The Message Passing Interface (MPI)

TMA4280—Introduction to Supercomputing

NTNU, IMF

January 16. 2017

Parallelism

— Decompose the execution into several tasks according to the work to be done: Function/Task Parallelism

  Example: Multiple models

— Decomposition of the data provided or domain onto which the problem is posed: Data Parallelism

  Example: Mesh partitioning (data distribution techniques)

Implementation

Single Programme Multiple Data (SPMD)

- single programme executed by all processors simultaneously
- the same or different instructions can be executed
- several data streams can come into play

This is the most common case.

MPMD: multiple programmes can be combined to solve a problem.

Recap

Shared Memory architecture:

- physical address space for all processors: global shared address space
- different solutions: SMP/UMA, DSM/NUMA
- cache coherency issue
- access to shared data needs protection (resource locking)

Pitfalls:

- lack of scalability CPU vs Memory (memory contention)
- complexity of design safe and efficient memory access

Recap

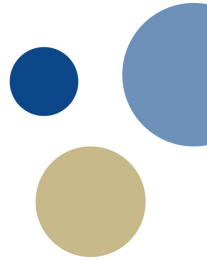
Distributed Memory Multiprocessors:

- each processor has a private physical address space
- hardware send/receive messages to communicate between processors
- scalability of memory (add nodes)

Pitfalls:

- need to implement data exchange between processors
- memory latency difference between local memory and remote memory
- fault tolerance

Shared Memory Model vs Distributed Memory Model



Requirement: Programming model to take advantage of distributed memory architectures:

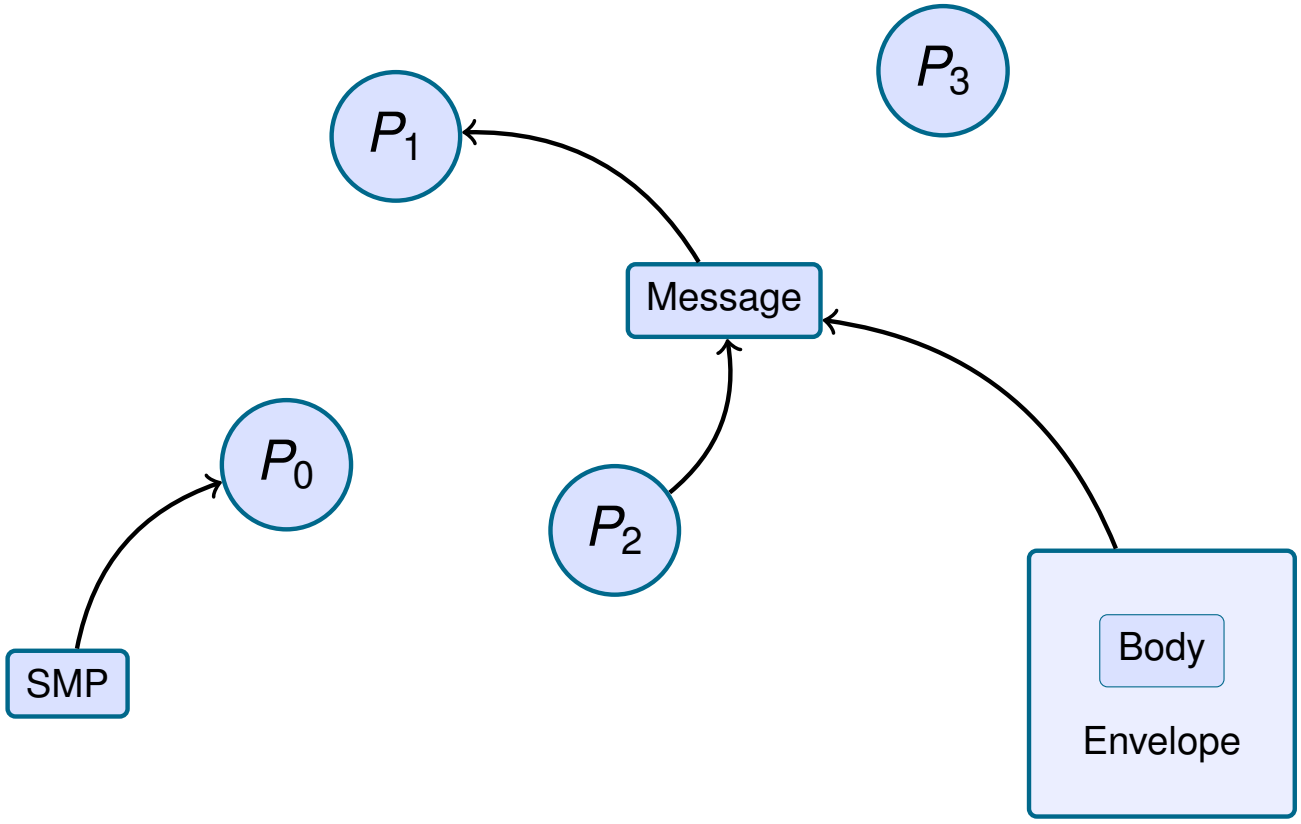
Distinguish Computer Architecture and Programming Model:
"a programming model is a view of the memory model"

