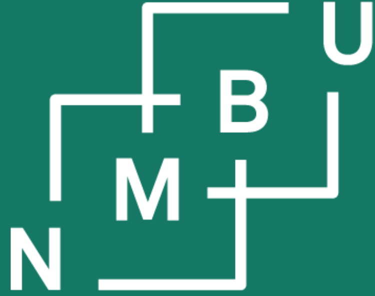
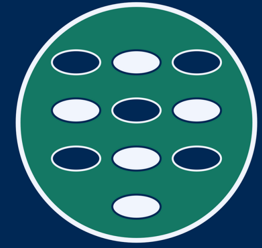


Norges miljø- og biovitenskapelige universitet



INF205
Resource-efficient programming

4 Concurrency

4.1 Parallel programming
4.2 Message passing interface
4.3 Domain decomposition
4.4 Robotics middleware
4.5 Concurrency theory
4.6 Parallel process models

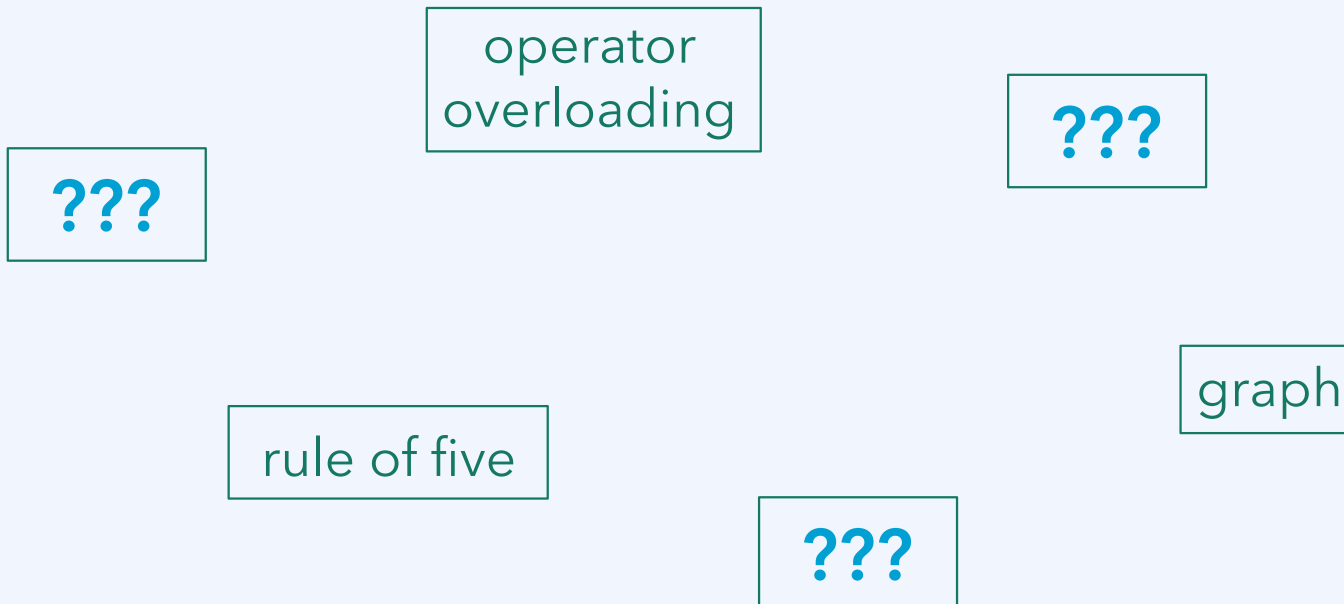
18th March 2024

Weekly glossary concepts

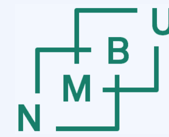
Norwegian University of Life Sciences

What are essential concepts from the previous lecture?

Let us include them in the **INF205 glossary**.[1]



[1]https://home.bawue.de/~horsch/teaching/inf205/glossary-en.html



Structure of the course

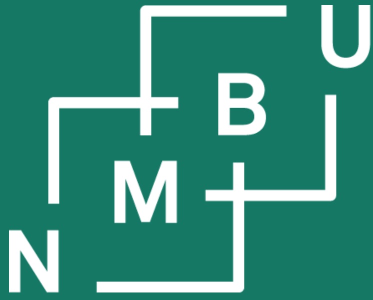
1) **Introduction** (week 6)
   - Getting started – the lecture last week.

2) **C++ programming** (weeks 7 and 8)
   - Essential features that make C/C++ different from Python; *e.g.*, dealing with memory allocation and deallocation explicitly, using pointers.

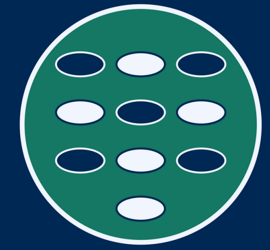
3) **Data structures** (weeks 9 to 11)
   - Linked data structures, containers, C++ standard template library.
   - Memory management for container data structures.

4) **Concurrency** (week 12 to 17)
   - MPI and ROS2 for parallel programming and concurrent processes.

5) **Production and optimization** (week 18 and 19)
   - Good practices and useful tools for programming projects.



