
Parallel programming / computation

Sultan ALPAR

s.alpar@iitu.edu.kz

IITU

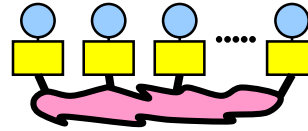
Lecture 1

MPI Overview

Outline

1. MPI Overview

- one program on several processors
- work and data distribution



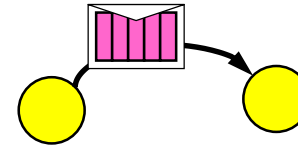
2. Process model and language bindings

- starting several MPI processes

```
MPI_Init()  
MPI_Comm_rank()
```

3. Messages and point-to-point communication

- the MPI processes can communicate



4. Nonblocking communication

- to avoid idle time, deadlocks and serializations

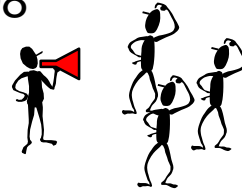


Outline

5. The New Fortran Module mpi_f08

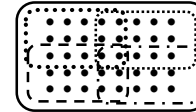
6. Collective communication

- (1) e.g., broadcast
- (2) e.g., nonblocking collectives, neighborhood communic.



7. Error handling

- error handler, codes, and classes

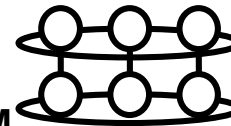


8. Groups & Communicators, Environmental Management

- (1) MPI_Comm_split, intra- & inter-communicators
- (2) Re-numbering on a cluster, collective communication on inter-communicators, info object, naming & attribute caching, implementation information, Sessions Model

9. Virtual topologies

- (1) A multi-dimensional process naming scheme
- (2) Neighborhood communication + MPI_BOTTOM
- (3) Optimization through reordering



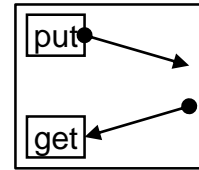
Handout is sorted by content ...

... whereas course is sorted by beginners / intermediate / advanced

Outline

10. One-sided Communication

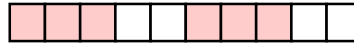
- Windows, remote memory access (RMA)
- Synchronization



11. Shared Memory One-sided Communication

- (1) `MPI_Comm_split_type` & `MPI_Win_allocate_shared`
Hybrid MPI and MPI shared memory programming
- (2) MPI memory models and synchronization rules

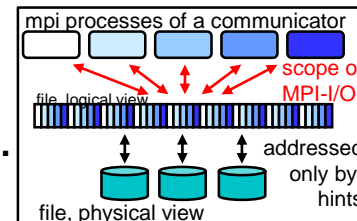
12. Derived datatypes



- (1) transfer any combination of typed data
- (2) advanced features, alignment, resizing

13. Parallel File I/O

- (1) Writing and reading a file in parallel
- (2) Fileviews
- (3) Shared Filepointers, Collective I/O ...



14. MPI and Threads

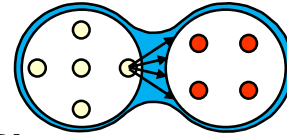
- e.g., hybrid MPI and OpenMP,
partitioned point-to-point communication

Outline

15. Probe, Persistent Requests, Cancel

16. Process Creation and Management

- **Spawning additional processes**
- **Singleton MPI_INIT**
- **Connecting two independent sets of MPI processes**



17. Other MPI features [1, 2, 13.1-3, 15, 16-18, 19.3, A, A.2, B]

18. Best practice

- **Parallelization strategies (e.g. Foster's Design Methodology)**
- **Performance considerations**
- **Pitfalls and progress / weak local**

19. Heat example

Summary

Appendix

Information about MPI

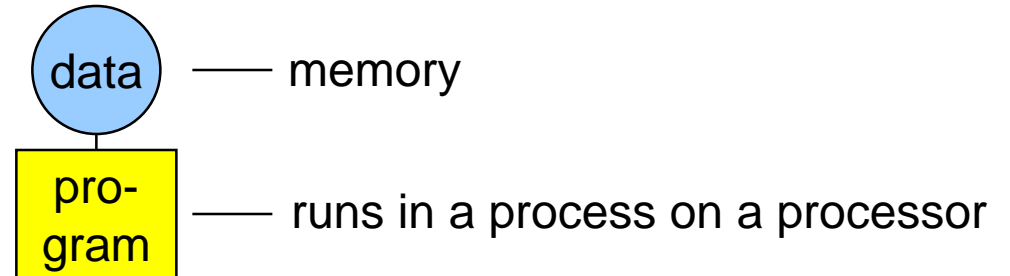


- **MPI: A Message-Passing Interface Standard**, Version 4.0 (June 9, 2021) (pdf & printed hardcover book [MPI-3.1 only] → online via www.mpi-forum.org)
- Marc Snir and William Gropp et al.: **MPI: The Complete Reference**, 1998. (*outdated*)
- William Gropp, Ewing Lusk and Anthony Skjellum: **Using MPI: Portable Parallel Programming With the Message-Passing Interface**. MIT Press, 3rd edition, Nov. 2014 (336 pages, ISBN 9780262527392), and William Gropp, Torsten Hoefler, Rajeev Thakur and Ewing Lusk: **Using Advanced MPI: Modern Features of the Message-Passing Interface**. MIT Press, Nov. 2014 (392 pages, ISBN 9780262527637).
- Peter S. Pacheco: **Parallel Programming with MPI**. Morgan Kaufmann Publishers, 1997 (*very good introduction, can be used as accompanying text for MPI lectures*).
- Neil MacDonald, Elspeth Minty, Joel Malard, Tim Harding, Simon Brown, Mario Antonioletti: **Parallel Programming with MPI**. Historical MPI course notes from EPCC. http://www.archer.ac.uk/training/course-material/2014/10/MPI_UCL/Notes/MPP-notes.pdf
- All MPI standard documents and errata via www.mpi-forum.org
- http://en.wikipedia.org/wiki/Message_Passing_Interface (English)
http://de.wikipedia.org/wiki/Message_Passing_Interface (German)
- **Tools:** see VI-HPS (Virtual Institute – High Productivity Supercomputing) <https://www.vi-hps.org/>
Tools Guide: <https://www.vi-hps.org/cms/upload/material/general/ToolsGuide.pdf> & [training events](#)
- **Python:** See [MPI for Python \(mpi4py.github.io\)](https://mpi4py.github.io/), and [MPI for Python documentation \(mpi4py.readthedocs.io\)](https://mpi4py.readthedocs.io/), and the [Reference \(mpi4py.readthedocs.io/en/stable/reference.html\)](https://mpi4py.readthedocs.io/en/stable/reference.html)

Outdated API reference: mpi4py.github.io/apiref/index.html

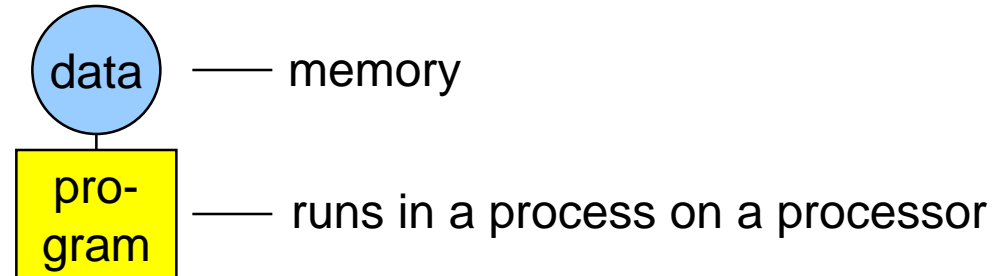
The Message-Passing Programming Paradigm

- Sequential Programming Paradigm

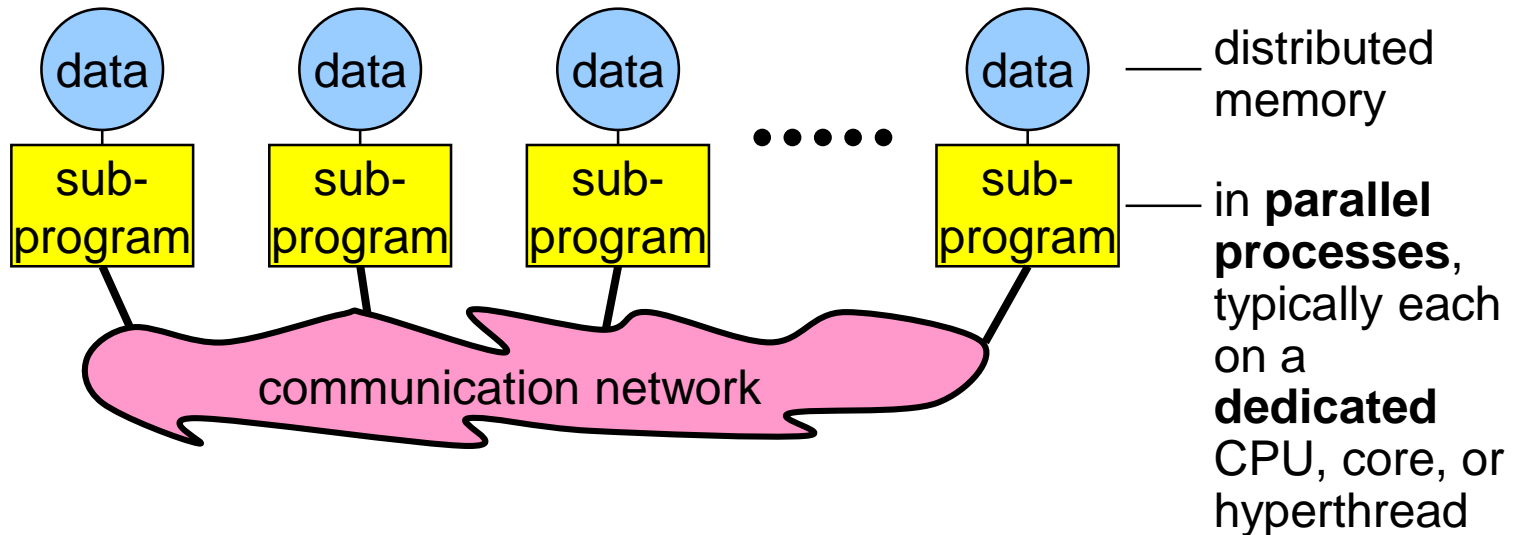


The Message-Passing Programming Paradigm

- Sequential Programming Paradigm

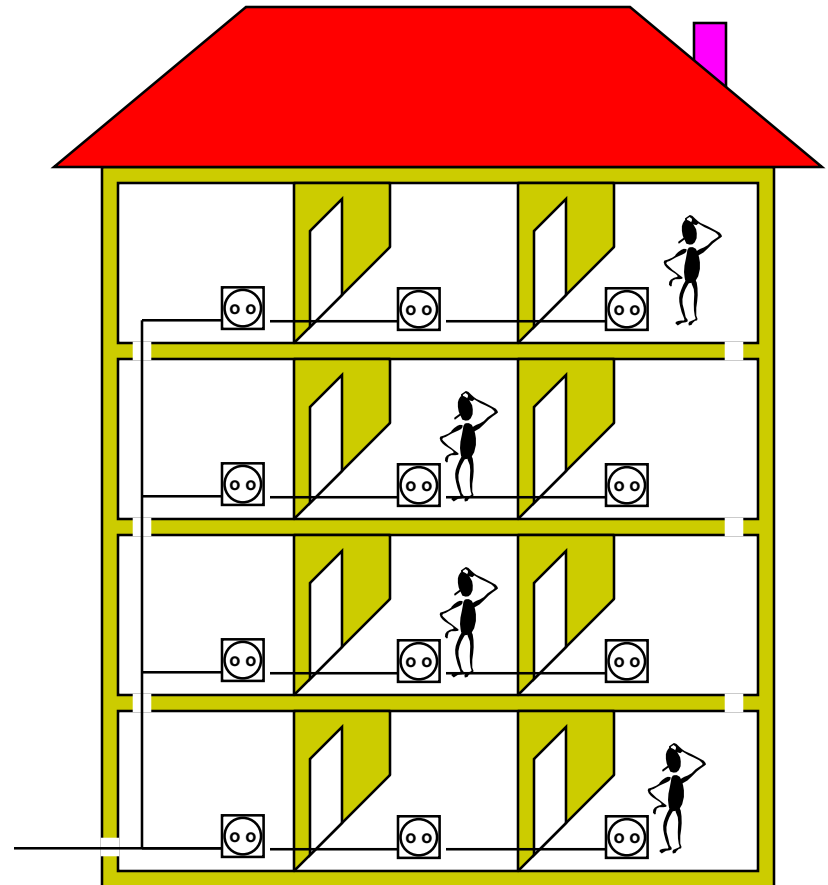


- Message-Passing Programming Paradigm



Analogy: Electric Installations in Parallel

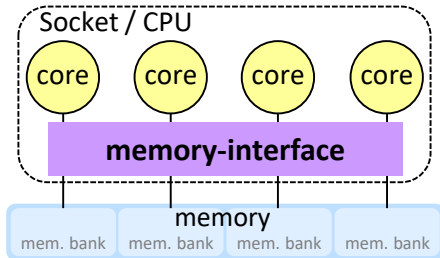
- MPI sub-program
= work of one electrician
on one floor
- MPI process on a dedicated
hardware
= the electrician
- data
= the electric installation
- MPI communication
= real communication
to guarantee that the wires
are coming at the same
position through the floor



Parallel hardware architectures



shared memory



Socket/CPU

→ memory interface

UMA (uniform memory access)

SMP (symmetric multi-processing)

All cores connected to all memory banks with same speed

Parallel execution streams on each core, e.g.,

$x[0 \dots 999] = \dots$ on 1st core

$x[1000 \dots 1999] = \dots$ on 2nd core

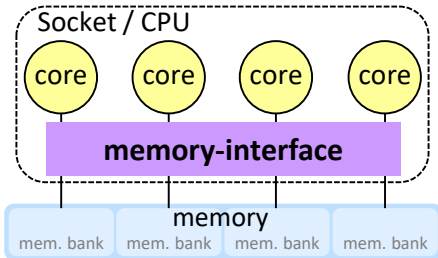
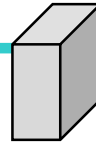
$x[2000 \dots 2999] = \dots$ on 3rd core

...

Parallel hardware architectures



shared memory



Socket/CPU

→ memory interface

UMA (uniform memory access)

SMP (symmetric multi-processing)

All cores connected to all memory banks with same speed

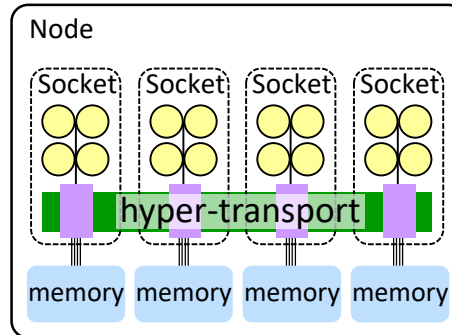
Parallel execution streams on each core, e.g.,

x[0 ... 999] = ... on 1st core

x[1000 ... 1999] = ... on 2nd core

x[2000 ... 2999] = ... on 3rd core

...



Node

→ hyper-transport

ccNUMA (cache-coherent non-uniform memory access)

→ Shared memory programming is possible

!! #CPUs x memory bandwidth !!

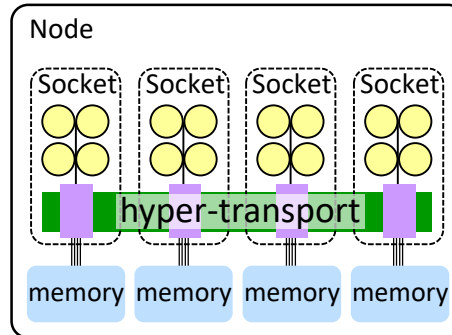
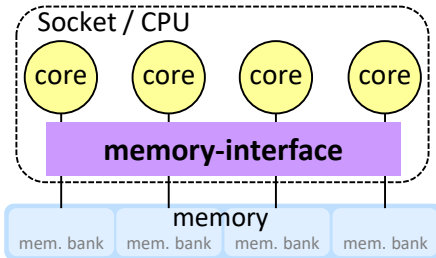
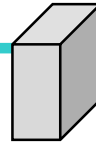
Performance problems:

- Each **parallel execution stream** should mainly access the memory of **its** CPU
→ **First-touch** strategy is needed to minimize remote memory access
- Threads should be **pinned** to the physical sockets

Parallel hardware architectures



shared memory



Socket/CPU

→ **memory interface**

UMA (uniform memory access)
SMP (symmetric multi-processing)
All cores connected to all memory banks with same speed

Parallel execution streams on each core, e.g.,
x[0 ... 999] = ... on 1st core
x[1000 ... 1999] = ... on 2nd core
x[2000 ... 2999] = ... on 3rd core
...