Concept of Message Passing: definition of an object model to represent the execution of the programme

- Encapsulation: the layer provides general facilities that can be used without knowledge of the implementation.
- Distribution:
  1. synchronous: sender/receiver wait until data has been sent/received
  2. asynchronous: sender/receiver can proceed after sending/receiving is intiated

Remark: Nowadays both models are combined to achieve scalability

The message passing model



Basic idea of Message Passing

```
send(address, length, destination, tag)
```

`address`	memory location defining the begining of the buffer
`length`	length in bytes of the send buffer
`destination`	receiving process rank
`tag`	arbitrary identifier

```
recv(address, maxlength, source, tag, status)
```

`address`	memory location defining the begining of the buffer
`maxlength`	maxlength in bytes of received data
`destination`	sending process rank
`tag`	arbitrary identifier
`status`	metadata: actual length

MPI: Message Passing Interface

Distributed memory architecture do not offer a shared address space

- a processor cannot get a memory reference from any location in the system
- a software layer is required to allow exchange of data between processors

The Message Passing Interface (MPI) specifies how a library should provide such facilities.

MPI is a **specification**, several implementations available:

- MPICH (Argonne National Laboratory)
- OpenMPI
- Vendor specific: Cray, Intel, etc . . .

MPI: Message Passing Interface

Advantages of the MPI message-passing model:

- standardization,
- portability,
- performance,
- expressiveness.

MPI 1.0 is a specification comprising about 128 functions or operations:

- one-to-one operations (or point-to-point communication);
- one-to-all operations;
- all-to-one operations;
- all-to-all operations.

The last three types are referred to as *collective* operations.

Header file

Interface described in the header file.

For C:

```
#include "mpi.h"
```

And for Fortran:

```
include 'mpif.h'
```

MPI with C

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

int main(int argc, char **argv)
{
    int rank, size, tag, i;
    MPI_Status status;
    char message[20];

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

    tag = 100;
    if (rank == 0) {
        strcpy(message, "Hello, world!");
        for (i = 1; i < size; i++)
            MPI_Send(message, 14, MPI_CHAR, i, tag, MPI_COMM_WORLD);
    }
    else
        MPI_Recv(message, 14, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);

    printf("Process %d: %s\n", rank, message);

    MPI_Finalize();
    return 0;
}
```

MPI with Fortran

```
program hello
  include 'mpif.h'

  integer rank, size, ierror, tag
  integer status(MPI_STATUS_SIZE)
  character(12) message

  call MPI_INIT(ierror);
  call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror);
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror);

  tag = 100;

  if (rank .eq. 0) then
     message = 'Hello, world'
     do i=1, size-1
        call MPI_SEND(message, 12, MPI_CHARACTER, i, tag, MPI_COMM_WORLD, ierror)
     enddo
  else
     call MPI_RECV(message, 12, MPI_CHARACTER, 0, tag, MPI_COMM_WORLD, status, ierror)
  endif

  print *, 'process', rank, ':', message

  call MPI_Finalize(ierror)

end program hello
```

Sample output with four processors

```
Process 0: Hello, world!
Process 1: Hello, world!
Process 3: Hello, world!
Process 2: Hello, world!
```

Note that output ordering is not guaranteed.

But: MPI is non-overtaking, message processed in order.

Calling functions

For C:

```
err = MPI_Xxxxx(parameters, ...);
```

Or ignoring the error code:

```
MPI_Xxxxx(parameters, ...);
```

For Fortran, everything is a subroutine:

```
call MPI_XXXXX(parameters, ..., err)
```

MPI objects are internal and accessed via handles.

Six essential functions

C	Fortran
`MPI_Init`	`call MPI_Init`
`MPI_Comm_size`	`call MPI_Comm_size`
`MPI_Comm_rank`	`call MPI_Comm_rank`
`MPI_Send`	`call MPI_Send`
`MPI_Recv`	`call MPI_Recv`
`MPI_Finalize`	`call MPI_Finalize`

Message = data + envelope

$$\texttt{MPI\_Send}(\underbrace{\textit{buffer}, \textit{count}, \textit{datatype}}_{\textit{data}}, \underbrace{\textit{dest}, \textit{tag}, \textit{comm}}_{\textit{envelope}});$$

$$\texttt{MPI\_Recv}(\underbrace{\textit{buffer}, \textit{count}, \textit{datatype}}_{\textit{data}}, \underbrace{\textit{source}, \textit{tag}, \textit{comm}, \&\textit{status}}_{\textit{envelope}});$$

Examples of predefined data types (C):

- `MPI_CHAR`
- `MPI_INT`
- `MPI_FLOAT`
- `MPI_DOUBLE`

Message buffers

A *buffer* is an array of **contiguous** values of a given data type,

- MPI defines several Basic Types,
- data structures may not be contiguous in memory so copy is required,
- MPI allows creation of *ad hoc* data types using MPI Derived Types:

```
int MPI_Type_contiguous (
int count, // replication count
MPI_Datatype oldtype, // old datatype
MPI_Datatype * newtype ) // new datatype

int MPI_Type_vector (
int count , // number of blocks
int blocklength ,// number of elements per block
int stride , // number of elements between block
MPI_Datatype oldtype , // old datatype
MPI_Datatype * newtype ) // new datatype
```

Interprocess communication

Execution of task is performed by MPI **MPI Processes**:

- MPI Processes belong to **Groups**,
- Each process is identified with a group by its **Rank**,
- MPI defines an object **Communicator** used by collection of processes to communicate.

- `MPI_COMM_WORLD` is the main communicator *i.e* it allows communication with all processes run with `mpirun`,
- a group+communicator consisting of a subset of *all* processes may be created to:
  - simplify the implementation,
  - assign specific tasks,
  - apply operators to a subset.

  For example for Monte-Carlo simulations, N perturbed problems may be solved in parallel.

Synchronization can be done explicitely with a **Barrier** or implicitely using blocking routines.

Point-to-point communication

`MPI_Send` and `MPI_Recv` are *blocking*. Deadlock example:

```
if (rank == 0) {
  MPI_Recv(...);
  MPI_Send(...);
}
else if (rank == 1) {
  MPI_Recv(...);
  MPI_Send(...);
}
```

Deadlock: execution hang due to incorrect scheduling of messages.

Point-to-point communication

How to avoid deadlock?

