Primena super-računara u astronomiji

Basic principles of parallelism

- **Parallelization** is the process of formulating a problem in a way that lends itself to concurrent execution by several "**execution units**"
- Ideally, the execution units are initially given some amount of work to do which they execute in exactly the same amount of time
- Using **N** units, a problem that takes a time **T** to be solved sequentially, will now take only **T/N**.
- We call this a speedup of **N**.
- How well a task can be parallelized is usually quantified by some **scalability metric**

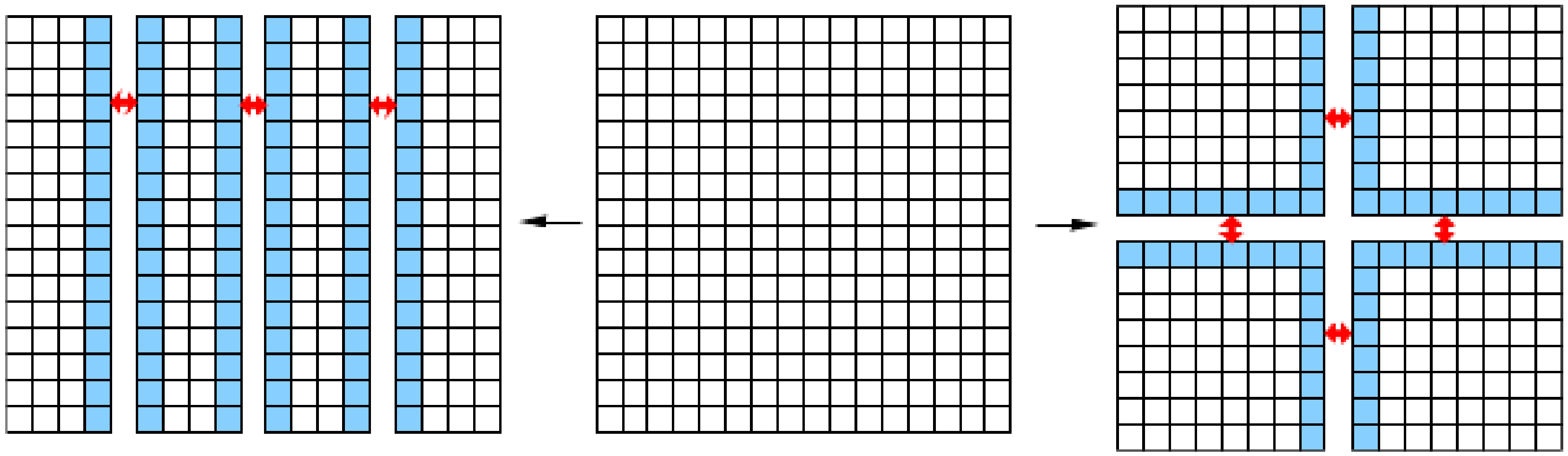
Parallelization strategies

- **Data parallelism:** Many simulations in science could be represented with a simplified picture of reality in which a computational domain is represented as a grid of discrete positions for the physical quantities under consideration. The work is distributed across processors, and a part of the grid is assign to each CPU. This is called **domain decomposition**.

- **Functional (or task) parallelism:** Solving a complete problem can be split into more or less disjoint subtasks. The tasks can be worked on in parallel, using appropriate amounts of resources so that load imbalance is kept under control.

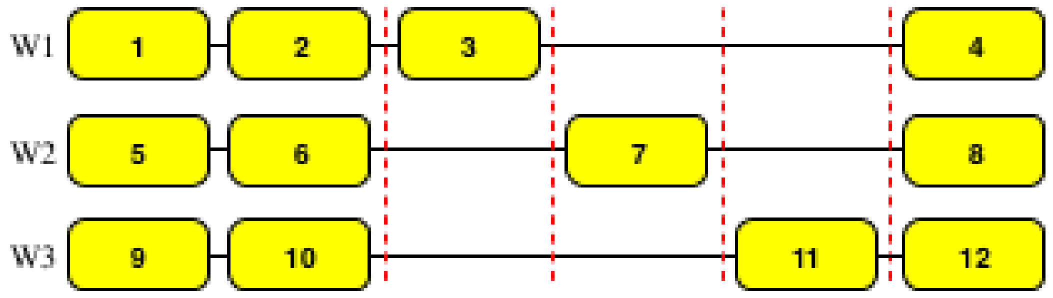
Data parallelism

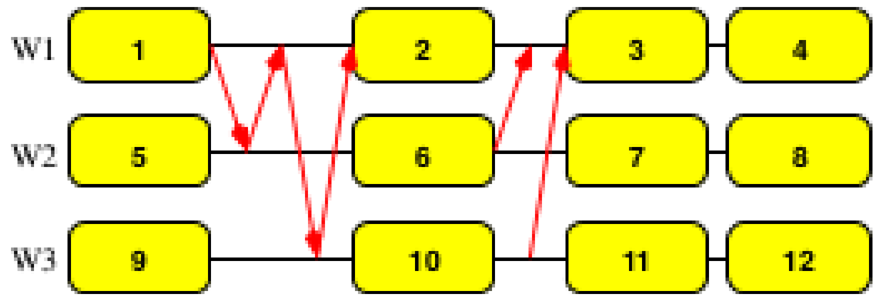
- As an example consider a two-dimensional simulation code that updates physical variables on a **n×n grid**
- Domain decomposition subdivides the computational domain into N subdomains
- The computational effort should be equal for all domains to avoid load imbalance
- It may be necessary to communicate data across domain boundaries
- The communication cost grows linearly with the distance that has to be bridged in order to calculate observables at a certain point of the grid