Node

→ **hyper-transport**

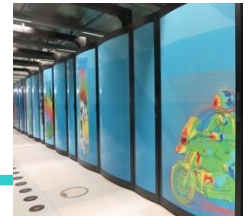
ccNUMA (cache-coherent non-uniform memory access)
→ Shared memory programming is possible
!! #CPUs x memory bandwidth !!

Performance problems:

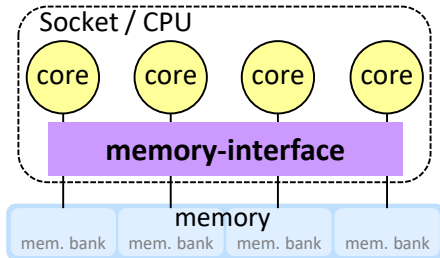
- Each **parallel execution stream** should mainly access the memory of **its** CPU
→ **First-touch** strategy is needed to minimize remote memory access
- Threads should be **pinned** to the physical sockets

Shared memory programming with OpenMP

Parallel hardware architectures



shared memory



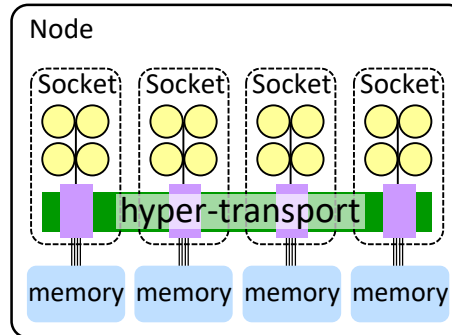
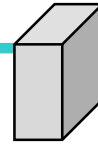
Socket/CPU

→ **memory interface**

UMA (uniform memory access)
SMP (symmetric multi-processing)
 All cores connected to all memory banks with same speed

Parallel execution streams on each core, e.g.,
 x[0 ... 999] = ... on 1st core
 x[1000 ... 1999] = ... on 2nd core
 x[2000 ... 2999] = ... on 3rd core
 ...

Shared memory programming with OpenMP



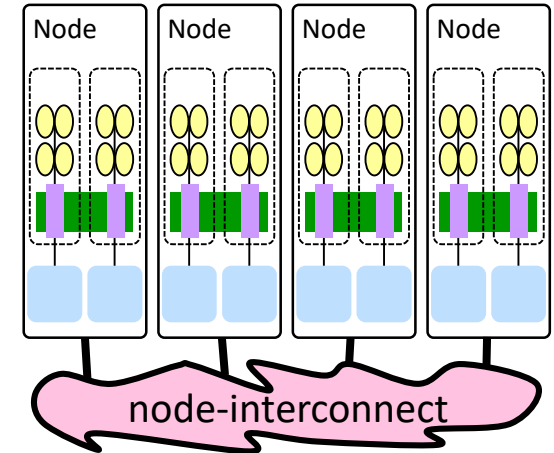
Node

→ **hyper-transport**

ccNUMA (cache-coherent non-uniform memory access)
 → Shared memory programming is possible
!! #CPUs x memory bandwidth !!
Performance problems:

- Each **parallel execution stream** should mainly access the memory of **its** CPU
 → **First-touch** strategy is needed to minimize remote memory access
- Threads should be **pinned** to the physical sockets

distributed memory



Cluster

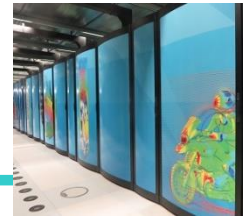
→ **node-interconnect**

NUMA (non-uniform memory access)
 !! fast access only on its own memory !!

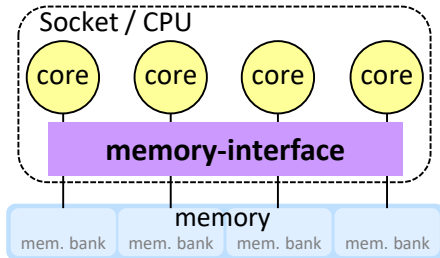
Many programming options:

- Shared memory / symmetric multi-processing inside of each node
- distributed memory parallelization on the node interconnect
- **Or simply one MPI process on each core**

Parallel hardware architectures



shared memory



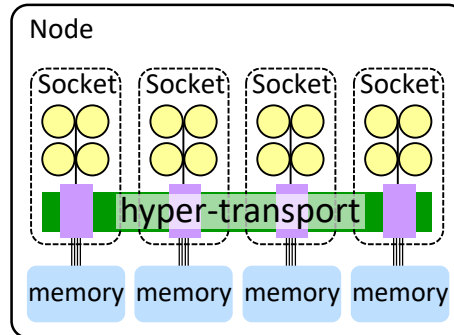
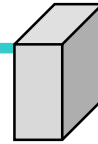
Socket/CPU

→ memory interface

UMA (uniform memory access)
SMP (symmetric multi-processing)
 All cores connected to all memory banks with same speed

Parallel execution streams on each core, e.g.,
 $x[0 \dots 999] = \dots$ on 1st core
 $x[1000 \dots 1999] = \dots$ on 2nd core
 $x[2000 \dots 2999] = \dots$ on 3rd core
 ...

Shared memory programming with OpenMP



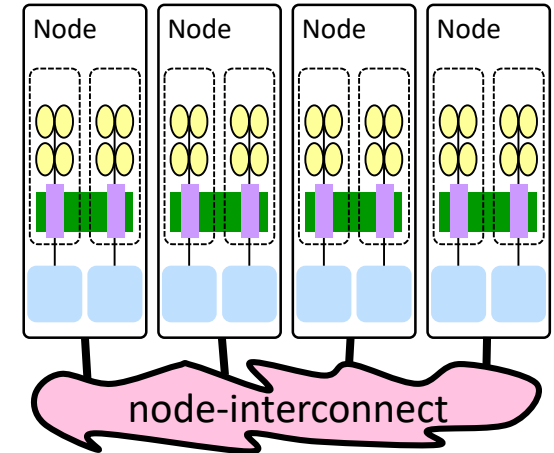
Node

→ hyper-transport

ccNUMA (cache-coherent non-uniform memory access)
 → Shared memory programming is possible
!! #CPUs x memory bandwidth !!
Performance problems:

- Each **parallel execution stream** should mainly access the memory of **its** CPU
 → **First-touch** strategy is needed to minimize remote memory access
- Threads should be **pinned** to the physical sockets

distributed memory



Cluster

→ node-interconnect

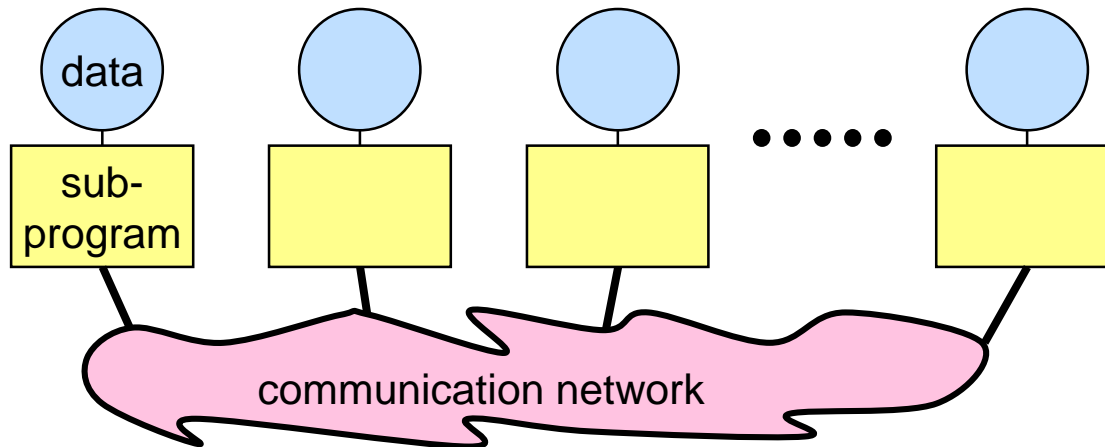
NUMA (non-uniform memory access)
!! fast access only on its own memory !!
Many programming options:

- Shared memory / symmetric multi-processing inside of each node
- distributed memory parallelization on the node interconnect
- **Or simply one MPI process on each core**

MPI works everywhere

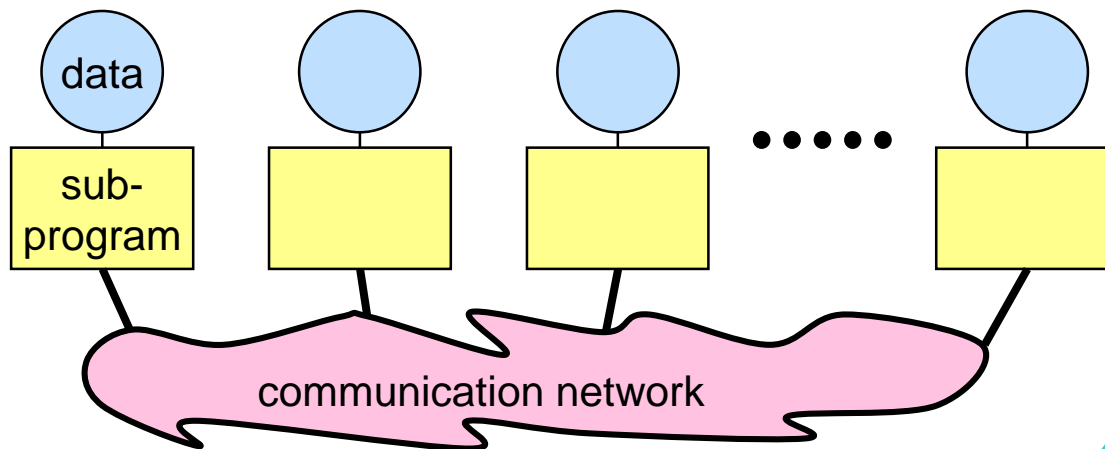
The Message-Passing Programming Paradigm

- 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),
 - the variables of each sub-program have
 - the same name
 - but different locations (distributed memory) and different data!
 - i.e., all variables are private
 - communicate via special send & receive routines (***message passing***)



The Message-Passing Programming Paradigm

- 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),
 - the variables of each sub-program have
 - the same name
 - but different locations (distributed memory) and different data!
 - i.e., all variables are private
 - communicate via special send & receive routines (**message passing**)

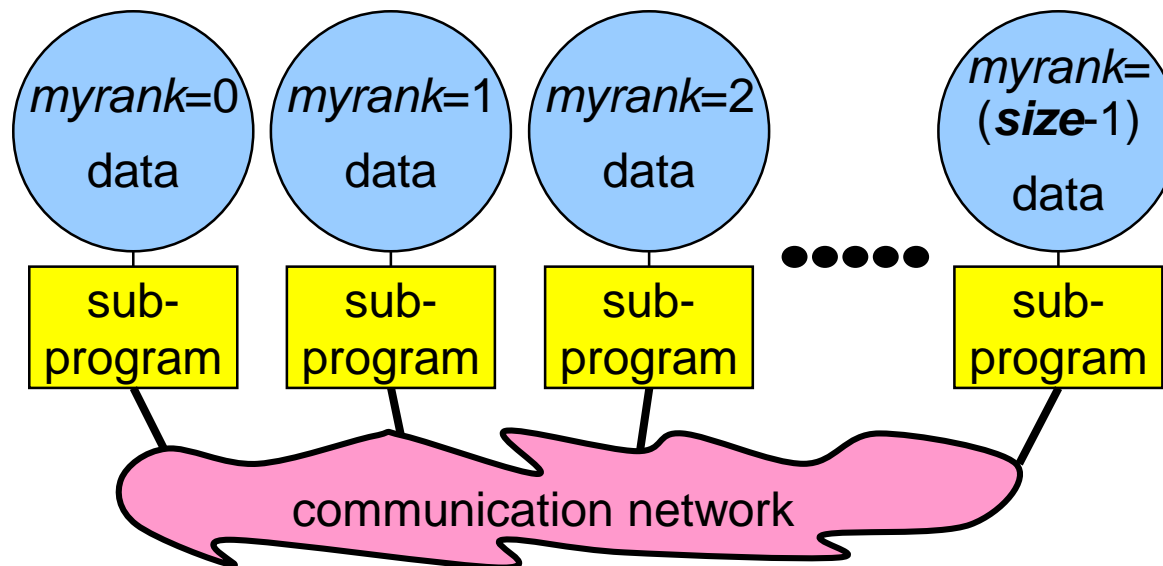


Caution

- completely different model compared to
- OpenMP
 - Python with concurrent futures
 - pthreads, shmem

Data and Work Distribution

- the value of *myrank* is returned by special library routine
- the system of *size* processes is started by special MPI initialization program (mpirun or mpiexec)
- all distribution decisions are based on *myrank*
- i.e., which process works on which data



What is SPMD?

- **S**ingle **P**rogram, **M**ultiple **D**ata
- Same (sub-)program runs on each processor

- MPI allows also MPMD, i.e., **M**ultiple **P**rogram, ...
- but some vendors may be restricted to SPMD
- MPMD can be emulated with SPMD

Emulation of Multiple Program (MPMD), Example

- ```
main(int argc, char **argv)
{
 if (myrank < /* process should run the ocean model */)
 {
 ocean(/* arguments */);
 }else{
 weather(/* arguments */);
 }
}
```

- 
- ```
PROGRAM
IF (myrank < ... ) THEN  !! process should run the ocean model
    CALL ocean ( some arguments )
ELSE
    CALL weather ( some arguments )
ENDIF
END
```

-
- ```
if (myrank <): # process should run the ocean model
 ocean(...)
else:
 weather(...)
```

# Emulation of Multiple Program (MPMD), Example

---

- ```
main(int argc, char **argv)
{
    if (myrank < .... /* process should run the ocean model */)
    {
        ocean( /* arguments */ );
    }else{
        weather( /* arguments */ );
    }
}
```

-
- ```
PROGRAM
IF (myrank < ...) THEN !! process should run the ocean model
 CALL ocean (some arguments)
ELSE
 CALL weather (some arguments)
ENDIF
END
```

- 
- ```
if (myrank < .... ):    # process should run the ocean model
    ocean( ... )
else:
    weather( ... )
```

**example for usage of
sub-groups of processes (→ Ch. 8)**

first-example.c



```
#include <stdio.h> first-example.c
```

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

```
int main(int argc, char *argv[])
```

```
{
```

```
MPI_Init(&argc, &argv);
```

```
MPI_Finalize();
```

```
} MPI course → Chap. 1 Overview
```

```
#include <stdio.h> first-example.c
```

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

```
int main(int argc, char *argv[])
```

```
{
```

```
  MPI_Init(&argc, &argv);
```

```
  MPI_Finalize();
```

```
}
```

Compiled, e.g., with: mpicc first-example.c

Started, e.g., with: mpiexec -n 4 ./a.out

Then, this code is running 4 times in parallel !

first-example.c

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

Compiled, e.g., with: mpicc first-example.c

Started, e.g., with: mpiexec -n 4 ./a.out

Then, this code is running 4 times in parallel !

```
int my_rank, num_procs; MPI-related data
```

```
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
```

Now, each process knows who it is:
number *my_rank* out of *num_procs* processes

```
    MPI_Finalize();
```


first-example.c

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[])
{
  int n; double result;
  int my_rank, num_procs;
```

Compiled, e.g., with: mpicc first-example.c

Started, e.g., with: mpiexec -n 4 ./a.out

Then, this code is running 4 times in parallel !

application-related data

MPI-related data

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
```

Now, each process knows who it is:
number *my_rank* out of *num_procs* processes

```
MPI_Finalize();
```

first-example.c

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[])
{
  int n; double result;
  int my_rank, num_procs;
```

Compiled, e.g., with: `mpicc first-example.c`
Started, e.g., with: `mpiexec -n 4 ./a.out`

Then, this code is running 4 times in parallel !

application-related data

MPI-related data

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
```

Now, each process knows who it is:
number *my_rank* out of *num_procs* processes

```
if (my_rank == 0)
{ printf("Enter the number of elements (n): \n");
  scanf("%d",&n);
}
```

reading the application data *n* from stdin only
by process 0

```
MPI_Finalize();
```

```
Enter the number of elements (n):
100
```

input/output

first-example.c

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[])
{
  int n; double result;
  int my_rank, num_procs;
```

Compiled, e.g., with: `mpicc first-example.c`
Started, e.g., with: `mpiexec -n 4 ./a.out`

Then, this code is running 4 times in parallel !

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
```

Now, each process knows who it is:
number *my_rank* out of *num_procs* processes

```
if (my_rank == 0)
{ printf("Enter the number of elements (n): \n");
  scanf("%d", &n);
}
```

reading the application data *n* from stdin only
by process 0

process 0 is sender, all other
processes are receivers

```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

broadcasting the content of variable *n* in process 0
into variables *n* in all other processes

```
MPI_Finalize();
```

```
Enter the number of elements (n):
100
```

input/output

first-example.c

Compiled, e.g., with: `mpicc first-example.c`
Started, e.g., with: `mpiexec -n 4 ./a.out`

Then, this code is running 4 times in parallel !

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

```
    int n; double result;
    int my_rank, num_procs;
```

```
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
```

```
    if (my_rank == 0)
    { printf("Enter the number of elements (n): \n");
      scanf("%d", &n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```
    result = 1.0 * my_rank * n;
```

```
    MPI_Finalize();
```

Now, each process knows who it is:
number *my_rank* out of *num_procs* processes

reading the application data *n* from stdin only
by process 0

process 0 is sender, all other
processes are receivers

broadcasting the content of variable *n* in process 0
into variables *n* in all other processes

doing some **application work** in each process

```
Enter the number of elements (n):
100
```

input/output

Slide 17 □

first-example.c

Compiled, e.g., with: mpicc first-example.c
Started, e.g., with: mpiexec -n 4 ./a.out

Then, this code is running 4 times in parallel !

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[])
{
  int n; double result;
  int my_rank, num_procs;
```

application-related data
MPI-related data

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
```

Now, each process knows who it is:
number *my_rank* out of *num_procs* processes

```
if (my_rank == 0)
{ printf("Enter the number of elements (n): \n");
  scanf("%d", &n);
}
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

reading the application data *n* from stdin only
by process 0

process 0 is sender, all other
processes are receivers

broadcasting the content of variable *n* in process 0
into variables *n* in all other processes

```
result = 1.0 * my_rank * n;
printf("I am process %i out of %i handling the %ith part of n=%i elements, result=%f\n",
       my_rank, num_procs, my_rank, n, result);
```

doing some **application work** in each process

```
MPI_Finalize();
```

input/output

```
Enter the number of elements (n):
100
I am process 0 out of 4 handling the 0th part of n=100 elements, result=0.0
I am process 2 out of 4 handling the 2th part of n=100 elements, result=200.0
I am process 3 out of 4 handling the 3th part of n=100 elements, result=300.0
I am process 1 out of 4 handling the 1th part of n=100 elements, result=100.0
```

first-example.c

Compiled, e.g., with: mpicc first-example.c
Started, e.g., with: mpiexec -n 4 ./a.out

Then, this code is running 4 times in parallel !