Noregs miljø- og biovitskaplege universitet



4 Concurrency

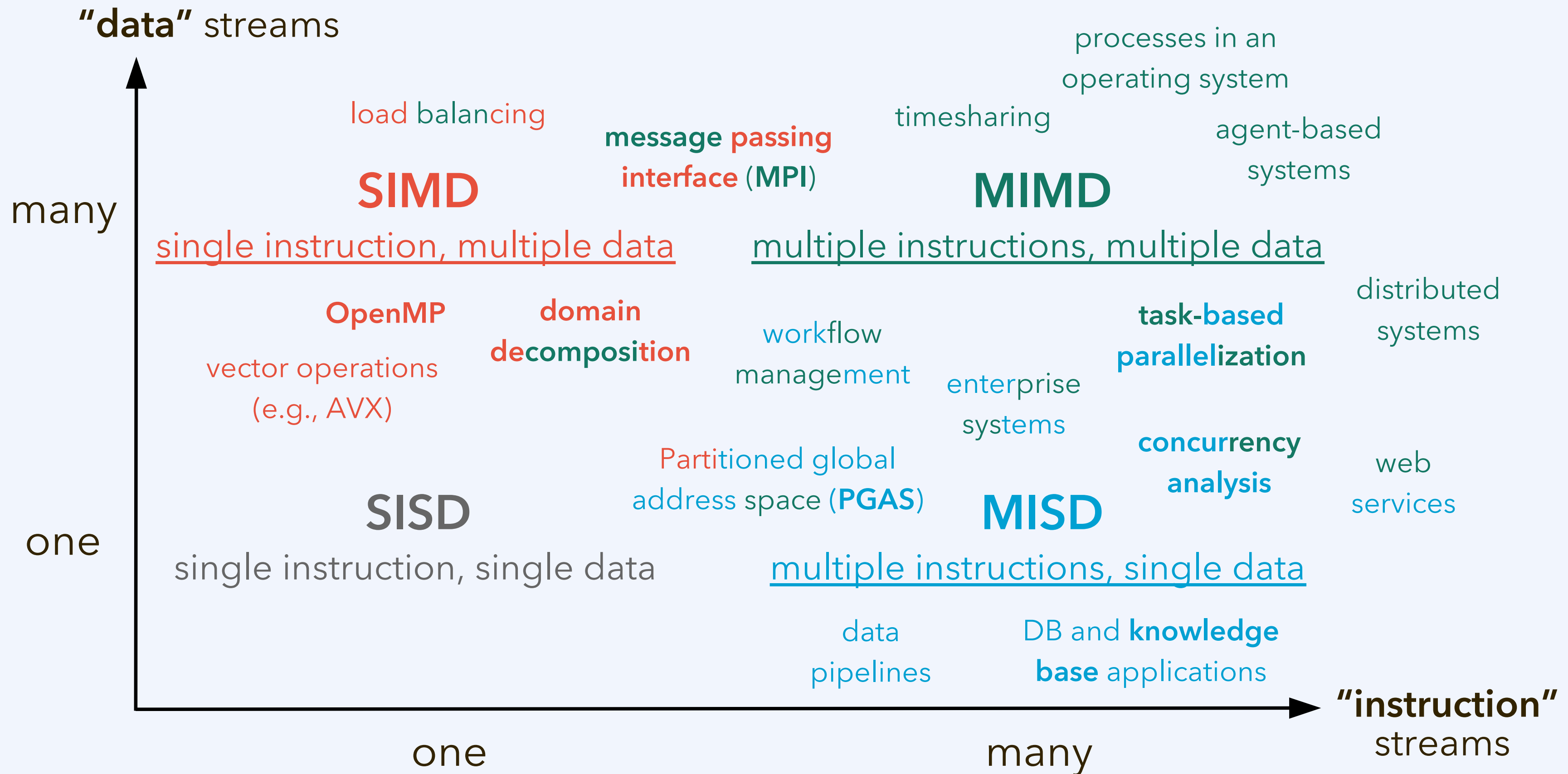
4.1 Parallel programming

INF205

18th March 2024

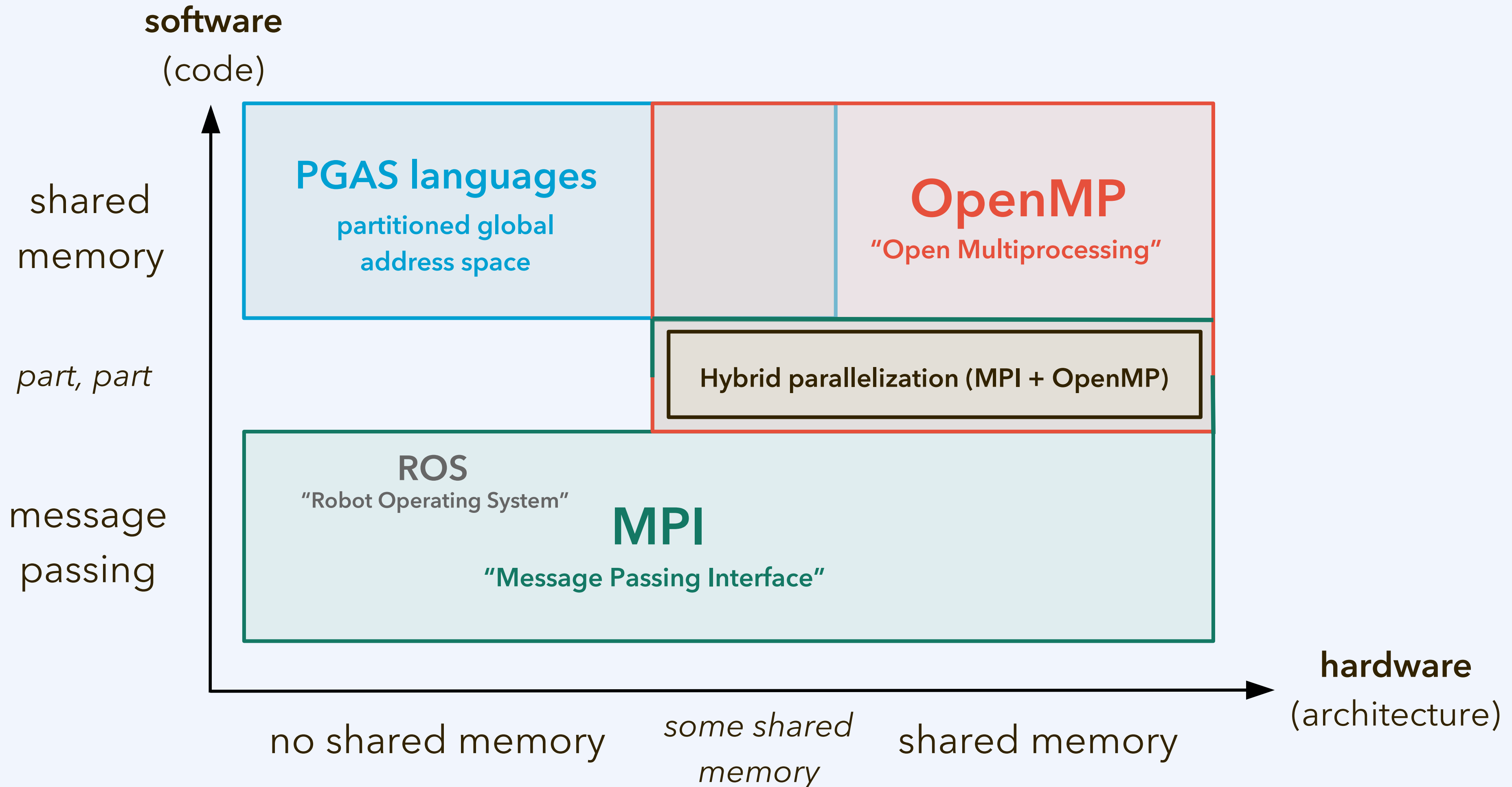
Paradigms of parallel programming

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**.
(shared on Canvas: **flynn_1972.pdf**)

Software vs. hardware architecture



Amdahl’s law

Assume a scenario where we can split a code into a fraction f that can be parellelized and the remainder $1 - f$ that is always sequential, never parallel.

Adding two vectors c[i] = a[i] + b[i], for i from 0 to 99 999, can be parallelized. Waiting for new instructions from the user cannot be parallelized.

Speedup is the *factor by which runtime decreases*; here, due to parallelization.

Amdahl’s law:

- Runtime with a single process is given by some $t_1 = (1 - f)t_1 + ft_1$.
- Now assume that we are parallelizing the code as perfectly as possible:
  - With n parallel processes, the runtime becomes $t_n = (1 - f)t_1 + ft_1/n$.
- Now assume that we have infinite computing resources at our hands:
  - With infinite parallel processes, the runtime becomes $t_\infty = (1 - f)\, t_1$.
- The **maximum possible speedup** for our code is $S_\infty = t_\infty/t_1 = 1 / (1 - f)$.

If $f = 99\%$ can be parallelized, speedup can never be greater than $S_\infty = 100$.