Functional parallelism

- Spliting a complete problem into disjoint subtasks that can be worked on in parallel
- For instance, assign some resources to communication and others to computational work





Performance models for parallel scalability

- In a simple model, the overall problem size ("amount of work") shall be **s + p = 1**
- The 1-CPU (serial) runtime for this case: $T_f^s = s + p$
- Solving the same problem on N CPUs: $T_f^p = s + \frac{p}{N}$
- This is called strong scaling because the amount of work stays constant no matter how many CPUs are used
- Here the goal of parallelization is minimization of time to solution for a given problem

Performance models for parallel scalability

- For larger problem sizes, for which available memory is the limiting factor, it is appropriate to scale the problem size with some power of N so that the total amount of work is $s + pN^{\alpha}$
- The serial runtime for the scaled problem is defined as:

$$T_v^s = s + pN^{\alpha}$$

- The parallel runtime is: $T_v^p = s + pN^{\alpha-1}$

- This approuch is called weak scaling

Scalability limitations

$$T_\mathrm{f}^\mathrm{s} = s + p$$

$$T_\mathrm{f}^\mathrm{p} = s + \frac{p}{N}$$

- Serial performance for **fixed problem size**:

$$P_\mathrm{f}^\mathrm{s} = \frac{s+p}{T_\mathrm{f}^\mathrm{s}} = 1$$

s + p = 1

- Parallel performance for **fixed problem size**:

$$P_\mathrm{f}^\mathrm{p} = \frac{s+p}{T_\mathrm{f}^\mathrm{p}(N)} = \frac{1}{s + \frac{1-s}{N}}$$

- Application speedup:

$$S_\mathrm{f} = \frac{P_\mathrm{f}^\mathrm{p}}{P_\mathrm{f}^\mathrm{s}} = \frac{1}{s + \frac{1-s}{N}}$$

- **Amdahl's Law:** limits application speedup for large **N** to **1/s**

Scalability limitations

- If we define "work" as only the parallelizable part of the calculation
- Serial performance for **fixed problem size**:

$$P_\mathrm{f}^{sp} = \frac{p}{T_\mathrm{f}^\mathrm{s}} = p$$

- Parallel performance for **fixed problem size**:

$$P_\mathrm{f}^{\mathrm{p}p} = \frac{p}{T_\mathrm{f}^\mathrm{p}(N)} = \frac{1-s}{s + \frac{1-s}{N}}$$

- Application speedup:

$$S_\mathrm{f}^{p} = \frac{P_\mathrm{f}^{\mathrm{p}p}}{P_\mathrm{f}^{sp}} = \frac{1}{s + \frac{1-s}{N}}$$

Scalability limitations

- In the case of **weak scaling** where workload grows with CPU count, the question to ask is "How much more work can my program do in a given amount of time when I put a larger problem on N CPUs?"

- Serial performance is:

$$P_{\mathrm{v}}^{\mathrm{s}} = \frac{s+p}{T_{\mathrm{f}}^{\mathrm{s}}} = 1$$

- Parallel performance is:

$$P_{\mathrm{v}}^{\mathrm{p}} = \frac{s+pN^{\alpha}}{T_{\mathrm{v}}^{\mathrm{p}}(N)} = \frac{s+(1-s)N^{\alpha}}{s+(1-s)N^{\alpha-1}} = S_{\mathrm{v}}$$

- In the special case α = 0 (strong scaling) we recover Amdahl's Law

Scalability limitations

- Parallel performance is:

$$P_{\mathrm{v}}^{\mathrm{p}} = \frac{s + pN^{\alpha}}{T_{\mathrm{v}}^{\mathrm{p}}(N)} = \frac{s + (1-s)N^{\alpha}}{s + (1-s)N^{\alpha-1}} = S_{\mathrm{v}}$$

- In the special case **α = 0** (strong scaling) we recover Amdahl's Law
- With **0 < α < 1**, we get for large CPU counts:

$$S_{\mathrm{v}} \xrightarrow{N \gg 1} \frac{s + (1-s)N^{\alpha}}{s} = 1 + \frac{p}{s} N^{\alpha}$$

- In the ideal case **α = 1**, it simplifies to:

$$S_{\mathrm{v}}(\alpha = 1) = s + (1-s)N$$ **Gustafson's Law**

- There is **always** a prefactor that depends on the serial fraction **s**, thus a large serial fraction can lead to a very small slope

