMM_WORLD
      buf:        buffer-like objects in contiguous
                  memory, e.g. numpy arrays
      source:     int or ANY_SOURCE
      tag:        int or ANY_TAG,
      status:     MPI.status() or None
```

Error handling implicit via python module **`Exception`**

MPI program: general outline

Fortran

```
program main
include ´mpif.h´
integer ierror

   ...
call MPI_INIT(ierror)
   ...
   invoking MPI routines
   ...
call MPI_FINALIZE(ierror)
   ...
end
```

C

```
#include "mpi.h"
int main(int argc,
         char **argv)
{
   ...
MPI_Init(&argc, &argv);
   ...
 invoking MPI routines
   ...
MPI_Finalize();
   ...
}
```

MPI program: general outline

python

```
from mpi4py import MPI


  ...
 invoking MPI routines
  ...
```

Starting a MPI-Program

The mpirun command (with implementation-dependent features) is available for all implementations.
Simplest form is:

```
mpirun -n 4 ./a.out
mpirun -n 4 python ./script.py
```

Invokes 4 processes, which execute in a SPMD-style the same executable a.out. (resp. script.py).

The MPI-2 standard specifies and recommends for all implementations a standardized startup command

```
mpiexec
```

MPI Routines for Environmental Management

MPI_INIT()

MPI_FINALIZE()

MPI_INITIALIZED(flag)

OUT flag true if MPI_INIT has been called (logical)

MPI_FINALIZED(flag)

OUT flag true if MPI_FINALIZED has been called (logical)

MPI_COMM_SIZE(comm, size)

IN comm communicator (handle)

OUT size number of processes in the group of comm (integer)

MPI_COMM_RANK(comm, rank)

IN comm communicator (handle)

OUT rank rank of the calling process in group of comm (integer)

MPI Routines for Environmental Management

MPI_GET_VERSION(version, subversion)

MPI_GET_PROCESSOR_NAME(name, len)

OUT	name	node name (string)
OUT	len	length of the result in name (integer)

MPI_WTIME()

Returns in a floating point number the seconds elapsed since some fixed time in the past

Fortran:	`double precision t; t = MPI_WTIME()`
C:	`double t = MPI_Wtime();`
Python:	`t = MPI.Wtime()`

MPI_WTICK()

Returns in a floating point number the resolution of MPI_WTIME in seconds

Simple MPI Program (Fortran)

```
program hello
implicit none
include ´mpif.h´
integer ierr, np, tid

call MPI_INIT(ier)
call MPI_COMM_SIZE(MPI_COMM_WORLD, np, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, tid, ierr)
call MPI_FINALIZE(ierr)
write(6,*)`hello`, np, tid

end
```

Simple MPI Program (C)

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

int main(int argc,char **argv)
{
  int np, tid;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&np);
  MPI_Comm_rank(MPI_COMM_WORLD,&tid);
  printf("hello %i %i \n",np,tid);
  MPI_Finalize();
  return 0;
}
```

Simple MPI Program (Python)

```
#hello.py
from mpi4py import MPI

comm = MPI.COMM_WORLD
tid = comm.Get_rank()
np = comm.Get_size()
print ("hello", np, tid)
```

Single Program Multiple Data (SPMD)

```
program spmd_example
include ´mpif.h´
integer ier, np, tid

call MPI_INIT(ier)
call MPI_COMM_SIZE(MPI_COMM_WORLD, np, ier)
call MPI_COMM_RANK(MPI_COMM_WORLD, tid, ier)
if (tid.eq.0) call sub0
if (tid.eq.1) call sub1
call MPI_FINALIZE(ier)

end
```