HPC Intermediate Training Event

EuroCC Training, 19th April 2021

MPI, OpenMP, and Hybrid Programming

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:
    - version 3.1 (MPI-3.1), since June 2015
    - Release Candidate for version 4.0 (MPI-4.0) as of November 2020
- Several free (open) or vendor-provided implementations, e.g.:
    - Mvapich
    - OpenMPI
    - IntelMPI

The Message Passing Interface

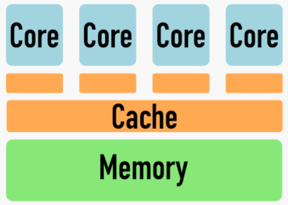
- MPI: An Application Programmer Interface (API)
  - A *library specification*; determines functions, their names and arguments, and their functionality
- A *de facto* standard for programming *distributed memory* systems
- Current specification:
  - version 3.1 (MPI-3.1), since June 2015
  - Release Candidate for version 4.0 (MPI-4.0) as of November 2020
- 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

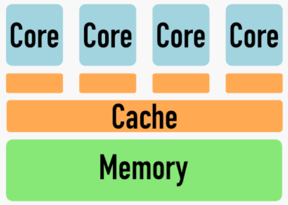
Shared memory



- 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)

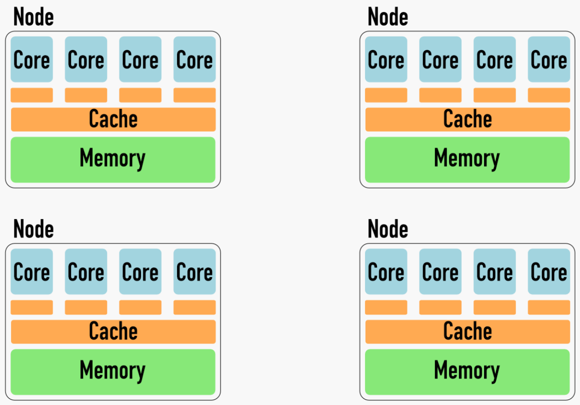
Shared vs Distributed memory paradigm

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)

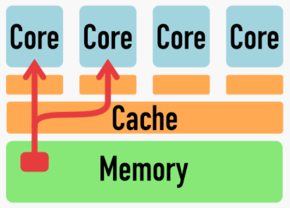
Distributed memory



- 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

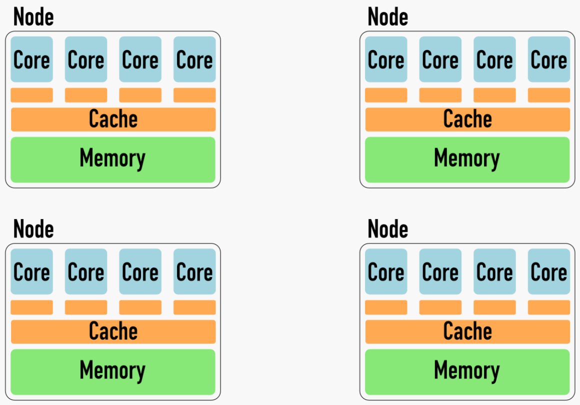
Shared memory



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)

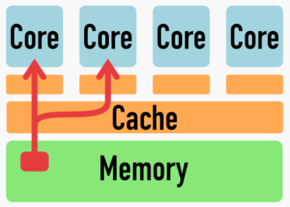
Distributed memory



- 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

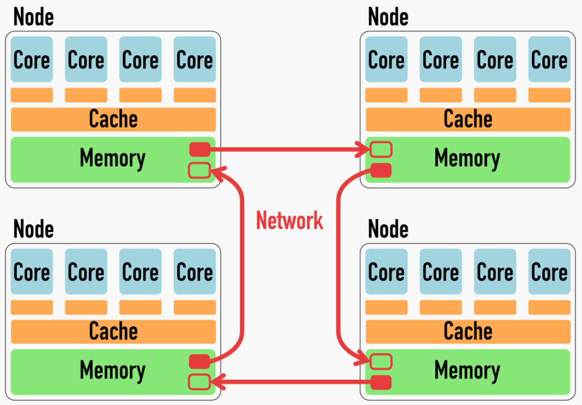
Shared memory



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

```
#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:

```
#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 $n_{\mathrm{proc}} - 1$ that identifies the calling process
- A complete example:

```
#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:

```
#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
```

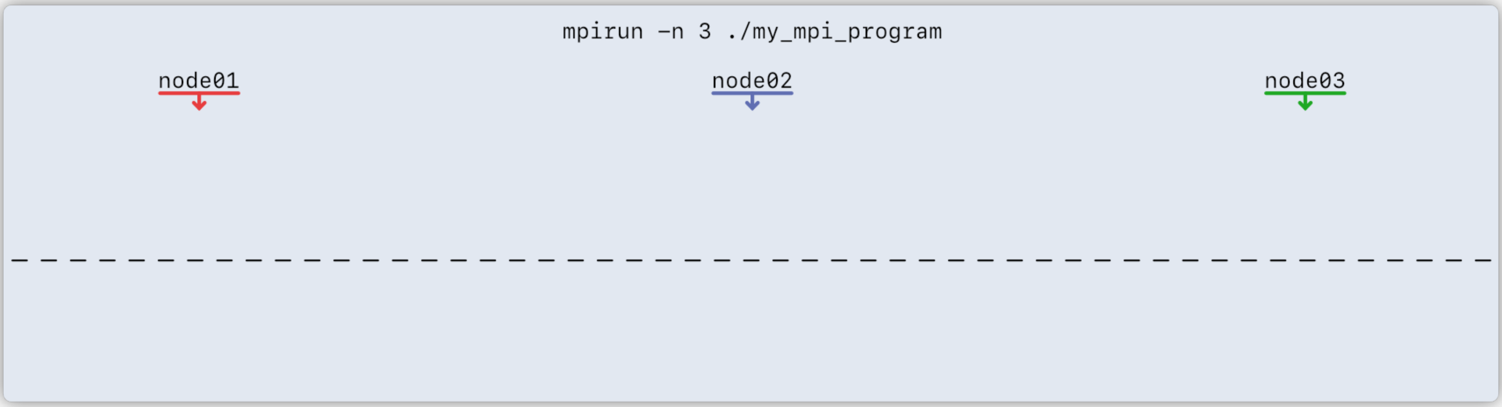
- Note that the order is random; Synchronization between processes can be achieved using the `MPI_Barrier()` function
- All processes must call `MPI_Barrier()`
- For any process to exit the barrier, all processes must have entered the barrier first

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

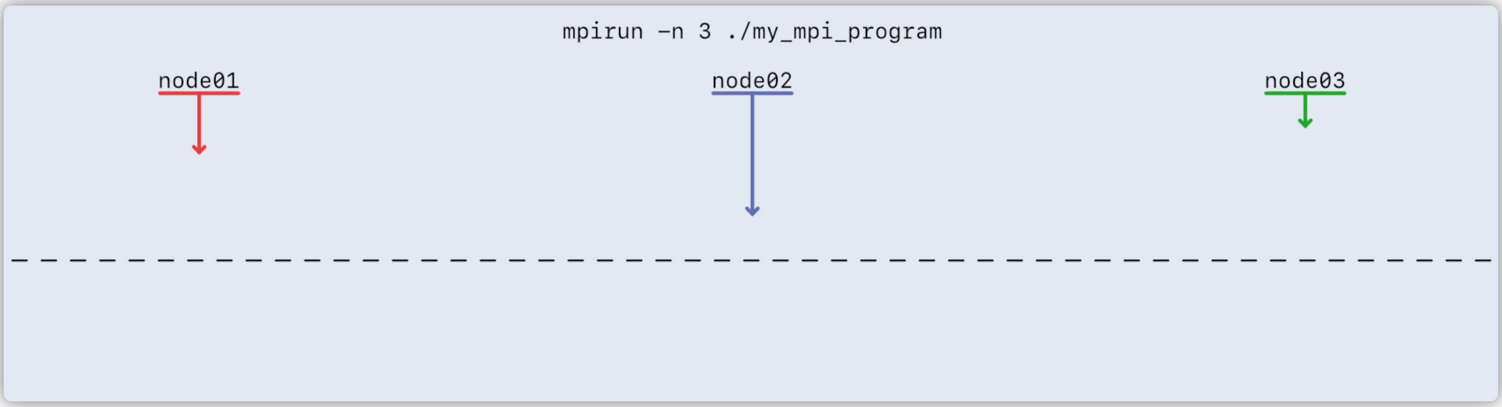


Synchronization

- 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

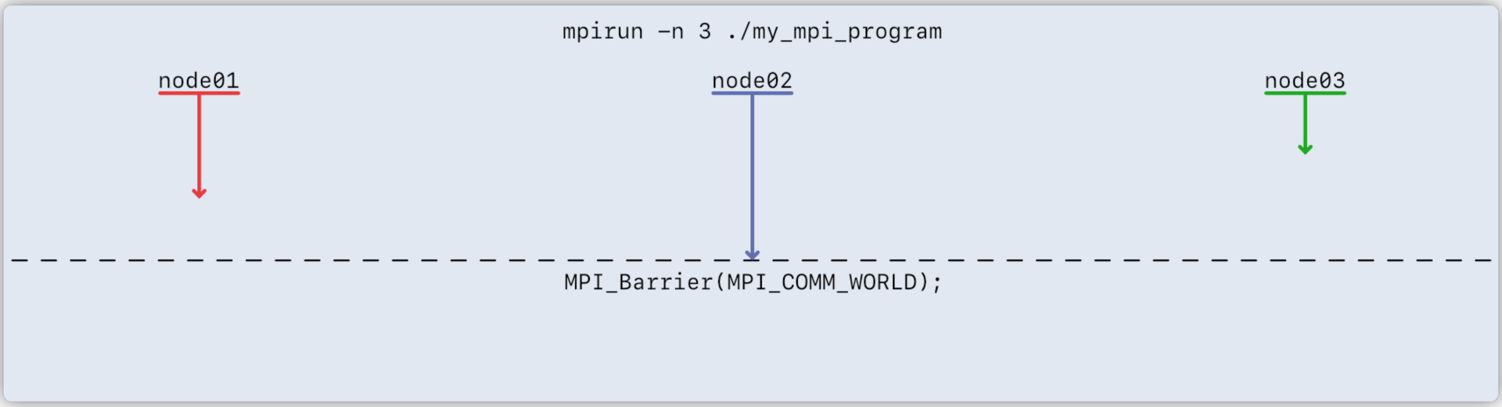


Synchronization

- 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

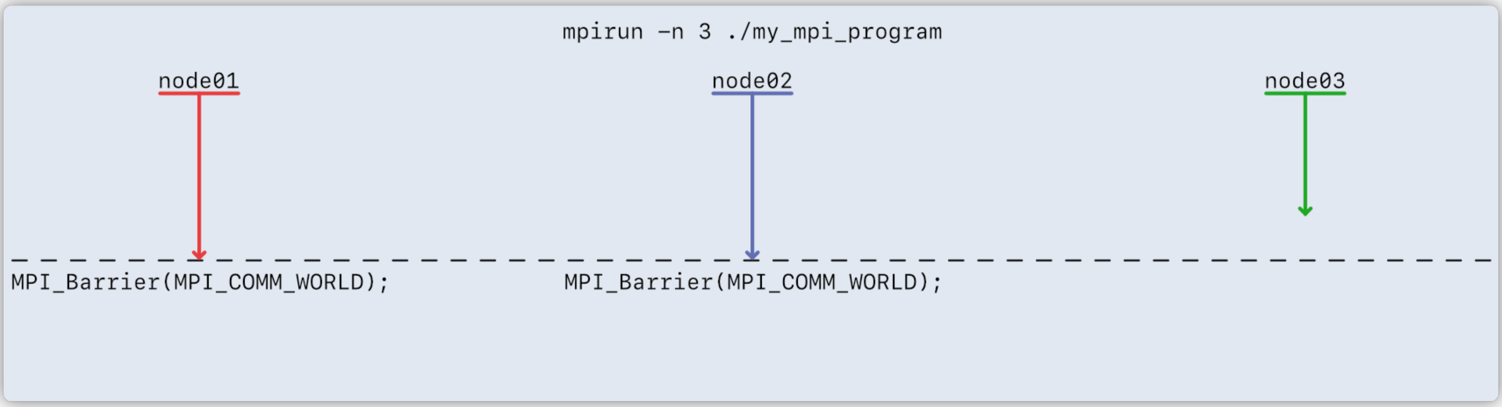


Synchronization

- 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

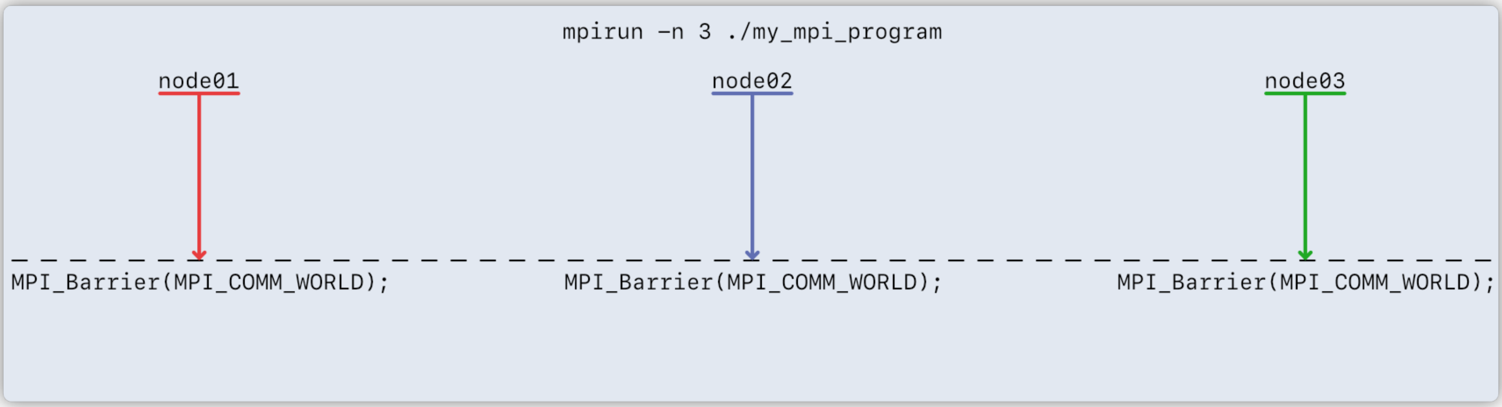

mpirun -n 3 ./my_mpi_program
node01
node02
node03
MPI_Barrier(MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);

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

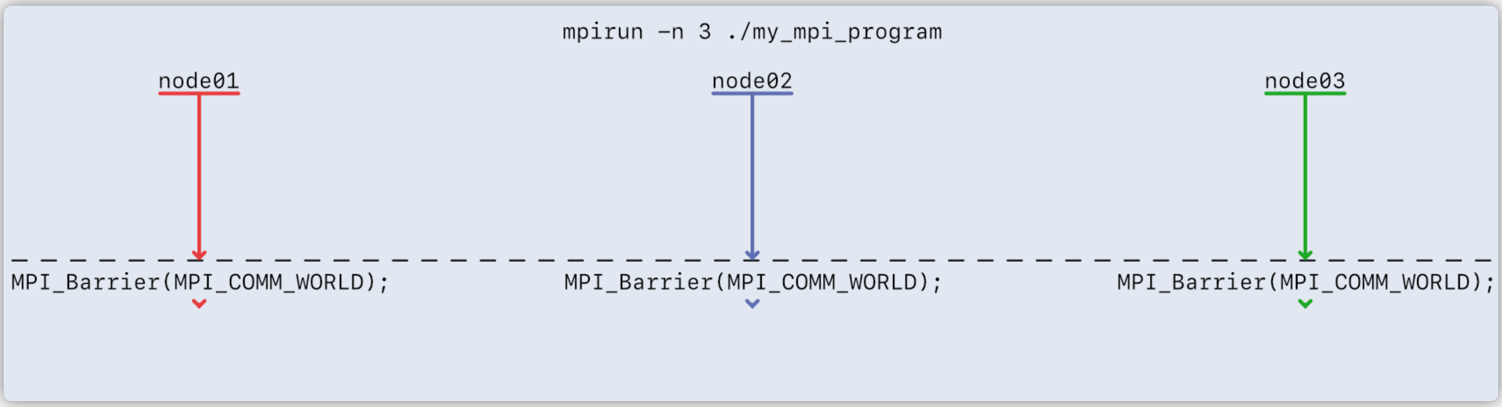

mpirun -n 3 ./my_mpi_program
node01
node02
node03
MPI_Barrier(MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);

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

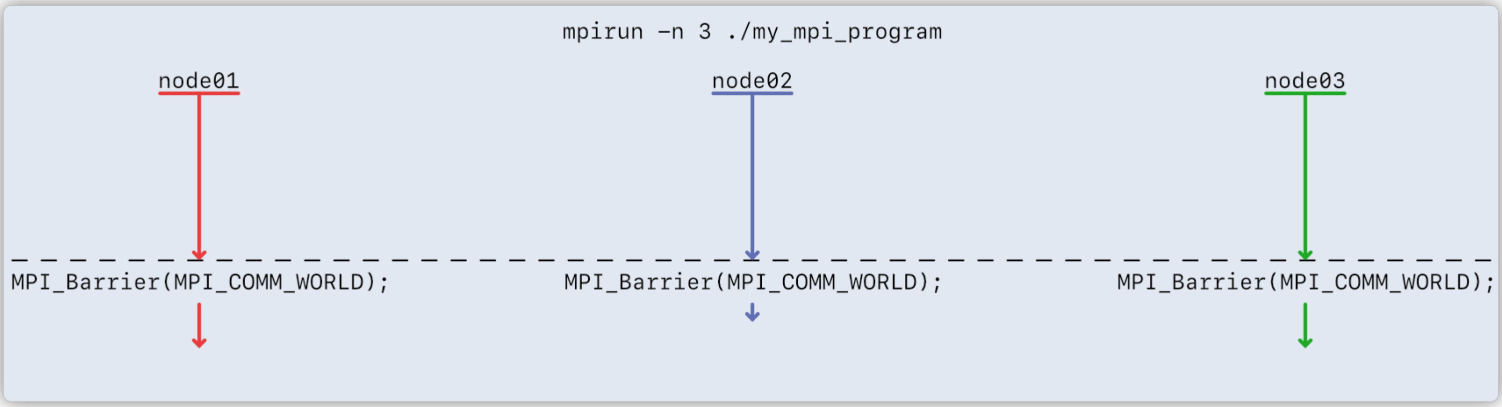

mpirun -n 3 ./my_mpi_program
node01
node02
node03
MPI_Barrier(MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);

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

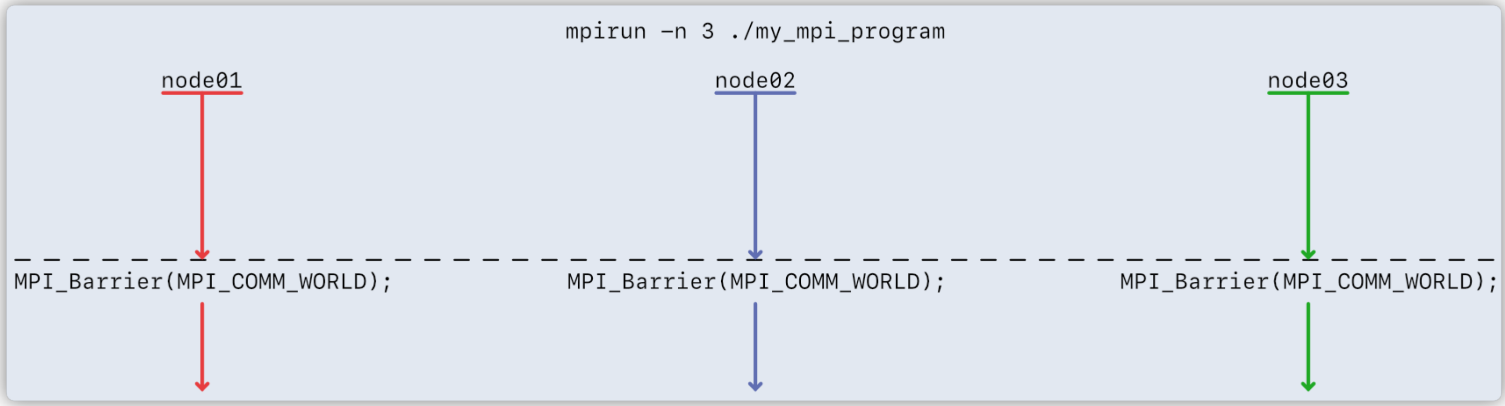

mpirun -n 3 ./my_mpi_program
node01
node02
node03
MPI_Barrier(MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);

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


mpirun -n 3 ./my_mpi_program
node01
node02
node03