Scalability limitations

- **Definition of "work" that only includes the parallel fraction p**
- Serial performance is: $P_v^{sp} = p$
- Parallel performance is: $P_v^{pp} = \frac{pN^\alpha}{T_v^p(N)} = \frac{(1-s)N^\alpha}{s+(1-s)N^{\alpha-1}}$
- Application speedup is: $S_v^p = \frac{P_v^{pp}}{P_v^{sp}} = \frac{N^\alpha}{s+(1-s)N^{\alpha-1}}$
- Speedup and performance are not identical and differ by a factor of p
- The overall work to be done (serial + parallel part) has not changed, scalability makes us believe that suddenly all is well and the application scales perfectly

Parallel efficiency

- Another point of interest is the question how effectively a given resource, i.e. CPU power, can be used in a parallel program
- Parallel efficiency is then defined as:

$$\varepsilon = \frac{\text{performance on } N \text{ CPUs}}{N \times \text{performance on one CPU}} = \frac{\text{speedup}}{N}$$

- In the case where "work" is defined as **$s + pN^{\alpha}$**, we get:

$$\varepsilon = \frac{S_{\mathrm{v}}}{N} = \frac{sN^{-\alpha} + (1-s)}{sN^{1-\alpha} + (1-s)}$$

- For **$\alpha = 0$** this yields **$1/(sN + (1 - s))$**, which is the expected ratio for the Amdahl case and approaches zero with large **N**

Parallel efficiency

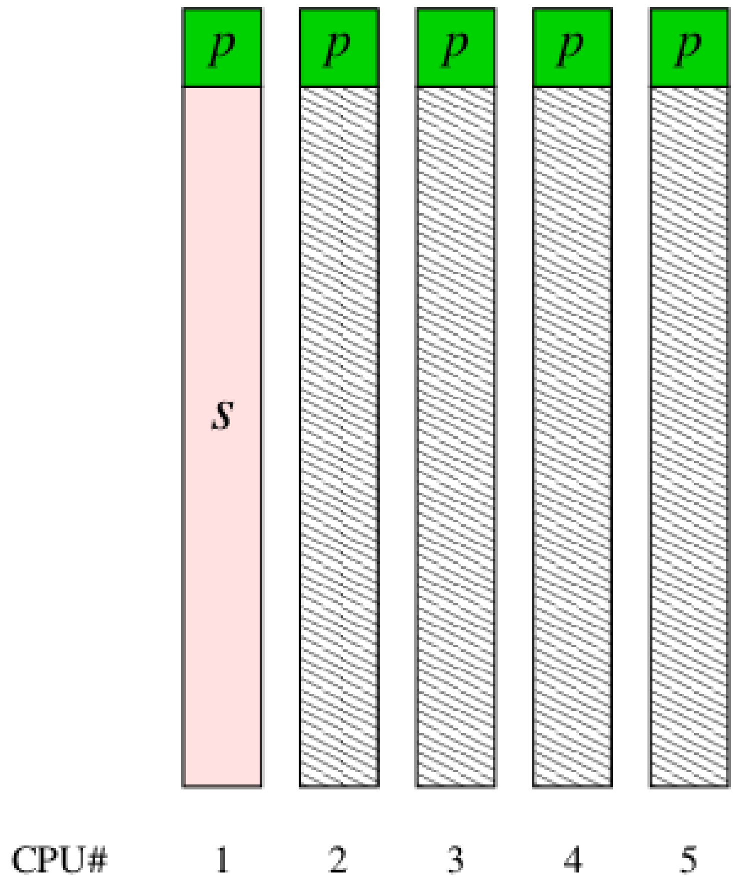
- For **$\alpha = 1$** we get **$s/N + (1-s)$**, which is also correct because the more CPUs are used the more CPU cycles are wasted, and, starting from **$\varepsilon = s+p = 1$** for **$N = 1$**, efficiency reaches a limit of **$1-s = p$** for large **N**
- Wasted CPU time grows linearly with **N**, though, but this issue is clearly visible with the definitions used
- Results change completely when the definition of "work" as **pN^α** is applied

$$\varepsilon_p = \frac{S_v^p}{N} = \frac{N^{\alpha-1}}{s+(1-s)N^{\alpha-1}}$$

- For **$\alpha = 1$** we now get **$\varepsilon_p = 1$**, which would imply perfect efficiency
- But, this is a weak scaling with an inappropriate definition of "work" that includes only the parallelizable part !

Parallel efficiency

- For $\alpha = 1$ we now get $\varepsilon_p = 1$, which would imply perfect efficiency
- It seems that no cycles are wasted with weak scaling
- However, if **s** is large, most of the CPU power is unused
- Although all processors except one are idle 90% of their time, the MFlops/sec rate is a factor of **N** higher when using **N** CPUs.