Parallel performance ("scaling") tests

In most cases, discussion of computational resources limits itself to "**space**" and "**time**." This is also motivated by tradition in theoretical computer science. In practice, then, *time usually becomes the main performance metric*, whereas *space becomes the main bottleneck* (memory access, communication, file I/O).

Strong scaling (**Amdahl**, *constant* problem size) on parallel architectures:

- Runtime reduction as number of processes increases (ideally, linear).
- Total CPU time increase as there are more processes (ideally, none).
- Rate of CPU operations (*e.g.*, FLOP/s) as fraction of peak performance.
- *Amdahl's law: Deterioration of performance at some point is inevitable.*

Weak scaling (**Gustafson**, *proportional* problem size) on parallel architectures:

- CPU time per problem size as problem and core usage are scaled up.
- Runtime increase during the scale-up.
- Rate of CPU operations (*e.g.*, FLOP/s) as fraction of peak performance.
- *Some algorithms and codes don't show a major decay in these metrics.*

Parallelization based on message passing

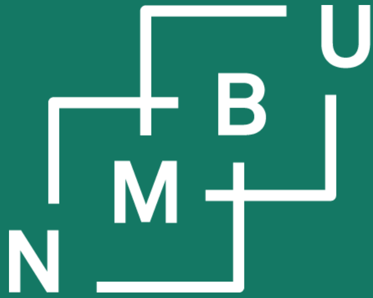
In high performance computing, message-passing based parallelization is usually done using **MPI**, the **message passing interface**.

Message passing is the most general paradigm of parallel programming.