MPI_Barrier(MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);

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:

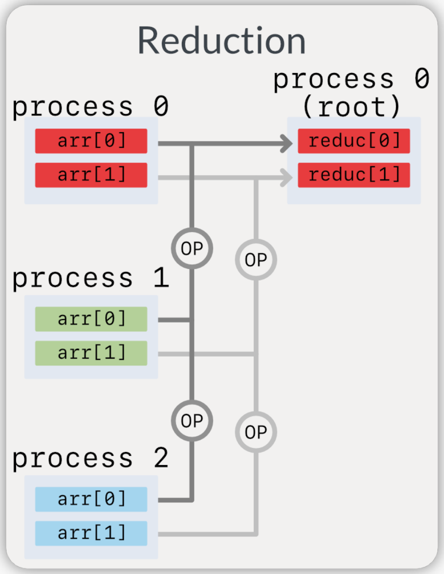
```
MPI_Reduce(
        const void *sendbuf, void *recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, int root,
        MPI_Comm comm
        )
```

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`



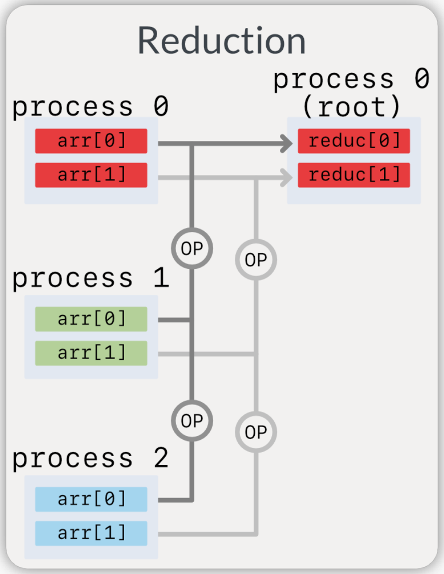
Collective operations: Reduction

- 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:

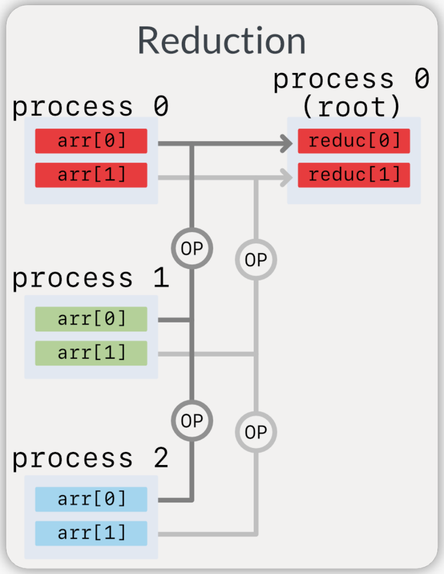
```
MPI_Reduce(
        const void *sendbuf, void *recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, int root,
        MPI_Comm comm
        )
```

- *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

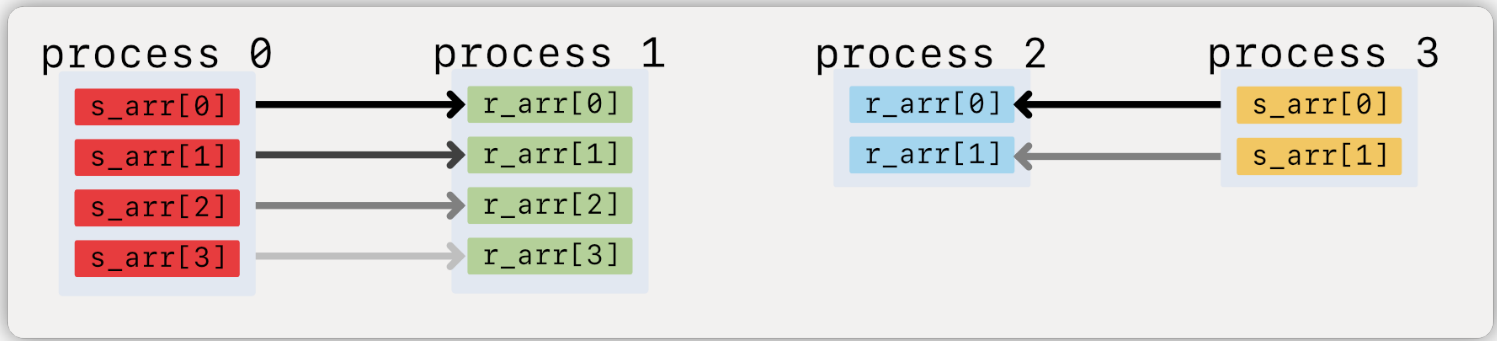
- Communications that involve transfer of data between two processes
- Most common case: send/receive
    - The sender process issues a send operation
    - The receiver process posts a receive operation

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

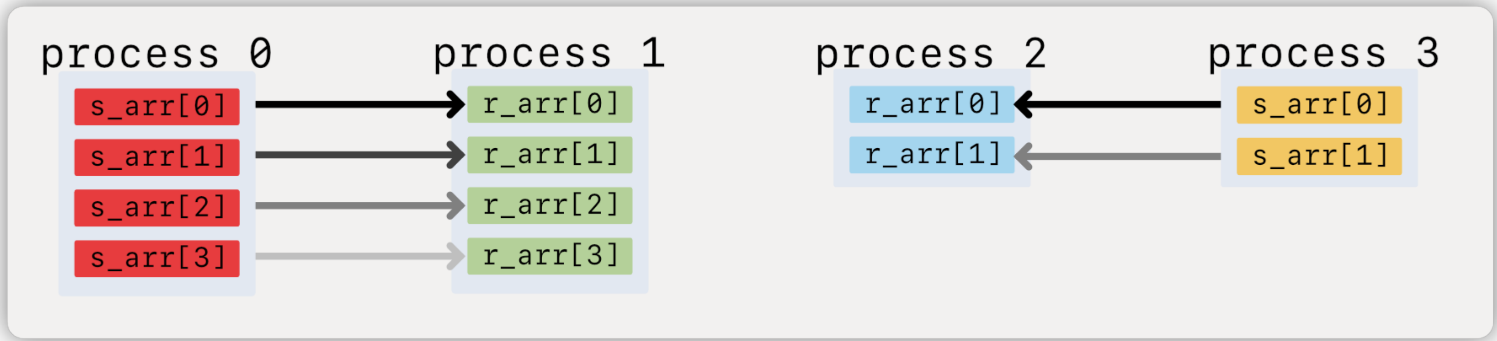
Point-to-point communication

- 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

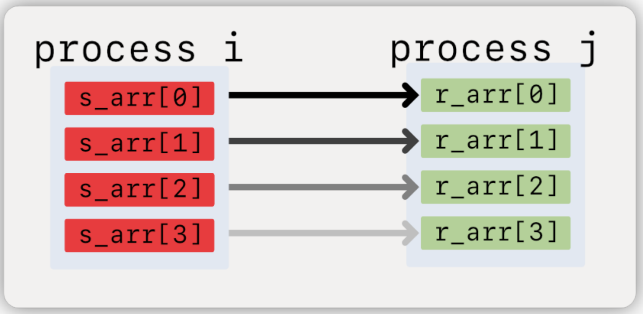
```
MPI_Send(void *buf, int n, MPI_Datatype type, int dest, int tag, MPI_Comm comm)
MPI_Recv(void *buf, int n, MPI_Datatype type, int srce, int tag, MPI_Comm comm, MPI_Status *status)
```

- 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

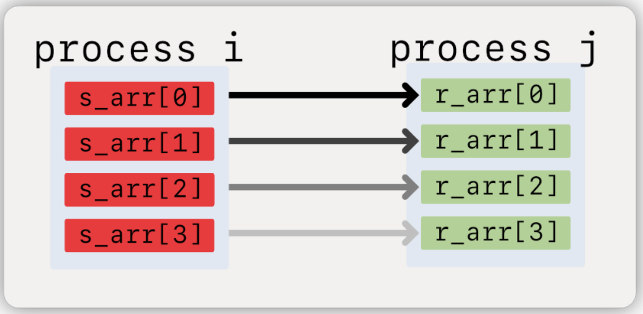
Point-to-point communication

- Send/Receive; a trivial example



Point-to-point communication

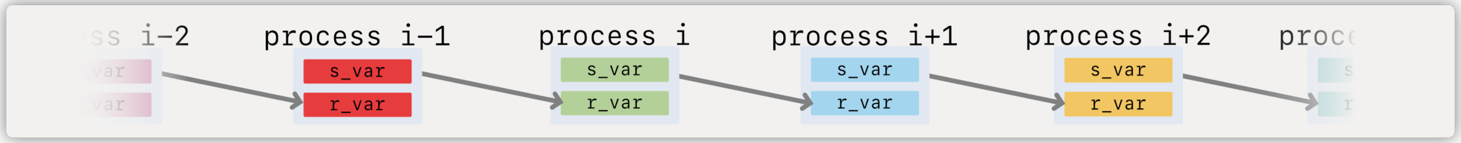
- Send/Receive; a trivial example