Refined performance models

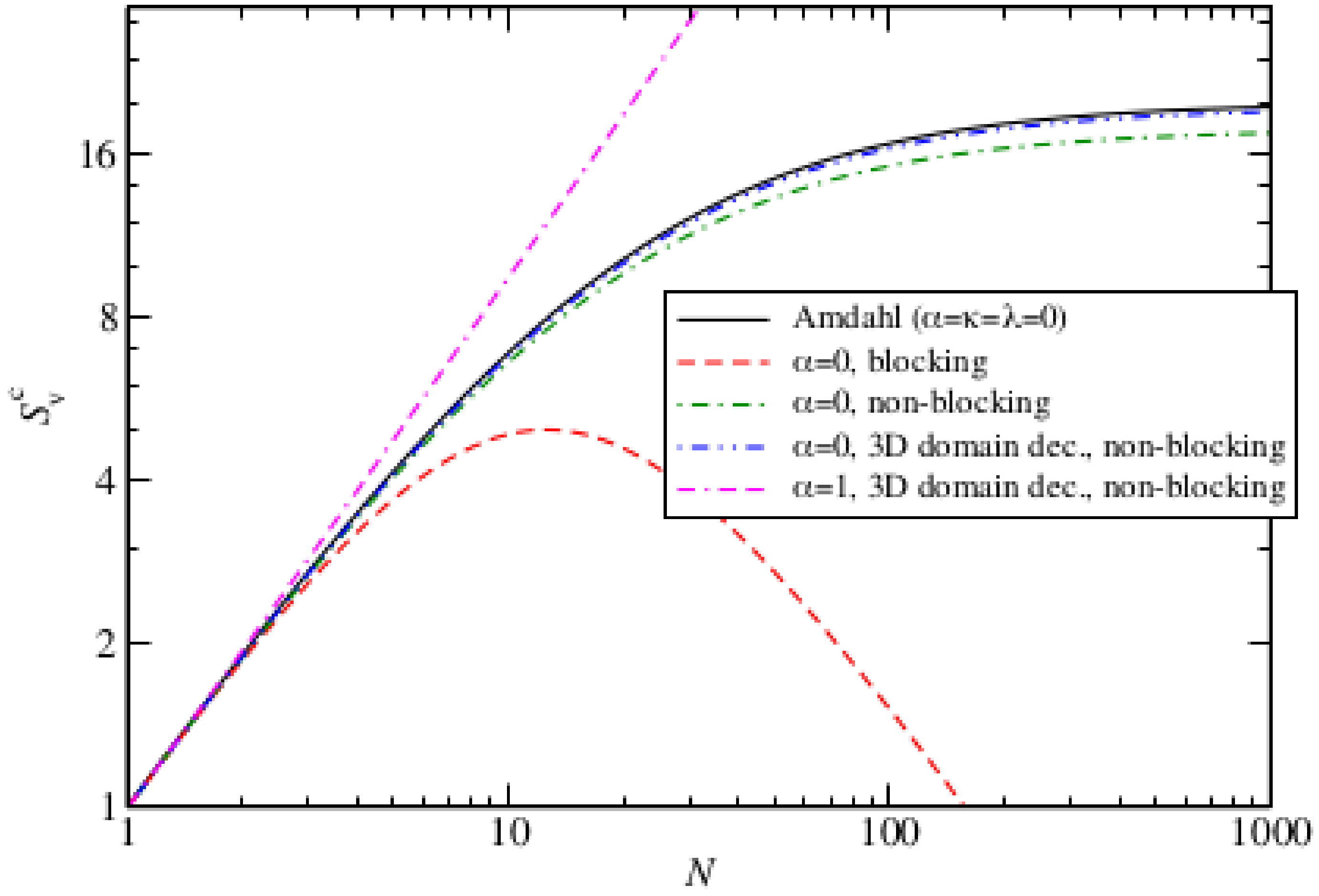
- There are situations where Amdahl’s and Gustafson’s Laws are not appropriate becausen the underlying model does not encompass components like communication, load imbalance, parallel startup overhead etc.
- In a simple communication model, parallel runtime is:

$$T_{\mathrm{v}}^{\mathrm{pc}} = s + pN^{\alpha-1} + c_\alpha(N)$$

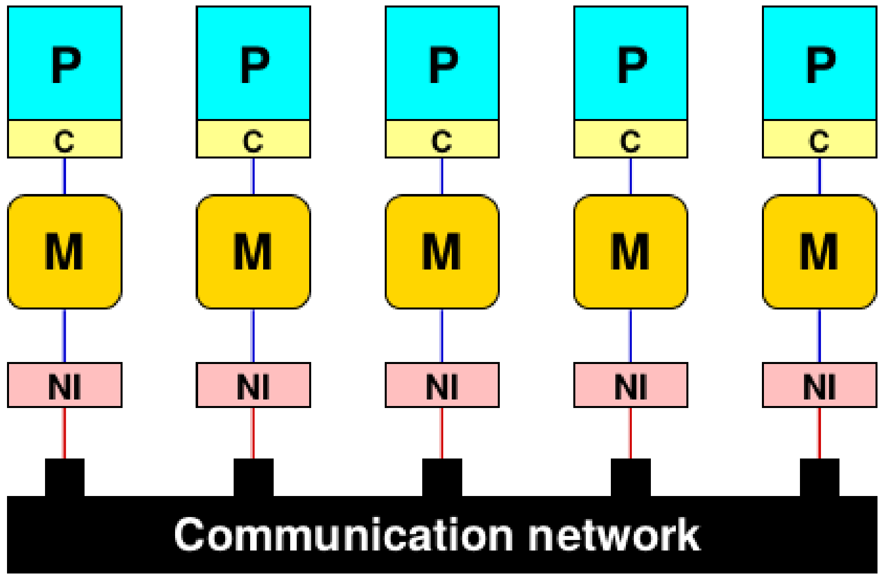
- The communication overhead $c_\alpha(N)$, can have a variety of forms

