Mitglied der Helmholtz-Gemeinschaft

JÜLICH FORSCHUNGSZENTRUM

Message-Passing Interface

Selected Topics and Best Practices

July 9, 2014 | Florian Janetzko

References and Literature

[EG10] Edgar Gabriel, *Introduction to MPI IV –MPI derived datatypes*, Lecture COSC 4397 Parallel Computation, University of Houston (2010).

[IF95] I. Foster. Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering. Reading, MA: Addison-Wesley, 1995. http://www.mcs.anl.gov/~itf/dbpp/

[MJQ04] M. J. Quinn. *Parallel Programming in C with MPI and OpenMP*, New York, NY: Mc Graw Hill, 2004.

[MPI] The MPI Forum. *MPI: A Message-Passing Interface Standard*, Version 3.0 (2012). http://www.mpi-forum.org/

[RR] Rolf Rabenseifner, *Optimization of MPI Applications*, University of Stuttgart High-Performance Computing-Center Stuttgart (HLRS)

[WG99] W. Gropp, E. Lusk, A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, 2nd ed., MIT Press, Cambridge (1999).

[WG99a] W. Gropp, E. Lusk, R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*, MIT Press, Cambridge (1999).

[WG05] William Gropp, Rusty Lusk, Rob Ross, and Rajeev Thakur, *Advanced MPI: I/O and One-Sided Communication,* Presentation at the SC2005 (2005) . http://www.mcs.anl.gov/research/projects/mpi/tutorial/advmpi/sc2005-advmpi.pdf

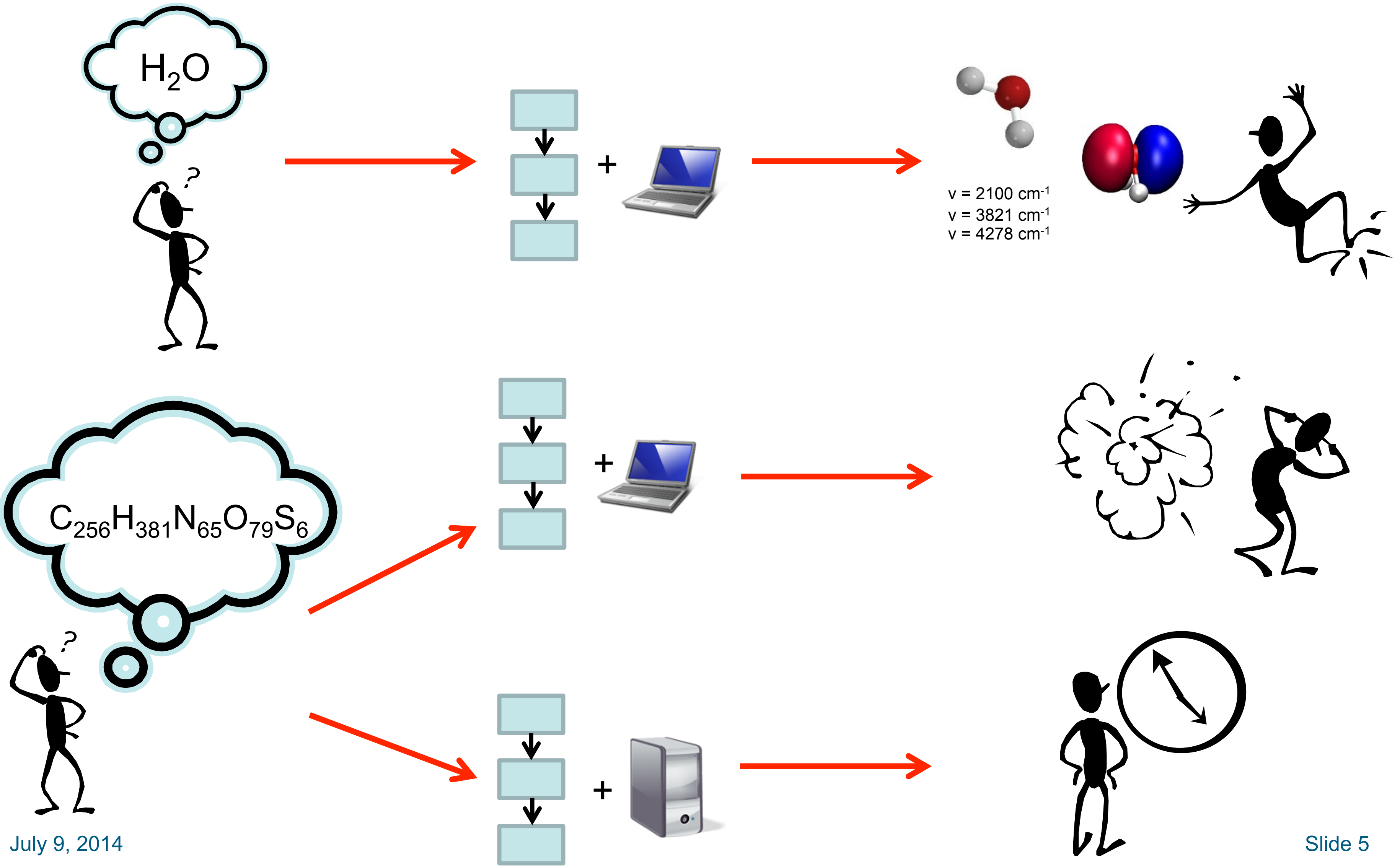
Outline

- Introduction
- Parallel Algorithms – an Example for a Design Strategy
- Message-Passing Interface – Overview
- MPI – Selected Topics and Best Practice

Outline – Introduction

- Introduction
  - Motivation – Why going Parallel?
  - Hardware – Basic Concepts
  - Software – Programming Concepts
- Parallel Algorithms – an Example for a Design Strategy
- Message-Passing Interface – Overview
- MPI – Selected Topics and Best Practice
- Summary

Motivation – Why going Parallel?



H_2O

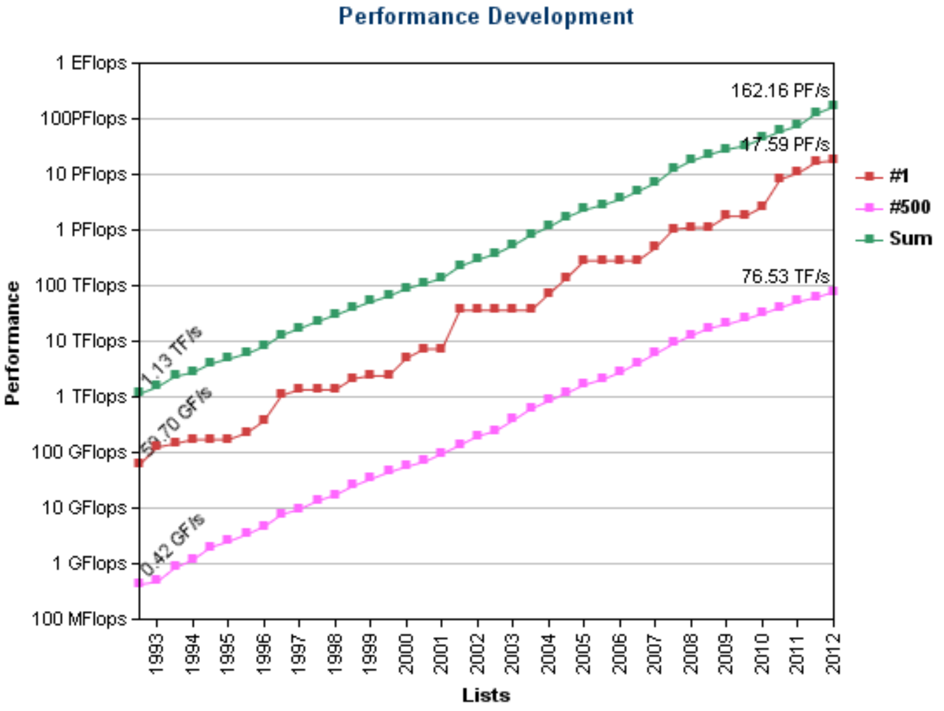
$v = 2100\ \mathrm{cm}^{-1}$
$v = 3821\ \mathrm{cm}^{-1}$
$v = 4278\ \mathrm{cm}^{-1}$

$C_{256}H_{381}N_{65}O_{79}S_6$

Motivation – Why going Parallel?

Fast growth of parallelism of HPC systems

- Multi-core
- CPUs



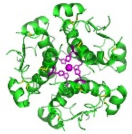
Hardware limitations

- CPU frequency
- Cooling
- Power consumption

Motivation – Why going Parallel?

Simulations requirements are increasing

- Scientific problem sizes become larger



- Better accuracy/resolution required

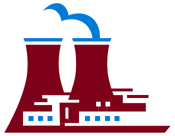
- New kinds of scientific problems arise

Hardware limitations

- CPU frequency

- Cooling

- Power consumption



➢ Parallel Computing

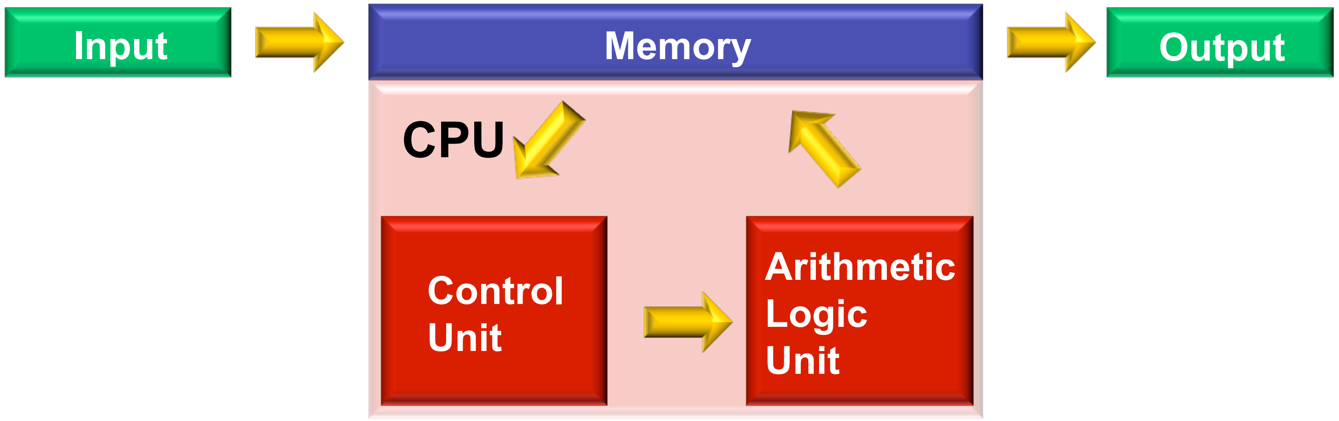
Outline – Introduction

- Introduction
  - Motivation – Why going Parallel?
  - Hardware – Basic Concepts
  - Software – Programming Concepts
- Parallel Algorithms – an Example for a Design Strategy
- Message-Passing Interface – Overview
- MPI – Selected Topics and Best Practice

Hardware Basics: von Neumann Architecture

Simulations' core requirements to hardware:

- Read data
- Perform instructions on data
- Write data/results



What to do in parallel? ➔ All (computation AND I/O)

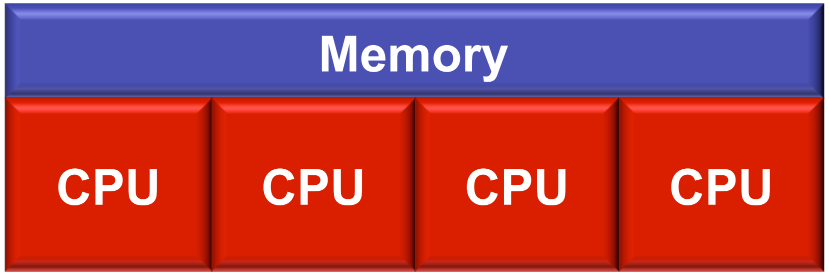
Hardware Basics: Flynn’s Taxonomy

- Classification of computer architectures

	Single Instruction	Multiple Instructions
Single Data	SISD	MISD
Multiple data	SIMD	MIMD

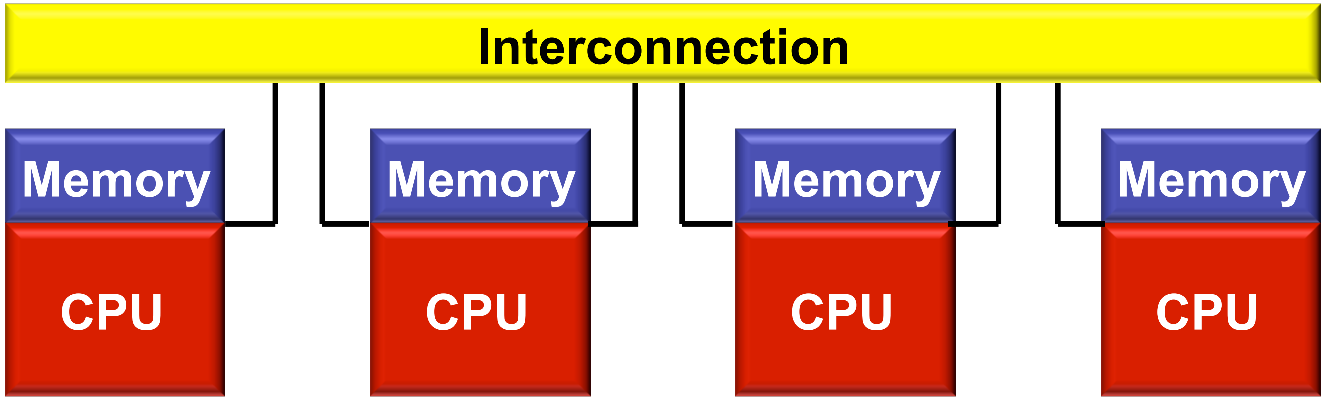
- SISD: Uniprocessor, Pentium
- SIMD: SSE instruction of x86, vector processors
- MISD: not common, used for fault tolerance, digital signal processing
- MIMD: Multiprocessor architecture

Multiprocessor Architectures: Shared Memory



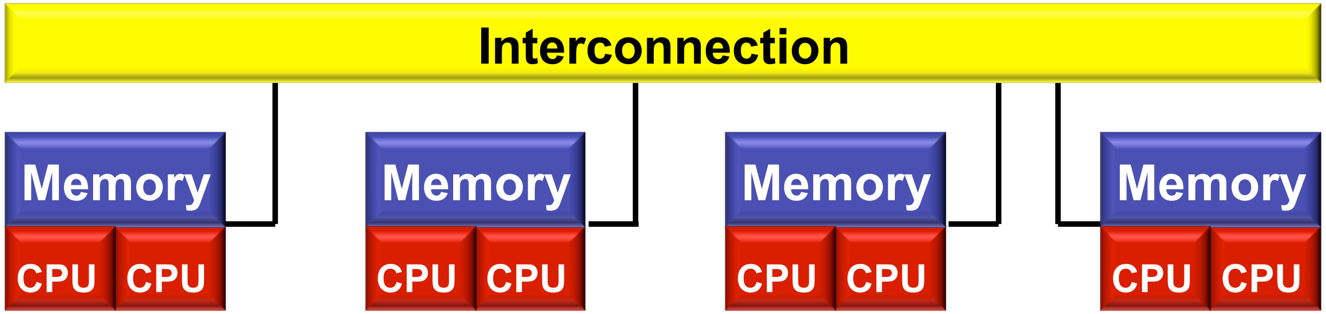
- All CPUs share the same memory
- Single address space
- Uniform memory access (UMA) multiprocessor
- Symmetric multiprocessor (SMP)

Multiprocessor Architectures: Distributed Memory



- Each CPU has its own memory and address space
- Non-uniform memory access (NUMA)
- Data exchange between memory of different CPUs
  - Via interconnect
  - Explicit data transfer necessary (message passing)

Nowadays Multiprocessor Architectures:
Hybrid Distributed-Shared Memory Architectures



