Cluster computing/Scripting/MPI

\\

NCC Beginner Training Event, 14^{th} - 16^{th} November 2022

\\

Giannis Koutsou,

Computation-based Science and Technology Research Center,

The Cyprus Institute

Day1 and Day2

Cluster computing and scripting

- Logging on to the system
- Basic scripting by example
- Submitting jobs, monitoring, etc.

Intro to MPI

- Overview of basic functions
- Examples

Day1 and Day2

Cluster computing and scripting

- Logging on to the system
- Basic scripting by example
- Submitting jobs, monitoring, etc.

Intro to MPI

- Overview of basic functions
- Examples

Two ways to get these slides

1. PDF from github: `https://github.com/CaSToRC-CyI/NCC-Beginner-Training-2022/raw/main/04_Scripting/slides.pdf`
2. Via browser: `https://sds402.online/ncc/`

Day1 and Day2

Cluster computing and scripting

- Logging on to the system
- Basic scripting by example
- Submitting jobs, monitoring, etc.

Intro to MPI

- Overview of basic functions
- Examples

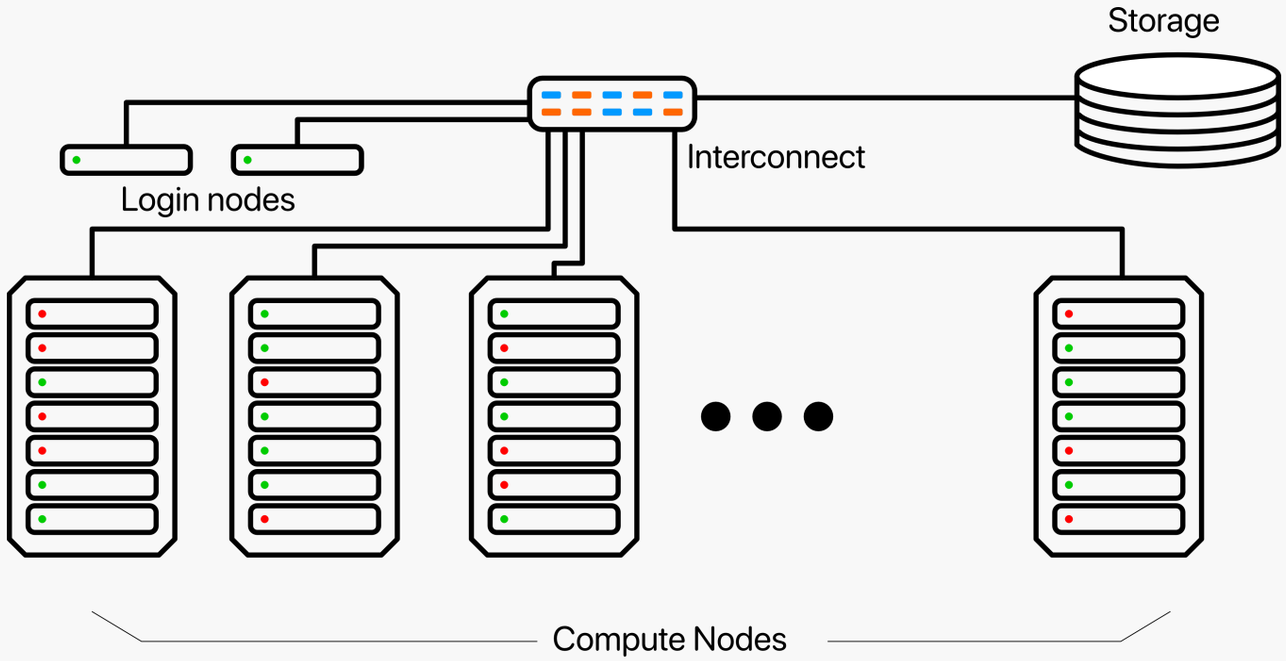
Two ways to get these slides

1. PDF from github: `https://github.com/CaSToRC-CyI/NCC-Beginner-Training-2022/raw/main/04_Scripting/slides.pdf`
2. Via browser: `https://sds402.online/ncc/`

Exercises

- Available on github:
  - `https://github.com/CaSToRC-CyI/NCC-Beginner-Training-2022/tree/main/04_Scripting`
  - `https://github.com/CaSToRC-CyI/NCC-Beginner-Training-2022/tree/main/05_MPI-Intro`
- On common storage on "Cyclone"
  - `/onyx/data/edu17/`

Cluster Computing



Specific configuration of our cluster "Cyclone"

- 33 nodes in total
  - Hostnames: `cn{01, ... ,17}`, `gpu{01, ... ,16}`
- We will be using CPU nodes: `cn??`
  - 2×20-core Intel Xeon (Gold) "Cascade Lake"
  - 192 GBytes RAM
- Common storage for our course: `/onyx/data/edu17/`

Cluster Computing

- Log in to a *login node* or *frontend node*. Login node in our case has hostname `front01`
- To run programs on *compute nodes*, a *job scheduler* is available
- Distinguish between *interactive* and *batch* jobs

Cluster Computing

- Log in to a *login node* or *frontend node*. Login node in our case has hostname `front01`
- To run programs on *compute nodes*, a *job scheduler* is available
- Distinguish between *interactive* and *batch* jobs

SLURM job scheduler

- See currently running and waiting jobs: `squeue`
- Ask for an interactive job: `salloc`
- Submit a batch job: `sbatch`
- Run an executable: `srun`

Cluster Computing Introductory Example

- Log in:

```
[localhost ~]$ ssh <username>@cyclone.hpcf.cyi.ac.cy
```

Cluster Computing Introductory Example

- Log in:

```
[localhost ~]$ ssh <username>@cyclone.hpcf.cyi.ac.cy
```

- Type `hostename`. This tells you the name of the node you are currently logged into:

```
[ikoutsou@front01 ~]$ hostname
front01
```

  this is the login node.

Cluster Computing Introductory Example

- Log in:

```
[localhost ~]$ ssh <username>@cyclone.hpcf.cyi.ac.cy
```

- Type `hostename`. This tells you the name of the node you are currently logged into:

```
[ikoutsou@front01 ~]$ hostname
front01
```

  this is the login node.

- Ask for one node (`-N 1`):

```
[ikoutsou@front01 ~]$ salloc -N 1 -p cpu --reservation=NCC -A edu17
salloc: Granted job allocation 197848
salloc: Waiting for resource configuration
salloc: Nodes cn17 are ready for job
[ikoutsou@cn17 ~]$
```

Cluster Computing Introductory Example

- Log in:

```
[localhost ~]$ ssh <username>@cyclone.hpcf.cyi.ac.cy
```

- Type `hostename`. This tells you the name of the node you are currently logged into:

```
[ikoutsou@front01 ~]$ hostname
front01
```

  this is the login node.

- Ask for one node (`-N 1`):

```
[ikoutsou@front01 ~]$ salloc -N 1 -p cpu --reservation=NCC -A edu17
salloc: Granted job allocation 197848
salloc: Waiting for resource configuration
salloc: Nodes cn17 are ready for job
[ikoutsou@cn17 ~]$
```

  - `-A edu17`: charge project with name `edu17`

Cluster Computing Introductory Example

- Log in:

```
[localhost ~]$ ssh <username>@cyclone.hpcf.cyi.ac.cy
```

- Type `hostename`. This tells you the name of the node you are currently logged into:

```
[ikoutsou@front01 ~]$ hostname
front01
```

  this is the login node.

- Ask for one node (`-N 1`):

```
[ikoutsou@front01 ~]$ salloc -N 1 -p cpu --reservation=NCC -A edu17
salloc: Granted job allocation 197848
salloc: Waiting for resource configuration
salloc: Nodes cn17 are ready for job
[ikoutsou@cn17 ~]$
```

  - `-A edu17`: charge project with name `edu17`
  - `--reservation=NCC`: use reservation `NCC`

Cluster Computing Introductory Example

- Log in:

```
[localhost ~]$ ssh <username>@cyclone.hpcf.cyi.ac.cy
```

- Type `hostename`. This tells you the name of the node you are currently logged into:

```
[ikoutsou@front01 ~]$ hostname
front01
```

  this is the login node.

- Ask for one node (`-N 1`):

```
[ikoutsou@front01 ~]$ salloc -N 1 -p cpu --reservation=NCC -A edu17
salloc: Granted job allocation 197848
salloc: Waiting for resource configuration
salloc: Nodes cn17 are ready for job
[ikoutsou@cn17 ~]$
```

  - `-A edu17`: charge project with name `edu17`
  - `--reservation=NCC`: use reservation `NCC`
  - `-p=cpu`: requests a node from the `cpu` partition

Cluster Computing Introductory Example

- Log in:

```
[localhost ~]$ ssh <username>@cyclone.hpcf.cyi.ac.cy
```

- Type `hostename`. This tells you the name of the node you are currently logged into:

```
[ikoutsou@front01 ~]$ hostname
front01
```

  this is the login node.

- Ask for one node (`-N 1`):

```
[ikoutsou@front01 ~]$ salloc -N 1 -p cpu --reservation=NCC -A edu17
salloc: Granted job allocation 197848
salloc: Waiting for resource configuration
salloc: Nodes cn17 are ready for job
[ikoutsou@cn17 ~]$
```

  - `-A edu17`: charge project with name `edu17`
  - `--reservation=NCC`: use reservation `NCC`
  - `-p=cpu`: requests a node from the `cpu` partition

- Type `hostname` again:

```
[ikoutsou@cn17 ~]$ hostname
cn17
```

  `cn17` is a compute node.

Cluster Computing Introductory Example

- Release the node (type `exit` or hit `ctrl-d`):

```
[ikoutsou@cn17 ~]$ exit
salloc: Relinquishing job allocation 197848
[ikoutsou@front01 ~]$
```

  we're back on `front01`

Cluster Computing Introductory Example

- Release the node (type `exit` or hit `ctrl-d`):

```
[ikoutsou@cn17 ~]$ exit
salloc: Relinquishing job allocation 197848
[ikoutsou@front01 ~]$
```

  we're back on `front01`

  Please **do not** hold nodes unnecessarily; when you have nodes `salloc`ed you may be blocking other users from using those nodes.

Cluster Computing Introductory Example

- Release the node (type `exit` or hit `ctrl-d`):

```
[ikoutsou@cn17 ~]$ exit
salloc: Relinquishing job allocation 197848
[ikoutsou@front01 ~]$
```

  we're back on `front01`

  Please **do not** hold nodes unnecessarily; when you have nodes `salloc`ed you may be blocking other users from using those nodes.

- Use `srun` instead of `salloc`:

```
[ikoutsou@front01 ~]$ srun -N 1 -p cpu -A edu17 --reservation=NCC hostname
cn15
```

Cluster Computing Introductory Example

- Release the node (type `exit` or hit `ctrl-d`):

```
[ikoutsou@cn17 ~]$ exit
salloc: Relinquishing job allocation 197848
[ikoutsou@front01 ~]$
```

  we're back on `front01`

  Please **do not** hold nodes unnecessarily; when you have nodes `salloc`ed you may be blocking other users from using those nodes.

- Use `srun` instead of `salloc`:

```
[ikoutsou@front01 ~]$ srun -N 1 -p cpu -A edu17 --reservation=NCC hostname
cn15
```

- Allocates a node, runs the specified command (in this case `hostname`), and then exits the node, releasing the allocation

Cluster Computing Introductory Example

- Release the node (type `exit` or hit `ctrl-d`):

```
[ikoutsou@cn17 ~]$ exit
salloc: Relinquishing job allocation 197848
[ikoutsou@front01 ~]$
```

  we're back on `front01`

  Please **do not** hold nodes unnecessarily; when you have nodes `salloc`ed you may be blocking other users from using those nodes.

- Use `srun` instead of `salloc`:

```
[ikoutsou@front01 ~]$ srun -N 1 -p cpu -A edu17 --reservation=NCC hostname
cn15
```

- Allocates a node, runs the specified command (in this case `hostname`), and then exits the node, releasing the allocation
- The output, `cn15`, reveals that we were allocated node `cn15` for this specific — very short — job

Cluster Computing Introductory Example

- Run multiple instances of `hostname` in parallel:

```
[ikoutsou@front01 ~]$ srun -N 1 -n 2 -p cpu -A edu17 --reservation=NCC hostname
cn15
cn15
```

Cluster Computing Introductory Example

- Run multiple instances of `hostname` in parallel:

```
[ikoutsou@front01 ~]$ srun -N 1 -n 2 -p cpu -A edu17 --reservation=NCC hostname
cn15
cn15
```

  - `-N 1`: use one node
  - `-n 2`: use two processes

Cluster Computing Introductory Example

- Run multiple instances of `hostname` in parallel:

```
[ikoutsou@front01 ~]$ srun -N 1 -n 2 -p cpu -A edu17 --reservation=NCC hostname
cn15
cn15
```

  - `-N 1`: use one node
  - `-n 2`: use two processes

- Run on more than one node:

```
[ikoutsou@front01 ~]$ srun -N 2 -n 2 -p cpu -A edu17 --reservation=NCC hostname
cn02
cn01
```

  runs one instance of `hostname` on each node

Cluster Computing Introductory Example

- Run multiple instances of `hostname` in parallel:

```
[ikoutsou@front01 ~]$ srun -N 1 -n 2 -p cpu -A edu17 --reservation=NCC hostname
cn15
cn15
```

  - `-N 1`: use one node
  - `-n 2`: use two processes

- Run on more than one node:

```
[ikoutsou@front01 ~]$ srun -N 2 -n 2 -p cpu -A edu17 --reservation=NCC hostname
cn02
cn01
```

  runs one instance of `hostname` on each node

- Try:

```
[ikoutsou@front01 ~]$ srun -N 2 -n 1 -p cpu -A edu17 --reservation=NCC hostname
```

Cluster Computing Introductory Example

- Run multiple instances of `hostname` in parallel:

```
[ikoutsou@front01 ~]$ srun -N 1 -n 2 -p cpu -A edu17 --reservation=NCC hostname
cn15
cn15
```

  - `-N 1`: use one node
  - `-n 2`: use two processes

- Run on more than one node:

```
[ikoutsou@front01 ~]$ srun -N 2 -n 2 -p cpu -A edu17 --reservation=NCC hostname
cn02
cn01
```

  runs one instance of `hostname` on each node

- Try:

```
[ikoutsou@front01 ~]$ srun -N 2 -n 1 -p cpu -A edu17 --reservation=NCC hostname
```

```
srun: Warning: can't run 1 processes on 2 nodes, setting nnodes to 1
cn17
```

Cluster Computing Introductory Example

- Make a directory. List it, see that it is there:

```
[ikoutsou@front01 ~]$ mkdir NCC-training
[ikoutsou@front01 ~]$ ls
NCC-training
[ikoutsou@front01 ~]$
```

Cluster Computing Introductory Example

- Make a directory. List it, see that it is there:

```
[ikoutsou@front01 ~]$ mkdir NCC-training
[ikoutsou@front01 ~]$ ls
NCC-training
[ikoutsou@front01 ~]$
```

- Change into it:

```
[ikoutsou@front01 ~]$ cd NCC-training/
[ikoutsou@front01 NCC-training]$
```

Cluster Computing Introductory Example

- Make a directory. List it, see that it is there:

```
[ikoutsou@front01 ~]$ mkdir NCC-training
[ikoutsou@front01 ~]$ ls
NCC-training
[ikoutsou@front01 ~]$
```

- Change into it:

```
[ikoutsou@front01 ~]$ cd NCC-training/
[ikoutsou@front01 NCC-training]$
```

- `pwd` will tell you where you are in the file system:

```
[ikoutsou@front01 NCC-training]$ pwd
/nvme/h/ikoutsou/NCC-training
```

Cluster Computing Introductory Example

- Make a directory. List it, see that it is there:

```
[ikoutsou@front01 ~]$ mkdir NCC-training
[ikoutsou@front01 ~]$ ls
NCC-training
[ikoutsou@front01 ~]$
```

- Change into it:

```
[ikoutsou@front01 ~]$ cd NCC-training/
[ikoutsou@front01 NCC-training]$
```

- `pwd` will tell you where you are in the file system:

```
[ikoutsou@front01 NCC-training]$ pwd
/nvme/h/ikoutsou/NCC-training
```

- `/` is referred to as the *root directory*

Cluster Computing Introductory Example

- Make a directory. List it, see that it is there:

```
[ikoutsou@front01 ~]$ mkdir NCC-training
[ikoutsou@front01 ~]$ ls
NCC-training
[ikoutsou@front01 ~]$
```

- Change into it:

```
[ikoutsou@front01 ~]$ cd NCC-training/
[ikoutsou@front01 NCC-training]$
```

- `pwd` will tell you where you are in the file system:

```
[ikoutsou@front01 NCC-training]$ pwd
/nvme/h/ikoutsou/NCC-training
```

- `/` is referred to as the *root directory*
- `.` is an alias for the *current directory*

Cluster Computing Introductory Example

- Make a directory. List it, see that it is there:

```
[ikoutsou@front01 ~]$ mkdir NCC-training
[ikoutsou@front01 ~]$ ls
NCC-training
[ikoutsou@front01 ~]$
```

- Change into it:

```
[ikoutsou@front01 ~]$ cd NCC-training/
[ikoutsou@front01 NCC-training]$
```

- `pwd` will tell you where you are in the file system:

```
[ikoutsou@front01 NCC-training]$ pwd
/nvme/h/ikoutsou/NCC-training
```

- `/` is referred to as the *root directory*
- `.` is an alias for the *current directory*
- `..` is an alias for the directory one level above

Cluster Computing Introductory Example

- Make a directory. List it, see that it is there:

```
[ikoutsou@front01 ~]$ mkdir NCC-training
[ikoutsou@front01 ~]$ ls
NCC-training
[ikoutsou@front01 ~]$
```

- Change into it:

```
[ikoutsou@front01 ~]$ cd NCC-training/
[ikoutsou@front01 NCC-training]$
```

- `pwd` will tell you where you are in the file system:

```
[ikoutsou@front01 NCC-training]$ pwd
/nvme/h/ikoutsou/NCC-training
```

- `/` is referred to as the *root directory*
- `.` is an alias for the *current directory*
- `..` is an alias for the directory one level above
- `~` is an alias for your *home directory*

Cluster Computing Introductory Example

- Make a directory. List it, see that it is there:

```
[ikoutsou@front01 ~]$ mkdir NCC-training
[ikoutsou@front01 ~]$ ls
NCC-training
[ikoutsou@front01 ~]$
```

- Change into it:

```
[ikoutsou@front01 ~]$ cd NCC-training/
[ikoutsou@front01 NCC-training]$
```

- `pwd` will tell you where you are in the file system:

```
[ikoutsou@front01 NCC-training]$ pwd
/nvme/h/ikoutsou/NCC-training
```

- `/` is referred to as the *root directory*
- `.` is an alias for the *current directory*
- `..` is an alias for the directory one level above
- `~` is an alias for your *home directory*
- E.g.:

```
[ikoutsou@front01 NCC-training]$ cd ../../
[ikoutsou@front01 h]$ pwd
/nvme/h
```

Cluster Computing Introductory Example

- `cd` without any additional arguments takes you home (equivalent to `cd ~`)

```
[ikoutsou@front01 h]$ cd
[ikoutsou@front01 ~]$ pwd
/nvme/h/ikoutsou
```

Cluster Computing Introductory Example

- `cd` without any additional arguments takes you home (equivalent to `cd ~`)

```
[ikoutsou@front01 h]$ cd
[ikoutsou@front01 ~]$ pwd
/nvme/h/ikoutsou
```

- Make a subdirectory under `NCC-training` for our first C program

```
[ikoutsou@front01 ~]$ cd NCC-training
[ikoutsou@front01 NCC-training]$ mkdir 01
[ikoutsou@front01 NCC-training]$ cd 01
[ikoutsou@front01 01]$
```

Cluster Computing Introductory Example

- `cd` without any additional arguments takes you home (equivalent to `cd ~`)

```
[ikoutsou@front01 h]$ cd
[ikoutsou@front01 ~]$ pwd
/nvme/h/ikoutsou
```

- Make a subdirectory under `NCC-training` for our first C program

```
[ikoutsou@front01 ~]$ cd NCC-training
[ikoutsou@front01 NCC-training]$ mkdir 01
[ikoutsou@front01 NCC-training]$ cd 01
[ikoutsou@front01 01]$
```

- Use `emacs` or `vim` (or any other text editor you're comfortable with) to type out our first program

Cluster Computing Introductory Example

- `cd` without any additional arguments takes you home (equivalent to `cd ~`)

```
[ikoutsou@front01 h]$ cd
[ikoutsou@front01 ~]$ pwd
/nvme/h/ikoutsou
```

- Make a subdirectory under `NCC-training` for our first C program

```
[ikoutsou@front01 ~]$ cd NCC-training
[ikoutsou@front01 NCC-training]$ mkdir 01
[ikoutsou@front01 NCC-training]$ cd 01
[ikoutsou@front01 01]$
```

- Use `emacs` or `vim` (or any other text editor you're comfortable with) to type out our first program
- E.g.:

```
[ikoutsou@front01 01]$ emacs -nw prog-01.c
```

Cluster Computing Introductory Example

- Type out the following program:

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>



int
main(int argc, char *argv[])
{
  char hname[256];
  pid_t p;
  gethostname(hname, 256);
  p = getpid();
  printf(" Hostname: %s, pid: %lu\n", hname, p);
  return 0;

}
```

Cluster Computing Introductory Example

- Type out the following program:

```c
#include <unistd.h>           // ⟵ provides definitions for gethostname() and getpid()
#include <stdio.h>            // ⟵ provides definitions for printf()
#include <sys/types.h>        // ⟵ defines the pid_t type
// main()
//   → argv[] is an array of strings which holds all command line arguments
//   → argc holds the number of elements of argv
int
main(int argc, char *argv[])
{
  char hname[256];          // ⟵ declare hname[] as an array of 256 characters (a string of length 256)
  pid_t p;                  // ⟵ declare p as a pid_t type, in this case, an unsigned long integer
  gethostname(hname, 256); // ⟵ call gethostname(), return hostname in hname which is 256 characters long
  p = getpid();             // ⟵ call getpid(), return value in p
  printf(" Hostname: %s, pid: %lu\n", hname, p); // print statement
  return 0; // ⟵ return a value of 0 to the operating system. By convention 0 means success.
            //     Non zero values indicate errors.
}
```

Cluster Computing Introductory Example

After saving and exiting, you are back at the command line

Cluster Computing Introductory Example

After saving and exiting, you are back at the command line

- `ls` should show the file `prog-01.c`, which you just typed in and saved:

```
[ikoutsou@front01 01]$ ls
prog-01.c
```

Cluster Computing Introductory Example

After saving and exiting, you are back at the command line

- `ls` should show the file `prog-01.c`, which you just typed in and saved:

```
[ikoutsou@front01 01]$ ls
prog-01.c
```

- It's time to *compile* it into an executable. Use the `gompi` module:

```
[ikoutsou@front01 01]$ module load gompi
[ikoutsou@front01 01]$ gcc prog-01.c -o p01
```

Cluster Computing Introductory Example

After saving and exiting, you are back at the command line

- `ls` should show the file `prog-01.c`, which you just typed in and saved:

```
[ikoutsou@front01 01]$ ls
prog-01.c
```

- It's time to *compile* it into an executable. Use the `gompi` module:

```
[ikoutsou@front01 01]$ module load gompi
[ikoutsou@front01 01]$ gcc prog-01.c -o p01
```

  - `-o p01` means "name the resulting executable `p01`". If you don't specify `-o` the executable name defaults to `a.out`

Cluster Computing Introductory Example

After saving and exiting, you are back at the command line

- `ls` should show the file `prog-01.c`, which you just typed in and saved:

```
[ikoutsou@front01 01]$ ls
prog-01.c
```

- It's time to *compile* it into an executable. Use the `gompi` module:

```
[ikoutsou@front01 01]$ module load gompi
[ikoutsou@front01 01]$ gcc prog-01.c -o p01
```

  - `-o p01` means "name the resulting executable `p01`". If you don't specify `-o` the executable name defaults to `a.out`

- Type `ls` to make sure it has been created. Then run it on the frontend node:

```
[ikoutsou@front01 01]$ ls
p01  prog-01.c
[ikoutsou@front01 01]$ ./p01
Hostname: front01, pid: 14848
```

Cluster Computing Introductory Example

After saving and exiting, you are back at the command line

- `ls` should show the file `prog-01.c`, which you just typed in and saved:

```
[ikoutsou@front01 01]$ ls
prog-01.c
```

- It's time to *compile* it into an executable. Use the `gompi` module:

```
[ikoutsou@front01 01]$ module load gompi
[ikoutsou@front01 01]$ gcc prog-01.c -o p01
```

  - `-o p01` means "name the resulting executable `p01`". If you don't specify `-o` the executable name defaults to `a.out`

- Type `ls` to make sure it has been created. Then run it on the frontend node:

```
[ikoutsou@front01 01]$ ls
p01  prog-01.c
[ikoutsou@front01 01]$ ./p01
Hostname: front01, pid: 14848
```

Note: commands like `gcc` or `ls` are globally accessible because their locations are included in your shell environment's search path. for `p01` though, which we just created, you need to explicitly give its path, in this case via `./` which means "current directory".

Cluster Computing Introductory Example

- Run your new program `p01` using `srun` on two nodes with two processes each

Cluster Computing Introductory Example

- Run your new program `p01` using `srun` on two nodes with two processes each

```
[ikoutsou@front01 01]$ srun -N 2 -n 4 -p cpu --reservation=NCC -A edu17 ./p01
Hostname: cn02, pid: 11037
Hostname: cn01, pid: 8207
Hostname: cn01, pid: 8209
Hostname: cn01, pid: 8208
[ikoutsou@front01 01]$
```

Cluster Computing Introductory Example

- Run your new program `p01` using `srun` on two nodes with two processes each

```
[ikoutsou@front01 01]$ srun -N 2 -n 4 -p cpu --reservation=NCC -A edu17 ./p01
Hostname: cn02, pid: 11037
Hostname: cn01, pid: 8207
Hostname: cn01, pid: 8209
Hostname: cn01, pid: 8208
[ikoutsou@front01 01]$
```

- Go nuts 😆 :

```
[ikoutsou@front01 01]$ srun -N 2 -n 48 -p cpu --reservation=NCC -A edu17 ./p01
Hostname: cn02, pid: 11294
Hostname: cn02, pid: 11300
 ...
Hostname: cn01, pid: 8482
Hostname: cn01, pid: 8474
[ikoutsou@front01 01]$
```

(You will see 48 lines; I suppressed some above)

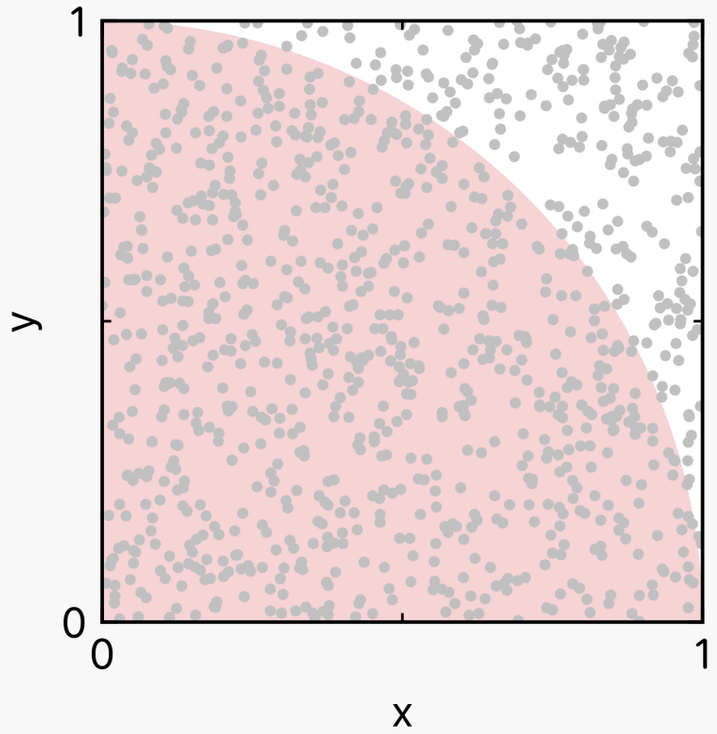
Cluster Computing Introductory Example

- Time for step 2 — writing a simple program to compute π in parallel

Cluster Computing Introductory Example

- Time for step 2 — writing a simple program to compute π in parallel

- Calculation of π via simple Monte Carlo:
  - Define a unit square. Set $n_{hit} = 0$
  - Randomly pick points (x, y) within the unit square
  - If $x^2 + y^2 < 1$, $n_{hit} += 1$
  - Repeat N times
- The ratio n_{hit}/N approaches the area of a circle quadrant $\Rightarrow \frac{\pi}{4}$



Cluster Computing Introductory Example

- First, make a new directory under `~/NCC-training/.`

```
[ikoutsou@front01 01]$ cd ../
[ikoutsou@front01 NCC-training]$ mkdir 02
[ikoutsou@front01 01]$ cd 02/
[ikoutsou@front01 02]$
```

Cluster Computing Introductory Example

- First, make a new directory under `~/NCC-training/.`