```
if (rank == 0) {
  MPI_Send(...);
  MPI_Recv(...);
}
else if (rank == 1) {
  MPI_Recv(...);
  MPI_Send(...);
}
```

Pairwise `MPI_Sendrecv` is usually preferred and may be optimized by vendors.

Communication modes

MPI may provide several versions of a routine: different communication modes

- Synchronous: send blocks until handshake is done and matching receive has started.
- Ready: send blocks until matching receive has started (no handshake).
- Buffered: send returns as soon as data is buffered internally.
- Standard: Synchronous or Buffered depending on message size and resources.

Assuming synchronous mode is the safest to avoid hiding deadlocks.

Collective operations

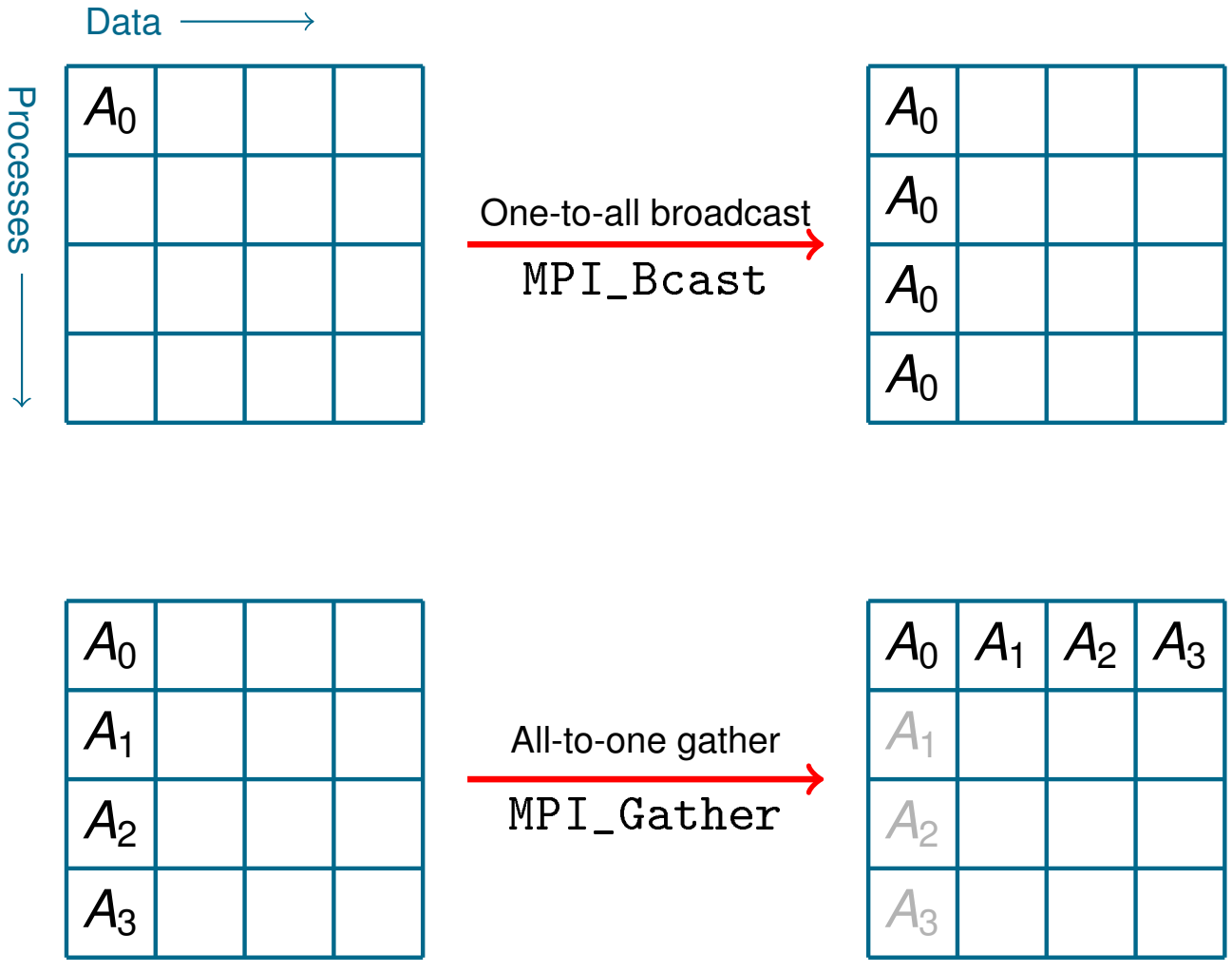
In general,

- Involves all of the processes in a group, and
- are more efficient and less tedious to use compared to point-to-point communication.

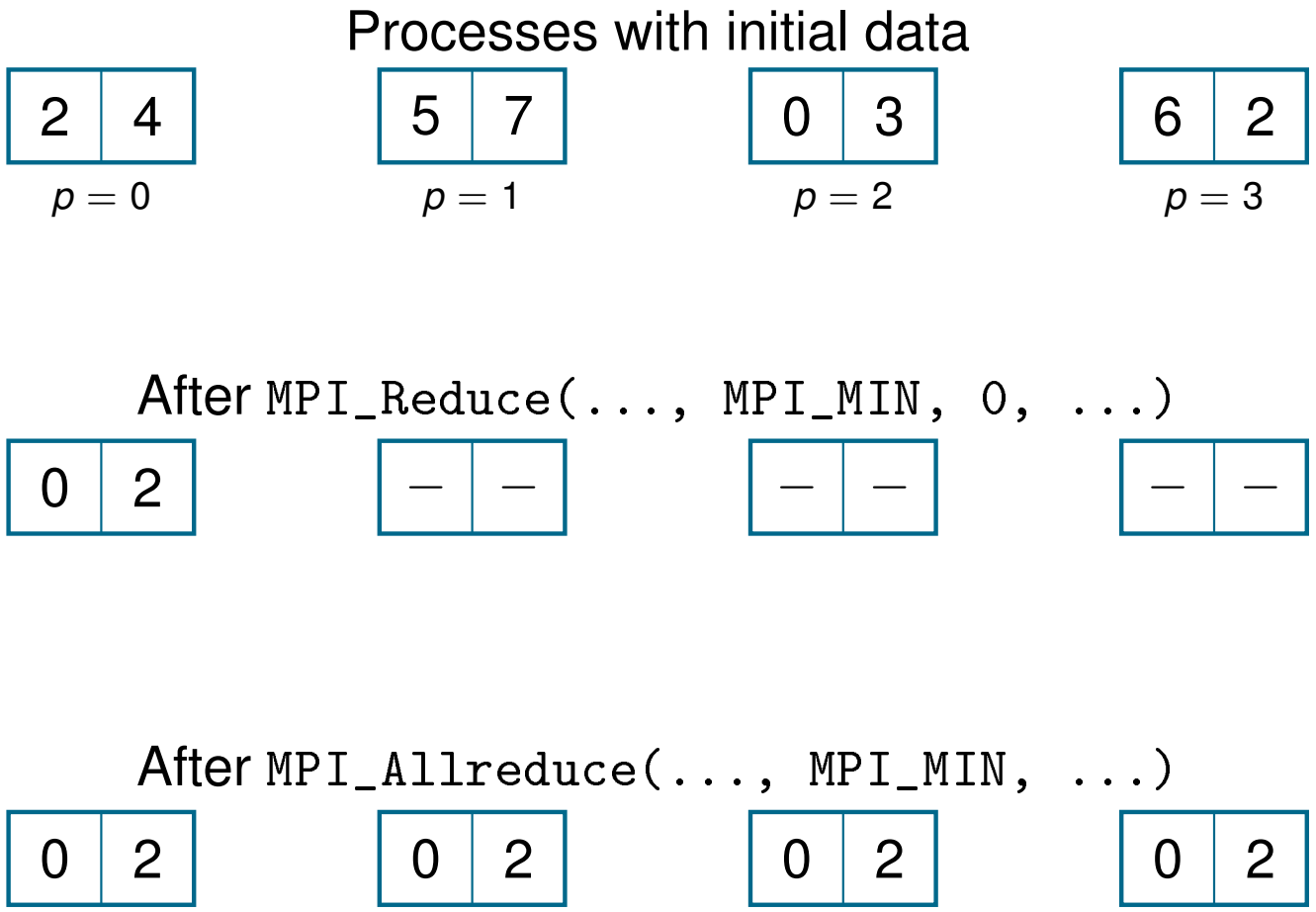
An example (synchronization between processes):

```
MPI_Barrier(comm);
```

Collective operations



Global reduction



Global reduction

`MPI_Reduce`(*sbuf*, *rbuf*, *count*, *datatype*, *op*, *root*, *comm*);

data *envelope*

`MPI_Allreduce`(*sbuf*, *rbuf*, *count*, *datatype*, *op*, *comm*);

data *envelope*

Examples of predefined operations (C):

- `MPI_SUM`
- `MPI_PROD`
- `MPI_MIN`
- `MPI_MAX`

Overview

- MPI provides a very expressive interface for Message Passing.
- Basic routines are defined for data exchange and operations.
- Extensions exist for parallel IO, shared memory, one-sided communication.
- Each function may come in different flavours corresponding to various communication modes.
- Ensuring proper scheduling of message to avoid deadlocks.
- Packing the data for communication may require copy or creation of MPI Derived Types to define a “view” of the data.
- Due to network latencies, communication patterns are crucial i.e profiling on a shared memory machine is not enough.