- SMP with up to 16 cores
- Several SMP are combined in one *compute node (CN)*
  - Shared memory within one CN
- CN are connected via a network
  - Distributed memory between different CNs

Processes and Threads

Process

- Instance of the OS to execute a program
- Executes one or multiple → threads of execution

Thread

- Smallest unit of processing
- Sequence of instructions

Threads can be created and destroyed within a process and

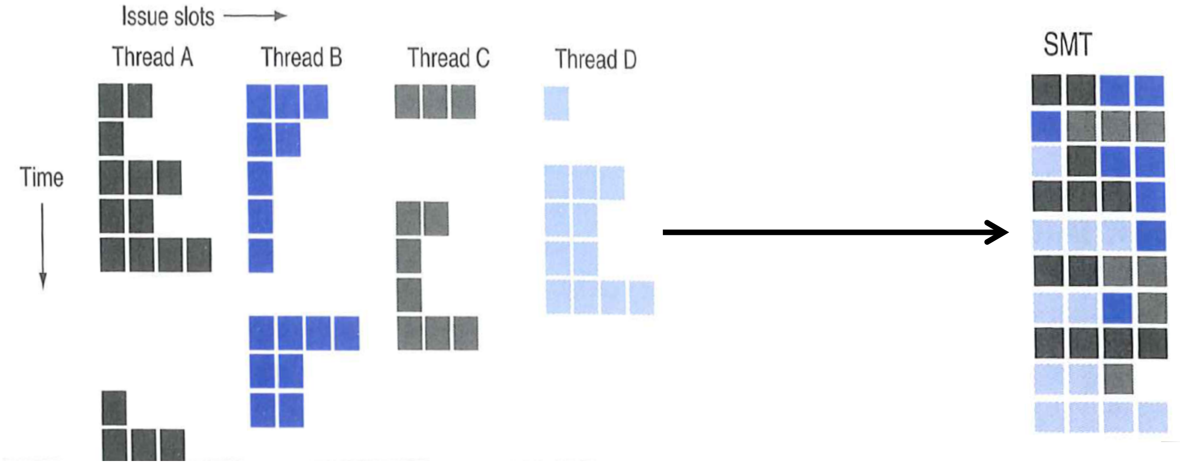
- Share the address space of the parent process (heap and static global data)
- Have a local stack

Multithreading

Multithreading:

- A process can execute several threads
- Threads can be created and destroyed at run-time
- Threads share heap and static global data but have their own stack and registers

Example: Simultaneous multithreading (SMT) (Hardware MT)



Outline – Introduction

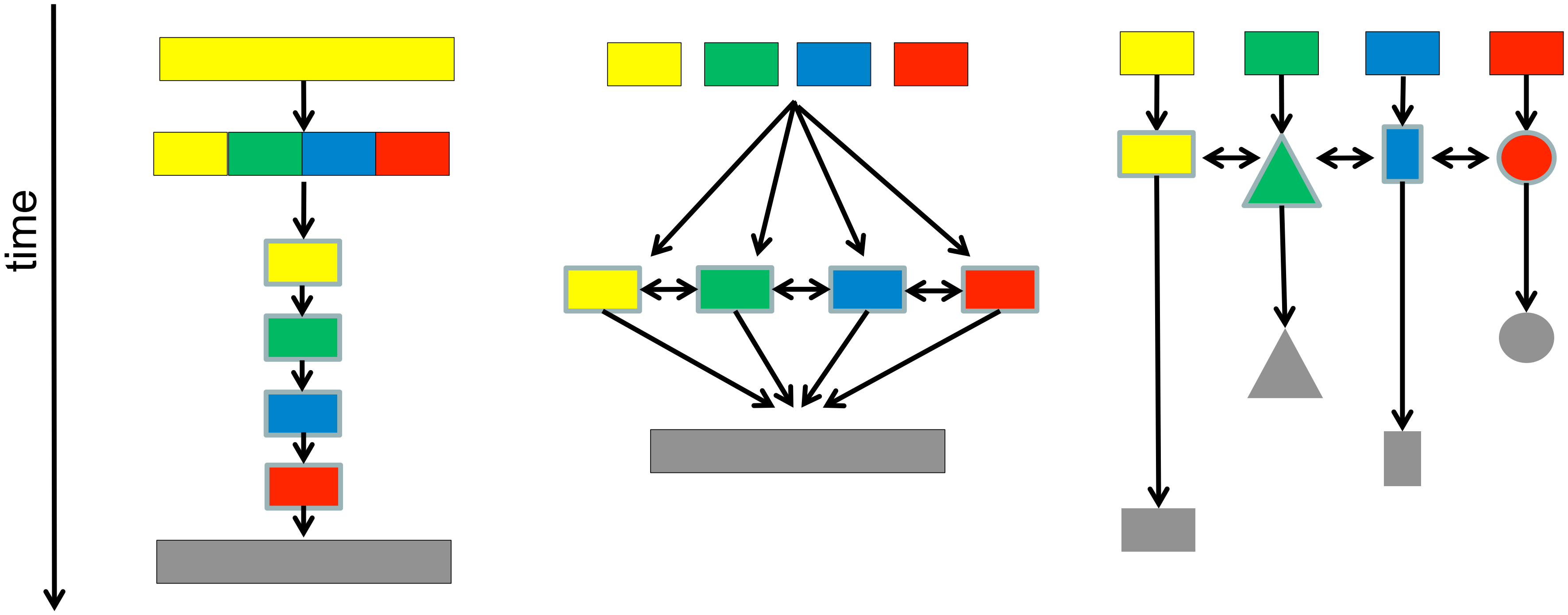
- Introduction
  - Motivation – Why going Parallel?
  - Hardware – Basic Concepts
  - Software – Programming Concepts
- Parallel Algorithms – an Example for a Design Strategy
- Message-Passing Interface – Overview
- MPI – Selected Topics and Best Practice

Software: Programming Paradigms

SPSD
(single program single data)

SPMD
(single program multiple data)

MPMD
(multiple programs multiple data)



JÜLICH
FORSCHUNGSZENTRUM

Parallel Programming Concepts

Classification according to process interaction

1. Message passing
   - Parallel processes exchange data by passing messages
   - Examples: PVM, MPI
2. Shared memory
   - Parallel threads share a global address space
   - Examples: POSIX threads, OpenMP
3. Implicit
   - Process interaction is not visible to the programmer
   - Examples: PGAS (CAF, UPC), GA

Outline – Introduction

- Introduction
- Parallel Algorithms – an Example for a Design Strategy
- Message-Passing Interface – Overview
- MPI – Selected Topics and Best Practice

Task/Channel Model

Task

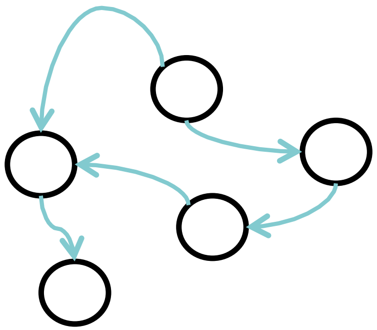
A program, its local memory, and a collection of I/O ports. Tasks can communicate with each other via → channels.

Primitive Task (ptask)

The smallest logical unit of instructions an algorithm can be split in.

Channel

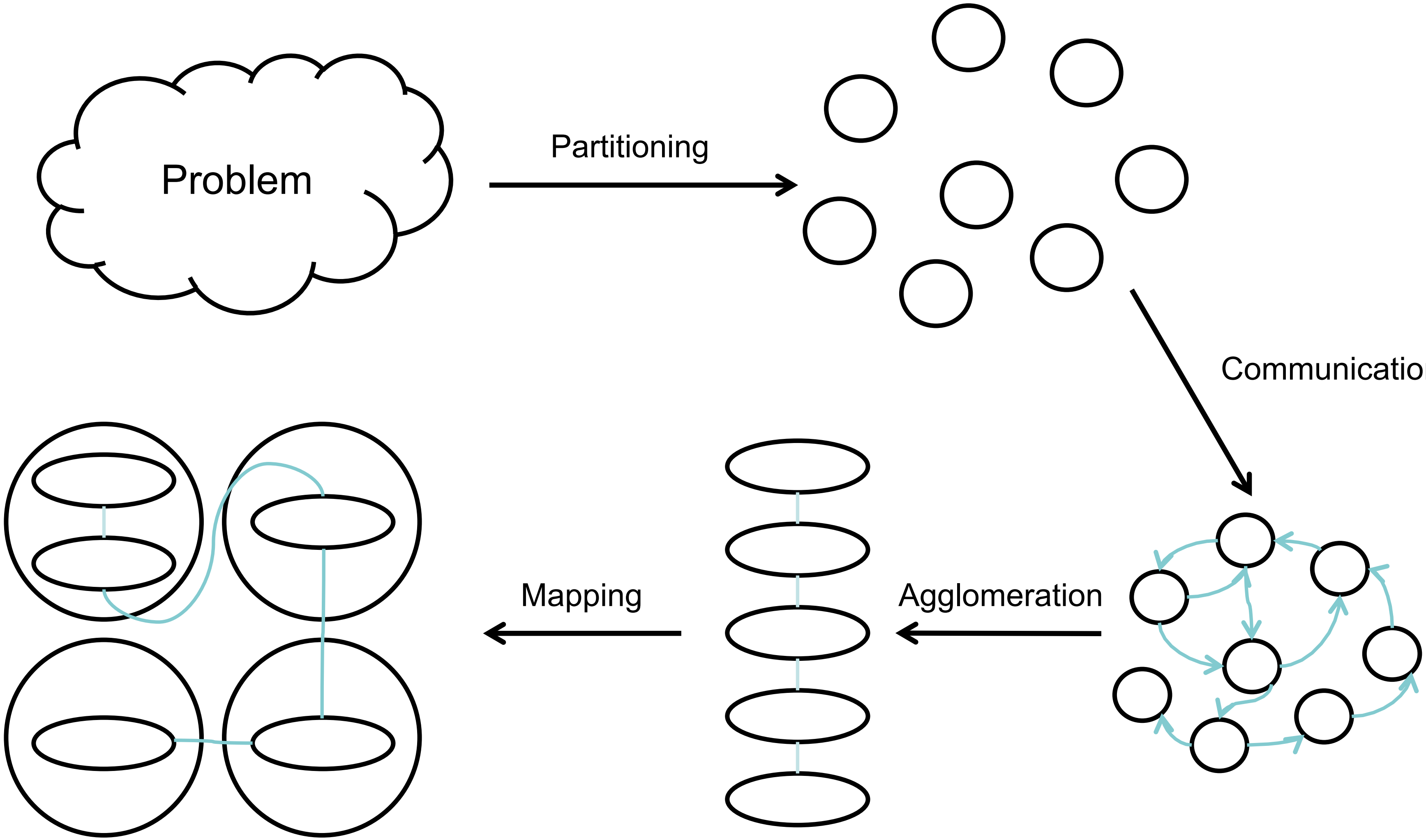
A message queue which connects the output port of one task with the input port of another task.



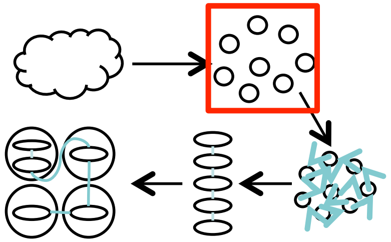
○ Primitive task (ptasks)

→ Communication channel

Foster‘s Design Methodology



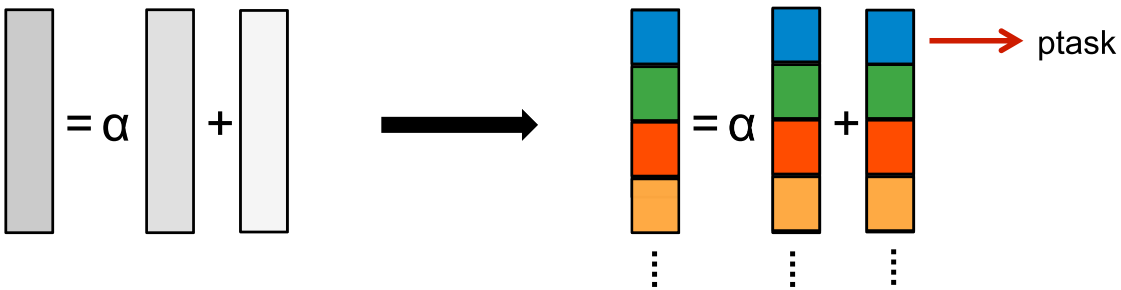
Partitioning



Splitting the problem into smaller pieces

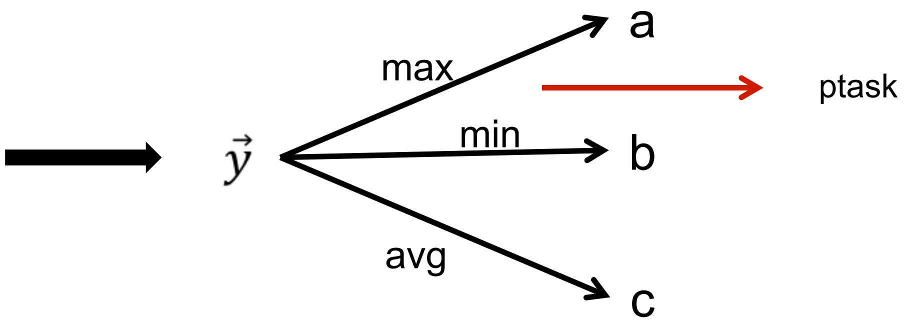
- Data-centric approach (Domain decomposition)

$$\vec{y} = \alpha\vec{y} + \vec{x}$$

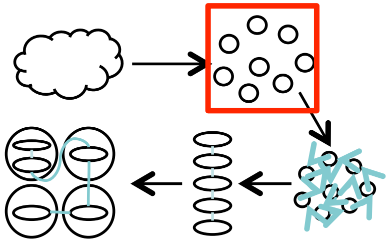


- Computation-centric approach

$$\vec{y} \xrightarrow{\max} \text{a};\ \vec{y} \xrightarrow{\min} \text{b};\ \vec{y} \xrightarrow{\text{avg}} \text{c};$$



Partitioning



Checklist for good partitioning schemes

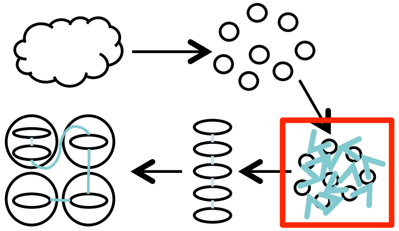
✓ The ratio tasks/number of cores should be at least 10:1

✓ Avoid redundant storage of data

✓ Try to have tasks of comparable size

✓ The number of tasks should scale with the problem size

Communication



Local Communication

- A task needs data from a small number of other tasks

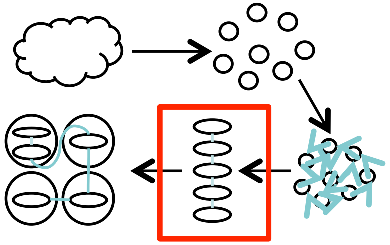
Global Communication

- A task needs data from all other tasks

Communication is part of the parallel overhead, so check

- ✓ Communication operations should be balanced among tasks
- ✓ Minimize communication
- ✓ Tasks can communicate concurrently
- ✓ Tasks can compute concurrently

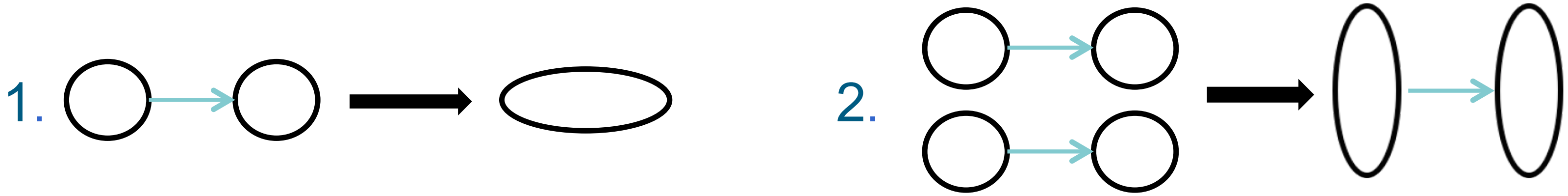
Agglomeration



Simplify the program

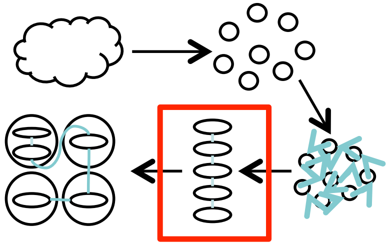
Increase locality

1. Grouping tasks together → elimination of communication
2. Grouping sending and receiving ptasks → less messages



Maintain scalability

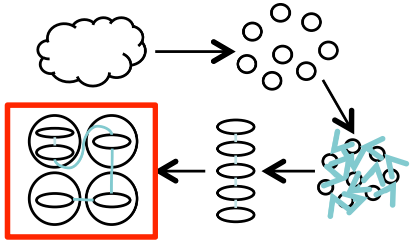
- Do not group too many tasks
- Extreme: group everything → serial code!



