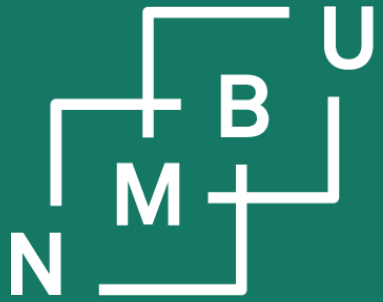
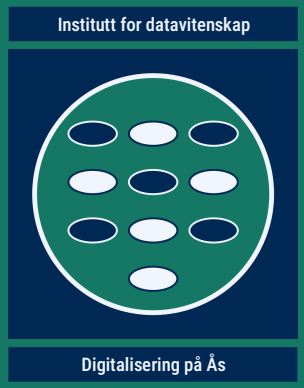


Norges miljø- og biovitenskapelige universitet



INF205
Resource-efficient programming

3 Concurrency

3.1 Parallel programming
3.2 Message passing interface
3.3 Collective communication
3.4 Concurrency-related concepts

Norwegian University of Life Sciences

MPI: Getting started

The target systems of MPI programs are often *clusters with thousands of cores*.

However, the code is not usually developed on these systems, but on the programmers' usual working environment. Even on a laptop/workstation, MPI makes you realize a *speedup*, since today these are all *multicore systems*.

To get started install an MPI environment, *e.g.*, **Open MPI** (package **openmpi**).

The **compiler command** becomes "**mpiCC …**" or similar (instead of "g++ …").
The *binary executable* produced by the compiler *will not run on its own*!

Instead: **mpirun -np** <number of processes> <executable>

This creates a number of parallel processes with ranks starting from 0.
Often the *process with rank 0* takes the role of the "master" or "scheduler".

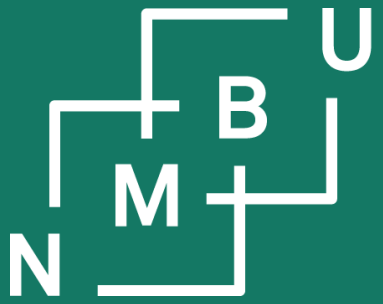
See also the Open MPI documentation: https://www.open-mpi.org/doc/v4.1/

MPI: Getting started … observations

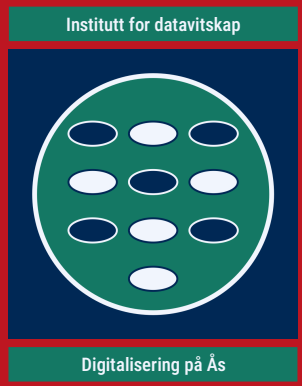
What we found during and after the tutorial sessions:

- Under Windows, it is possible to install MS MPI.
  - Compilation using MS MPI works together with Visual Studio; it worked in at least one case; in at least one, there were problems.
  - It also worked with Code::Blocks in at least one case
  - To execute the program (and vary the number of processes), a terminal is still needed; working with paths can be complicated
- Even with OpenMPI it can be necessary to use “mpic++”, not “mpiCC”.
- It can be necessary to install package “libopenmpi-dev” explicitly.
- macOS is POSIX compliant so that it mostly works exactly like Linux.
- Nobody is now using MVAPICH, all are with OpenMPI or MS MPI. (**?**)

Q: What other technical issues did you solve, what was unexpected?