```
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);
}
```

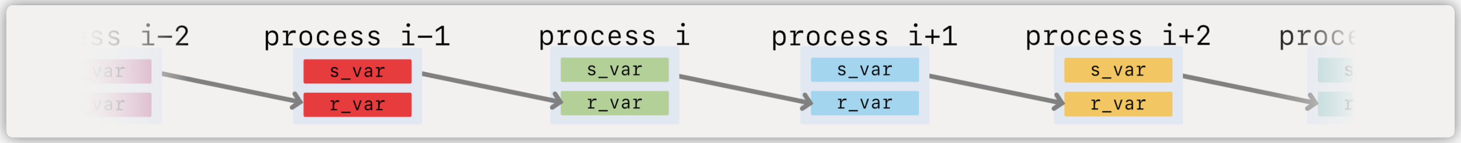
Point-to-point communication

- 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

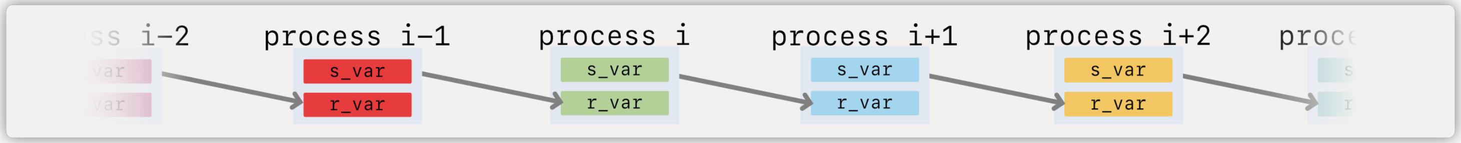


- This will not work:

```
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



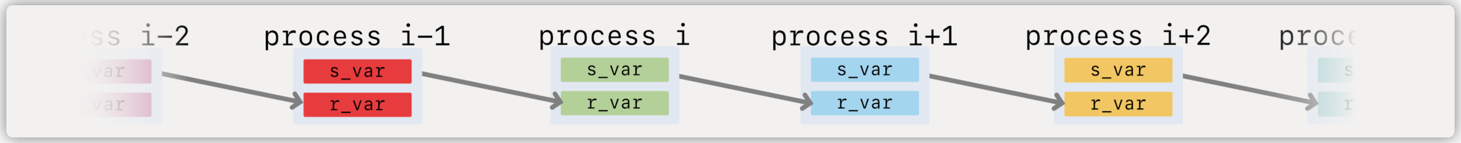
- This will not work:

```
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



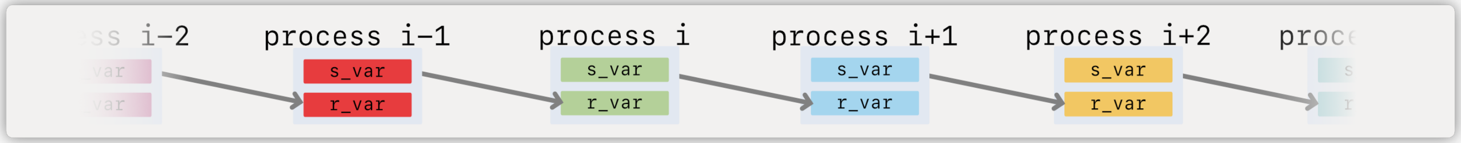
- This will not work:

```
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



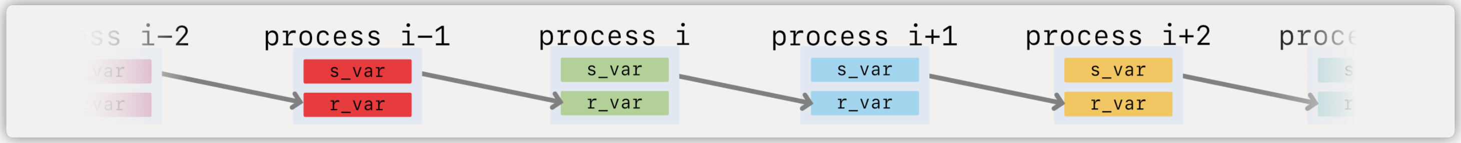
- This will not work:

```
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



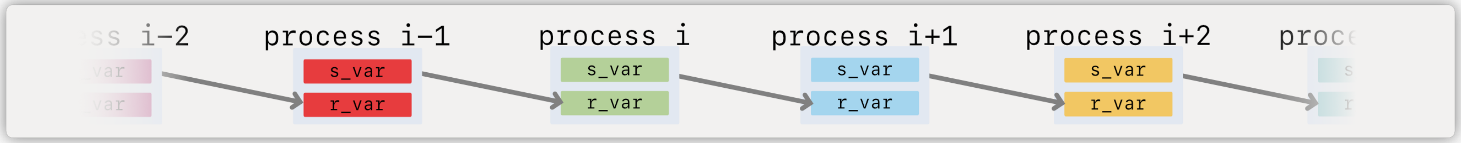
- This will not work:

```
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

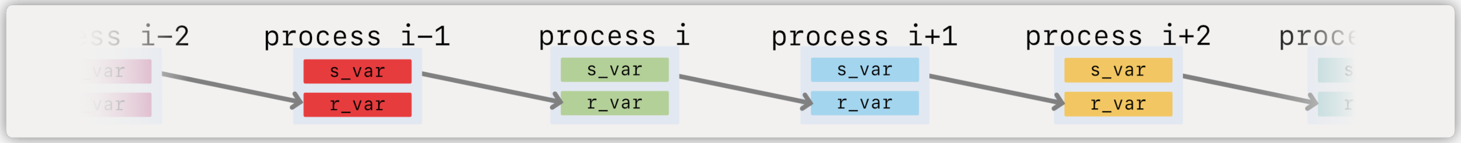


- A more efficient and elegant solution is to use `MPI_Sendrecv()`:

```
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

```
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`:

```
/*
 * 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

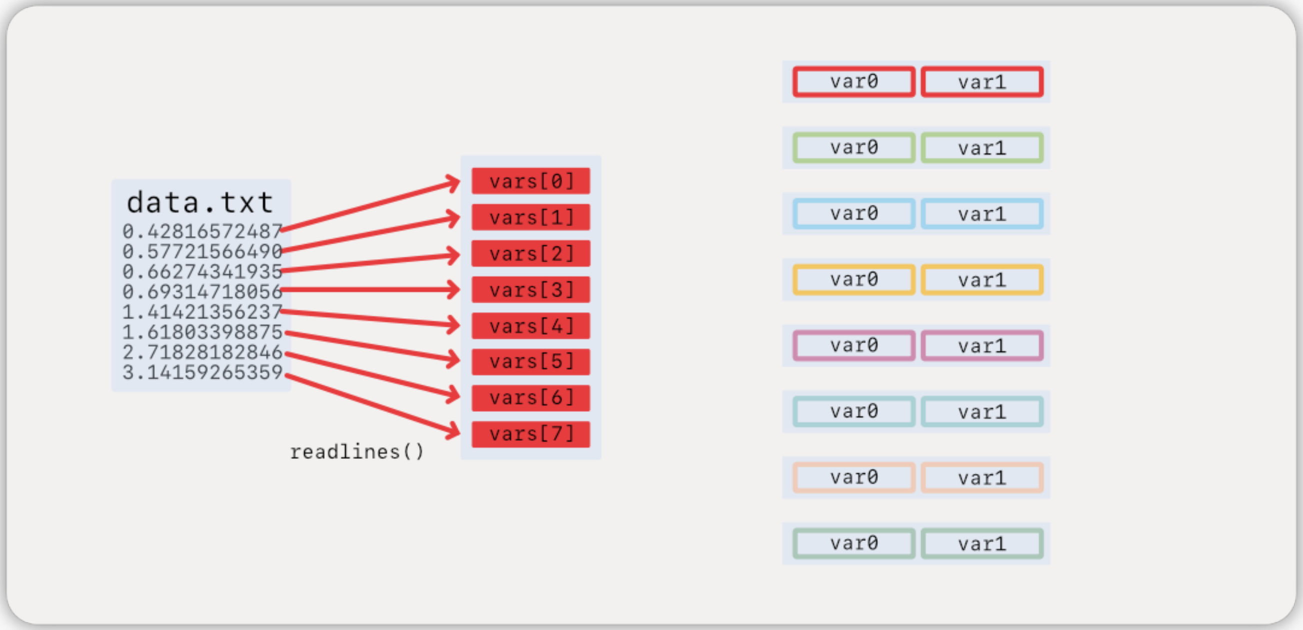
- This exercise demonstrates `MPI_Send()` and `MPI_Recv()`
- `data.txt` is the same as in `ex04`, with eight elements:

```
[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

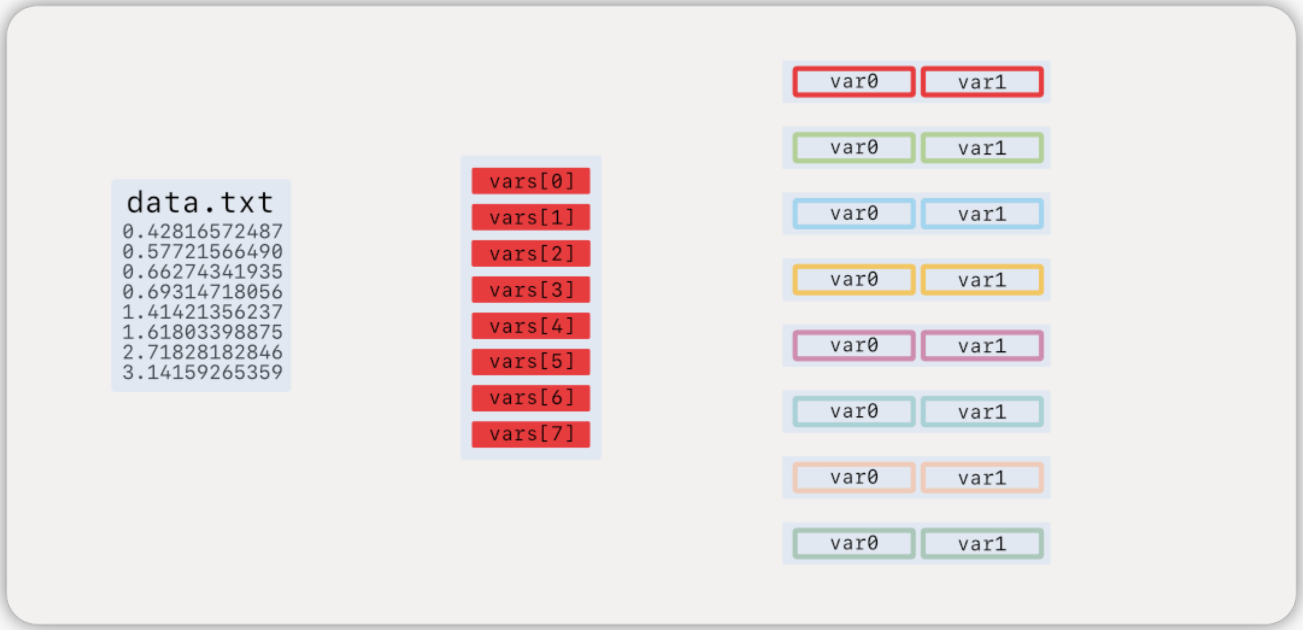
Exercises

Ex07



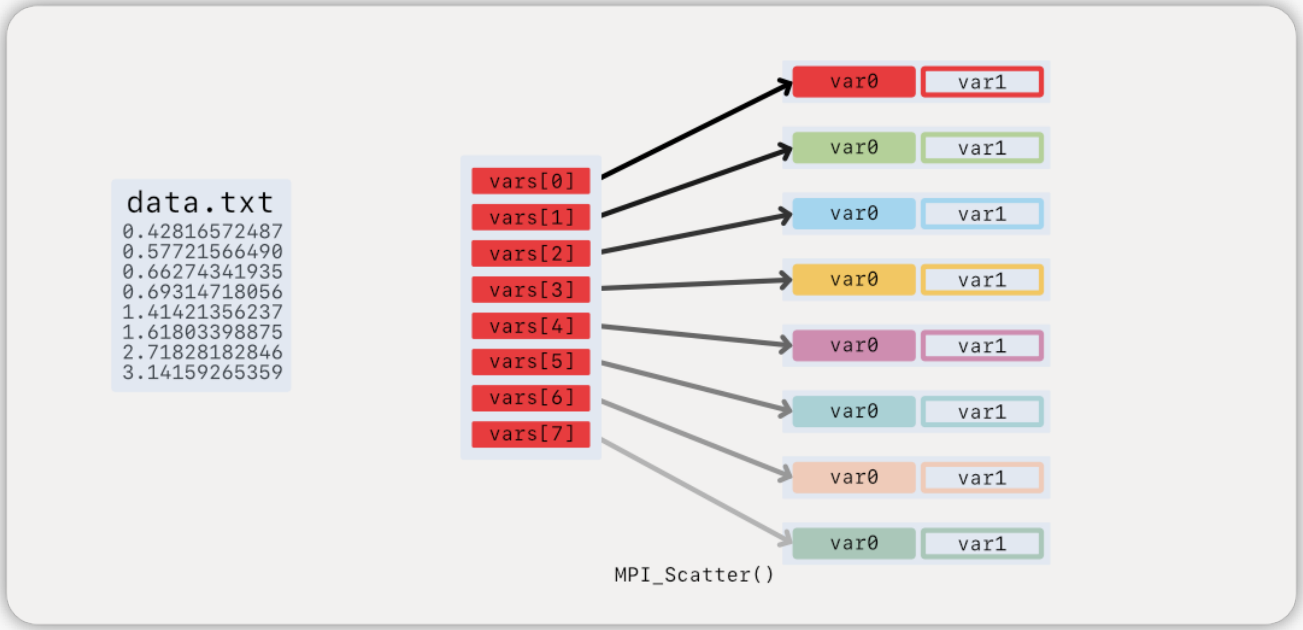
Exercises

Ex07



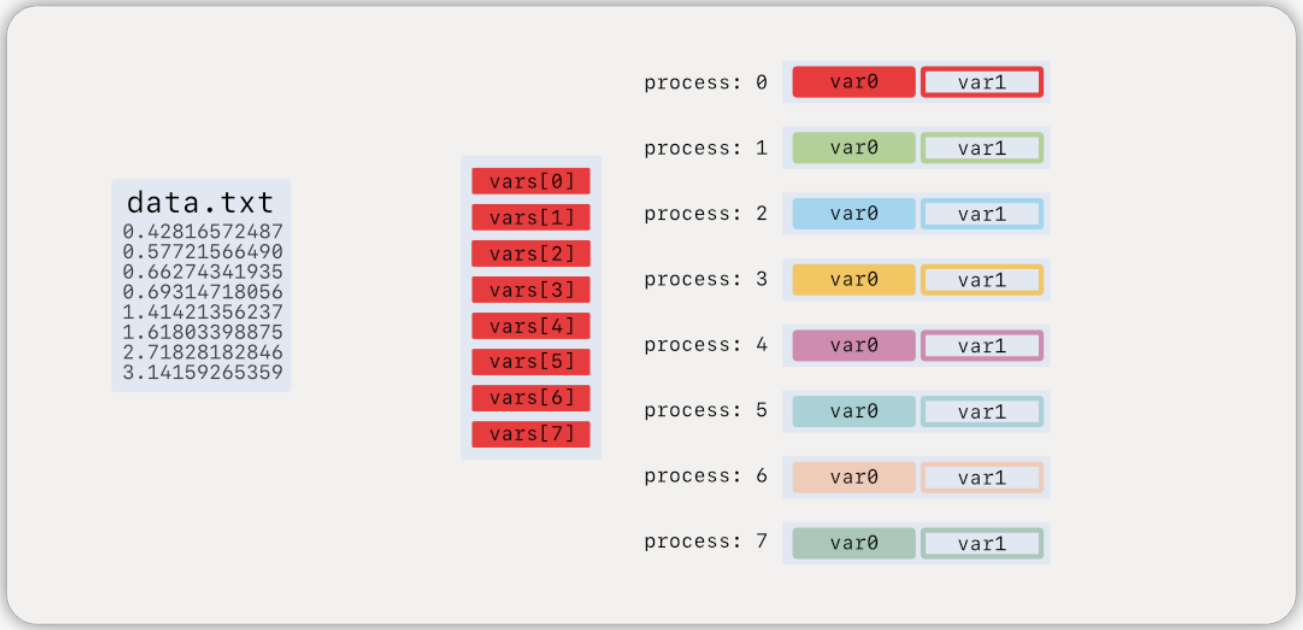
Exercises

Ex07



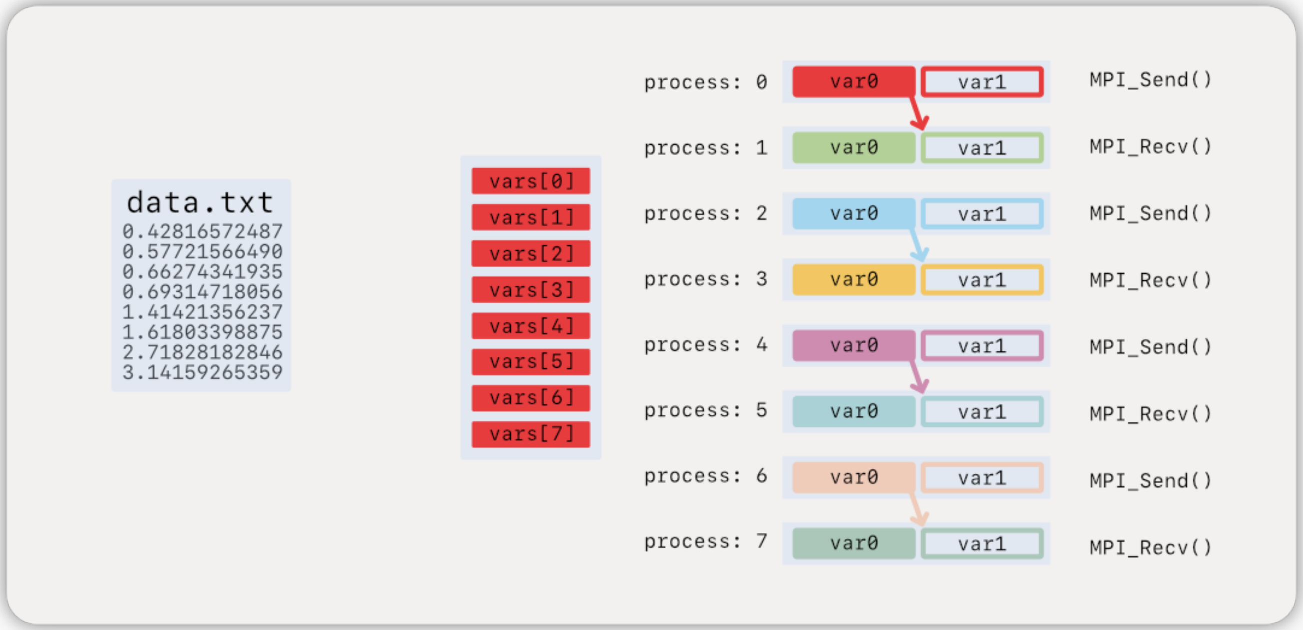
Exercises

Ex07



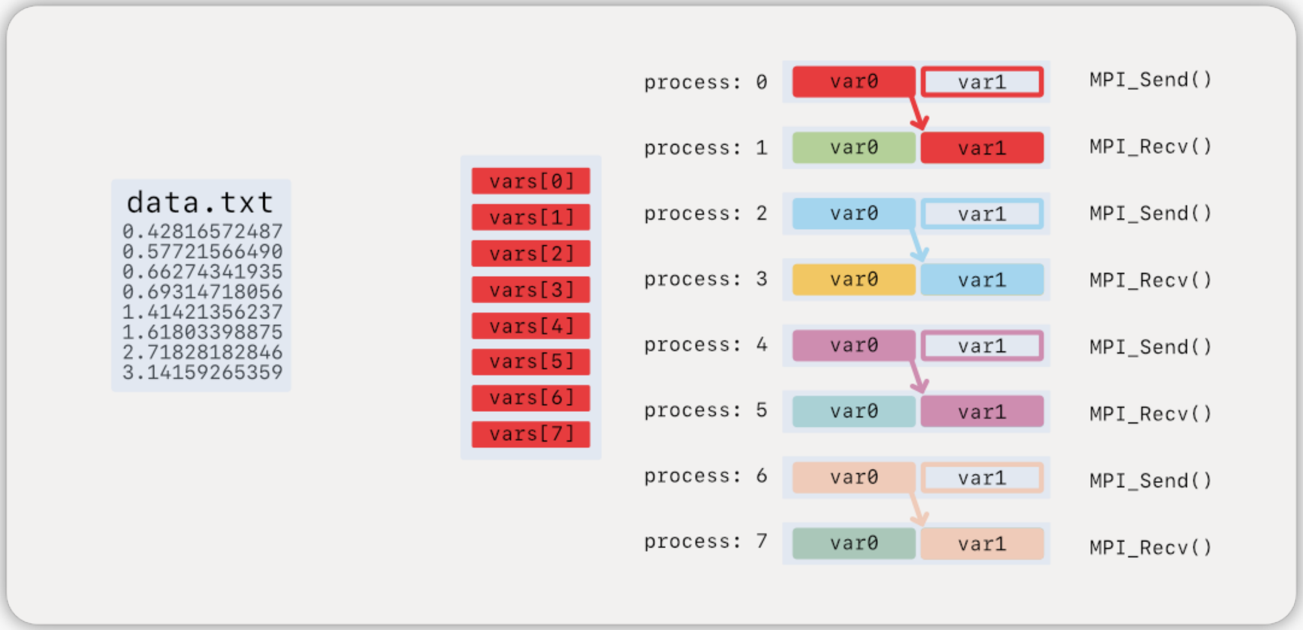
Exercises

Ex07



Exercises

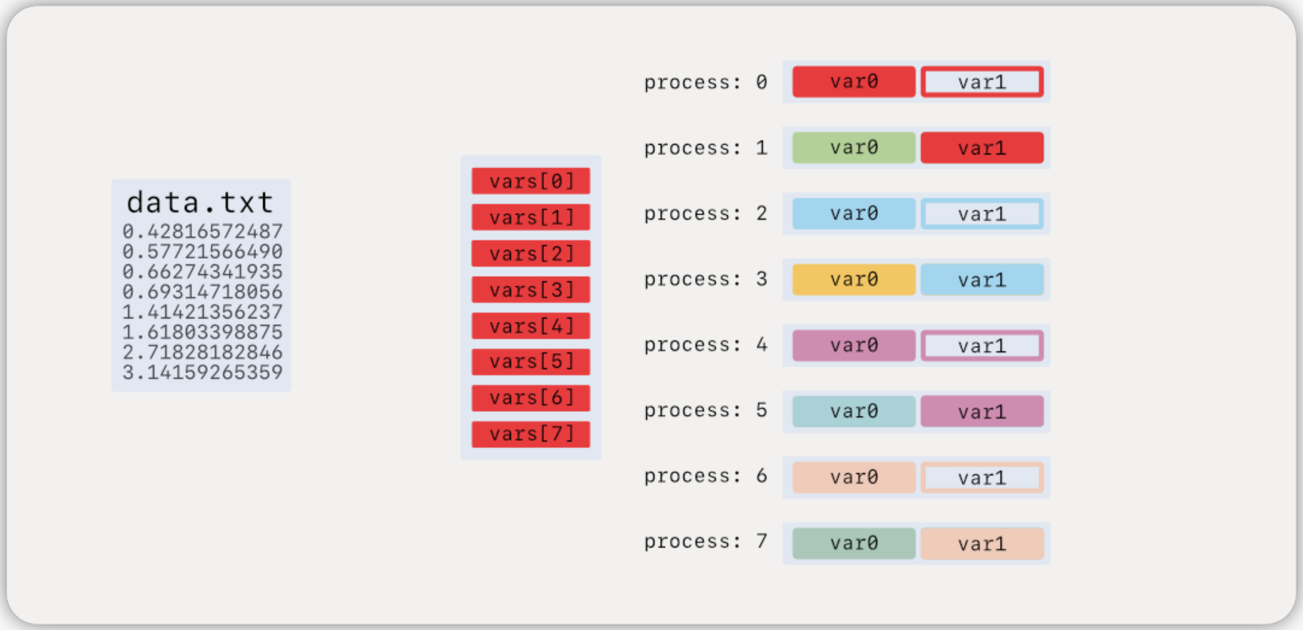
Ex07



data.txt
0.42816572487
0.57721566490
0.66274341935
0.69314718056
1.41421356237
1.61803398875
2.71828182846
3.14159265359
vars[0]
vars[1]
vars[2]
vars[3]
vars[4]
vars[5]
vars[6]
vars[7]
process: 0 var0 var1 MPI_Send()
process: 1 var0 var1 MPI_Recv()
process: 2 var0 var1 MPI_Send()
process: 3 var0 var1 MPI_Recv()
process: 4 var0 var1 MPI_Send()
process: 5 var0 var1 MPI_Recv()
process: 6 var0 var1 MPI_Send()
process: 7 var0 var1 MPI_Recv()

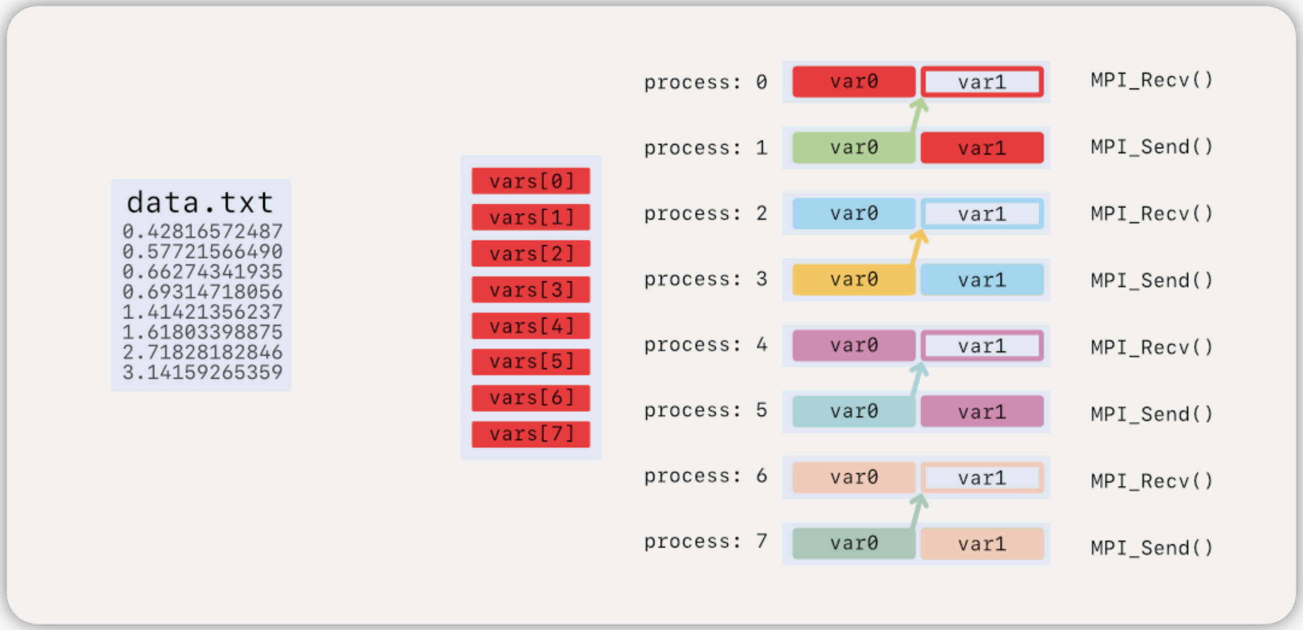
Exercises

Ex07



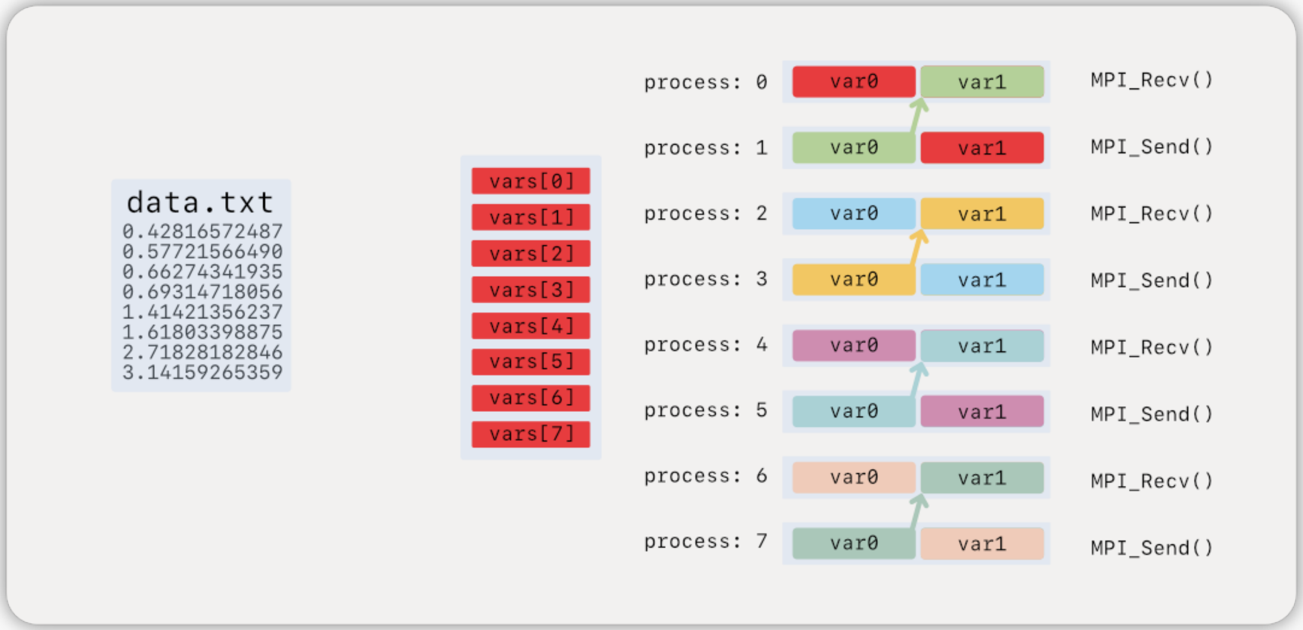