Refined performance models

- Predicted parallel scalability for different models at s = 0.05



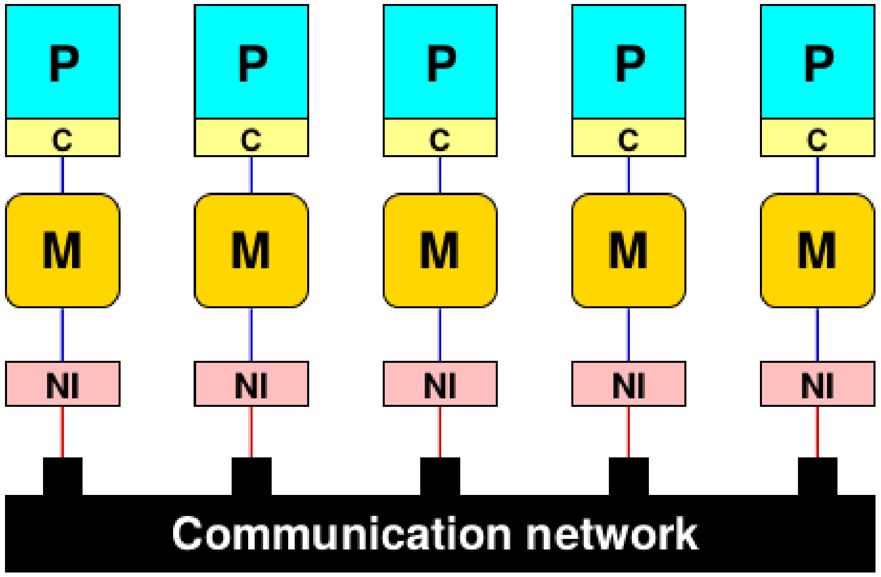
Distributed-memory computing



- **Simplified block diagram (programming model) of a distributed-memory parallel computer**

- Each processor **P** (with its own local cache **C**) is connected to exclusive local memory **M**
- Each node comprises at least one network interface (NI) that mediates the connection to a communication network
- There is a number of advanced technologies that have ten times the bandwidth and 1/10 th of the latency of Gbit Ethernet

Distributed-memory computing



- **Simplified block diagram (programming model) of a distributed-memory parallel computer**

- The most favourable design consists of a non-blocking “wirespeed” network that can switch N/2 connections between its N participants without any bottlenecks
- Problem: non-blocking switch fabrics become vastly expensive on very large installations

Message Passing

- Distributed-memory parallel programming requires the use of explicit message passing (MP), i.e. communication between processes
- There is an established standard for message passing called **MPI** (Message Passing Interface)
- MPI conforms to the following rules:
  - The same program runs on all processes (Single Program Multiple Data - SPMD). All processes taking part in a parallel calculation can be distinguished by a unique identifier called rank
  - The program is written in a sequential language like Fortran, C or C++. Data exchange, i.e. sending and receiving of messages, is done via calls to an appropriate library
  - All variables in a process are local to this process. There is no concept of shared memory