Noregs miljø- og biovitskaplege universitet



3 Concurrency

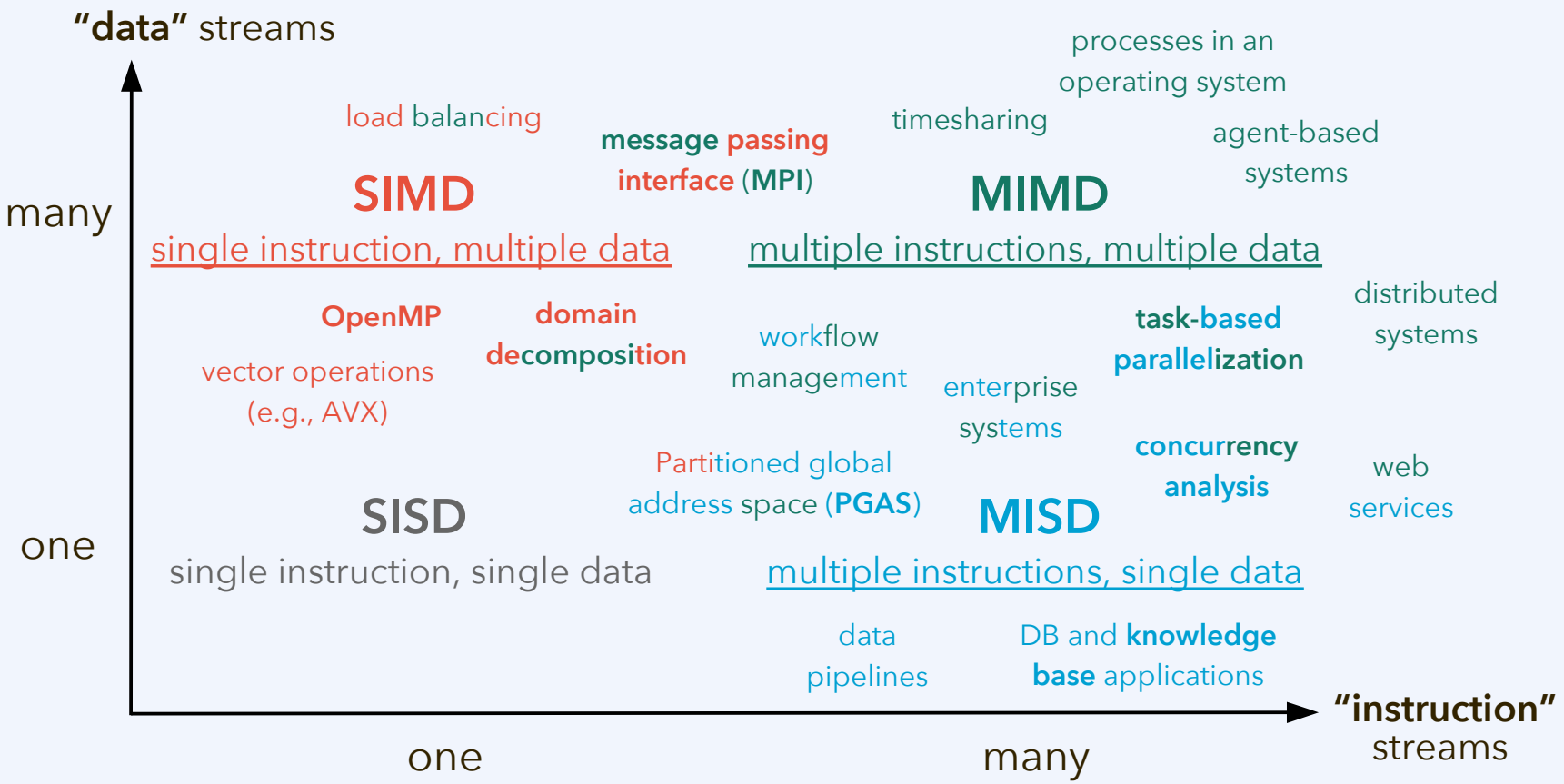
3.1 Parallel programming

INF205

26th October 2022

Paradigms of parallel programming

X-"instruction" x-"data" taxonomy as devised by Flynn:[1]



[1]M. J. Flynn, *IEEE Transact. Comput.* **C-21**(9): 940–960, doi:10.1109/tc.1972.5009071, **1972**.

Message passing

Message passing is the most general paradigm of parallel programming.

It can be carried out *irrespective whether* or not the *processes* (can also be called **ranks** in MPI) are executed on the same computing node and *have shared memory access*. It only assumes that they can exchange messages.

Challenges of message passing based parallelization:

- *Idle time* while processes are engaged in *blocking communication*.
- What if there are very many processes, do they all message each other?
- What if the recipient would already have had access to the data?
- Processes need to figure out what information they must give to others.

In high performance computing, message-passing based parallelization is usually done using **MPI**, the message passing interface.

Shared memory

In shared memory parallelization, all processes (in OpenMP, **threads**) have access to all the data. At each step, a task will usually work on part of the data.

The main advantage of shared memory parallelization is that it avoids sending messages from process to process. Instead, we may assume that all processes can immediately "see" all that was done by the others.

Challenges of shared memory parallelization:

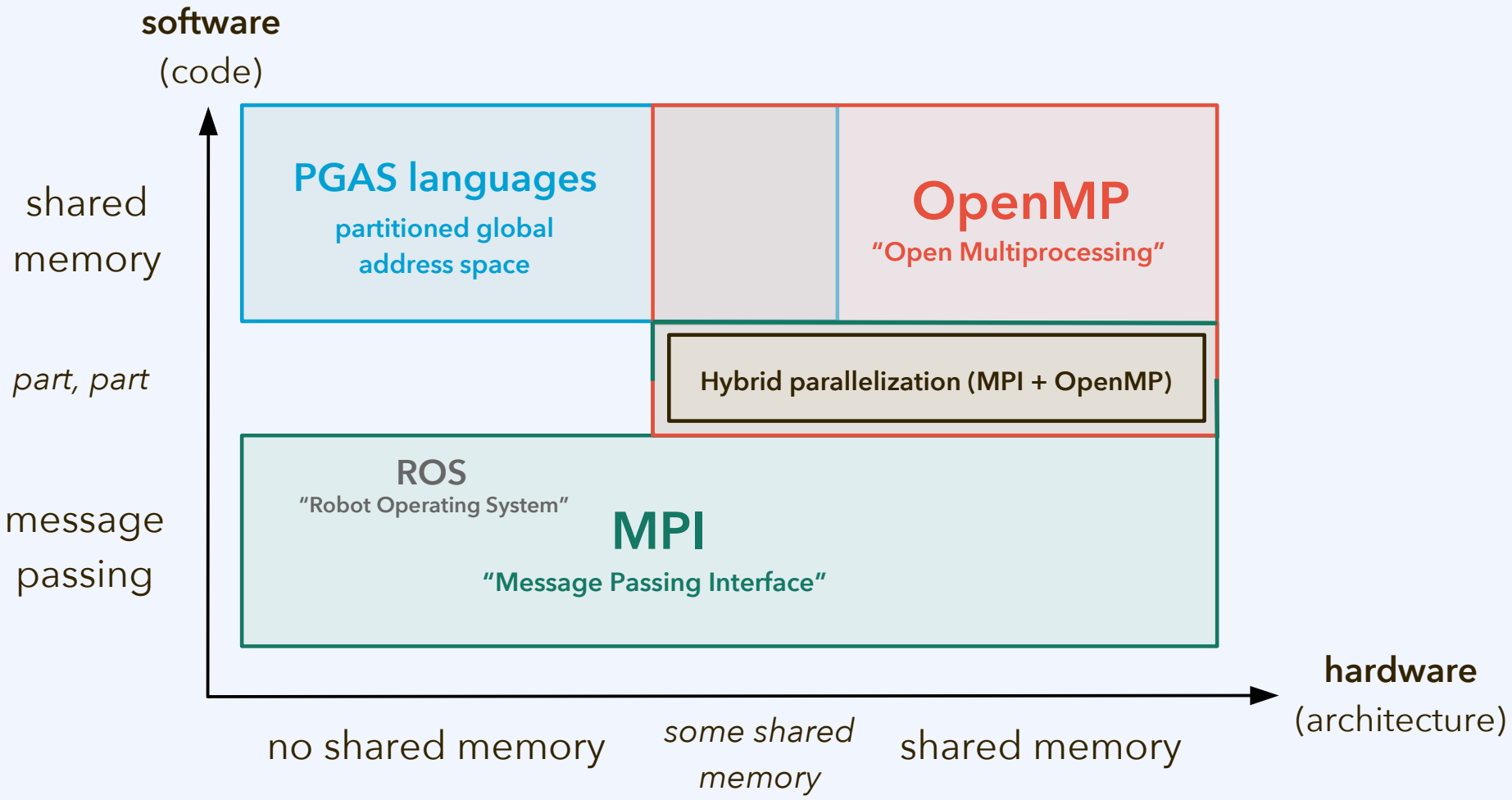
- Race conditions: Order of access to a data item influences the result.
- Synchronization delay from tools designed to avoid race conditions.

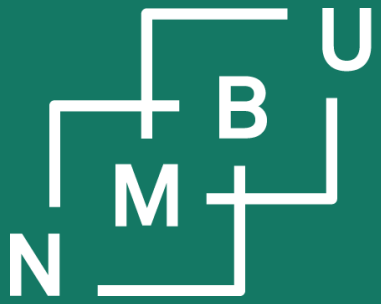
- What if at the hardware level, not all the CPUs can access all the data?
- What if they can, but some can access an item much faster than others?

True shared-memory parallelization is mostly done using **OpenMP**. *Partitioned global address spaces* are used to write code *as if* there was shared memory.

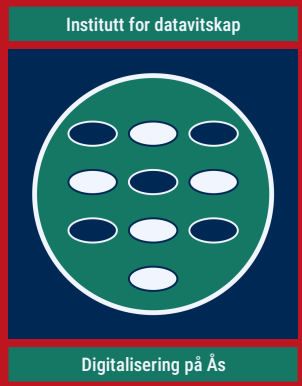
Norwegian University of Life Sciences

Software vs. hardware architecture





Noregs miljø- og biovitskaplege universitet



Programming project work

INF205

26th October 2022

INF205 programming project

What are the required actions?