Agglomeration

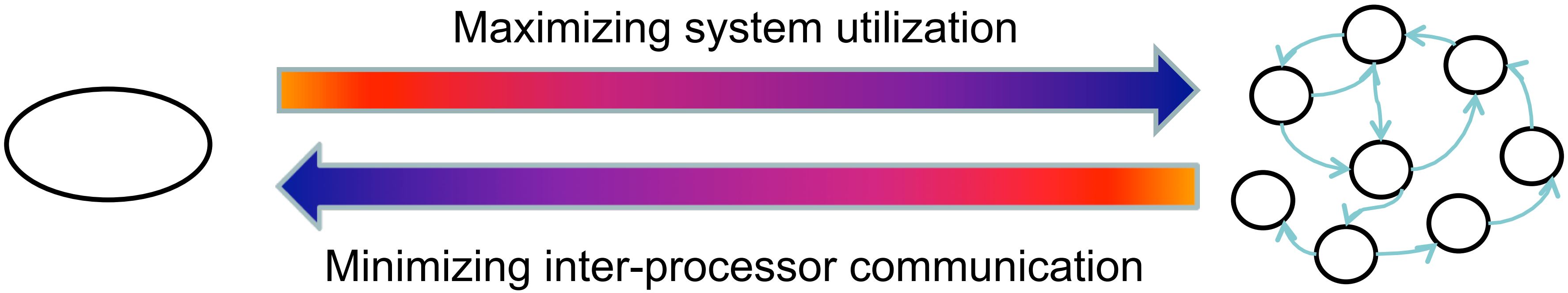
Checklist for good agglomeration scheme

- ✓ Increase locality
- ✓ Check that replicated computations use less time than the communication they replace
- ✓ Computation and communication per task is balanced
- ✓ Number of tasks is an increasing function of the problem size
- ✓ Number of tasks fits to target (HPC) architecture



Mapping

Assigning tasks to cores – sometimes conflicting goals



Maximizing system utilization

Minimizing inter-processor communication

Considerations/checklist:

- ✓ One task or multiple tasks per processor
- ✓ Ratio tasks to processors
- ✓ Static or dynamic allocation of tasks to processors
- ✓ Hybrid programming approach?

Outline

- Introduction
- Parallel Algorithms – an Example for a Design Strategy
- Message-Passing Interface – Overview
  - Introduction and History
  - Basic concepts and terms
  - General usage
- MPI – Selected Topics and Best Practices

Introduction – What is MPI?

MPI (Message-Passing Interface)

- Industry standard for a message-passing programming model
- Provides specifications
- Implemented as a library with language bindings for Fortran and C
- Portable across different computer architectures

- Purpose: provision of a means for communication between processes

Brief History

<1992 Several message-passing libraries were developed

- PVM, P4, LAM …

1992 SC92: Several developers for message-passing libraries agreed to develop a standard for message passing

1994 MPI-1 standard published

1997 Development of MPI-2 standard started

2008 MPI-2.1

2009 MPI-2.2

2012 MPI-3.0, current version of the MPI standard

Outline

- Introduction
- Parallel Algorithms – an Example for a Design Strategy
- Message-Passing Interface – Overview
  - Introduction and History
  - Basic concepts and terms
  - General usage
- MPI – Selected Topics and Best Practices

MPI Terminology – Basics

Task

An instance, sub-program or process of an MPI program

Group

An ordered set of process identifiers (henceforth: processes)

Rank

A unique number assigned to each task of an MPI program within a group **(start at 0)**

Context

A property that allows the partitioning of the communication space

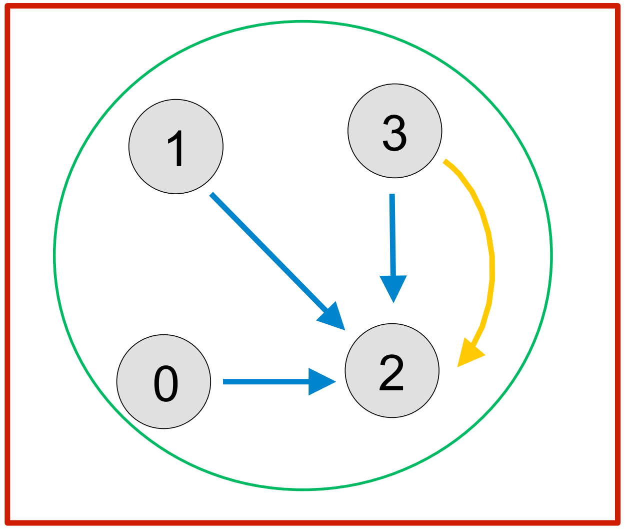
Communicator

Scope for communication operations within or between groups (intra-communicator or inter-communicator). Combines the concepts of group and context.

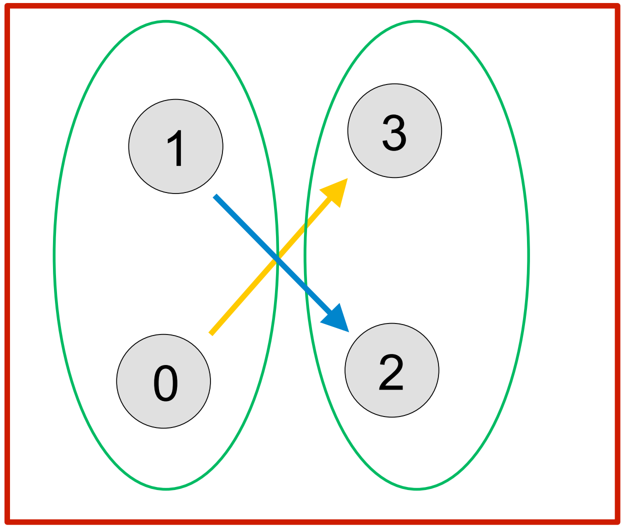
MPI Terminology – Communicators, Groups, Context

(MPI 3.0, 6)

Intra-communicator



Inter-communicator



●	Task	□	Communicator
0	Rank	→	Communication in context A
○	Group	→	Communication in context B

MPI Terminology – Data Types

Basic Data Types

Data types which are defined within the MPI standard

- Basic data types for Fortran and C are **different**
- Examples:

Fortran

Fortran type	MPI basic type
INTEGER	MPI_INTEGER
REAL	MPI_REAL
CHARACTER	MPI_CHARACTER

C/C++

C type	MPI basic type
signed int	MPI_INT
float	MPI_FLOAT
char	MPI_CHAR

Derived Data Types

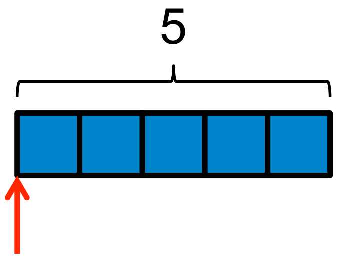
Data types which are constructed from basic (or derived) data types

MPI Terminology – Messages

Message

A packet of data which needs to be exchanged between processes

- Packet of data:
  - An array of elements of an MPI data type (basic or derived data type)
  - Described by
    - Position in memory (address)
    - Number of elements
    - MPI data type 🟦



- Information for sending and receiving messages
  - Source and destination process (ranks)
  - Source and destination location
  - Source and destination data type
  - Source and destination data size

MPI Terminology – Properties of Procedures (I)

Blocking

A procedure is blocking if return from the procedure indicates that the user is allowed to reuse resources specified in the call to the procedure.

Nonblocking

If a procedure is nonblocking it will return as soon as possible from to the calling process. However, the user is not allowed to reuse resources specified in the call to the procedure before the communication has been completed by an appropriate call at the calling process.

Examples

- Blocking

- Nonblocking

MPI terminology – Properties of procedures (II)

Collective

A procedure is collective if all processes in a group need to invoke the procedure

Synchronous

A synchronized operation will complete successfully only if the (required) matching operation has started (send – receive).

Buffered (Asynchronous)

A buffered operation may complete successfully before a (required) matching operation has started (send – receive).

(Non-)Blocking – (A)Synchronous

(Non)-Blocking

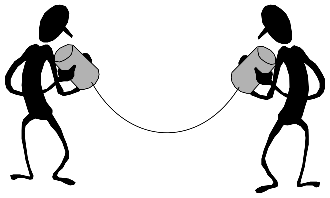
- Statement about *reusability of message buffer*

(A)Synchronous

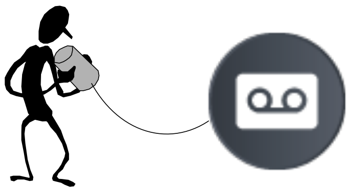
- Statement about *matching communication call*

Example

- Blocking, synchronous sending:
  ☎
  - Will return from call when buffer can be reused
  - After return receiving has started



- Blocking, asynchronous sending:
  ☎
  - Will return from call when buffer can be reused
  - After return, receiving has not started necessarily, message may be buffered internally



Outline

- Introduction
- Parallel Algorithms – an Example for a Design Strategy
- Message-Passing Interface – Overview
  - Introduction and History
  - Basic concepts and terms
  - General usage
- MPI – Selected Topics and Best Practices

The MPI infrastructure – Linking

Program must be linked against an MPI library

- Usually done using compiler wrappers

C/C++

```
mpicc myprog.c –o myprog
mpiCC myprog.cc –o myprog
```

Fortran

```
mpif90 myprog.f90 –o myprog
```

Names of these wrappers are not standardized! The prefix mpi is very common, however, other prefixes and names are possible, e.g. mpcc for the IBM XL C compiler on AIX.

The MPI infrastructure – Launching applications

Program must be started with the MPI start-up mechanism

```
mpirun [options] my_application.exe
mpiexec [options] my_application.exe
```

Names of these start-up mechanisms are not standardized! The above commands are very common, however, other mechanisms are possible, e.g. poe on AIX or runjob on Blue Gene/Q (e.g. JUQUEEN).

Language bindings

Language bindings for

- Fortran (Fortran77, Fortran90, Fortran2008 compatible)
- ISO C

Definitions included using header files

C/C++

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

Fortran

```
include 'mpif.h'   ! Fortran 77
use mpi            ! Fortran 90
use mpi_f08        ! Fortran 2008           (new in 3.0)
```

For Fortran the Fotran77/Fortran90 bindings are used throughout this talk

Nomenclature of MPI functions

Generic format of MPI functions

C/C++

```
error = MPI_Function(parameter,...);
```

Fortran

```
call MPI_FUNCTION(parameter,...,ierror)
```

Never ever forget the **ierror** parameter in Fortran calls because this may lead to unpredictable behavior!

MPI Namespace:

MPI_ and PMPI_ prefixes must not be used for user-defined functions or variables since they are used by MPI!

Example: initialization and finalization of MPI

C/C++

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

Fortran

```
MPI_INIT(IERROR)
  INTEGER :: IERROR
```

C/C++

```
int MPI_Finalize(void);
```

Fortran

```
MPI_FINALIZE(IERROR)
  INTEGER :: IERROR
```

Example: getting total number of tasks

C/C++

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

```
…
ierror = MPI_Comm_size(MPI_COMM_WORLD, &size);
…
```

Fortran

```
MPI_COMM_SIZE(COMM, SIZE, IERROR)
  INTEGER :: COMM, SIZE, IERROR
```

```
…
call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
…
```

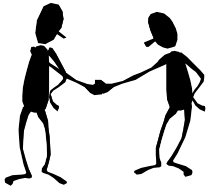
Outline

- Introduction
- Parallel Algorithms – an Example for a Design Strategy
- Message-Passing Interface – Overview
- MPI – Selected Topics and Best Practices
    - General Hints and Remarks
    - Point-to-Point Communication
    - Collective Communication
    - Derived Datatypes
    - MPI_Info Object
    - One-sided Communication

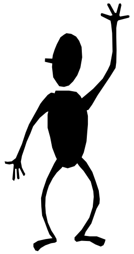
General performance considerations – I

Communication protocols

- Rendezvous protocol
  - Optimized for high bandwidth
  - Needs an initial handshake between involved tasks → high latency



- Eager protocol
  - Low latency but low bandwidth
  - Useful for many small messages



- Which protocol is used is determined by the eager limit (message size in bytes > eager limit → rendezvous)
- Check environment variables!

General performance considerations – II

Communication overhead

Transfer time = latency + message length / bandwidth

- Latency: Startup for message handling
- Bandwidth: Transfer of bytes

- For **n** messages

Transfer time = n*latency + total message length / bandwidth

- Try to avoid communication
- Send few big messages instead of many small ones
- Chose the appropriate protocol

General hints and recommendations

💡1 Avoid communication if possible (usually)

- No communication is the fastest communication

💡2 Use as few resources as possible

- Keeps small memory/communication footprint

💡3 Provide as much information to MPI as possible

- Allows MPI to choose best way of delivering messages
- Allows MPI to optimize/reorder communication

💡4 Give MPI the freedom to optimize

- Let MPI choose best way of communication

⚠ Common mistakes

Wrong API usage

- Missing **ierror** argument in Fortran
- Collective routines not called on all ranks of **com**

Wrong variable declarations

- Using **INTEGER** where **MPI_OFFSET_KIND** or **MPI_ADDRESS_KIND** is needed
- **status** variable not declared with dimension **MPI_STATUS_SIZE** (Fortran)

Nonblocking communication

- Reusing buffers before it is save to do so
- Missing MPI_Wait[…]

Outline

- Introduction
- Parallel Algorithms – an Example for a Design Strategy
- Message-Passing Interface – Overview
- MPI – Selected Topics and Best Practices
  - General Hints and Remarks
  - Point-to-Point Communication
  - Collective Communication
  - Derived Datatypes
  - MPI_Info Object
  - One-sided Communication

Point-to-point communication

- Communication between two processes
  - ! Note: A process can send messages to itself !
- A source process sends a message to a destination process by a call to an MPI send routine
- A destination process needs to post a receive by a call to an MPI receive routine
- The destination process is specified by its rank in the communicator, e.g. **MPI_COMM_WORLD**
- Every message sent with a point-to-point call, needs to be matched by a receive.