```
[ikoutsou@front01 01]$ cd ../
[ikoutsou@front01 NCC-training]$ mkdir 02
[ikoutsou@front01 01]$ cd 02/
[ikoutsou@front01 02]$
```

- Copy the program `pi.c` from `/onyx/data/edu17/02/pi.c`

```
[ikoutsou@front01 02]$ cp /onyx/data/edu17/02/pi.c .
```

- Inspect `pi.c`, e.g.:

```
[ikoutsou@front01 02]$ emacs -nw pi.c
```

Cluster Computing Introductory Example

- First, make a new directory under `~/NCC-training/.`

```
[ikoutsou@front01 01]$ cd ../
[ikoutsou@front01 NCC-training]$ mkdir 02
[ikoutsou@front01 01]$ cd 02/
[ikoutsou@front01 02]$
```

- Copy the program `pi.c` from `/onyx/data/edu17/02/pi.c`

```
[ikoutsou@front01 02]$ cp /onyx/data/edu17/02/pi.c .
```

- Inspect `pi.c`, e.g.:

```
[ikoutsou@front01 02]$ emacs -nw pi.c
```

Cluster Computing Introductory Example

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

int
main(int argc, char *argv[])
{
  unsigned long int N = 10000;
  unsigned long int nhit = 0;
  for(int i=0; i<N; i++) {
    double x = drand48();
    double y = drand48();
    if((x*x + y*y) < 1)
      nhit++;
  }
  double pi = 4.0 * (double)nhit/(double)N;
  printf(" N = %16d   pi = %lf\n", N, pi);
  return 0;
}
```

Cluster Computing Introductory Example

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

int
main(int argc, char *argv[])
{
  unsigned long int N = 10000;
  unsigned long int nhit = 0;
  for(int i=0; i<N; i++) {
    double x = drand48();
    double y = drand48();
    if((x*x + y*y) < 1)
      nhit++;
  }
  double pi = 4.0 * (double)nhit/(double)N;
  printf(" N = %16d   pi = %lf\n", N, pi);
  return 0;
}
```

- Compile and run on frontend:

```
[ikoutsou@front01 02]$ gcc pi.c -o pi
[ikoutsou@front01 02]$ ./pi
N =            10000   pi = 3.136400
```

Cluster Computing Introductory Example

- Now run, e.g. on 4 processes:

Cluster Computing Introductory Example

- Now run, e.g. on 4 processes:

```
[ikoutsou@front01 02]$ srun -N 1 -n 4 -p cpu -A edu17 --reservation=NCC ./pi
N =            10000   pi = 3.148800
N =            10000   pi = 3.148800
N =            10000   pi = 3.148800
N =            10000   pi = 3.148800
```

Cluster Computing Introductory Example

- Now run, e.g. on 4 processes:

```
[ikoutsou@front01 02]$ srun -N 1 -n 4 -p cpu -A edu17 --reservation=NCC ./pi
N =            10000   pi = 3.148800
N =            10000   pi = 3.148800
N =            10000   pi = 3.148800
N =            10000   pi = 3.148800
```

  We get exactly the same result four times 🤔

Cluster Computing Introductory Example

- Now run, e.g. on 4 processes:

```
[ikoutsou@front01 02]$ srun -N 1 -n 4 -p cpu -A edu17 --reservation=NCC ./pi
N =            10000   pi = 3.148800
N =            10000   pi = 3.148800
N =            10000   pi = 3.148800
N =            10000   pi = 3.148800
```

We get exactly the same result four times 🤔

We need to seed the random number generator differently for each process

Cluster Computing Introductory Example

- Now run, e.g. on 4 processes:

```
[ikoutsou@front01 02]$ srun -N 1 -n 4 -p cpu -A edu17 --reservation=NCC ./pi
N =            10000   pi = 3.148800
N =            10000   pi = 3.148800
N =            10000   pi = 3.148800
N =            10000   pi = 3.148800
```

  We get exactly the same result four times 🤔

  We need to seed the random number generator differently for each process

- Use the process id (`pid`) from the previous example, to seed the random number generator

Cluster Computing Introductory Example

- Use the process id (`pid`) from the previous example, to seed the random number generator

Cluster Computing Introductory Example

- Use the process id (`pid`) from the previous example, to seed the random number generator

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
int
main(int argc, char *argv[])
{
  unsigned long int N = 10000;
  unsigned long int nhit = 0;

  pid_t p = getpid(); // ⟵ Add this
  srand48(p);         // ⟵ Add this

  for(int i=0; i<N; i++) {
    double x = drand48();
    double y = drand48();
    if((x*x + y*y) < 1)
      nhit++;
  }
  double pi = 4.0 * (double)nhit/(double)N;
  printf(" N = %16d   pi = %lf\n", N, pi);
  return 0;
}
```

Cluster Computing Introductory Example

- Use the process id (`pid`) from the previous example, to seed the random number generator

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
int
main(int argc, char *argv[])
{
  unsigned long int N = 10000;
  unsigned long int nhit = 0;

  pid_t p = getpid(); // ⟵ Add this
  srand48(p);         // ⟵ Add this

  for(int i=0; i<N; i++) {
    double x = drand48();
    double y = drand48();
    if((x*x + y*y) < 1)
      nhit++;
  }
  double pi = 4.0 * (double)nhit/(double)N;
  printf(" N = %16d   pi = %lf\n", N, pi);
  return 0;
}
```

- `srand48()` sets the random number generator seed

Cluster Computing Introductory Example

- Use the process id (`pid`) from the previous example, to seed the random number generator

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
int
main(int argc, char *argv[])
{
  unsigned long int N = 10000;
  unsigned long int nhit = 0;

  pid_t p = getpid(); // ⟵ Add this
  srand48(p);         // ⟵ Add this

  for(int i=0; i<N; i++) {
    double x = drand48();
    double y = drand48();
    if((x*x + y*y) < 1)
      nhit++;
  }
  double pi = 4.0 * (double)nhit/(double)N;
  printf(" N = %16d   pi = %lf\n", N, pi);
  return 0;
}
```

- `srand48()` sets the random number generator seed
- Need a unique seed for each instance of the program $\Rightarrow$ use process id.

Cluster Computing Introductory Example

- Compile again and run:

```
[ikoutsou@front01 02]$ gcc pi.c -o pi
[ikoutsou@front01 02]$ srun -N 1 -n 4 -p cpu -A edu17 --reservation=NCC ./pi
N =            10000   pi = 3.150800
N =            10000   pi = 3.143200
N =            10000   pi = 3.151200
N =            10000   pi = 3.152800
```

Cluster Computing Introductory Example

- Compile again and run:

```
[ikoutsou@front01 02]$ gcc pi.c -o pi
[ikoutsou@front01 02]$ srun -N 1 -n 4 -p cpu -A edu17 --reservation=NCC ./pi
N =            10000   pi = 3.150800
N =            10000   pi = 3.143200
N =            10000   pi = 3.151200
N =            10000   pi = 3.152800
```

- Now we would like to average over these four values to obtain a better estimate of π

Cluster Computing Introductory Example

- Compile again and run:

```
[ikoutsou@front01 02]$ gcc pi.c -o pi
[ikoutsou@front01 02]$ srun -N 1 -n 4 -p cpu -A edu17 --reservation=NCC ./pi
N =            10000   pi = 3.150800
N =            10000   pi = 3.143200
N =            10000   pi = 3.151200
N =            10000   pi = 3.152800
```

- Now we would like to average over these four values to obtain a better estimate of π
- First redirect the output to a file, e.g.:

```
[ikoutsou@front01 02]$ gcc pi.c -o pi
[ikoutsou@front01 02]$ srun -N 1 -n 4 -p cpu -A edu17 --reservation=NCC ./pi > pi-out.txt
```

Cluster Computing Introductory Example

- Compile again and run:

```
[ikoutsou@front01 02]$ gcc pi.c -o pi
[ikoutsou@front01 02]$ srun -N 1 -n 4 -p cpu -A edu17 --reservation=NCC ./pi
N =            10000   pi = 3.150800
N =            10000   pi = 3.143200
N =            10000   pi = 3.151200
N =            10000   pi = 3.152800
```

- Now we would like to average over these four values to obtain a better estimate of π
- First redirect the output to a file, e.g.:

```
[ikoutsou@front01 02]$ gcc pi.c -o pi
[ikoutsou@front01 02]$ srun -N 1 -n 4 -p cpu -A edu17 --reservation=NCC ./pi > pi-out.txt
```

- Now `pi-out.txt` contains the four lines of output

```
[ikoutsou@front01 02]$ ls
pi  pi.c  pi-out.txt
[ikoutsou@front01 02]$ more pi-out.txt
N =            10000   pi = 3.113600
N =            10000   pi = 3.128400
N =            10000   pi = 3.156800
N =            10000   pi = 3.148400
[ikoutsou@front01 02]$
```

Cluster Computing Introductory Example

- The program `awk` allows us to add over columns of a file.

Cluster Computing Introductory Example

- The program `awk` allows us to add over columns of a file.
- E.g.:

```
[ikoutsou@front01 02]$ cat pi-out.txt | awk '{pi_sum+=$6}; END {printf "%8.6f\n", pi_sum/NR}'
3.136800
```

Cluster Computing Introductory Example

- The program `awk` allows us to add over columns of a file.
- E.g.:

```
[ikoutsou@front01 02]$ cat pi-out.txt | awk '{pi_sum+=$6}; END {printf "%8.6f\n", pi_sum/NR}'
3.136800
```

- Use more processes

```
[ikoutsou@front01 02]$ srun -N 2 -n 80 -p cpu -A edu17 --reservation=NCC ./pi > pi-out.txt
[ikoutsou@front01 02]$ cat pi-out.txt | awk '{pi_sum+=$6}; END {printf "%8.6f\n", pi_sum/NR}'
3.142435
```

Cluster Computing Introductory Example

- The program `awk` allows us to add over columns of a file.
- E.g.:

```
[ikoutsou@front01 02]$ cat pi-out.txt | awk '{pi_sum+=$6}; END {printf "%8.6f\n", pi_sum/NR}'
3.136800
```

- Use more processes

```
[ikoutsou@front01 02]$ srun -N 2 -n 80 -p cpu -A edu17 --reservation=NCC ./pi > pi-out.txt
[ikoutsou@front01 02]$ cat pi-out.txt | awk '{pi_sum+=$6}; END {printf "%8.6f\n", pi_sum/NR}'
3.142435
```

  - `$6` is the sixth column in the file, the value for π per process
  - `pi_sum` is our summation variabl
  - `NR` is an AWK internal variable, the number of rows

Cluster Computing Introductory Example

- The program `awk` allows us to add over columns of a file.
- E.g.:

```
[ikoutsou@front01 02]$ cat pi-out.txt | awk '{pi_sum+=$6}; END {printf "%8.6f\n", pi_sum/NR}'
3.136800
```

- Use more processes

```
[ikoutsou@front01 02]$ srun -N 2 -n 80 -p cpu -A edu17 --reservation=NCC ./pi > pi-out.txt
[ikoutsou@front01 02]$ cat pi-out.txt | awk '{pi_sum+=$6}; END {printf "%8.6f\n", pi_sum/NR}'
3.142435
```

  - `$6` is the sixth column in the file, the value for π per process
  - `pi_sum` is our summation variabl
  - `NR` is an AWK internal variable, the number of rows

- Let's wrap this up in a script "for posterity"

Cluster Computing Introductory Example

- The program `awk` allows us to add over columns of a file.
- E.g.:

```
[ikoutsou@front01 02]$ cat pi-out.txt | awk '{pi_sum+=$6}; END {printf "%8.6f\n", pi_sum/NR}'
3.136800
```

- Use more processes

```
[ikoutsou@front01 02]$ srun -N 2 -n 80 -p cpu -A edu17 --reservation=NCC ./pi > pi-out.txt
[ikoutsou@front01 02]$ cat pi-out.txt | awk '{pi_sum+=$6}; END {printf "%8.6f\n", pi_sum/NR}'
3.142435
```

  - `$6` is the sixth column in the file, the value for π per process
  - `pi_sum` is our summation variabl
  - `NR` is an AWK internal variable, the number of rows

- Let's wrap this up in a script "for posterity"
- In fact, we'll write a *Slurm batch script*

Cluster Computing Introductory Example

- Copy from `/onyx/data/edu17/02/pi.sh`

```
#!/bin/bash
#SBATCH -J pi
#SBATCH -o pi.txt
#SBATCH -e pi.err
#SBATCH -p cpu
#SBATCH -A edu17
#SBATCH --reservation=NCC
#SBATCH -t 00:02:00
#SBATCH -n 80
#SBATCH -N 2

### Add these two lines
srun ./pi > pi-out.txt
cat pi-out.txt | awk '{sum+=$6}; END {printf "%8.6f\n", sum/NR}'
```

Cluster Computing Introductory Example

- Copy from `/onyx/data/edu17/02/pi.sh`

```
#!/bin/bash
#SBATCH -J pi
#SBATCH -o pi.txt
#SBATCH -e pi.err
#SBATCH -p cpu
#SBATCH -A edu17
#SBATCH --reservation=NCC
#SBATCH -t 00:02:00
#SBATCH -n 80
#SBATCH -N 2

### Add these two lines
srun ./pi > pi-out.txt
cat pi-out.txt | awk '{sum+=$6}; END {printf "%8.6f\n", sum/NR}'
```

- All Slurm options — that you so far used after `srun` — are now included in the lines starting with `#SBATCH`

Cluster Computing Introductory Example

- Copy from `/onyx/data/edu17/02/pi.sh`

```
#!/bin/bash
#SBATCH -J pi
#SBATCH -o pi.txt
#SBATCH -e pi.err
#SBATCH -p cpu
#SBATCH -A edu17
#SBATCH --reservation=NCC
#SBATCH -t 00:02:00
#SBATCH -n 80
#SBATCH -N 2

### Add these two lines
srun ./pi > pi-out.txt
cat pi-out.txt | awk '{sum+=$6}; END {printf "%8.6f\n", sum/NR}'
```

- All Slurm options — that you so far used after `srun` — are now included in the lines starting with `#SBATCH`
- Thus `srun` is now run without options

Cluster Computing Introductory Example

- Copy from `/onyx/data/edu17/02/pi.sh`

```
#!/bin/bash
#SBATCH -J pi
#SBATCH -o pi.txt
#SBATCH -e pi.err
#SBATCH -p cpu
#SBATCH -A edu17
#SBATCH --reservation=NCC
#SBATCH -t 00:02:00
#SBATCH -n 80
#SBATCH -N 2

### Add these two lines
srun ./pi > pi-out.txt
cat pi-out.txt | awk '{sum+=$6}; END {printf "%8.6f\n", sum/NR}'
```

- All Slurm options — that you so far used after `srun` — are now included in the lines starting with `#SBATCH`
- Thus `srun` is now run without options
- Additional options include:
  - `-J`: sets the job name
  - `-o` and `-e`: set the files where the output and error should be redirected
  - `-t`: sets a time limit. The job will be killed if it exceeds this time (here 2 minutes)

Cluster Computing Introductory Example

- Submit the job

```
[ikoutsou@front01 02]$ sbatch pi.sh
Submitted batch job 198021
```

Cluster Computing Introductory Example

- Submit the job

```
[ikoutsou@front01 02]$ sbatch pi.sh
Submitted batch job 198021
```

- Query its status. Filter only your jobs:

```
[ikoutsou@front01 02]$ squeue -u $(whoami)
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
            198021       cpu       pi ikoutsou PD       0:00      2 (Reservation)
```

- Status: `PD`, `R`, `CG`: "Pending", "Running", "Completing"

Cluster Computing Introductory Example

- Submit the job

```
[ikoutsou@front01 02]$ sbatch pi.sh
Submitted batch job 198021
```

- Query its status. Filter only your jobs:

```
[ikoutsou@front01 02]$ squeue -u $(whoami)
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
            198021       cpu       pi ikoutsou PD       0:00      2 (Reservation)
```

- Status: `PD`, `R`, `CG`: "Pending", "Running", "Completing"
- After the program completes:
  - File `pi.err` contains any errors (hopefully empty)
  - File `pi.txt` contains the output of `awk` — what would be printed to the screen had you used `srun` like before
  - File `pi-out.txt` should also contain new values from the `srun` that was run during the script

MPI — Outline

- Overview of the Message Passing Interface (MPI)
- Basics of MPI
  - Distributed memory paradigm (as compared to shared memory)
  - Start-up and initialization
- Synchronization
- Collectives
- Point-to-point communication

The Message Passing Interface

- MPI: An Application Programmer Interface (API)
  - A *library specification*; determines functions, their names and arguments, and their functionality
- A *de facto* standard for programming *distributed memory* systems
- Current specification is version 4 (MPI-4.0), released June 9, 2021
  - For most systems you can reliably assume MPI-3.1 is in place
- Several free (open) or vendor-provided implementations, e.g.:
  - Mvapich
  - OpenMPI
  - IntelMPI

The Message Passing Interface

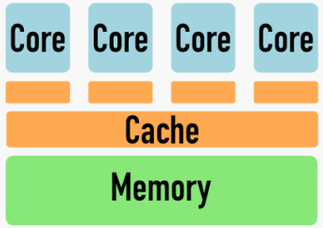
- MPI: An Application Programmer Interface (API)
  - A *library specification*; determines functions, their names and arguments, and their functionality
- A *de facto* standard for programming *distributed memory* systems
- Current specification is version 4 (MPI-4.0), released June 9, 2021
  - For most systems you can reliably assume MPI-3.1 is in place
- Several free (open) or vendor-provided implementations, e.g.:
  - Mvapich
  - OpenMPI
  - IntelMPI

Distributed memory programming

- Each process has its own memory domain
- MPI functions facilitate:
  - Obtaining environment information about the running process, e.g., process id, number of processes, etc.
  - Achieving *communication* between processes, e.g. synchronization, copying of data, etc.

Shared vs Distributed memory paradigm

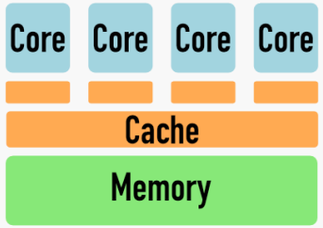
Shared memory



- Multiple processes share common memory (common *memory address space*)
- E.g. multi-core CPU, multi-socket node, GPU threads, etc.
- Programming models: OpenMP, pthreads, MPI, CUDA (sort of)

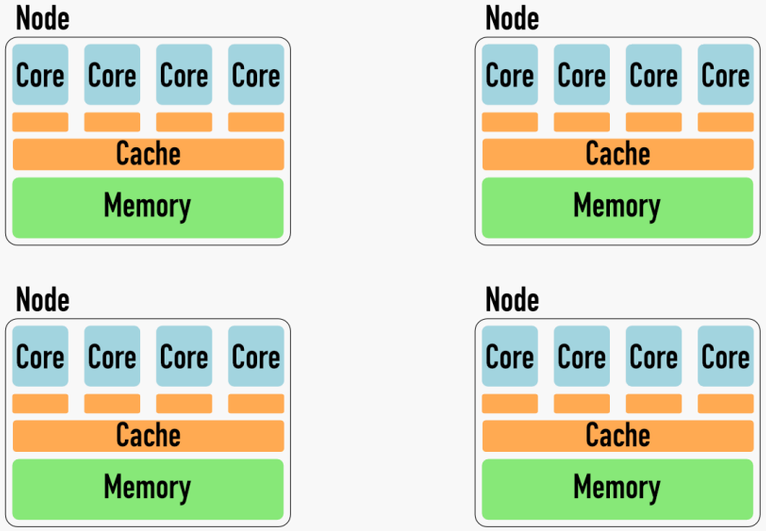
Shared vs Distributed memory paradigm

Shared memory



- Multiple processes share common memory (common *memory address space*)
- E.g. multi-core CPU, multi-socket node, GPU threads, etc.
- Programming models: OpenMP, pthreads, MPI, CUDA (sort of)

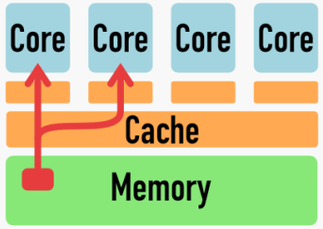
Distributed memory



- Processes have distinct memory domains (different *memory address space*)
- E.g. multiple nodes within a cluster, multiple GPUs within a node
- Programming models: **MPI**

Shared vs Distributed memory paradigm

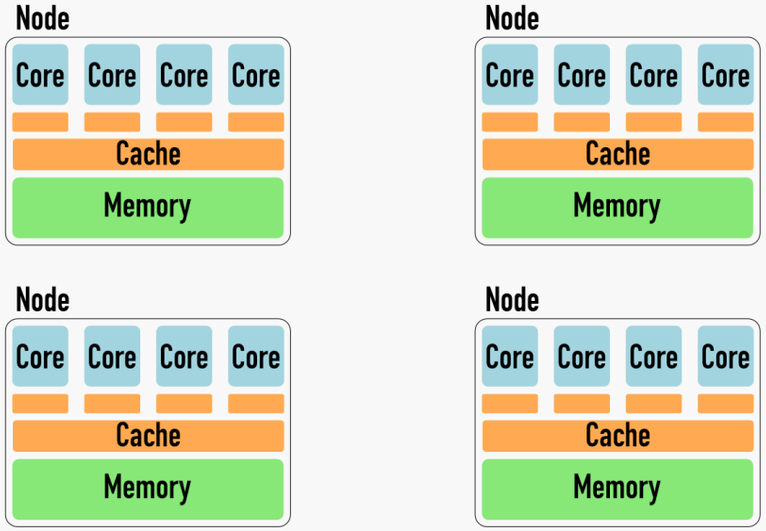
Shared memory



Data shared via memory

- Multiple processes share common memory (common *memory address space*)
- E.g. multi-core CPU, multi-socket node, GPU threads, etc.
- Programming models: OpenMP, pthreads, MPI, CUDA (sort of)

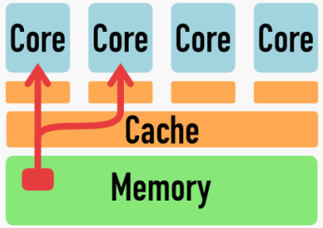
Distributed memory



- Processes have distinct memory domains (different *memory address space*)
- E.g. multiple nodes within a cluster, multiple GPUs within a node
- Programming models: **MPI**

Shared vs Distributed memory paradigm

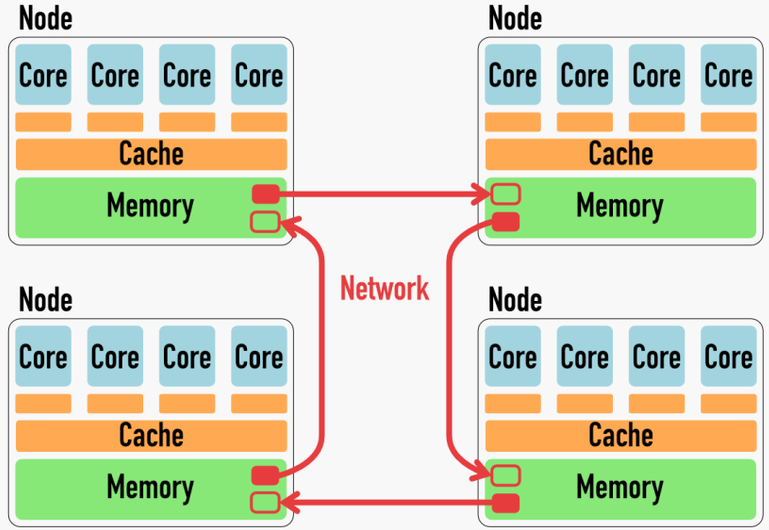
Shared memory



Data shared via memory

- Multiple processes share common memory (common *memory address space*)
- E.g. multi-core CPU, multi-socket node, GPU threads, etc.
- Programming models: OpenMP, pthreads, MPI, CUDA (sort of)

Distributed memory



Data shared via explicit communication over a network

- Processes have distinct memory domains (different *memory address space*)
- E.g. multiple nodes within a cluster, multiple GPUs within a node
- Programming models: **MPI**

Running a program in parallel

- Trivially, in Linux it is simple to run a program in parallel

```
ssh node01 ./my_program &
ssh node02 ./my_program &
ssh node03 ./my_program &
```

`my_program` will run on each node identically

Running a program in parallel

- Trivially, in Linux it is simple to run a program in parallel

```
ssh node01 ./my_program &
ssh node02 ./my_program &
ssh node03 ./my_program &
```

`my_program` will run on each node identically

- An MPI program is run in a similar way, but via a wrapper script that also initializes the parallel environment (environment variables, etc.)

```
mpirun -H node01,node02,node03 ./my_mpi_program
```

Running a program in parallel

- Trivially, in Linux it is simple to run a program in parallel

```
ssh node01 ./my_program &
ssh node02 ./my_program &
ssh node03 ./my_program &
```

`my_program` will run on each node identically

- An MPI program is run in a similar way, but via a wrapper script that also initializes the parallel environment (environment variables, etc.)

```
mpirun -H node01,node02,node03 ./my_mpi_program
```

- In practice, a scheduler is used which determines which nodes you are currently allocated, meaning you usually will not need to explicitly specify the hostnames

```
mpirun ./my_mpi_program
```

Running a program in parallel

- Trivially, in Linux it is simple to run a program in parallel

```
ssh node01 ./my_program &
ssh node02 ./my_program &
ssh node03 ./my_program &
```

  `my_program` will run on each node identically

- An MPI program is run in a similar way, but via a wrapper script that also initializes the parallel environment (environment variables, etc.)

```
mpirun -H node01,node02,node03 ./my_mpi_program
```

- In practice, a scheduler is used which determines which nodes you are currently allocated, meaning you usually will not need to explicitly specify the hostnames

```
mpirun ./my_mpi_program
```

- Depending on the system, instead of `mpirun` you may be required `mpiexec` or `srun` which take similar (but not identical) arguments

Compiling an MPI program

- An MPI program includes calls to MPI functions

Compiling an MPI program

- An MPI program includes calls to MPI functions
  - In C, we include a single header file with all function definitions, macros, and constants

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

Compiling an MPI program

- An MPI program includes calls to MPI functions
  - In C, we include a single header file with all function definitions, macros, and constants

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

  - Need to link against MPI libraries; precise invocation depends on the compiler, the MPI implementation used, its version, etc., e.g.:

```
gcc -o my_mpi_program my_mpi_program.c -I/opt/mpi/include -L/opt/mpi/lib -lmpi
```

Compiling an MPI program

- An MPI program includes calls to MPI functions
  - In C, we include a single header file with all function definitions, macros, and constants

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

  - Need to link against MPI libraries; precise invocation depends on the compiler, the MPI implementation used, its version, etc., e.g.:

    ```
    gcc -o my_mpi_program my_mpi_program.c -I/opt/mpi/include -L/opt/mpi/lib -lmpi
    ```

- Thankfully, knowing the locations of the MPI library and include files is never needed in practice; implementations come with wrappers that set the appropriate include paths and linker options:

  ```
  mpicc -o my_mpi_program my_mpi_program.c
  ```

Initialization

- MPI functions begin with the `MPI_` prefix in C

Initialization

- MPI functions begin with the `MPI_` prefix in C
- Call `MPI_Init()` first, before any other MPI call:

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

  where `argc` and `argv` are the typical names used for the command line variables passed to `main()`

Initialization

- MPI functions begin with the `MPI_` prefix in C
- Call `MPI_Init()` first, before any other MPI call:

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

  where `argc` and `argv` are the typical names used for the command line variables passed to `main()`
- Before the end of the program, call `MPI_Finalize()`, otherwise the MPI runtime may assume your program finished in error

Initialization

- MPI functions begin with the `MPI_` prefix in C
- Call `MPI_Init()` first, before any other MPI call:

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

  where `argc` and `argv` are the typical names used for the command line variables passed to `main()`
- Before the end of the program, call `MPI_Finalize()`, otherwise the MPI runtime may assume your program finished in error

```c
#include <mpi.h>

int
main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);
  /*
     ...
     ...
     ...
  */
  MPI_Finalize();
  return 0;
}
```

Initialization

- Two functions you will almost always call
  - `MPI_Comm_size()`: gives the number of parallel process running (n_{proc})
  - `MPI_Comm_rank()`: determines the *rank* of the process, i.e. a unique number between 0 and $n_{\text{proc}} - 1$ that identifies the calling process
- A complete example:

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

int
main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);
  int nproc, rank;
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf(" This is rank = %d of nproc = %d\n", rank, nproc);
  MPI_Finalize();
  return 0;
}
```

Initialization

- Two functions you will almost always call
  - `MPI_Comm_size()`: gives the number of parallel process running (n_{proc})
  - `MPI_Comm_rank()`: determines the *rank* of the process, i.e. a unique number between 0 and $n_{\text{proc}} - 1$ that identifies the calling process
- A complete example:

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

int
main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);
  int nproc, rank;
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf(" This is rank = %d of nproc = %d\n", rank, nproc);
  MPI_Finalize();
  return 0;
}
```

- `MPI_COMM_WORLD` is an *MPI communicator*. This specific communicator is the default communicator, defined in `mpi.h`, and trivially specifies *all* processes
- A user can partition processes into subgroups by defining custom communicators, but this will not be covered here

Initialization

- Two functions you will almost always call
  - `MPI_Comm_size()`: gives the number of parallel process running (n_{proc})
  - `MPI_Comm_rank()`: determines the *rank* of the process, i.e. a unique number between 0 and $n_{\text{proc}} - 1$ that identifies the calling process
- A complete example:

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

int
main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);
  int nproc, rank;
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf(" This is rank = %d of nproc = %d\n", rank, nproc);
  MPI_Finalize();
  return 0;
}
```

- No assumptions can safely be made about the order in which the `printf()` statements occur, i.e. the order in which each process prints is practically random

Synchronization

- Compiling and running the previous program (assuming it is saved as `example.c`)

```
[user@front01 ~]$ mpicc -o example example.c
[user@front01 ~]$ mpirun -n 5 example
 This is rank = 3 of nproc = 5
 This is rank = 1 of nproc = 5
 This is rank = 2 of nproc = 5
 This is rank = 4 of nproc = 5
 This is rank = 0 of nproc = 5
```

Synchronization

- Compiling and running the previous program (assuming it is saved as `example.c`)

```
[user@front01 ~]$ mpicc -o example example.c
[user@front01 ~]$ mpirun -n 5 example
 This is rank = 3 of nproc = 5
 This is rank = 1 of nproc = 5
 This is rank = 2 of nproc = 5
 This is rank = 4 of nproc = 5
 This is rank = 0 of nproc = 5
```

- Note that the order is random; Synchronization between processes can be achieved using the `MPI_Barrier()` function

Synchronization

- Compiling and running the previous program (assuming it is saved as `example.c`)

```
[user@front01 ~]$ mpicc -o example example.c
[user@front01 ~]$ mpirun -n 5 example
 This is rank = 3 of nproc = 5
 This is rank = 1 of nproc = 5
 This is rank = 2 of nproc = 5
 This is rank = 4 of nproc = 5
 This is rank = 0 of nproc = 5
```

- Note that the order is random; Synchronization between processes can be achieved using the `MPI_Barrier()` function
- All processes must call `MPI_Barrier()`

Synchronization

- Compiling and running the previous program (assuming it is saved as `example.c`)

```
[user@front01 ~]$ mpicc -o example example.c
[user@front01 ~]$ mpirun -n 5 example
 This is rank = 3 of nproc = 5
 This is rank = 1 of nproc = 5
 This is rank = 2 of nproc = 5
 This is rank = 4 of nproc = 5
 This is rank = 0 of nproc = 5
```

- Note that the order is random; Synchronization between processes can be achieved using the `MPI_Barrier()` function
- All processes must call `MPI_Barrier()`
- For any process to exit the barrier, all processes must have entered the barrier first

Synchronization

- Compiling and running the previous program (assuming it is saved as `example.c`)

```
[user@front01 ~]$ mpicc -o example example.c
[user@front01 ~]$ mpirun -n 5 example
 This is rank = 3 of nproc = 5
 This is rank = 1 of nproc = 5
 This is rank = 2 of nproc = 5
 This is rank = 4 of nproc = 5
 This is rank = 0 of nproc = 5
```

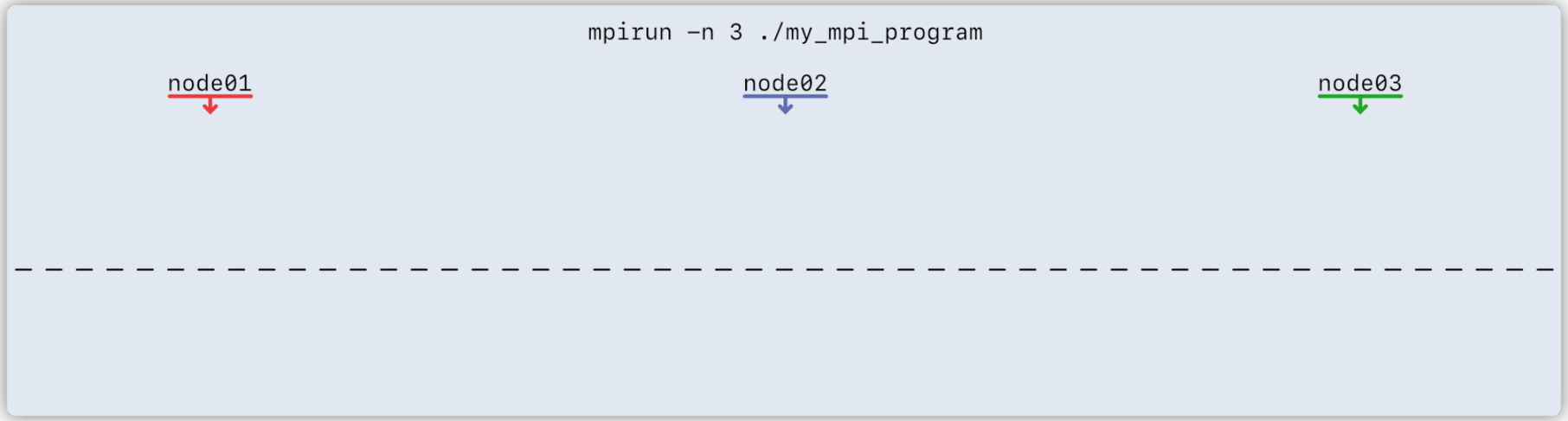
- Note that the order is random; Synchronization between processes can be achieved using the `MPI_Barrier()` function
- All processes must call `MPI_Barrier()`
- For any process to exit the barrier, all processes must have entered the barrier first

Synchronization

- Compiling and running the previous program (assuming it is saved as `example.c`)

```
[user@front01 ~]$ mpicc -o example example.c
[user@front01 ~]$ mpirun -n 5 example
 This is rank = 3 of nproc = 5
 This is rank = 1 of nproc = 5
 This is rank = 2 of nproc = 5
 This is rank = 4 of nproc = 5
 This is rank = 0 of nproc = 5
```

- Note that the order is random; Synchronization between processes can be achieved using the `MPI_Barrier()` function
- All processes must call `MPI_Barrier()`
- For any process to exit the barrier, all processes must have entered the barrier first

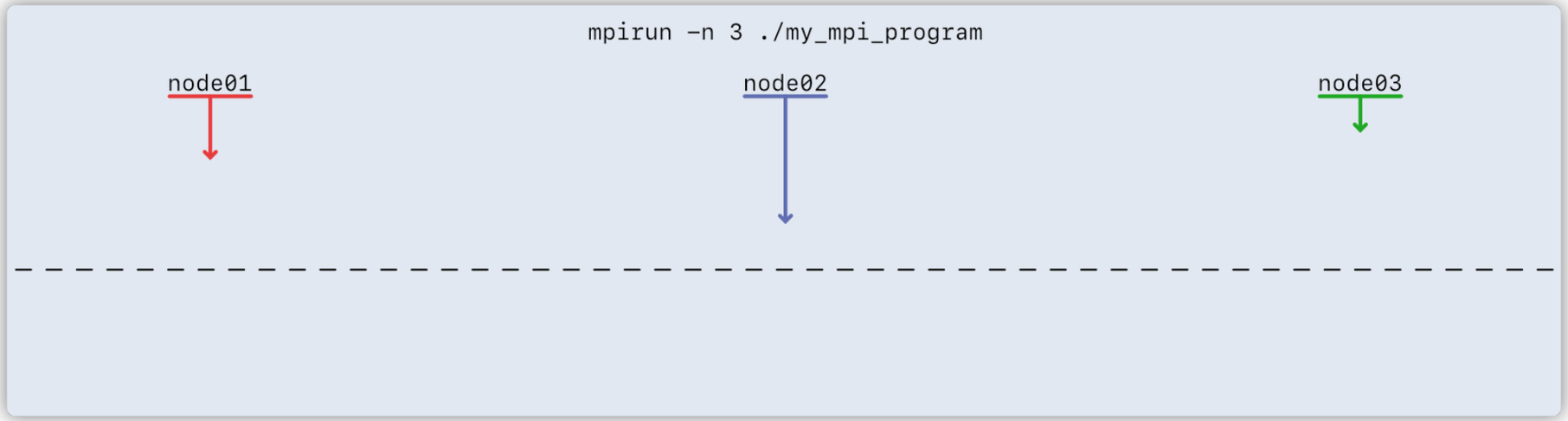


Synchronization

- Compiling and running the previous program (assuming it is saved as `example.c`)

```
[user@front01 ~]$ mpicc -o example example.c
[user@front01 ~]$ mpirun -n 5 example
 This is rank = 3 of nproc = 5
 This is rank = 1 of nproc = 5
 This is rank = 2 of nproc = 5
 This is rank = 4 of nproc = 5
 This is rank = 0 of nproc = 5
```

- Note that the order is random; Synchronization between processes can be achieved using the `MPI_Barrier()` function
- All processes must call `MPI_Barrier()`
- For any process to exit the barrier, all processes must have entered the barrier first

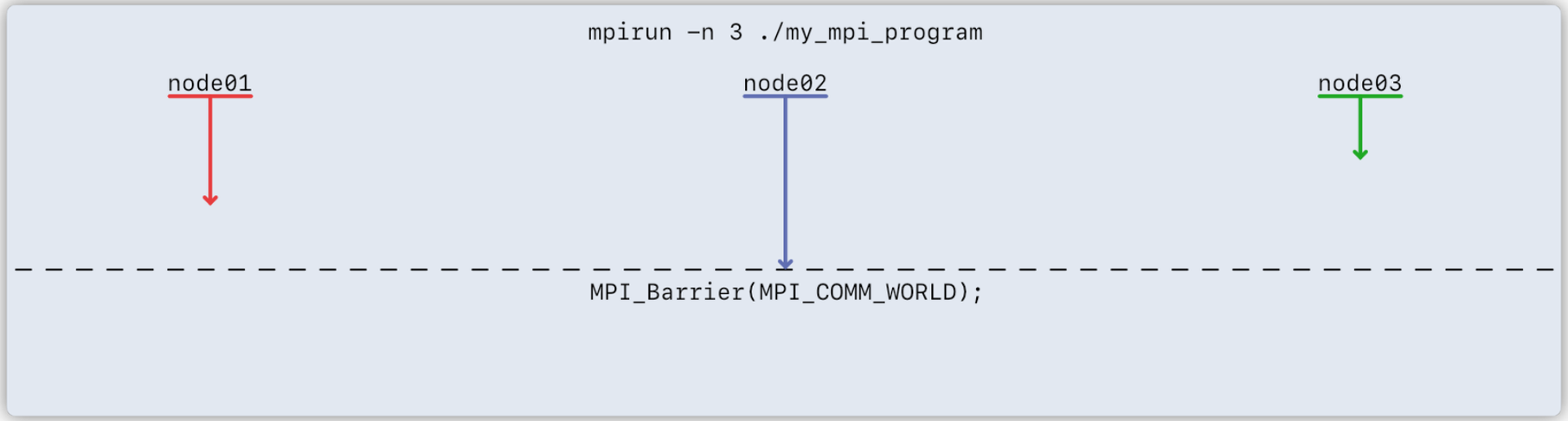


Synchronization

- Compiling and running the previous program (assuming it is saved as `example.c`)

```
[user@front01 ~]$ mpicc -o example example.c
[user@front01 ~]$ mpirun -n 5 example
 This is rank = 3 of nproc = 5
 This is rank = 1 of nproc = 5
 This is rank = 2 of nproc = 5
 This is rank = 4 of nproc = 5
 This is rank = 0 of nproc = 5
```

- Note that the order is random; Synchronization between processes can be achieved using the `MPI_Barrier()` function
- All processes must call `MPI_Barrier()`
- For any process to exit the barrier, all processes must have entered the barrier first

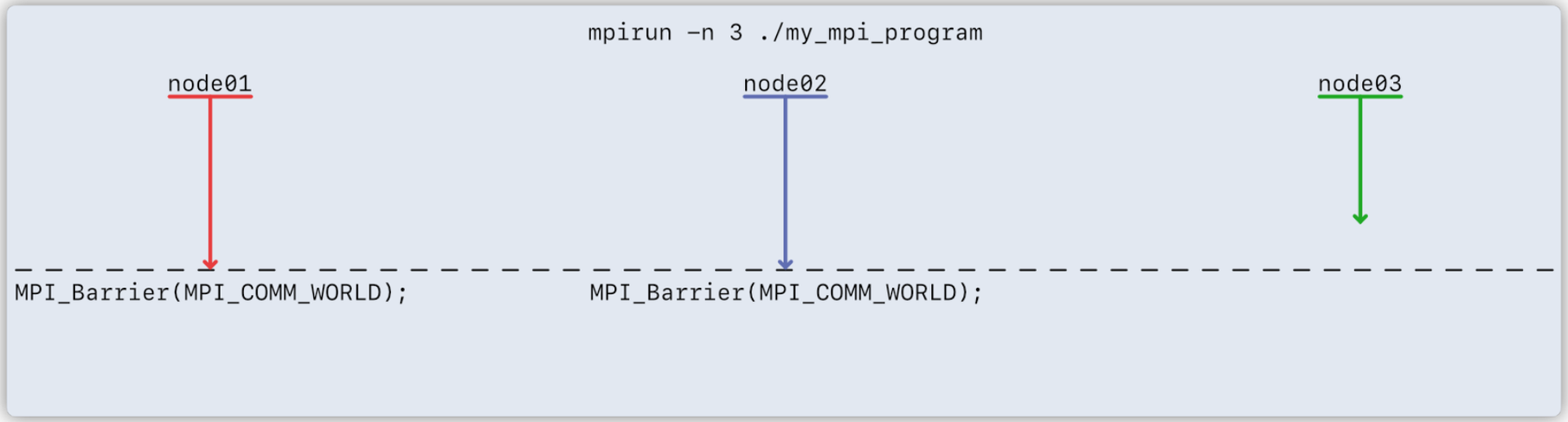


Synchronization

- Compiling and running the previous program (assuming it is saved as `example.c`)

```
[user@front01 ~]$ mpicc -o example example.c
[user@front01 ~]$ mpirun -n 5 example
 This is rank = 3 of nproc = 5
 This is rank = 1 of nproc = 5
 This is rank = 2 of nproc = 5
 This is rank = 4 of nproc = 5
 This is rank = 0 of nproc = 5
```

- Note that the order is random; Synchronization between processes can be achieved using the `MPI_Barrier()` function
- All processes must call `MPI_Barrier()`
- For any process to exit the barrier, all processes must have entered the barrier first

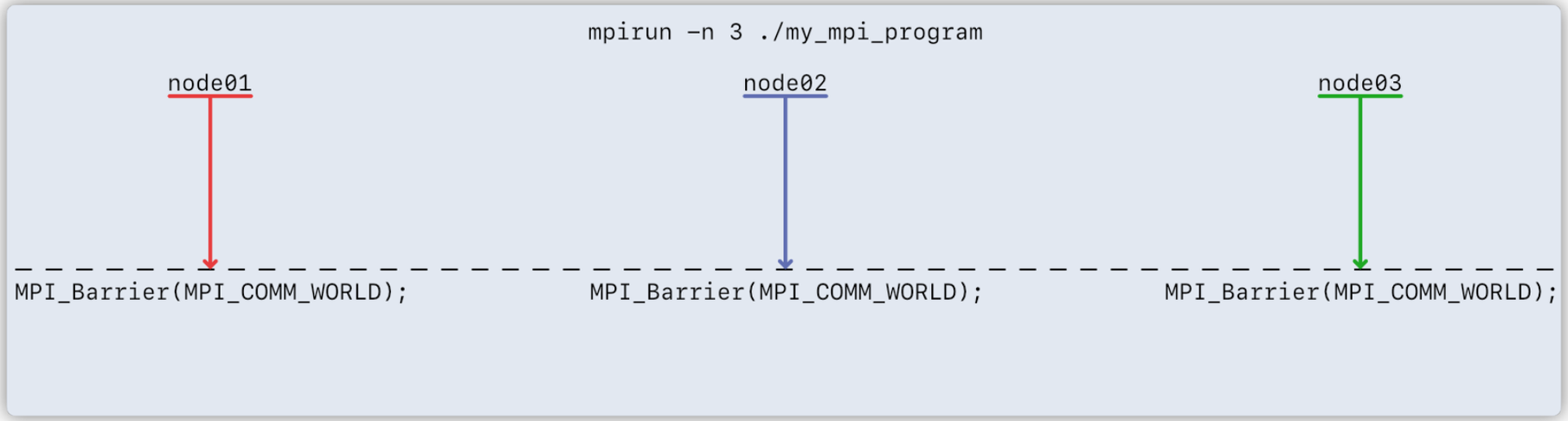


Synchronization

- Compiling and running the previous program (assuming it is saved as `example.c`)

```
[user@front01 ~]$ mpicc -o example example.c
[user@front01 ~]$ mpirun -n 5 example
 This is rank = 3 of nproc = 5
 This is rank = 1 of nproc = 5
 This is rank = 2 of nproc = 5
 This is rank = 4 of nproc = 5
 This is rank = 0 of nproc = 5
```

- Note that the order is random; Synchronization between processes can be achieved using the `MPI_Barrier()` function
- All processes must call `MPI_Barrier()`
- For any process to exit the barrier, all processes must have entered the barrier first

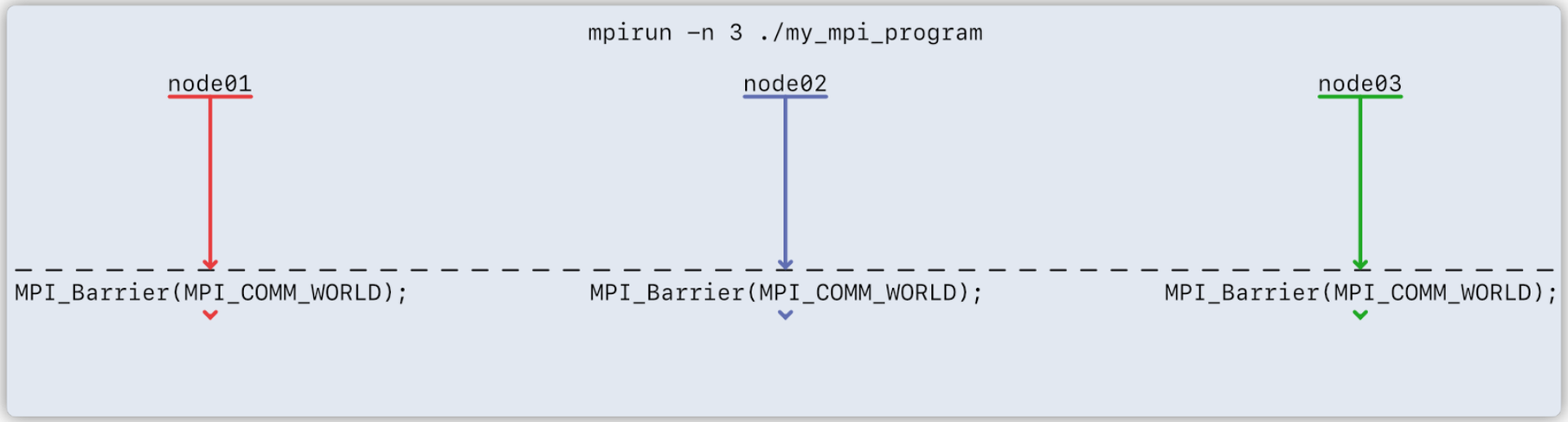


Synchronization

- Compiling and running the previous program (assuming it is saved as `example.c`)

```
[user@front01 ~]$ mpicc -o example example.c
[user@front01 ~]$ mpirun -n 5 example
 This is rank = 3 of nproc = 5
 This is rank = 1 of nproc = 5
 This is rank = 2 of nproc = 5
 This is rank = 4 of nproc = 5
 This is rank = 0 of nproc = 5
```

- Note that the order is random; Synchronization between processes can be achieved using the `MPI_Barrier()` function
- All processes must call `MPI_Barrier()`
- For any process to exit the barrier, all processes must have entered the barrier first

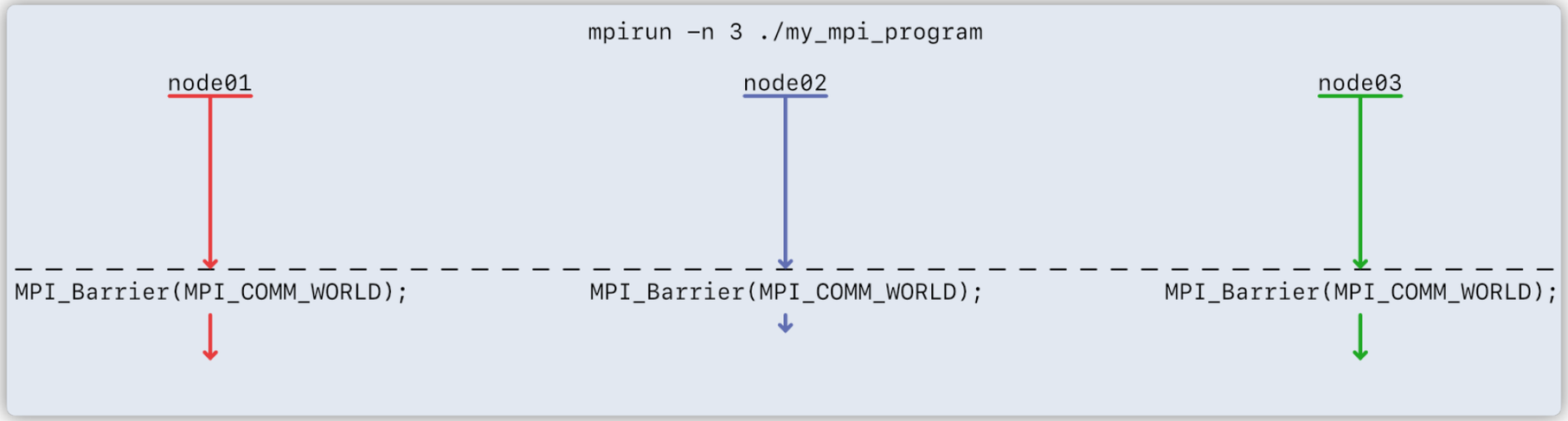


Synchronization

- Compiling and running the previous program (assuming it is saved as `example.c`)

```
[user@front01 ~]$ mpicc -o example example.c
[user@front01 ~]$ mpirun -n 5 example
 This is rank = 3 of nproc = 5
 This is rank = 1 of nproc = 5
 This is rank = 2 of nproc = 5
 This is rank = 4 of nproc = 5
 This is rank = 0 of nproc = 5
```

- Note that the order is random; Synchronization between processes can be achieved using the `MPI_Barrier()` function
- All processes must call `MPI_Barrier()`
- For any process to exit the barrier, all processes must have entered the barrier first

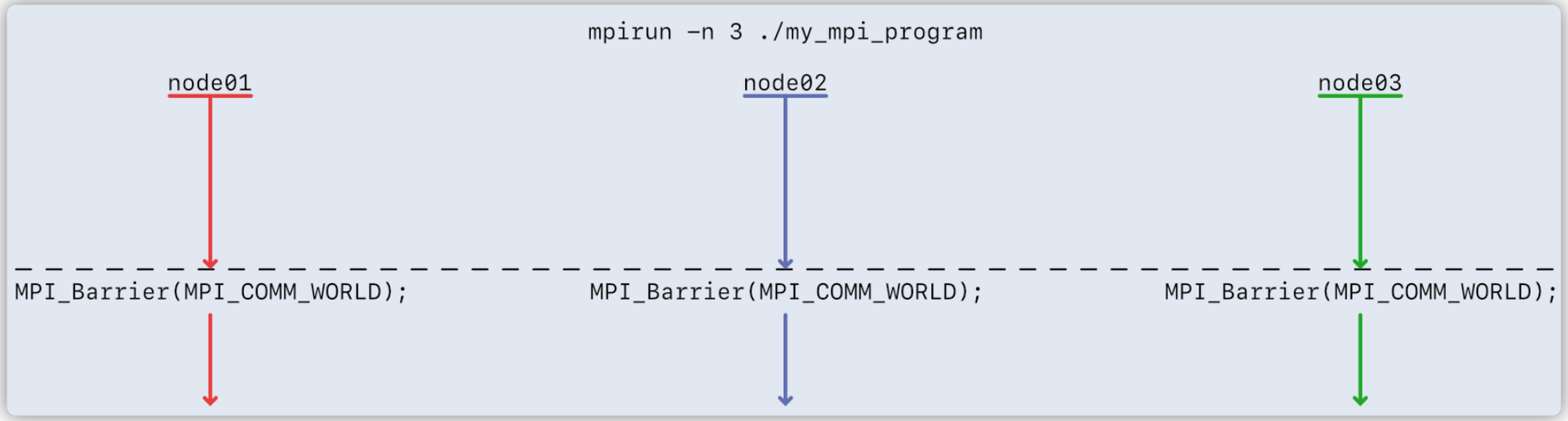


Synchronization

- Compiling and running the previous program (assuming it is saved as `example.c`)

```
[user@front01 ~]$ mpicc -o example example.c
[user@front01 ~]$ mpirun -n 5 example
 This is rank = 3 of nproc = 5
 This is rank = 1 of nproc = 5
 This is rank = 2 of nproc = 5
 This is rank = 4 of nproc = 5
 This is rank = 0 of nproc = 5
```

- Note that the order is random; Synchronization between processes can be achieved using the `MPI_Barrier()` function
- All processes must call `MPI_Barrier()`
- For any process to exit the barrier, all processes must have entered the barrier first

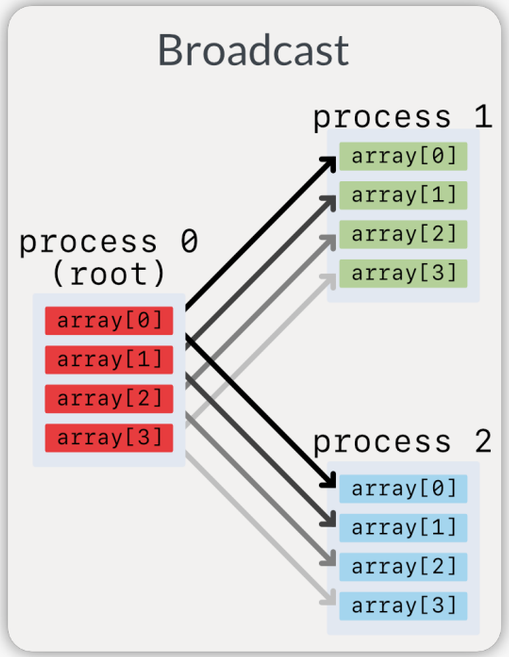


Collective operations

- The first set of communication functions we will look at are *collective operations*
- Collective: all processes must be involved in the operation (as opposed to *point-to-point* communications)
- Examples (this list is not exhaustive!):
    - Broadcast a variable from one process to all processes (Broadcast)
    - Distribute elements of an array on one process to multiple processes (Scatter)
    - Collect elements of arrays scattered over processes into a single process (Gather)
    - Sum a variable over all processes (Reduction)

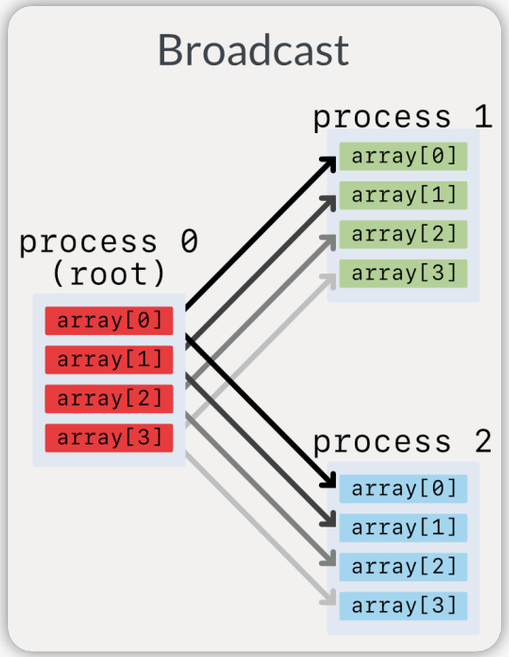
Collective operations

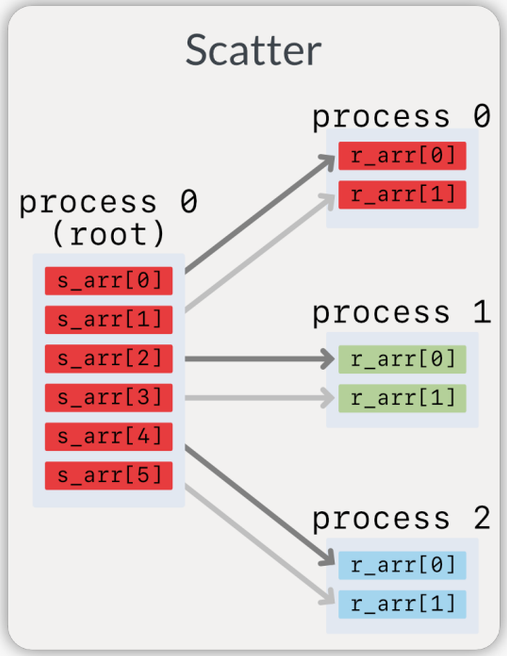
- The first set of communication functions we will look at are *collective operations*
- Collective: all processes must be involved in the operation (as opposed to *point-to-point* communications)
- Examples (this list is not exhaustive!):
  - Broadcast a variable from one process to all processes (Broadcast)
  - Distribute elements of an array on one process to multiple processes (Scatter)
  - Collect elements of arrays scattered over processes into a single process (Gather)
  - Sum a variable over all processes (Reduction)



Collective operations

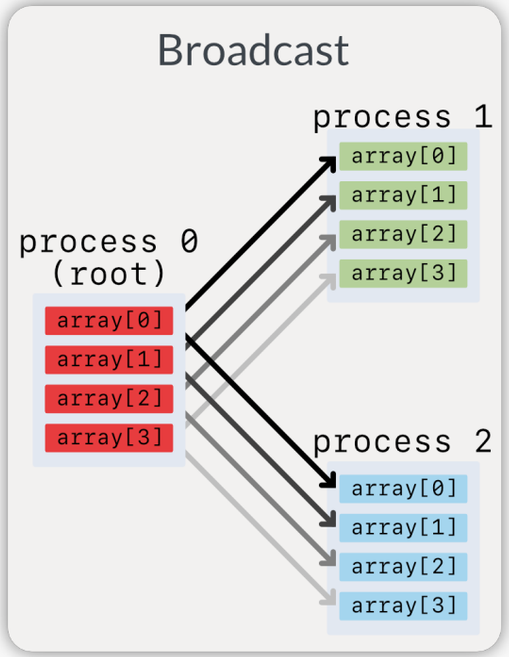
- The first set of communication functions we will look at are *collective operations*
- Collective: all processes must be involved in the operation (as opposed to *point-to-point* communications)
- Examples (this list is not exhaustive!):
  - Broadcast a variable from one process to all processes (Broadcast)
  - Distribute elements of an array on one process to multiple processes (Scatter)
  - Collect elements of arrays scattered over processes into a single process (Gather)
  - Sum a variable over all processes (Reduction)

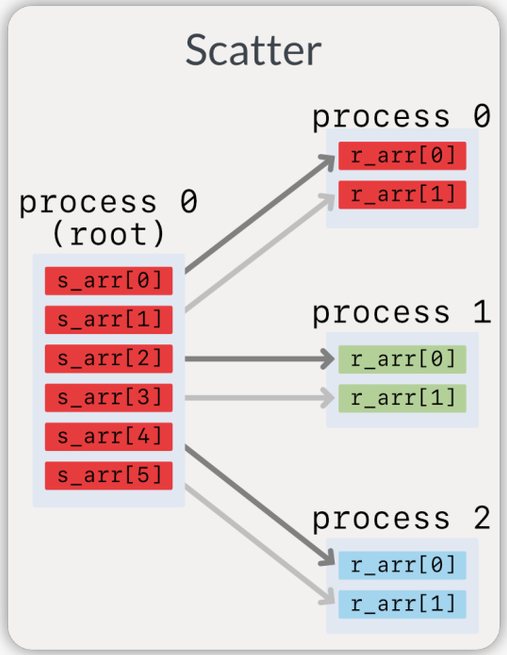


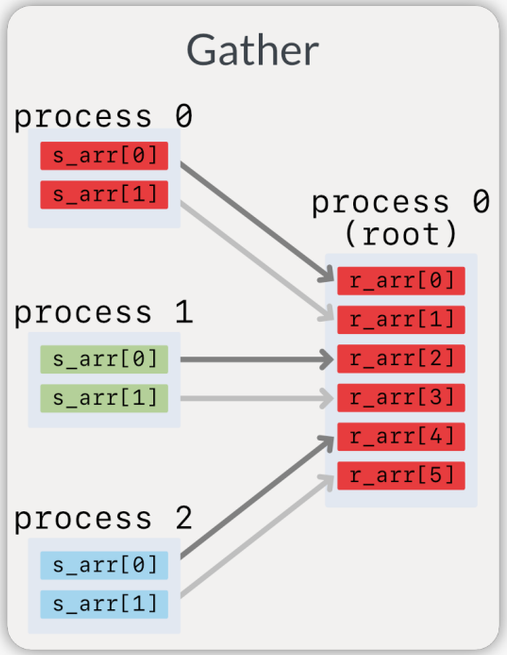


Collective operations

- The first set of communication functions we will look at are *collective operations*
- Collective: all processes must be involved in the operation (as opposed to *point-to-point* communications)
- Examples (this list is not exhaustive!):
  - Broadcast a variable from one process to all processes (Broadcast)
  - Distribute elements of an array on one process to multiple processes (Scatter)
  - Collect elements of arrays scattered over processes into a single process (Gather)
  - Sum a variable over all processes (Reduction)







Collective operations: Broadcast

- Broadcast:

```
MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```

Collective operations: Broadcast

- Broadcast:

```
MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```

  - *Example*: Broadcast from rank 0 (root), the four-element, double precision array `arr[]`

```
MPI_Bcast(arr, 4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Collective operations: Broadcast

- Broadcast:

```
MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```

  - *Example*: Broadcast from rank 0 (root), the four-element, double precision array `arr[]`

```
MPI_Bcast(arr, 4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

  - *Example*: Broadcast from rank 0 (root), the scalar integer variable `var`

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

Collective operations: Broadcast

- Broadcast:

```
MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```

  - *Example*: Broadcast from rank 0 (root), the four-element, double precision array `arr[]`

```
MPI_Bcast(arr, 4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

  - *Example*: Broadcast from rank 0 (root), the scalar integer variable `var`

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

- The `MPI_Datatype` is important since MPI uses it to estimate the size in bytes that need to be transfered

Collective operations: Broadcast

- Broadcast:

```
MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```

  - *Example*: Broadcast from rank 0 (root), the four-element, double precision array `arr[]`

```
MPI_Bcast(arr, 4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

  - *Example*: Broadcast from rank 0 (root), the scalar integer variable `var`

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

- The `MPI_Datatype` is important since MPI uses it to estimate the size in bytes that need to be transfered
- Full list of types available in MPI documentation. E.g. see: https://www.mpich.org/static/docs/latest/www3/Constants.html

Collective operations: Scatter

- Scatter:

```
MPI_Scatter(
          const void *sendbuf, int sendcount, MPI_Datatype sendtype,
          void *recvbuf, int recvcount, MPI_Datatype recvtype,
          int root, MPI_Comm comm
          );
```

Collective operations: Scatter

- Scatter:

```
MPI_Scatter(
          const void *sendbuf, int sendcount, MPI_Datatype sendtype,
          void *recvbuf, int recvcount, MPI_Datatype recvtype,
          int root, MPI_Comm comm
          );
```

  - `sendcount` is the number of elements to be sent to *each* process

Collective operations: Scatter

- Scatter:

```
MPI_Scatter(
          const void *sendbuf, int sendcount, MPI_Datatype sendtype,
          void *recvbuf, int recvcount, MPI_Datatype recvtype,
          int root, MPI_Comm comm
          );
```

  - `sendcount` is the number of elements to be sent to *each* process
  - `sendbuf` is only relevant in the root process

Collective operations: Scatter

- Scatter:

```
MPI_Scatter(
          const void *sendbuf, int sendcount, MPI_Datatype sendtype,
          void *recvbuf, int recvcount, MPI_Datatype recvtype,
          int root, MPI_Comm comm
          );
```

  - `sendcount` is the number of elements to be sent to *each* process
  - `sendbuf` is only relevant in the root process
  - *Example*: distribute a 12-element array from process 0, assuming 3 processes in total (including root)

```
double arr_all[12]; /* ⟵ this only needs to be defined on process with rank == 0 */
double arr_proc[4];
MPI_Scatter(arr_all, 4, MPI_DOUBLE, arr_proc, 4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Collective operations: Scatter

- Scatter:

```
MPI_Scatter(
          const void *sendbuf, int sendcount, MPI_Datatype sendtype,
          void *recvbuf, int recvcount, MPI_Datatype recvtype,
          int root, MPI_Comm comm
          );
```

  - `sendcount` is the number of elements to be sent to *each* process
  - `sendbuf` is only relevant in the root process
  - *Example*: distribute a 12-element array from process 0, assuming 3 processes in total (including root)

```
double arr_all[12]; /* ⟵ this only needs to be defined on process with rank == 0 */
double arr_proc[4];
MPI_Scatter(arr_all, 4, MPI_DOUBLE, arr_proc, 4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

  - *Example*: distribute each element of a 4-element array to 4 processes in total (including root)

```
double arr[4]; /* ⟵ this only needs to be defined on process with rank == 0 */
double element;
MPI_Scatter(arr, 1, MPI_DOUBLE, &element, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Collective operations: Gather

- Gather:

```
MPI_Gather(
        const void *sendbuf, int sendcount, MPI_Datatype sendtype,
        void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
        MPI_Comm comm
        )
```

Collective operations: Gather

- Gather:

```
MPI_Gather(
        const void *sendbuf, int sendcount, MPI_Datatype sendtype,
        void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
        MPI_Comm comm
        )
```

  - `recvcount` is the number of elements to be received by *each* process

Collective operations: Gather

- Gather:

```
MPI_Gather(
        const void *sendbuf, int sendcount, MPI_Datatype sendtype,
        void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
        MPI_Comm comm
        )
```

  - `recvcount` is the number of elements to be received by *each* process
  - `recvbuf` is only relevant in the root process

Collective operations: Gather

- Gather:

```
MPI_Gather(
        const void *sendbuf, int sendcount, MPI_Datatype sendtype,
        void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
        MPI_Comm comm
        )
```

  - `recvcount` is the number of elements to be received by *each* process
  - `recvbuf` is only relevant in the root process
  - *Example*: collect a 9-element array at process 0, by concatenating 3 elements from each of 3 processes in total (including root)

```
double arr_all[9]; /* ⟵ this only needs to be defined on process with rank == 0 */
double arr_proc[3];
MPI_Gather(arr_proc, 3, MPI_DOUBLE, arr_all, 3, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Collective operations: Gather

- Gather:

```
MPI_Gather(
        const void *sendbuf, int sendcount, MPI_Datatype sendtype,
        void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
        MPI_Comm comm
        )
```

  - `recvcount` is the number of elements to be received by *each* process
  - `recvbuf` is only relevant in the root process
  - *Example*: collect a 9-element array at process 0, by concatenating 3 elements from each of 3 processes in total (including root)

```
double arr_all[9]; /* ⟵ this only needs to be defined on process with rank == 0 */
double arr_proc[3];
MPI_Gather(arr_proc, 3, MPI_DOUBLE, arr_all, 3, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

  - *Example*: collect a 4-element array at process 0, by concatenating an element from each of 4 processes in total (including root)

```
double arr[4]; /* ⟵ this only needs to be defined on process with rank == 0 */
double element;
MPI_Gather(&element, 1, MPI_DOUBLE, arr, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Collective operations: Reduction

- Reduction:

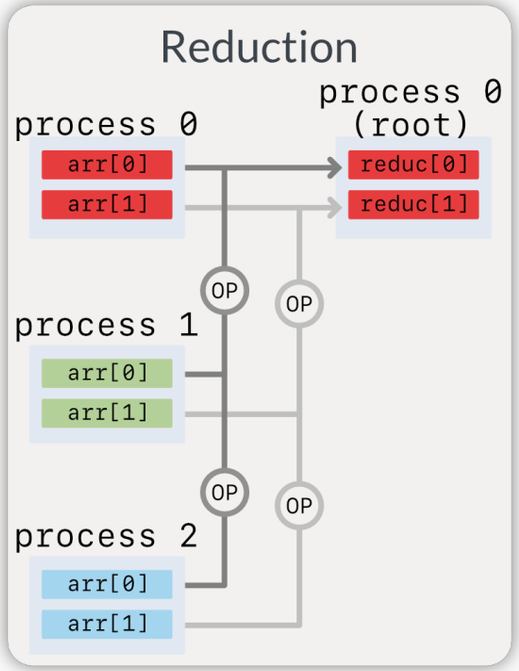
```
MPI_Reduce(
        const void *sendbuf, void *recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, int root,
        MPI_Comm comm
        )
```

Collective operations: Reduction

- Reduction:

```
MPI_Reduce(
        const void *sendbuf, void *recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, int root,
        MPI_Comm comm
        )
```

- Notes:
  - `MPI_Op` is an operation, e.g. `MPI_SUM`, `MPI_PROD`, etc.
  - The correct result of the operation depends on specifying the datatype correctly
  - `count` is the number of elements of the arrays and is the same for send and receive, e.g. in the example on the right, `count == 2`
  - The operation is over all processes in `comm`



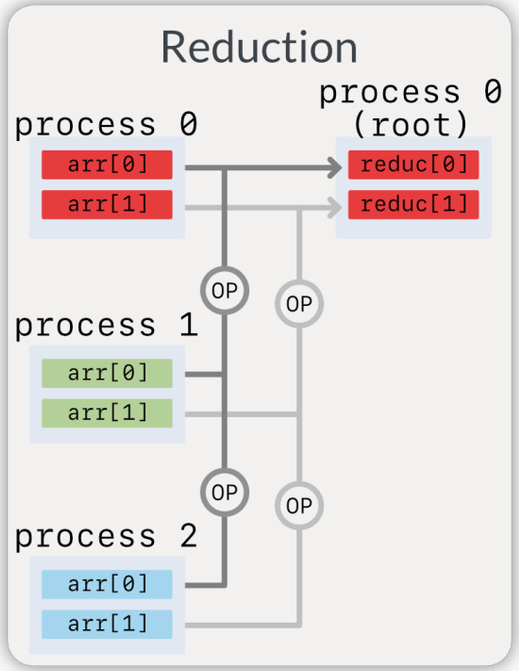
Collective operations: Reduction

- Reduction:

```
MPI_Reduce(
        const void *sendbuf, void *recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, int root,
        MPI_Comm comm
        )
```

- *Example*: Sum each element of a 3-element array over all processes

```
double s_arr[3];
double r_arr[3]; /* ⟵ only needs to      *
                  *     be defined on root */
MPI_Reduce(s_arr, r_arr, 3, MPI_DOUBLE,
           MPI_SUM, 0, MPI_COMM_WORLD);
```



Collective operations: Reduction

- Reduction:

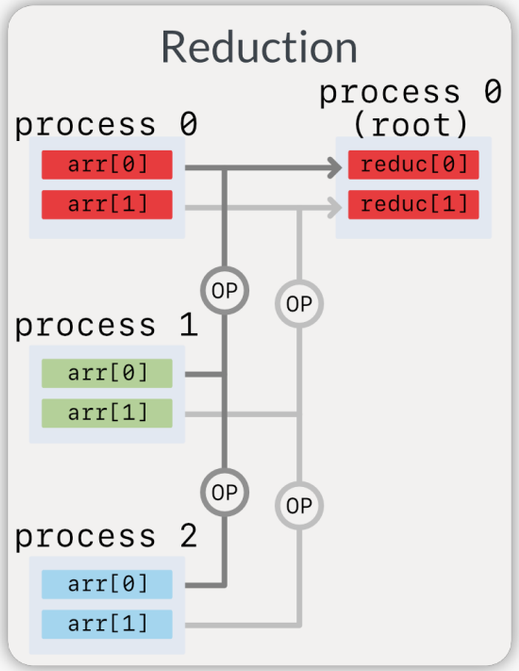
```
MPI_Reduce(
        const void *sendbuf, void *recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, int root,
        MPI_Comm comm
        )
```

- *Example*: Sum each element of a 3-element array over all processes

```
double s_arr[3];
double r_arr[3]; /* ⟵ only needs to      *
                  *     be defined on root */
MPI_Reduce(s_arr, r_arr, 3, MPI_DOUBLE,
           MPI_SUM, 0, MPI_COMM_WORLD);
```

- *Example*: Sum variable `var` over all processes

```
double var;
double sum; /* ⟵ only needs to      *
             *     be defined on root */
MPI_Reduce(&var, &sum, 1, MPI_DOUBLE,
           MPI_SUM, 0, MPI_COMM_WORLD);
```



Collective operations

Some additional notes on variants of the collectives we have covered

- `MPI_Scatterv()` and `MPI_Gatherv()`
  - Allow specifying *varying* number of elements to be distributed or collected to or from the process pool
  - Need specifying additional arguments containing offsets of the send or receive buffer

Collective operations

Some additional notes on variants of the collectives we have covered

- `MPI_Scatterv()` and `MPI_Gatherv()`
  - Allow specifying *varying* number of elements to be distributed or collected to or from the process pool
  - Need specifying additional arguments containing offsets of the send or receive buffer
- `MPI_Allreduce()`
  - Same as `MPI_Reduce()`, but result is placed on all processes in the pool
  - Result is equivalent to `MPI_Reduce()` followed by an `MPI_Bcast()`

Collective operations

Some additional notes on variants of the collectives we have covered

- `MPI_Scatterv()` and `MPI_Gatherv()`
  - Allow specifying *varying* number of elements to be distributed or collected to or from the process pool
  - Need specifying additional arguments containing offsets of the send or receive buffer
- `MPI_Allreduce()`
  - Same as `MPI_Reduce()`, but result is placed on all processes in the pool
  - Result is equivalent to `MPI_Reduce()` followed by an `MPI_Bcast()`
- In-place operations
  - For some functions, can replace the send or receive buffer with `MPI_IN_PLACE`
    $\longrightarrow$ which buffer depends on the specific MPI function

Collective operations

Some additional notes on variants of the collectives we have covered

- `MPI_Scatterv()` and `MPI_Gatherv()`
  - Allow specifying *varying* number of elements to be distributed or collected to or from the process pool
  - Need specifying additional arguments containing offsets of the send or receive buffer
- `MPI_Allreduce()`
  - Same as `MPI_Reduce()`, but result is placed on all processes in the pool
  - Result is equivalent to `MPI_Reduce()` followed by an `MPI_Bcast()`
- In-place operations
  - For some functions, can replace the send or receive buffer with `MPI_IN_PLACE`
    $\longrightarrow$ which buffer depends on the specific MPI function
  - Instructs MPI to use the **same** buffer for receive and send

Collective operations

Some additional notes on variants of the collectives we have covered

- `MPI_Scatterv()` and `MPI_Gatherv()`
  - Allow specifying *varying* number of elements to be distributed or collected to or from the process pool
  - Need specifying additional arguments containing offsets of the send or receive buffer
- `MPI_Allreduce()`
  - Same as `MPI_Reduce()`, but result is placed on all processes in the pool
  - Result is equivalent to `MPI_Reduce()` followed by an `MPI_Bcast()`
- In-place operations
  - For some functions, can replace the send or receive buffer with `MPI_IN_PLACE`
    $\longrightarrow$ which buffer depends on the specific MPI function
  - Instructs MPI to use the **same** buffer for receive and send
  - E.g. below, the sum will be placed in `var` of the root process (process with `rank == 0`):

```
if(rank != 0) {
    MPI_Reduce(&var, null, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
} else {
    MPI_Reduce(MPI_IN_PLACE, &var, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
}
```

Point-to-point communication

- Communications that involve transfer of data between two processes

Point-to-point communication

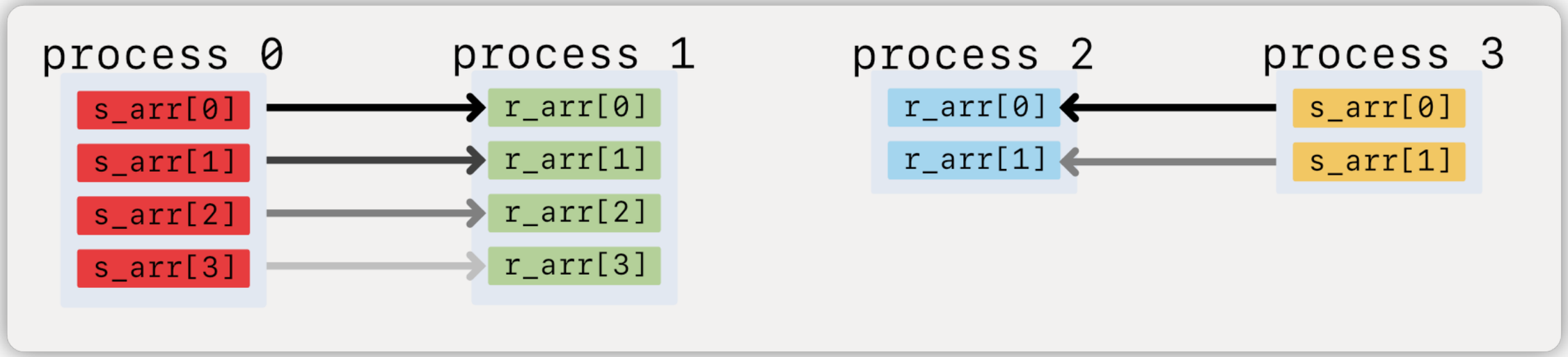
- Communications that involve transfer of data between two processes
- Most common case: send/receive
    - The sender process issues a send operation
    - The receiver process posts a receive operation

Point-to-point communication

- Communications that involve transfer of data between two processes
- Most common case: send/receive
  - The sender process issues a send operation
  - The receiver process posts a receive operation
- Asynchronous in nature: caution needed for preventing *deadlocks*, e.g.
  - Sending to a process which has not posted a matching receive
  - Posting a receive which does not have a matching send

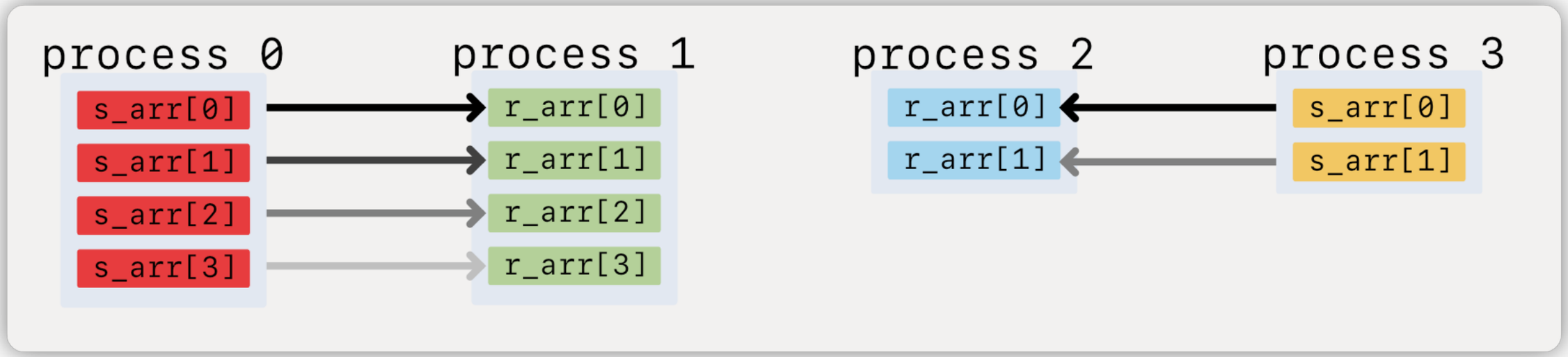
Point-to-point communication

- Communications that involve transfer of data between two processes
- Most common case: send/receive
  - The sender process issues a send operation
  - The receiver process posts a receive operation
- Asynchronous in nature: caution needed for preventing *deadlocks*, e.g.
  - Sending to a process which has not posted a matching receive
  - Posting a receive which does not have a matching send



Point-to-point communication

- Communications that involve transfer of data between two processes
- Most common case: send/receive
  - The sender process issues a send operation
  - The receiver process posts a receive operation
- Asynchronous in nature: caution needed for preventing *deadlocks*, e.g.
  - Sending to a process which has not posted a matching receive
  - Posting a receive which does not have a matching send



Two point-to-point communications are depicted above
↳ between i) process 0 and 1 and between ii) process 2 and 3

Point-to-point communication

- Send/Receive

```
MPI_Send(void *buf, int n, MPI_Datatype type, int dest, int tag, MPI_Comm comm)
MPI_Recv(void *buf, int n, MPI_Datatype type, int srce, int tag, MPI_Comm comm, MPI_Status *status)
```

Point-to-point communication

- Send/Receive

```
MPI_Send(void *buf, int n, MPI_Datatype type, int dest, int tag, MPI_Comm comm)
MPI_Recv(void *buf, int n, MPI_Datatype type, int srce, int tag, MPI_Comm comm, MPI_Status *status)
```

- Note the need to specify a source and destination rank (`srce` and `dest`)
- `n` in `MPI_Recv` specifies the max number of elements that can be received

Point-to-point communication

- Send/Receive

```
MPI_Send(void *buf, int n, MPI_Datatype type, int dest, int tag, MPI_Comm comm)
MPI_Recv(void *buf, int n, MPI_Datatype type, int srce, int tag, MPI_Comm comm, MPI_Status *status)
```

- Note the need to specify a source and destination rank (`srce` and `dest`)
- `n` in `MPI_Recv` specifies the max number of elements that can be received
- The `tag` variables tags the message. In the receiving process, it must match what the sender specified

Point-to-point communication

- Send/Receive

```
MPI_Send(void *buf, int n, MPI_Datatype type, int dest, int tag, MPI_Comm comm)
MPI_Recv(void *buf, int n, MPI_Datatype type, int srce, int tag, MPI_Comm comm, MPI_Status *status)
```

- Note the need to specify a source and destination rank (`srce` and `dest`)
- `n` in `MPI_Recv` specifies the max number of elements that can be received
- The `tag` variables tags the message. In the receiving process, it must match what the sender specified

  $\longrightarrow$ Use of `MPI_ANY_TAG` in place of `tag` in `MPI_Recv()` means "accept messages with any value for `tag`"

Point-to-point communication

- Send/Receive

```
MPI_Send(void *buf, int n, MPI_Datatype type, int dest, int tag, MPI_Comm comm)
MPI_Recv(void *buf, int n, MPI_Datatype type, int srce, int tag, MPI_Comm comm, MPI_Status *status)
```

- Note the need to specify a source and destination rank (`srce` and `dest`)
- `n` in `MPI_Recv` specifies the max number of elements that can be received
- The `tag` variables tags the message. In the receiving process, it must match what the sender specified

  $\longrightarrow$ Use of `MPI_ANY_TAG` in place of `tag` in `MPI_Recv()` means "accept messages with any value for `tag`"
- Use of `MPI_ANY_SOURCE` in `MPI_Recv()` means "accept data from any source"

Point-to-point communication

- Send/Receive

```
MPI_Send(void *buf, int n, MPI_Datatype type, int dest, int tag, MPI_Comm comm)
MPI_Recv(void *buf, int n, MPI_Datatype type, int srce, int tag, MPI_Comm comm, MPI_Status *status)
```

- Note the need to specify a source and destination rank (`srce` and `dest`)
- `n` in `MPI_Recv` specifies the max number of elements that can be received
- The `tag` variables tags the message. In the receiving process, it must match what the sender specified

  $\longrightarrow$ Use of `MPI_ANY_TAG` in place of `tag` in `MPI_Recv()` means "accept messages with any value for `tag`"
- Use of `MPI_ANY_SOURCE` in `MPI_Recv()` means "accept data from any source"
- `status` can be used to query the result of the receive (e.g. how many elements were received). We will use `MPI_STATUS_IGNORE` in place of `status`, which ignores the status

Point-to-point communication

- Send/Receive; a trivial example

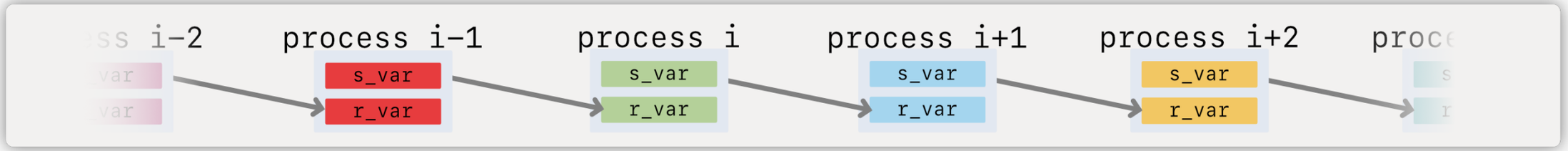
Point-to-point communication

- Send/Receive; a trivial example


```
if(rank == i) {
  MPI_Send(s_arr, 4, MPI_DOUBLE, j, 0, MPI_COMM_WORLD);
}
if(rank == j) {
  MPI_Recv(r_arr, 4, MPI_DOUBLE, i, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

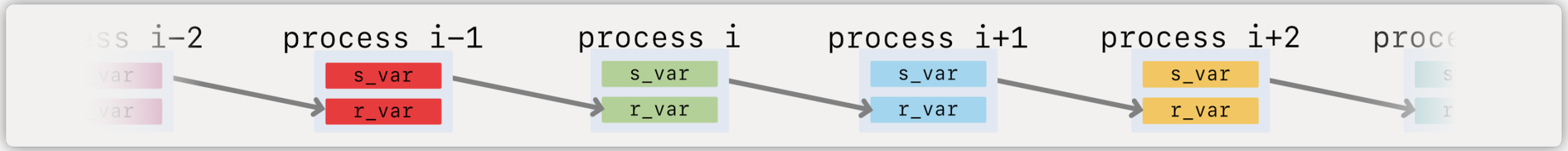
Point-to-point communication

- It is common in parallel applications to require that every process communicates with another process, e.g. a neighboring process



Point-to-point communication

- It is common in parallel applications to require that every process communicates with another process, e.g. a neighboring process

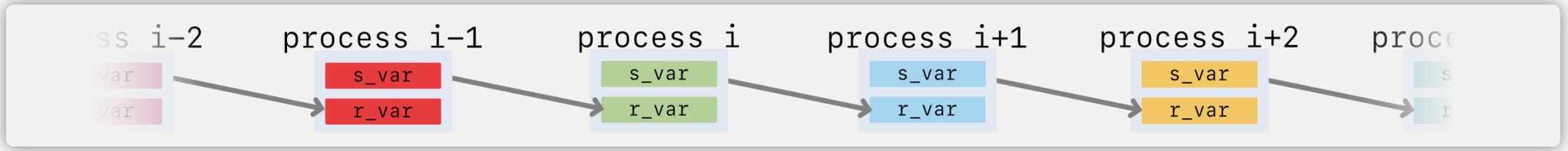


- This will not work:

```
MPI_Send(&s_var, 1, MPI_DOUBLE, rank+1, 0, MPI_COMM_WORLD);
MPI_Recv(&r_var, 1, MPI_DOUBLE, rank-1, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Point-to-point communication

- It is common in parallel applications to require that every process communicates with another process, e.g. a neighboring process



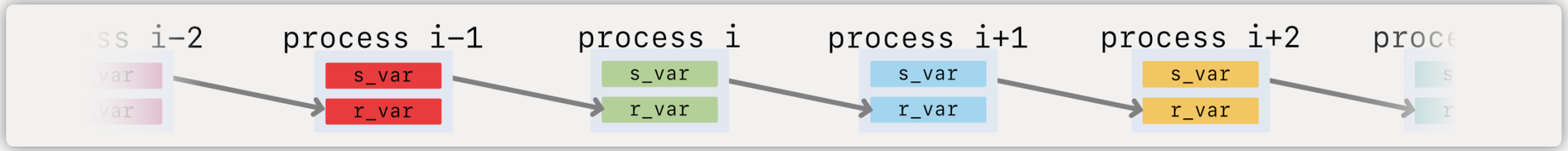
- This will not work:

```
MPI_Send(&s_var, 1, MPI_DOUBLE, rank+1, 0, MPI_COMM_WORLD);
MPI_Recv(&r_var, 1, MPI_DOUBLE, rank-1, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

- It results in a deadlock:
  - an `MPI_Recv()` can only be posted once an `MPI_Send()` completes
  - an `MPI_Send()` can only complete if a matching `MPI_Recv()` is posted

Point-to-point communication

- It is common in parallel applications to require that every process communicates with another process, e.g. a neighboring process



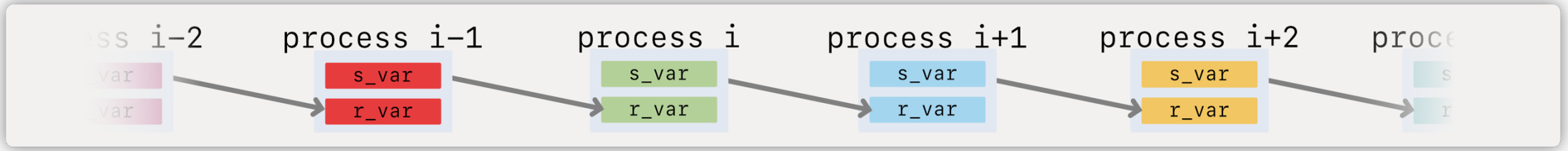
- This will not work:

```
MPI_Send(&s_var, 1, MPI_DOUBLE, rank+1, 0, MPI_COMM_WORLD);
MPI_Recv(&r_var, 1, MPI_DOUBLE, rank-1, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

- It results in a deadlock:
  - an `MPI_Recv()` can only be posted once an `MPI_Send()` completes
  - an `MPI_Send()` can only complete if a matching `MPI_Recv()` is posted
- One can serialize the communications, i.e. use a loop to determine the order of send/receives

Point-to-point communication

- It is common in parallel applications to require that every process communicates with another process, e.g. a neighboring process



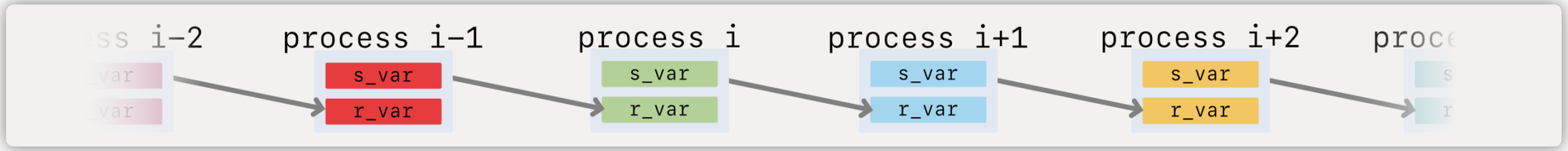
- This will not work:

```
MPI_Send(&s_var, 1, MPI_DOUBLE, rank+1, 0, MPI_COMM_WORLD);
MPI_Recv(&r_var, 1, MPI_DOUBLE, rank-1, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

- It results in a deadlock:
  - an `MPI_Recv()` can only be posted once an `MPI_Send()` completes
  - an `MPI_Send()` can only complete if a matching `MPI_Recv()` is posted
- One can serialize the communications, i.e. use a loop to determine the order of send/receives
  - Serializes communications that may be done faster in parallel

Point-to-point communication

- It is common in parallel applications to require that every process communicates with another process, e.g. a neighboring process



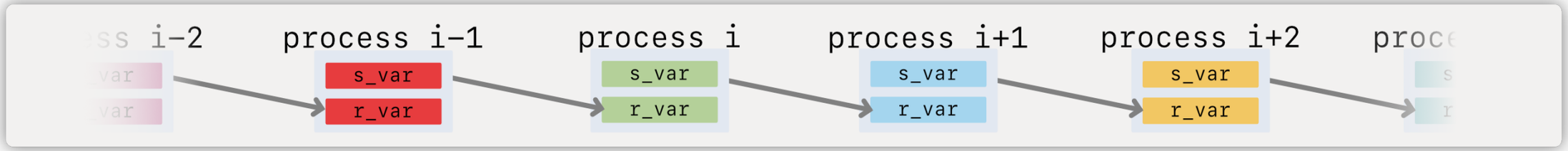
- This will not work:

```
MPI_Send(&s_var, 1, MPI_DOUBLE, rank+1, 0, MPI_COMM_WORLD);
MPI_Recv(&r_var, 1, MPI_DOUBLE, rank-1, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

- It results in a deadlock:
  - an `MPI_Recv()` can only be posted once an `MPI_Send()` completes
  - an `MPI_Send()` can only complete if a matching `MPI_Recv()` is posted
- One can serialize the communications, i.e. use a loop to determine the order of send/receives
  - Serializes communications that may be done faster in parallel
  - Inelegant, obscure, and error-prone

Point-to-point communication

- It is common in parallel applications to require that every process communicates with another process, e.g. a neighboring process

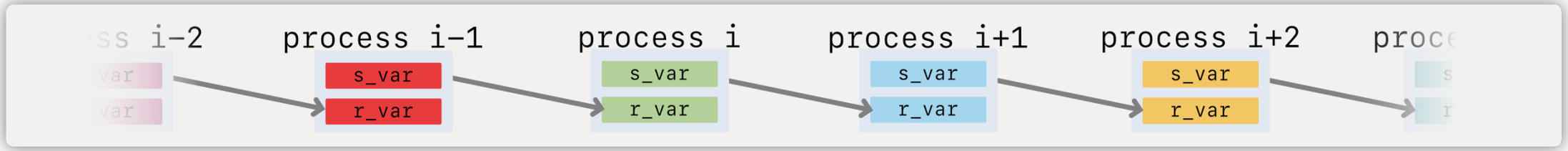


- A more efficient and elegant solution is to use `MPI_Sendrecv()`:

```
MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag,
             void *recvbuf, int recvcount, MPI_Datatype recvtype, int srce, int recvtag,
             MPI_Comm comm, MPI_Status *status)
```

Point-to-point communication

- It is common in parallel applications to require that every process communicates with another process, e.g. a neighboring process



- A more efficient and elegant solution is to use `MPI_Sendrecv()`:

```
MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag,
             void *recvbuf, int recvcount, MPI_Datatype recvtype, int srce, int recvtag,
             MPI_Comm comm, MPI_Status *status)
```

- For the depicted example:

```
MPI_Sendrecv(&s_var, 1, MPI_DOUBLE, rank+1, 0,
             &r_var, 1, MPI_DOUBLE, rank-1, MPI_ANY_TAG,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Point-to-point communications

Some additional notes on variants of the point-to-point communications we have covered

Point-to-point communications

Some additional notes on variants of the point-to-point communications we have covered

- `MPI_Isend()` and `MPI_Irecv()`
  - *Non-blocking* variants. The `I` stands for "immediate"

Point-to-point communications

Some additional notes on variants of the point-to-point communications we have covered

- `MPI_Isend()` and `MPI_Irecv()`
  - *Non-blocking* variants. The `I` stands for "immediate"
  - Functions return immediately, i.e. the functions don't block waiting for `sendbuf` to be sent or `recvbuf` to be received

Point-to-point communications

Some additional notes on variants of the point-to-point communications we have covered

- `MPI_Isend()` and `MPI_Irecv()`
  - *Non-blocking* variants. The `I` stands for "immediate"
  - Functions return immediately, i.e. the functions don't block waiting for `sendbuf` to be sent or `recvbuf` to be received
  - The function `MPI_Wait()` is used to block until the operation has complete

```
MPI_Isend(sendbuf, ... , request);
/*
 * More code can come here, provided it
 * does not modify sendbuf, which is
 * assumed to be "in-flight"
 */
MPI_Wait(request, ... );
```

Point-to-point communications

Some additional notes on variants of the point-to-point communications we have covered

- `MPI_Isend()` and `MPI_Irecv()`
  - *Non-blocking* variants. The `I` stands for "immediate"
  - Functions return immediately, i.e. the functions don't block waiting for `sendbuf` to be sent or `recvbuf` to be received
  - The function `MPI_Wait()` is used to block until the operation has complete

```
MPI_Isend(sendbuf, ..., request);
/*
 * More code can come here, provided it
 * does not modify sendbuf, which is
 * assumed to be "in-flight"
 */
MPI_Wait(request, ...);
```

- `MPI_Sendrecv_replace()`
  - Like `MPI_Sendrecv()` but with a single `buf` rather than separate `sendbuf` and `recvbuf`

Point-to-point communications

Some additional notes on variants of the point-to-point communications we have covered

- `MPI_Isend()` and `MPI_Irecv()`
  - *Non-blocking* variants. The `I` stands for "immediate"
  - Functions return immediately, i.e. the functions don't block waiting for `sendbuf` to be sent or `recvbuf` to be received
  - The function `MPI_Wait()` is used to block until the operation has complete

```
MPI_Isend(sendbuf, ... , request);
/*
 * More code can come here, provided it
 * does not modify sendbuf, which is
 * assumed to be "in-flight"
 */
MPI_Wait(request, ... );
```

- `MPI_Sendrecv_replace()`
  - Like `MPI_Sendrecv()` but with a single `buf` rather than separate `sendbuf` and `recvbuf`
    ↳ The receive message overwrites the send message

Exercises

Exercises

```
cp -r /onyx/data/edu12/mpi/ex .
```

- Exercises follow a common structure

Exercises

```
cp -r /onyx/data/edu12/mpi/ex .
```

- Exercises follow a common structure
- With the previous training event being a prerequisite, it is assumed that you:
  - Know how to login to Cyclone, navigate the filesystem, and modify files
  - Are familiar with Slurm, the job scheduler, and its commands: `sbatch`, `squeue`, etc.
  - Are familiar with the modules system

Exercises

```
cp -r /onyx/data/edu12/mpi/ex .
```

- Exercises follow a common structure
- With the previous training event being a prerequisite, it is assumed that you:
  - Know how to login to Cyclone, navigate the filesystem, and modify files
  - Are familiar with Slurm, the job scheduler, and its commands: `sbatch`, `squeue`, etc.
  - Are familiar with the modules system
- Each folder includes (where `${n}` below is the exercise number, i.e. `01`, `02`, etc.):
  - A makefile (`Makefile`)
  - A `.c` source code file (`ex${n}.c`)
  - A submit script (`sub-${n}.sh`)

Exercises

```
cp -r /onyx/data/edu12/mpi/ex .
```

- Exercises follow a common structure
- With the previous training event being a prerequisite, it is assumed that you:
  - Know how to login to Cyclone, navigate the filesystem, and modify files
  - Are familiar with Slurm, the job scheduler, and its commands: `sbatch`, `squeue`, etc.
  - Are familiar with the modules system
- Each folder includes (where `${n}` below is the exercise number, i.e. `01`, `02`, etc.):
  - A makefile (`Makefile`)
  - A `.c` source code file (`ex${n}.c`)
  - A submit script (`sub-${n}.sh`)
- Our workflow will typically be:
  - Modify `ex${n}.c` as instructed
  - Compile by typing `make`
  - Submit the job script `sbatch sub-${n}.sh`
  - Look at the output, which can be found in `ex${n}-output.txt`

Exercises

```
cp -r /onyx/data/edu12/mpi/ex .
```

- Exercises follow a common structure
- With the previous training event being a prerequisite, it is assumed that you:
  - Know how to login to Cyclone, navigate the filesystem, and modify files
  - Are familiar with Slurm, the job scheduler, and its commands: `sbatch`, `squeue`, etc.
  - Are familiar with the modules system
- Each folder includes (where `${n}` below is the exercise number, i.e. `01`, `02`, etc.):
  - A makefile (`Makefile`)
  - A `.c` source code file (`ex${n}.c`)
  - A submit script (`sub-${n}.sh`)
- Our workflow will typically be:
  - Modify `ex${n}.c` as instructed
  - Compile by typing `make`
  - Submit the job script `sbatch sub-${n}.sh`
  - Look at the output, which can be found in `ex${n}-output.txt`
- Note that if you have modified `ex${n}.c` correctly, the job should complete in less than one minute

Exercises

- Exercises are mostly complete but require some minor modifications by you

Exercises

- Exercises are mostly complete but require some minor modifications by you
- This is mostly to "encourage" reading and understanding the code

Exercises

- Exercises are mostly complete but require some minor modifications by you
- This is mostly to "encourage" reading and understanding the code
- The MPI functions demonstrated in each exercise are:
  - `ex01`: Use of `MPI_Comm_rank()` and `MPI_Comm_size()`
  - `ex02`: Use of `MPI_Barrier()`
  - `ex03`: Use of `MPI_Bcast()`
  - `ex04`: Use of `MPI_Scatter()`
  - `ex05`: Use of `MPI_Scatter()` and `MPI_Gather()`
  - `ex06`: Use of `MPI_Scatter()` and `MPI_Reduce()`
  - `ex07`: Use of `MPI_Send()` and `MPI_Recv()`
  - `ex08`: Use of `MPI_Sendrecv()`

Exercises

- Exercises are mostly complete but require some minor modifications by you
- This is mostly to "encourage" reading and understanding the code
- The MPI functions demonstrated in each exercise are:
  - `ex01`: Use of `MPI_Comm_rank()` and `MPI_Comm_size()`
  - `ex02`: Use of `MPI_Barrier()`
  - `ex03`: Use of `MPI_Bcast()`
  - `ex04`: Use of `MPI_Scatter()`
  - `ex05`: Use of `MPI_Scatter()` and `MPI_Gather()`
  - `ex06`: Use of `MPI_Scatter()` and `MPI_Reduce()`
  - `ex07`: Use of `MPI_Send()` and `MPI_Recv()`
  - `ex08`: Use of `MPI_Sendrecv()`
- All exercises have been tested with OpenMPI and the GNU Compiler. Please use:

  ```
  module load gompi
  ```

  for all exercises.

Exercises

Ex01

- Modify `ex01.c` to call `MPI_Comm_size()` and `MPI_Comm_rank()` with the appropriate arguments

```c
int nproc, rank;
/*
* TODO: call `MPI_Comm_size()' and `MPI_Comm_rank()' with the
* appropriate arguments
*/
MPI_Comm_size(/* TODO */);
MPI_Comm_rank(/* TODO */);
```

Exercises

Ex01

- Modify `ex01.c` to call `MPI_Comm_size()` and `MPI_Comm_rank()` with the appropriate arguments

```
int nproc, rank;
/*
* TODO: call `MPI_Comm_size()' and `MPI_Comm_rank()' with the
* appropriate arguments
*/
MPI_Comm_size(/* TODO */);
MPI_Comm_rank(/* TODO */);
```

- To compile, type `make`. Remember to first load the appropriate module (`gompi`):

```
[user@front01 ex01]$ module load gompi
[user@front01 ex01]$ make
mpicc -c ex01.c
mpicc -o ex01 ex01.o
[user@front01 ex01]$
```

Exercises

Ex01

- A job script has been prepared to run `ex01`:

```
[user@front01 ex01]$ cat sub-01.sh
#!/bin/bash
#SBATCH --job-name=01
#SBATCH --nodes=2
#SBATCH --ntasks=8
#SBATCH --output=ex01-output.txt
#SBATCH --time=00:01:00

module load gompi
mpirun ./ex01
```

Exercises

Ex01

- A job script has been prepared to run `ex01`:

```
[user@front01 ex01]$ cat sub-01.sh
#!/bin/bash
#SBATCH --job-name=01
#SBATCH --nodes=2
#SBATCH --ntasks=8
#SBATCH --output=ex01-output.txt
#SBATCH --time=00:01:00

module load gompi
mpirun ./ex01
```

  - 2 nodes, 8 processes, meaning 4 processes per node

Exercises

Ex01

- A job script has been prepared to run `ex01`:

```
[user@front01 ex01]$ cat sub-01.sh
#!/bin/bash
#SBATCH --job-name=01
#SBATCH --nodes=2
#SBATCH --ntasks=8
#SBATCH --output=ex01-output.txt
#SBATCH --time=00:01:00

module load gompi
mpirun ./ex01
```

  - 2 nodes, 8 processes, meaning 4 processes per node
  - program output will be redirected to file `ex01-output.txt`

Exercises

Ex01

- A job script has been prepared to run `ex01`:

```
[user@front01 ex01]$ cat sub-01.sh
#!/bin/bash
#SBATCH --job-name=01
#SBATCH --nodes=2
#SBATCH --ntasks=8
#SBATCH --output=ex01-output.txt
#SBATCH --time=00:01:00

module load gompi
mpirun ./ex01
```

  - 2 nodes, 8 processes, meaning 4 processes per node
  - program output will be redirected to file `ex01-output.txt`
  - requests 1 minute. If not done by then, the scheduler will kill the job

Exercises

Ex01

- Submit the job script:

```
[user@front01 ex01]$ sbatch sub-01.sh
Submitted batch job 69711
[user@front01 ex01]$
```

Exercises

Ex01

- Submit the job script:

```
[user@front01 ex01]$ sbatch sub-01.sh
Submitted batch job 69711
[user@front01 ex01]$
```

- Check status of job:

```
[user@front01 ex01]$ squeue -u $USER
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
69712       cpu       01      user  R       0:00      2 cn[01-02]
[user@front01 ex01]$
```

Exercises

Ex01

- Submit the job script:

```
[user@front01 ex01]$ sbatch sub-01.sh
Submitted batch job 69711
[user@front01 ex01]$
```

- Check status of job:

```
[user@front01 ex01]$ squeue -u $USER
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
69712       cpu       01      user  R       0:00      2 cn[01-02]
[user@front01 ex01]$
```

- The job runs very quickly. You may see no output above if the job has finished:

```
[user@front01 ex01]$ squeue -u $USER
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
[user@front01 ex01]$
```

Exercises

Ex01

- If done, the file `ex01-output.txt` should have been created
- Inspect the file:

```
[user@front01 ex01]$ cat ex01-output.txt
This is rank = 3 of nproc = 8 on node: cn01
This is rank = 0 of nproc = 8 on node: cn01
This is rank = 1 of nproc = 8 on node: cn01
This is rank = 2 of nproc = 8 on node: cn01
This is rank = 7 of nproc = 8 on node: cn02
This is rank = 5 of nproc = 8 on node: cn02
This is rank = 6 of nproc = 8 on node: cn02
This is rank = 4 of nproc = 8 on node: cn02
[user@front01 ex01]$
```

Exercises

Ex01

- If done, the file `ex01-output.txt` should have been created
- Inspect the file:

```
[user@front01 ex01]$ cat ex01-output.txt
This is rank = 3 of nproc = 8 on node: cn01
This is rank = 0 of nproc = 8 on node: cn01
This is rank = 1 of nproc = 8 on node: cn01
This is rank = 2 of nproc = 8 on node: cn01
This is rank = 7 of nproc = 8 on node: cn02
This is rank = 5 of nproc = 8 on node: cn02
This is rank = 6 of nproc = 8 on node: cn02
This is rank = 4 of nproc = 8 on node: cn02
[user@front01 ex01]$
```

- Note the order is nondeterministic; whichever process reaches the print statement first prints

Exercises

Ex01

- If done, the file `ex01-output.txt` should have been created
- Inspect the file:

```
[user@front01 ex01]$ cat ex01-output.txt
This is rank = 3 of nproc = 8 on node: cn01
This is rank = 0 of nproc = 8 on node: cn01
This is rank = 1 of nproc = 8 on node: cn01
This is rank = 2 of nproc = 8 on node: cn01
This is rank = 7 of nproc = 8 on node: cn02
This is rank = 5 of nproc = 8 on node: cn02
This is rank = 6 of nproc = 8 on node: cn02
This is rank = 4 of nproc = 8 on node: cn02
[user@front01 ex01]$
```

- Note the order is nondeterministic; whichever process reaches the print statement first prints
- We can use synchronization to serialize the print statements and ensure the correct order

Exercises

Ex02

- `ex02.c` is similar to `ex01.c`
- A for-loop is included over the print statement:

```
/*
* TODO: add an MPI_Barrier() to ensure the ranks print in-order
*/
for(int i=0; i<nproc; i++) {
    if(rank == i)
        printf(" This is rank = %d of nproc = %d on node: %s\n", rank, nproc, hname);
}
```

Exercises

Ex02

- `ex02.c` is similar to `ex01.c`
- A for-loop is included over the print statement:

```
/*
* TODO: add an MPI_Barrier() to ensure the ranks print in-order
*/
for(int i=0; i<nproc; i++) {
    if(rank == i)
        printf(" This is rank = %d of nproc = %d on node: %s\n", rank, nproc, hname);
}
```

- Study the code and carefully place an `MPI_Barrier()` so that the print statements will be executed in rank ordered

Exercises

Ex02

- `ex02.c` is similar to `ex01.c`
- A for-loop is included over the print statement:

```
/*
* TODO: add an MPI_Barrier() to ensure the ranks print in-order
*/
for(int i=0; i<nproc; i++) {
    if(rank == i)
        printf(" This is rank = %d of nproc = %d on node: %s\n", rank, nproc, hname);
}
```

- Study the code and carefully place an `MPI_Barrier()` so that the print statements will be executed in rank ordered
- As before, when done:
  - use `make` to compile and
  - `sbatch sub-02.sh` to submit the prepared job script

Exercises

Ex02

- Inspect the file `ex02-output.txt`:

```
[user@front01 ex02]$ cat ex02-output.txt
This is rank = 0 of nproc = 8 on node: cn01
This is rank = 1 of nproc = 8 on node: cn01
This is rank = 2 of nproc = 8 on node: cn01
This is rank = 3 of nproc = 8 on node: cn01
This is rank = 4 of nproc = 8 on node: cn02
This is rank = 5 of nproc = 8 on node: cn02
This is rank = 6 of nproc = 8 on node: cn02
This is rank = 7 of nproc = 8 on node: cn02
[user@front01 ex02]$
```

Exercises

Ex02

- Inspect the file `ex02-output.txt`:

```
[user@front01 ex02]$ cat ex02-output.txt
This is rank = 0 of nproc = 8 on node: cn01
This is rank = 1 of nproc = 8 on node: cn01
This is rank = 2 of nproc = 8 on node: cn01
This is rank = 3 of nproc = 8 on node: cn01
This is rank = 4 of nproc = 8 on node: cn02
This is rank = 5 of nproc = 8 on node: cn02
This is rank = 6 of nproc = 8 on node: cn02
This is rank = 7 of nproc = 8 on node: cn02
[user@front01 ex02]$
```

- The print statements should appear in rank order

Exercises

Ex03

- This exercise demonstrates `MPI_Bcast()`
- A file `data.txt` is included:

```
[user@front01 ex03]$ cat data.txt
3.14159265359
[user@front01 ex03]$
```

Exercises

Ex03

- This exercise demonstrates `MPI_Bcast()`
- A file `data.txt` is included:

```
[user@front01 ex03]$ cat data.txt
3.14159265359
[user@front01 ex03]$
```

- In `ex03.c`, the root process (process with `rank == 0`) calls the `readline()` function to read the single line from the file

Exercises

Ex03

- This exercise demonstrates `MPI_Bcast()`
- A file `data.txt` is included:

```
[user@front01 ex03]$ cat data.txt
3.14159265359
[user@front01 ex03]$
```

- In `ex03.c`, the root process (process with `rank == 0`) calls the `readline()` function to read the single line from the file
- Your task is to use a `MPI_Bcast()` to broadcast the variable read to all processes:

```
/*
 * TODO: add an MPI_Bcast() with the appropriate arguments to
 * broadcast variable `val' from the root process to all processes
 */
MPI_Bcast(/* TODO */);
```

Exercises

Ex03

- If done correctly, `ex03-output.txt` should include the following output:

```
[user@front01 ex03]$ cat ex03-output.txt
This is rank = 7 of nproc = 8 on node: cn02 | Got from root var = 3.141593
This is rank = 6 of nproc = 8 on node: cn02 | Got from root var = 3.141593
This is rank = 5 of nproc = 8 on node: cn02 | Got from root var = 3.141593
This is rank = 4 of nproc = 8 on node: cn02 | Got from root var = 3.141593
This is rank = 3 of nproc = 8 on node: cn01 | Got from root var = 3.141593
This is rank = 0 of nproc = 8 on node: cn01 | Got from root var = 3.141593
This is rank = 2 of nproc = 8 on node: cn01 | Got from root var = 3.141593
This is rank = 1 of nproc = 8 on node: cn01 | Got from root var = 3.141593
[user@front01 ex03]$
```

  With the order being undetermined also in this case

Exercises

Ex04

- This exercise demonstrates the scatter operation
- The file `data.txt` now includes eight lines:

```
[user@front01 ex04]$ cat data.txt
0.42816572487
0.57721566490
0.66274341935
0.69314718056
1.41421356237
1.61803398875
2.71828182846
3.14159265359
[user@front01 ex04]$
```

Exercises

Ex04

- This exercise demonstrates the scatter operation
- The file `data.txt` now includes eight lines:

```
[user@front01 ex04]$ cat data.txt
0.42816572487
0.57721566490
0.66274341935
0.69314718056
1.41421356237
1.61803398875
2.71828182846
3.14159265359
[user@front01 ex04]$
```

- The root process reads the eight lines into the eight-element double precision array `vals[]` using the function `readlines()`

```
/*
 * root process: read `n' lines of "data.txt" into array `vars[]'
 */
int nelems = 8;
double vars[nelems];
if(rank == 0) {
    char fname[] = "data.txt";
    readlines(nelems, vars, fname);
}
```

Exercises

Ex04

- Your task is to scatter the array `vals[]` so that each of the eight processes receives one element of the array into variable `var`:

```
double var;
/*
 * TODO: use an MPI_Scatter() to distribute the elements of `vars[]'
 * to the processes, one element for each process. Assume the number
 * of processes is the same as the number of elements.
 */
MPI_Scatter(/* TODO */);
```

Exercises

Ex04

- Your task is to scatter the array `vals[]` so that each of the eight processes receives one element of the array into variable `var`:

```
double var;
/*
 * TODO: use an MPI_Scatter() to distribute the elements of `vars[]'
 * to the processes, one element for each process. Assume the number
 * of processes is the same as the number of elements.
 */
MPI_Scatter(/* TODO */);
```

- Once done, compile (`make`) and submit the job script (`sbatch sub-04.sh`)

Exercises

Ex04

- Your task is to scatter the array `vals[]` so that each of the eight processes receives one element of the array into variable `var`:

```
double var;
/*
 * TODO: use an MPI_Scatter() to distribute the elements of `vars[]'
 * to the processes, one element for each process. Assume the number
 * of processes is the same as the number of elements.
 */
MPI_Scatter(/* TODO */);
```

- Once done, compile (`make`) and submit the job script (`sbatch sub-04.sh`)
- If done correctly, you should see the following in `ex04-output.txt`:

```
[user@front01 ex04]$ cat ex04-output.txt
This is rank = 2 of nproc = 8 on node: cn01 | Got from root var = 0.662743
This is rank = 3 of nproc = 8 on node: cn01 | Got from root var = 0.693147
This is rank = 1 of nproc = 8 on node: cn01 | Got from root var = 0.577216
This is rank = 4 of nproc = 8 on node: cn02 | Got from root var = 1.414214
This is rank = 5 of nproc = 8 on node: cn02 | Got from root var = 1.618034
This is rank = 6 of nproc = 8 on node: cn02 | Got from root var = 2.718282
This is rank = 0 of nproc = 8 on node: cn01 | Got from root var = 0.428166
This is rank = 7 of nproc = 8 on node: cn02 | Got from root var = 3.141593
[user@front01 ex04]$
```

Exercises

Ex04

- Your task is to scatter the array `vals[]` so that each of the eight processes receives one element of the array into variable `var`:

```
double var;
/*
 * TODO: use an MPI_Scatter() to distribute the elements of `vars[]'
 * to the processes, one element for each process. Assume the number
 * of processes is the same as the number of elements.
 */
MPI_Scatter(/* TODO */);
```

- Once done, compile (`make`) and submit the job script (`sbatch sub-04.sh`)
- If done correctly, you should see the following in `ex04-output.txt`:

```
[user@front01 ex04]$ cat ex04-output.txt
This is rank = 2 of nproc = 8 on node: cn01 | Got from root var = 0.662743
This is rank = 3 of nproc = 8 on node: cn01 | Got from root var = 0.693147
This is rank = 1 of nproc = 8 on node: cn01 | Got from root var = 0.577216
This is rank = 4 of nproc = 8 on node: cn02 | Got from root var = 1.414214
This is rank = 5 of nproc = 8 on node: cn02 | Got from root var = 1.618034
This is rank = 6 of nproc = 8 on node: cn02 | Got from root var = 2.718282
This is rank = 0 of nproc = 8 on node: cn01 | Got from root var = 0.428166
This is rank = 7 of nproc = 8 on node: cn02 | Got from root var = 3.141593
[user@front01 ex04]$
```

- **Pro tip**: you can sort the output by piping through `sort`, i.e. `cat ex04-output.txt | sort`

Exercises

Ex05

- This exercise demonstrates the gather operation
- The exercise starts like `ex04`:
  - The root process reads the eight values from `data.txt` and stores them in `vars[]`

Exercises

Ex05

- This exercise demonstrates the gather operation
- The exercise starts like `ex04`:
  - The root process reads the eight values from `data.txt` and stores them in `vars[]`
- We would like:

Exercises

Ex05

- This exercise demonstrates the gather operation
- The exercise starts like `ex04`:
  - The root process reads the eight values from `data.txt` and stores them in `vars[]`
- We would like:
  - The elements of `vars[]` to be scattered, one element to each of eight processes (same as `ex04`)

Exercises

Ex05

- This exercise demonstrates the gather operation
- The exercise starts like `ex04`:
  - The root process reads the eight values from `data.txt` and stores them in `vars[]`
- We would like:
  - The elements of `vars[]` to be scattered, one element to each of eight processes (same as `ex04`)
  - Each process to divide its element, stored in `var`, by two

Exercises

Ex05

- This exercise demonstrates the gather operation
- The exercise starts like `ex04`:
  - The root process reads the eight values from `data.txt` and stores them in `vars[]`
- We would like:
  - The elements of `vars[]` to be scattered, one element to each of eight processes (same as `ex04`)
  - Each process to divide its element, stored in `var`, by two
  - The process' `var` variables to be gathered back into `vars[]` of the root process

Exercises

Ex05

- This exercise demonstrates the gather operation
- The exercise starts like `ex04`:
  - The root process reads the eight values from `data.txt` and stores them in `vars[]`
- We would like:
  - The elements of `vars[]` to be scattered, one element to each of eight processes (same as `ex04`)
  - Each process to divide its element, stored in `var`, by two
  - The process' `var` variables to be gathered back into `vars[]` of the root process

```
/* TODO: use an MPI_Scatter() to distribute the elements of `vars[]'
 * to the processes, one element for each process. Assume the number
 * of processes is the same as the number of elements. Same as in
 * exercise ex04.
 */
MPI_Scatter(/* TODO */);

/* Divide by two on each rank */
var = var*0.5;

/* TODO: use an MPI_Gather() to collect `var' from each rank to the
 * array `vars[]' on the root process
 */
MPI_Gather(/* TODO */);
```

Exercises

Ex05

- At the end, the root process prints all elements of `vars[]`

```
/*
 * root process: print the elements of `vars[]' obtained via the
 * `MPI_Gather()'
 */
if(rank == 0)
    for(int i=0; i<nelems; i++)
        printf(" vars[%d] = %lf\n", i, vars[i]);
```

Exercises

Ex05

- At the end, the root process prints all elements of `vars[]`

```
/*
 * root process: print the elements of `vars[]' obtained via the
 * `MPI_Gather()'
 */
if(rank == 0)
    for(int i=0; i<nelems; i++)
        printf(" vars[%d] = %lf\n", i, vars[i]);
```

- Inspect the output `ex05-output.txt` and ensure the result is correct:

```
[user@front01 ex05]$ cat ex05-output.txt
vars[0] = 0.214083
vars[1] = 0.288608
vars[2] = 0.331372
vars[3] = 0.346574
vars[4] = 0.707107
vars[5] = 0.809017
vars[6] = 1.359141
vars[7] = 1.570796
[user@front01 ex05]$
```

Exercises

Ex06

- This exercise demonstrates the reduction operation
- `data.txt` is now a file with 2520 random numbers, one per line (`nelems = 2520`)

Exercises

Ex06

- This exercise demonstrates the reduction operation
- `data.txt` is now a file with 2520 random numbers, one per line (`nelems = 2520`)
- We would like:

Exercises

Ex06

- This exercise demonstrates the reduction operation
- `data.txt` is now a file with 2520 random numbers, one per line (`nelems = 2520`)
- We would like:
  - The root process to read all elements into an array `vars[]`

Exercises

Ex06

- This exercise demonstrates the reduction operation
- `data.txt` is now a file with 2520 random numbers, one per line (`nelems = 2520`)
- We would like:
  - The root process to read all elements into an array `vars[]`
  - The elements to be scattered to all processes
    ↳ Each process should receive `nelems_loc = nelems / nproc` elements

Exercises

Ex06

- This exercise demonstrates the reduction operation
- `data.txt` is now a file with 2520 random numbers, one per line (`nelems = 2520`)
- We would like:
  - The root process to read all elements into an array `vars[]`
  - The elements to be scattered to all processes
    ↳ Each process should receive `nelems_loc = nelems / nproc` elements
  - Each process to sum its local elements, storing the result into `sum_loc`

Exercises

Ex06

- This exercise demonstrates the reduction operation
- `data.txt` is now a file with 2520 random numbers, one per line (`nelems = 2520`)
- We would like:
  - The root process to read all elements into an array `vars[]`
  - The elements to be scattered to all processes
    ↳ Each process should receive `nelems_loc = nelems / nproc` elements
  - Each process to sum its local elements, storing the result into `sum_loc`
  - To use a reduction operation to obtain the grand total over all 2520 elements

Exercises

Ex06

- This exercise demonstrates the reduction operation
- `data.txt` is now a file with 2520 random numbers, one per line (`nelems = 2520`)
- We would like:
  - The root process to read all elements into an array `vars[]`
  - The elements to be scattered to all processes
    ↳ Each process should receive `nelems_loc = nelems / nproc` elements
  - Each process to sum its local elements, storing the result into `sum_loc`
  - To use a reduction operation to obtain the grand total over all 2520 elements
- Note that in `ex06.c` we explicitly check that the number of processes divides the number of elements in `data.txt`:

```c
/*
 * Abort if the number of processes does not divide `nelems' exactly
 */
int nelems = 2520;
if(nelems % nproc =/= 0) {
    fprintf(stderr, " nelems = %d not divisible by nproc = %d\n", nelems, nproc);
    MPI_Abort(MPI_COMM_WORLD, 1);
}
```

Exercises

Ex06

- Your `TODO`s are to complete the scatter and reduction operations:

```
/*
 * TODO: use an MPI_Scatter() to distribute the elements of `vars[]'
 * to each process' `vars_loc[]' array
 */
MPI_Scatter(/* TODO */);

/*
 * `sum_loc' holds the sum over each process' local elements
 */
double sum_loc = 0;
for(int i=0; i<nelems_loc; i++)
sum_loc += vars_loc[i];

/*
 * TODO: use an MPI_Reduce() to sum `sum_loc' over all processes and
 * store in `sum' of the root process
 */
double sum;
MPI_Reduce(/* TODO */);
```

Exercises

Ex06

- The root process prints the result at the end. If correct, you should see:

```
[user@front01 ex06]$ cat ex06-output.txt
Used 8 processes, sum = 1266.960662
[user@front01 ex06]$
```

Exercises

Ex06

- The root process prints the result at the end. If correct, you should see:

```
[user@front01 ex06]$ cat ex06-output.txt
Used 8 processes, sum = 1266.960662
[user@front01 ex06]$
```

- Note that `ex06.c` allows running with any number of processes which divide 2520 exactly

Exercises

Ex06

- The root process prints the result at the end. If correct, you should see:

```
[user@front01 ex06]$ cat ex06-output.txt
Used 8 processes, sum = 1266.960662
[user@front01 ex06]$
```

- Note that `ex06.c` allows running with any number of processes which divide 2520 exactly
- Try, for example, modifying `sub-06.sh` to use 40 processes (20 per node)

Exercises

Ex06

- The root process prints the result at the end. If correct, you should see:

```
[user@front01 ex06]$ cat ex06-output.txt
Used 8 processes, sum = 1266.960662
[user@front01 ex06]$
```

- Note that `ex06.c` allows running with any number of processes which divide 2520 exactly
- Try, for example, modifying `sub-06.sh` to use 40 processes (20 per node)
- You should see an identical sum:

```
[user@front01 ex06]$ cat ex06-output.txt
Used 40 processes, sum = 1266.960662
[user@front01 ex06]$
```

Exercises

Ex07

- This exercise demonstrates `MPI_Send()` and `MPI_Recv()`
- `data.txt` is the same as in `ex04`, with eight elements:

```
[user@front01 ex07]$ cat data.txt
0.42816572487
0.57721566490
0.66274341935
0.69314718056
1.41421356237
1.61803398875
2.71828182846
3.14159265359
[user@front01 ex07]$
```

Exercises

Ex07

- This exercise demonstrates `MPI_Send()` and `MPI_Recv()`
- `data.txt` is the same as in `ex04`, with eight elements:

```
[user@front01 ex07]$ cat data.txt
0.42816572487
0.57721566490
0.66274341935
0.69314718056
1.41421356237
1.61803398875
2.71828182846
3.14159265359
[user@front01 ex07]$
```

- All elements are read by the root process into array `vars[]`

Exercises

Ex07

- This exercise demonstrates `MPI_Send()` and `MPI_Recv()`
- `data.txt` is the same as in `ex04`, with eight elements:

```
[user@front01 ex07]$ cat data.txt
0.42816572487
0.57721566490
0.66274341935
0.69314718056
1.41421356237
1.61803398875
2.71828182846
3.14159265359
[user@front01 ex07]$
```

- All elements are read by the root process into array `vars[]`
- The elements are then scattered, one to each process, and stored in variable `var0`

Exercises

Ex07

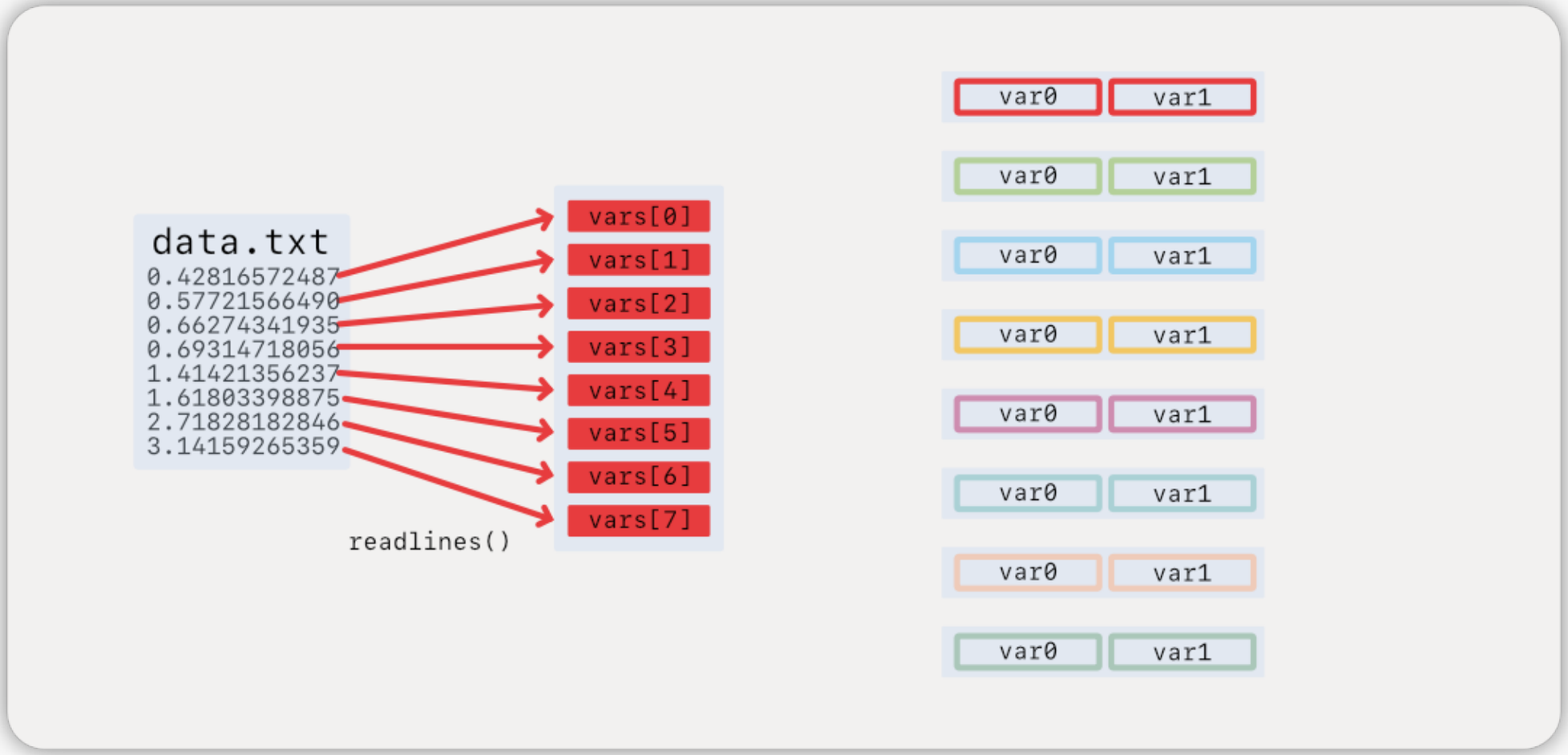
- This exercise demonstrates `MPI_Send()` and `MPI_Recv()`
- `data.txt` is the same as in `ex04`, with eight elements:

```
[user@front01 ex07]$ cat data.txt
0.42816572487
0.57721566490
0.66274341935
0.69314718056
1.41421356237
1.61803398875
2.71828182846
3.14159265359
[user@front01 ex07]$
```

- All elements are read by the root process into array `vars[]`
- The elements are then scattered, one to each process, and stored in variable `var0`
- Using `MPI_Send()` and `MPI_Recv()`, We would like that:
  - All processes with even ranks store in variable `var1` the value of `var0` corresponding to their next odd rank
  - All processes with odd ranks store in variable `var1` the value of `var0` corresponding to their previous even rank

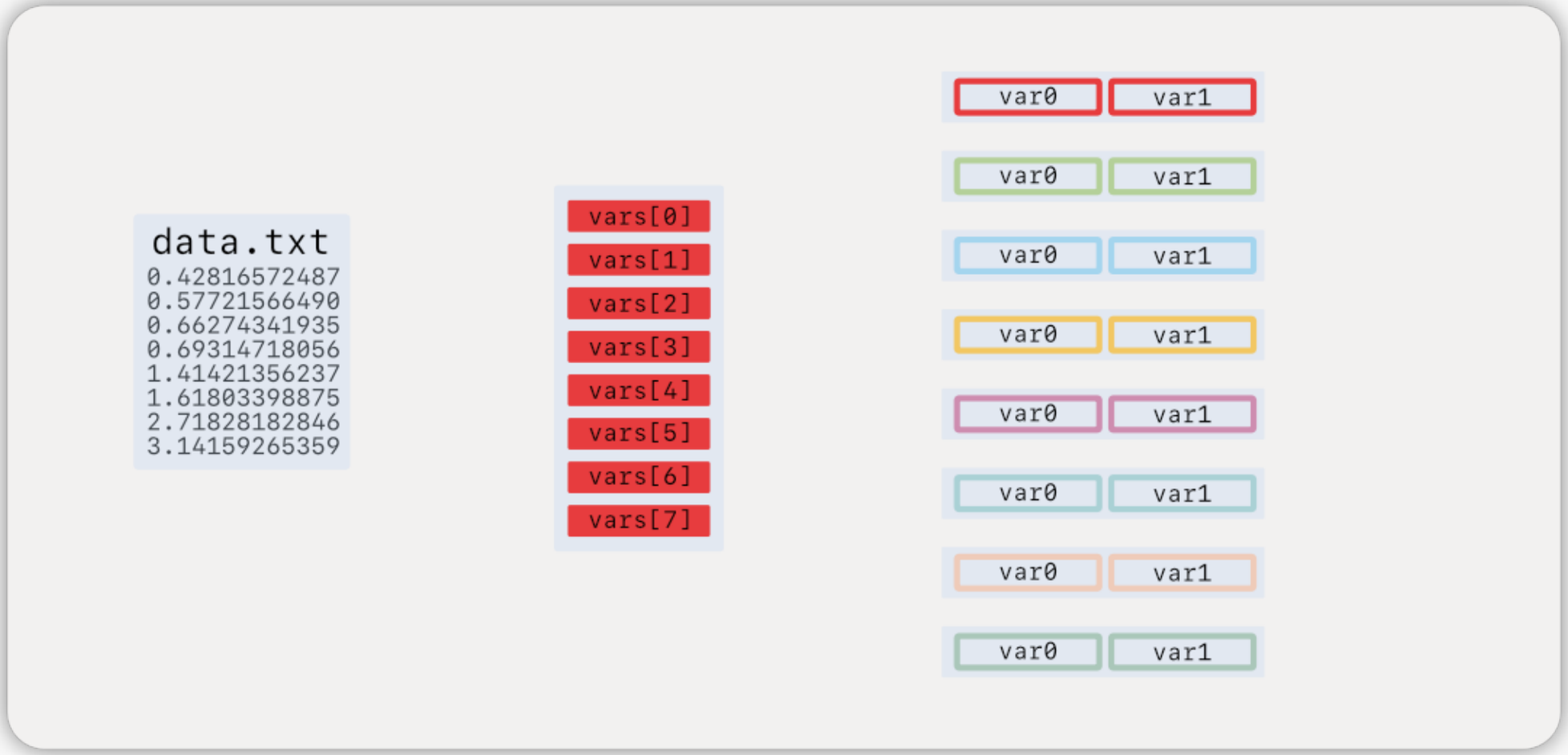
Exercises

Ex07



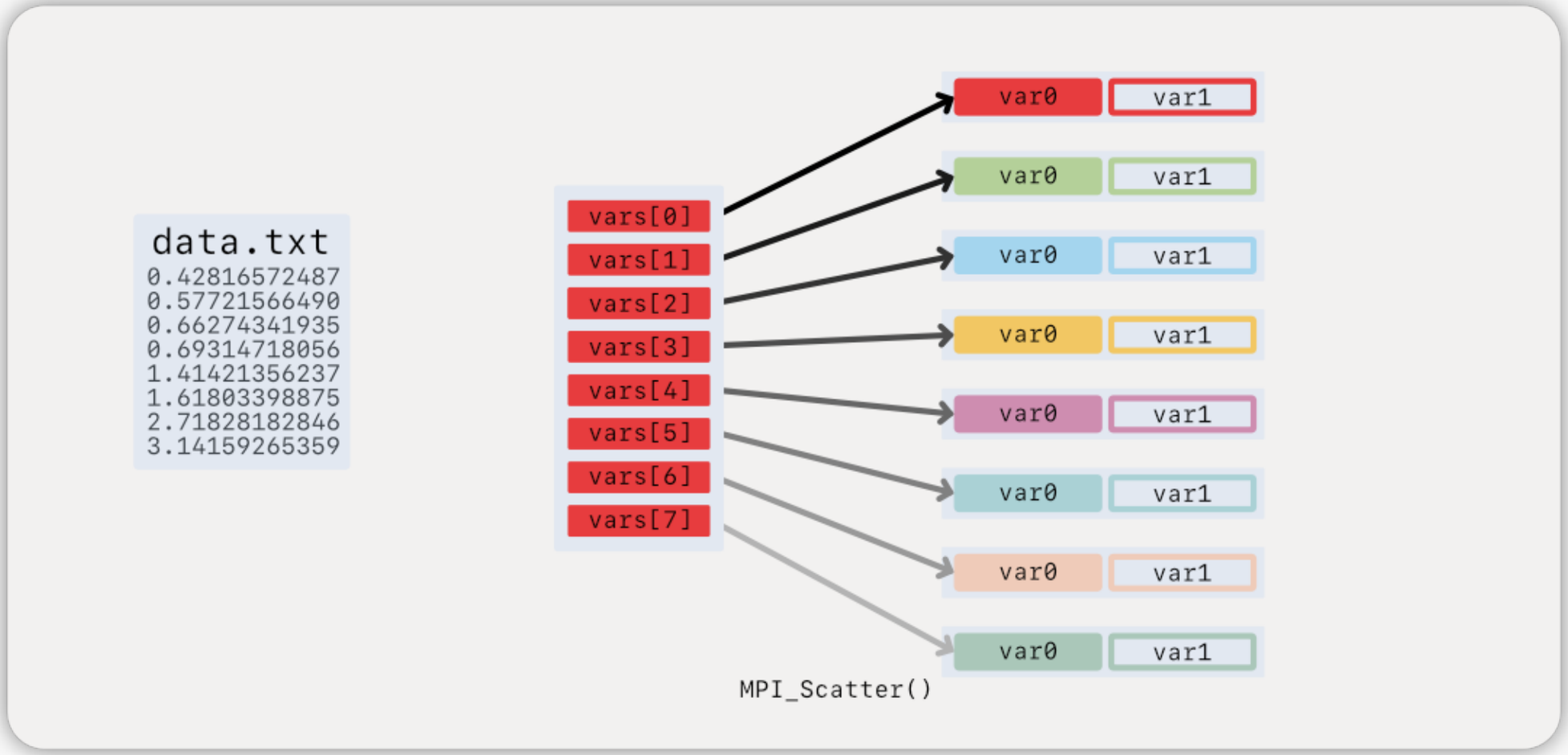
Exercises

Ex07



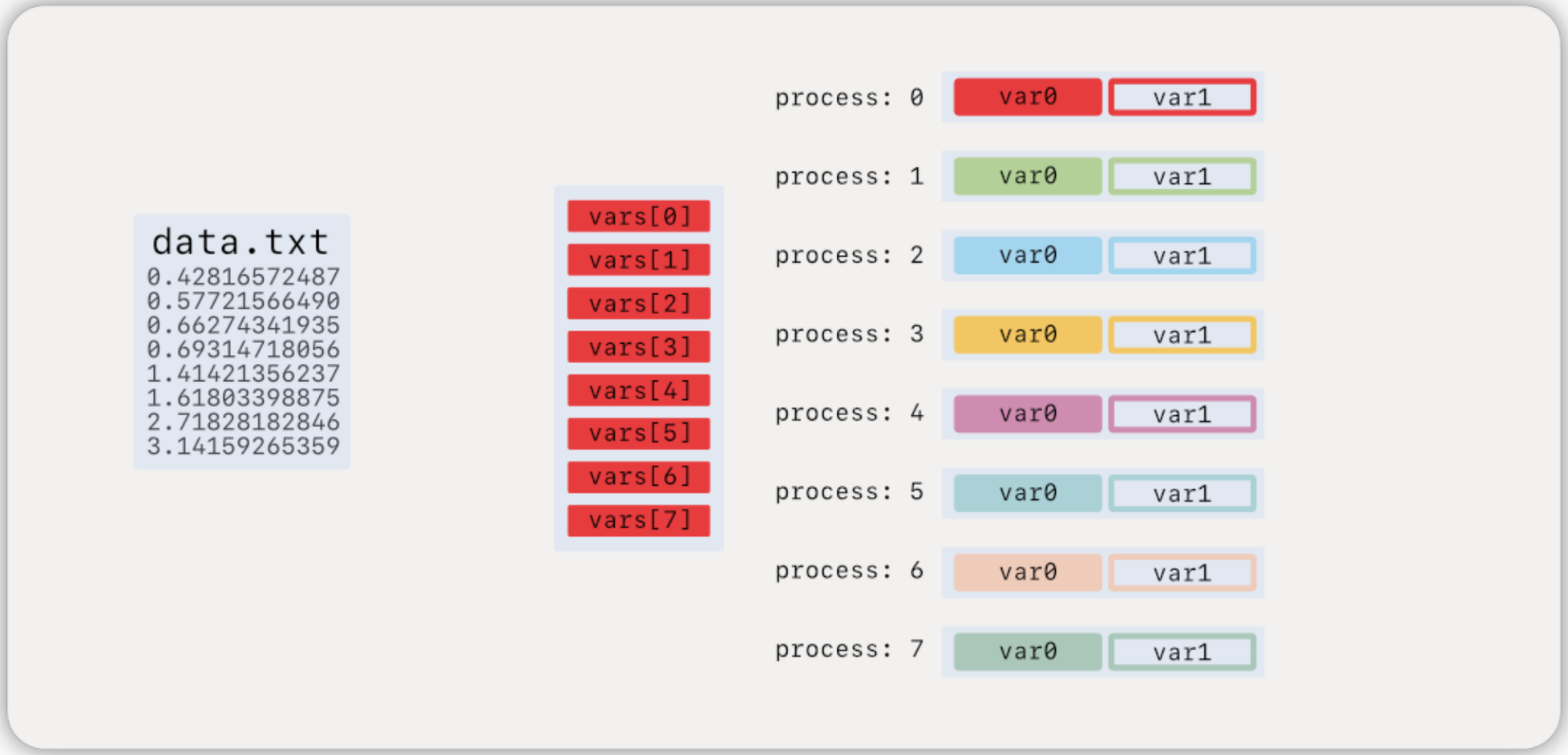
Exercises

Ex07



Exercises

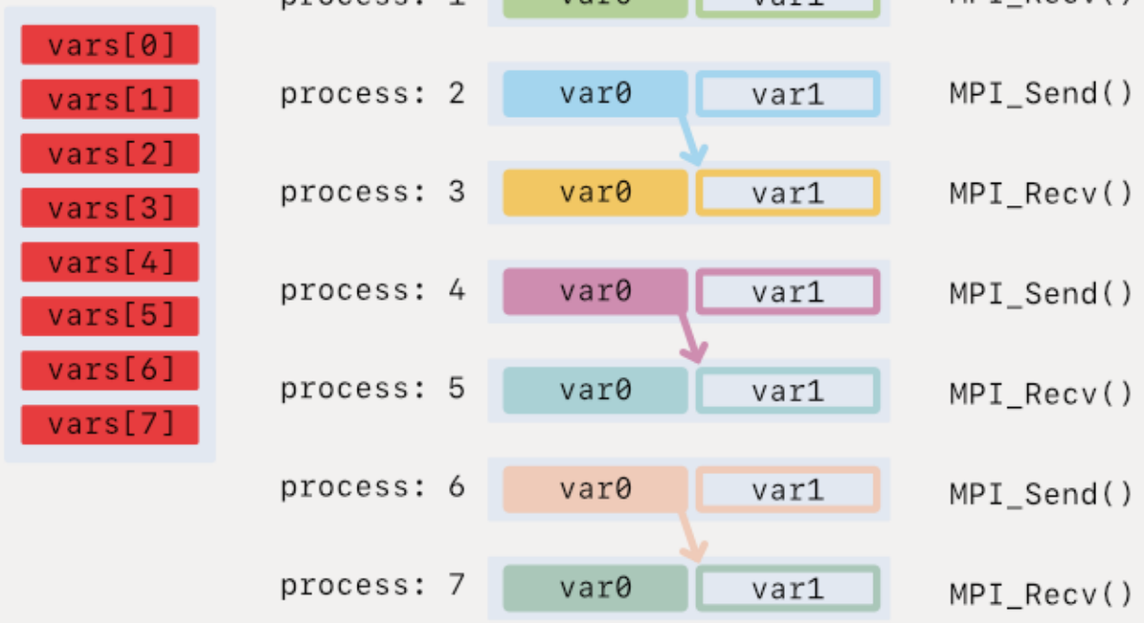
Ex07



Exercises

Ex07

data.txt
0.42816572487
0.57721566490
0.66274341935
0.69314718056
1.41421356237
1.61803398875
2.71828182846
3.14159265359

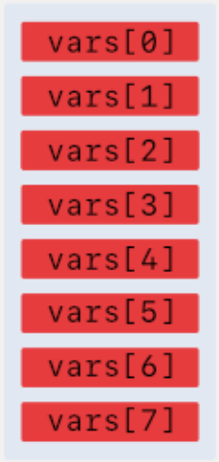


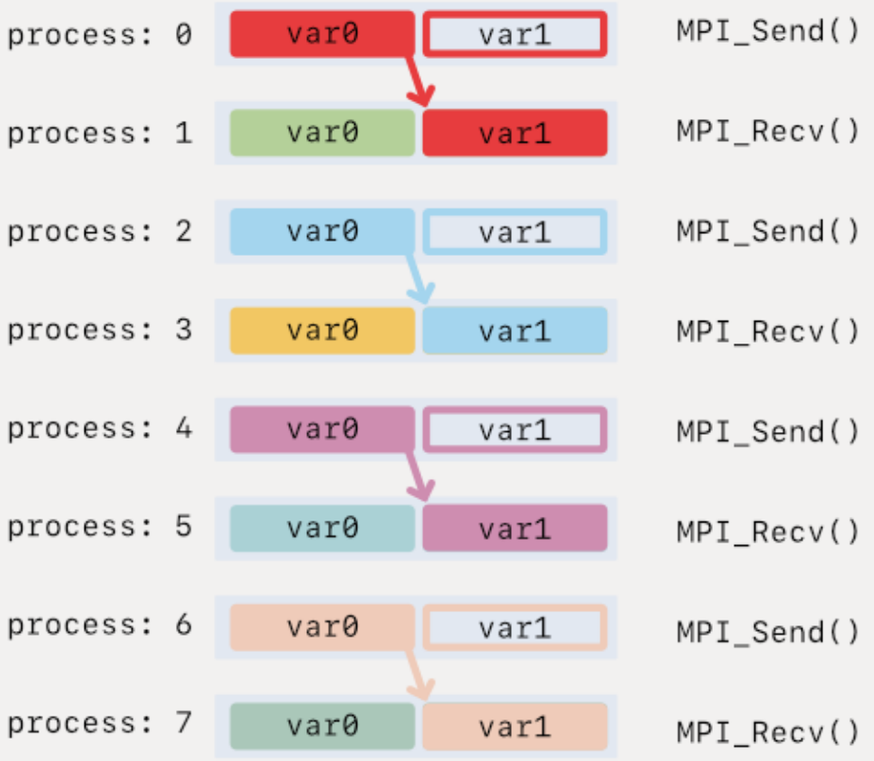
Exercises

Ex07

data.txt

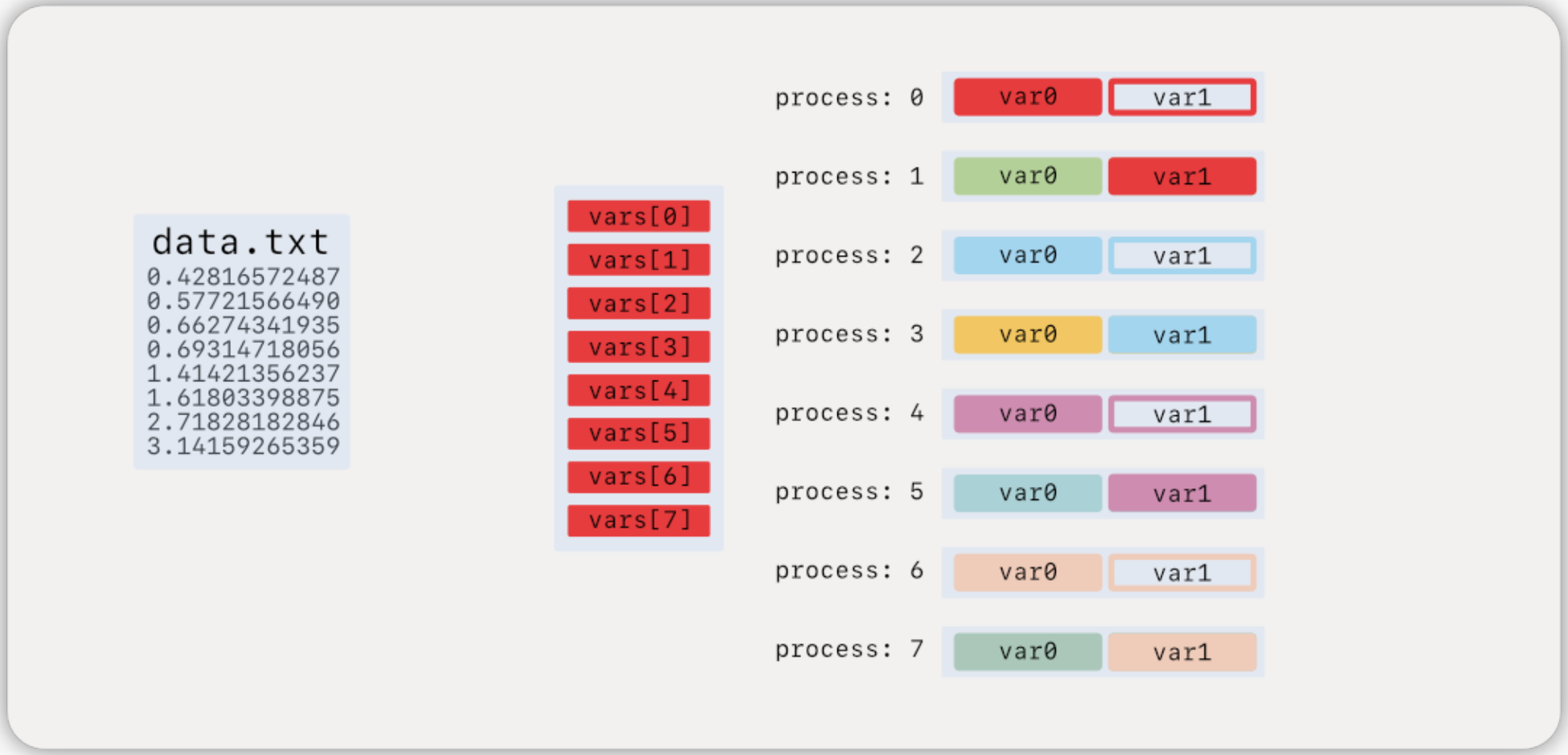
```
0.42816572487
0.57721566490
0.66274341935
0.69314718056
1.41421356237
1.61803398875
2.71828182846
3.14159265359
```





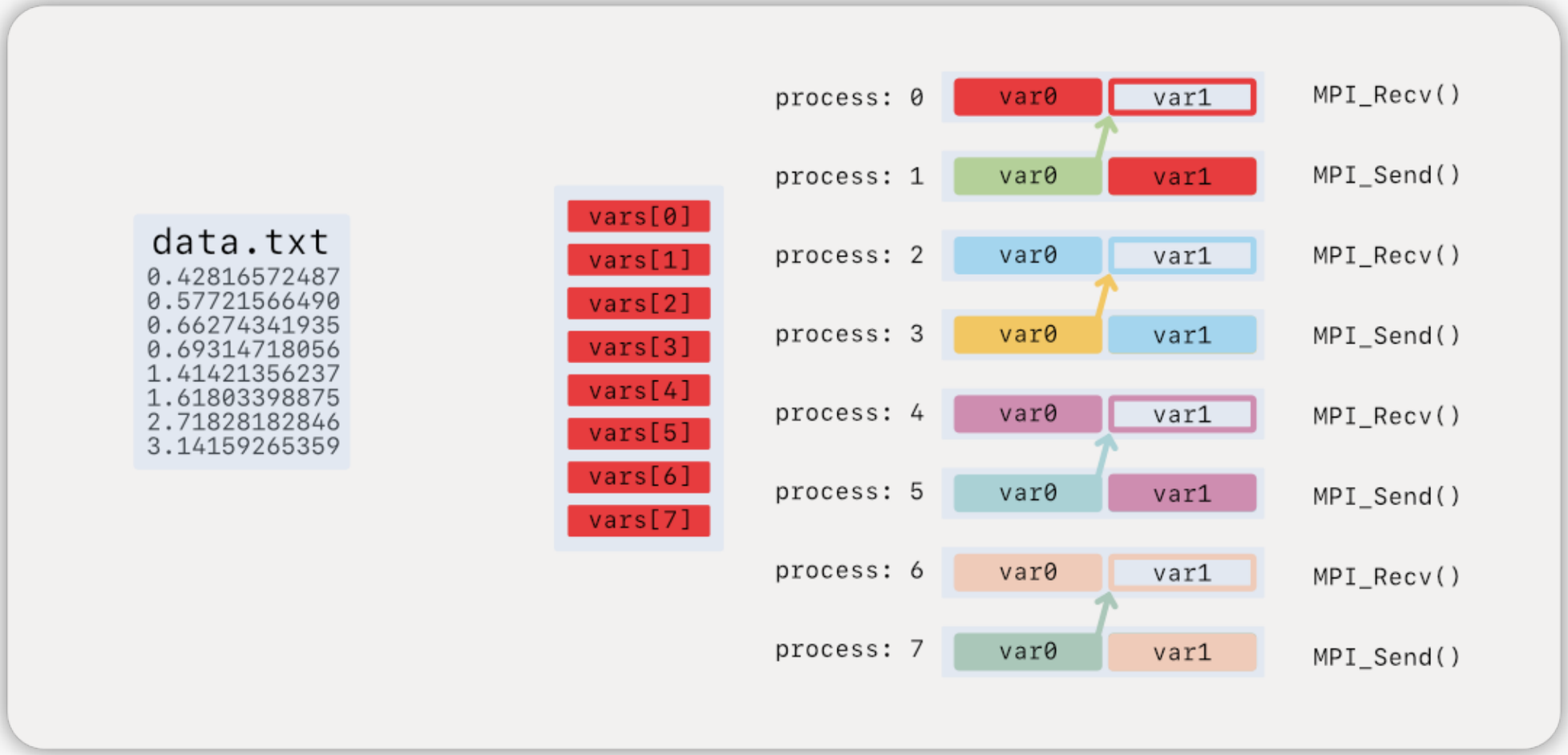
Exercises

Ex07



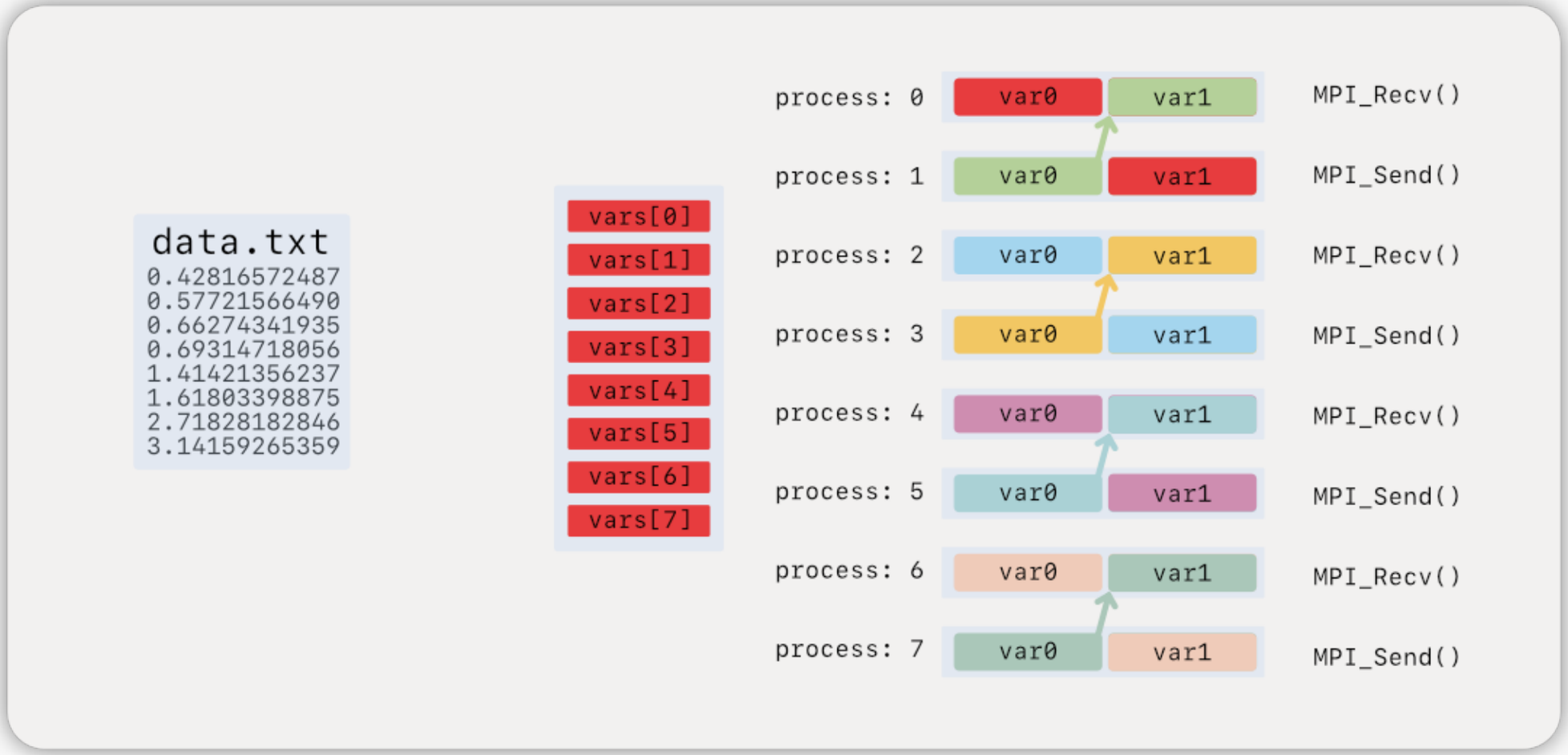
Exercises

Ex07



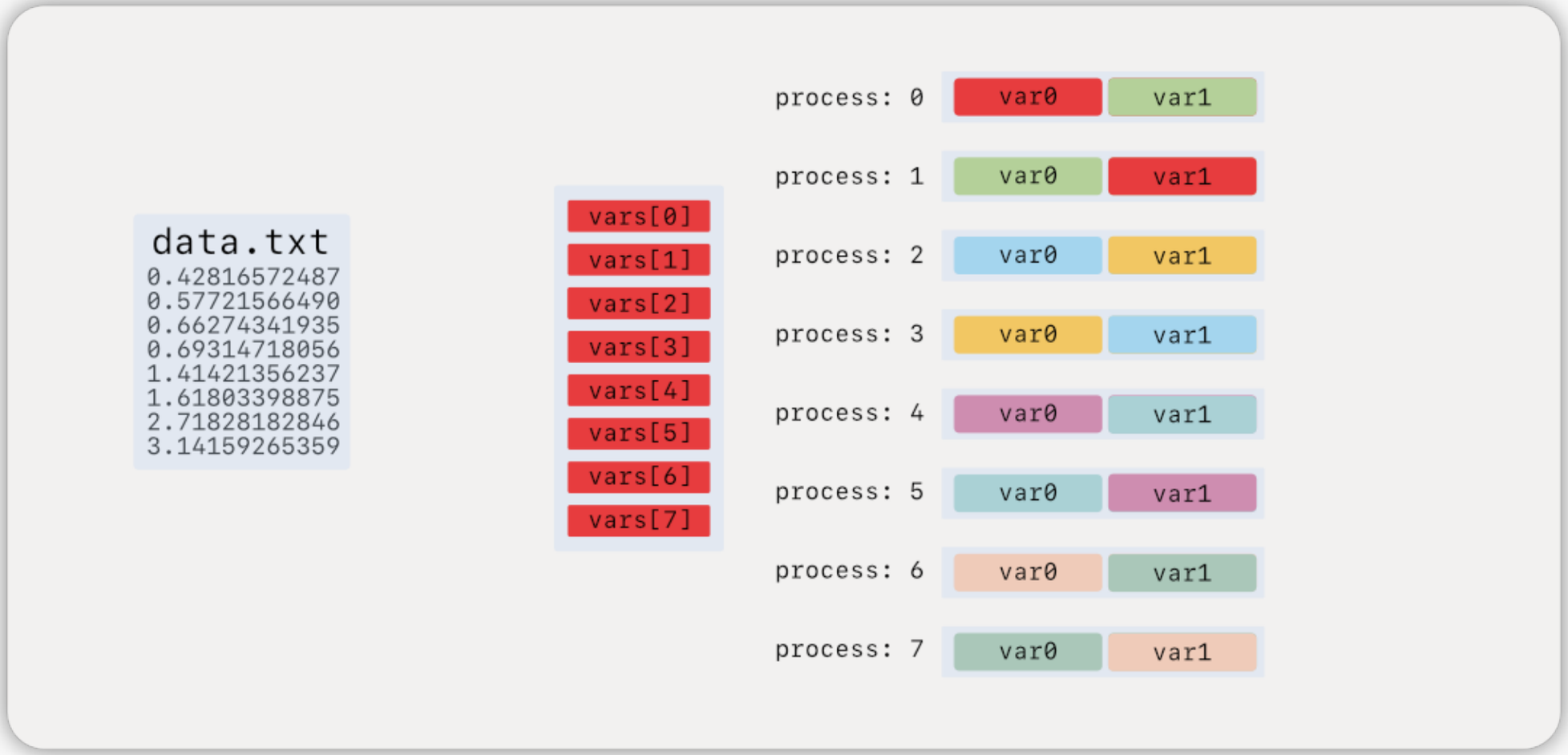
Exercises

Ex07



Exercises

Ex07



Exercises

Ex07

- The `TODO`s are for completing the arguments of the `MPI_Recv()`s and `MPI_Send()`s

```
double var1;
/*
 * TODO: Use `MPI_Send()' and `MPI_Recv()' appropriately, so that
 * `var0' of each even rank is copied into `var1' of the next odd
 * rank
 */
if(rank % 2 == 0) {
    MPI_Send(/* TODO */);
} else {
    MPI_Recv(/* TODO */);
}

/*
 * TODO: Use `MPI_Send()' and `MPI_Recv()' appropriately, so that
 * `var0' of each odd rank is copied into `var1' of the previous
 * even rank
 */
if(rank % 2 == 1) {
    MPI_Send(/* TODO */);
} else {
    MPI_Recv(/* TODO */);
}
```

Exercises

Ex07

- The correct output should look like this:

```
[user@front01 ex07]$ cat ex07-output.txt |sort
This is rank = 0 of nproc = 8 on node: cn01 | var0 = 0.428166   var1 = 0.577216
This is rank = 1 of nproc = 8 on node: cn01 | var0 = 0.577216   var1 = 0.428166
This is rank = 2 of nproc = 8 on node: cn01 | var0 = 0.662743   var1 = 0.693147
This is rank = 3 of nproc = 8 on node: cn01 | var0 = 0.693147   var1 = 0.662743
This is rank = 4 of nproc = 8 on node: cn02 | var0 = 1.414214   var1 = 1.618034
This is rank = 5 of nproc = 8 on node: cn02 | var0 = 1.618034   var1 = 1.414214
This is rank = 6 of nproc = 8 on node: cn02 | var0 = 2.718282   var1 = 3.141593
This is rank = 7 of nproc = 8 on node: cn02 | var0 = 3.141593   var1 = 2.718282
[user@front01 ex07]$
```

Exercises

Ex08

- This exercise demonstrates the use of `MPI_Sendrecv()`
- The same `data.txt` with eight elements is used as before
- The elements are read by the root process and scattered to all processes as before

Exercises

Ex08

- This exercise demonstrates the use of `MPI_Sendrecv()`
- The same `data.txt` with eight elements is used as before
- The elements are read by the root process and scattered to all processes as before
- Now each process has three variables:
  - `var_proc` contains the element received from the scatter
  - `var_next` is to contain `var_proc` of the next process
  - `var_prev` is to contain `var_proc` of the previous process

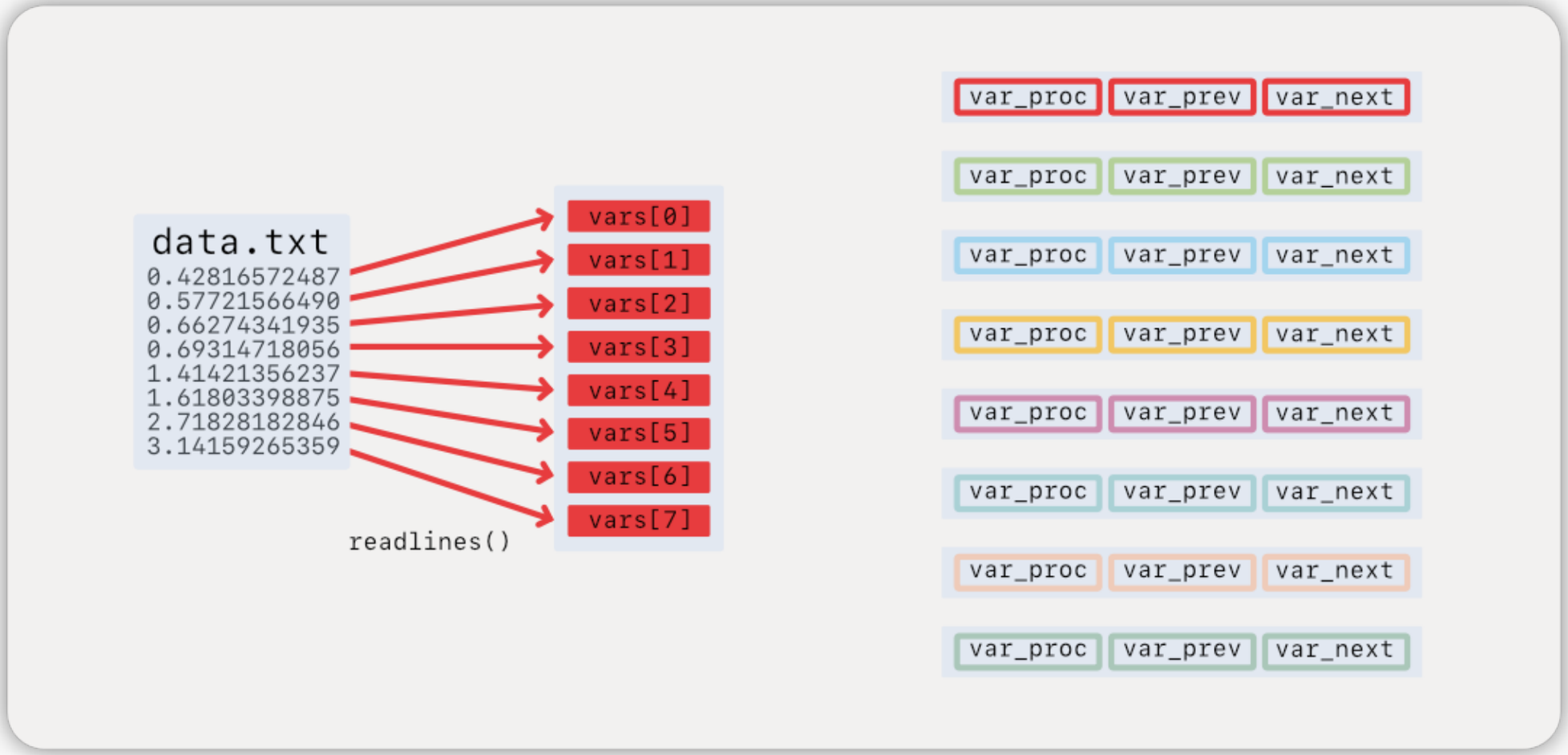
Exercises

Ex08

- This exercise demonstrates the use of `MPI_Sendrecv()`
- The same `data.txt` with eight elements is used as before
- The elements are read by the root process and scattered to all processes as before
- Now each process has three variables:
  - `var_proc` contains the element received from the scatter
  - `var_next` is to contain `var_proc` of the next process
  - `var_prev` is to contain `var_proc` of the previous process
- Use two `MPI_Sendrecv()` to achieve this

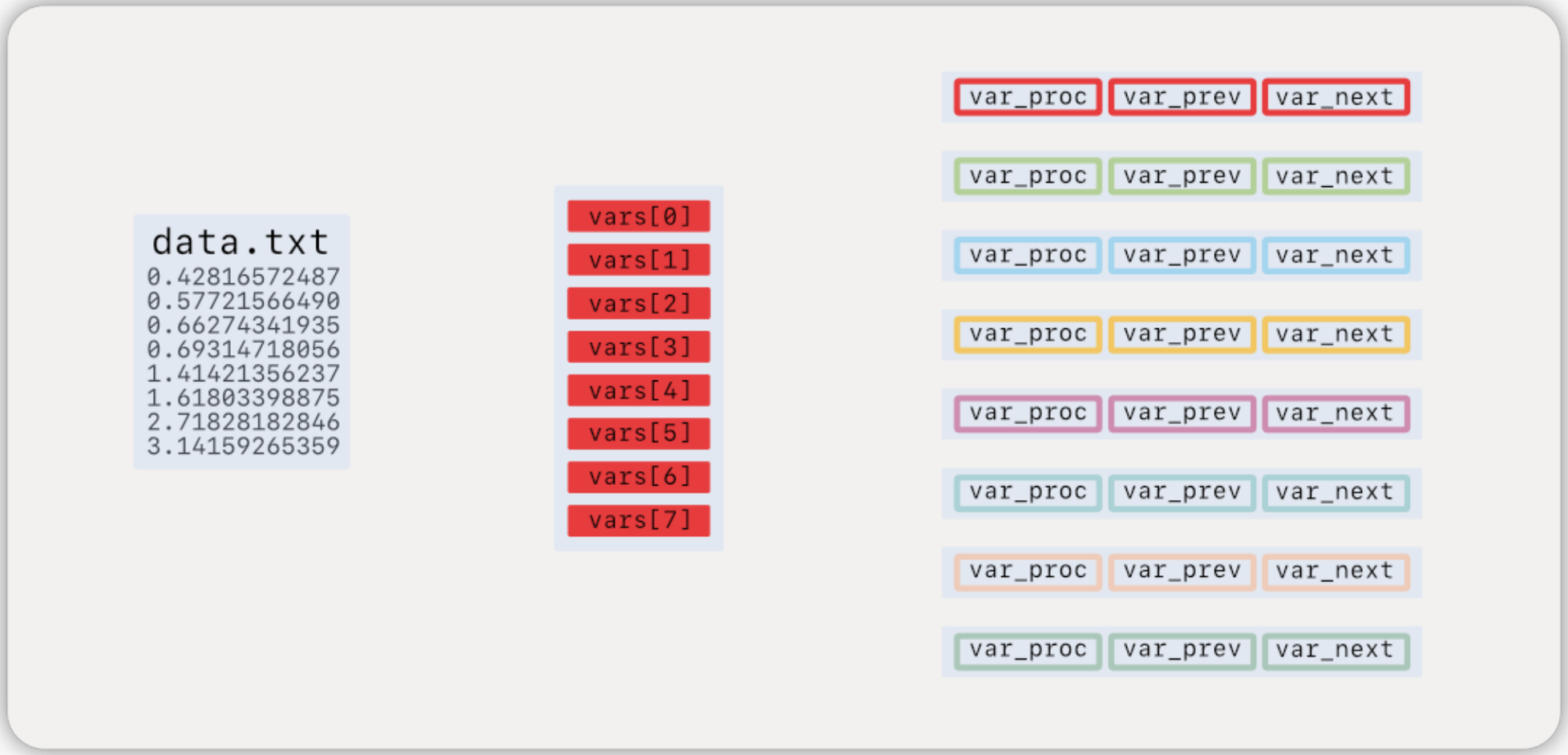
Exercises

Ex08



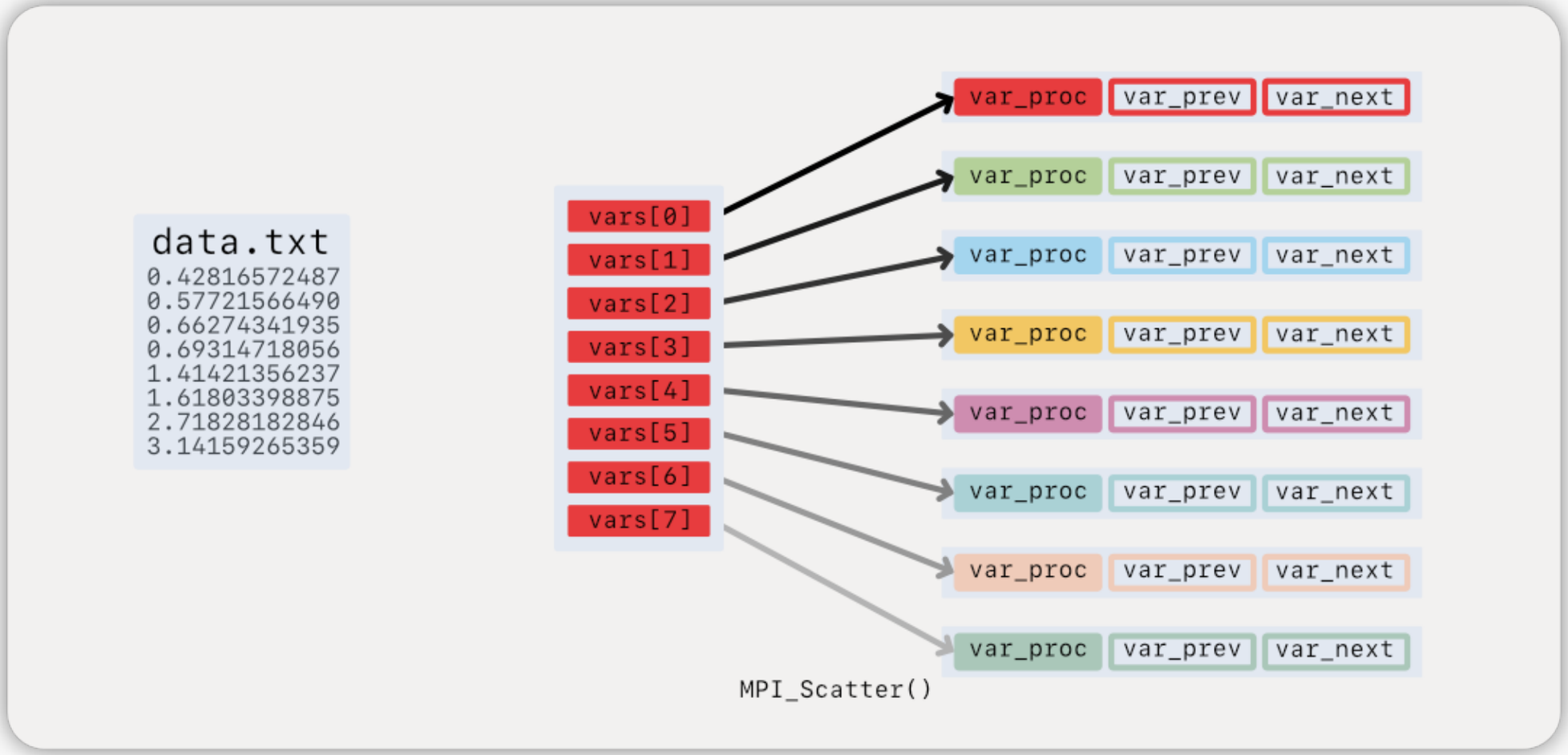
Exercises

Ex08



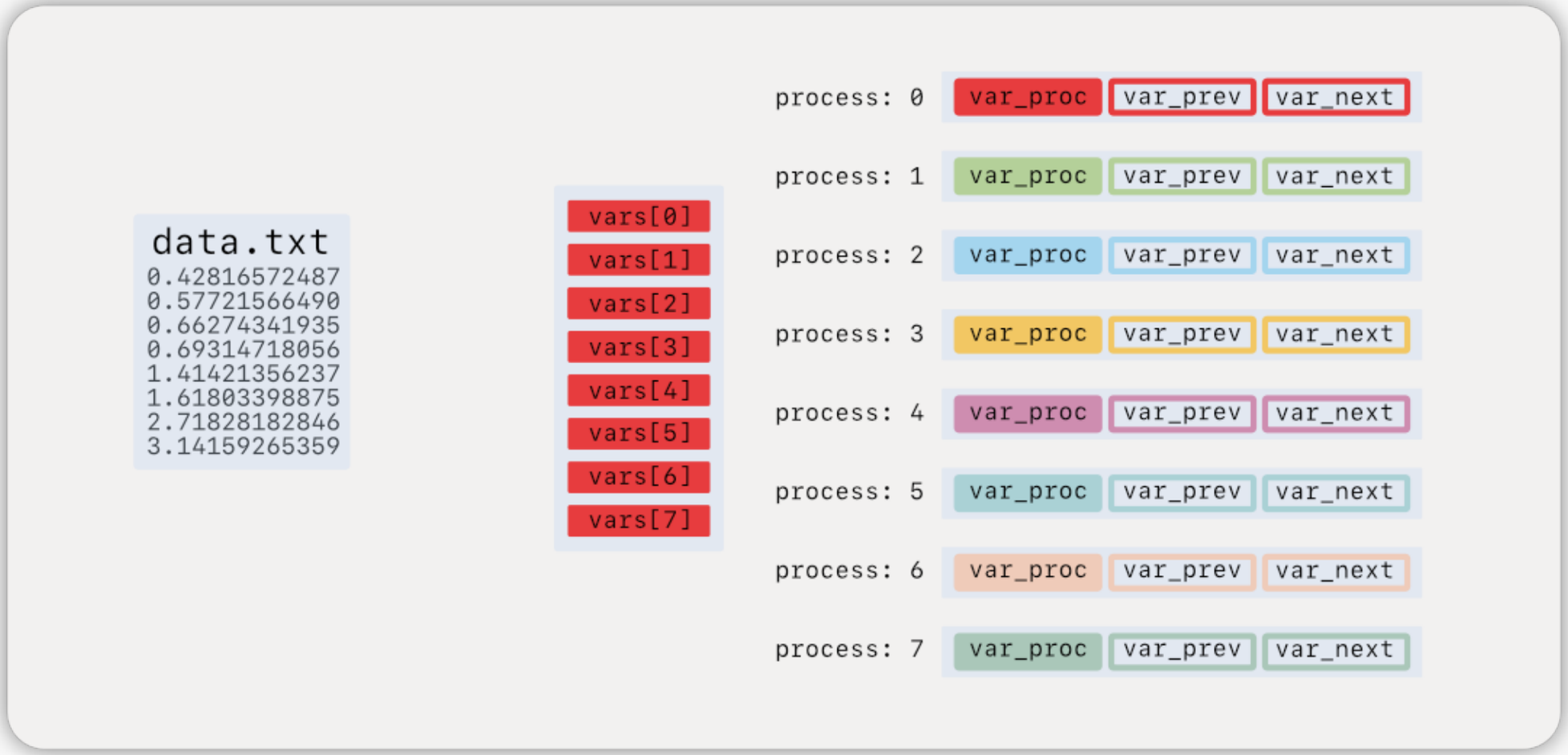
Exercises

Ex08



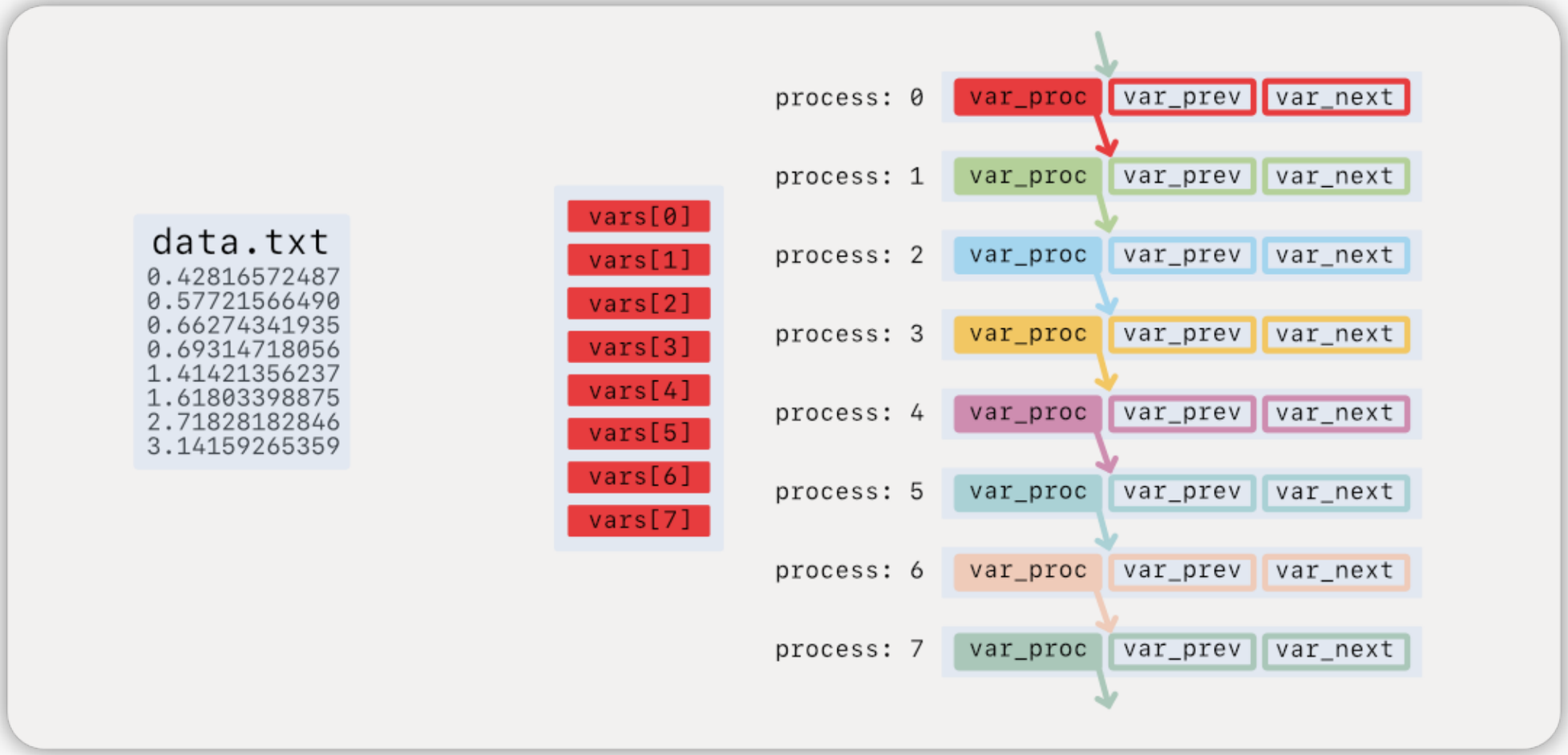
Exercises

Ex08



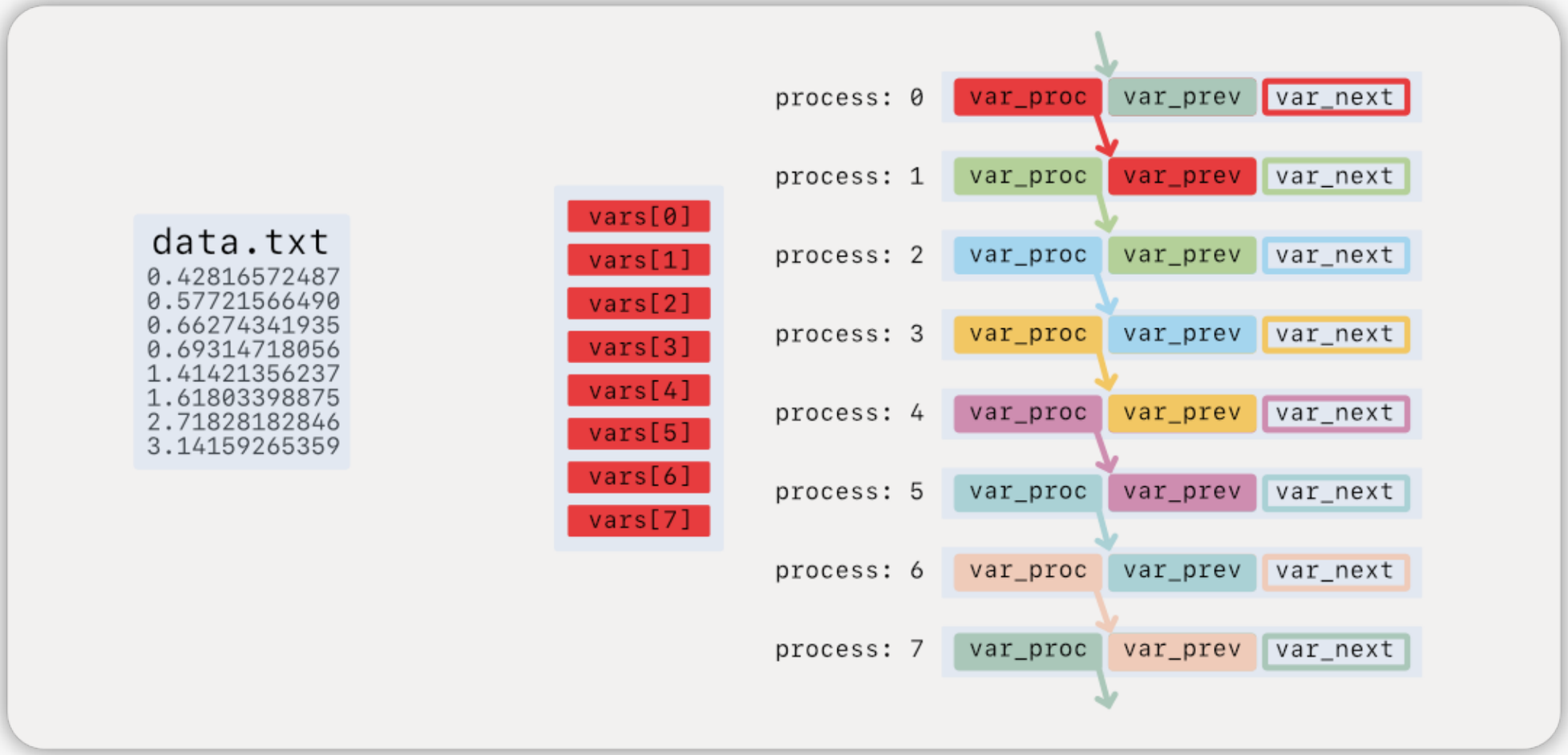
Exercises

Ex08



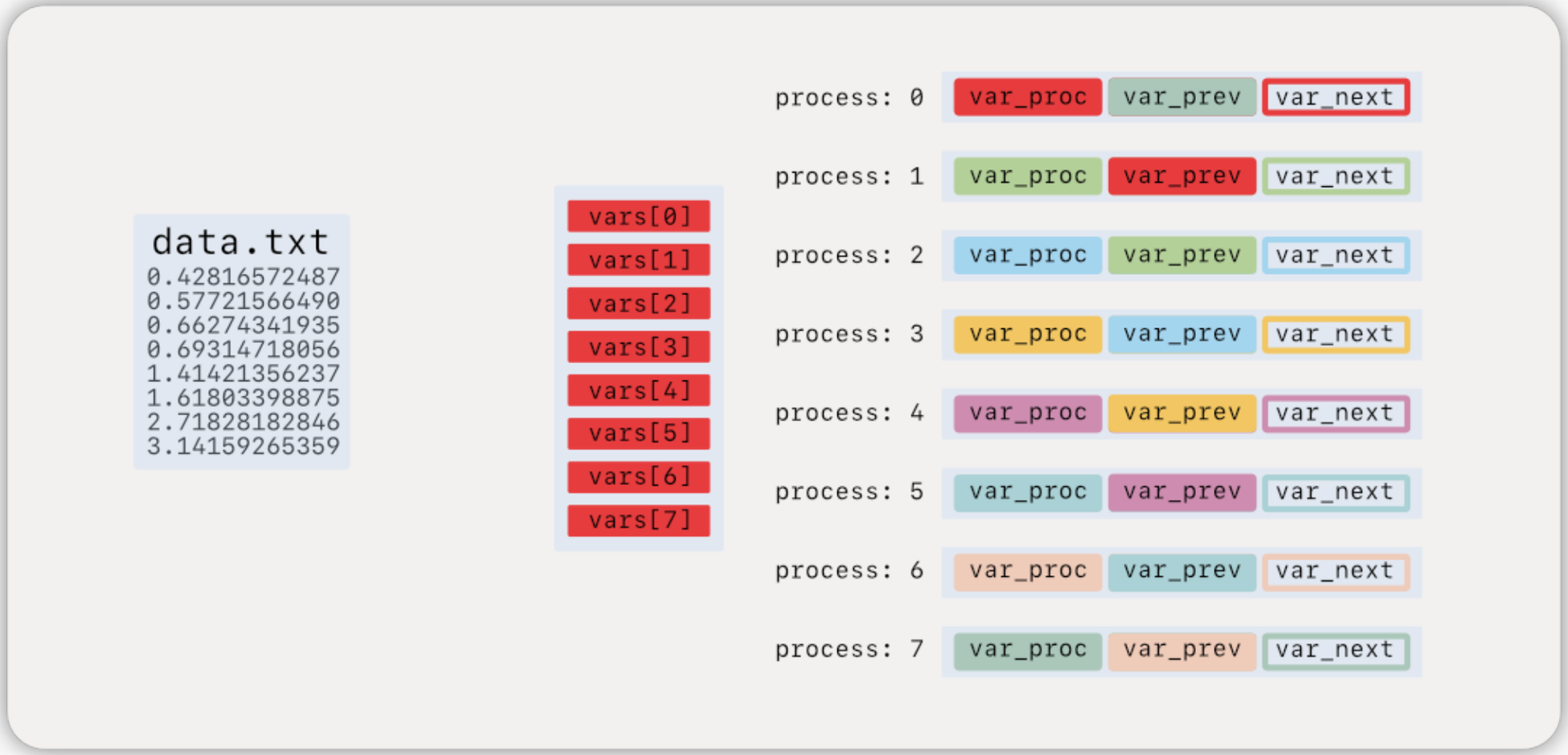
Exercises

Ex08



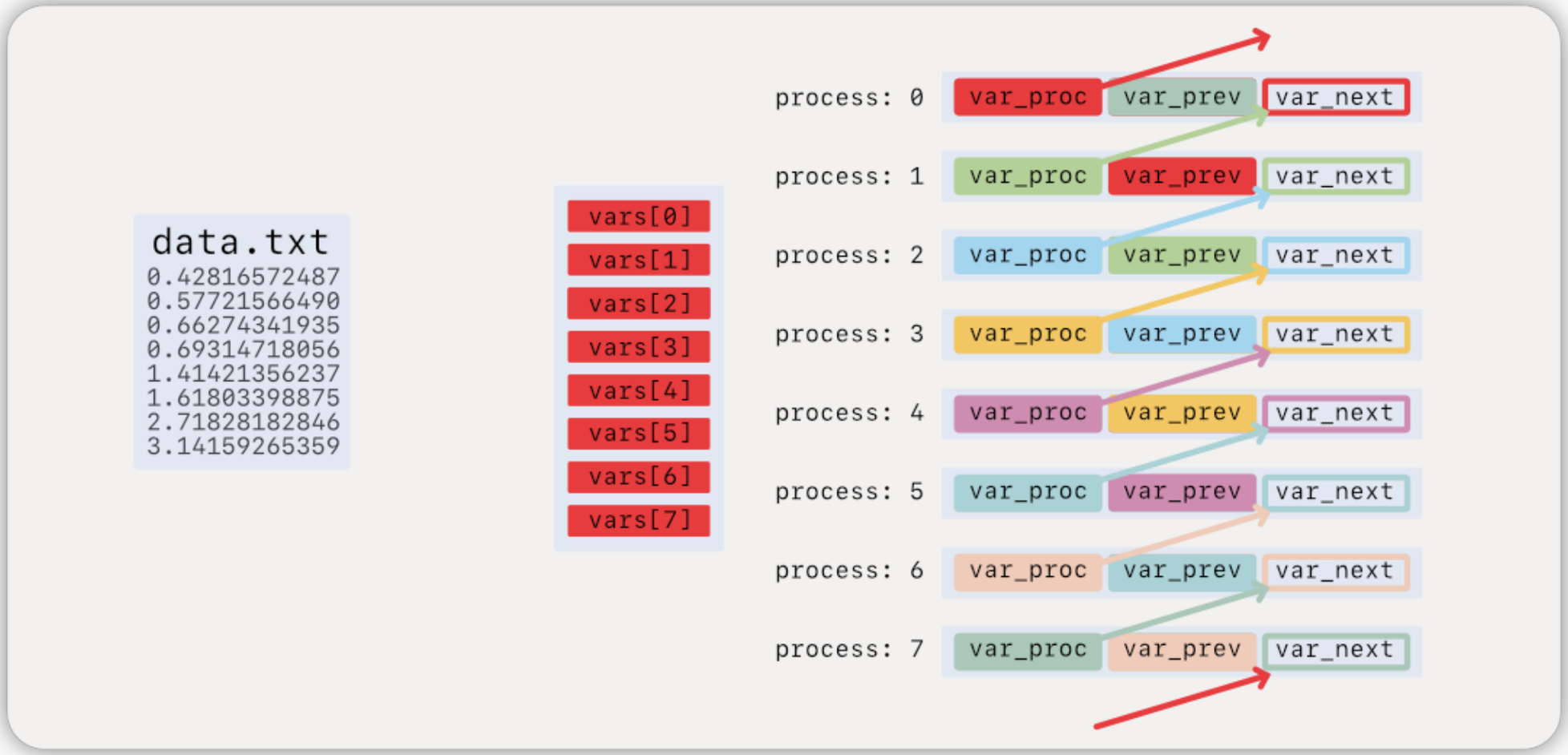
Exercises

Ex08



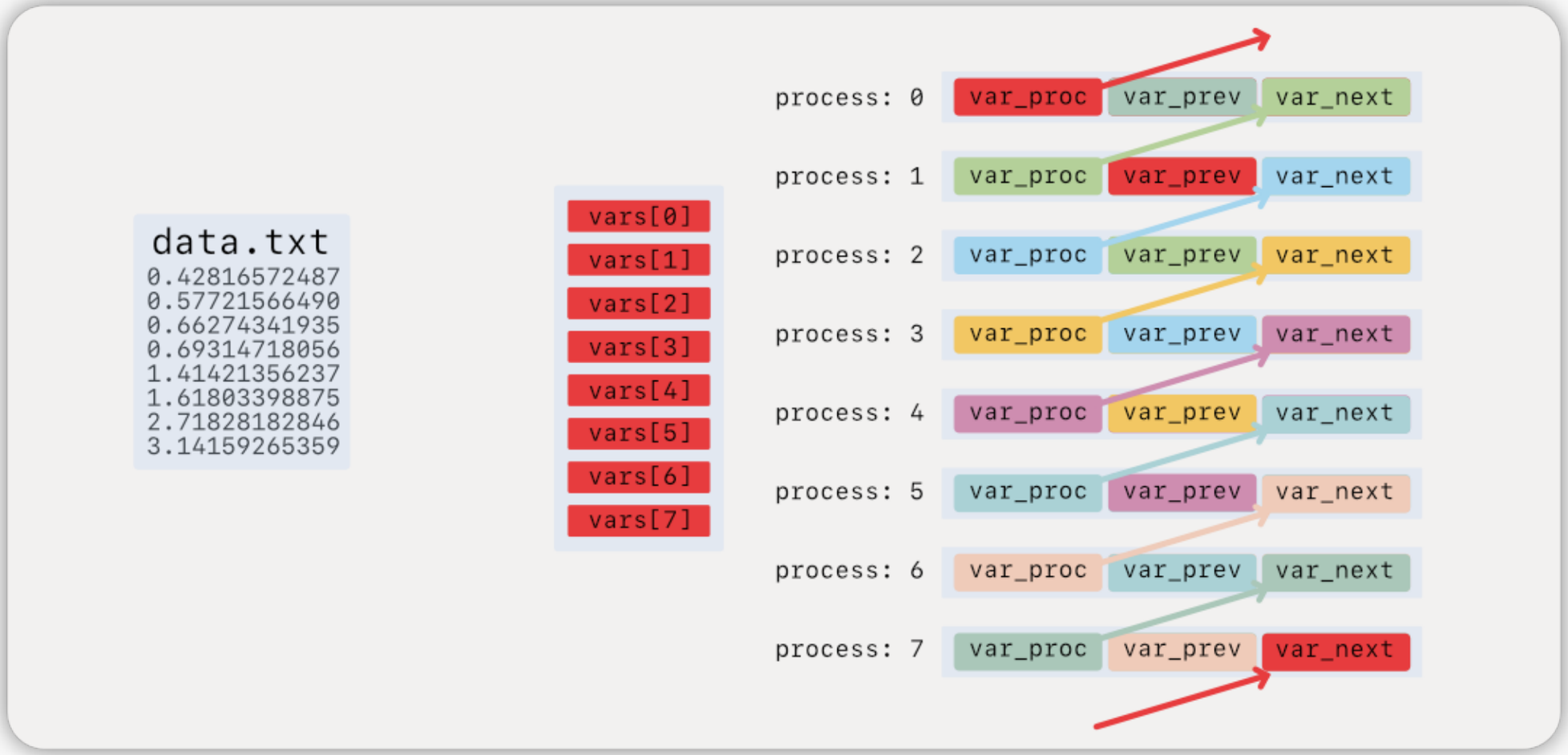
Exercises

Ex08



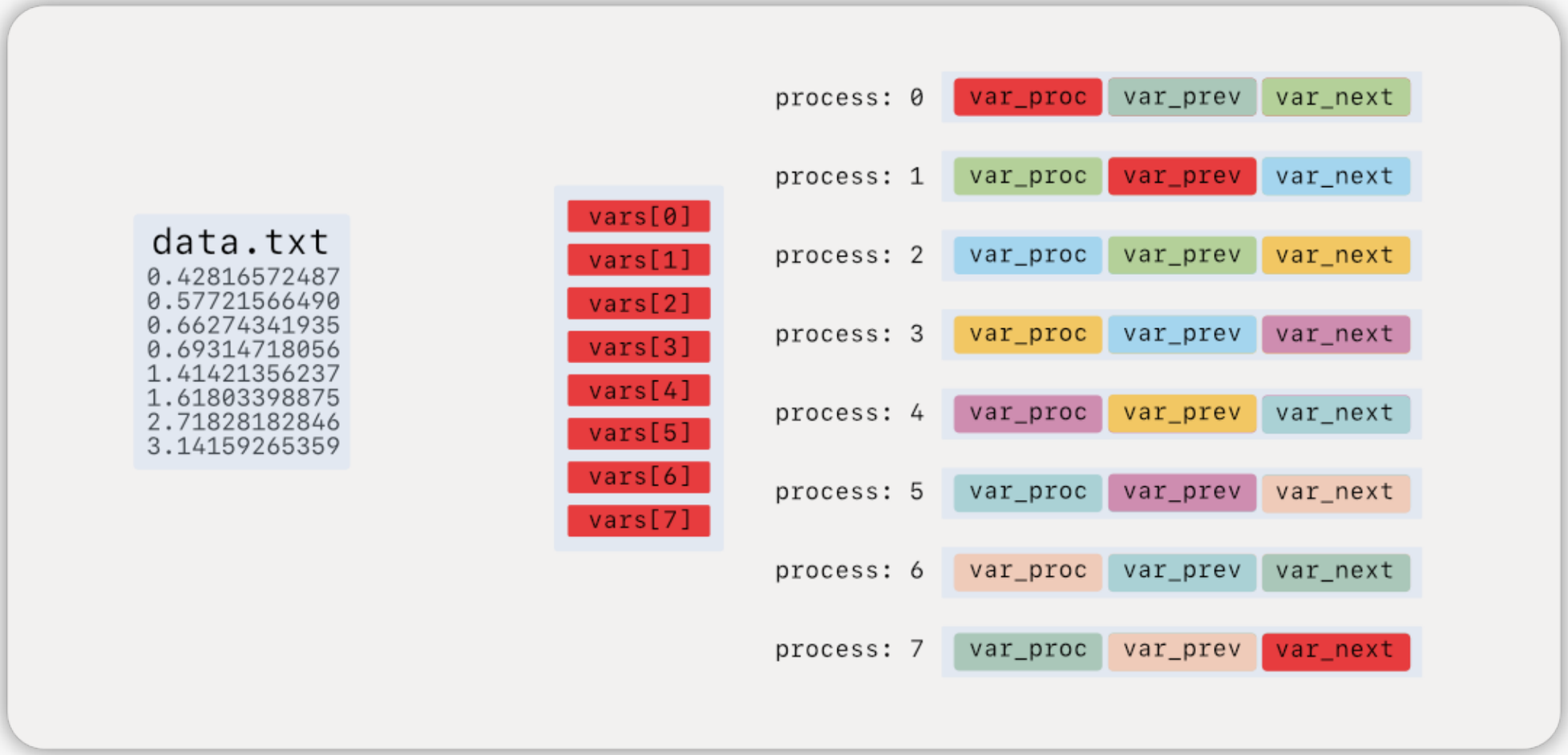
Exercises

Ex08



Exercises

Ex08



Exercises

Ex08

- The `TODO`s are for completing the arguments of the `MPI_Sendrecv()`s

```
double var_next, var_prev;
/*
 * TODO: Use `MPI_Sendrecv()' appropriately, so that `var_proc' of
 * each rank is copied into `var_prev' of the next rank. Assume
 * periodicity of ranks, i.e. if the sender is the last process
 * (`rank == nproc - 1') then send to the first process (`rank ==
 * 0')
 */
MPI_Sendrecv(/* TODO */);

/*
 * TODO: Use `MPI_Sendrecv()' appropriately, so that `var_proc' of
 * each rank is copied into `var_next' of the previous rank. Assume
 * periodicity of ranks, i.e. if the sender is the first process
 * (`rank == 0') then send to the last process (`rank == nproc - 1')
 */
MPI_Sendrecv(/* TODO */);
```

Exercises

Ex08

- The correct output should look like this:

```
[user@front01 ex08]$ cat ex08-output.txt | sort
This is rank = 0 of nproc = 8 on node: cn01 | var_proc = 0.428166   var_prev = 3.141593   var_next = 0.577216
This is rank = 1 of nproc = 8 on node: cn01 | var_proc = 0.577216   var_prev = 0.428166   var_next = 0.662743
This is rank = 2 of nproc = 8 on node: cn01 | var_proc = 0.662743   var_prev = 0.577216   var_next = 0.693147
This is rank = 3 of nproc = 8 on node: cn01 | var_proc = 0.693147   var_prev = 0.662743   var_next = 1.414214
This is rank = 4 of nproc = 8 on node: cn02 | var_proc = 1.414214   var_prev = 0.693147   var_next = 1.618034
This is rank = 5 of nproc = 8 on node: cn02 | var_proc = 1.618034   var_prev = 1.414214   var_next = 2.718282
This is rank = 6 of nproc = 8 on node: cn02 | var_proc = 2.718282   var_prev = 1.618034   var_next = 3.141593
This is rank = 7 of nproc = 8 on node: cn02 | var_proc = 3.141593   var_prev = 2.718282   var_next = 0.428166
[user@front01 ex08]$
```