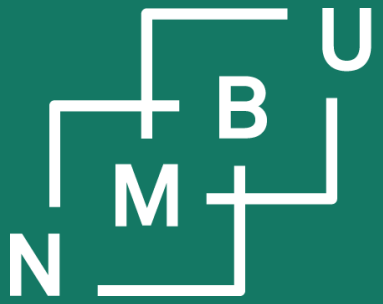
- Submit a group status report each week, from calendar week 43 to 47:
  - **Week 43:** Confirm composition of your group.
    Deadline: 31st October 2022.
  - **Week 44:** Make a final decision on the topic that your group works on.
    Deadline: 7th November 2022.
  - **Week 45:** Briefly summarize your design decisions and/or design alternatives regarding data structures, algorithms/performance, and concurrency.
    Deadline: 14th November 2022.
  - **Week 46:** Confirm the scheduled presentation date for your group.
    Deadline: 21st November 2022.
  - **Week 47:** Quantify each individual participant's contribution to the different aspects of the group work (for the individualized part of the grade).
    Deadline: 28th November 2022.
- **Week 48:** Submit your code and any additional documentation.
  Deadline: 5th December 2022.
- Give a presentation on your group work. (In weeks 48, 49, and 50.)

Grading scheme

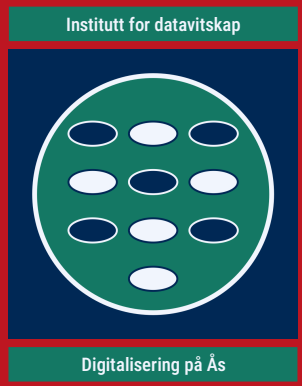
- 45% code/programming (group grade)
  - 15% data structures
  - 15% algorithms/performance
  - 15% concurrency
- 45% documentation (group grade)
  - 25% group status reports
  - 10% comments and code intelligibility
  - 10% other documentation (*e.g.*, submitted slides)
- 10% individual contribution

The group presentations are not graded, but a mandatory activity.
Slides may be submitted; if that is done, they are part of the documentation.

The group status reports (weeks 43 to 47) are also part of the documentation.



Noregs miljø- og biovitskaplege universitet



3 Concurrency

3.1 Parallel computing

3.2 Message passing interface

INF205

26th October 2022

Task decomposition by MPI rank

An MPI program needs to *initialize* and *finalize* the MPI environment.
Every process needs to *know its rank* (and, usually, the *number of processes*).

```
#include <mpi.h>

int main(int argc, char** argv)
{
  MPI_Init(&argc, &argv);

  int rank = 0;  // what is the rank of this process?
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  int size = 0;  // how many processes are there?
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  … // here comes the actual program

  MPI_Finalize();
}
```

(See the **mpi-primes** example code.)

Often the rank no. of a process, together with the number of processes, is already enough input to implement a basic parallelization scheme.

This is also the case for our prime-number test example:

5	7	11	13	17	19	23 …
0	0	1	1	2	2	3 …

From the documentation: "Open MPI accepts the C/C++ argc and argv arguments to main, but neither modifies, interprets, nor distributes them".

Task decomposition by MPI rank

Message passing: Whether or not processes *would* be able to access the same data, we operate under the assumption that there is *no shared memory*.

```
int main(int argc, char** argv)
{
  …
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  …
  for(n = 6*(rank+1) - 1; n < limit; n += 6*size)
    if(is_prime(n)) counted_primes++;
  for(n = 6*(rank+1) + 1; n < limit; n += 6*size)
    if(is_prime(n)) counted_primes++;

  …
}
```

(See the **mpi-primes** example code.)

It is a common strategy to parallelize the domain (parts of the problem) in terms of ownership of associated data.

Often, rank 0 is in charge of doing an overall evaluation based on all processes' data.

5	7	11	13	17	19	23 …
0	0	1	1	2	2	3 …

MPI send and receive

The most basic communication step is send/receive from one rank to another.

```
int MPI_Send(
  void* content, int count, MPI_Datatype type,
  int destination_rank, int tag, MPI_Comm handle
);
```

content is the address from which the source data are read; it is often an array, but can also be a pointer to a single data item

```
int MPI_Recv(
  void* buffer, int count, MPI_Datatype type,
  int source_rank, int tag, MPI_Comm handle,
  MPI_Status* status );
```

buffer is an address to which the received data can be written; the programmer needs to take care of memory allocation, *etc.*

count is the number of data items

type is their type as an MPI environment expression (*e.g.*, MPI_SHORT_INT, MPI_INT64_T, MPI_FLOAT, …)

tag is an identifier; send and receive must have the same tag

destination_rank is the rank of the process with the matching MPI_Recv(…) operation

source_rank is the rank of the process with the matching MPI_Send(…) operation

(Standard values from handle and status are MPI_COMM_WORLD and MPI_STATUS_IGNORE.)