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[])
{
  int n; double result;
  int my_rank, num_procs;
```

application-related data
MPI-related data

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
```

Now, each process knows who it is:
number *my_rank* out of *num_procs* processes

```
if (my_rank == 0)
{ printf("Enter the number of elements (n): \n");
  scanf("%d",&n);
}
```

reading the application data *n* from stdin only
by process 0

process 0 is sender, all other
processes are receivers

```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

broadcasting the content of variable *n* in process 0
into variables *n* in all other processes

```
result = 1.0 * my_rank * n;
printf("I am process %i out of %i handling the %ith part of n=%i elements, result=%f\n",
       my_rank, num_procs, my_rank, n, result);
```

doing some **application work** in each process

```
if (my_rank != 0)
{ MPI_Send(&result, 1, MPI_DOUBLE, 0, 99, MPI_COMM_WORLD);
}
else
```

send to process 0

sending some results from
all processes (except 0) to process 0

```
MPI_Finalize();
```

input/output

```
Enter the number of elements (n):
100
I am process 0 out of 4 handling the 0th part of n=100 elements, result=0.0
I am process 2 out of 4 handling the 2th part of n=100 elements, result=200.0
I am process 3 out of 4 handling the 3th part of n=100 elements, result=300.0
I am process 1 out of 4 handling the 1th part of n=100 elements, result=100.0
```

first-example.c

Compiled, e.g., with: mpicc first-example.c
Started, e.g., with: mpiexec -n 4 ./a.out

Then, this code is running 4 times in parallel !

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[])
{
  int n; double result;
  int my_rank, num_procs;
```

application-related data
MPI-related data

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
```

Now, each process knows who it is: number *my_rank* out of *num_procs* processes

```
if (my_rank == 0)
{ printf("Enter the number of elements (n): \n");
  scanf("%d",&n);
}
```

reading the application data *n* from stdin only by process 0

```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

process 0 is sender, all other processes are receivers

broadcasting the content of variable *n* in process 0 into variables *n* in all other processes

```
result = 1.0 * my_rank * n;
printf("I am process %i out of %i handling the %ith part of n=%i elements, result=%f\n",
       my_rank, num_procs, my_rank, n, result);
```

doing some **application work** in each process

```
if (my_rank != 0)
{ MPI_Send(&result, 1, MPI_DOUBLE, 0, 99, MPI_COMM_WORLD);
}
```

send to process 0

sending some results from all processes (except 0) to process 0

```
else
```

Process 0: receiving all these messages and, e.g., printing them

```
MPI_Finalize();
```

input/output

```
Enter the number of elements (n):
100
I am process 0 out of 4 handling the 0th part of n=100 elements, result=0.0
I am process 2 out of 4 handling the 2th part of n=100 elements, result=200.0
I am process 3 out of 4 handling the 3th part of n=100 elements, result=300.0
I am process 1 out of 4 handling the 1th part of n=100 elements, result=100.0
```

well sorted output from process 0

first-example.c

Compiled, e.g., with: mpicc first-example.c
Started, e.g., with: mpiexec -n 4 ./a.out

Then, this code is running 4 times in parallel !

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

```
{
  int n; double result;
  int my_rank, num_procs;
```

```
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
```

```
  if (my_rank == 0)
  { printf("Enter the number of elements (n): \n");
    scanf("%d",&n);
  }
```

```
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```
  result = 1.0 * my_rank * n;
  printf("I am process %i out of %i handling the %ith part of n=%i elements,result=%f\n",
         my_rank, num_procs, my_rank, n, result);
```

```
  if (my_rank != 0)
  { MPI_Send(&result,1,MPI_DOUBLE,0,99,MPI_COMM_WORLD);
  }
  else
  { int rank;
    printf("I'm proc 0: My own result is %f \n",result);
  }
```

```
  MPI_Finalize();
}
```

application-related data

MPI-related data

Now, each process knows who it is:
number *my_rank* out of *num_procs* processes

reading the application data *n* from stdin only
by process 0

process 0 is sender, all other
processes are receivers

broadcasting the content of variable *n* in process 0
into variables *n* in all other processes

doing some **application work** in each process

send to process 0

sending some results from
all processes (except 0) to process 0

Process 0: receiving all these messages and, e.g., printing them

Enter the number of elements (n):
100

input/output

I am process 0 out of 4 handling the 0th part of n=100 elements, result=0.0
I am process 2 out of 4 handling the 2th part of n=100 elements, result=200.0
I am process 3 out of 4 handling the 3th part of n=100 elements, result=300.0
I am process 1 out of 4 handling the 1th part of n=100 elements, result=100.0
I'm proc 0: My own result is 0.0

well sorted output
from process 0

first-example.c

Compiled, e.g., with: mpicc first-example.c
Started, e.g., with: mpiexec -n 4 ./a.out

Then, this code is running 4 times in parallel !

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[])
{
  int n; double result;
  int my_rank, num_procs;
```

application-related data

MPI-related data

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
```

Now, each process knows who it is: number *my_rank* out of *num_procs* processes

```
if (my_rank == 0)
{ printf("Enter the number of elements (n): \n");
  scanf("%d",&n);
}
```

reading the application data *n* from stdin only by process 0

process 0 is sender, all other processes are receivers

```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

broadcasting the content of variable *n* in process 0 into variables *n* in all other processes

```
result = 1.0 * my_rank * n;
printf("I am process %i out of %i handling the %ith part of n=%i elements, result=%f\n",
       my_rank, num_procs, my_rank, n, result);
```

doing some application work in each process

```
if (my_rank != 0)
{ MPI_Send(&result, 1, MPI_DOUBLE, 0, 99, MPI_COMM_WORLD);
}
```

send to process 0

sending some results from all processes (except 0) to process 0

```
else
{ int rank;
  printf("I'm proc 0: My own result is %f \n",result);
  for (rank=1; rank<num_procs; rank++)
  {
  }
}
```

Process 0: receiving all these messages and, e.g., printing them

```
MPI_Finalize();
```

input/output

```
Enter the number of elements (n):
100
I am process 0 out of 4 handling the 0th part of n=100 elements, result=0.0
I am process 2 out of 4 handling the 2th part of n=100 elements, result=200.0
I am process 3 out of 4 handling the 3th part of n=100 elements, result=300.0
I am process 1 out of 4 handling the 1th part of n=100 elements, result=100.0
I'm proc 0: My own result is 0.0
```

well sorted output from process 0

first-example.c

Compiled, e.g., with: mpicc first-example.c
Started, e.g., with: mpiexec -n 4 ./a.out

Then, this code is running 4 times in parallel !

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[])
{
  int n; double result;
  int my_rank, num_procs;
```

application-related data
MPI-related data

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
```

Now, each process knows who it is:
number *my_rank* out of *num_procs* processes

```
if (my_rank == 0)
{ printf("Enter the number of elements (n): \n");
  scanf("%d",&n);
}
```

reading the application data *n* from stdin only
by process 0

```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

process 0 is sender, all other
processes are receivers

broadcasting the content of variable *n* in process 0
into variables *n* in all other processes

```
result = 1.0 * my_rank * n;
printf("I am process %i out of %i handling the %ith part of n=%i elements,result=%f\n",
       my_rank, num_procs, my_rank, n, result);
```

doing some **application work** in each process

```
if (my_rank != 0)
{ MPI_Send(&result,1,MPI_DOUBLE,0,99,MPI_COMM_WORLD);
}
```

send to process 0

sending some results from
all processes (except 0) to process 0

```
else
{ int rank;
  printf("I'm proc 0: My own result is %f \n",result);
  for (rank=1; rank<num_procs; rank++)
  {
    MPI_Recv(&result,1,MPI_DOUBLE,rank,99,
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }
}
```

Process 0: receiving all these messages and, e.g., printing them

receiving the message from process *rank*

```
Enter the number of elements (n):
100
I am process 0 out of 4 handling the 0th part of n=100 elements, result=0.0
I am process 2 out of 4 handling the 2th part of n=100 elements, result=200.0
I am process 3 out of 4 handling the 3th part of n=100 elements, result=300.0
I am process 1 out of 4 handling the 1th part of n=100 elements, result=100.0
I'm proc 0: My own result is 0.0
```

input/output

well sorted output
from process 0

```
MPI_Finalize();
```

first-example.c

Compiled, e.g., with: `mpicc first-example.c`
Started, e.g., with: `mpiexec -n 4 ./a.out`

Then, this code is running 4 times in parallel !

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[])
{
    int n; double result;
    int my_rank, num_procs;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

    if (my_rank == 0)
    { printf("Enter the number of elements (n): \n");
      scanf("%d",&n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    result = 1.0 * my_rank * n;
    printf("I am process %i out of %i handling the %ith part of n=%i elements, result=%f\n",
           my_rank, num_procs, my_rank, n, result);

    if (my_rank != 0)
    { MPI_Send(&result, 1, MPI_DOUBLE, 0, 99, MPI_COMM_WORLD);
    }
    else
    { int rank;
      printf("I'm proc 0: My own result is %f \n",result);
      for (rank=1; rank<num_procs; rank++)
      {
          MPI_Recv(&result, 1, MPI_DOUBLE, rank, 99,
                  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
          printf("I'm proc 0: received result of
                process %i is %f \n", rank, result);
      }
    }

    MPI_Finalize();
}
```

application-related data

MPI-related data

Now, each process knows who it is:
number *my_rank* out of *num_procs* processes

reading the application data *n* from stdin only
by process 0

process 0 is sender, all other
processes are receivers

broadcasting the content of variable *n* in process 0
into variables *n* in all other processes

doing some **application work** in each process

send to process 0

sending some results from
all processes (except 0) to process 0

Process 0: receiving all these messages and, e.g., printing them

receiving the message from process *rank*

```
Enter the number of elements (n):
100
I am process 0 out of 4 handling the 0th part of n=100 elements, result=0.0
I am process 2 out of 4 handling the 2th part of n=100 elements, result=200.0
I am process 3 out of 4 handling the 3th part of n=100 elements, result=300.0
I am process 1 out of 4 handling the 1th part of n=100 elements, result=100.0
I'm proc 0: My own result is 0.0
I'm proc 0: received result of process 1 is 100.0
I'm proc 0: received result of process 2 is 200.0
I'm proc 0: received result of process 3 is 300.0
```

input/output

well sorted output
from process 0

Exercise 1

Please run this first example on your exercise system:

Already done as part of the course preparation

Exercise 1

C

cd **MPI/tasks/C/Ch1**

Fortran

cd **MPI/tasks/F_30/Ch1**

Python

cd **MPI/tasks/PY/Ch1**

Basis for the entire course

- Initialize your compile and MPI environment, e.g., module load gnu openmpi (*on my system* 😊)
- ls -l → **first-example.c** or **first-example_30.f90** or **first-example.py**

Caution: OpenMPI is an MPI library, whereas Open**MP** is a shared memory parallel programming model

C

mpicc first-example.c
and **mpirun -np 4 ./a.out**

Fortran

mpif90 first-example_30.f90

Python

mpirun -np 4 python3 first-example.py (no compilation, parallel start of the interpreter)

- (*or equivalent commands on your system*)
- As input, you may choose: 100 (→ output should be similar to previous slide)
CAUTION: The previous printout “Enter the number” may be missing. Do not wait! Just type 100 [RETURN] - it should work!
- Look at the sequence of the output lines for several runs with 4 to 10 processes

That’s all, done.

Exercise 1 – Solution + Questions

```
mpif90 first-example_30.f90
```

```
mpirun -np 6 ./a.out
```

```
Enter the number of elements (n):
```

```
100
```

```
I am process 0 out of 6 handling the 0th part of n= 100 elements, result= 0.00
```

```
I am process 1 out of 6 handling the 1th part of n= 100 elements, result= 100.00
```

```
I am process 2 out of 6 handling the 2th part of n= 100 elements, result= 200.00
```

```
I am process 3 out of 6 handling the 3th part of n= 100 elements, result= 300.00
```

```
I am process 4 out of 6 handling the 4th part of n= 100 elements, result= 400.00
```

```
I am process 5 out of 6 handling the 5th part of n= 100 elements, result= 500.00
```

```
I'm proc 0: My own result is 0.00
```

```
I'm proc 0: received result of process 1 is 100.00
```

```
I'm proc 0: received result of process 2 is 200.00
```

```
I'm proc 0: received result of process 3 is 300.00
```

```
I'm proc 0: received result of process 4 is 400.00
```

```
I'm proc 0: received result of process 5 is 500.00
```

Exercise 1 – Solution + Questions

```
mpif90 first-example_30.f90
```

```
mpirun -np 6 ./a.out
```

```
Enter the number of elements (n):
```

```
100
```

```
I am process 0 out of 6 handling the 0th part of n= 100 elements, result= 0.00
```

```
I am process 1 out of 6 handling the 1th part of n= 100 elements, result= 100.00
```

```
I am process 2 out of 6 handling the 2th part of n= 100 elements, result= 200.00
```

```
I am process 3 out of 6 handling the 3th part of n= 100 elements, result= 300.00
```

```
I am process 4 out of 6 handling the 4th part of n= 100 elements, result= 400.00
```

```
I am process 5 out of 6 handling the 5th part of n= 100 elements, result= 500.00
```

```
I'm proc 0: My own result is 0.00
```

```
I'm proc 0: received result of process 1 is 100.00
```

```
I'm proc 0: received result of process 2 is 200.00
```

```
I'm proc 0: received result of process 3 is 300.00
```

```
I'm proc 0: received result of process 4 is 400.00
```

```
I'm proc 0: received result of process 5 is 500.00
```

Normally, you'll never see this perfect output !

Exercise 1 – Solution + Questions

```
mpif90 first-example_30.f90
```

```
mpirun -np 6 ./a.out
```

```
Enter the number of elements (n):
```

```
100
```

```
I am process 0 out of 6 handling the 0th part of n= 100 elements, result= 0.00
```

```
I am process 1 out of 6 handling the 1th part of n= 100 elements, result= 100.00
```

```
I am process 2 out of 6 handling the 2th part of n= 100 elements, result= 200.00
```

```
I am process 3 out of 6 handling the 3th part of n= 100 elements, result= 300.00
```

```
I am process 4 out of 6 handling the 4th part of n= 100 elements, result= 400.00
```

```
I am process 5 out of 6 handling the 5th part of n= 100 elements, result= 500.00
```