Parts of messages

1. Data part
   - Contains actual data to be sent/received
   - Needs three specifications
     1. Initial address (send/receive buffer): **BUF**
     2. Number of elements to be sent/received: **COUNT**
     3. Datatype of the elements: **DATATYPE**
2. Message envelop
   - Contains information to distinguish messages
     1. Source process: **SOURCE**
     2. Destination process: **DEST**
     3. A marker: **TAG**
     4. The context of processes: **COMM**

Mitglied der Helmholtz-Gemeinschaft

JÜLICH FORSCHUNGSZENTRUM

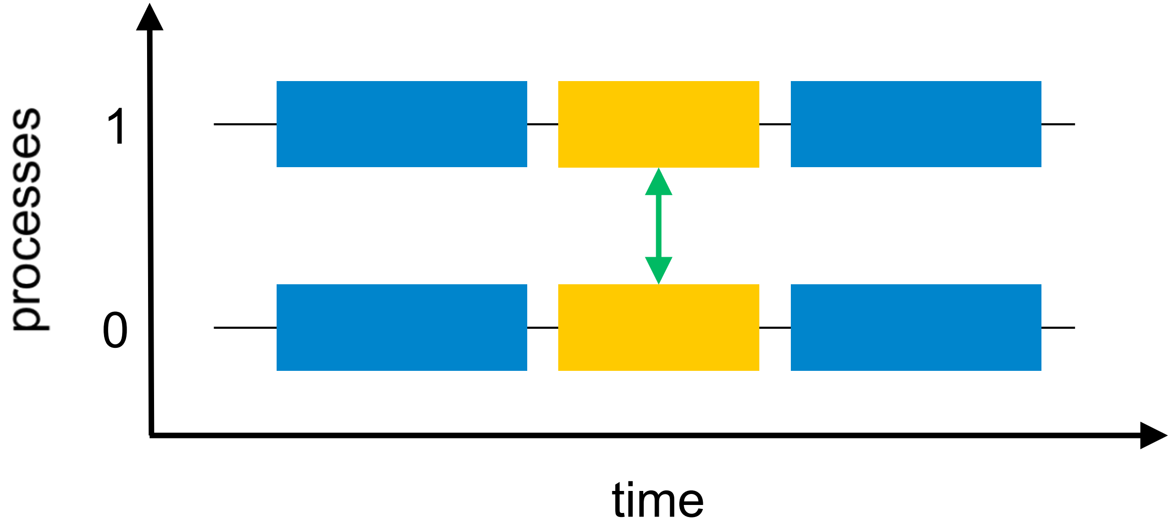
Point-to-Point Communication

Blocking communication

July 9, 2014 | Florian Janetzko

Blocking Communication

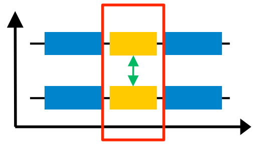
Computation interrupted by communication



🟦 Computation

🟨 Communication

Sending messages



C/C++

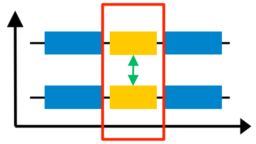
```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
```

Fortran

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
  <type>  :: BUF(*)
  INTEGER :: COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

- **BUF** is the address of the message to be sent, with **COUNT** elements of type **DATATYPE**
- **DEST** is the rank of the destination process within the communicator **COMM**
- **TAG** is a marker used to distinguish different messages

Receiving messages



C/C++

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

Fortran

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

- **BUF**, **COUNT** and **DATATYPE** refer to the receive buffer
- **SOURCE** is the rank of the sending source process within the communicator **COMM** (can be **MPI_ANY_SOURCE**)
- **TAG** is a marker used to prescribe that only a message with the specified tag should be received (can be **MPI_ANY_TAG**)
- **STATUS** (output) contains information about the received message

Send modes

Synchronous send: MPI_Ssend

- Only completes when the receive has started

Buffered send: MPI_Bsend

- Always completes (unless an error occurs) irrespective of whether a receive has been posted or not
- Needs a user-defined buffer (→ **MPI_BUFFER_ATTACH**, MPIS3.0, 3.6)

Standard send: MPI_Send

- Either synchronous or buffered
- Uses an internal buffer if buffered

Ready send: MPI_Rsend

- Always completes (unless an error occurs) irrespective of whether a receive has been posted or not
- May be started only if the matching receive is already posted

MPI_Send

Standard send: MPI_Send

- **Either** synchronous or buffered
- Uses an internal buffer if buffered

- Depends on the MPI implementation
- Do *not assume* either case:
  - It can buffer

    On the sender side

    On the receiver side
  - It can wait for the matching receive to start

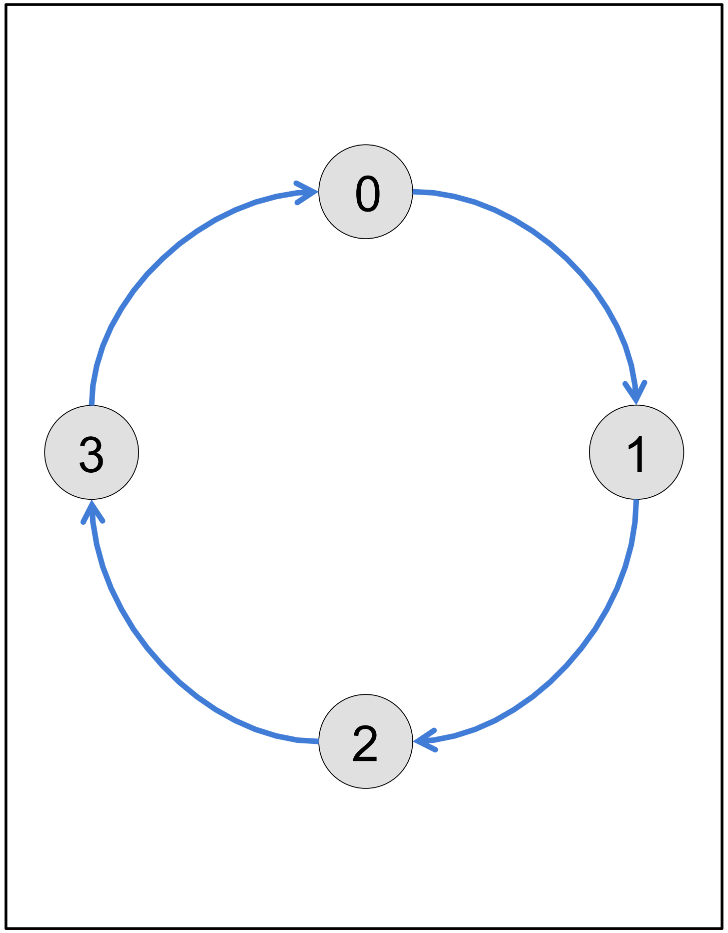
MPI_Rsend

Ready send: MPI_Rsend

- Always completes (unless an error occurs) irrespective of whether a receive has been posted or not
- May be started only if the matching receive is already posted

- User’s responsibility for writing a correct program
- Error-prone, use *only* if absolutely necessary and you *really know* what you are doing!

Pitfall 1 – Blocking point-to-point communication

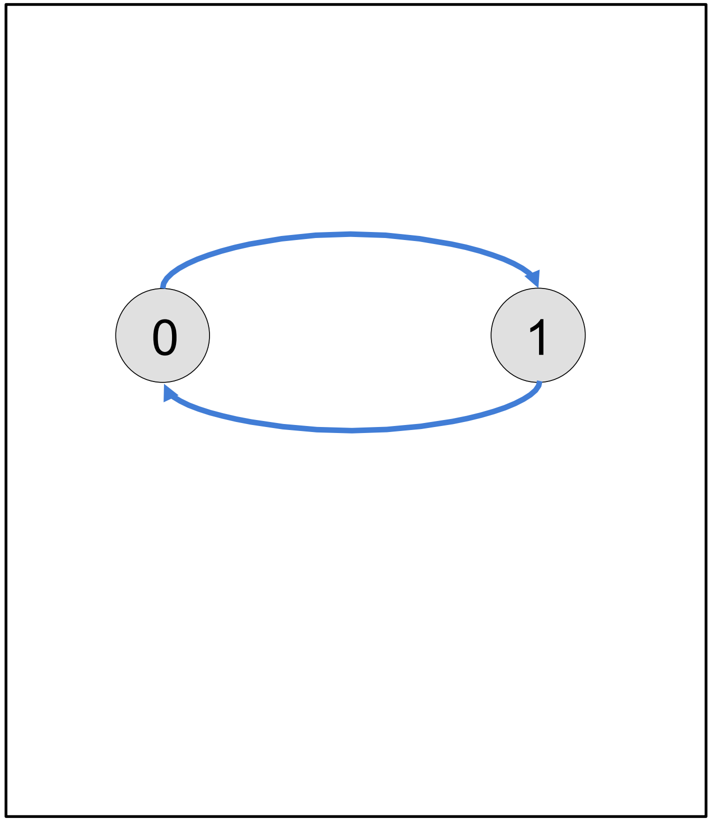


```
…
Call MPI_Ssend(…,dest=my_right_neighbor,…)
Call MPI_Recv(…,source=my_left_neighbor,…)
…
```

- Processes are waiting for sends or receives which can never be posted → **Deadlock**
- Do not have all processes sending or receiving at the same time with blocking calls
  - Use special communication patterns for example, even-odd
  - Use **MPI_Sendrecv**
  - Use nonblocking routines

⚠ Pitfall 2 – Blocking point-to-point communication

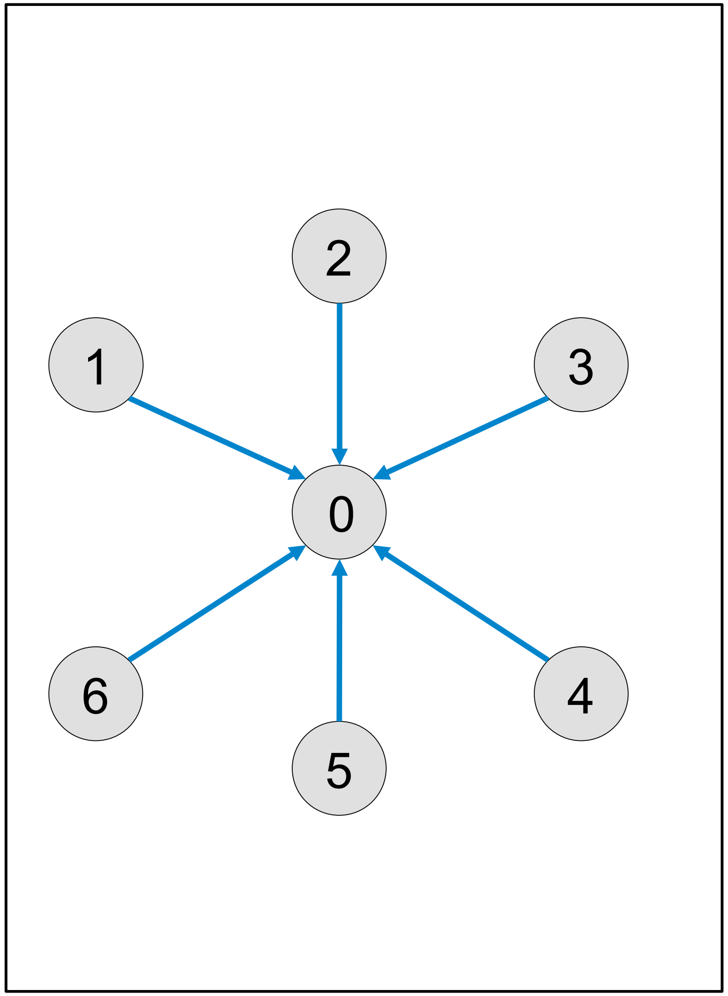
Assumption: MPI_Send is implemented as buffered send



```
if (myrank == 0) {
  ierr = MPI_Send(…,dest=1,…)
  ierr = MPI_Recv(…,source=1,…)
}
else {
  ierr = MPI_Send(…,dest=0,…)
  ierr = MPI_Recv(…,source=0,…)
}
```

- **MPI_Send** will return immediately if the message was buffered
- If buffer is filled, **MPI_Send** will be *synchronous*! → **Deadlock**
- Avoid posting many sends/large buffer without corresponding receives or better: DO NOT ASSUME BUFFERING!

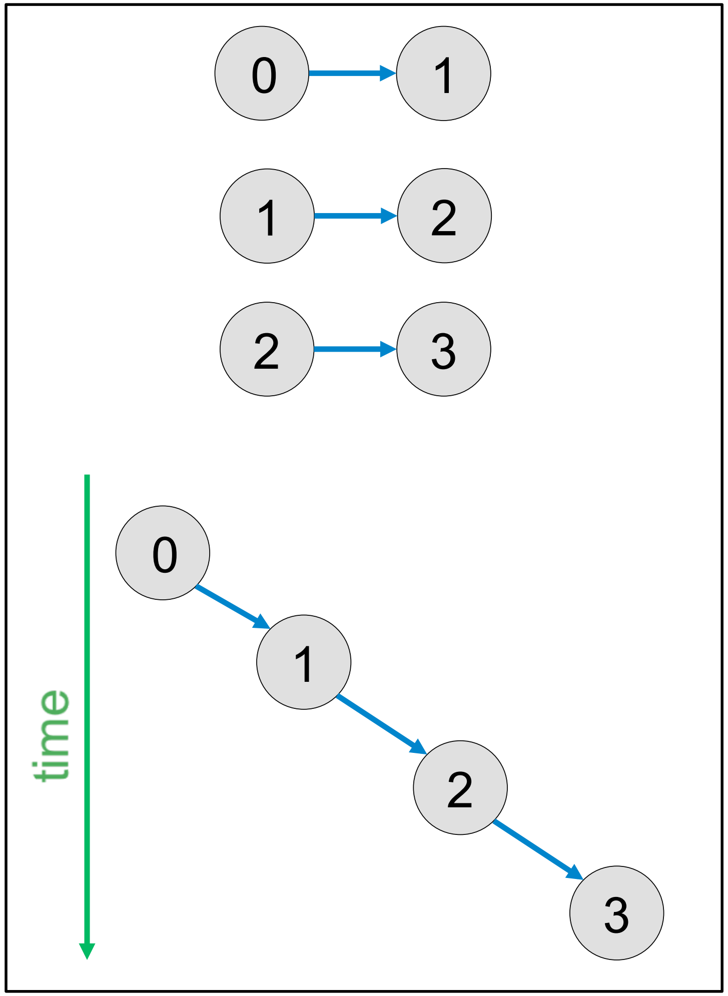
⚠ Pitfall 3 – Blocking point-to-point communication



```
if (my_rank != 0){
  for (i=1;i<=100000;i++){
    ierr = MPI_Send(…,dest=0,…)
  }
}
else {
  → receive messages
}
```

- Will fill-up message and/or envelop buffers
  → **Performance penalty**
  - Try to combine messages
  - Use → collective communication
  - Posting receives before sends reduces buffer space

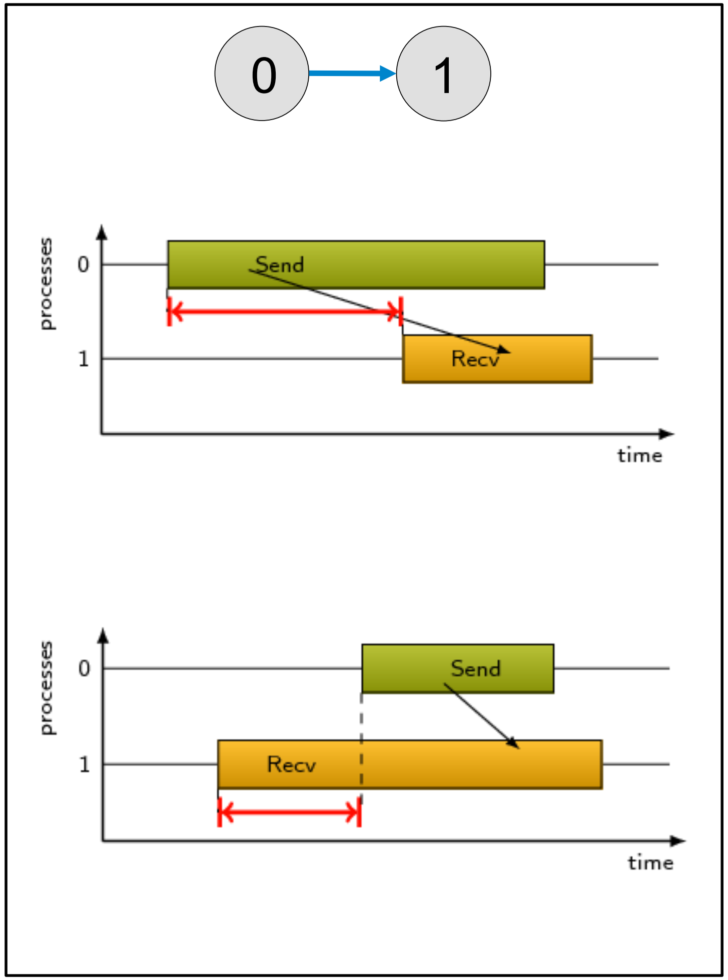
⚠ Pitfall 4 – Blocking point-to-point communication



```
if (my_rank == 0) then
  call MPI_Ssend(…,dest=1,…)
else if (myrank == 1) then
  call MPI_Recv(…,source=0,…)
  call MPI_Ssend(…,dest=2,…)
else if (myranks == 2) then
  call MPI_Recv(…,source=1,…)
  call MPI_Ssend(…,dest=3,…)
else if (myrank == 3) then
  call MPI_Recv(…,source=2,…)
endif
```