MPI ping-pong example

*"increment counter, then read **1** item of type **int64_t** from **&counter**"*

*"send it **to rank 1**"*

```
if(rank == 0)
  MPI_Send(&(++counter), 1, MPI_INT64_T, 1, 1, MPI_COMM_WORLD);

if(rank == 1)
  MPI_Recv(
    &buffer, 1, MPI_INT64_T, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE
  );
```

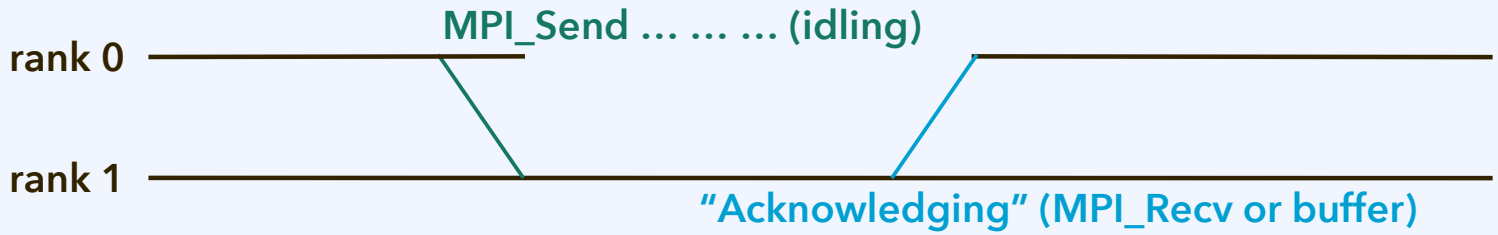
the **tag** for send and receive must be the same

*"write **1** item of type **int64_t** to **&buffer**"*

*"receive it **from rank 0**"*

One of the processes (say, rank 0) will reach the send/receive first.

Blocking communication: That process is **idle**, waiting for the other process.



Blocking communication

Depending on the MPI environment, with **MPI_Send**, the process may wait for the other process to reach its MPI_Recv (**synchronization**), *or* it could wait for a signal from the receiving process that data are written into a **receive buffer**.

- Both the **synchronized** and the **buffered** send implementation require waiting for a signal from the receiving process. The sending process remains blocked (idle). Hence, this is blocking communication.
- Use **MPI_Ssend**, with two 's', if you want to enforce synchronization.

There is also "local blocking" send, **MPI_Bsend**, which uses a send buffer.

Blocking communication: One process is **idle**, waiting for the other process.

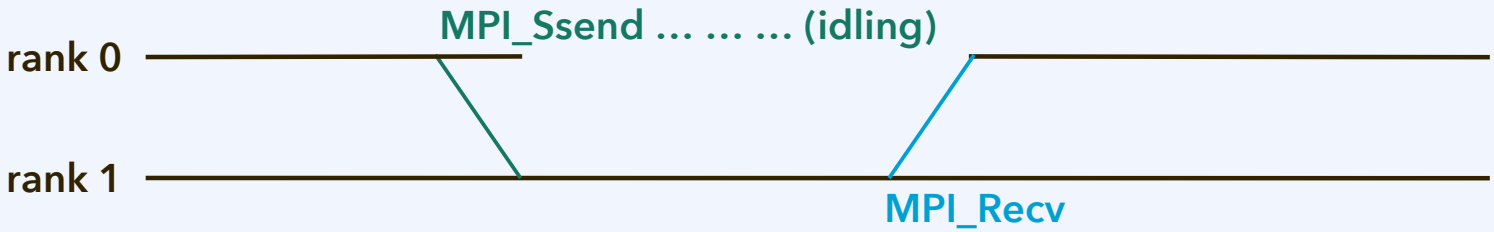
Non-blocking communication

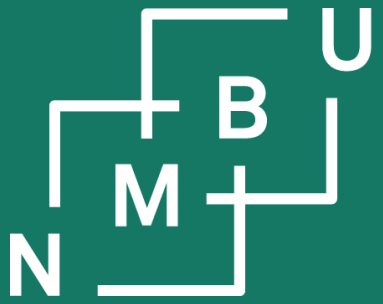
All the MPI communication operations also have **non-blocking variants**. Their names begin with "I" for "immediate". **MPI_Isend** and **MPI_Irecv** are like MPI_Send and MPI_Recv, but return immediately. They also create a "**handle**".

MPI_Test informs us if the action related to the handle has already completed.

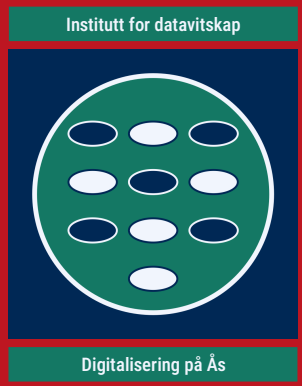
Non-blocking communication: Return immediately, **work in background** later.

Blocking communication: One process is **idle**, waiting for the other process.