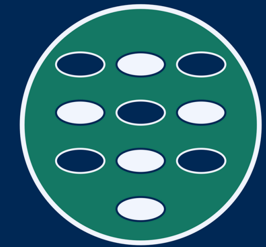
Message passing **does not require** that processes (also called **ranks** in MPI) are executed on the same computing node and have **shared memory access**. It only assumes that they can exchange messages.

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.



Noregs miljø- og biovitskaplege universitet



Course schedule and fourth worksheet

INF205

18th March 2024

Course schedule for the coming weeks

Week 12 (17th – 23rd March 2024):

- Monday, 18th March 2024:
  - 1st lecture on concurrency
  - 4th worksheet released
- Wednesday, 20th March 2024:
  - Tutorial session
  - Programming project opened

Week 13 (24th – 30th March 2024):

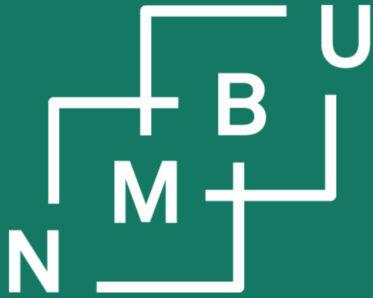
- No teaching
- Sign-up for presentation slots (fourth worksheet)

Week 14 (31st March – 6th April 2024):

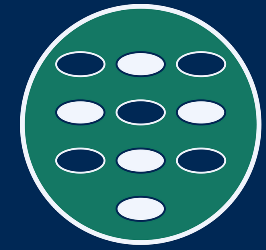
- No lecture (Easter Monday)
- Tuesday, 2nd April 2024: Submission deadline for the 4th worksheet
- Wednesday, 3rd April 2024: Discussion of the 4th worksheet

Week 15 (7th – 13th April 2024):

- Monday, 8th April 2024:
  - 2nd lecture on concurrency
  - 5th worksheet released
- Wednesday, 10th April 2024: Tutorial session



Noregs miljø- og biovitskaplege universitet

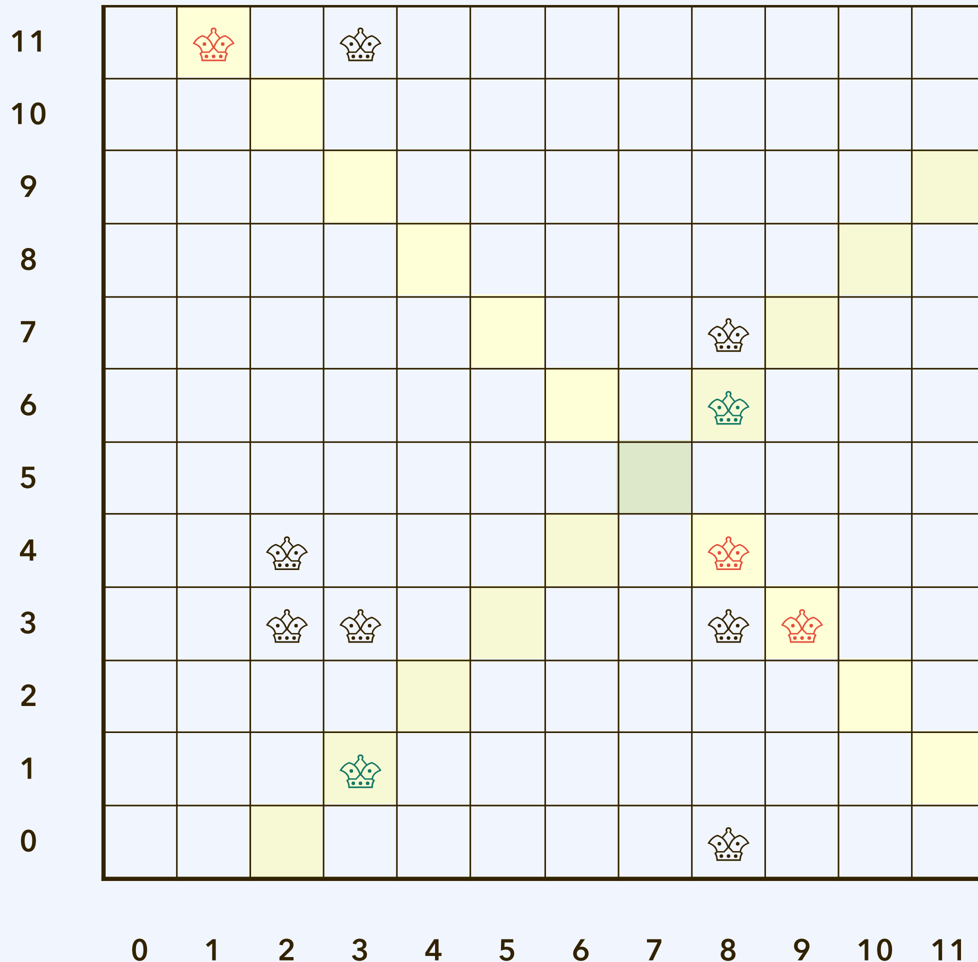


Info on the programming project

INF205

18th March 2024

Topic 1 code example: N-queens problem (variants)



./random-config-debug 12 12 12

2 with x = 2. (Contribution: 2).
3 with x = 3. (Contribution: 4).
5 with x = 8. (Contribution: 8).

4 with y = 3. (Contribution: 6).
2 with y = 4. (Contribution: 2).
2 with y = 11. (Contribution: 2).