Message Passing

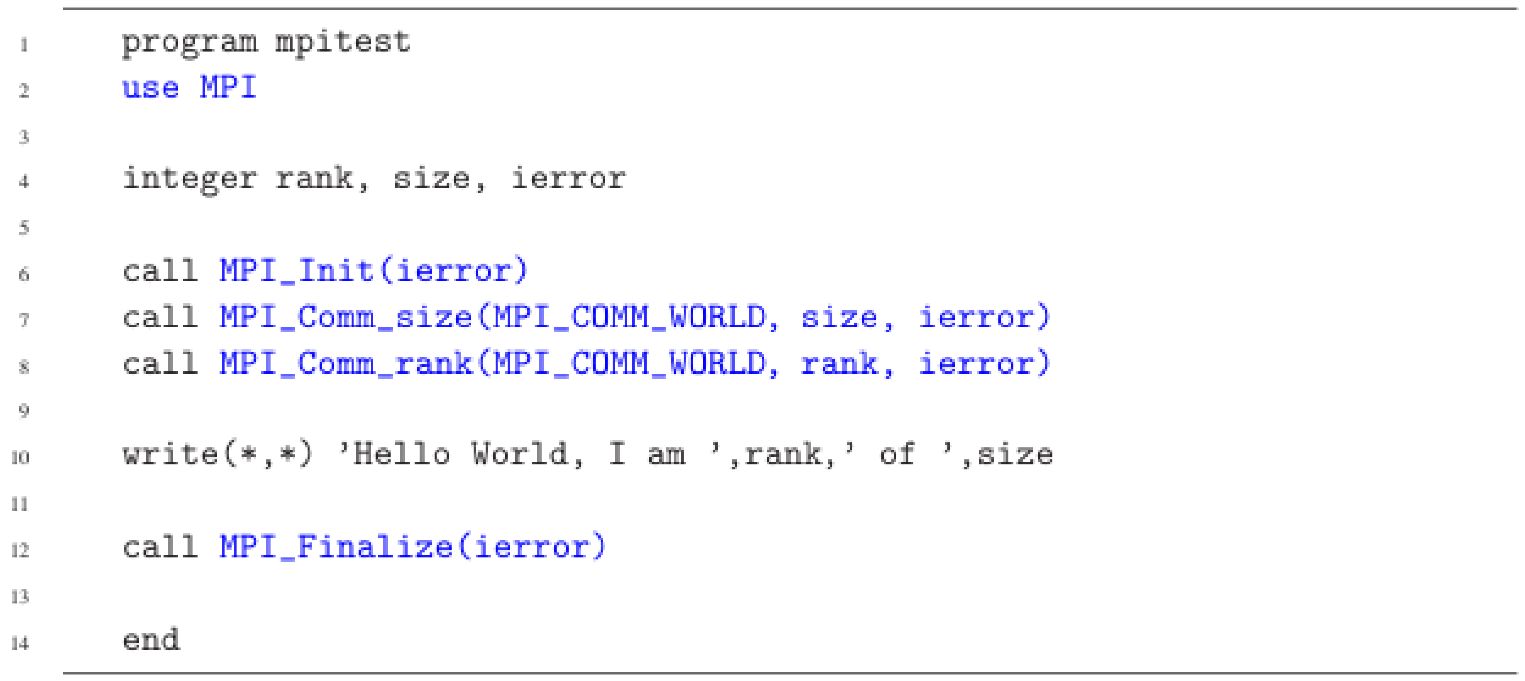
- In a message passing program, messages move data between processes
- For a message to be transmitted in an orderly manner, **some parameters have to be fixed in advance**:
    - Which processor is sending the message
    - Where is the data on the sending processor
    - What kind of data is being sent
    - How much data is there
    - Which process/es is/are going to receive the message
    - Where should the data be left on the receiving process(es)
    - How much data are the receiving processes prepared to accept
- All MPI calls that actually transfer data have to specify those parameters in some way

A brief glance on MPI

```
program mpitest
use MPI

integer rank, size, ierror

call MPI_Init(ierror)
call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

write(*,*) 'Hello World, I am ',rank,' of ',size

call MPI_Finalize(ierror)

end
```

- The first call in every MPI code should go to **MPI_Init** and initializes the parallel environment
- After initialization, MPI has set up a so-called communicator, called **MPI_COMM_WORLD**
- The calls to **MPI_Comm_size** and **MPI_Comm_rank** serve to determine the number of processes (size) in the parallel program and the unique identifier (rank) of the calling process

A brief glance on MPI



```fortran
program mpitest
use MPI

integer rank, size, ierror

call MPI_Init(ierror)
call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

write(*,*) ’Hello World, I am ’,rank,’ of ’,size

call MPI_Finalize(ierror)

end
```

- The ranks in a communicator, in this case **MPI_COMM_WORLD,** are numbered starting from zero up to N−1
- The parallel program is shut down by a call to **MPI_Finalize**
- The output of the program could look like the following:
    - ***Hello World, I am 3 of 4***
    - ***Hello World, I am 0 of 4***
    - ***Hello World, I am 2 of 4***
    - ***Hello World, I am 1 of 4***