Noregs miljø- og biovitskaplege universitet



3 Concurrency

3.1 Parallel computing

3.2 Message passing interface

3.3 Collective communication

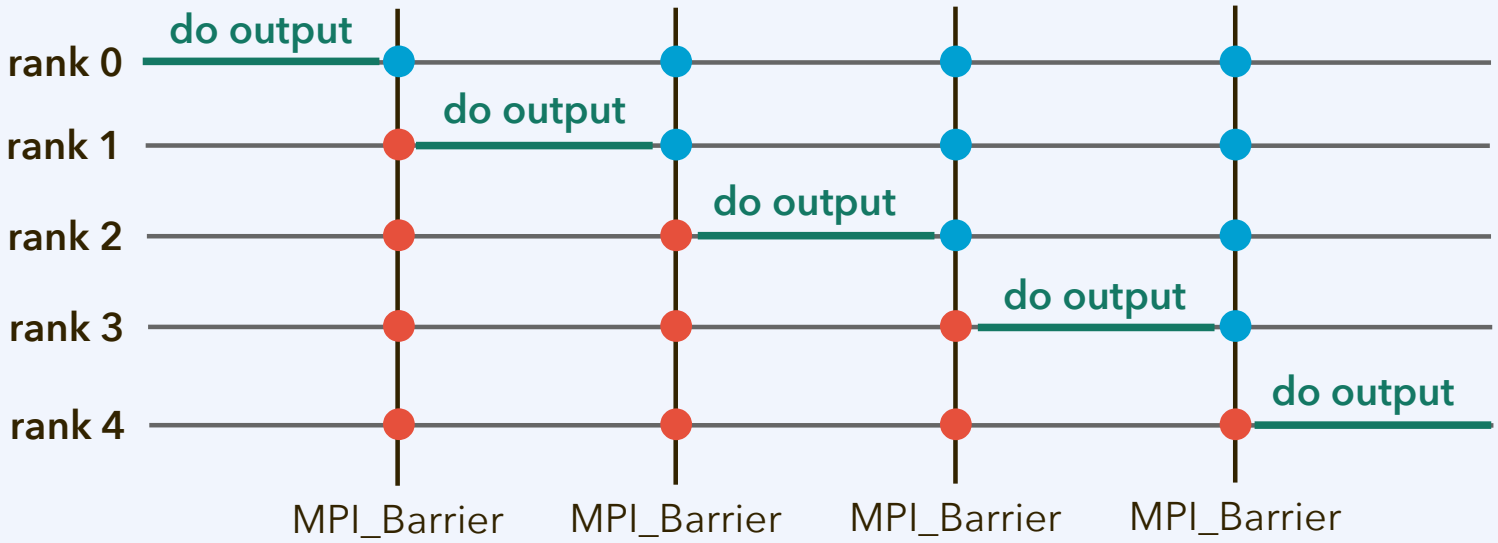
INF205

26th October 2022

Synchronization

MPI_Barrier(comm**)** enforces **synchronization** between all processes.

Example: Make all processes output some array content in order.




```
for(int i = 0; i < rank; i++) MPI_Barrier(MPI_COMM_WORLD);
std::cout << ...;
for(int i = rank; i < size; i++) MPI_Barrier(MPI_COMM_WORLD);
```

Collective communication

Send/receive is done from *one sender* process to *one recipient* process.
In a **collective communication** step, *all the MPI ranks participate* jointly.

- **Broadcast:** MPI_Bcast(buffer, count, type, root, handle)
  After the broadcast, *all processes' buffers* contain the value that used to be in the buffer of the root process. Rank 0 is often used as the root process.

- **Scatter:** MPI_Scatter(content, count, type, buffer, count, type, root, handle)
  Like broadcast, but *content* is *split (scattered) over the recipients' buffers*.

- **Reduce:** MPI_Reduce(content, buffer, count, type, *operation*, root, handle)
  Content from all the processes is *aggregated* into the buffer of the root process. For example, add up all the values (with *MPI_SUM* as *operation*).

- **Gather:** MPI_Gather(content, count, type, buffer, count, type, root, handle)
  The gather operation is the *opposite of scatter*. Split content from all processes is written into one big buffer at the root process.

Broadcast and scatter

"**content** has space for **15 character** items"

"take original content from **rank 0**"

See example "**collective-communication**".

Broadcast operation:

- **MPI_Bcast(content, 15, MPI_CHAR, 0, MPI_COMM_WORLD)**

Scatter operation:

- **MPI_Scatter(content, 3, MPI_CHAR, local_chunk, 3, MPI_CHAR, 0, …)**

"split up **content** into messages containing **3 character** items"

"receive **3 character** items and write them to **local_chunk**"

```
Initalizing char content[15].
      rank 0: 'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' 'l' 'm' 'n' 'o'
      rank 1: '' '' '' '' '' '' '' '' '' '' '' '' '' '' ''
      rank 2: '' '' '' '' '' '' '' '' '' '' '' '' '' '' ''
      …
Broadcasting content[15].
      rank 0: 'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' 'l' 'm' 'n' 'o'
      rank 1: 'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' 'l' 'm' 'n' 'o'
      rank 2: 'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' 'l' 'm' 'n' 'o'
      …
Scattering content[15] to local_chunk[3].
      rank 0: 'a' 'b' 'c'
      rank 1: 'd' 'e' 'f'
      rank 2: 'g' 'h' 'i'
      …
```

Gather and reduce

See example "**collective-communication**".