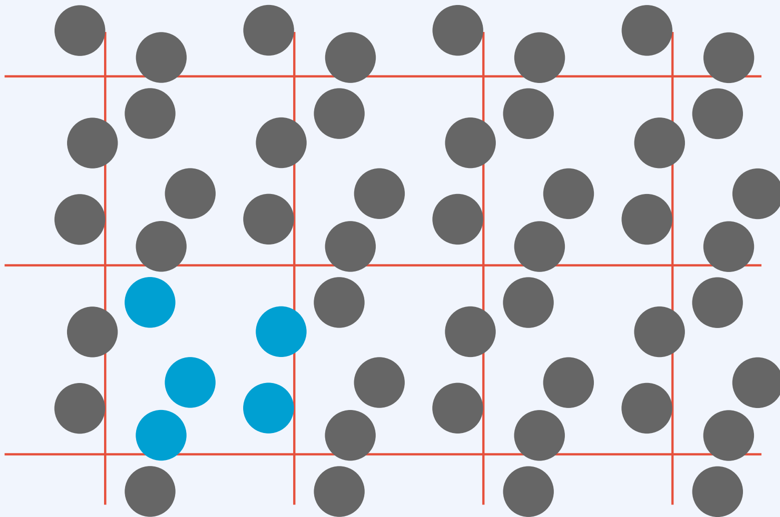
2 with x+y = 6. (Contribution: 2).
3 with x+y = 12. (Contribution: 4).
2 with x+y = 14. (Contribution: 2).

2 with x-y = 2. (Contribution: 2).
==
Threats counted: 34.

Example file: queens-count-threats.zip

Topic 2 code example: Configurations of spheres

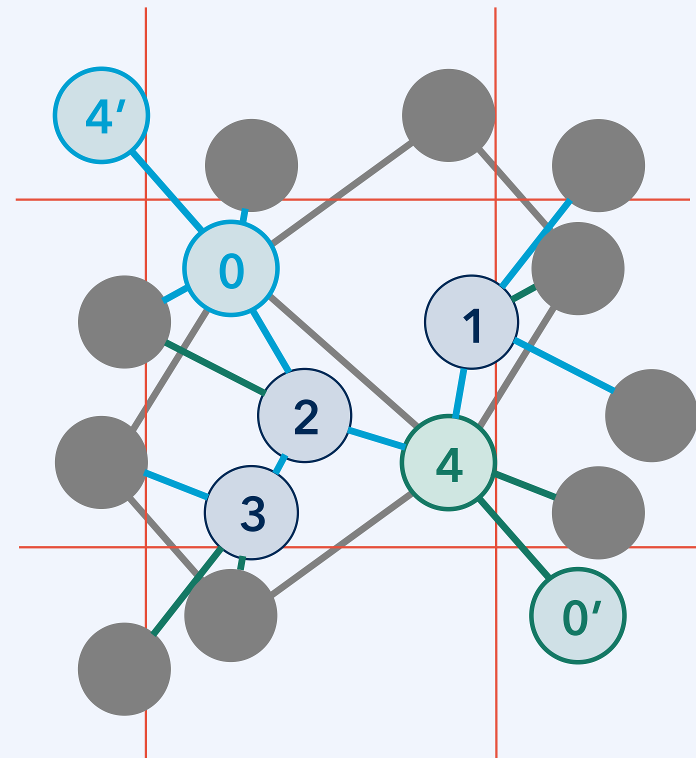
Periodic boundary condition (PBC)



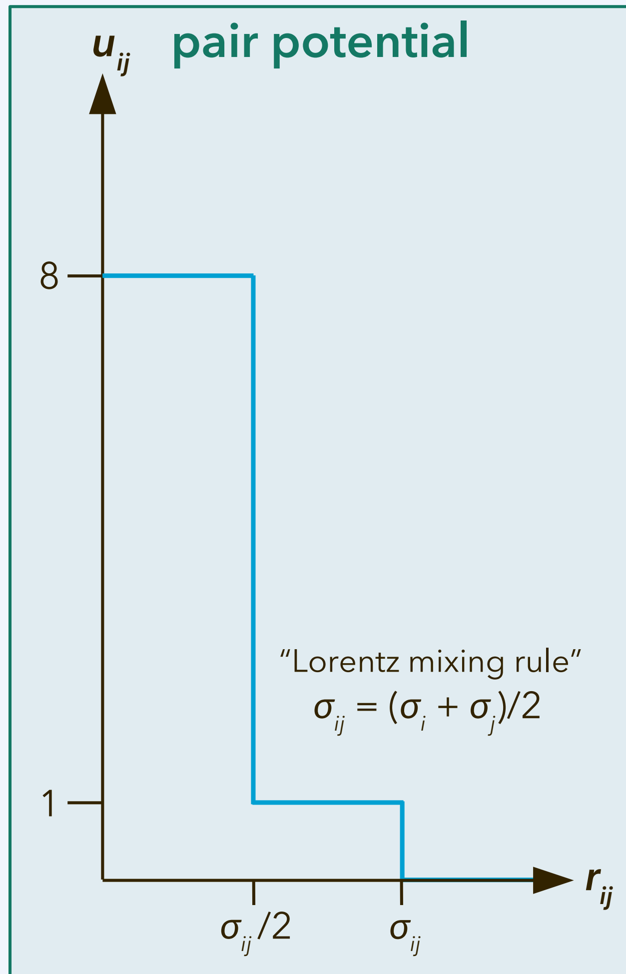
PBC: Assume that the simulation box repeats periodically in all directions.