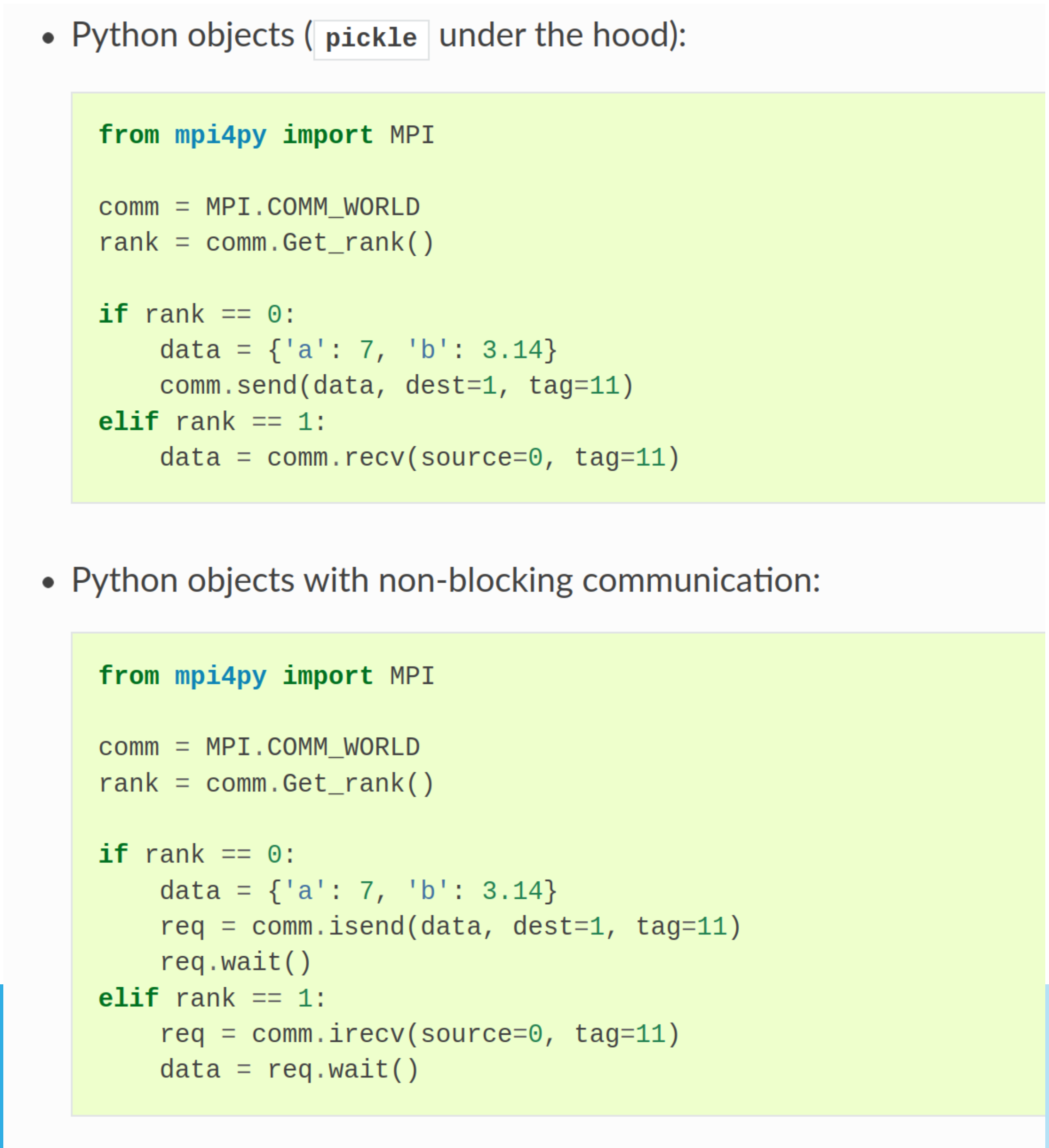
```fortran
   integer stat(MPI_STATUS_SIZE)
   call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
   call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)
! integration limits
   a=0.d0
   b=2.d0
   res=0.d0
! limits for "me"
   mya=a+rank*(b-a)/size
   myb=mya+(b-a)/size
! integrate f(x) over my own chunk - actual work
   psum = integrate(mya,myb)
! rank 0 collects partial results
   if(rank.eq.0) then
      res=psum
      do i=1,size-1
         call MPI_Recv(tmp, &     ! receive buffer
                       1,    &    ! array length
                                  ! datatype
                       MPI_DOUBLE_PRECISION,&
                       i,    &    ! rank of source
                       0,    &    ! tag (additional label)
                                  ! communicator
                       MPI_COMM_WORLD,&
                       stat,&     ! status array (msg info)
                       ierror)
         res=res+tmp
      enddo
      write (*,*) 'Result: ',res
! ranks != 0 send their results to rank 0
   else
      call MPI_Send(psum,   &    ! send buffer
                    1,      &    ! array length
                    MPI_DOUBLE_PRECISION,&
                    0,      &    ! rank of destination
                    0,      &    ! tag
                    MPI_COMM_WORLD,ierror)
   endif
```

- MPI program fragment that computes an integral over some function f(x) in parallel
- Each MPI process gets assigned a subinterval of the integration domain
- Some other function can then perform the actual integration

A brief glance on MPI

- All MPI functionalities described so far have the property that the call returns to the user program only after the message transfer has progressed far enough so that the send/receive buffer can be used without problems
- This is called blocking communication
- Non-blocking MPI is a way in which computation and communication may be overlapped
- Non-blocking and blocking MPI calls are mutually compatible, i.e. a message sent via a blocking send can be matched by a non-blocking receive

A brief glance on MPI - Python



- Python objects (`pickle` under the hood):