Gathering operation (all ranks to the root rank):

- **MPI_Gather(local_chunk, 3, MPI_CHAR, content, 3, MPI_CHAR, 0, …)**

Scatter operation (all ranks to the root rank):

- **MPI_Reduce(local_chunk, reduced, 3, MPI_BYTE, MPI_MAX, 0, …)**

```
Scattering content[15] to local_chunk[3].
      rank 0: 'a' 'b' 'c'
      rank 1: 'd' 'e' 'f'
      rank 2: 'g' 'h' 'i'
      …
Gathering using MPI_Gather.
      rank 0: 'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' 'l' 'm' 'n' 'o'
      rank 1: '' '' '' '' '' '' '' '' '' '' '' '' '' '' ''
      rank 2: '' '' '' '' '' '' '' '' '' '' '' '' '' '' ''
      …
Reducing local chunks into 'reduced' using MPI_Reduce with MPI_MAX.
      rank 0: 'm' 'n' 'o'
      rank 1: '' '' ''
      rank 2: '' '' ''
      …
```

Name	Meaning
MPI_MAX	**maximum**
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical xor
MPI_BXOR	bit-wise xor
MPI_MAXLOC	max value, location
MPI_MINLOC	min value, location

Allgather and allreduce

See example "**collective-communication**".

Gathering operation (all ranks to all ranks):

- **MPI_Allgather(local_chunk, 3, MPI_CHAR, content, 3, MPI_CHAR, …)**

Scatter operation (all ranks to all ranks):

- **MPI_Allreduce(local_chunk, reduced, 3, MPI_BYTE, MPI_MAX, …)**

```
Scattering content[15] to local_chunk[3].
      rank 0: 'a' 'b' 'c'
      rank 1: 'd' 'e' 'f'
      rank 2: 'g' 'h' 'i'
      …
Gathering using MPI_Allgather.
      rank 0: 'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' 'l' 'm' 'n' 'o'
      rank 1: 'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' 'l' 'm' 'n' 'o'
      rank 2: 'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' 'l' 'm' 'n' 'o'
      …
Reducing local chunks into 'reduced' using MPI_Allreduce with MPI_MAX.
      rank 0: 'm' 'n' 'o'
      rank 1: 'm' 'n' 'o'
      rank 2: 'm' 'n' 'o'
      …
```

Name	Meaning
MPI_MAX	**maximum**
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical xor
MPI_BXOR	bit-wise xor
MPI_MAXLOC	max value, location
MPI_MINLOC	min value, location

Discussion

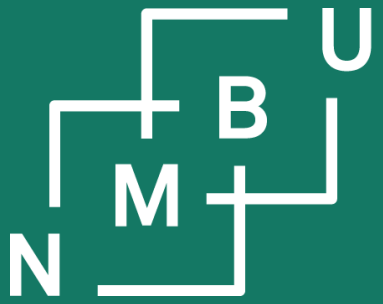
What MPI operation(s) would we use for the following?

- There are n processes (ranks).
- Each rank generates $k = 65536$ floating-point random numbers between 0 and 1.
- Now there are $k \cdot n$ random numbers. We would like all of them together to become a **unit vector** $\mathbf{x} = (x_0, \ldots, x_{kn-1})$ such that $\mathbf{x}^2 = 1$.
- We definitely don't want to send all the values to all processes, especially if k becomes even greater, but do this as efficiently as possible.

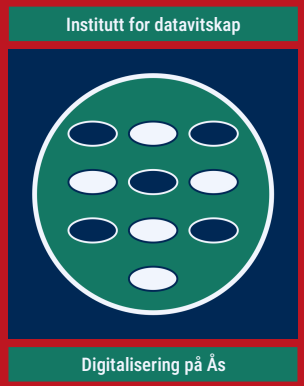
Discussed MPI operations

MPI_Send	MPI_Isend
MPI_Recv	MPI_Irecv
MPI_Wait	MPI_Test
MPI_Bcast	MPI_Ibcast
MPI_Scatter	MPI_Iscatter
MPI_Reduce	MPI_Ireduce
MPI_Gather	MPI_Igather
MPI_Allgather	MPI_Iallgather
MPI_Allreduce	MPI_Iallreduce

(See the **unit-vector** example for a code where the implementation is missing.)