MIC: Each particle interacts only with closest replica of each other particle.

Minimum image convention (MIC)



Topic 2 code example: Configurations of spheres

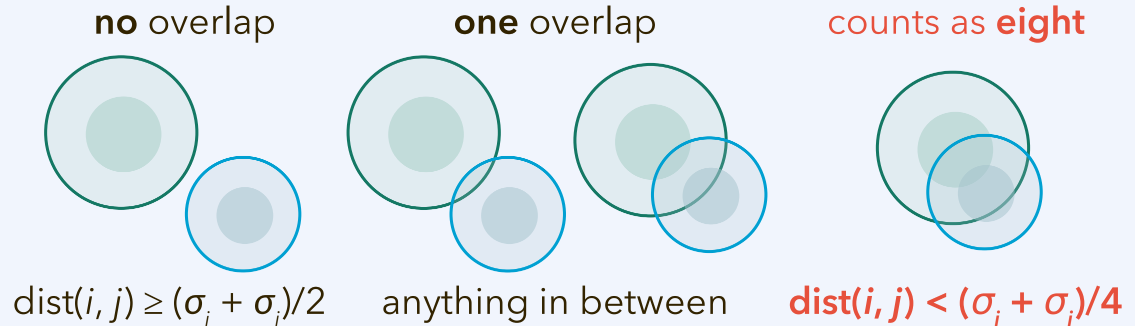



```
int Sphere::check_overlap(const Sphere* other, const double box_size[3]) const
{
   // square distance between the centre of i and the centre of j
   double square_distance = 0.0;
   for(int d = 0; d < 3; d++) {
      double dist_d = other->coords[d] - this->coords[d];

      // apply minimum image convention
      if(dist_d > 0.5*box_size[d]) dist_d -= box_size[d];
      else if(dist_d < -0.5*box_size[d]) dist_d += box_size[d];

      square_distance += dist_d*dist_d;
   }

   // is the square distance smaller than the square of the sum of radii?
   double sum_of_radii = 0.5 * (this->size + other->size);
   int overlap = 0;
   if(square_distance < 0.25*sum_of_radii*sum_of_radii) overlap = 8;  // soft shielding
   else if(square_distance < sum_of_radii*sum_of_radii) overlap = 1;  // normal overlap
   return overlap;
}
```



See implementation in **repulsive-spheres.zip**, sphere.cpp, line 51.

Other problems – and using libraries

How about other special-interest problems?

- It is a good idea to work on special problems that you are interested in.
- Provide a clear description as part of your submission to worksheet 4.
- We will need a well-defined benchmark, and discussions to specify it in a clear way, and I need to understand it well enough to grade it.

Should we use external libraries, or should we develop all from scratch?

- It is one of the learning outcomes to work with external libraries.
- But we have seen that even the STL can be sometimes beaten by simple bespoke code that you write yourself for a special purpose.
- With your project code you are meant to demonstrate what you have learnt. Your own development must not be totally trivial.

If you are reusing a complicated algorithm, data structure, or file format, going beyond the content of INF205, and there is a library, just use the library!

Reuse of external code

Are you legally allowed to use external code?

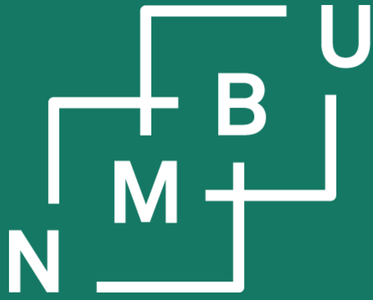
- You need a license; which is it? Check its terms and conditions.
  - Some licenses, even if they allow you to reuse the code and create derivative works, cannot be combined with each other.
  - For example, the GPL and CC NC licenses cannot be combined.
  - To alleviate this issue, libraries are often released under the LGPL.

How about the code examples from the INF205 lecture material?

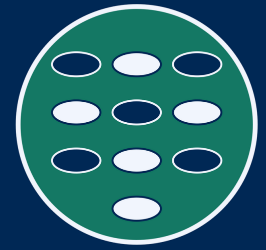
- Released under the conditions of the CC BY-NC-SA 4.0 License.

Would it not be plagiarism or fraud to submit others' material?

- It is, if you submit others' developments *as if they were your own*.
- If it is not absolutely clear from your submission that you are reusing somebody else's work (when you actually are), it may be a fraud attempt.
- That is also the case for the lecture material; it must be documented, *e.g.*, for clarity in case it goes to an appeals examiner (klagesensor).



