# Parallel computing
# MPI, OpenMP, using GPU

Claudia Blaas-Schenner

VSC Research Center, TU Wien

06/2023

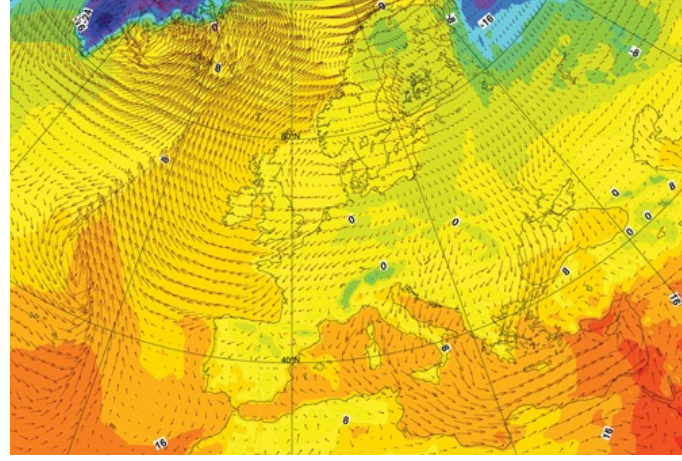
Univerza v Ljubljani

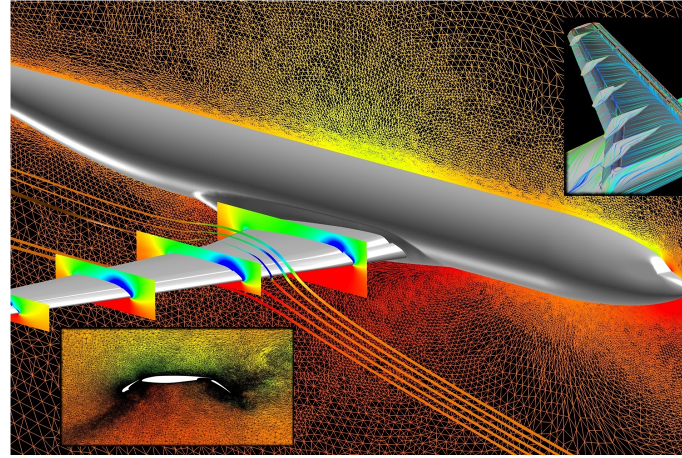
Co-funded by the Erasmus+ Programme of the European Union

This project has been funded with support from the European Commission.
This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

# HPC solves societal challenges

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP





*The days when scientists did not have to care about the hardware are over,*
*and so are the days when compute centers did not have to worry about the scientific application!*

source:
www.prace-ri.eu

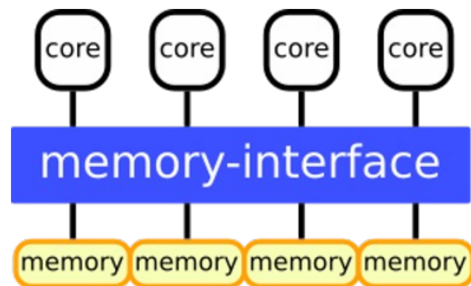
## remarkable repeated success stories:

- recurring core part of Nobel Prizes in Physics & Chemistry
- saving billions with better weather forecasting
- improving human health with genomics, personalized medicine
- 3-4% better fuel efficiency of aircraft & wind turbines every year
- disrupting communication, transportation and manufacturing
- design of future materials from scratch based on desired properties
- batteries & supercapacitors
- artificial intelligence, machine learning, sensors, open data

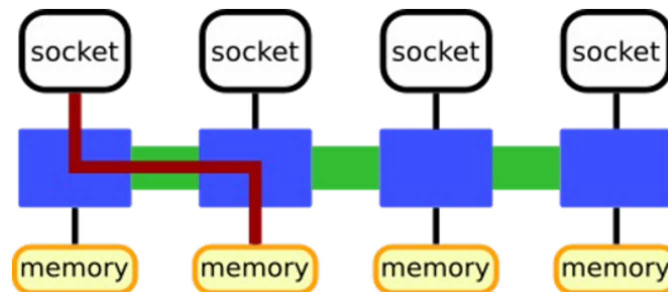
**OpenMP**: shared memory (socket, node)

**MPI**: distributed memory (socket, node, **cluster**)

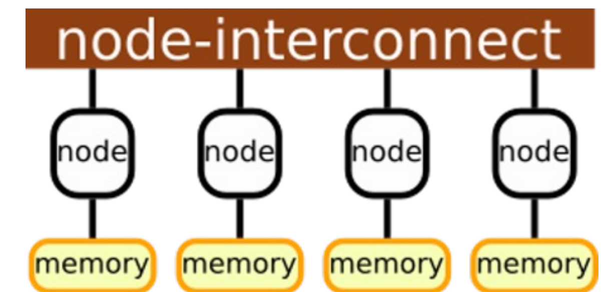
**CUDA**: accelerated nodes




**socket**
UMA (uniform memory access)
SMP (symmetric multi-processing)




**node**
ccNUMA (cache-coherent non-uniform ...)
**first touch, pinning!**



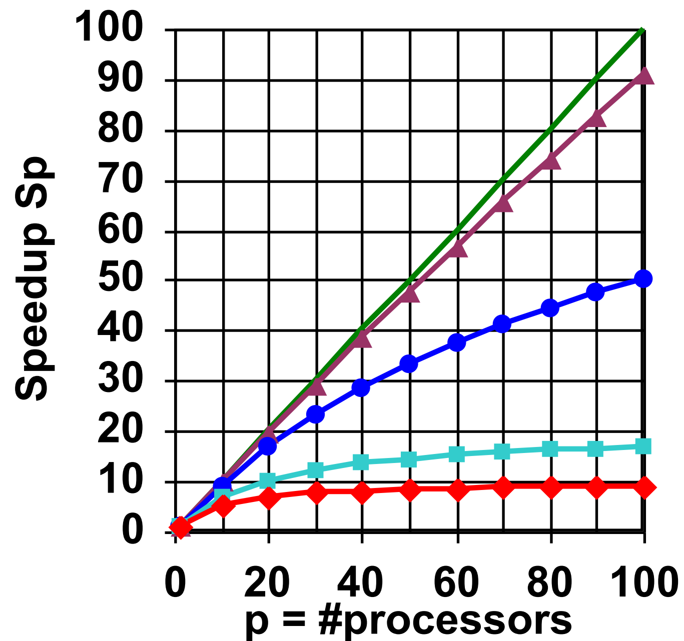

**cluster**
NUMA (non-uniform memory access)
fast access to own memory only

# Amdahl's law



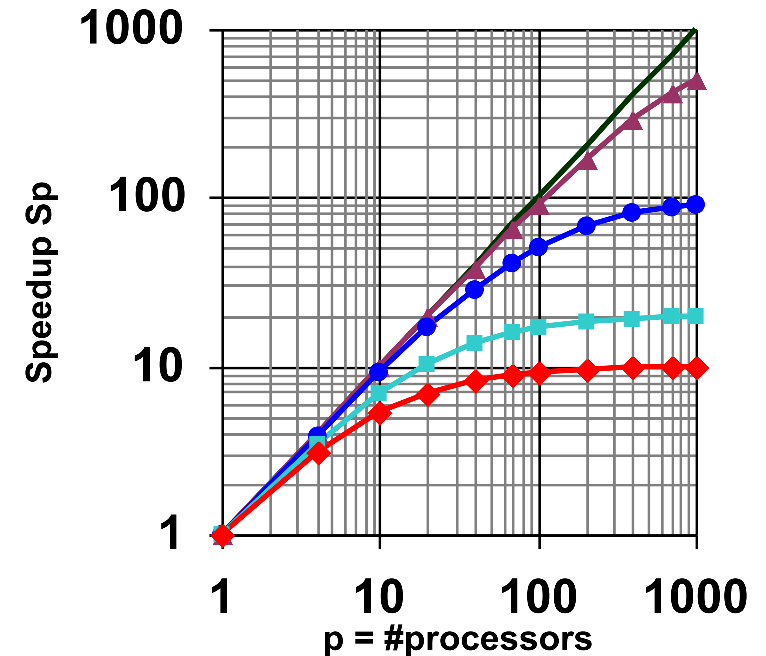
$T_{parallel,\,p} = f \cdot T_{serial} + (1\text{-}f) \cdot T_{serial} \,/\, p$ f ... sequential part of code **neglecting time for communication**

$S_p = T_{serial} \,/\, T_{parallel,\,p} = 1 \,/\, (f + (1\text{-}f) \,/\, p)$ Speedup is limited: $S_p < 1 \,/\, f$ **neglecting load imbalance**