Noregs miljø- og biovitskaplege universitet



3 Concurrency

INF205

26th October 2022

Discrete event systems

Terminology related to concurrency is often taken from the domain of **discrete event systems** (for example, *finite automata*). Adopting such an approach:

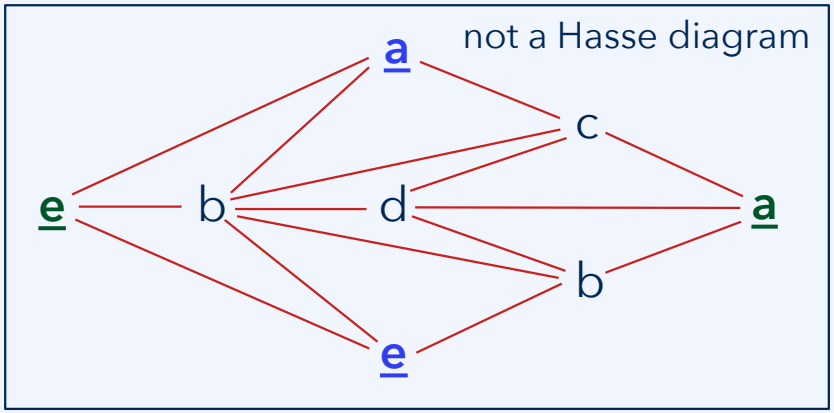
- A system can be in any of a finite number of **states** (or **configurations**).
- **Events**, or **transitions** between states, are thought of as instantaneous.
- A concurrent **process** is a (partially) temporally ordered set of events.

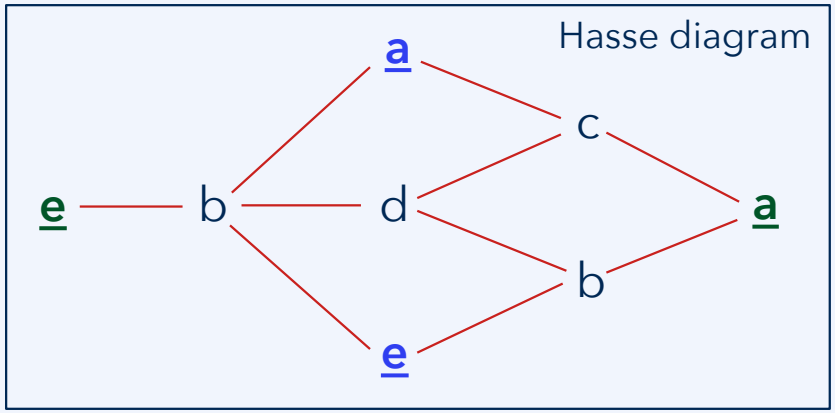
- Two events or transitions ***t*** and ***t'*** can be …
  - … **concurrent** whenever they are both enabled (*i.e.*, both can occur), one does not inhibit the other, and ***t*·*t'*** has the same outcome as ***t'*·*t***; in other words, they are concurrent if we don't say which comes first.
  - … **causally dependent** if they both occur, and it is important to say which comes first, either because only one order is possible or because it will have an impact on the outcome.

- **Limitation:** This model cannot make two transitions strictly synchronous.

Diagrams for partially ordered sets

By convention, **Hasse diagrams** are often used to denote causal dependency of events. These diagrams remove *any* ***indirect*** *or* ***redundant*** *dependencies*:





Two events are **directly or indirectly causally dependent** if one is specified to occur (conclude) before the other occurs (begins). Above: **e** and **a** are indirectly dependent.

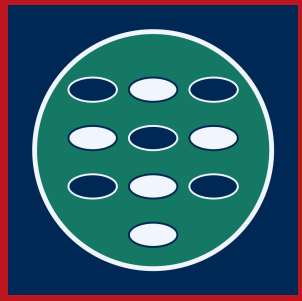
Events are **concurrent** if they are not directly or indirectly causally dependent – it does not matter which occurs first. Above: **e** and **a** are concurrent.

Attention

This notation only shows the **transitions** (events). The **states** (configurations) of the system are not shown.

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

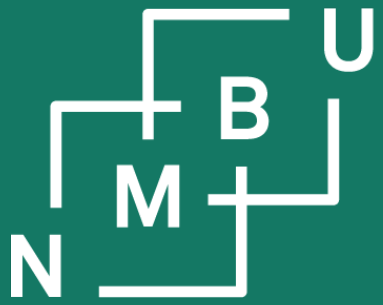


Digitalisering på Ås

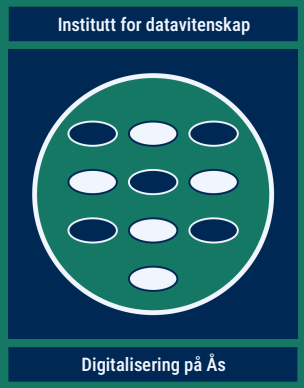
Conclusion

INF205

26th October 2022



Norges miljø- og biovitenskapelige universitet



INF205
Resource-efficient programming

3 Concurrency

3.1 Parallel programming
3.2 Message passing interface
3.3 Collective communication
3.4 Concurrency-related concepts