Noregs miljø- og biovitskaplege universitet



4 Concurrency

4.1 Parallel programming

4.2 Message passing

INF205

18th March 2024

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://docs.open-mpi.org/en/v5.0.x/

MPI rank (own number) and size (total number)

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();
}
```

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".

Example file: mpi-primes.zip

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
);
```

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

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

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
  );
```

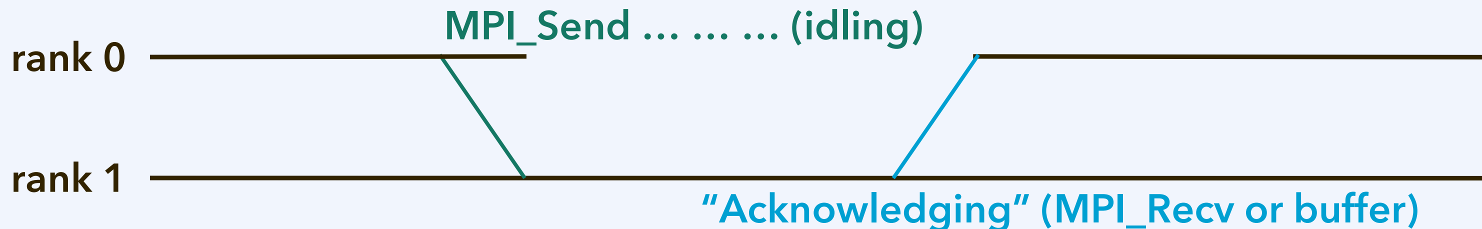
the **tag** for send and receive must be the same

*"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.



Example file: pingpong.zip

Stream-based serialization of data

Observation:

- It is not straightforward to unwrap more complex data structures.
- We were already using streams for serialization, in particular file I/O.
- The same stream serialization can be used to transfer objects via MPI.

If a **stringstream s** is used to store the data, the method **s**.str().c_str() can be used for sending a char array, *e.g.*, with MPI_Send.

Size in characters: **s**.str().size()   // +1 for '\0'

attention, pitfall!

overloaded operators

```
std::istream& operator>>(
  std::istream& is, Graph& g
) {
  g.in(&is);
  return is;
}
std::ostream& operator<<(
  std::ostream& os, const
Graph& g
) {
  g.out(&os);
  return os;
}
```

Prerequisite: The input and output methods (and operators) must be aligned.

Example file: graph.cpp (in **graph-stream.zip** archive)

Stream-based serialization of data

```
if(rank == 0) {
  // open in-filestream
  std::ifstream indata(argv[1]);

  // read graph object from file
  indata >> g;
  indata.close();

  // write into stringstream
  std::stringstream text << g;

  // inform recipient about content size
  message_size = text.str().size() + 1;
  MPI_Send(
    &message_size, 1, MPI_INT,
    1, 1, MPI_COMM_WORLD
  );

  // send content to recipient
  MPI_Send(
    text.str().c_str(), message_size,
    MPI_CHAR, 1, 2, MPI_COMM_WORLD
  );
}
```

message tag 1

message tag 2

```
if(rank == 1) {
  // get information about the content size
  MPI_Recv(
    &message_size, 1, MPI_INT, 0, 1,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE
  );

  // allocate buffer and receive the content
  char* buffer = new char[message_size]();
  MPI_Recv(
    buffer, message_size, MPI_CHAR,
    0, 2, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE
  );

  // write into stringstream
  std::stringstream text << buffer;
  delete[] buffer;
  buffer = nullptr;

  // read graph object from stringstream
  text >> g;
}
```

Example file: copy-graph-demo.cpp (in **graph-stream.zip** archive)

Collective communication

Norwegian University of Life Sciences

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.

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

Example file: collective-communication.zip

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

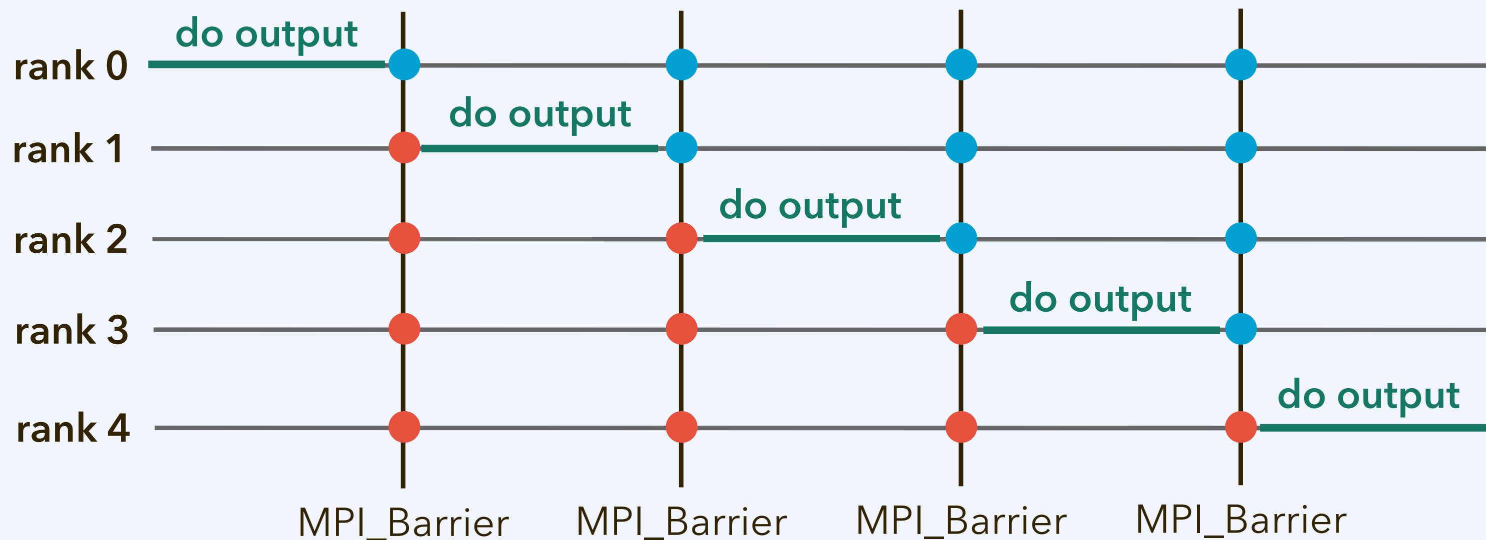
Example file: collective-communication.zip

Norwegian University of Life Sciences

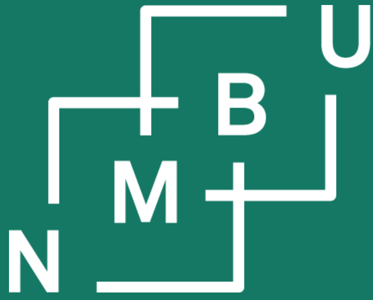
Synchronization

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

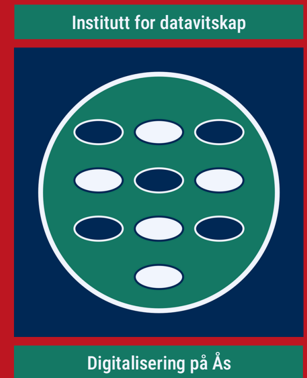
Example: Make all processes output some array content in order.