**Sp = p (ideal speedup)**
**f=0.1% => Sp < 1000**
**f= 1% => Sp < 100**
**f= 5% => Sp < 20**
**f= 10% => Sp < 10**

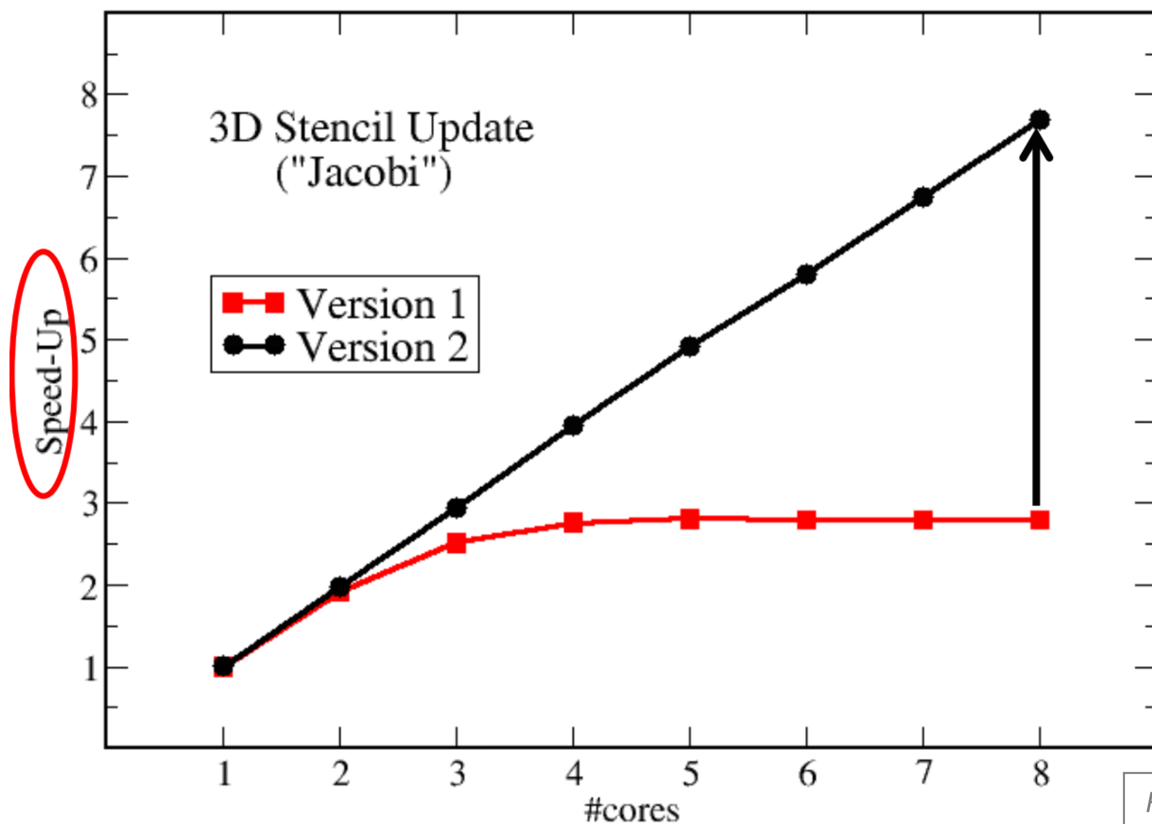
*Figures courtesy of Rolf Rabenseifner.*

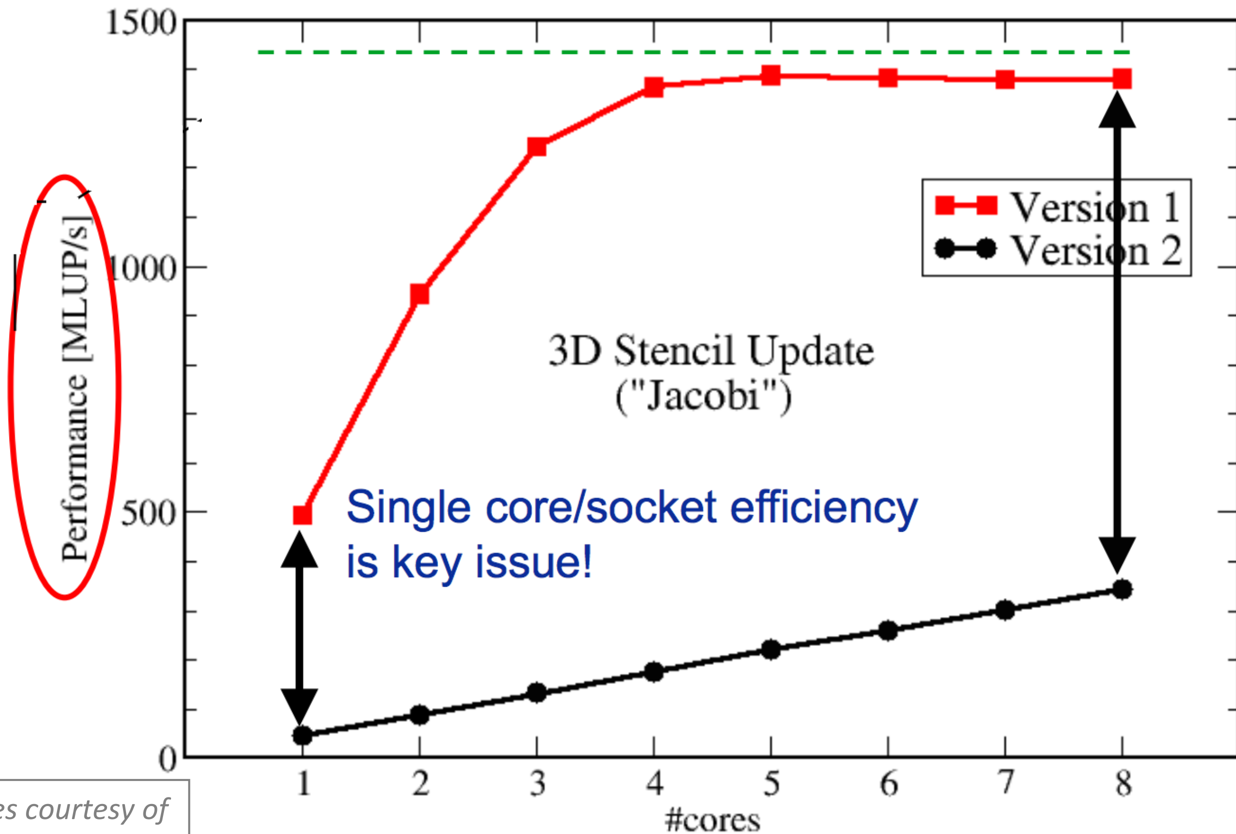



**Speedup = ratio – no absolute performance !**

# scalability vs. performance





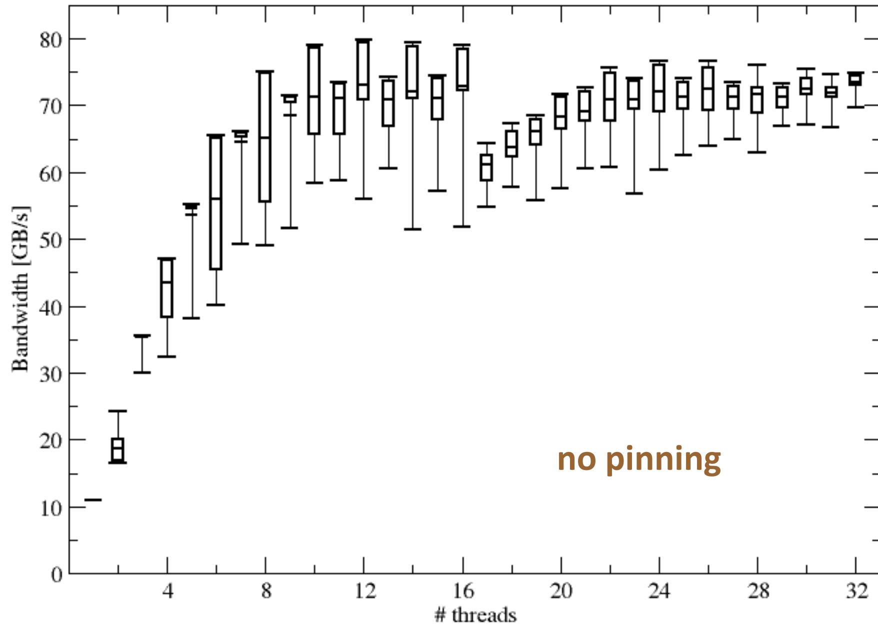

*Figures courtesy of Georg Hager.*

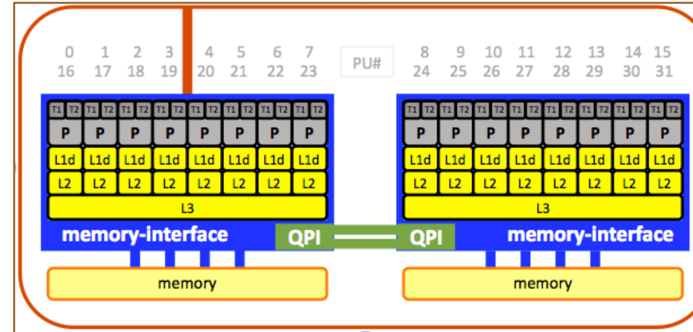
3D Stencil Update ("Jacobi"):

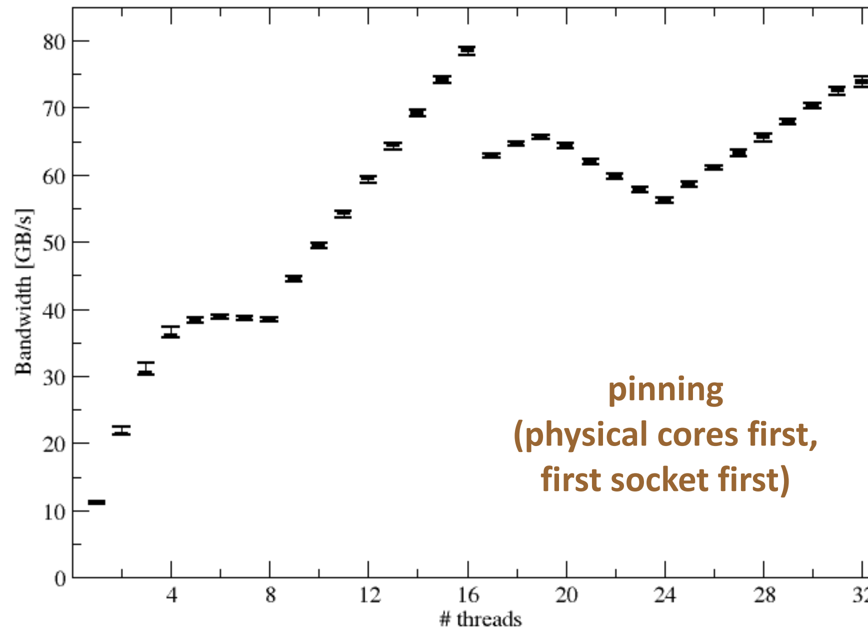
```
y(i,j,k) = b*(  x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k)
              + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1))
```

# pinning ?

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP




**OpenMP**
STREAM benchmark

*Benchmark & plots courtesy of Georg Hager.*

**MPI** will give the very same picture **!**

why should we care about **pinning** ?