```
I'm proc 0: My own result is 0.00
```

```
I'm proc 0: received result of process 1 is 100.00
```

```
I'm proc 0: received result of process 2 is 200.00
```

```
I'm proc 0: received result of process 3 is 300.00
```

```
I'm proc 0: received result of process 4 is 400.00
```

```
I'm proc 0: received result of process 5 is 500.00
```

Normally, you'll never see this perfect output !



Why ?

Exercise 1 – Solution + Answers

mpif90 first-example_30.f90

mpirun -np 6 ./a.out

Enter the number of elements (n):

100

I am process 4 out of 6 handling the 4th part of n= 100 elements, result= 400.00

I am process 0 out of 6 handling the 0th part of n= 100 elements, result= 0.00

I'm proc 0: My own result is 0.00

I am process 2 out of 6 handling the 2th part of n= 100 elements, result= 200.00

I am process 1 out of 6 handling the 1th part of n= 100 elements, result= 100.00

I'm proc 0: received result of process 1 is 100.00

I'm proc 0: received result of process 2 is 200.00

I'm proc 0: received result of process 3 is 300.00

I am process 3 out of 6 handling the 3th part of n= 100 elements, result= 300.00

I'm proc 0: received result of process 4 is 400.00

I am process 5 out of 6 handling the 5th part of n= 100 elements, result= 500.00

I'm proc 0: received result of process 5 is 500.00

Exercise 1 – Solution + Answers

```
mpif90 first-example_30.f90
```

```
mpirun -np 6 ./a.out
```

```
Enter the number of elements (n):
```

```
100
```

```
I am process 4 out of 6 handling the 4th part of n= 100 elements, result= 400.00
```

```
I am process 0 out of 6 handling the 0th part of n= 100 elements, result= 0.00
```

```
I'm proc 0: My own result is 0.00
```

```
I am process 2 out of 6 handling the 2th part of n= 100 elements, result= 200.00
```

```
I am process 1 out of 6 handling the 1th part of n= 100 elements, result= 100.00
```

```
I'm proc 0: received result of process 1 is 100.00
```

```
I'm proc 0: received result of process 2 is 200.00
```

```
I'm proc 0: received result of process 3 is 300.00
```

```
I am process 3 out of 6 handling the 3th part of n= 100 elements, result= 300.00
```

```
I'm proc 0: received result of process 4 is 400.00
```

```
I am process 5 out of 6 handling the 5th part of n= 100 elements, result= 500.00
```

```
I'm proc 0: received result of process 5 is 500.00
```

General rule:

The output of each process is in the well defined sequence of its sub-program, see, e.g., the **bold text** from process 0!

Exercise 1 – Solution + Answers

```
mpif90 first-example_30.f90
```

```
mpirun -np 6 ./a.out
```

```
Enter the number of elements (n):
```

```
100
```

```
I am process 4 out of 6 handling the 4th part of n= 100 elements, result= 400.00
```

```
I am process 0 out of 6 handling the 0th part of n= 100 elements, result= 0.00
```

```
I'm proc 0: My own result is 0.00
```

```
I am process 2 out of 6 handling the 2th part of n= 100 elements, result= 200.00
```

```
I am process 1 out of 6 handling the 1th part of n= 100 elements, result= 100.00
```

```
I'm proc 0: received result of process 1 is 100.00
```

```
I'm proc 0: received result of process 2 is 200.00
```

```
I'm proc 0: received result of process 3 is 300.00
```

```
I am process 3 out of 6 handling the 3th part of n= 100 elements, result= 300.00
```

```
I'm proc 0: received result of process 4 is 400.00
```

```
I am process 5 out of 6 handling the 5th part of n= 100 elements, result= 500.00
```