Exercises

Ex07


data.txt
0.42816572487
0.57721566490
0.66274341935
0.69314718056
1.41421356237
1.61803398875
2.71828182846
3.14159265359
vars[0]
vars[1]
vars[2]
vars[3]
vars[4]
vars[5]
vars[6]
vars[7]
process: 0 var0 var1 MPI_Recv()
process: 1 var0 var1 MPI_Send()
process: 2 var0 var1 MPI_Recv()
process: 3 var0 var1 MPI_Send()
process: 4 var0 var1 MPI_Recv()
process: 5 var0 var1 MPI_Send()
process: 6 var0 var1 MPI_Recv()
process: 7 var0 var1 MPI_Send()

Exercises

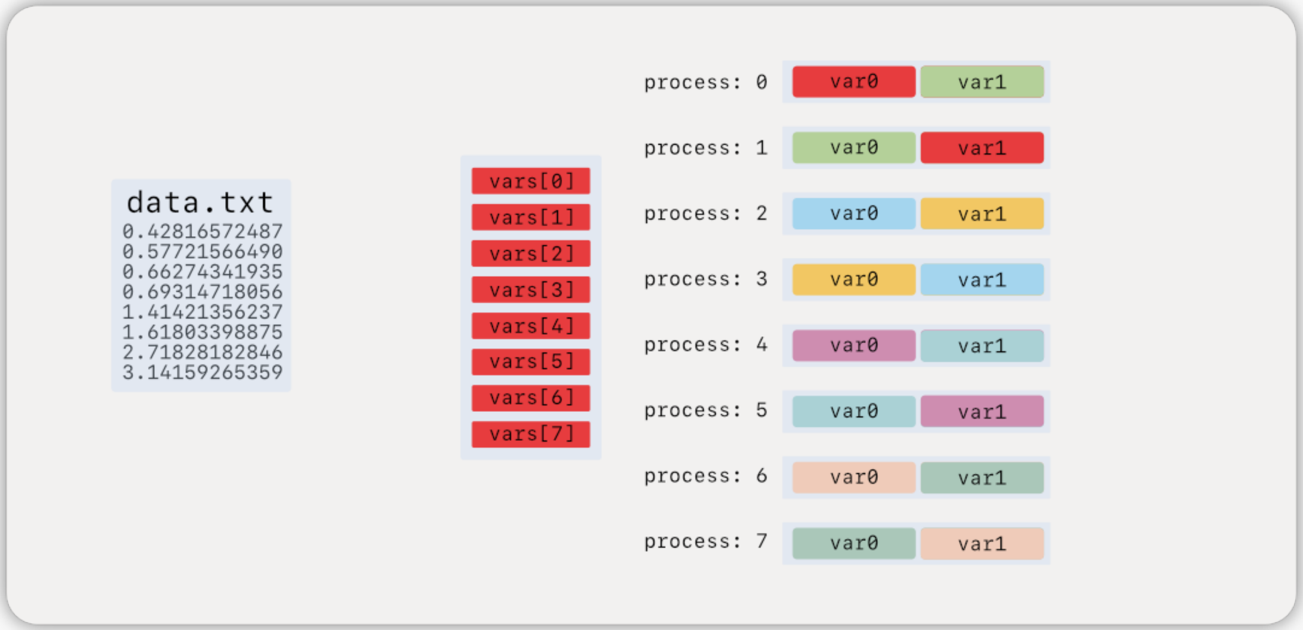
Ex07



data.txt
0.42816572487
0.57721566490
0.66274341935
0.69314718056
1.41421356237
1.61803398875
2.71828182846
3.14159265359
vars[0]
vars[1]
vars[2]
vars[3]
vars[4]
vars[5]
vars[6]
vars[7]
process: 0 var0 var1 MPI_Recv()
process: 1 var0 var1 MPI_Send()
process: 2 var0 var1 MPI_Recv()
process: 3 var0 var1 MPI_Send()
process: 4 var0 var1 MPI_Recv()
process: 5 var0 var1 MPI_Send()
process: 6 var0 var1 MPI_Recv()
process: 7 var0 var1 MPI_Send()