- eliminating performance variations
- making use of architectural features
- avoiding resource contention

# HPC = computation – communication – I/O

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

| LATENCY | ← typical values → | BANDWIDTH |
|---|---|---|
| 1–2 ns | L1 cache | 100 GB/s |
| 3–10 ns | L2/L3 cache | 50 GB/s |
| 100 ns | memory | 10 GB/s |
| 1–10 µs | HPC networks | 1–8 GB/s |
| 50 µs | Gigabit Ethernet | 100 MB/s |
| 500 µs | Solid state disk | 100 MB/s |
| 10 ms | Local hard disk | 50 MB/s |
| 50 ms | Internet | 10 MB/s |

*HPC*

**computation node / core** — *exclusive*

**communication message passing** — *exclusive (BF)*

**I/O parallel FS** — *shared with all users*

Understand HW features! | Know your code! | Know the sys. environment! | → Take control!

➔ **Avoiding slow data paths is the key to most performance optimizations!**

# OpenMP

standard - defined for C/C++ and Fortran

OpenMP 5.2 Specifications (PDF) and Reference Guides (PDF)

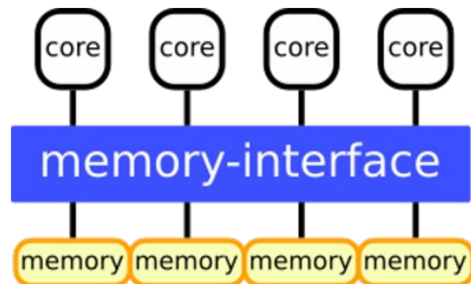
## **OpenMP**: shared memory (socket, node)

**Several sockets with multi-processors (node)**

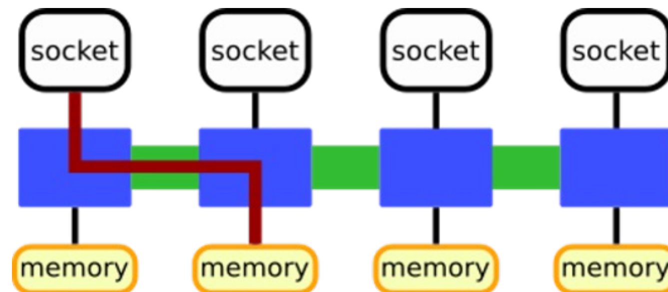
- memory is shared among all CPUs
- single / global address space
- uniform / non-uniform memory access

## OpenMP works only on shared memory!

- **socket** - UMA
- **node** - ccNUMA




**socket**
UMA (uniform memory access)
SMP (symmetric multi-processing)




**node**
ccNUMA (cache-coherent non-uniform ...)
**first touch, pinning!**

# OpenMP



- the **easiest way to parallelize** your code
- requires a shared memory system (allows to exploit node level parallelism)
- portable across shared memory architectures (standard since 1997)
- extension to C/C++ and Fortran
  (using directives, environment variables, and some library routines)

- focuses mostly on **parallelizing loops with independent iterations**
  (less and less true with each version)

**philosophy of OpenMP**

→parallelization with as little modification to the sequential program as possible

→incremental approach to parallelization

# OpenMP - nomenclature

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

- **thread** is a set of sequential instructions that are executed in order
- **thread** is a software construct - **core** is a hardware construct

  → often each thread in a program is mapped to a single core

- **shared memory model** assumes that all threads read and write from the same memory

- **distributed memory model** means that no shared memory is available
  (in this case communication has to be done by sending messages)

# OpenMP - simple example



To start with OpenMP is easy

```
#pragma omp parallel for
for (i = 0; i < n; i++)
{
    out[i] = in [i];
}
```

Divides the loop iterations into pieces
that are then executed in parallel by different threads.

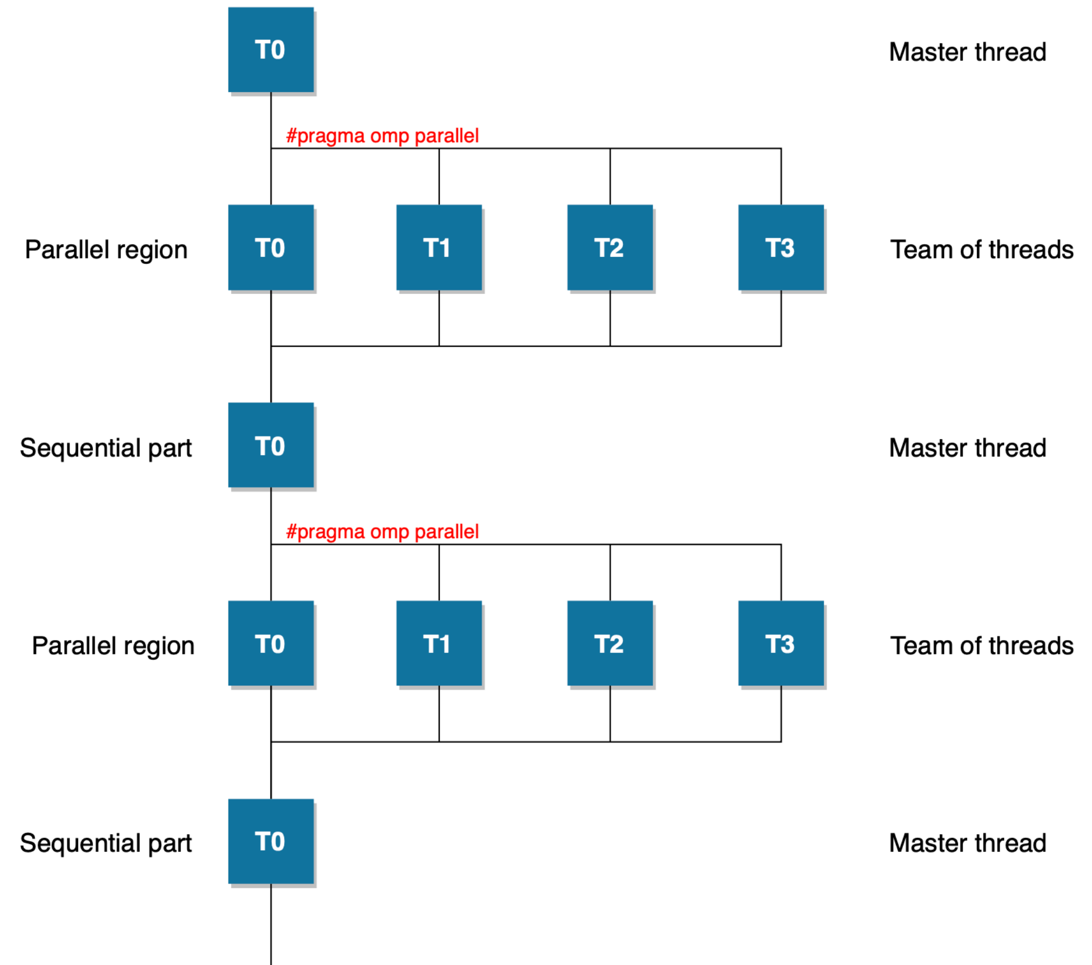
→ **it must be possible to determine the number of iterations at the time the loop starts execution**

# OpenMP - execution model

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

**fork-join model**

- program begins as a single process (**master thread**)
- at the **beginning** of a parallel region a **team of threads** is created
- at the **end** of a parallel region **threads synchronize** (implied barrier)
- at the end of a parallel region execution **continues sequentially**



# OpenMP - directives & parallel region

- OpenMP **directive format**

```
#pragma omp directive_name [clause, [[,] clause] ... ]
```

- a **parallel region** creates a team of threads that (potentially) execute the workload

```
#pragma omp parallel
{
    printf("Hello World!\n");
}
```

→ code is executed redundantly

# OpenMP - library functions



- the **header file omp.h** provides library functions
- **omp_get_num_threads()** → returns the number of threads in the current team
- **omp_get_thread_num()** → returns the thread id (0 to omp_get_num_threads()-1)

```
#pragma omp parallel
{
    if (omp_get_thread_num() == 0)
    printf(" Number of threads: %1d \n”, omp_get_num_threads();

    printf(" Hello world from thread %1d \n”, omp_get_thread_num();
}
```

- **compile**

  cc -fopenmp program.c

- **run**
  ./a.out

- **prints**

  Number of threads: 3
  Hello world from thread 2
  Hello world from thread 0
  Hello world from thread 1

# OpenMP - execution model

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

- there is **no guarantee in which order** the threads are executed
- if a specific order is desired this has to be enforced (might be **very expensive**)
- the thread id can be used to divide the work among threads (a lot of boilerplate)

**OpenMP provides facilities to**
**automatically divide the work among the threads in a team**

→ the corresponding directives are called **worksharing directives** (e.g., **for**)

→ **reduction**, combining multiple values into a single one, is a common pattern

# OpenMP - number of threads

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

- the **number of threads** used by OpenMP can be set by using environment variables

```
# set number of threads for the entire session
export OMP_NUM_THREADS=4;  ./program
# or only for one execution of the program
OMP_NUM_THREADS=4 ./program
```

- the **default**, often the number of hyperthreads in the system, is usually **not** an optimal choice

→ **rule of thumb:** number of threads = number of cores

# OpenMP - data environment



- **shared memory model:** all threads can write and read from main memory
- there are two types of variables
  → **shared** variables are common to all threads (usually arrays, global variables, ...)
  → **private** variables are duplicated on each thread (local variables, loop counters, ...)

- **by default all variables are shared**
- exceptions
  → local variables defined inside an OpenMP directive
  → loop control variables for a parallel for loop
  → variables that are declared in a called function

- **a variable can be explicitly declared as private or shared**

# OpenMP - a word of warning



- OpenMP is easy to write, but it is also easy to get wrong.
- OpenMP **delegates a lot of responsibility to the programmer**.