```
I'm proc 0: received result of process 5 is 500.00
```

General rule:

The output of each process is in the well defined sequence of its sub-program, see, e.g., the **bold text** from process 0!

The output from different processes can be intermixed in any sequence!

Most MPI libraries try to not intersect output lines 😊

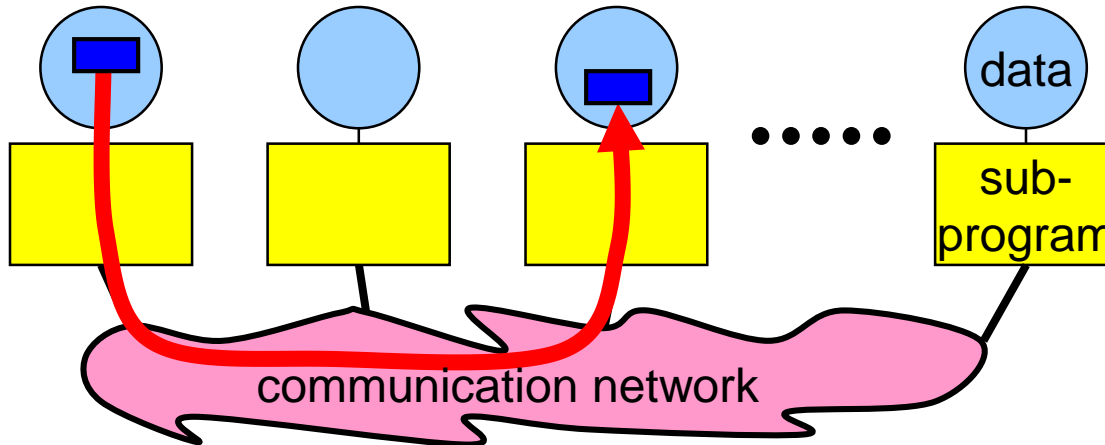
Access


- A sub-program needs to be connected to a message passing system
- A message passing system is similar to:
 - mail box
 - phone line
 - fax machine
 - etc.
- MPI:
 - sub-program must be linked with an MPI library
 - sub-program must use include file of this MPI library
 - the total program (i.e., all sub-programs of the program) must be started with the MPI startup tool

Access

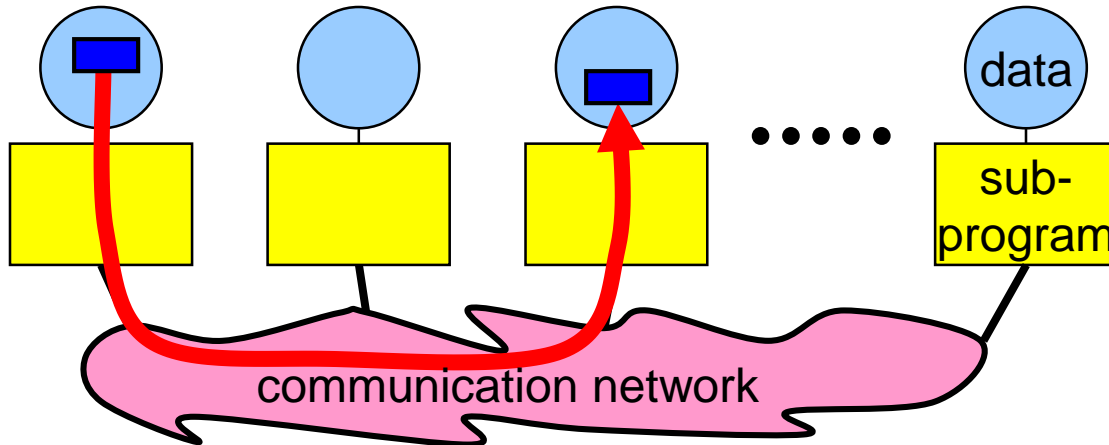
- A sub-program needs to be connected to a message passing system
- A message passing system is similar to:
 - mail box
 - phone line
 - fax machine
 - etc.
- MPI:
 - sub-program must be linked with an MPI library
 - sub-program must use include file of this MPI library
 - the total program (i.e., all sub-programs of the program) must be started with the MPI startup tool


Messages



- Messages are packets of data moving between sub-programs
 - Necessary information for the message passing system:
 - sending process
 - source location
 - source data type
 - source data size
 - receiving process
 - destination location
 - destination data type
 - destination buffer size
- } i.e., the ranks
- } 

Messages



- Messages are packets of data moving between sub-programs
 - Necessary information for the message passing system:
 - sending process
 - source location
 - source data type
 - source data size
 - receiving process
 - destination location
 - destination data type
 - destination buffer size
- } i.e., the ranks
- } 
- basic or **derived** datatypes

Addressing

- Messages need to have addresses to be sent to.
- Addresses are similar to:
 - mail addresses
 - phone number
 - fax number
 - etc.
- MPI: addresses are ranks of the MPI processes (sub-programs)

Receiving

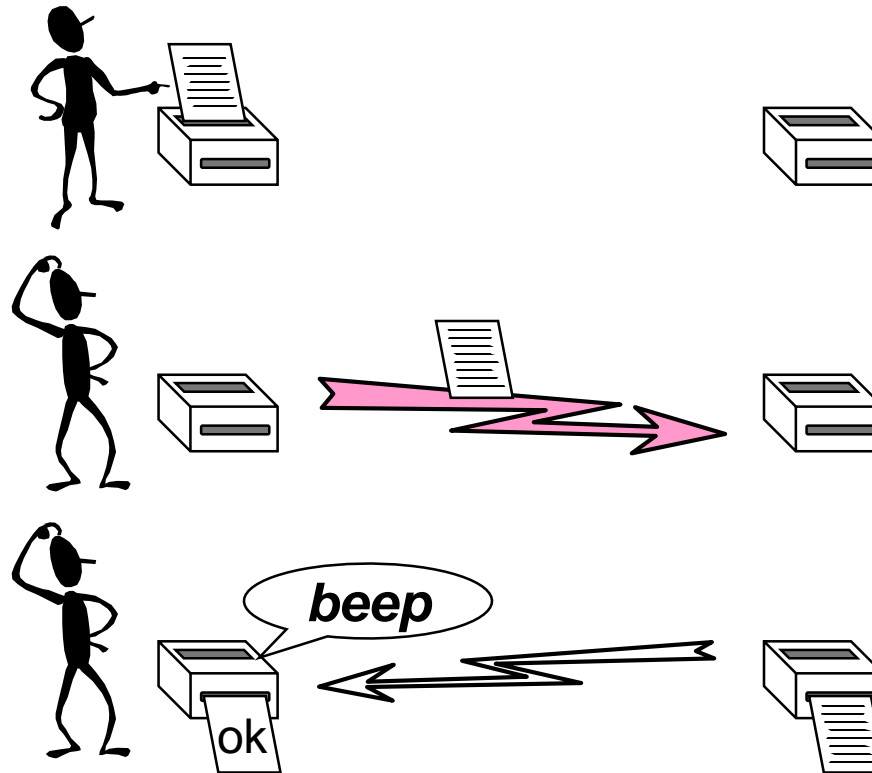
- All messages must be received.

Point-to-Point Communication

- Simplest form of message passing.
- One process sends a message to another.
- Different types of point-to-point communication:
 - synchronous send
 - buffered = asynchronous send

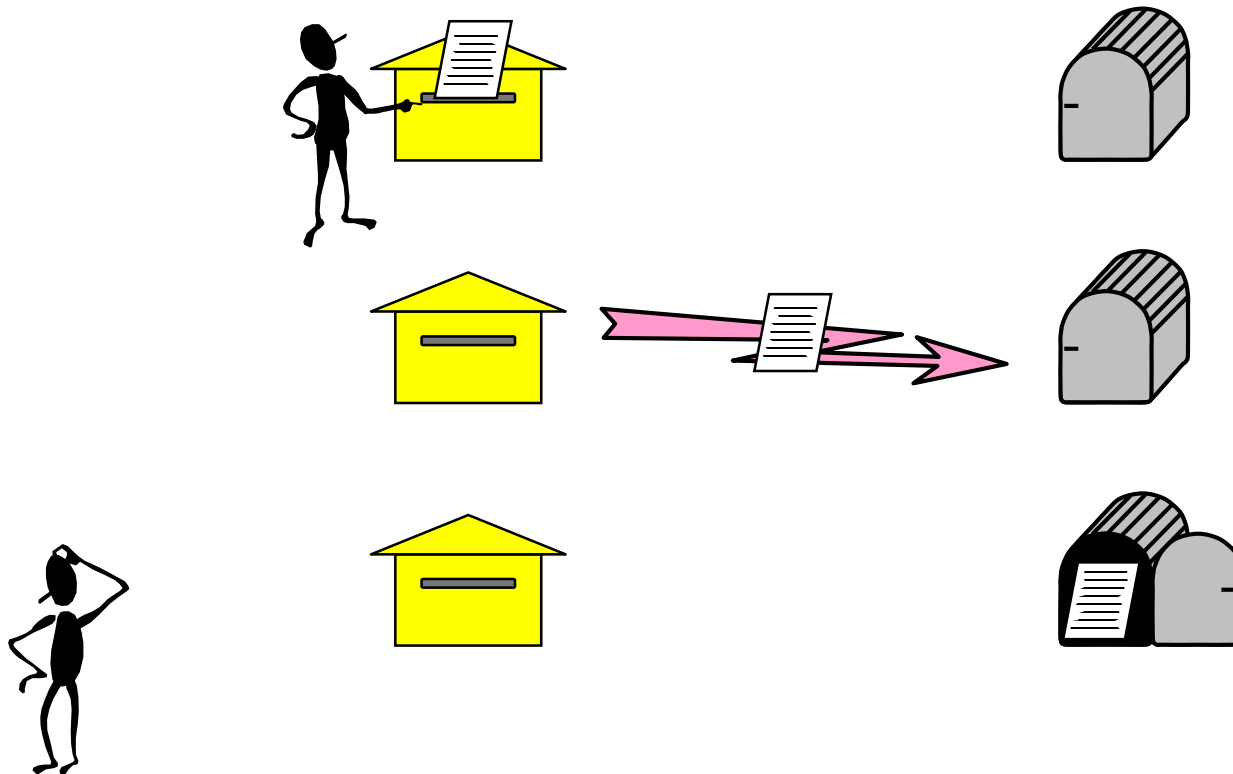
Synchronous Sends

- The sender gets an information that the message is received.
- Analogue to the *beep* or *okay-sheet* of a fax.



Buffered = Asynchronous Sends

- Only know when the message has left.



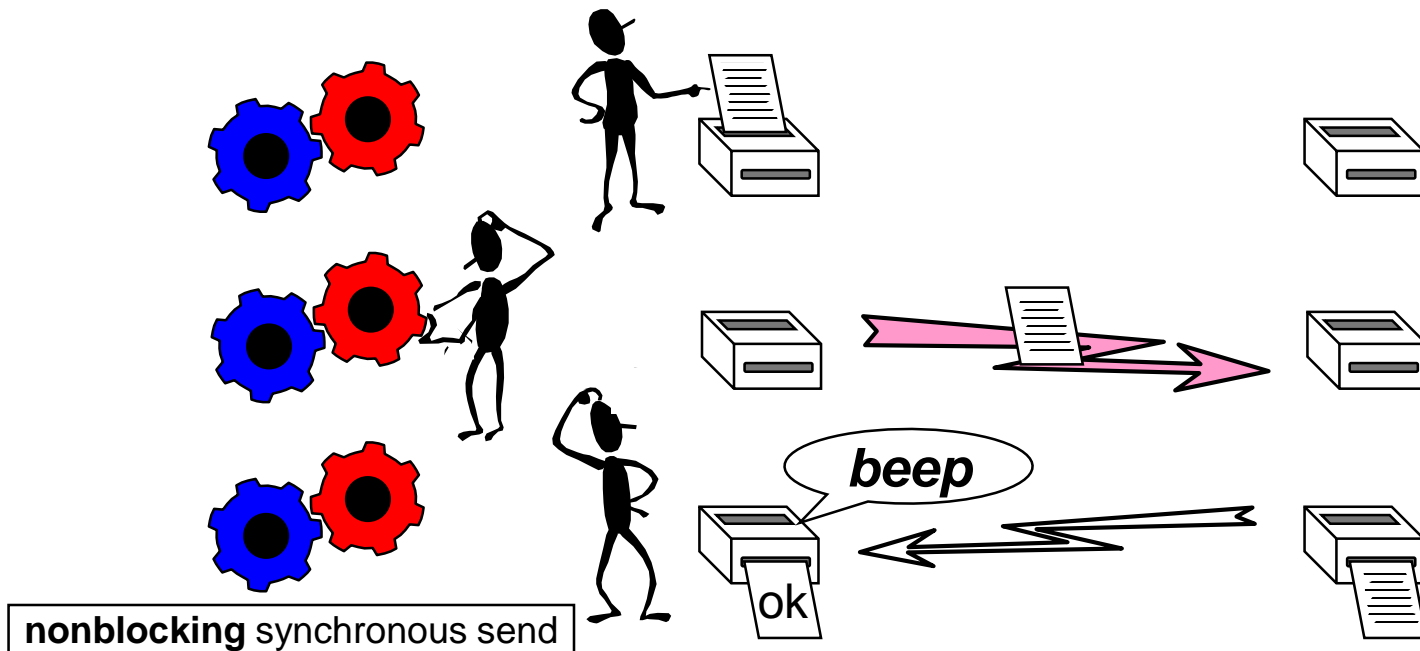
Blocking Operations

- Operations are activities, such as
 - sending (a message)
 - receiving (a message)
- Some operations may **block** until another process acts:
 - synchronous send operation **blocks until** receive is posted;
 - receive operation **blocks until** message was sent.
- Relates to the completion of an operation.
- Blocking subroutine returns only when the operation has completed.

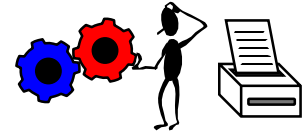
Nonblocking Operations

Nonblocking operations consist of:

- A nonblocking procedure call: it returns immediately and allows the sub-program to perform other work
- At some later time the sub-program must *test* or *wait* for the completion of the nonblocking operation



Non-Blocking Operations (cont'd)



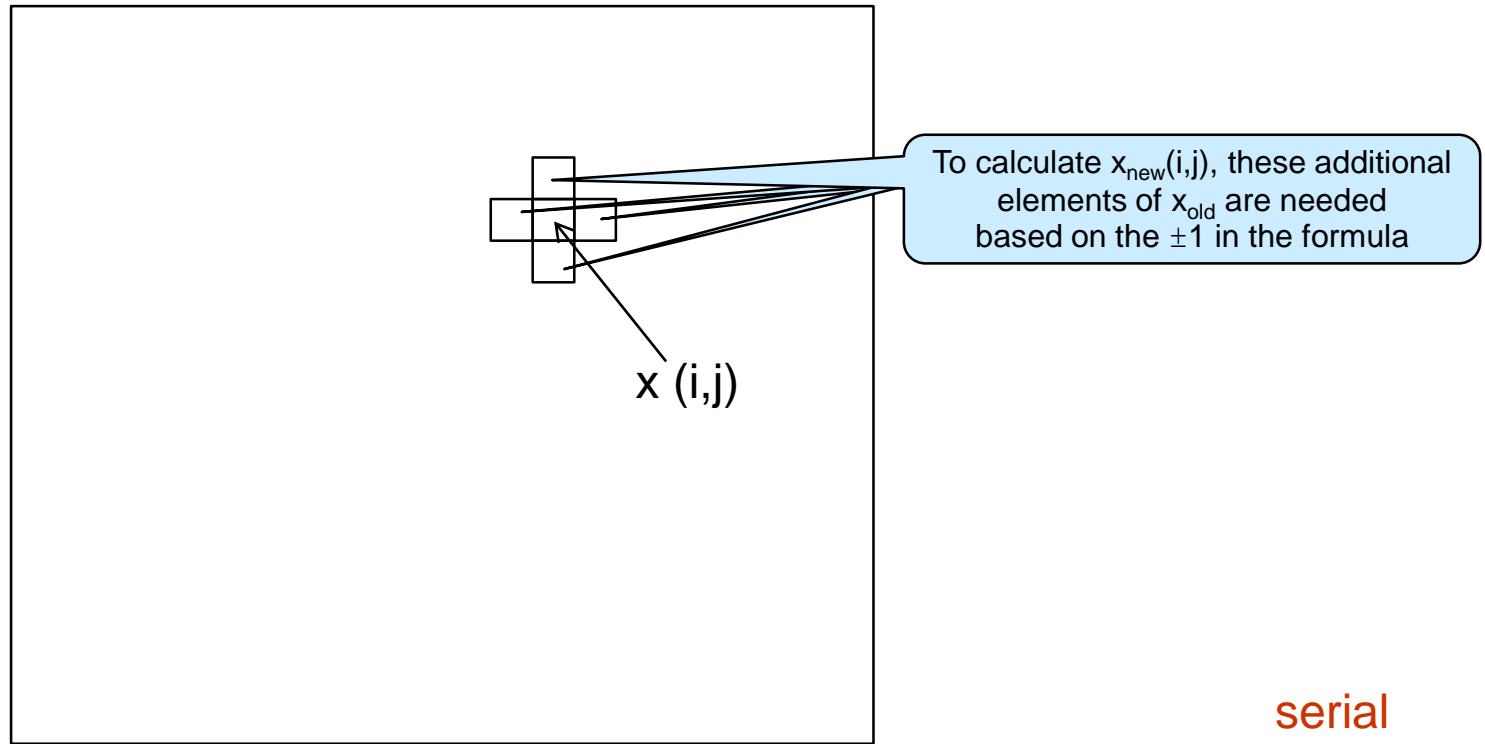
- All nonblocking procedures must have a matching wait (or test) procedure. (Some system or application resources can be freed only when the nonblocking operation is completed.)
- A nonblocking procedure immediately followed by a matching wait is equivalent to a blocking procedure.
- Nonblocking procedures are not the same as sequential subroutine calls:
 - the operation may continue while the application executes the next statements!

Interrupt: Example & Exercise 2

- Before we further go through the MPI chapter overview on
 - Collective Communication
 - Parallel file I/O
- Lets look at halo communication
- plus a short exercise 2

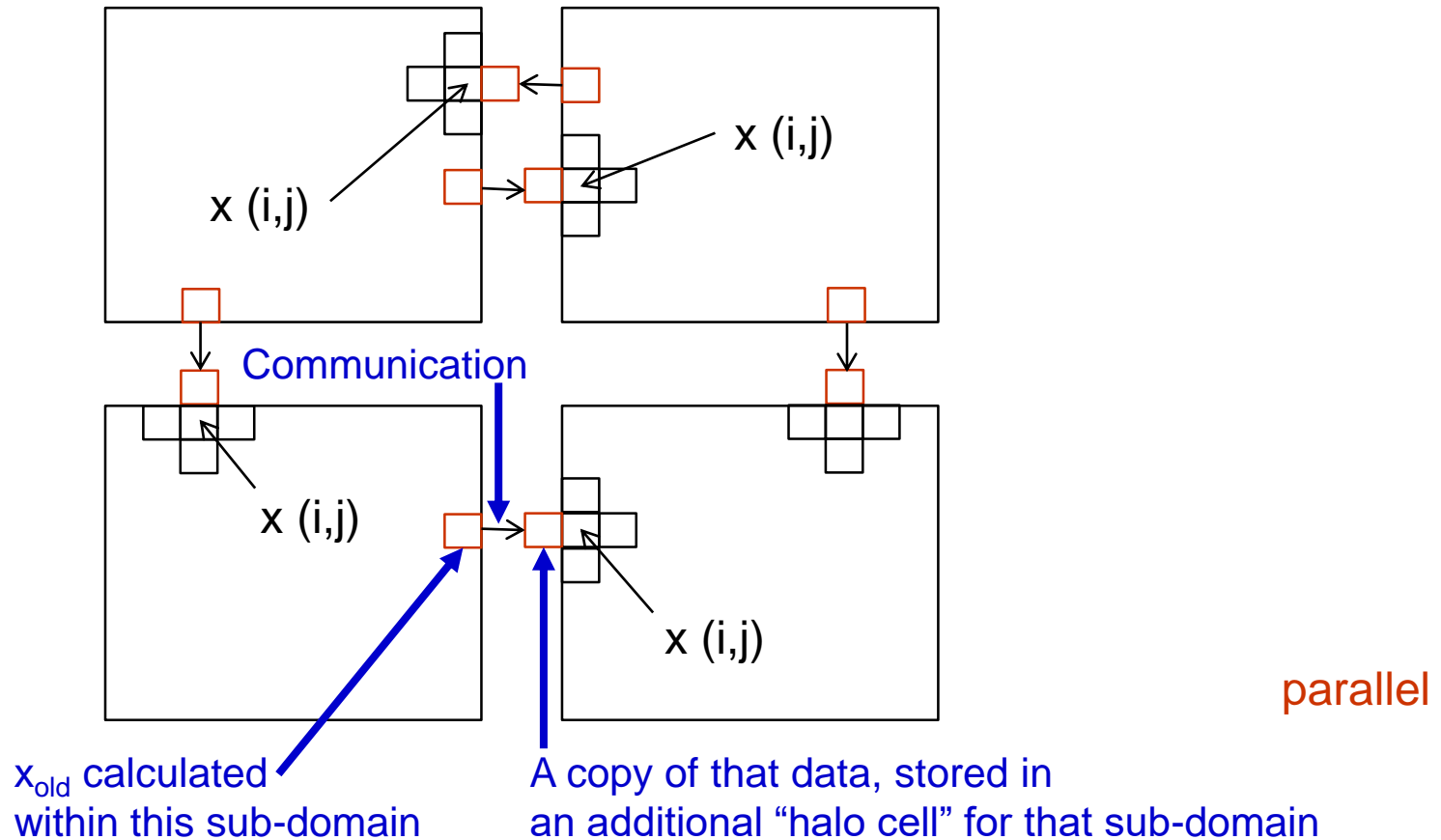