- Usage of synchronized sends might lead to **serialization**
- Use buffered send or → nonblocking send/receive

⚠ Pitfall 5 – Blocking point-to-point communication



Example:

Communication between task 0 and 1

- Sender waits for receiver to call corresponding receive operation
- Performance penalty
- Use nonblocking calls

- Receiver waits for sender to call corresponding send operation
- Performance penalty
- Use nonblocking calls

Blocking point-to-point – Recommendations

MPI_Send / **MPI_Sendrecv** should give best performance 💡3 💡4

- Minimal transfer time since MPI can optimize communication
- Can be implemented as synchronous or buffered send – do not assume either case

Synchronous send **MPI_Ssend**

- High latency, good bandwidth
- No buffers are used (saving resources) 💡2
- Risk of idle times, serialization, deadlocks

Buffered send **MPI_Bsend**

- Low latency, buffer does not scale with message size
- Try → **MPI_Isend** instead

Mitglied der Helmholtz-Gemeinschaft

JÜLICH FORSCHUNGSZENTRUM

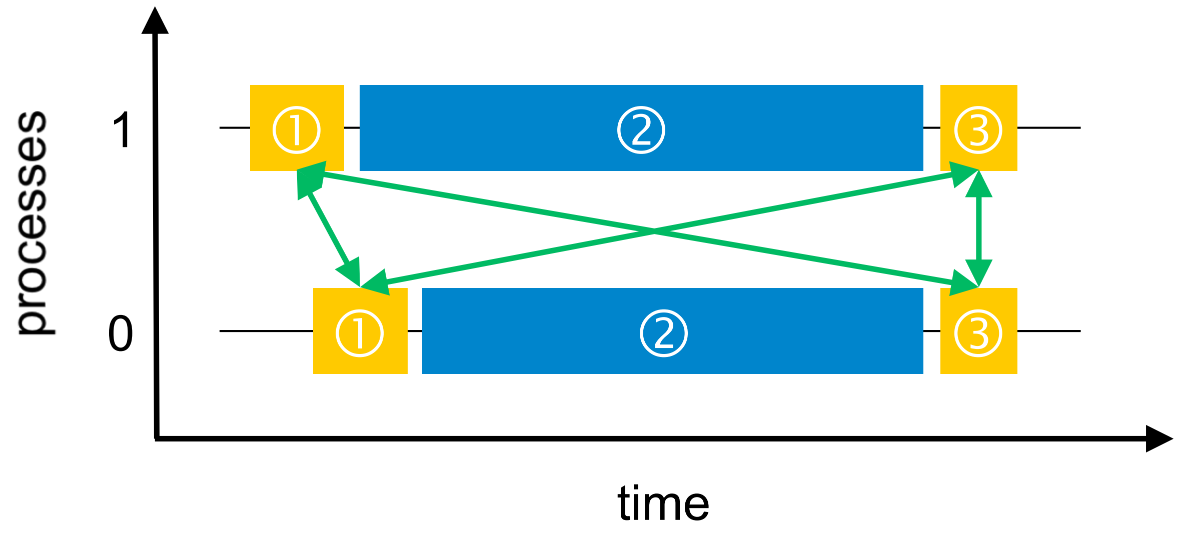
Point-to-Point Communication

Nonblocking communication

July 9, 2014 | Florian Janetzko

Nonblocking Communication

Solution for many pitfalls in blocking communication

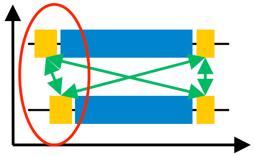


🟦 Computation

🟨 Communication

① Initialization of communication
② Attending other work/test for completion
③ Completion of communication

Phase ① – General



Nonblocking MPI routines

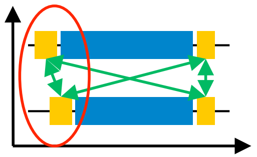
- Prefix

  MPI_I… (‘I’ for ‘immediate)

- Properties

  Nonblocking routines return before the communication has completed

  Nonblocking routines have the same arguments as their blocking counterparts except for an extra **request** argument



Phase ① – Communication modes

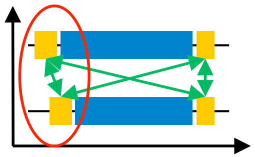
Send modes

Synchronous send:	**MPI_Issend**
Buffered send:	**MPI_Ibsend**
Standard send:	**MPI_Isend**
Ready send:	**MPI_Irsend**

Receive all modes

Receive:	**MPI_Irecv**
Probe:	**MPI_Iprobe**

Phase ① – Nonblocking send



C/C++

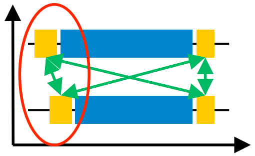
```
int MPI_Isend(void *buf, int count, MPI_Datatype
              datatype, int dest, int tag, MPI_Comm comm,
              MPI_Request *request)
```

Fortran

```
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,
      IERROR)
  <type>  :: BUF(*)
  INTEGER :: COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

- Nonblocking send routines have the same arguments as their blocking counterparts except for the extra **REQUEST** argument
- Send buffer **BUF** must not be accessed before the send has been successfully tested for completion with **MPI_WAIT** or **MPI_TEST**

Phase ① – Nonblocking receive



C/C++

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Request *request)
```

Fortran

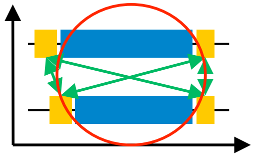
```
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM,
        REQUEST, IERROR)
 <type>  :: BUF(*)
 INTEGER :: COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
```

- Nonblocking receive routines have the same arguments as its blocking counterpart except for the extra **REQUEST** argument
- Send buffer BUF must not be accessed before the receive has been successfully tested for completion with **MPI_WAIT** or **MPI_TEST**

The **REQUEST** handle

- Used for nonblocking communication
- Request handles must be stored in a variable of sufficient scope
  - C/C++ : **MPI_Request**
  - Fortran : **INTEGER**
- A nonblocking communication routine returns a value for the request handle
- This value is used by **MPI_WAIT** or **MPI_TEST** to test when the communication has completed
- If the communication has completed the request handle is set to **MPI_REQUEST_NULL**

Phase ② – Test



C/C++

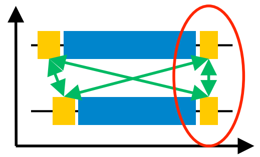
```
int MPI_Test(MPI_Request *request, int *flag,
             MPI_Status *status)
```

Fortran

```
MPI_TEST(REQUEST, STATUS, FLAG, IERROR)
 LOGICAL :: FLAG
 INTEGER :: REQUEST, STATUS, IERROR
```

- If communication associated with REQUEST is complete call returns flag=true, otherwise flag=false (nonblocking call)
- If several communications are pending (MPIS3.0, 3.7.5)
  - MPI_Testall
  - MPI_Testsome
  - MPI_Testany

Phase ③ – Wait



C/C++

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

Fortran

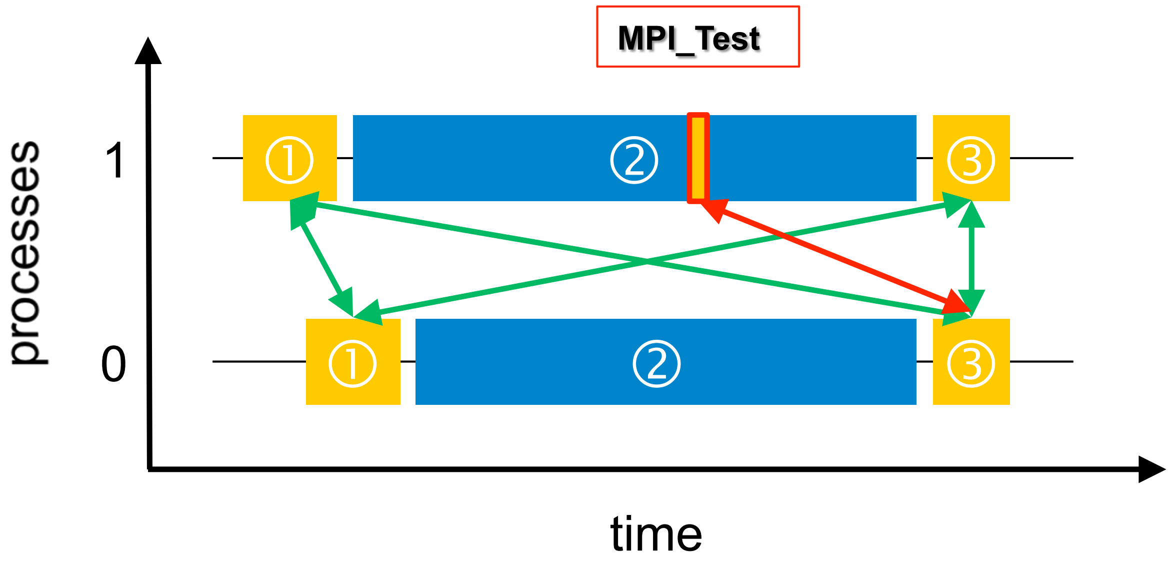
```
MPI_WAIT(REQUEST, STATUS, IERROR)
  INTEGER :: REQUEST, STATUS, IERROR
```

- Waits until communication associated with **REQUEST** is completed (call is blocking)
- If several communications are pending (MPIS3.0, 3.7.5)
  - **MPI_Waitall**
  - **MPI_Waitsome**
  - **MPI_Waitany**

⚠ Pitfall – Not truly nonblocking commnication

Communication advances in general in MPI routines

- **MPI_Isend[…], MPI_Irecv[…], MPI_Test[…], MPI_Wait[…]**



→ Communication *may not be truly asynchronous*!

⚠ Pitfall – Not truly nonblocking commnication

Usually nonblocking communication is used to

- Avoid deadlocks
- Avoid idle times to wait for receiver or sender

Therefore

- Use nonblocking routines for these cases
- Do not spend too much time in this
- Check system documentation (environment variables)

⚠ Pitfall – Environment settings

On some systems *special environment* variables must be set to benefit from nonblocking communication

- One-sided or nonblocking point-to-point communication
  - Blue Gene/P

    export DCMF_INTERRUPT=1
  - Blue Gene/Q

    export PAMID_ASYNC_PROGRESS=1

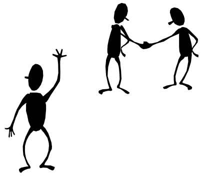
Check information for your system!

Also check other environment settings on the system!

Pitfall – Remember the communication protocol

MPI usually *switches* protocols depending on message size

- Large messages → rendezvous protocol
- Small messages → eager protocol



Trade-off between latency and bandwidth

- Rendezvous protocol might lead to time penalties if sender is blocked while waiting for receiver
- Eager protocol might lead to time penalties when large messages are send with low bandwidth

Check limit for rendezvous protocol and adjust it to your needs!

Outline

- Introduction
- Parallel Algorithms – an Example for a Design Strategy
- Message-Passing Interface – Overview
- MPI – Selected Topics and Best Practices
  - General Hints and Remarks
  - Point-to-Point Communication
  - Collective Communication
  - Derived Datatypes
  - MPI_Info Object
  - One-sided Communication

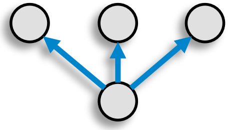
Characteristics of collective communication

- Collective action over a communicator.
  - All processes of the communicator must communicate, i.e. all processes must call the collective routine.
- Synchronization may or may not occur
- Collective operations can be blocking or nonblocking (MPI3.0)
- No tags are used

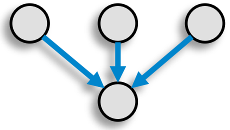
Collective communication – overview

Blocking collectives

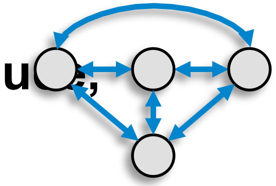
- One-to-all
  - **MPI_Bcast, MPI_Scatter[v]**



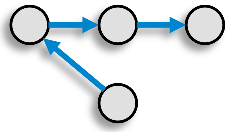
- All-to-one
  - **MPI_Gather[v], MPI_Reduce**



- All-to-all
  - **MPI_Allgather[v], MPI_All_to_all[v,w], MPI_Allreduce, MPI_Reduce_scatter**



- Other
  - **MPI_Scan, MPI_Exscan**



Nonblocking collectives **(MPI 3.0, 5.12)**

- Use the same semantics as above, just add an ‘I’ and a request:
  - **MPI_I<*routine name*>(…, request)**

Example: broadcast

Blocking

C/C++

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm)
```

Fortran

```
MPI_BCAST(BUF, COUNT, DATATYPE, ROOT, COMM, IERROR)
  <type>  :: BUF(*)
  INTEGER :: COUNT, DATATYPE, ROOT, COMM, IERROR
```

Nonblocking

C/C++

```
int MPI_Ibcast(void *buf, int count, MPI_Datatype datatype,
               int root, MPI_Comm comm, MPI_Request *req)
```

Fortran

```
MPI_IBCAST(BUF, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR)
  <type>  :: BUF(*)
  INTEGER :: COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR
```

Properties of nonblocking collective routines

New in **MPI3.0, 5.12** (available in newer MPI implementations)

- Properties similar to nonblocking *point-to-point* communication
  - ① Initialization of communication
  - ② Attending other work/test for completion
  - ③ Completion of communication
  - Return before the communication has completed
  - Have the same arguments as their blocking counterparts except for an extra request argument
  - Same completion calls (e.g. **MPI_Wait** etc.)

EXCEPTION: *nonblocking* collective operations cannot be matched with *blocking* collective operations

General hints for collective routines

Use if suitable for your algorithm

- Should be vendor optimized
- Provides MPI with all information (↔ split into P2P)
  - ➢ Should give best performance (also for **MPI I/O**!!)

Don’t use **MPI_Barrier** (except for debugging purposes)

Try to avoid all-to-all communication

⚠ Pitfall 1 – Collective communication

```
…
if (my_rank == 0)
{
  MPI_Bcast(&result, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
else
{
  MPI_Recv(&result, 1, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, stat);
}
…
```

Collective routines

- ALL ranks of a communicator have to execute them
- Do not mix P2P and collective routines!

⚠ Pitfall 2 – Collective communication

```
…
if (my_rank == 0)
{
  MPI_Bcast(&result, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
else
{
  MPI_Ibcast(&result, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD, &request);
}
…
```

Do not mix blocking and nonblocking collectives!

Pitfall 3 – Collective communication