- ensure that the code can be parallelized

  → make sure that the **loop iterations are independent**

- **NO race conditions!** - a program with a race condition is always wrong...
  a race condition occurs when multiple threads are allowed to access the same memory location and at least one access is a write

# example - pi serial

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```c
#include <time.h>
#include <stdio.h>
#include <math.h>

int main(int argc, char *argv[])
{
    int num_threads, i, n = 10000000;
    double pi, sum, h, x;
    double time, time_s, time_e;
    double PI25DT = 3.141592653589793238462643;

    num_threads = 1;

    h = 1.0 / (double)n;
    sum = 0.0;

    time_s = clock();

    for (i = 0; i < n; i++)
    {
        x = h * ((double)i + 0.5);
        sum += 4.0 / (1.0 + x*x);
    }

    pi = h * sum;

    time_e = clock();

    printf("serial, time, pi, error: %1d, %.3f, %.16f, %.16f\n",
        num_threads, ((time_e-time_s)/1e3), pi, fabs(pi-PI25DT));
}
```

# example - pi OpenMP

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```c
#include <omp.h>
#include <stdio.h>
#include <math.h>

int main(int argc, char *argv[])
{
    int num_threads, i, n = 10000000;
    double pi, sum, h, x;
    double time, time_s, time_e;
    double PI25DT = 3.141592653589793238462643;

    #pragma omp parallel
    {
    #pragma omp single
    num_threads = omp_get_num_threads();
    }

    h = 1.0 / (double)n;
    sum = 0.0;

    time_s = omp_get_wtime();

    #pragma omp parallel for private(x) shared(h) reduction(+:sum)
    for (i = 0; i < n; i++)
    {
        x = h * ((double)i + 0.5);
        sum += 4.0 / (1.0 + x*x);
    }

    pi = h * sum;

    time_e = omp_get_wtime();

    printf ("num_threads, time, pi, error: %02d, %.3f, %.16f, %.16f\n",
            num_threads, ((time_e-time_s)*1e3), pi, fabs(pi-PI25DT));
}
```

# results - pi OpenMP

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

- `cd PI`
- `ml OpenMPI/4.1.1-GCC-10.2.0-Java-1.8.0_221`
- `vi pi_openmp.c`
- **`cc -fopenmp -o pi_openmp pi_openmp.c`**
- **`OMP_NUM_THREADS=1 ./pi_openmp`** → 1,2,4,8,16,32

```
num_threads, time, pi, error: 01, 34.713, 3.1415926535897309, 0.0000000000000622
num_threads, time, pi, error: 02, 18.123, 3.1415926535899228, 0.0000000000001297
num_threads, time, pi, error: 04,  8.933, 3.1415926535896697, 0.0000000000001235
num_threads, time, pi, error: 08,  4.363, 3.1415926535898038, 0.0000000000000107
num_threads, time, pi, error: 16,  2.618, 3.1415926535898024, 0.0000000000000093
num_threads, time, pi, error: 32,  1.352, 3.1415926535898024, 0.0000000000000093
```

# MPI

standard - defined for C/C++ and Fortran

MPI: A Message-Passing Interface Standard Version 4.0 (PDF))

**python** (not part of the MPI standard): https://mpi4py.readthedocs.io/

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

## **MPI:** distributed memory (socket, node, cluster)

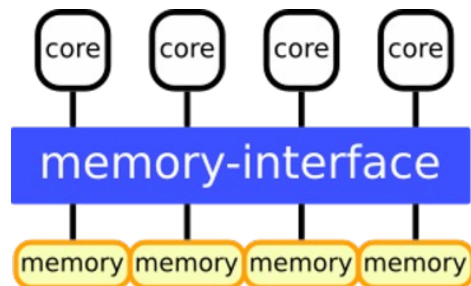
### Several sockets with multi-processors (node)

- memory is shared among all CPUs
- single / global address space
- uniform / non-uniform memory access

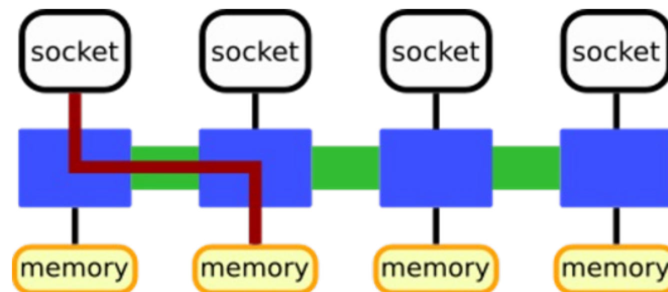
## MPI works everywhere!

### Multi-computers with various architectures (cluster)

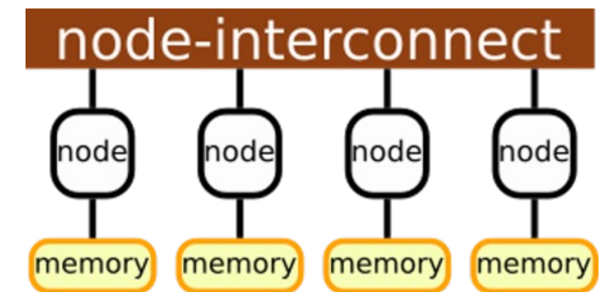
- set of nodes interconnected by a network
- each node has separated memory
- slower access to memories of other processors




**socket**
UMA (uniform memory access)
SMP (symmetric multi-processing)




**node**
ccNUMA (cache-coherent non-uniform ...)
**first touch, pinning!**

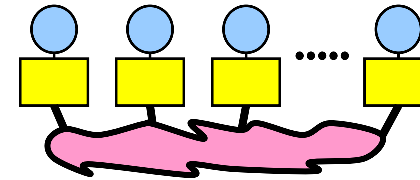



**cluster**
NUMA (non-uniform memory access)
fast access to own memory only

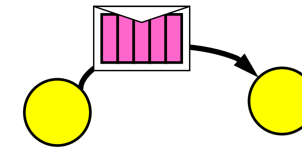
# MPI basics

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

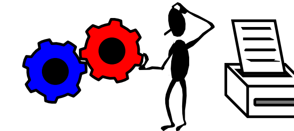
- **overview, process model and language bindings**
  - one program on several processors
  - work and data distribution
  - starting several MPI processes



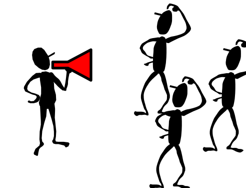
- **messages and point-to-point communication**
  - the MPI processes can communicate



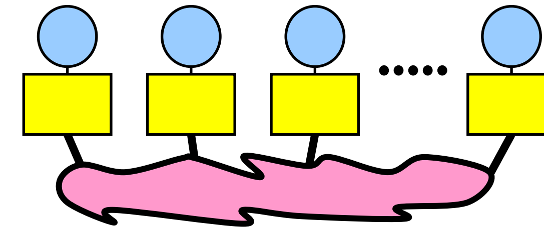
- **nonblocking communication**
  - to avoid idle times, serializations, and deadlocks



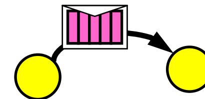
- **collective communication**
  - e.g. broadcast, reduction, …



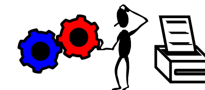
- **overview, process model and language bindings**
  - one program on several processors
  - work and data distribution
  - starting several MPI processes



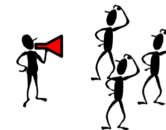
- **messages and point-to-point communication**
  - the MPI processes can communicate



- **nonblocking communication**
  - to avoid idle times, serializations, and deadlocks

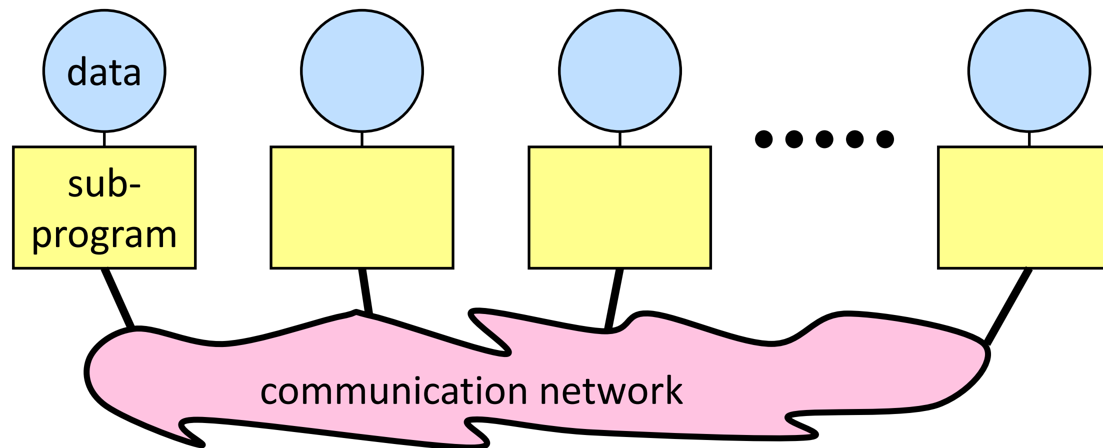


- **collective communication**
  - e.g. broadcast, reduction, …



each processor in a message passing program runs a ***sub-program***

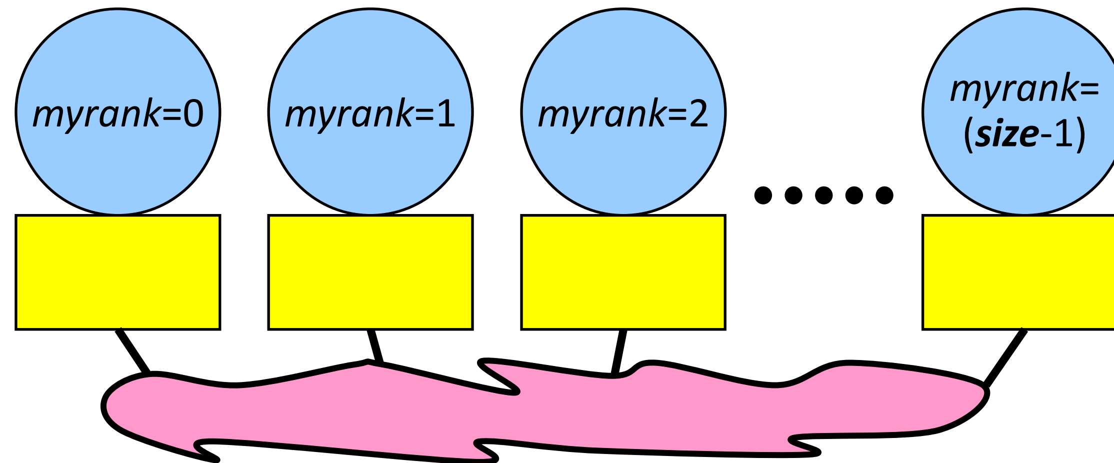
- written in a conventional sequential language, e.g., C, Fortran, or python
- typically the same on each processor (SPMD), all variables are private
- communicate via special send & receive routines (***message passing***)