Example: Domain decomposition – serial

- $x_{\text{new}}(i,j) = f(x_{\text{old}}(i-1,j), x_{\text{old}}(i,j), x_{\text{old}}(i+1,j), x_{\text{old}}(i,j-1), x_{\text{old}}(i,j+1))$



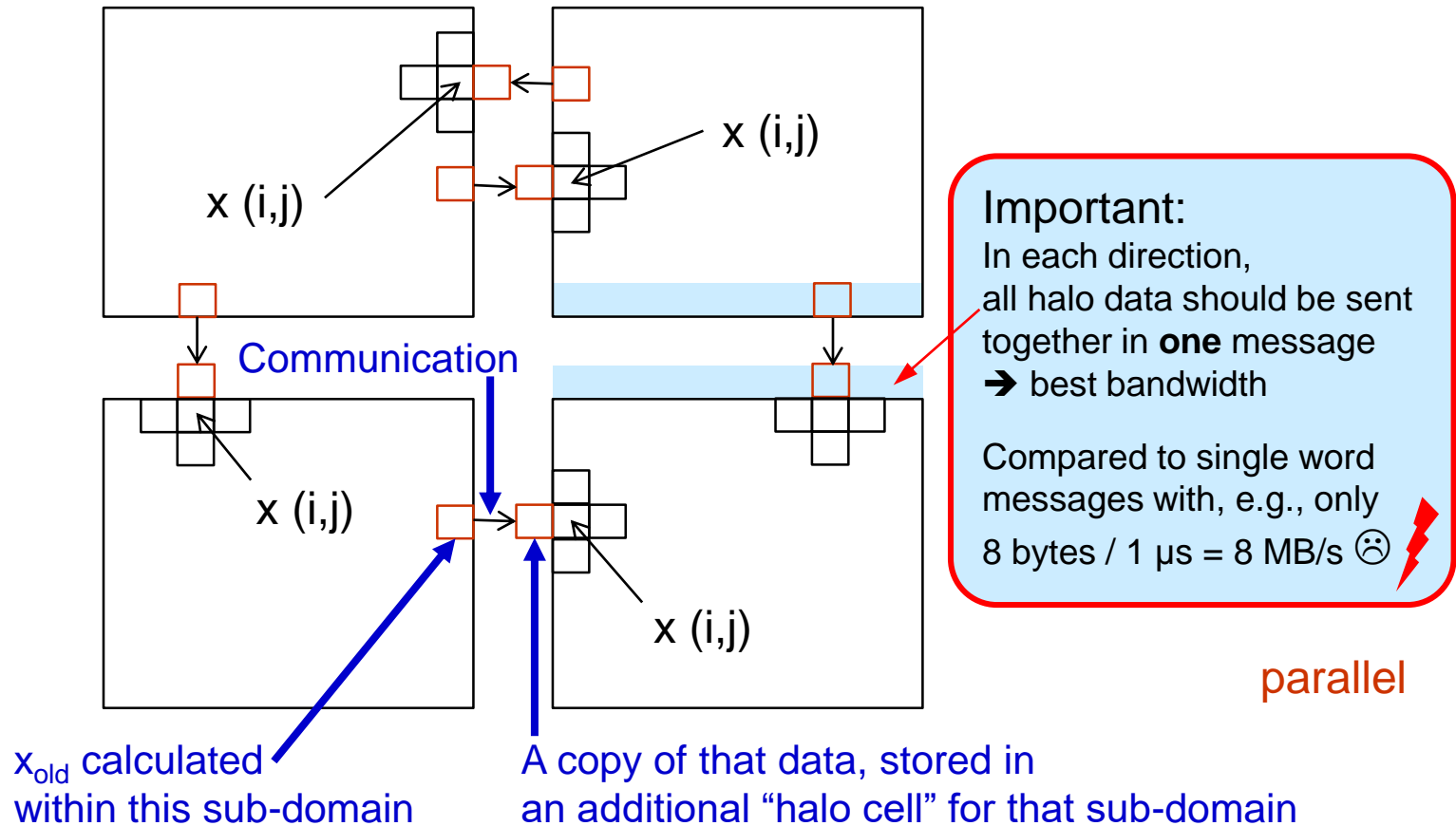
Example: Domain decomposition – parallel

- $x_{\text{new}}(i,j) = f(x_{\text{old}}(i-1,j), x_{\text{old}}(i,j), x_{\text{old}}(i+1,j), x_{\text{old}}(i,j-1), x_{\text{old}}(i,j+1))$



Example: Domain decomposition – parallel

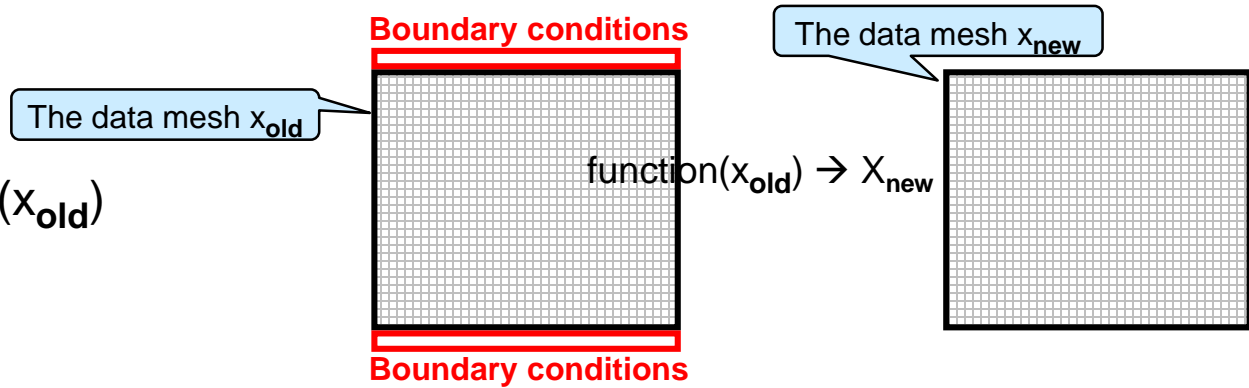
- $x_{\text{new}}(i,j) = f(x_{\text{old}}(i-1,j), x_{\text{old}}(i,j), x_{\text{old}}(i+1,j), x_{\text{old}}(i,j-1), x_{\text{old}}(i,j+1))$



Communication: Send inner data into halo storage

One iteration in the
serial code:

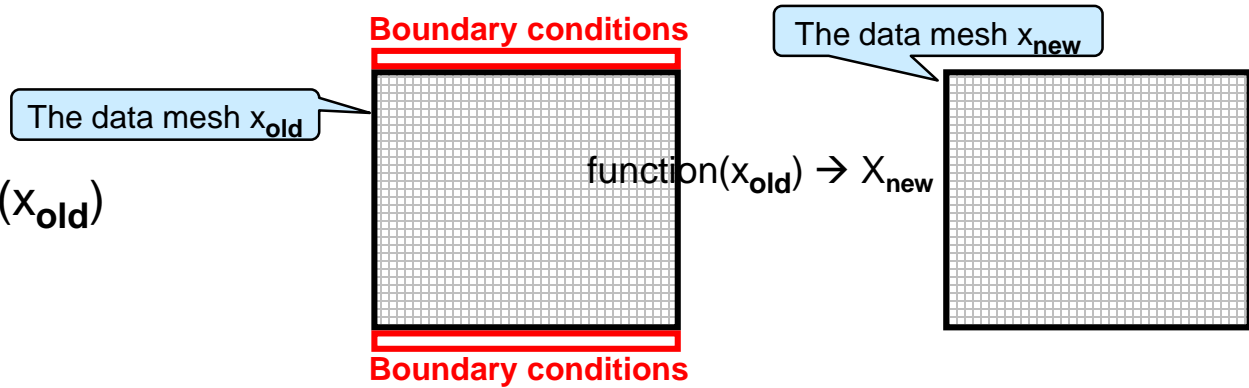
- $x_{\text{new}} = \text{function}(x_{\text{old}})$
- $x_{\text{old}} = x_{\text{new}}$



Communication: Send inner data into halo storage

One iteration in the
serial code:

- $x_{\text{new}} = \text{function}(x_{\text{old}})$
- $x_{\text{old}} = x_{\text{new}}$



parallel code:

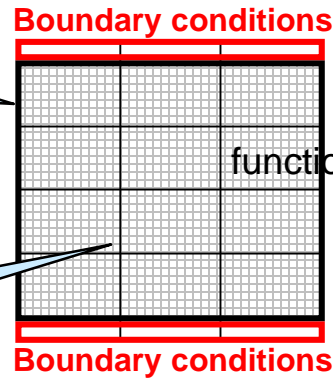
Communication: Send inner data into halo storage

One iteration in the
serial code:

- $X_{\text{new}} = \text{function}(x_{\text{old}})$
- $X_{\text{old}} = X_{\text{new}}$

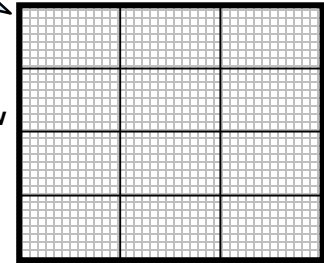
Preparing the domain decomposition
of the data mesh into
3x4 subdomains for 12 processes

The data mesh x_{old}



function(x_{old}) \rightarrow X_{new}

The data mesh x_{new}



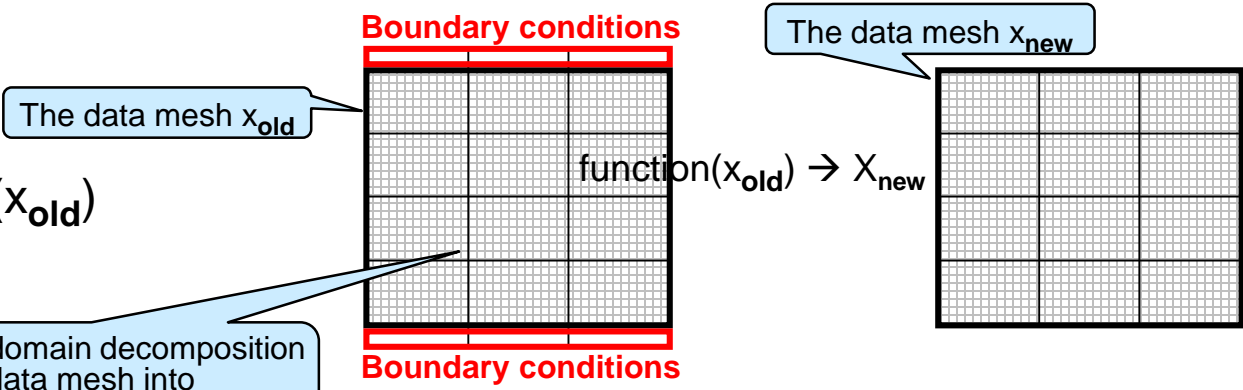
parallel code:

Communication: Send inner data into halo storage

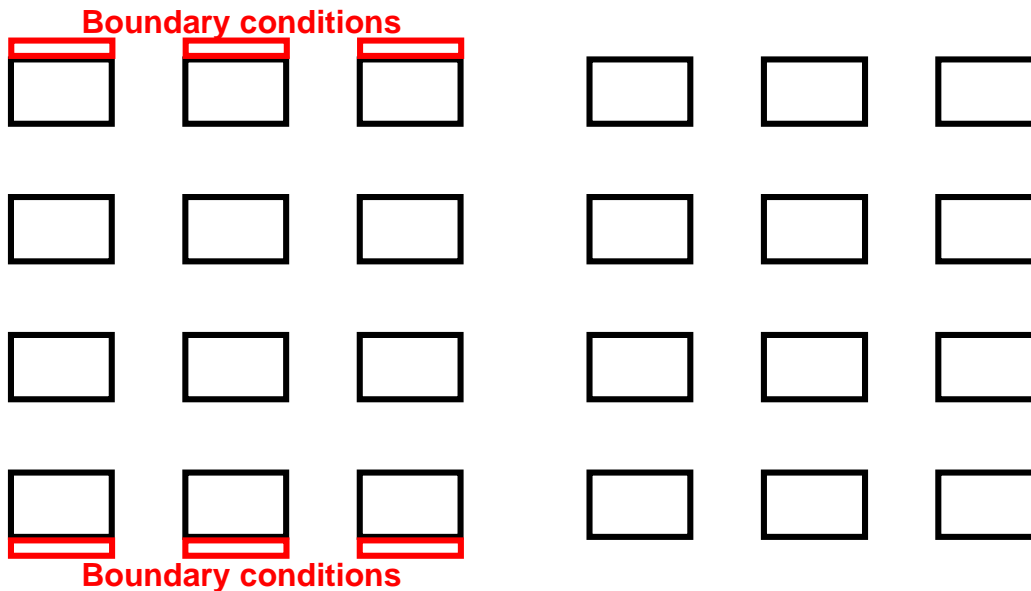
One iteration in the **serial** code:

- $X_{new} = \text{function}(x_{old})$
- $X_{old} = X_{new}$

Preparing the domain decomposition of the data mesh into 3x4 subdomains for 12 processes



parallel code:



x_{old} & **boundary conditions** distributed on 3x4=12 MPI processes

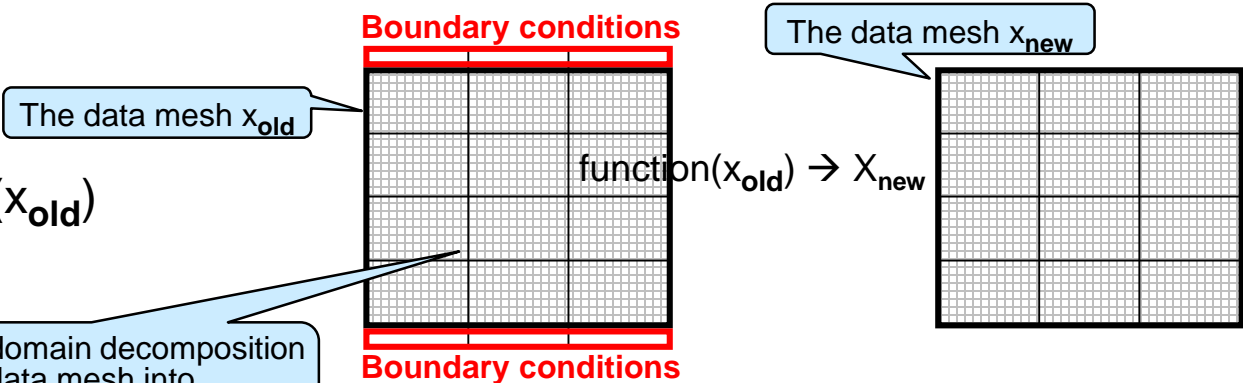
x_{new} , distributed on same MPI processes

Communication: Send inner data into halo storage


One iteration in the **serial** code:

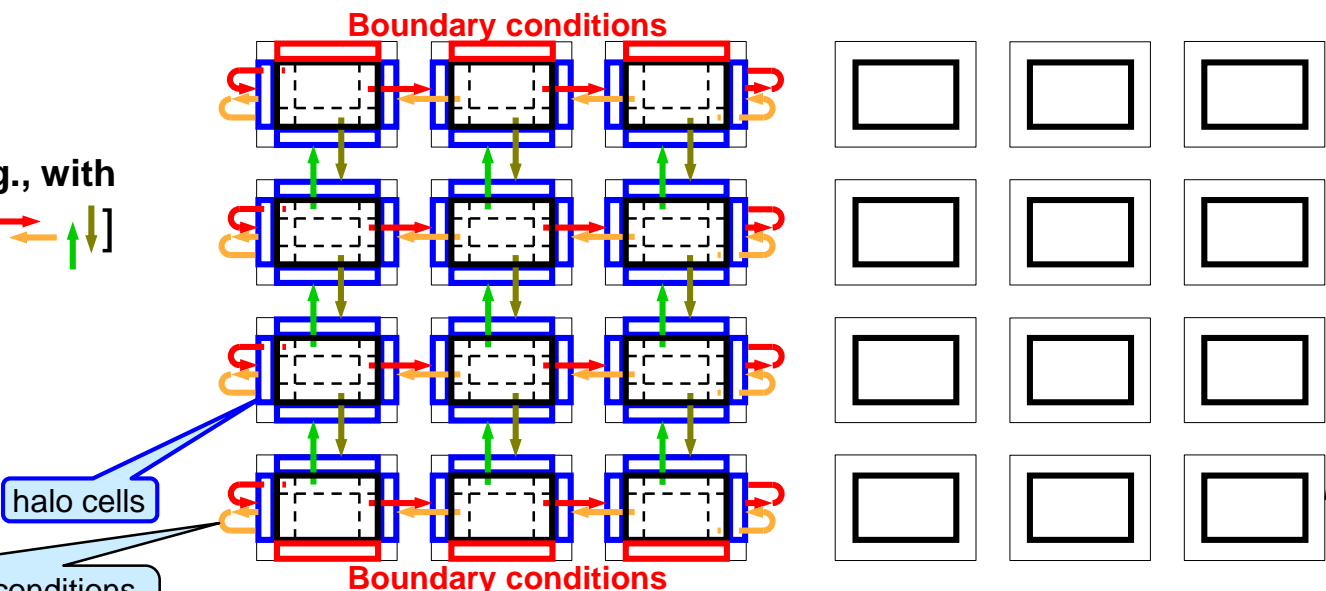
- $X_{new} = \text{function}(x_{old})$
- $X_{old} = X_{new}$

Preparing the domain decomposition of the data mesh into 3x4 subdomains for 12 processes



parallel code:

- Update halo
[=Communication, e.g., with
4 x MPI_Sendrecv 



horizontally cyclic boundary conditions
→ communication around rings

x_{old} & boundary conditions distributed on 3x4=12 MPI processes

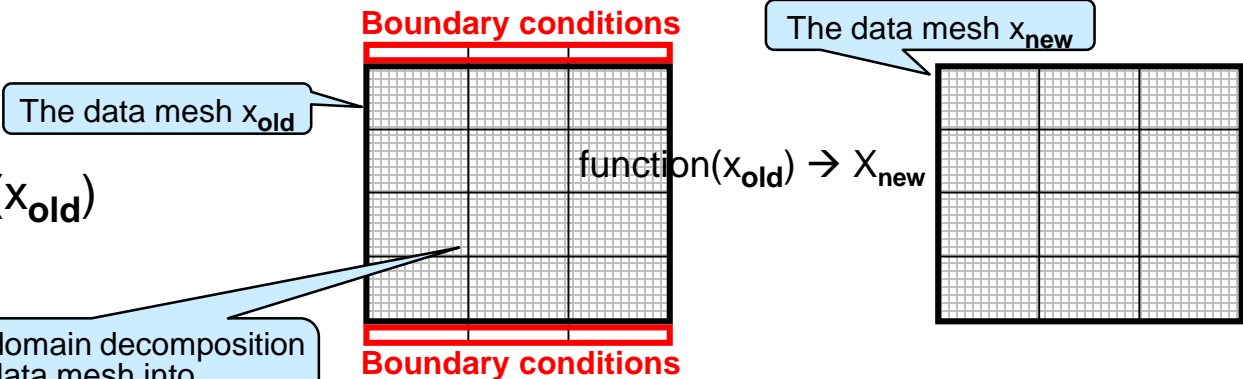
x_{new} , distributed on same MPI processes

Maybe same memory layout for x_{old} and x_{new} to allow pointer exchange instead of data copy $x_{old} = x_{new}$

Communication: Send inner data into halo storage


One iteration in the **serial** code:

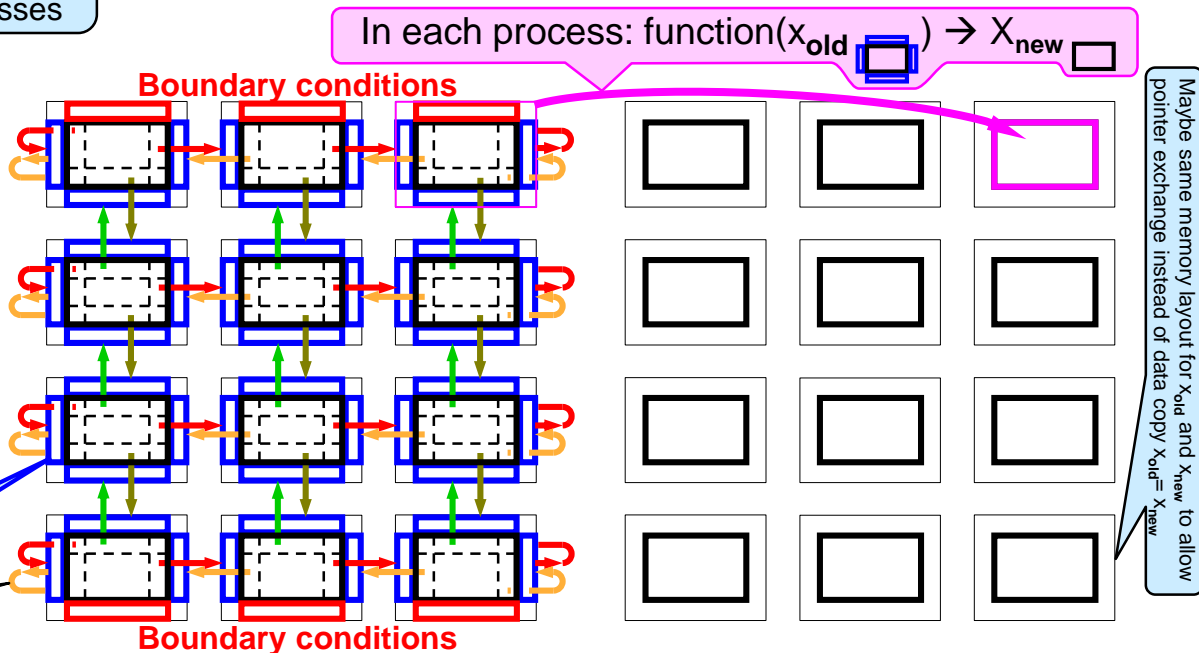
- $X_{new} = \text{function}(x_{old})$
- $X_{old} = X_{new}$



Preparing the domain decomposition of the data mesh into 3x4 subdomains for 12 processes

parallel code:

- Update halo
[=Communication, e.g., with 4 x MPI_Sendrecv 
- $X_{new} \square = \text{function}(x_{old} \square)$



halo cells

horizontally cyclic boundary conditions
→ communication around rings

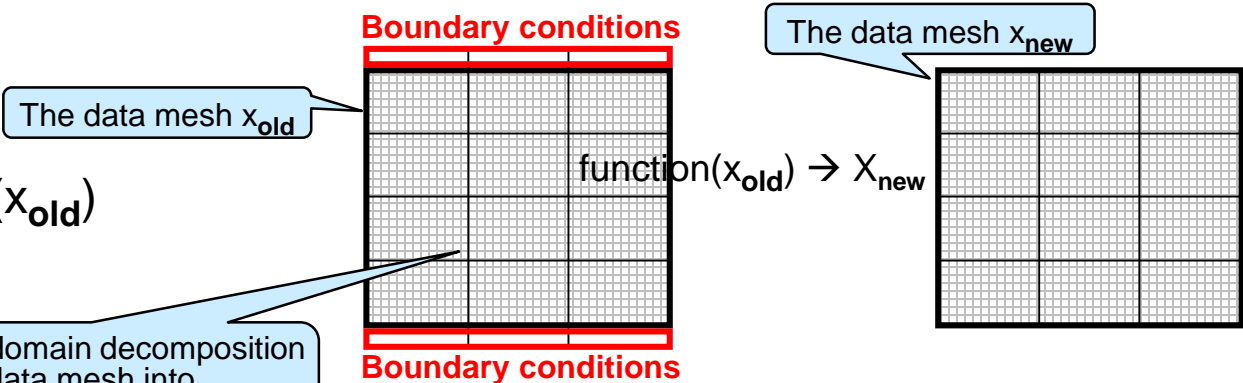
x_{old} & boundary conditions distributed on 3x4=12 MPI processes

x_{new} , distributed on same MPI processes

Communication: Send inner data into halo storage


One iteration in the **serial** code:

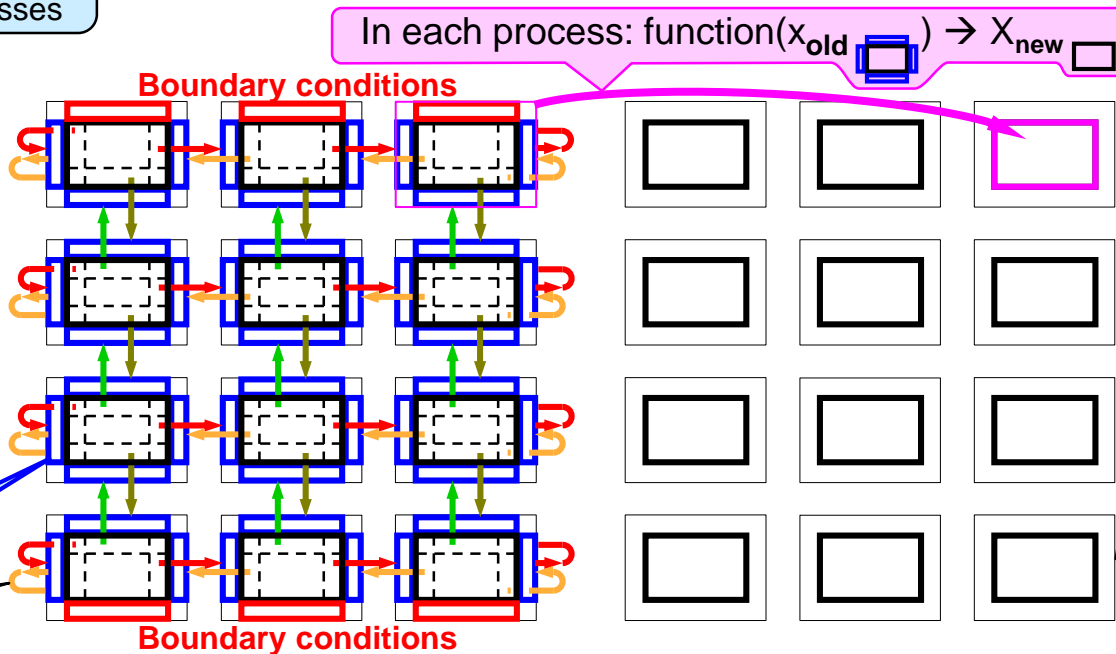
- $X_{new} = \text{function}(x_{old})$
- $X_{old} = X_{new}$



Preparing the domain decomposition of the data mesh into 3x4 subdomains for 12 processes

parallel code:

- Update halo
[=Communication, e.g., with 4 x MPI_Sendrecv 
- $X_{new} \square = \text{function}(x_{old} \square)$
- $X_{old} \square = X_{new} \square$



halo cells

horizontally cyclic boundary conditions
→ communication around rings

x_{old} & boundary conditions distributed on 3x4=12 MPI processes

x_{new} , distributed on same MPI processes

Maybe same memory layout for x_{old} and x_{new} to allow pointer exchange instead of data copy $x_{old} = x_{new}$

Example code