```
…
if (my_rank == 0)
{
  MPI_Bcast(&result1, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast(&result2, 1, MPI_DOUBLE, 1, MPI_COMM_WORLD);
}
else
{
 MPI_Bcast(&result2, 1, MPI_DOUBLE, 1, MPI_COMM_WORLD);
 MPI_Bcast(&result1, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
…
```

Blocking collective

- Operations must be executed in the same order on all participating tasks
- Otherwise a deadlock will occur

Outline

- Introduction
- Parallel Algorithms – an Example for a Design Strategy
- Message-Passing Interface – Overview
- MPI – Selected Topics and Best Practices
  - General Hints and Remarks
  - Point-to-Point Communication
  - Collective Communication
  - Derived Datatypes
  - MPI_Info Object
  - One-sided Communication

Motivation

With MPI communication calls only multiple consecutive elements of the same type can be sent

Buffers may be non-contiguous in memory

- Sending only the real/imaginary part of a buffer of complex doubles
- Sending sub-blocks of matrices

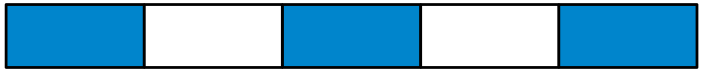
Buffers may be of mixed type

- User defined data structures

```
struct buff_layout {
  int i[4];
  double d[5];
} buffer;
```

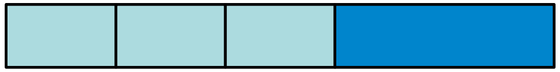
Solutions without MPI derived datatypes

Non-contiguous data of a single type



- Consecutive MPI calls to send and receive each element in turn
  - ➢ *Additional latency costs due to multiple calls*
- Copy data to a single buffer before sending it
  - ➢ *Additional latency costs due to memory copy*

Contiguous data of mixed types



- Consecutive MPI calls to send and receive each element in turn
  - ➢ *Additional latency costs due to multiple calls*

Derived datatypes

- General MPI datatypes describe a buffer layout in memory by specifying
  - A sequence of basic datatypes
  - A sequence of integer (byte) displacements
- Derived datatypes are derived from basic datatypes using constructors
- MPI datatypes are referenced by an opaque handle

MPI datatypes are opaque objects! Using the **sizeof()** operator on an MPI datatype handle will return the size of the handle, neither the size nor the extent of an MPI datatype.

Creating a derived datatype: Type map

Any derived datatype is defined by its type map

- A list of basic datatypes
- A list of displacements (positive, zero, or negative)
- Any type matching is done by comparing the sequence of basic datatypes in the type maps

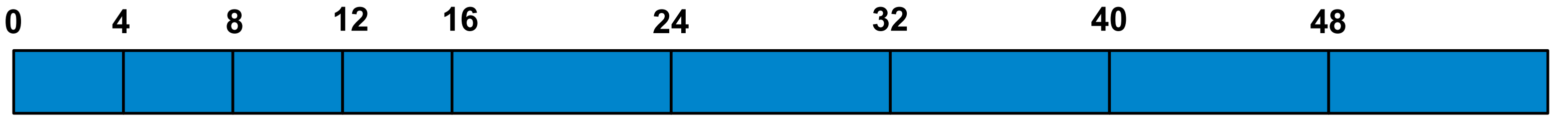
General type map:

Datatype	Displacement
datatype 0	displacement of datatype 0
datatype 1	dispalcement of datatype 1
…	…

Example of a type map

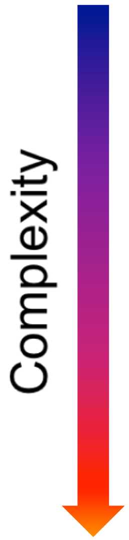
```
struct buff_layout {
  int i[4];
  double d[5];
} buffer;
```

Datatype	Displacement
MPI_INT	0
MPI_INT	4
MPI_INT	8
MPI_INT	12
MPI_DOUBLE	16
MPI_DOUBLE	24
MPI_DOUBLE	32
MPI_DOUBLE	40
MPI_DOUBLE	48



Datatype constructors

Available MPI datatype constructors:



- **MPI_Type_contiguous**
- **MPI_Type_vector**
- **MPI_Type_indexed**
- **MPI_Type_indexed_block**
- **MPI_Type_create_struct**
- **MPI_Type_create_subarray**
- **MPI_Type_create_darray**

Use the simplest derived datatype that suits your needs. The more complex the datatype the slower is its handling.

Some constructors have alternative routines with displacements in bytes instead of multiples of datatypes

- **MPI_Type_create_h[vector,indexed,…]**

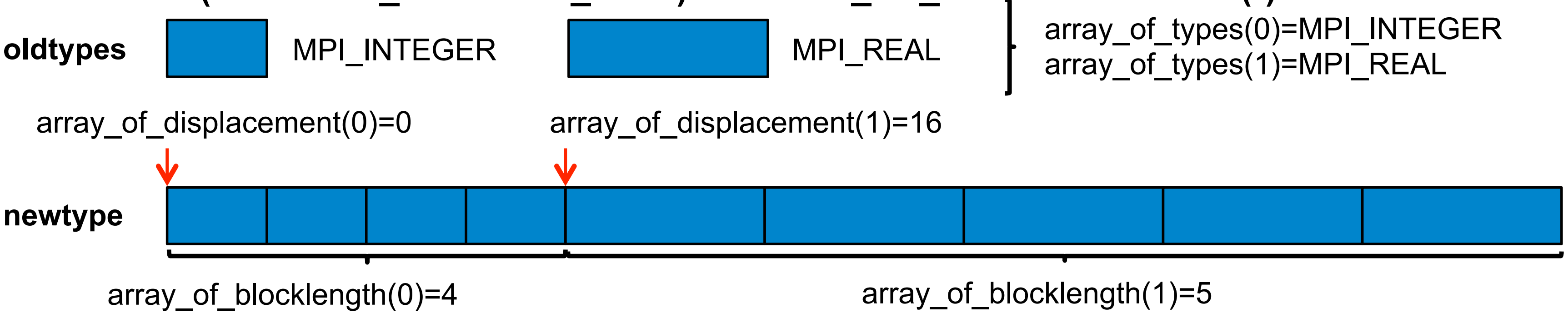
Struct data

C/C++

```
int MPI_Type_create_struct(int count, int *array_of_blocklengths,
    MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types,
    MPI_Datatype *newtype)
```

Fortran

```
MPI_TYPE_CREATE_STRUCT(COUNT,
    ARRAY_OF_BLOCKLENGTHS,ARRAY_OF_DISPLACEMENTS,
                ARRAY_OF_TYPES, NEWTYPE,IERROR)
 INTEGER :: COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_TYPES(*), NEWTYPE,
         IERROR
 INTEGER(KIND=MPI_ADDRESS_KIND) :: ARRAY_OF_DISPLACEMENTS(*)
```


oldtypes
MPI_INTEGER
MPI_REAL
array_of_types(0)=MPI_INTEGER
array_of_types(1)=MPI_REAL
array_of_displacement(0)=0
array_of_displacement(1)=16
newtype
array_of_blocklength(0)=4
array_of_blocklength(1)=5

Committing and freeing derived datatypes

C/C++

```
int MPI_Type_commit(MPI_Datatype *datatype)
```

Fortran

```
MPI_TYPE_COMMIT(DATATYPE, IERROR)
 INTEGER :: DATATYPE, IERROR
```

- Before it can be used in a communication, each derived datatype has to be committed

C/C++

```
int MPI_Type_free(MPI_Datatype *datatype)
```

Fortran

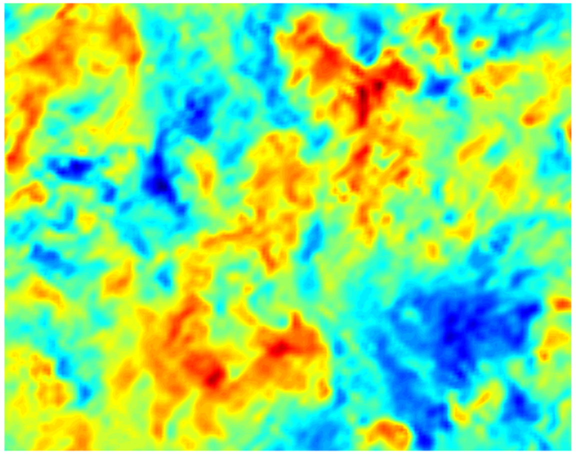
```
MPI_TYPE_FREE(DATATYPE, IERROR)
 INTEGER :: DATATYPE, IERROR
```

- Mark a datatype for deallocation
- Datatype will be deallocated when all pending operations are finished

Example: exchanging velocity information

For each point there are

- 3 coordinates
- 3 color values (r, g, b, ϵ [0, 255])



C/C++

```
struct pointstrct
{
  int    r;
  int    g;
  int    b;
  double x;
  double y;
  double z;
} point;
```

Fortran

```
type pointstrct
  integer   :: r
  integer   :: g
  integer   :: b
  real*8    :: x
  real*8    :: y
  real*8    :: z
end type pointstrct
type (pointstrct) :: point
```

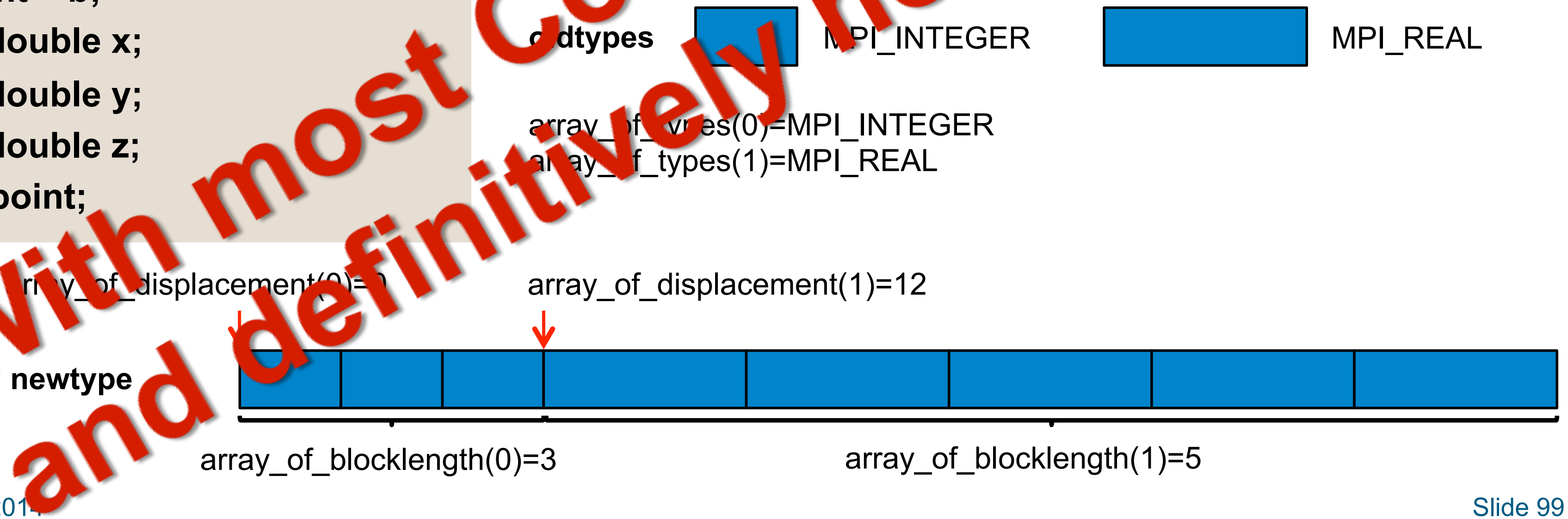
Pitfall 1 – Derived datatypes for structs/types

Constructing derived datatype with **MPI_Type_create_struct**

C/C++

```
struct pointstrct
{
  int    r;
  int    g;
  int    b;
  double x;
  double y;
  double z;
} point;
```

```
int MPI_Type_create_struct(int count,
                 int *array_of_blocklengths,
                 MPI_Aint *array_of_displacements,
                 MPI_Datatype *array_of_types,
                 MPI_Datatype *newtype)
```


oldtypes
MPI_INTEGER
MPI_REAL
array_of_types(0)=MPI_INTEGER
array_of_types(1)=MPI_REAL
array_of_displacement(0)=0
array_of_displacement(1)=12
newtype
array_of_blocklength(0)=3
array_of_blocklength(1)=5

With most Compilers wrong and definitively not portable!

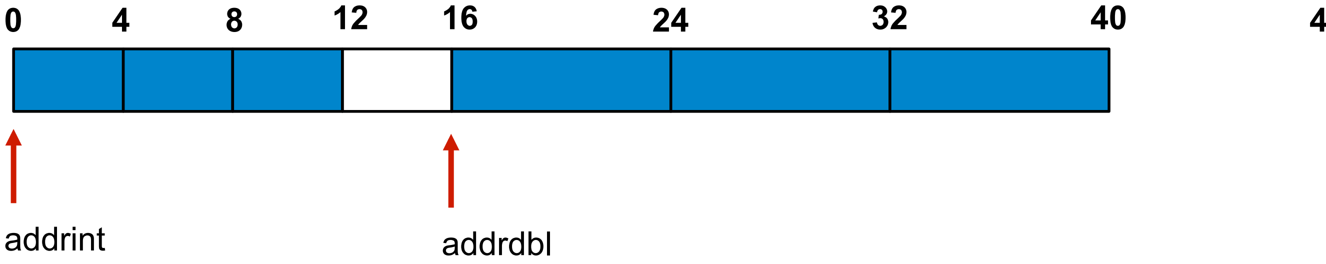
⚠ Pitfall 1 – Derived datatypes for structs/types

Constructing derived datatype with **MPI_Type_create_struct**

- Take (possible) alignment into account!

C/C++

```
struct pointstrct
{
  int    r;
  int    g;
  int    b;
  double x;
  double y;
  double z;
} point;
```



Do not rely on C’s address operator &, as ANSI C does not guarantee pointer values to be absolute addresses. Furthermore, address space may be segmented. Always use **MPI_Get_address**, which also guarantees portability (Address relative to **MPI_BOTTOM**).

⚠ Pitfall 1 – Derived datatypes for structs/types

C/C++

```
int MPI_Get_address(void *location, MPI_Aint *address)
```

- Finding addresses and relative displacements of memory blocks

```
MPI_Aint addr_block_1, addr_block_2;
MPI_Aint displacement = 0;

MPI_Get_address(&block_1, &addr_block_1);
MPI_Get_address(&block_2, &addr_block_2);

displacement = addr_block_2 - addr_block_1;
```

Fortran

```
MPI_GET_ADDRESS(LOCATION,ADDRESS,IERROR)
 <type>  :: LOCATION(*)
 INTEGER(KIND=MPI_ADDRESS_KIND) :: ADDRESS
 INTEGER :: IERROR
```

Correct derived datatypes for structs (C)

C/C++

```
MPI_Get_address(&point,&addrbase);
MPI_Get_address(&point.x,&addrdbl);

displ[0] = 0;
displ[1] = addrdbl – addrbase;

type[0] = MPI_INT;
type[1] = MPI_DOUBLE;

length[0] = 3;
length[1] = 3;

MPI_Type_create_struct(2,length,displ,type,&mypoint);
MPI_Type_commit(&mypoint);
```

```
struct pointstrct
{
  int    r;
  int    g;
  int    b;
  double x;
  double y;
  double z;
} point;
```

Correct derived datatypes for types (FORTRAN)

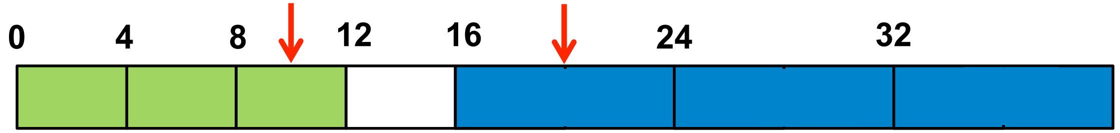
Fortran

```
call MPI_Address(point,addrbase,ierror)
call MPI_Address(point%x,addrdbl,ierror)

displ(0) = 0
displ(1) = addrdbl – addrbase

type(0) = MPI_INTEGER
type(1) = MPI_REAL8

length(0) = 3
length(1) = 3

call MPI_Type_create_struct(2,length,displ,type,mypoint,ierror)
call MPI_Type_commit(mypoint,ierror)
```