# data & work distribution



- the system of ***size*** processes is started by special MPI initialization program
- the value of ***myrank*** is returned by special library routine
- all distribution decisions are based on ***myrank***



# MPI process model

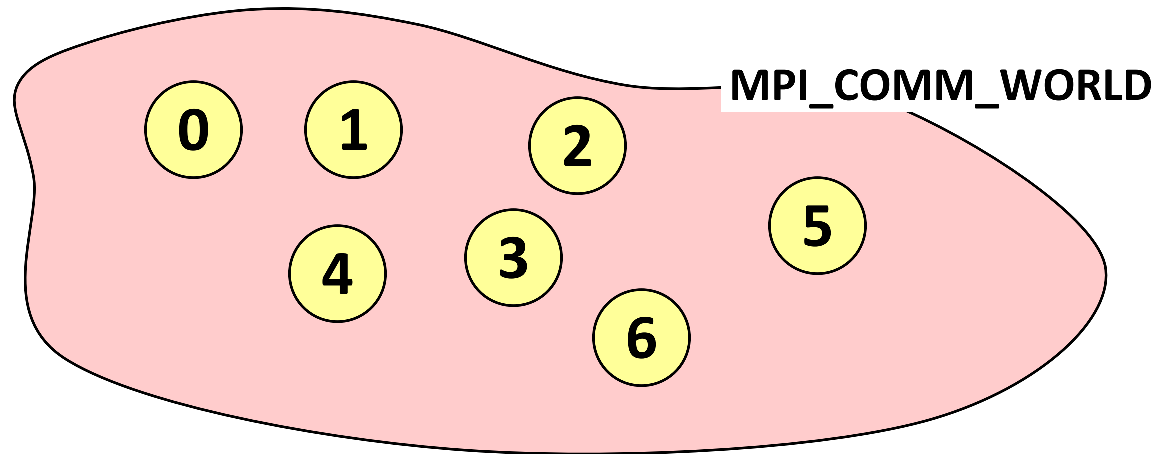
SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

- must be linked with an MPI library → `mpicc`
- must be started with the MPI startup tool → `mpirun -n # ./a.out`

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
MPI_Init(&argc, &argv);
...
MPI_Finalize();
}
```

- MPI function format → `MPI_Xxxxxx(parameter,...);`

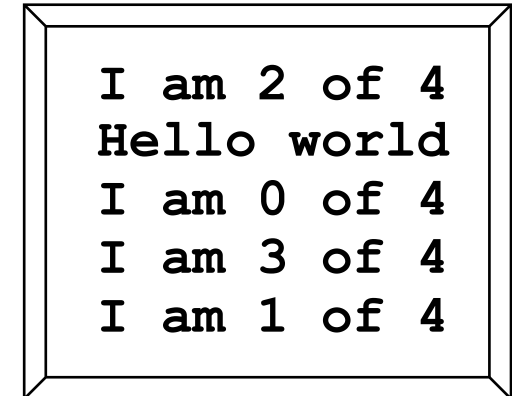
- all processes (= sub-programs) of one MPI program are combined in the **communicator** **MPI_COMM_WORLD** (predefined handle)
- **size** is the number of processes in a communicator
- each process has its own **rank** in a communicator
  starting with 0 – ending with (size-1)



# example: Hello world!

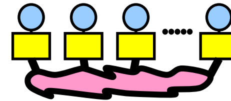
SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
   int rank, size;

   MPI_Init(&argc, &argv);

   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   if (my_rank == 0)
   {
     printf ("Hello world!\n");
   }
   printf("I am process %i out of %i\n", rank, size);

   MPI_Finalize();
}
```

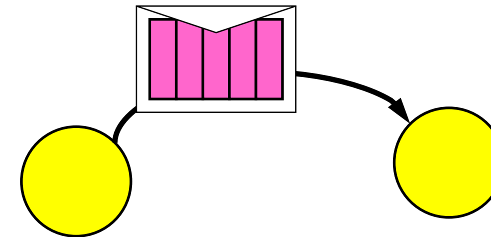


# point-to-point communication

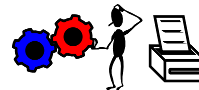
- **overview, process model and language bindings**
  - one program on several processors
  - work and data distribution
  - starting several MPI processes



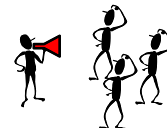
- **messages and point-to-point communication**
  - the MPI processes can communicate



- **nonblocking communication**
  - to avoid idle times, serializations, and deadlocks