```
ib_global = 0; ie_global=n-1; // global xold, xnew: arrays with n elements and indexes 0 .. n-1
```

```
for(...) / e.g. timesteps  
{
```

```
    numerical_func( xold, xnew, ib_global, ie_global);  
    tmp=xold; xold=xnew; xnew=tmp; // exchanging role of xold and xnew  
}
```

□

Example code

```

ib_global = 0; ie_global=n-1; // global xold, xnew: arrays with n elements and indexes 0 .. n-1

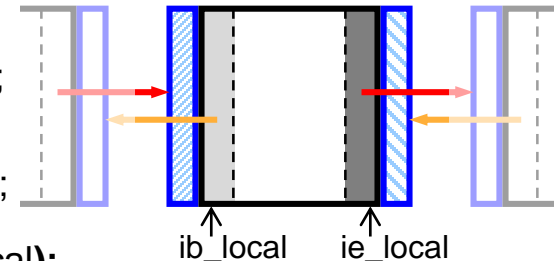
MPI_Init(null, null);
MPI_Comm_size(MPI_COMM_WORLD, &size); // size = number of processes
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); // myrank = index of the process from 0 to size-1

ib_local = ib_global + myrank *  $\overbrace{((ie\_global-ib\_global)/size + 1)}^{[n/size]}$  );
ie_local = min(ib_global + (myrank+1) * ((ie_global-ib_global)/size + 1), ie_global);

left = (myrank-1 To prohibit modulo of negative number + size) % size; right = (myrank+1) % size; // neighbor ranks
for(...) // e.g. timesteps
{ MPI_Sendrecv( /*sndbuf*/ xold, 1, right_black_part, right, 111,
               /*rcvbuf*/ xold, 1, left_blue_halo, left, 111, ...);
  MPI_Sendrecv( /*sndbuf*/ xold, 1, left_black_part, left, 222,
               /*rcvbuf*/ xold, 1, right_blue_halo, right, 222, ...);