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
```

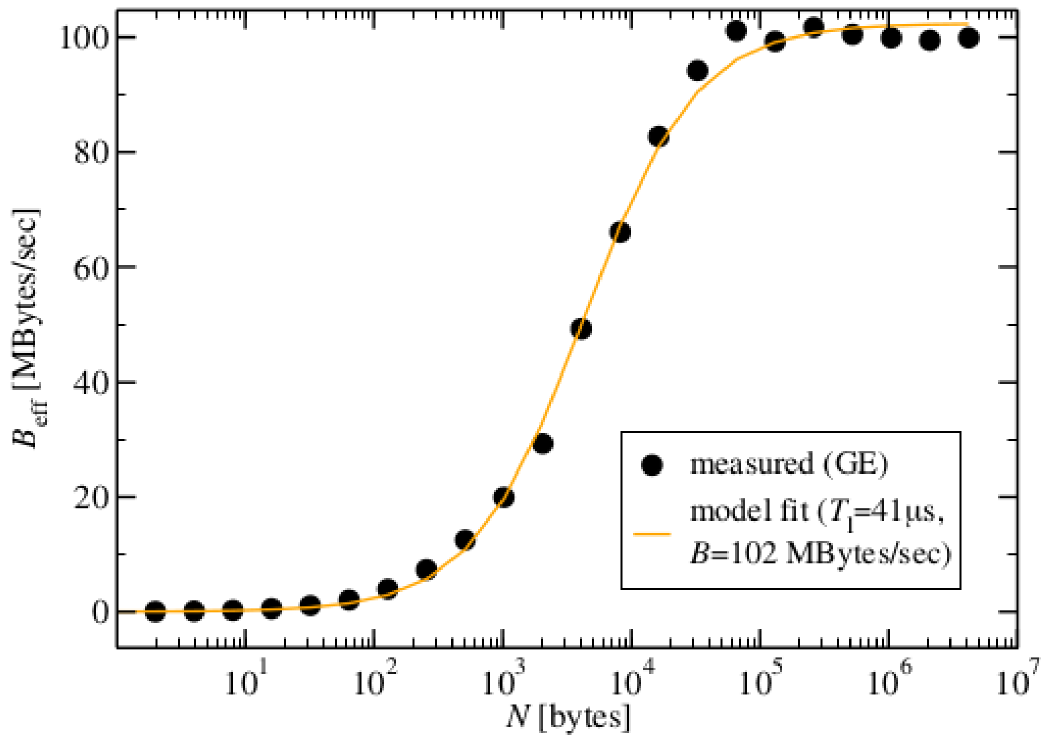
- Python objects with non-blocking communication:

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    req = comm.isend(data, dest=1, tag=11)
    req.wait()
elif rank == 1:
    req = comm.irecv(source=0, tag=11)
    data = req.wait()
```

Basic performance characteristics of networks

- There are various options for the choice of a network in a distributed-memory computer
- Assuming that the total transfer time for a message of size **N** is: $T = T_l + \frac{N}{B}$
-
- The effective bandwidth is: $B_{\text{eff}} = \frac{N}{T_l + \frac{N}{B}}$

Basic performance characteristics of networks



- **Fit of the model for effective bandwidth to data measured on a Gbit Ethernet network**

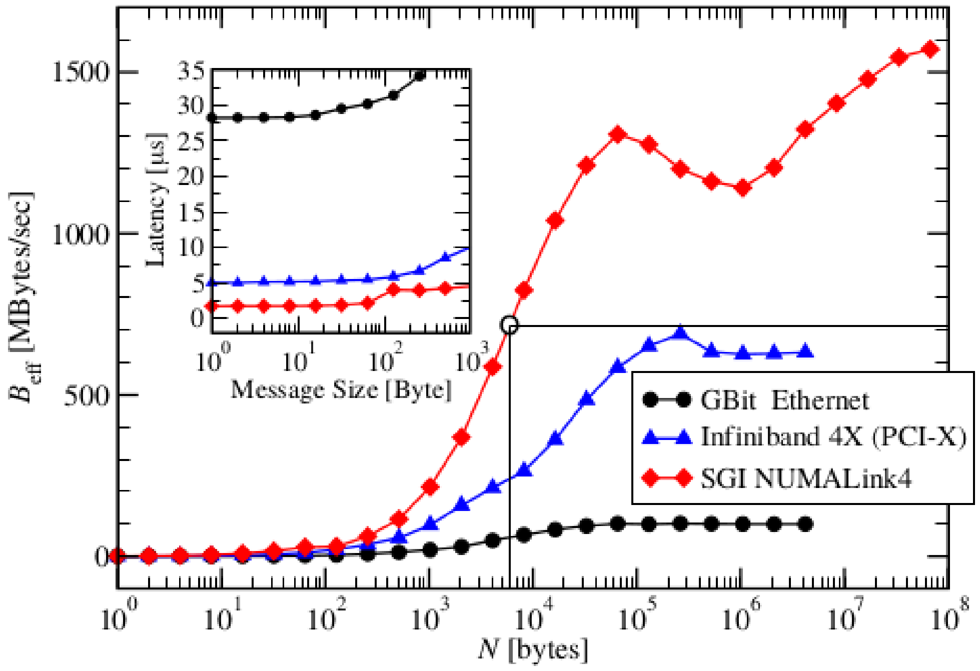
- This simple model is able to describe the gross features well

Basic performance characteristics of networks

- For the measurement of effective bandwidth the PingPong benchmark is frequently used

```
S = get_walltime()
if(rank.eq.0) then
  call MPI_Send(buf,N,MPI_BYTE,1,0,...)
  call MPI_Recv(buf,N,MPI_BYTE,1,0,...)
else
  call MPI_Recv(buf,N,MPI_BYTE,0,0,...)
  call MPI_Send(buf,N,MPI_BYTE,0,0,...)
endif
E = get_walltime()
MBYTES  = 2*N/(E-S)/1.d6      ! MByte/sec rate
TIME    = (E-S)/2*1.d6        ! transfer time in microsecs
                              ! for single message
```

- **Result of the PingPong benchmark for three different networks**



Basic performance characteristics of networks

$$\frac{B_{\text{eff}}(\beta B, T_1)}{B_{\text{eff}}(B, T_1)} = \frac{1 + N/N_{1/2}}{1 + N/\beta N_{1/2}}$$