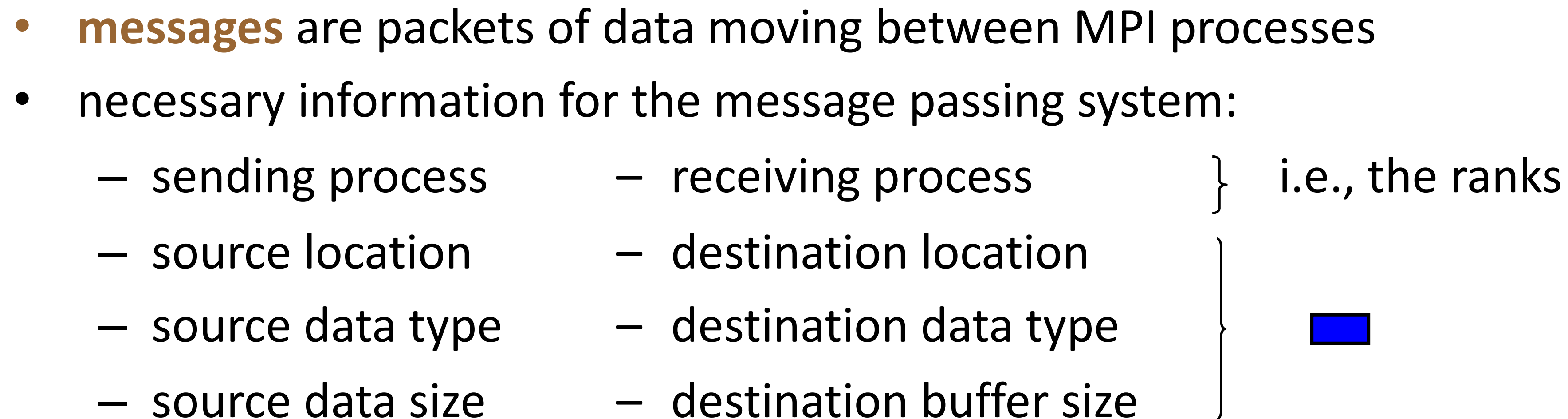


- **collective communication**
  - e.g. broadcast, reduction, …

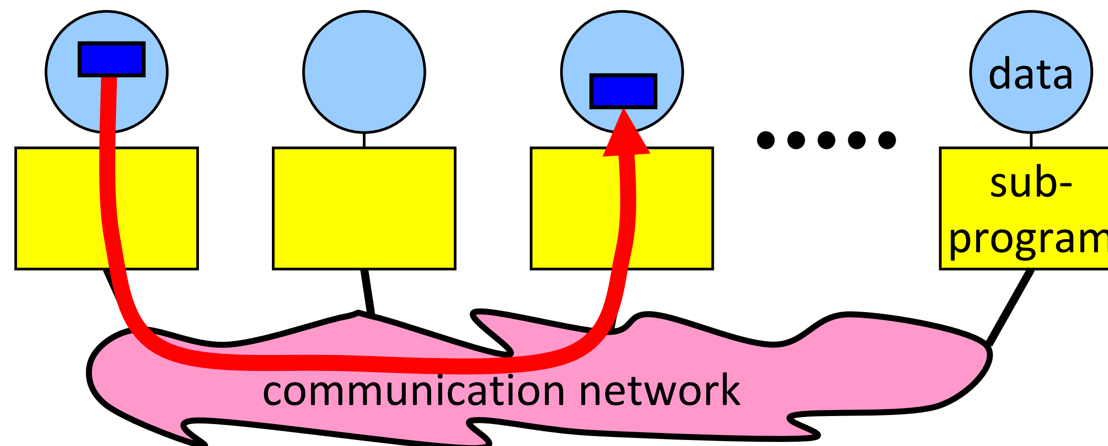


# point-to-point communication





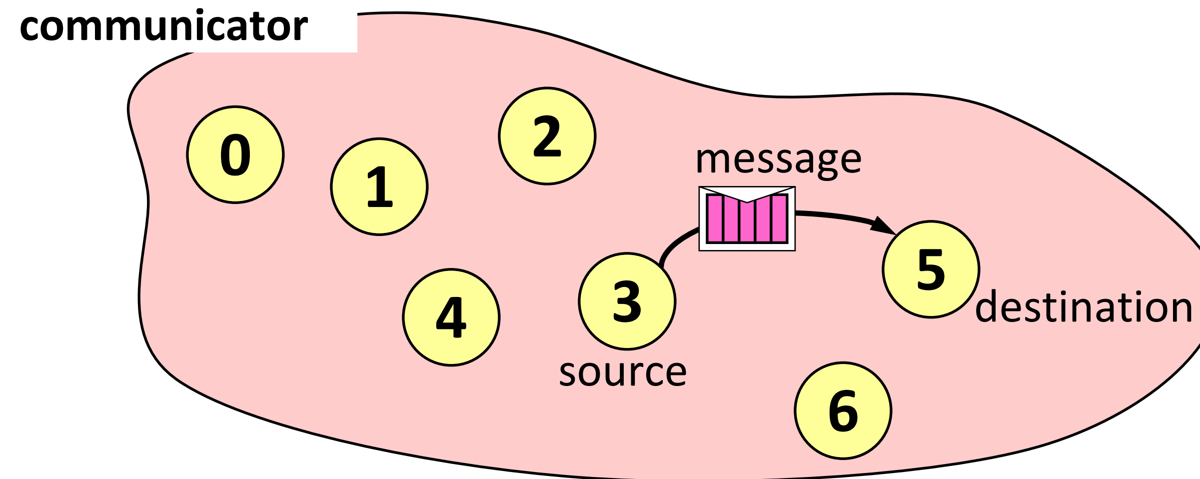
- **messages** are packets of data moving between MPI processes
- necessary information for the message passing system:
  - sending process – receiving process } i.e., the ranks
  - source location – destination location
  - source data type – destination data type
  - source data size – destination buffer size



# point-to-point communication



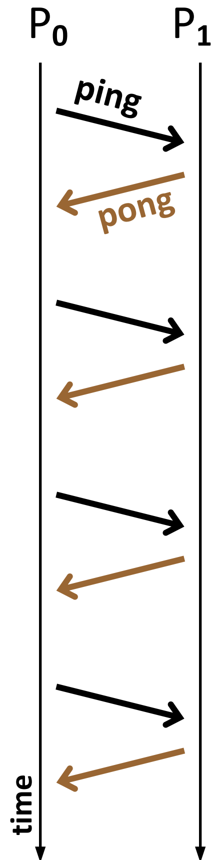
- communication between **two** processes
- **source** process sends message to **destination** process
- communication takes place within a **communicator**, e.g., MPI_COMM_WORLD
- processes are identified by their **ranks** in the communicator



# example: ping pong

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

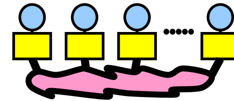
```
start = MPI_Wtime();

for (i = 1; i <= 50; i++)
{
  if (my_rank == 0)
  {
     MPI_Send(buffer, 1, MPI_FLOAT, 1, 17, MPI_COMM_WORLD);
     MPI_Recv(buffer, 1, MPI_FLOAT, 1, 23, MPI_COMM_WORLD, &status);
  }
  else if (my_rank == 1)
  {
     MPI_Recv(buffer, 1, MPI_FLOAT, 0, 17, MPI_COMM_WORLD, &status);
     MPI_Send(buffer, 1, MPI_FLOAT, 0, 23, MPI_COMM_WORLD);
  }
}

finish = MPI_Wtime();

if (my_rank == 0)
  printf("Time for one messsage: %f micro seconds.\n",
            finish - start) / (2 * 50) * 1e6 );
```



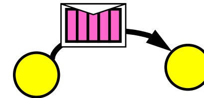
# nonblocking communication

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

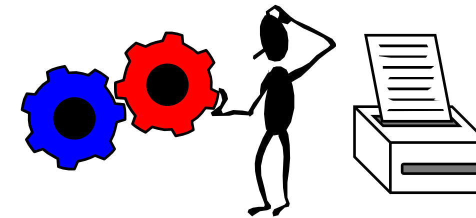
- **overview, process model and language bindings**
  - one program on several processors
  - work and data distribution
  - starting several MPI processes



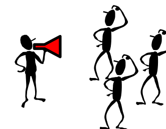
- **messages and point-to-point communication**
  - the MPI processes can communicate



- **nonblocking communication**
  - to avoid idle times, serializations, and deadlocks



- **collective communication**
  - e.g. broadcast, reduction, …



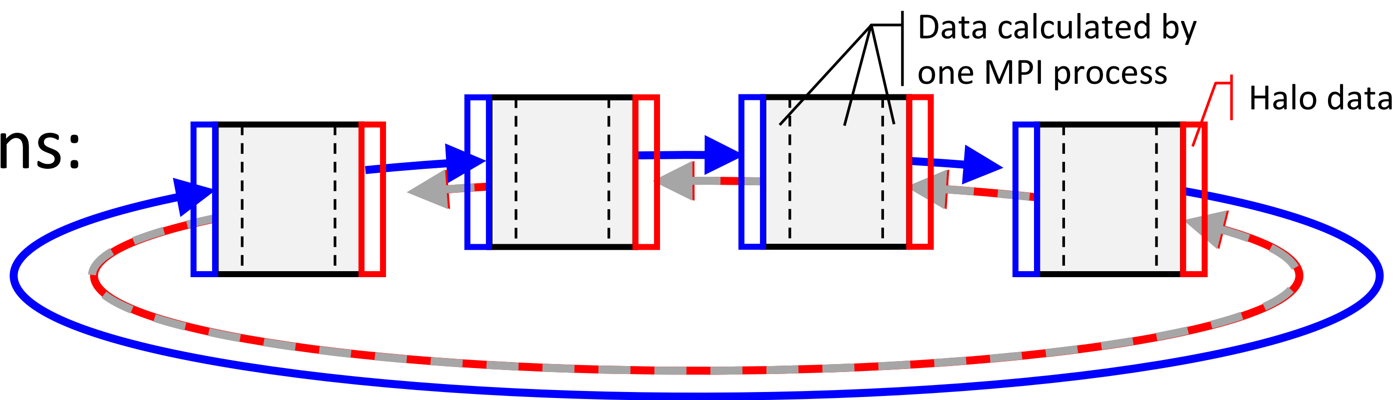
# nonblocking communication



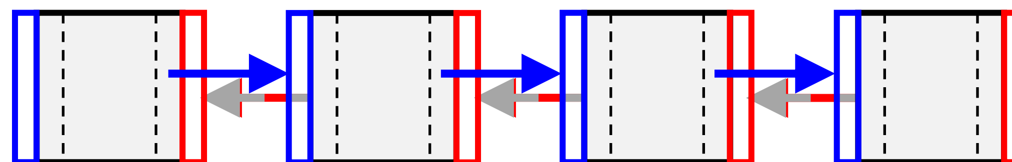
→ **to avoid idle times, serializations and deadlocks**

→ **halo communication**

cyclic boundary conditions:




non-cyclic boundary:



# blocking → risk deadlocks & serializations

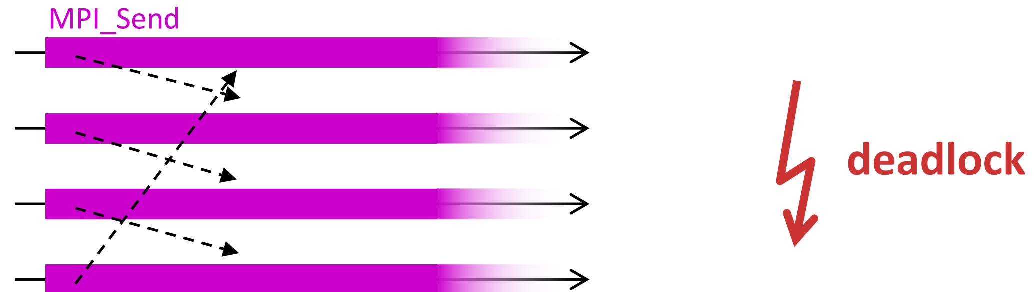
SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

if the MPI library chooses the synchronous protocol

timelines of all processes

cyclic boundary:

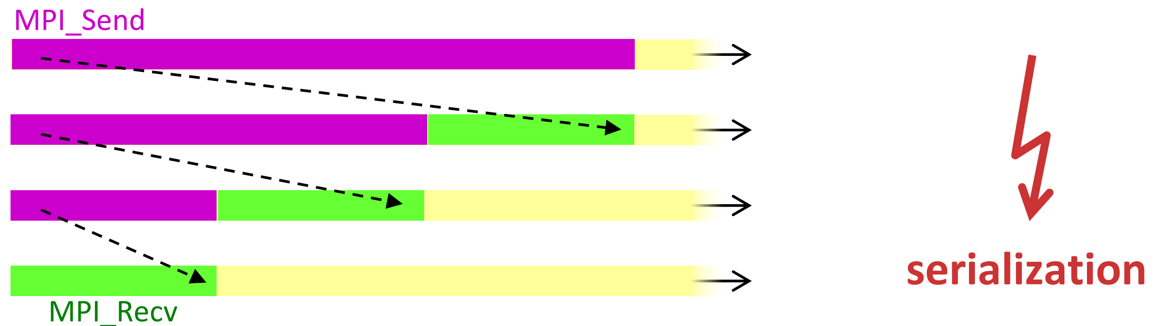
```
MPI_Send(..., right, ...)
MPI_Recv( ..., left,  ...)
```




deadlock

non-cyclic boundary:

```
if (myrank < size-1)
  MPI_Send(..., right, ...);
if (myrank > 0)
  MPI_Recv( ..., left,  ...);
```




serialization

# cyclic communication → other bad ideas

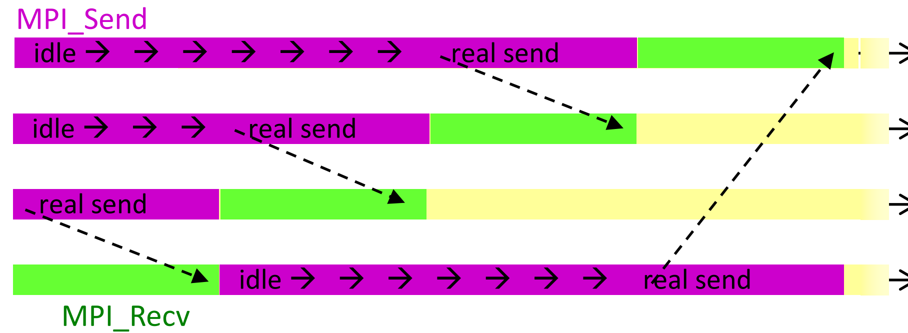
SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