```
type pointvel
  integer   :: r
  integer   :: g
  integer   :: b
  real*8    :: x
  real*8    :: y
  real*8    :: z
end type pointvel
type (pointstrct) :: point
```

Pitfall 2 – Sending parts of structures

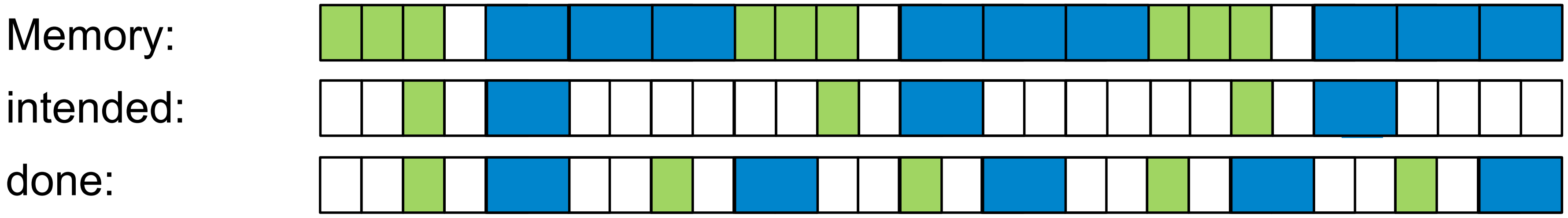
```
struct pointstrct
{
  int    r;
  int    g;
  int    b;
  double x;
  double y;
  double z;
} point[10];
```



Just want to send **b** and x of all elements

```
MPI_Get_address(&point,&addrbase);
MPI_Get_address(&point.b,&addrb);
MPI_Get_address(&point.x,&addrdbl);
displ[0] = addrb - addrbase;
displ[1] = addrdbl – addrbase;
type[0] = MPI_INT;
type[1] = MPI_DOUBLE;
length[0] = 1;
length[1] = 1;
MPI_Type_create_struct(2,length,displ,type,&newpoint);
MPI_Type_commit(&newpoint);
```

Wrong!
(except for first Element)



Sending parts of structures

MPI-1 solution

- Include lower and upper bounds (**MPI_LB, MPI_UB**)

```
MPI_Get_address(&point[0],&addrbase);
MPI_Get_address(&point[0].b,&addrb)
MPI_Get_address(&point[0].x,&addrx);
MPI_Get_address(&point[1],&addrn);
displ[0] = 0;
displ[1] = addrdb – addrbase;
displ[2] = addrdx – addrbase;
displ[3] = addrdn – addrbase;
type[0] = MPI_LB;     length[0] = 1;
type[1] = MPI_INT;    length[0] = 1;
type[2] = MPI_DOUBLE; length[0] = 1;
type[3] = MPI_UB;     length[0] = 1;

MPI_Type_create_struct(4,length,displ,type,&mypoint);
MPI_Type_commit(&mypoint);
```

Removed from standard!

Do not use!

Sending parts of structures

MPI-2 solution

- Resize the derived datatype **newpoint**
  - Size of a datatype: number of bytes actually transferred
  - Extend of a datatype: UB - LB

C/C++

```
int MPI_Type_create_resized(MPI_Datatype oldtype,
                            MPI_Aint lb, MPI_Aint extent,
                            MPI_Datatype* newtype)
```

Fortran

```
MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE,
                        IERROR)
 INTEGER :: OLDTYPE, NEWTYPE, IERROR
 INTEGER(KIND=MPI_ADDRESS_KIND) :: LB, EXTENT
```

Sending parts of structures

C/C++

```
[…]
MPI_Get_address(&point[0],&addrbase);
MPI_Get_address(&point[1],&addrn);

lb = addrbase;
extend = addrn – addrbase;

MPI_Type_create_resized(newpoint,lb,extend,&npoint);
MPI_Type_commit(&npoint);
```

Outline

- Introduction
- Parallel Algorithms – an Example for a Design Strategy
- Message-Passing Interface – Overview
- MPI – Selected Topics and Best Practices
  - General Hints and Remarks
  - Point-to-Point Communication
  - Collective Communication
  - Derived Datatypes
  - MPI_Info Object
  - One-sided Communication

MPI_Info object (MPIS 3.0, 9+13.2.8)

3

- Can be used to pass hints for optimization to MPI (file system dependent)
- Consists of (key,value) pairs, where key and value are strings
- A key may have only one value
- **MPI_INFO_NULL** is always a valid **MPI_Info** object
- The maximum key size is **MPI_MAX_INFO_KEY**
- The maximum value size is **MPI_MAX_INFO_VALUE** (implementation dependent)

MPI_MAX_INFO_VALUE might be very large! It is not advisable to declare strings of that size!

Create and free MPI_Info objects

C/C++

```
int MPI_Info_create(MPI_Info info)
```

Fortran

```
MPI_INFO_CREATE(INFO, IERROR)
 INTERGER :: INFO, IERROR
```

- The created info objects contains no (key,value) pairs

C/C++

```
int MPI_Info_free(MPI_Info info)
```

Fortran

```
MPI_INFO_FREE(INFO, IERROR)
 INTERGER :: INFO, IERROR
```

- The info object is freed and set to **MPI_INFO_NULL**

Set and delete (key,value) pairs

C/C++

```
int MPI_Info_set(MPI_Info info, char *key, char *value)
```

Fortran

```
MPI_INFO_SET(INFO, KEY, VALUE, IERROR)
  CHARACTER(*) :: KEY, VALUE
  INTERGER     :: INFO, IERROR
```

C/C++

```
int MPI_Info_delete(MPI_Info info, char *key)
```

Fortran

```
MPI_INFO_DELETE(INFO, KEY, IERROR)
  CHARACTER(*) :: KEY
  INTERGER     :: INFO, IERROR
```

Retrieve active (key,value) pairs of an info object

C/C++

```
int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)
```

Fortran

```
MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)
 INTERGER     :: INFO, NKEYS, IERROR
```

C/C++

```
int MPI_Info_get_nthkey(MPI_Info info, int n, char *key);
```

Fortran

```
MPI_INFO_GET_NTHKEY(INFO, N, KEY, IERROR)
 CHARACTER(*):: KEY
 INTERGER     :: INFO, N, IERROR
```

Retrieve active (key,value) pairs of an info object

C/C++

```
int MPI_Info_get_valuelen(MPI_Info info, const char *key,
                          int *valuelen, int *flag)
```

Fortran

```
MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
  CHARACTER (*) :: KEY
  INTERGER      :: INFO, VALUELEN, IERROR
  LOGICAL       :: FLAG
```

Retrieve active (key,value) pairs of an info object

C/C++

```
int MPI_Info_get(MPI_Info info, char *key,
                 int valuelen, char *value, int *flag)
```

Fortran

```
MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)
  CHARACTER(*):: KEY, VALUE
  INTERGER     :: INFO, VALUELEN, IERROR
  LOGICAL      :: FLAG
```

The function returns in **flag** either **true** if **key** is defined in **info**, otherwise it returns **false**

Example: Info objects for MPI I/O

Two possibilities

1. Pass info object when opening file
   - Information about file system properties
   - Hints for MPI about access to file
   - Information about buffering

2. Associate info object with open file
   - Same information can be passed
   - Some I/O properties cannot be changed if the file is already open → Information may be ignored

1. Associate info objects when opening a file

C/C++

```
int MPI_File_open(MPI_Comm comm, char *filename,
                  int amode, MPI_Info info, MPI_File *fh)
```

Fortran

```
MPI_FILE_OPEN(COMM, FILENAME ,AMODE, INFO, FH, IERROR)
 CHARACTER*(*) :: FILENAME
 INTEGER       ::COMM,AMODE,INFO,FH,IERROR
```

2. Associate info objects with an open file

C/C++

```
int MPI_File_set_info(MPI_File fh, MPI_Info info)
```

Fortran

```
MPI_FILE_SET_INFO(FH, INFO, IERROR)
 INTEGER    :: FH, INFO, IERROR
```

- Info items that cannot be changed for an open file need to be set when opening the file
- MPI implementation may choose to ignore the hints in this call

Example keys for MPI I/O MPI_Info objects

access_style (comma-separated list)

- Specify manner in which the file will be accessed
- **read_once**, **write_once**, **read_mostly**, **write_mostly**, random

collective_buffering (true | false)

- Specifies whether application benefits from collective buffering

cb_buffer_size (integer, bytes)

- Size of the buffer for collective buffering

cb_block_size (integer, bytes)

- Size of chunks in which data is accessed

striping_factor (integer)

- Number of devices to stripe over

striping_unit (integer, bytes)

- No of bytes on each device

⚠ Pitfalls – MPI_Info objects

Unknown keys

- Which keys are supported depends on the MPI implementation
- MPI may choose to ignore unsupported keys

Values cannot become effective

- Some values can only be changed in certain routines (e.g. in MPI I/O some values must be set when opening files)

! Check always which keys are available and whether they are active and set correctly! !

Outline

- Introduction
- Parallel Algorithms – an Example for a Design Strategy
- Message-Passing Interface – Overview
- MPI – Selected Topics and Best Practices
  - General Hints and Remarks
  - Point-to-Point Communication
  - Collective Communication
  - Derived Datatypes
  - MPI_Info Object
  - One-sided Communication

Motivation

Point-to-point and collective MPI routines

- Sender needs to know which data to send and to which process
- Receiver needs to wait for sender (cannot initiate transfer)

Drawbacks for some communication patterns

- Sending process might not know what to send or to which process to send
- Receiving process needs to initiate transfer

One-sided communication

- RMA (Remote Memory Access)

Terminology

Origin

The process triggering the one-sided operation, specifying all needed parameters.

Target

The process providing access to its memory through a defined window. The target does not explicitly participate in the data transfer.

Active target communication: Both origin and target process are involved in the communication.
Passive target communication: Only the origin process(es) is (are) involved in the communication.

Terminology

Window

A block of memory opened for remote access through MPI RMA operations. Its definition is collective on all processes using this window. Only designated targets have to specify a valid buffer, origins can use a special placeholder to obtain a handle without opening memory for remote access.

Exposure epoch (→ Target process)

An exposure epoch is the time interval some defined data access is allowed on a window. It starts and ends with synchronizations calls on the target process (only for active target communication).

Access epoch (→ Origin process)

An access epoch is the time interval from the origin process’ start signal of data access to its end signal of data access on a window.

Terminology – Overview

origin tasks: l and n
target task : m

🟪 local memory

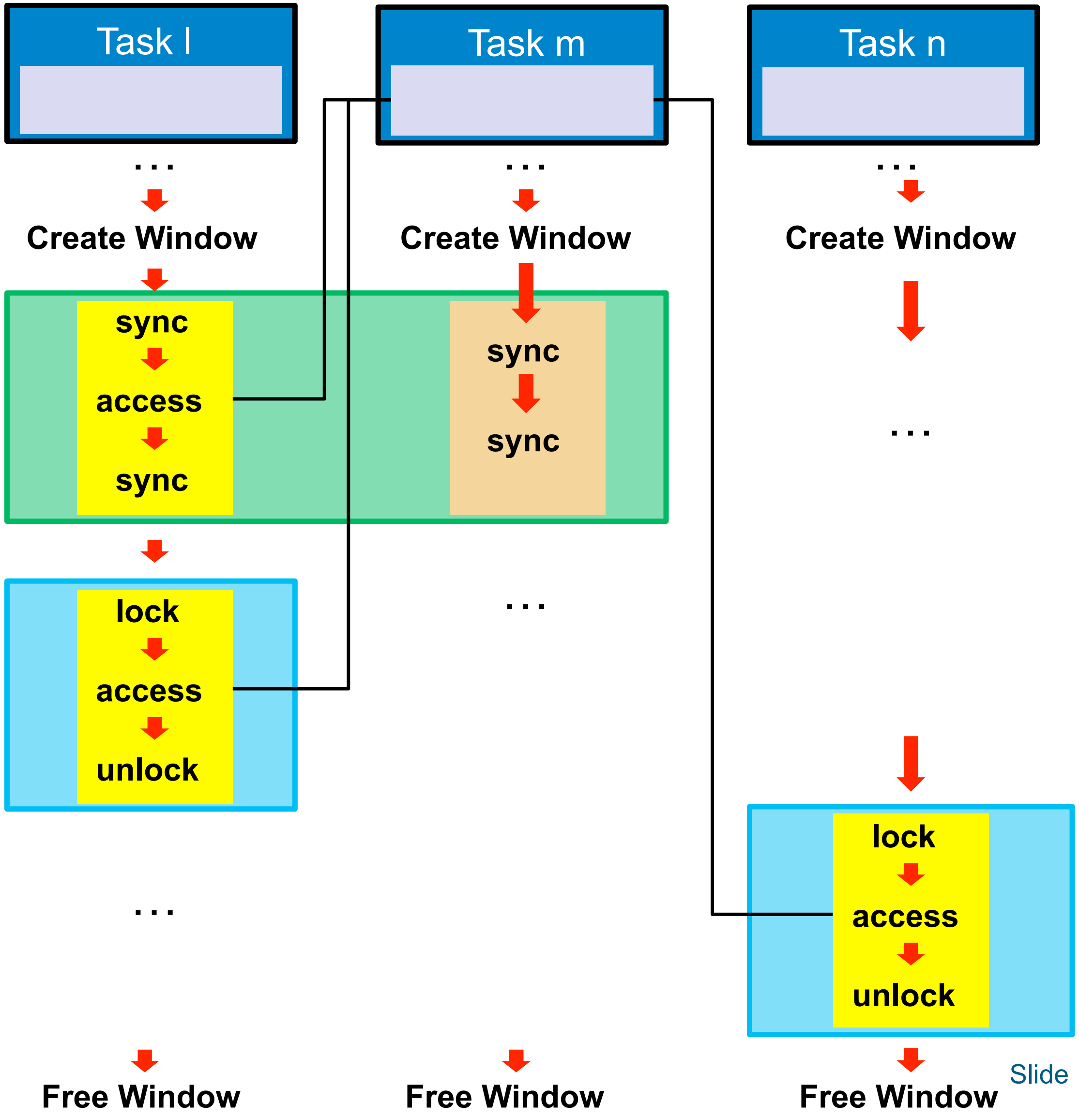
🟧 window

🟨 access epoch

🟫 exposure epoch

🟩 active target communication

🟦 passive target communication



Initialization – Window creation (I)

C/C++

```
int MPI_Win_create(void *base, MPI_Aint *size,
                   int disp_unit, MPI_Info info,
                   MPI_Comm comm, MPI_Win *win)
```

Fortran

```
MPI_WIN_CREATE(BASE, SIZE, DISP_UNIT, INFO, COMM, WIN,
               IERROR)
 <type>  :: BASE(*)
 INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
 INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
```

- Memory local to each process is allocated by the user
- Collective operation over **COMM**
- **SIZE** is the size of the memory part in bytes
- **DISP_UNIT** sets the offset handling (e.g. **sizeof(type)**)
- **INFO** handle can be set to **MPI_INFO_NULL**

Initialization – Window creation (II)

C/C++

```
int MPI_Win_allocate(MPI_Aint *size, int disp_unit,
                     MPI_Info info, MPI_Comm comm,
                     void *baseptr, MPI_Win *win)
```

Fortran

```
MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR,
                 WIN, IERROR)
 INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
 INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
```

- Memory local to each process is allocated by MPI
- Collective operation over **COMM**
- **SIZE** is the size of the memory part in bytes
- **DISP_UNIT** sets the offset handling (e.g. **sizeof(type)**)
- **INFO** handle can be set to **MPI_INFO_NULL**

Initialization – Window creation (III)

C/C++

```
int MPI_Win_allocate_shared
    (MPI_Aint *size, int disp_unit, MPI_Info info,
     MPI_Comm comm, void *baseptr, MPI_Win *win)
```

Fortran