  numerical_func( xold, xnew, ib_global ib_local, ie_global ie_local);
  tmp=xold; xold=xnew; xnew=tmp; // exchanging role of xold and xnew
}

```



□

Exercise 2: Calculating the size of the subdomains

Goal: Divide a given amount of mesh elements in one dimension into subdomains

- Given:
 - The number of processes: `num_procs` (e.g., **4**, i.e., 4 subdomains)
 - The number of mesh elements: `n` (e.g., **17 or 5**)
 - The numerical workload of each element is identical
 - The mesh elements are numbered from **0** to **n-1**

Exercise 2: Calculating the size of the subdomains

Goal: Divide a given amount of mesh elements in one dimension into subdomains

- Given:
 - The number of processes: num_procs (e.g., **4**, i.e., 4 subdomains)
 - The number of mesh elements: n (e.g., **17 or 5**)
 - The numerical workload of each element is identical
 - The mesh elements are numbered from **0** to **n-1**

#mesh elements
per subdomain

- **Two possible solutions:**
 - (A) $17=5+5+5+2$ or $5=2+2+1+0$
 - (B) $17=5+4+4+4$ or $5=2+1+1+1$

Exercise 2: Calculating the size of the subdomains

Goal: Divide a given amount of mesh elements in one dimension into subdomains

- Given:
 - The number of processes: num_procs (e.g., 4, i.e., 4 subdomains)
 - The number of mesh elements: n (e.g., 17 or 5)
 - The numerical workload of each element is identical
 - The mesh elements are numbered from 0 to n-1

#mesh elements
per subdomain

- Two possible solutions:
 - (A) $17=5+5+5+2$ or $5=2+2+1+0$
 - (B) $17=5+4+4+4$ or $5=2+1+1+1$

- Output should be like (with B)

Or -1 if 0 elements

I am process 1 out of 4, responsible for the 4 elements with indexes 5 .. 8
 I am process 0 out of 4, responsible for the 5 elements with indexes 0 .. 4
 I am process 3 out of 4, responsible for the 4 elements with indexes 13 .. 16
 I am process 2 out of 4, responsible for the 4 elements with indexes 9 .. 12

Exercise 2: Calculating the size of the subdomains

Goal: Divide a given amount of mesh elements in one dimension into subdomains

- Given:
 - The number of processes: num_procs (e.g., 4, i.e., 4 subdomains)
 - The number of mesh elements: n (e.g., 17 or 5)
 - The numerical workload of each element is identical
 - The mesh elements are numbered from 0 to n-1

#mesh elements
per subdomain

- Two possible solutions:
 - (A) $17=5+5+5+2$ or $5=2+2+1+0$
 - (B) $17=5+4+4+4$ or $5=2+1+1+1$

- Output should be like (with B)

Or -1 if 0 elements

I am process 1 out of 4, responsible for the 4 elements with indexes 5 .. 8

I am process 0 out of 4, responsible for the 5 elements with indexes 0 .. 4

I am process 3 out of 4, responsible for the 4 elements with indexes 13 .. 16

I am process 2 out of 4, responsible for the 4 elements with indexes 9 .. 12

In MPI/tasks/...

- Use:
 - C** C/Ch1/first-dd-a.c and C/Ch1/first-dd-b.c
 - or **Fortran** F_30/Ch1/first-dd-a_30.f90 and F_30/Ch1/first-dd-b_30.f90
 - or **Python** PY/Ch1/first-dd-a_30.py and PY/Ch1/first-dd-b_30.py
- Test both programs with 4 processes and 9, 8, 7, ... 1 elements

Exercise 2: Calculating the size of the subdomains

Goal: Divide a given amount of mesh elements in one dimension into subdomains

- Given:
 - The number of processes: num_procs (e.g., 4, i.e., 4 subdomains)
 - The number of mesh elements: n (e.g., 17 or 5)
 - The numerical workload of each element is identical
 - The mesh elements are numbered from 0 to n-1

#mesh elements
per subdomain

- Two possible solutions:
 - (A) $17=5+5+5+2$ or $5=2+2+1+0$
 - (B) $17=5+4+4+4$ or $5=2+1+1+1$

- Output should be like (with B)

Or -1 if 0 elements

I am process 1 out of 4, responsible for the 4 elements with indexes 5 .. 8

I am process 0 out of 4, responsible for the 5 elements with indexes 0 .. 4

I am process 3 out of 4, responsible for the 4 elements with indexes 13 .. 16

I am process 2 out of 4, responsible for the 4 elements with indexes 9 .. 12

In MPI/tasks/...

- Use:
 - C** C/Ch1/first-dd-a.c and C/Ch1/first-dd-b.c
 - or **Fortran** F_30/Ch1/first-dd-a_30.f90 and F_30/Ch1/first-dd-b_30.f90
 - or **Python** PY/Ch1/first-dd-a_30.py and PY/Ch1/first-dd-b_30.py
- Test both programs with 4 processes and 9, 8, 7, ... 1 elements

Which algorithm would you prefer, and why?

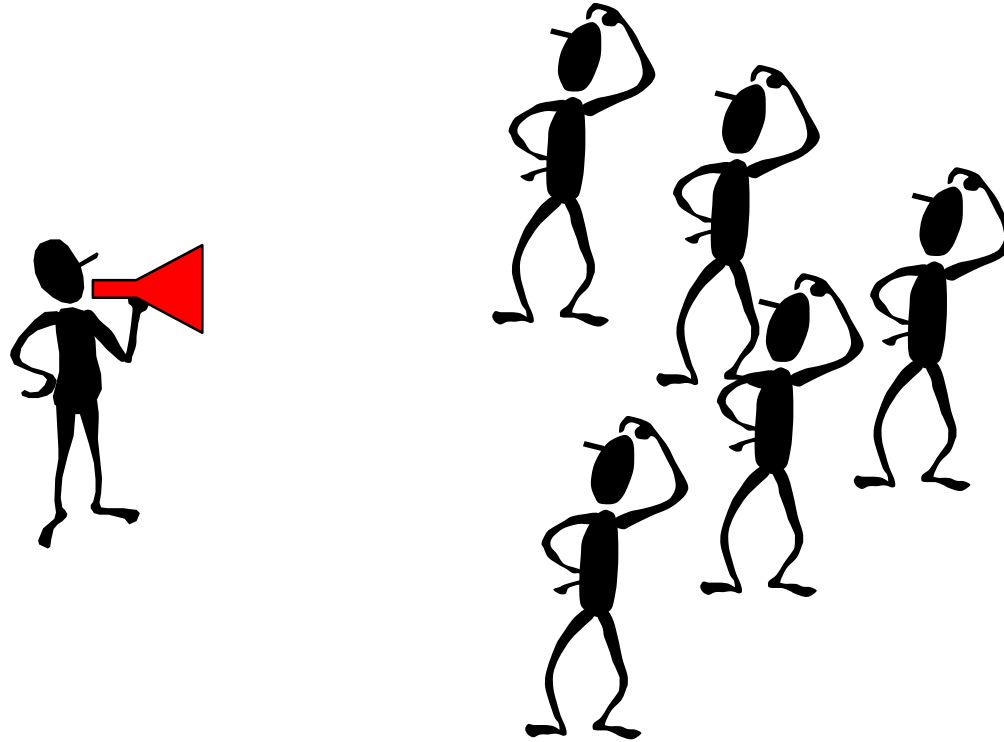
Which are the major principles of A and B?

Collective Communications

- Collective communication routines are higher level routines.
- Several processes are involved at a time.
- May allow optimized internal implementations, e.g., tree based algorithms.
- Can be built out of point-to-point communications.

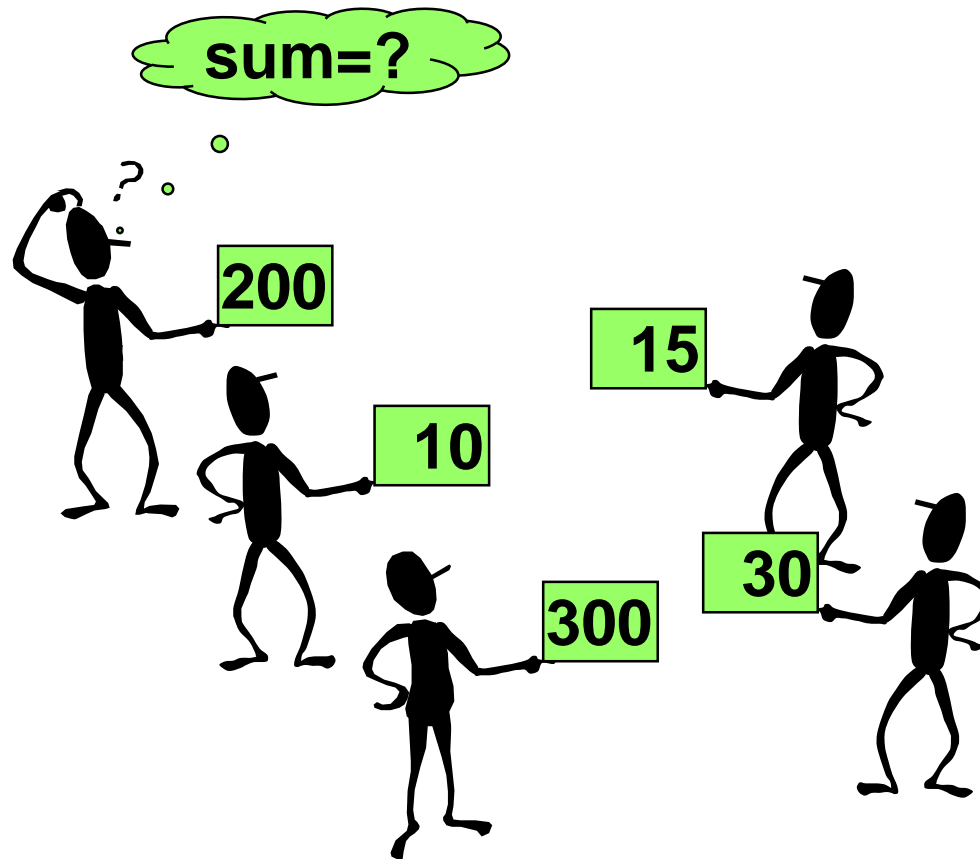
Broadcast

- A one-to-many communication.



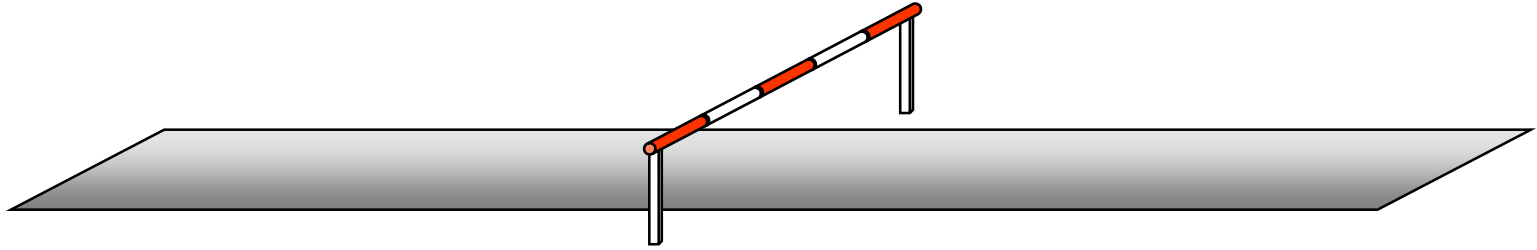
Reduction Operations

- Combine data from several processes to produce a single result.



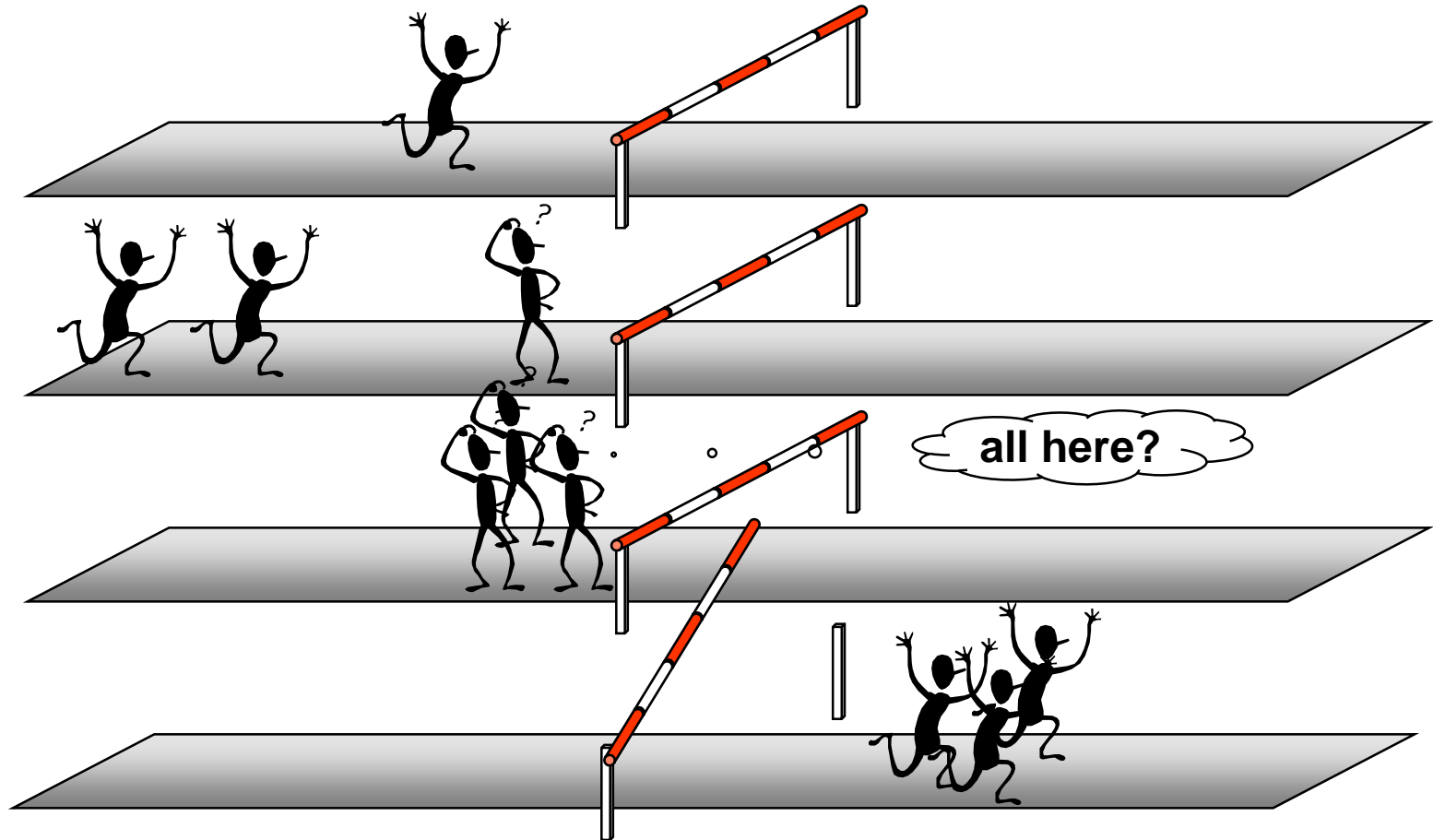
Barriers

- Synchronize processes.

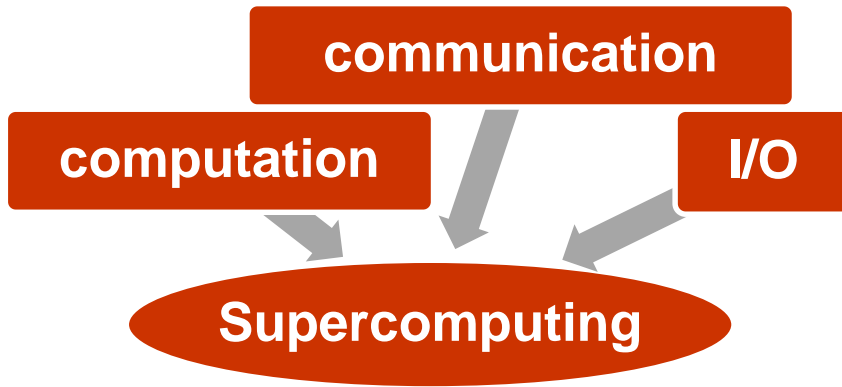


Barriers

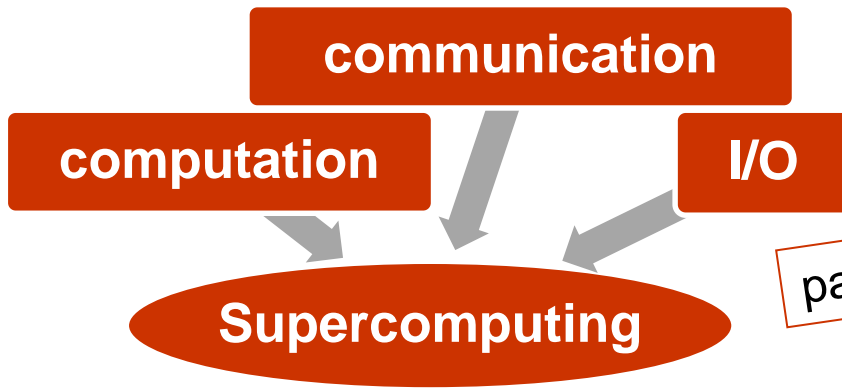
- Synchronize processes.



Parallel File I/O

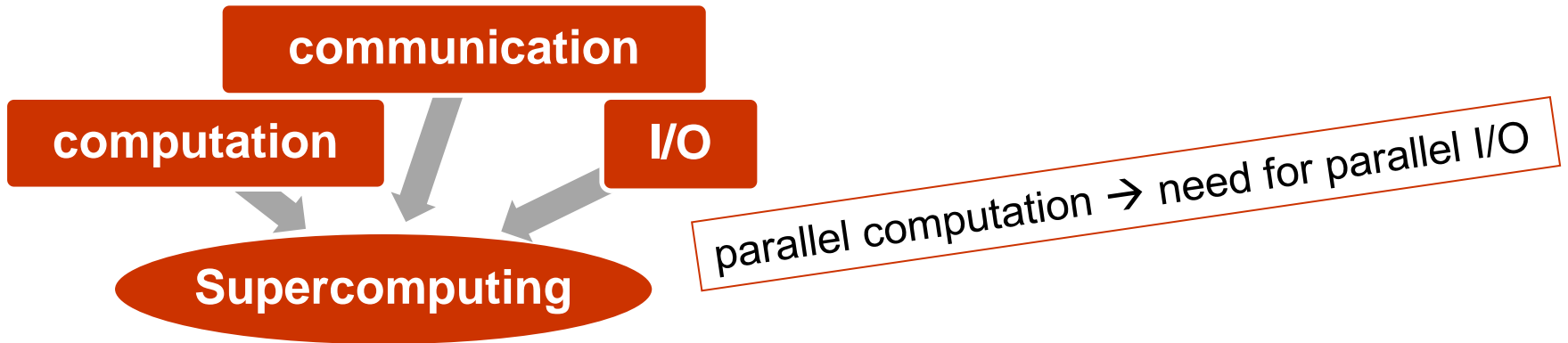


Parallel File I/O



parallel computation → need for parallel I/O

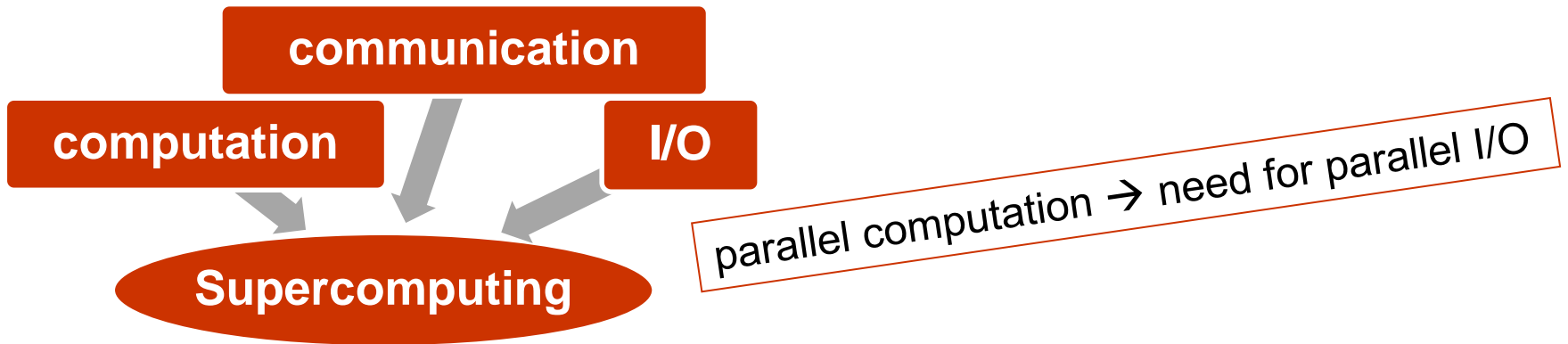
Parallel File I/O



calculation on	time for computation	time for <u>serial</u> I/O
1 core = sequential	64 min = 98.5 % of total time	1 min = 1.5 % of total time
64 cores = in parallel	1 min = 50 % of total time	1 min = 50 % of total time

Table: example with serial I/O

Parallel File I/O

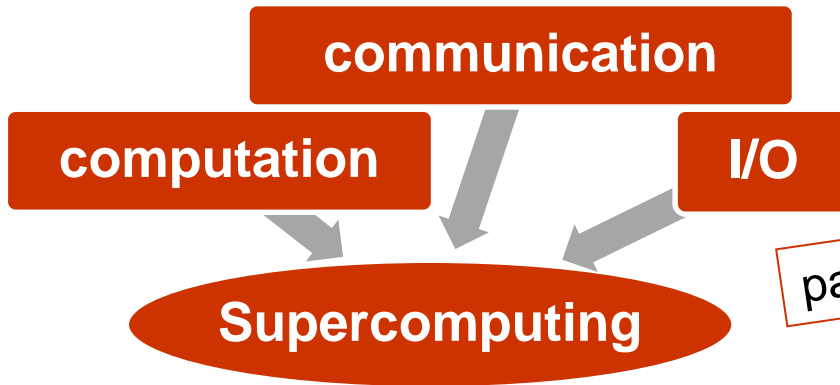


calculation on	time for computation	time for <u>serial</u> I/O
1 core = sequential	64 min = 98.5 % of total time	1 min = 1.5 % of total time
64 cores = in parallel	1 min = 50 % of total time	1 min = 50 % of total time

Table: example with serial I/O

1) waste of resources
2) negative side effects on other users

Parallel File I/O



parallel computation → need for parallel I/O

→ do parallel I/O → Ch. 13 !

calculation on	time for computation	time for <u>serial</u> I/O
1 core = sequentiel	64 min = 98.5 % of total time	1 min = 1.5 % of total time
64 cores = in parallel	1 min = 50 % of total time	1 min = 50 % of total time

Table: example with serial I/O

1) waste of resources
2) negative side effects on other users

Speedup, Efficiency, Scaleup, and Weak Scaling

- Definition: $T(p,N)$ = **time to solve problem of total size N on p processors**

- Parallel speedup: $S(p,N) = T(1,N) / T(p,N)$
compute **same problem** with more processors in **shorter time**

- Parallel Efficiency: $E(p,N) = S(p,N) / p$

Three different ways of reporting the success

Speedup, Efficiency, Scaleup, and Weak Scaling

- Definition: $T(p,N)$ = **time to solve problem of total size N on p processors**

- Parallel speedup: $S(p,N) = T(1,N) / T(p,N)$
compute **same problem** with more processors in **shorter time**

- Parallel Efficiency: $E(p,N) = S(p,N) / p$

- Scaleup: $Sc(p,N) = N / n$ with $T(1,n) = T(p,N)$
compute **larger problem** with more processors in **same time**

Three different ways of reporting the success

Speedup, Efficiency, Scaleup, and Weak Scaling

- Definition: $T(p,N)$ = **time to solve problem of total size N on p processors**

- Parallel speedup: $S(p,N) = T(1,N) / T(p,N)$
compute **same problem** with more processors in **shorter time**

- Parallel Efficiency: $E(p,N) = S(p,N) / p$

- Scaleup: $Sc(p,N) = N / n$ with $T(1,n) = T(p,N)$
compute **larger problem** with more processors in **same time**

- Weak scaling: $T(p, p \cdot n) / T(1,n)$ is reported,
i.e., problem size per process (n) is fixed

Three different ways of reporting the success

Speedup, Efficiency, Scaleup, and Weak Scaling

• Definition: $T(p,N)$ = **time to solve problem of total size N on p processors**

• Parallel speedup: $S(p,N) = T(1,N) / T(p,N)$
compute **same problem** with more processors in **shorter time**

• Parallel Efficiency: $E(p,N) = S(p,N) / p$

• Scaleup: $Sc(p,N) = N / n$ with $T(1,n) = T(p,N)$
compute **larger problem** with more processors in **same time**

• Weak scaling: $T(p, p \cdot n) / T(1,n)$ is reported,
i.e., problem size per process (n) is fixed

Three different ways of reporting the success

• Problems:

- Absolute MFLOPS rate / hardware peak performance?
- Super-scalar speedup: $S(p,N) > p$, e.g., due to cache^{*}) usage for large p :
 - $T(1,N)$ may be based on a huge number of N data elements in the memory in the one process, whereas
 - $T(p,N)$ may be based on *cache based execution* due to only N/p data elements per process
- $S(p,N)$ close to p or far less? → see Amdahl's Law on next slide

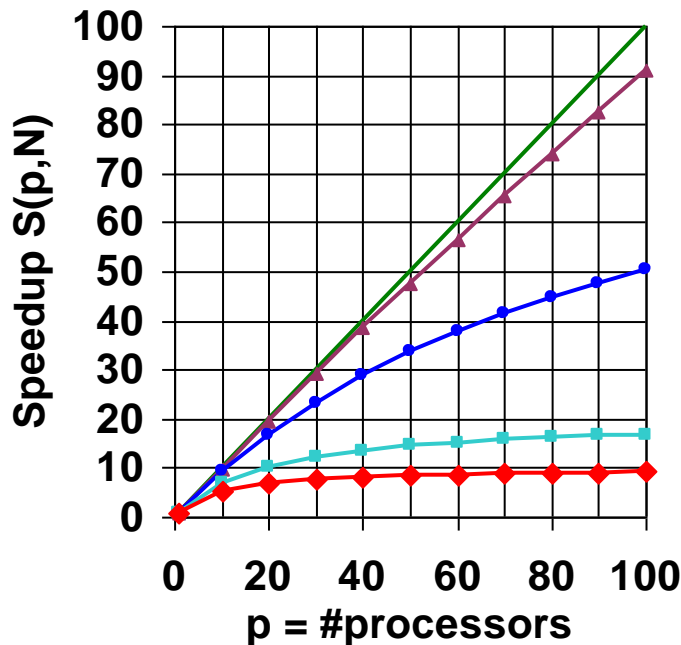
Amdahl's Law

$$T(p,N) = f \cdot T(1,N) + (1-f) \cdot T(1,N) / p$$

f ... sequential part of code that can not be done in parallel

$$S(p,N) = T(1,N) / T(p,N) = 1 / (f + (1-f) / p)$$

For $p \rightarrow$ infinity, speedup is limited by $S(p,N) < 1 / f$



— $S(p,N) = p$ (ideal speedup)

— $f=0.1\% \Rightarrow S(p,N) < 1000$

— $f= 1\% \Rightarrow S(p,N) < 100$

— $f= 5\% \Rightarrow S(p,N) < 20$

— $f= 10\% \Rightarrow S(p,N) < 10$

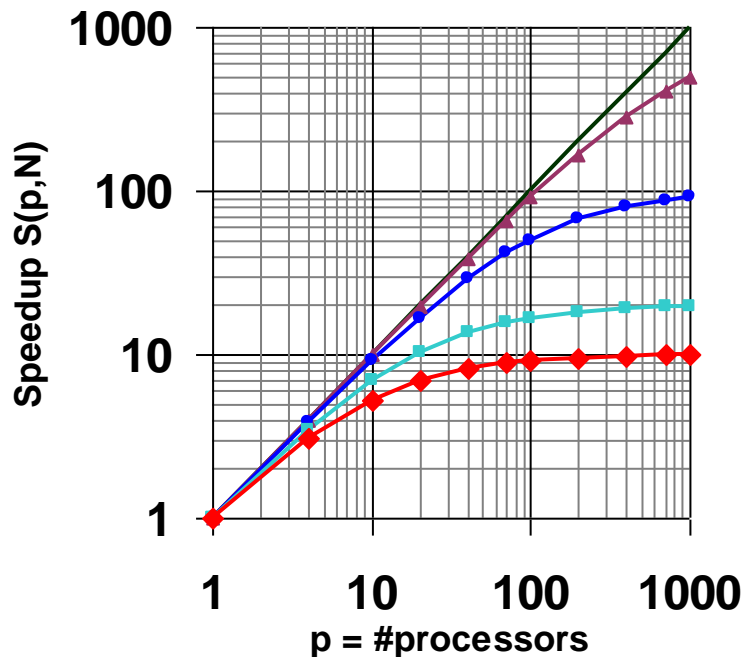
Amdahl's Law (double-logarithmic)

$$T(p,N) = f \cdot T(1,N) + (1-f) \cdot T(1,N) / p$$

f ... sequential part of code that can not be done in parallel

$$S(p,N) = T(1,N) / T(p,N) = 1 / (f + (1-f) / p)$$

For $p \rightarrow$ infinity, speedup is limited by $S(p,N) < 1 / f$



— $S(p,N) = p$ (ideal speedup)

—▲ $f=0.1\% \Rightarrow S(p,N) < 1000$

—● $f= 1\% \Rightarrow S(p,N) < 100$

—■ $f= 5\% \Rightarrow S(p,N) < 20$

—◆ $f= 10\% \Rightarrow S(p,N) < 10$

Quiz on Chapter 1 – Overview

Two developers report about their limited success when parallelizing an application:

- A. “My application is now running in parallel with 1000 MPI processes and my major limiting factor for scaling is
 - that I need about 90% of the whole compute time for MPI communication.”
- B. “My application is now running in parallel with 1000 MPI processes and my major limiting factor for scaling is
 - that I could not parallelize about 10% of the execution time of my sequential program.”

What are your answers for

- In your opinion, who was **more successful, A or B**, or both **almost equally**?
- Can you calculate an estimate for the parallel efficiency of the parallel run reported by A and B?