```
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);
```



Noregs miljø- og biovitskaplege universitet



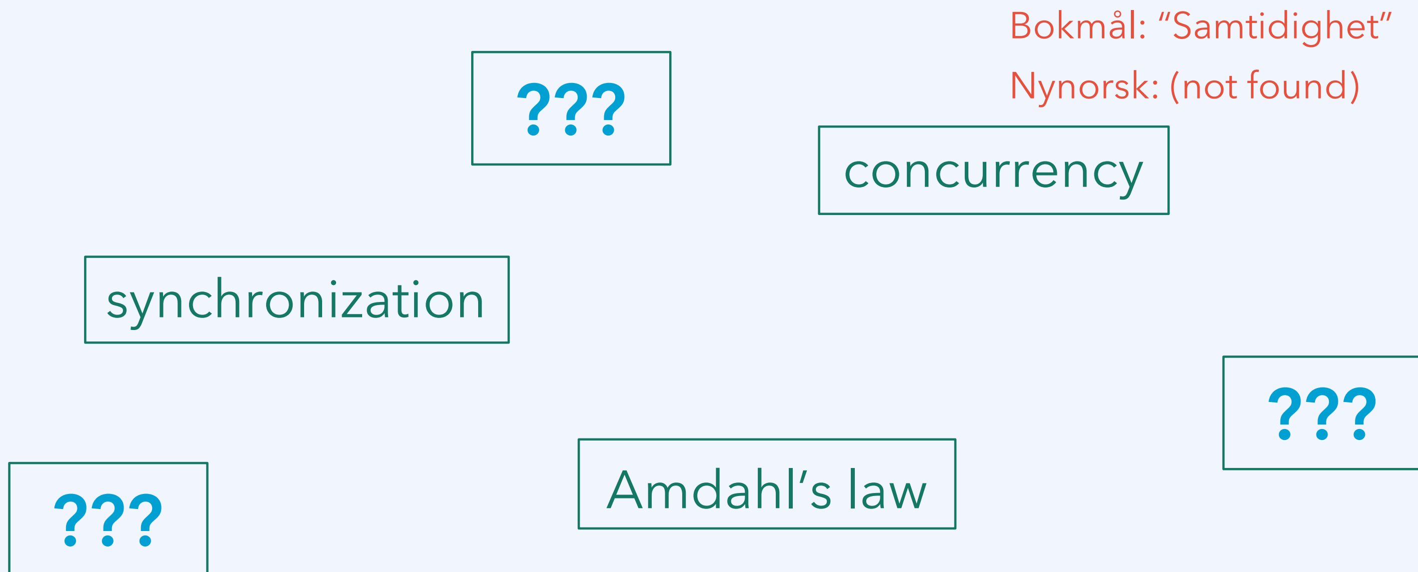
Conclusion

Norwegian University of Life Sciences

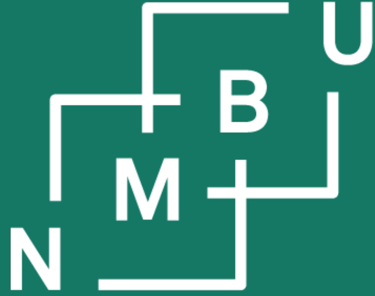
Weekly glossary concepts

What are essential concepts from this lecture?

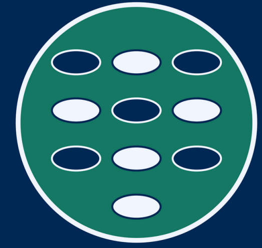
Let us include them in the **INF205 glossary**.[1]



[1] https://home.bawue.de/~horsch/teaching/inf205/glossary-en.html



Norges miljø- og biovitenskapelige universitet



INF205
Resource-efficient programming

4 Concurrency

4.1 Parallel programming
4.2 Message passing interface
4.3 Domain decomposition
4.4 Robotics middleware
4.5 Concurrency theory
4.6 Parallel process models