```
MPI_WIN_ALLOCATE_SHARED
    (SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)
 INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
 INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
```

- Shared Memory is allocated by MPI
- Collective operation over **COMM**
- **SIZE** is the size of the memory part in bytes
- **DISP_UNIT** sets the offset handling (e.g. **sizeof(type)**)
- **INFO** handle can be set to **MPI_INFO_NULL**

Get Address of other Tasks’ Memory Segments

C/C++

```
int MPI_Win_shared_query
    (MPI_Win win, int rank, MPI_Aint *size,
     int *disp_unit, void *baseptr)
```

Fortran

```
MPI_WIN_SHARED_QUERY
    (WIN, RANK, SIZE, DISP_UNIT, BASEPTR, IERROR)
 INTEGER :: WIN, RANK, DISP_UNIT, IERROR
 INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
```

- Returns the address of the memory segment of **RANK**
- May return different (process-local) addresses for the same segment on different processes
- Returns **SIZE**, **DISP_UNIT**, and **BASEPTR**

Initialization – Window creation (IV)

C/C++

```
int MPI_Win_create_dynamic (MPI_Info info, MPI_Comm comm,
                            MPI_Win *win)
```

Fortran

```
MPI_WIN_CREATE_DYNAMIC (INFO, COMM, WIN, IERROR)
 INTEGER :: INFO, COMM, WIN, IERROR
```

- Returns a window without memory
- Collective operation over **COMM**
- **INFO** handle can be set to **MPI_INFO_NULL**

Attaching Memory to a Dynamic Window

C/C++

```
int MPI_Win_attach (MPI_Win win, void *base,
                   MPI_Aint size)
```

Fortran

```
MPI_WIN_ATTACH (WIN, BASE, SIZE, IERROR)
  INTEGER :: WIN, IERROR
  <type>  :: BASE(*)
  INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
```

- Attaches a local memory region beginning at **BASE** for RMA
- Must not contain any part already attached to a window
- User has to ensure that **MPI_WIN_ATTACH** at the target has completed before it is accessed remotely and that enough memory is available

Detaching Memory to a Dynamic Window

C/C++

```
int MPI_Win_detach (MPI_Win win, void *base)
```

Fortran

```
MPI_WIN_DETACH (WIN, BASE, IERROR)
  INTEGER :: WIN, IERROR
  <type>  :: BASE(*)
```

- Detaches a local memory region beginning at **BASE**
- The arguments **WIN** and **BASE** must match the corresponding arguments in a previous **MPI_WIN_ATTACH** call
- Memory becomes detached when window is freed

MPI RMA operation put

C/C++

```
int MPI_Put(void* origin_addr, int origin_count,
            MPI_Datatype origin_type, int target_rank,
            MPI_Aint target_disp, int target_count,
            MPI_Datatype target_type, MPI_Win win)
```

Fortran

```
MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_TYPE,
        TARGET_RANK, TARGET_DISP, TARGET_COUNT,
        TARGET_TYPE, WIN, IERROR)
 <type>  :: ORIGIN_ADDR(*)
 INTEGER :: ORIGIN_COUNT, ORIGIN_TYPE, TARGET_RANK,
            TARGET_COUNT, TARGET_TYPE, WIN, IERROR
 INTEGER(KIND=MPI_ADDRESS_KIND) :: TARGET_DISP
```

- Transfer origin → target
- No matching call on target side

MPI RMA operation accumulate

C/C++

```
int MPI_Accumulate(void* origin_addr, int origin_count,
        MPI_Datatype origin_type, int target_rank,
        MPI_Aint target_disp, int target_count,
        MPI_Datatype target_type, MPI_Op op, MPI_Win win)
```

Fortran

```
MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_TYPE,
        TARGET_RANK, TARGET_DISP, TARGET_COUNT,
        TARGET_TYPE, OP, WIN, IERROR)
 <type>  :: ORIGIN_ADDR(*)
 INTEGER :: ORIGIN_COUNT, ORIGIN_TYPE, TARGET_RANK,
          TARGET_COUNT, TARGET_TYPE, OP, WIN, IERROR
 INTEGER(KIND=MPI_ADDRESS_KIND) :: TARGET_DISP
```

- Buffer elements on target side are combined with operation **OP**

MPI RMA operation get

C/C++

```
int MPI_Get(void* origin_addr, int origin_count,
            MPI_Datatype origin_type, int target_rank,
            MPI_Aint target_disp, int target_count,
            MPI_Datatype target_type, MPI_Win win)
```

Fortran

```
MPI_GET (ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_TYPE,
      TARGET_RANK, TARGET_DISP, TARGET_COUNT,
      TARGET_TYPE, WIN, IERROR)
 <type>  :: ORIGIN_ADDR(*)
 INTEGER :: ORIGIN_COUNT, ORIGIN_TYPE, TARGET_RANK,
           TARGET_COUNT, TARGET_TYPE, WIN, IERROR
 INTEGER(KIND=MPI_ADDRESS_KIND) :: TARGET_DISP
```

- Transfer target → origin
- No matching call on target side

Request-Based MPI RMA operations (MPI3.0,11.3.5)

Similar syntax

- Start with **MPI_R**…
- A request handle is added
  - **MPI_RPUT**
  - **MPI_RGET**
  - **MPI_RACCUMULATE**

Only for passive target communication

C/C++

```
int MPI_Rput(void* origin_addr, int origin_count,
          MPI_Datatype origin_type, int target_rank,
          MPI_Aint target_disp, int target_count,
          MPI_Datatype target_type, MPI_Win win,
          MPI_Request *req)
```

Further MPI RMA operations (MPI3.0,11.3.4)

New in MPI3.0

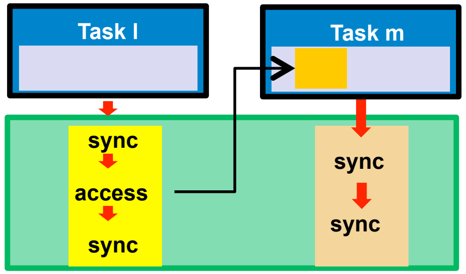
Further MPI RMA operations are available

- **MPI_PUT_ACCUMULATE**, **MPI_RPUT_ACCUMULATE**
- **MPI_GET_ACCUMULATE**, **MPI_RGET_ACCUMULATE**
- **MPI_FETCH_AND_OP**
- **MPI_COMPARE_AND_SWAP**

Synchronization schemes

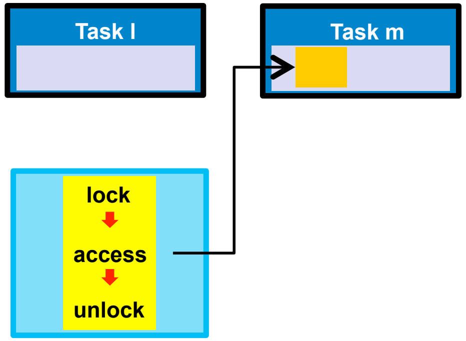
Active target synchronization (ATC): origin and target participate equally in synchronizing the RMA operations.

- Collective synchronization with fence
- General active target synchronization (GATS)

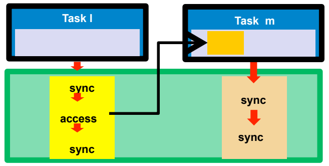


Passive target synchronization (PTC): target process is not explicitly taking part in the synchronization of the accessing RMA operation.

- Synchronization with locks

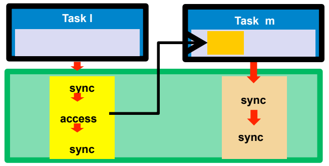


ATC: Synchronization with fence



- Collective call on communicator used for window creation
- Contains an implicit barrier
- Data access has to occur between two fence calls
- Written and read data is only accessible after completing fence
  - Local and remote accesses must not occur between the same fence calls
- Access and exposure epoch matching is done automatically

ATC: Synchronization with fence



C/C++

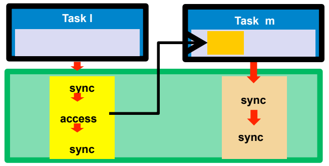
```
int MPI_Win_fence(int assert, MPI_Win win)
```

Fortran

```
MPI_WIN_FENCE(ASSERT, WIN, IERROR)
  INTEGER :: ASSERT, WIN, IERROR
```

- All processes of that window participate in the synchronization
- All-to-all communication pattern on the communicator
- May be too expensive if a sparse communication pattern is used
- **ASSERT** may be used for optimization (**ASSERT=0** is always valid, **MPIS3.0, 11.5.5**)

ATC: General active target synchronization (GATS)



- Pairwise synchronization of processes on subgroups of communicator used for window definition
- Individual calls for access and exposure epochs
  - MPI_Win_start / MPI_Win_complete for access epoch
  - MPI_Win_post / MPI_Win_wait for exposure epoch
- Accesses to local data only after epoch is closed
  - Data read from remote processes (access epoch) is accessible after **MPI_Win_complete**
  - Data written by remote processes (exposure epoch) is accessible after **MPI_Win_wait**
- Mind the order of calls with process-local access and exposure epochs

GATS: Access epoch

C/C++

```
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
int MPI_Win_complete(MPI_Win win)
```

Fortran

```
MPI_WIN_START(GROUP, ASSERT, WIN, IERROR)
  INTEGER :: GROUP, ASSERT, WIN, IERROR
MPI_WIN_COMPLETE(WIN, IERROR)
  INTEGER :: WIN, IERROR
```

- Start opens an access epoch, in which any number of one-sided calls can be posted
- All one-sided calls may be nonblocking, therefore data buffers are accessible only after completion of the access epoch
- **GROUP** must contain all processes that opened an exposure epoch for the local processes (MPI groups: **MPIS3.0, 6.3**)

GATS: Exposure epoch

C/C++

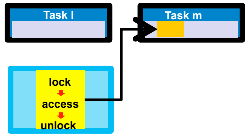
```
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
int MPI_Win_wait(MPI_Win win)
```

Fortran

```
MPI_WIN_POST(GROUP, ASSERT, WIN, IERROR)
  INTEGER :: GROUP, ASSERT, WIN, IERROR
MPI_WIN_WAIT(WIN, IERROR)
  INTEGER :: WIN, IERROR
```

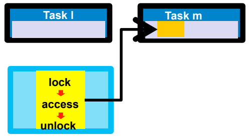
- Post starts an exposure epoch on win for accesses of all processes in **GROUP**
- Post only has local dependencies and returns when the exposure epoch is set up
- Wait ends an exposure epoch and waits for acknowledgements of all processes in group, regardless of actual accesses

PTC: General remarks



- Explicit synchronization and RMA operations only on the origin process
- Local and remote accesses need to be embraced by calls to **MPI_Win_lock** and **MPI_Win_unlock**
  - Needed to ensure serial consistency of memory updates
- Shared and exclusive locks available
- Order of accesses is not guaranteed and has to be handled otherwise
- Lock and any number of following RMA operations are allowed to be nonblocking

PTC: Lock and Unlock



C/C++

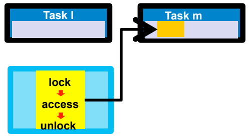
```
int MPI_Win_lock(int lock_type, int rank, int assert,
                 MPI_Win win)
int MPI_Win_unlock(int rank, MPI_Win win)
```

Fortran

```
MPI_WIN_LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR)
  INTEGER :: LOCK_TYPE, RANK, GROUP, ASSERT, WIN, IERROR
MPI_WIN_UNLOCK(RANK, WIN, IERROR)
  INTEGER :: RANK, WIN, IERROR
```

- **LOCK_TYPE** can be either **MPI_LOCK_EXCLUSIVE** or **MPI_LOCK_SHARED**
- Lock is set on a specific process identified by **RANK**
    - **RANK** is relative to the communicator used to define **WIN**

PTC: Lock_all and Unlock_all



C/C++

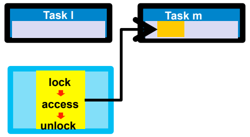
```
int MPI_Win_lock_all(int assert, MPI_Win win)
int MPI_Win_unlock_all(MPI_Win win)
```

Fortran

```
MPI_WIN_LOCK_ALL(ASSERT, WIN, IERROR)
  INTEGER :: ASSERT, WIN, IERROR
MPI_WIN_UNLOCK_ALL(WIN, IERROR)
  INTEGER :: RANK, WIN, IERROR
```

- Starts a shared access epoch from origin to all ranks
- *Not* a collective operation
- **Assert** can be used to provide additional information to optimize performance (**assert=0** is always valid) **MPI3.0, 11.5.5**

PTC: Flush



C/C++

```
int MPI_Win_flush (int rank, MPI_Win win)
int MPI_Win_flush_all(MPI_Win win)
```

Fortran

```
MPI_WIN_FLUSH(RANK, WIN, IERROR)
  INTEGER :: RANK, WIN, IERROR
MPI_WIN_FLUSH_ALL(WIN, IERROR)
  INTEGER :: RANK, WIN, IERROR
```

- Completes all outstanding RMA operations of the calling process to *the target pro*cess on the specified window (**FLUSH**)
- All RMA operations of the calling process to *any* target on the specified window are completed (**FLUSH_ALL**)
- Communication is completed on return of the call
- For further calls see **MPI3.0, 11.5.4**

Adding/retrieving Information about Windows

Attributes **MPI3.0, 11.2.6**

- Attributes can be cached to Windows (e.g. memory model)

Group **MPI3.0, 11.2.6**

- The group of processes attached to a window can be retrieved with **MPI_WIN_GET_GROUP**

Info object **MPI3.0, 11.2.7**

- An Info object can be associated to windows with **MPI_WIN_SET_INFO** / **MPI_WIN_GET_INFO**

Further/Advanced MPI topics

Some MPI topics are beyond the scope of this talk

- Groups and communicators
  - Group management **(MPI3.0, 6.3)**
  - Communicator management **(MPI3.0, 6.4)**
  - Inter-communicators **(MPI3.0, 6.6)**
- Process topologies **(MPI3.0,7)**
- Error handling **(MPI3.0,8.3 – 8.5)**
- Process creation and management **(MPI3.0,10)**
- MPI I/O **(MPI3,0,13)**
- Tools support **(MPI3.0,14)**

- MPI extension for Blue Gene/Q (MPIX)

Summary

In this talk we discussed

- Hard- and software concepts
- A concept for design of parallel programs
- Basics of MPI and selected topics

To design and write parallel code with MPI: think!

- Analyze you algorithm
- What hardware the code should run on?
- What is already available (algorithms, libraries, …)?

Summary

When using MPI

- Avoid communication if possible
- Use as few resources as possible
- Provide as much information to MPI as possible
- Give MPI the freedom to optimize
- Check the MPI environment on the target system
  - Message transfer protocol (eager limit)
  - Switch for asynchronous communication

Summary

To optimize parallel code

→ See the next talk

References and Literature

[EG10] Edgar Gabriel, *Introduction to MPI IV –MPI derived datatypes*, Lecture COSC 4397 Parallel Computation, University of Houston (2010).

[IF95] I. Foster. Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering. Reading, MA: Addison-Wesley, 1995. http://www.mcs.anl.gov/~itf/dbpp/

[MJQ04] M. J. Quinn. *Parallel Programming in C with MPI and OpenMP*, New York, NY: Mc Graw Hill, 2004.

[MPI] The MPI Forum. *MPI: A Message-Passing Interface Standard*, Version 3.0 (2012). http://www.mpi-forum.org/

[RR] Rolf Rabenseifner, *Optimization of MPI Applications*, University of Stuttgart High-Performance Computing-Center Stuttgart (HLRS)

[WG99] W. Gropp, E. Lusk, A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, 2nd ed., MIT Press, Cambridge (1999).

[WG99a] W. Gropp, E. Lusk, R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*, MIT Press, Cambridge (1999).

[WG05] William Gropp, Rusty Lusk, Rob Ross, and Rajeev Thakur, *Advanced MPI: I/O and One-Sided Communication,* Presentation at the SC2005 (2005) . http://www.mcs.anl.gov/research/projects/mpi/tutorial/advmpi/sc2005-advmpi.pdf

Thanks!