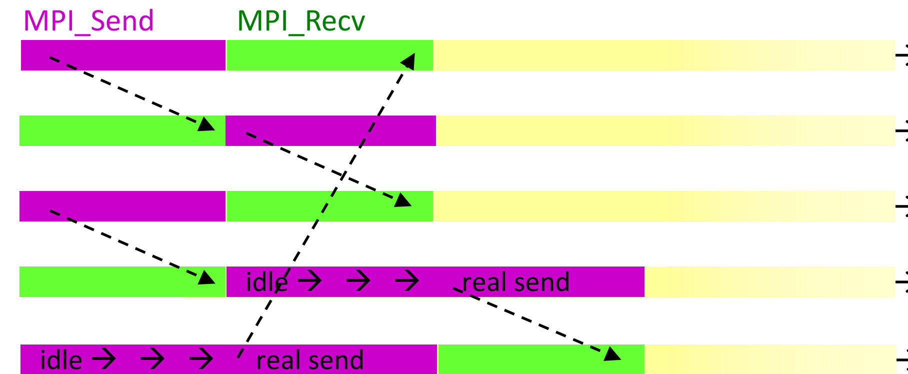
```
if (myrank < size-1) {
   MPI_Send(…, right, …);
   MPI_Recv( …, left,  …);
} else {
   MPI_Recv( …, left,  …);
   MPI_Send(…, right, …);
}
```

```
if (myrank%2 == 0) {
   MPI_Send(…, right, …);
   MPI_Recv( …, left,  …);
} else {
   MPI_Recv( …, left,  …);
   MPI_Send(…, right, …);
}
```

if the MPI library chooses the synchronous protocol

timelines of all processes



**serialization**

# nonblocking communication

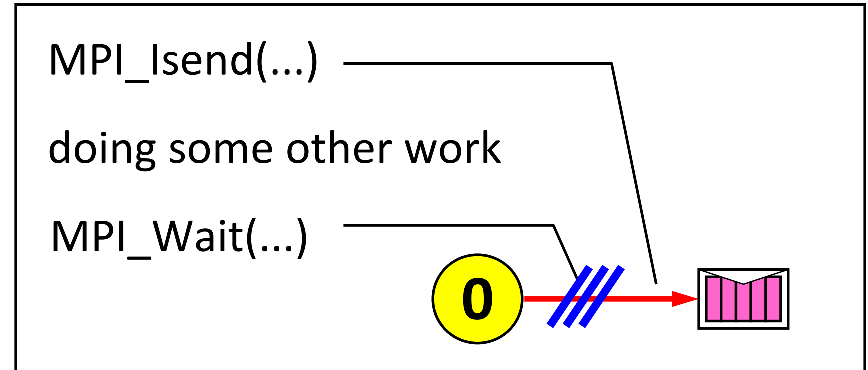


separate communication into **three phases**:

- initiate nonblocking communication
  - routine name starting with MPI_**I**...
  - incomplete
  - local, returns immediately,
    returns independently of any other process' activity

→ do some work (perhaps involving other communications?)

- wait for nonblocking communication to **complete**
  - the send buffer is read out, or
  - the receive buffer is filled in




the definition of nonblocking is clarified in

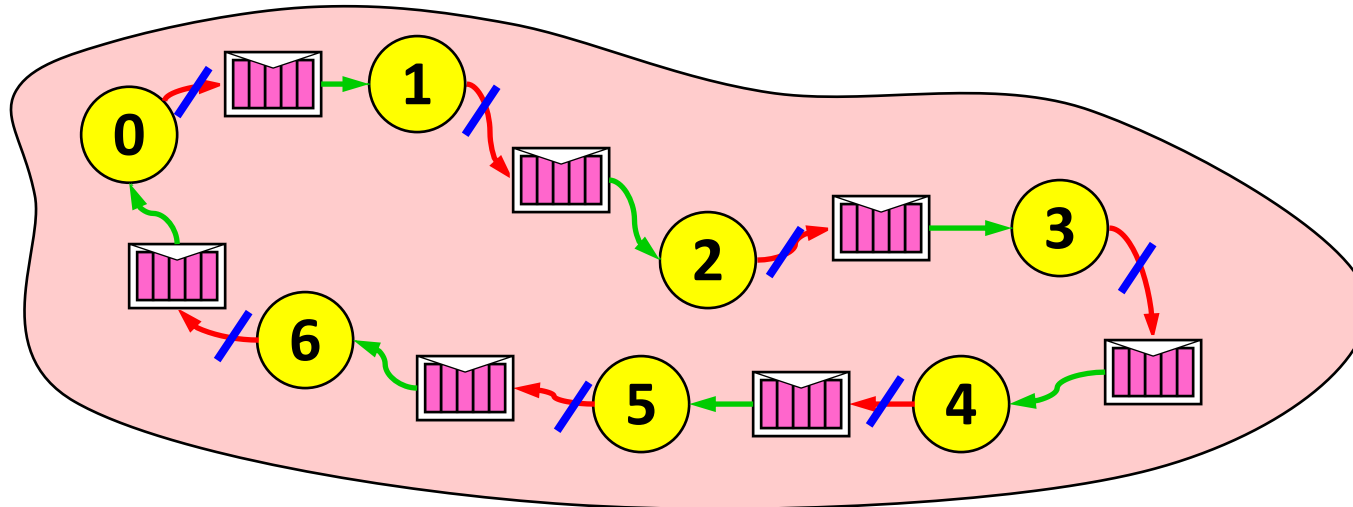
**MPI-4.0**

reading: MPI-4.0/2.4 & MPI-4.0/3.7

# nonblocking send

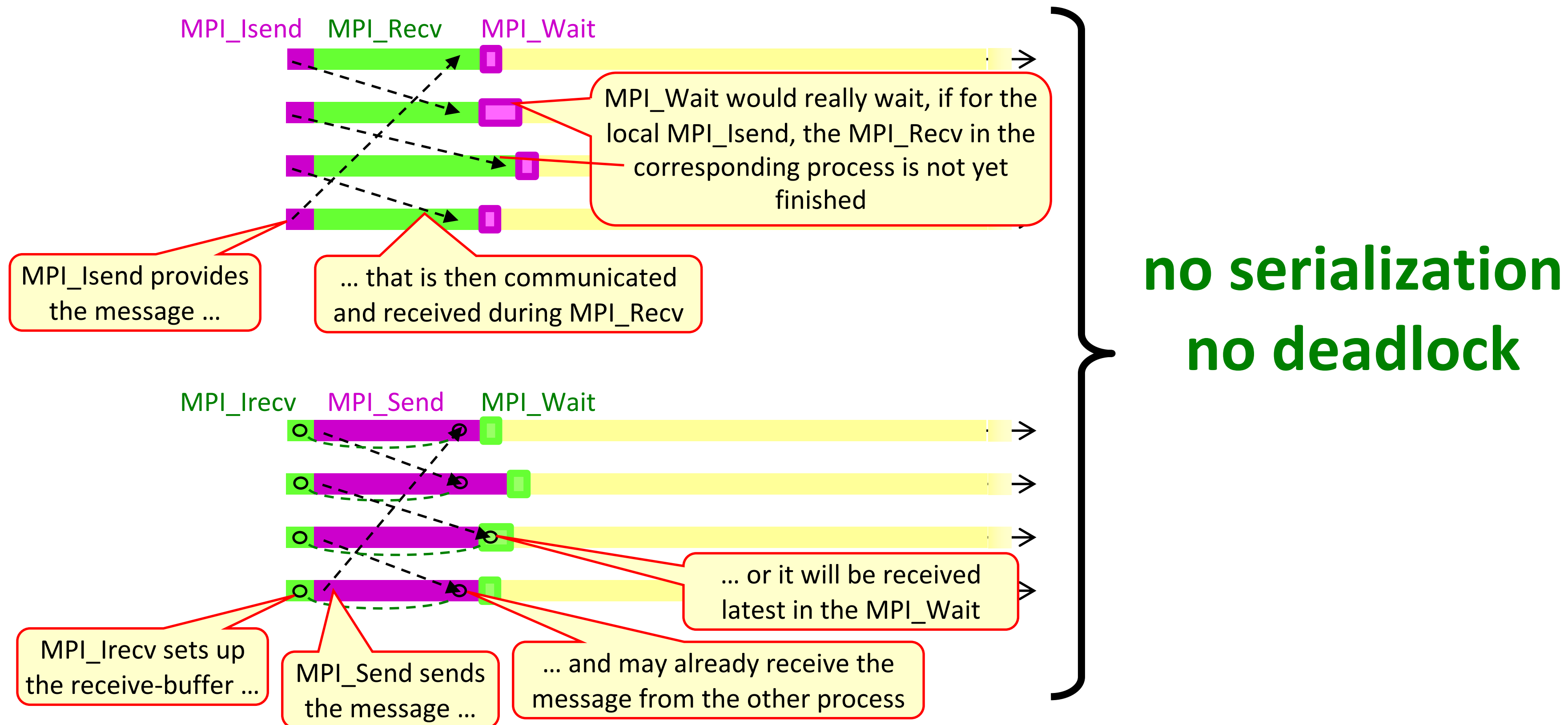


- Initiate nonblocking send
  - in the ring example: Initiate nonblocking send to the right neighbor
- Do some work:
  - in the ring example: Receiving the message from left neighbor
- Now, the message transfer can be completed
- Wait for nonblocking send to complete



# nonblocking timelines






**no serialization**
**no deadlock**

# use cases for nonblocking comm.



→ **to avoid idle times, serializations and deadlocks**
(as if overlapping of communication with other communication)

→ **real overlapping of**

- several communications
- communication and computation

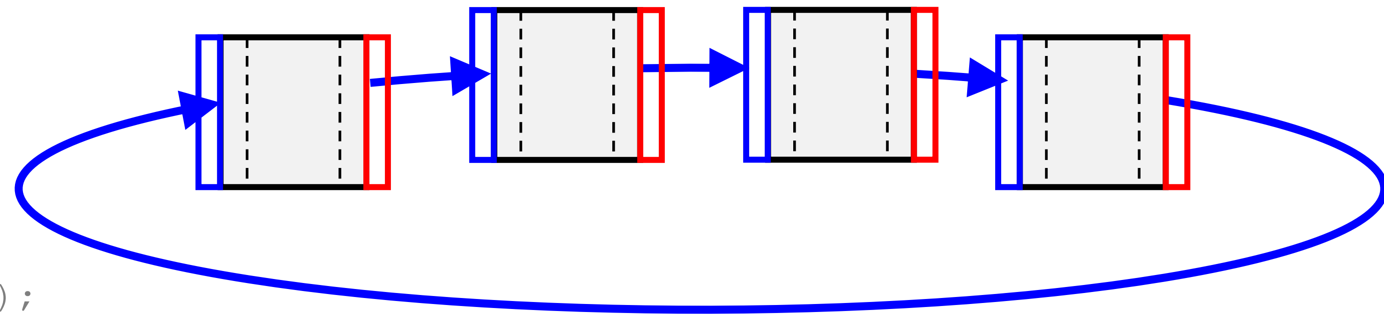
→ **other MPI features: Send-Receive in one routine**

- MPI_Sendrecv & MPI_Sendrecv_replace (blocking → prevent serializations & deadlocks)
- combines the triple "MPI_Irecv + Send + Wait" into one routine
- MPI_Isendrecv & MPI_Isendrecv_replace (nonblocking → minimize idle times) ← **new MPI 4.0**

# example: ring

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```
int snd_buf, rcv_buf;
int right, left;
int sum, rank, size, i;
MPI_Status  status;
MPI_Request request;

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

right = (rank+1)      % size;
left  = (rank-1+size) % size;
sum = 0;
snd_buf = rank;
for( i = 0; i < size; i++)
{
  MPI_Issend(&snd_buf, 1, MPI_INT, right, 17, MPI_COMM_WORLD, &request);
  MPI_Recv ( &rcv_buf, 1, MPI_INT, left,  17, MPI_COMM_WORLD, &status);
  MPI_Wait(&request, &status);
  snd_buf = rcv_buf;
  sum += rcv_buf;
}
printf ("PE%i:\tSum = %i\n", my_rank, sum);
MPI_Finalize();
```

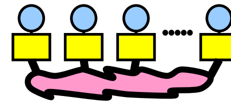


Synchronous **send** (**Issend**) instead of standard send (**Isend**) is used only to demonstrate the use of the nonblocking routine resolves the deadlock (or serialization) problem.
A real application would use standard **Isend().**

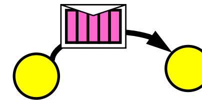
# collective communication

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

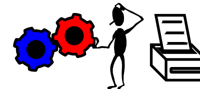
- **overview, process model and language bindings**
  - one program on several processors
  - work and data distribution
  - starting several MPI processes



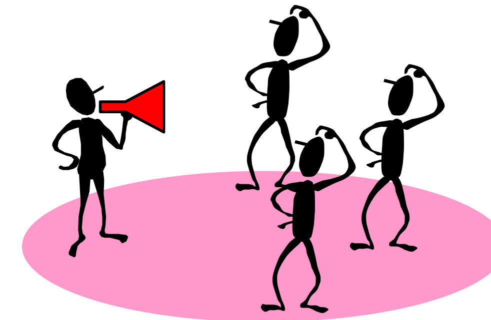
- **messages and point-to-point communication**
  - the MPI processes can communicate



- **non-blocking communication**
  - to avoid idle times, serializations, and deadlocks



- **collective communication**
  - e.g. broadcast, reduction, …



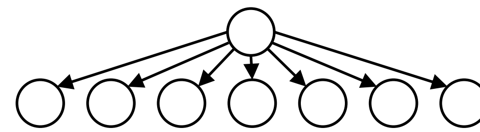
# collective communication

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

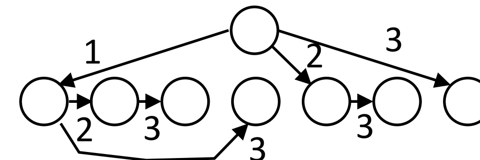
- **all processes in a communicator** processes are involved
- can be built out of point-to-point communications, but …
- allow **optimized** internal implementations (by MPI libraries)
- examples:
  - **broadcast**, scatter, gather
  - **reduction operations** (global sum, maximum, etc.)
  - barrier synchronization (do NOT use in production code!)
  - neighbor communication in a virtual process grid

You need not to care about it !
It is the job of the MPI library !!!

Should be faster than any programming with point-to-point messages!



Sequential algorithm
O(# processes)




Tree based algorithm
$O(\log_2(\# \text{ processes}))$

# example - pi serial

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```c
#include <time.h>
#include <stdio.h>
#include <math.h>

int main(int argc, char *argv[])
{
    int num_threads, i, n = 10000000;
    double pi, sum, h, x;
    double time, time_s, time_e;
    double PI25DT = 3.141592653589793238462643;

    num_threads = 1;

    h = 1.0 / (double)n;
    sum = 0.0;

    time_s = clock();

    for (i = 0; i < n; i++)
    {
        x = h * ((double)i + 0.5);
        sum += 4.0 / (1.0 + x*x);
    }

    pi = h * sum;

    time_e = clock();

    printf("serial, time, pi, error: %1d, %.3f, %.16f, %.16f\n",
       num_threads, ((time_e-time_s)/1e3), pi, fabs(pi-PI25DT));
}
```

# example - pi MPI

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

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

int main(int argc, char *argv[])
{
    int rank, size, i, n = 10000000;
    double mypi, pi, sum, h, x;
    double time, time_s, time_e;
    double PI25DT = 3.141592653589793238462643;

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

    h = 1.0 / (double)n;
    sum = 0.0;
    time_s = MPI_Wtime ();

    for (i = rank; i < n; i += size )
    {
        x = h * ((double)i + 0.5);
        sum += 4.0 / (1.0 + x*x);
    }

    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    time_e = MPI_Wtime ();

    if (rank == 0)
    printf ("size, time, pi, error: %02d, %.3f, %.16f, %.16f\n",
            size, ((time_e-time_s)*1e3), pi, fabs(pi-PI25DT));

    MPI_Finalize();
}
```

# results - pi MPI

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

- `cd PI`
- `ml OpenMPI/4.1.1-GCC-10.2.0-Java-1.8.0_221`
- `vi pi_mpi.c`
- **`mpicc -o pi_mpi pi_mpi.c`**
- **`mpirun -n 1 ./pi_mpi`** → 1,2,4,8,16,32

```
size, time, pi, error: 01, 35.339, 3.1415926535897309, 0.0000000000000622
size, time, pi, error: 02, 17.625, 3.1415926535899850, 0.0000000000001918
size, time, pi, error: 04,  9.157, 3.1415926535896861, 0.0000000000001070
size, time, pi, error: 08,  4.955, 3.1415926535898069, 0.0000000000000138
size, time, pi, error: 16,  2.451, 3.1415926535897931, 0.0000000000000000
size, time, pi, error: 32,  2.789, 3.1415926535897847, 0.0000000000000084
```

no standard

different options

# GPU

ISO standard parallelism

OpenACC

OpenMP

CUDA

# CPU & GPU



- GPU & GPU are fundamentally different

- **CPU is a latency reducing architecture** - optimized for serial tasks

+ very large main memory
+ very fast clock speed
+ latency optimized via large caches
+ small number of threads can run very quickly
- relatively low memory bandwiths
- cache missed very costly
- low performance / watt

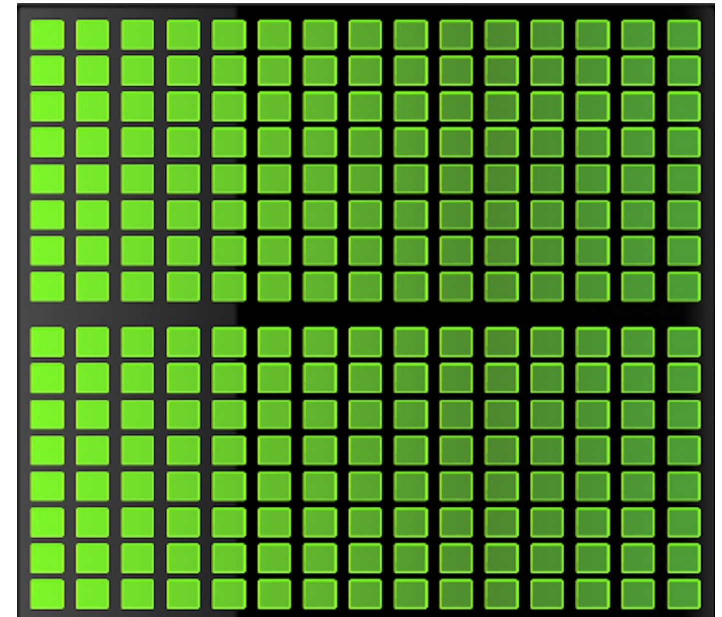
# CPU & GPU



- GPU & GPU are fundamentally different

- **GPU is all about hiding latency** - optimized for parallel tasks

+ high-bandwidths main memory
+ significantly more compute resources
+ latency tolerant via parallelism
+ high throughput
+ high performance / watt
- relatively low memory capacity
- low per-thread performance



# Thank you for your attention!

http://sctrain.eu/

Co-funded by the Erasmus+ Programme of the European Union

This project has been funded with support from the European Commission.
This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.