Exercises

Ex07



Exercises

Ex07

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

```c
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

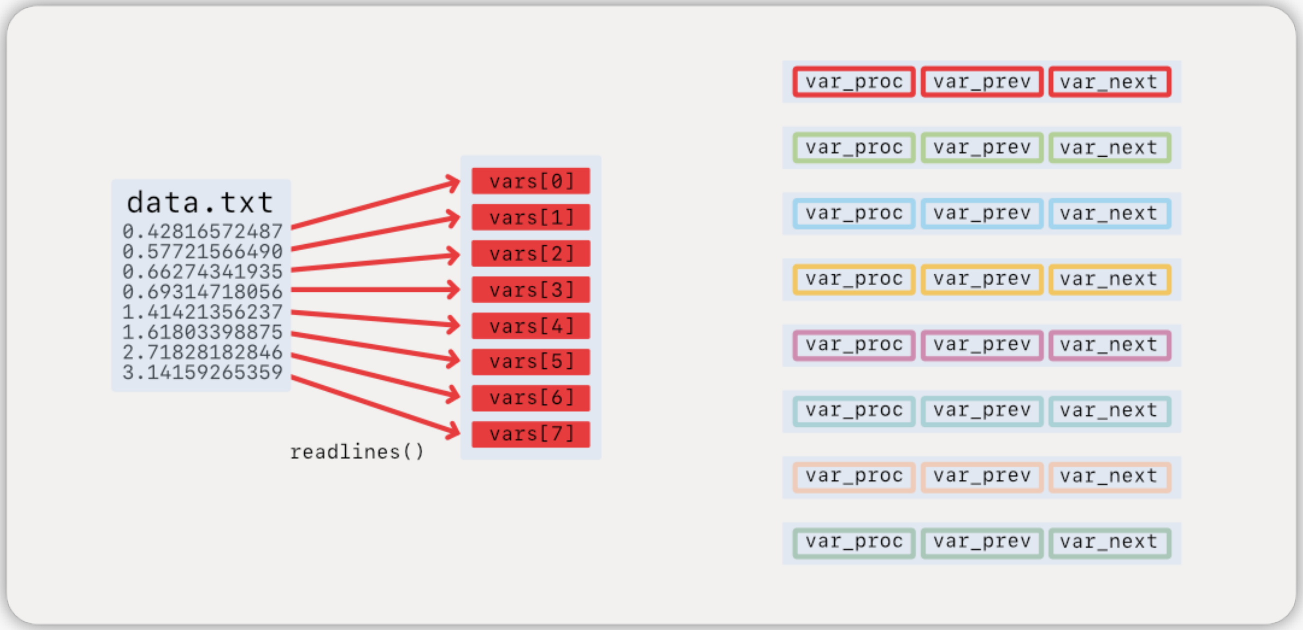
Exercises

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

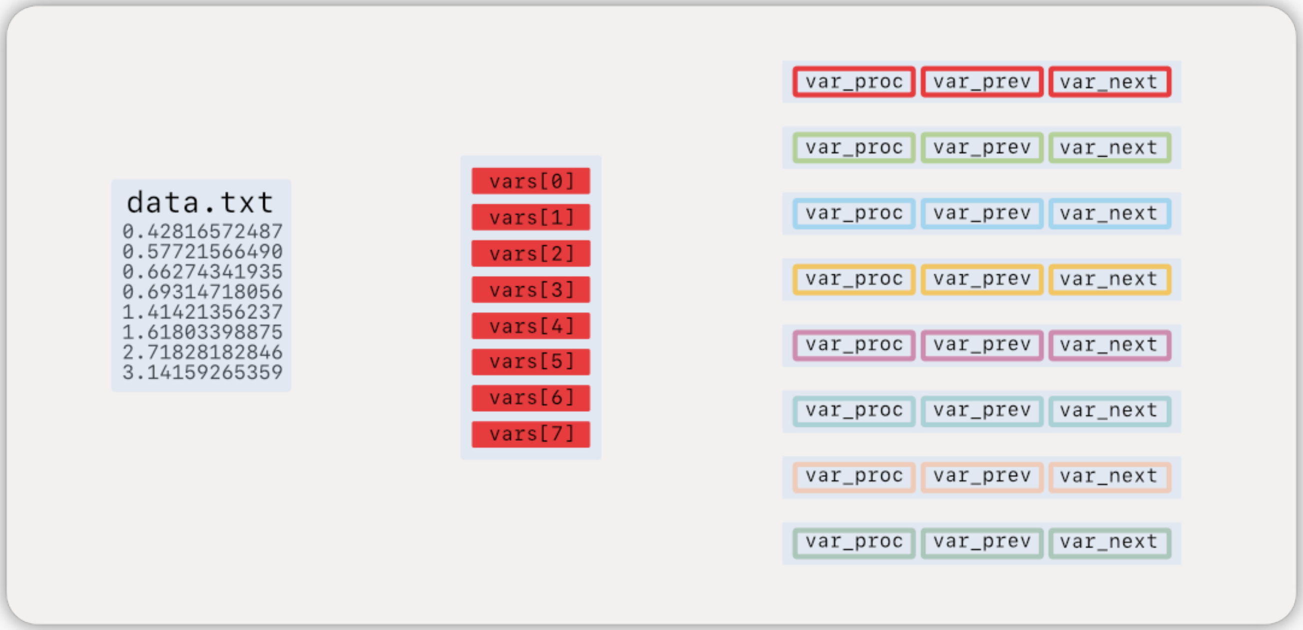
Exercises

Ex08



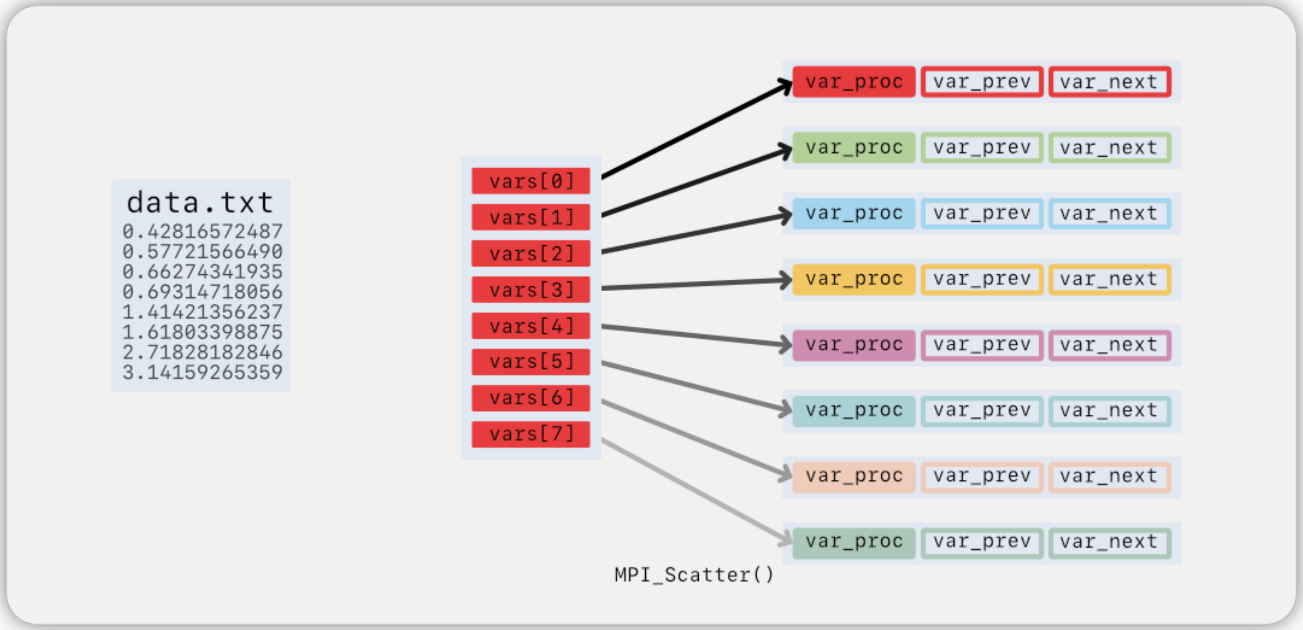
Exercises

Ex08


data.txt
0.42816572487
0.57721566490
0.66274341935
0.69314718056
1.41421356237
1.61803398875
2.71828182846
3.14159265359
vars[0]
vars[1]
vars[2]
vars[3]
vars[4]
vars[5]
vars[6]
vars[7]
var_proc var_prev var_next
var_proc var_prev var_next
var_proc var_prev var_next
var_proc var_prev var_next
var_proc var_prev var_next
var_proc var_prev var_next
var_proc var_prev var_next
var_proc var_prev var_next

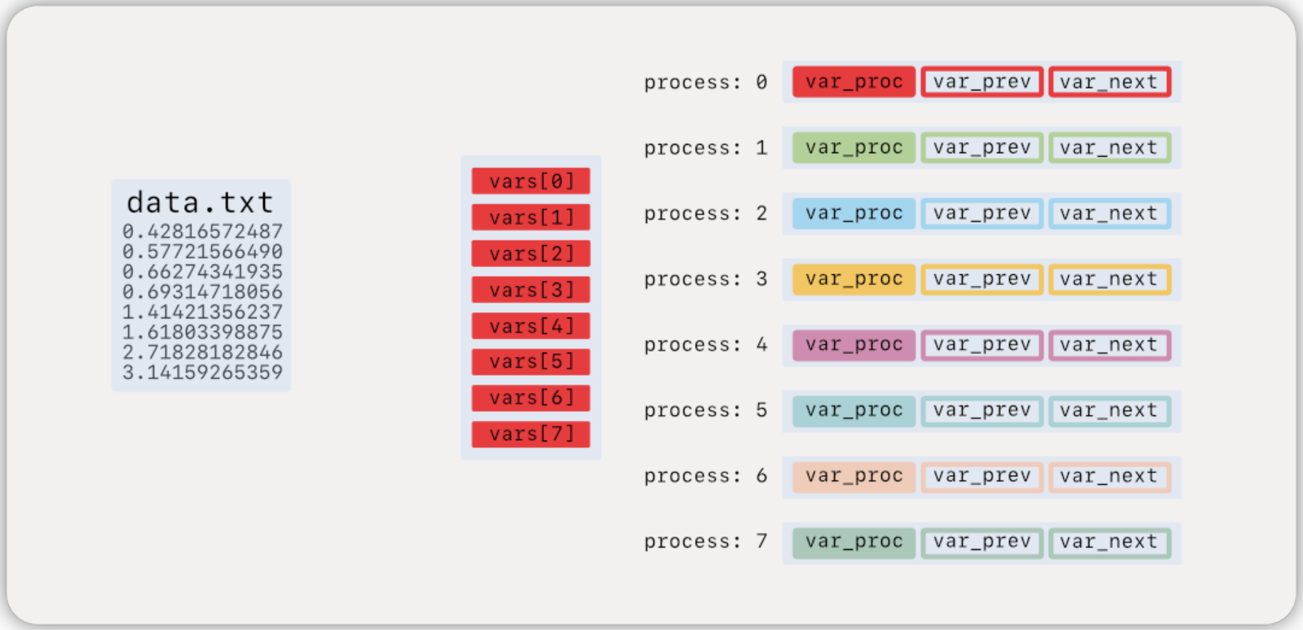
Exercises

Ex08



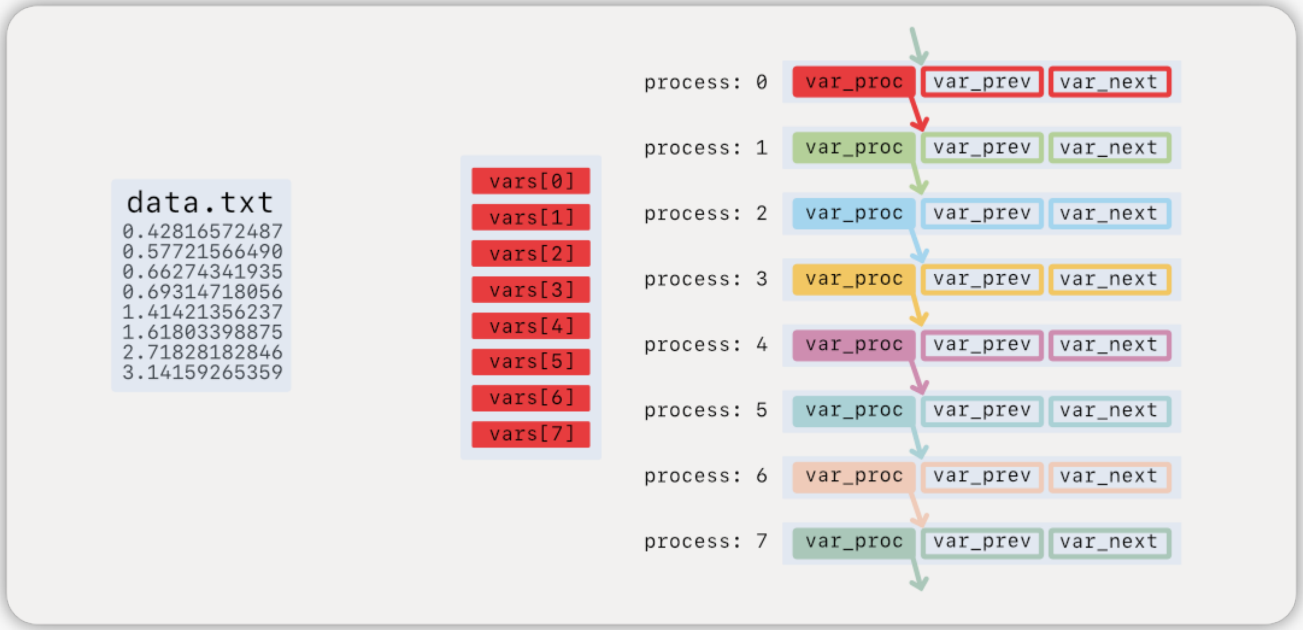
Exercises

Ex08


data.txt
0.42816572487
0.57721566490
0.66274341935
0.69314718056
1.41421356237
1.61803398875
2.71828182846
3.14159265359
vars[0]
vars[1]
vars[2]
vars[3]
vars[4]
vars[5]
vars[6]
vars[7]
process: 0 var_proc var_prev var_next
process: 1 var_proc var_prev var_next
process: 2 var_proc var_prev var_next
process: 3 var_proc var_prev var_next
process: 4 var_proc var_prev var_next
process: 5 var_proc var_prev var_next
process: 6 var_proc var_prev var_next
process: 7 var_proc var_prev var_next

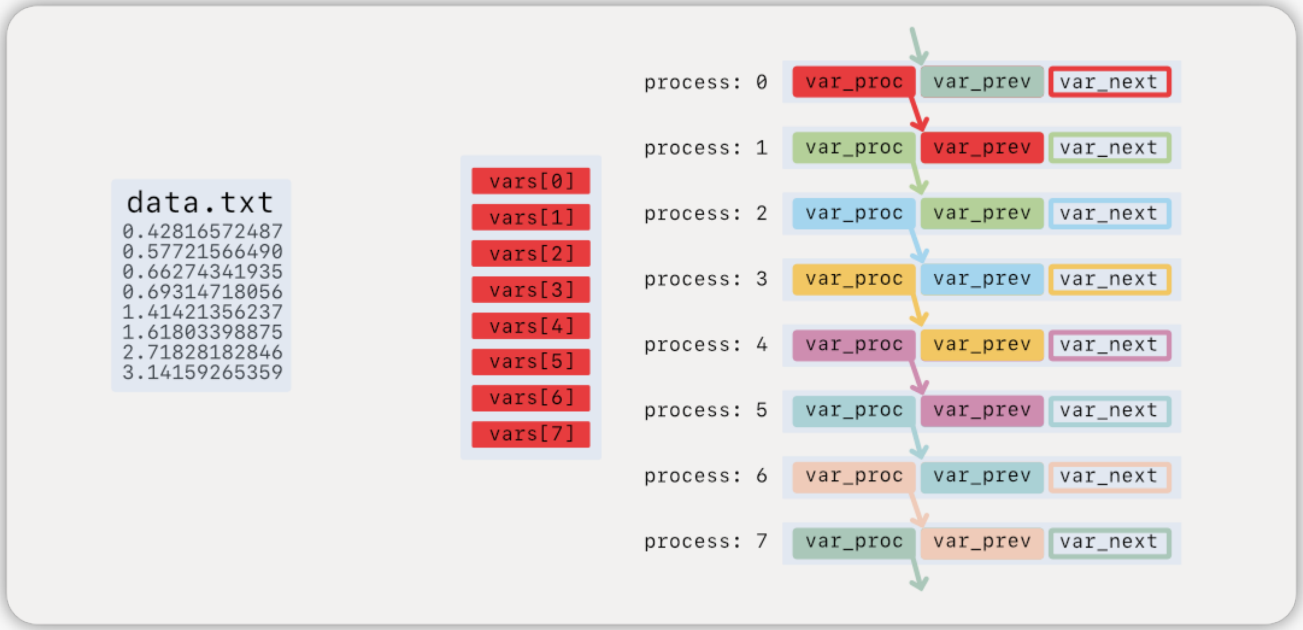
Exercises

Ex08



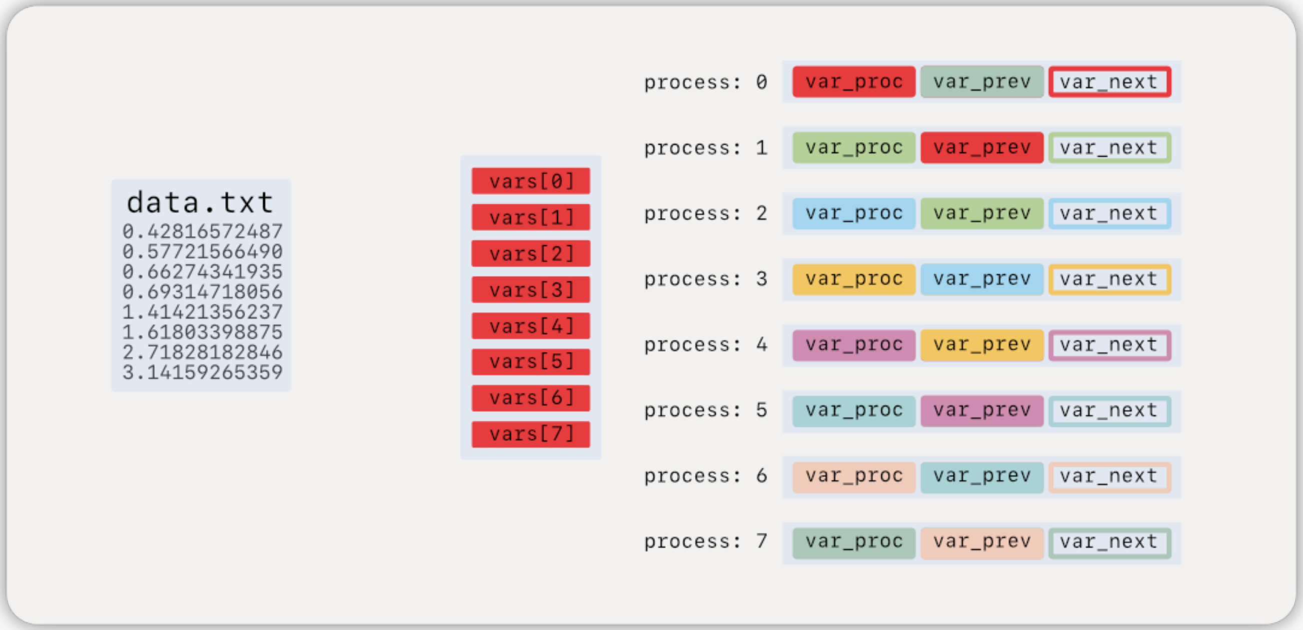
Exercises

Ex08


data.txt
0.42816572487
0.57721566490
0.66274341935
0.69314718056
1.41421356237
1.61803398875
2.71828182846
3.14159265359
vars[0]
vars[1]
vars[2]
vars[3]
vars[4]
vars[5]
vars[6]
vars[7]
process: 0 var_proc var_prev var_next
process: 1 var_proc var_prev var_next
process: 2 var_proc var_prev var_next
process: 3 var_proc var_prev var_next
process: 4 var_proc var_prev var_next
process: 5 var_proc var_prev var_next
process: 6 var_proc var_prev var_next
process: 7 var_proc var_prev var_next

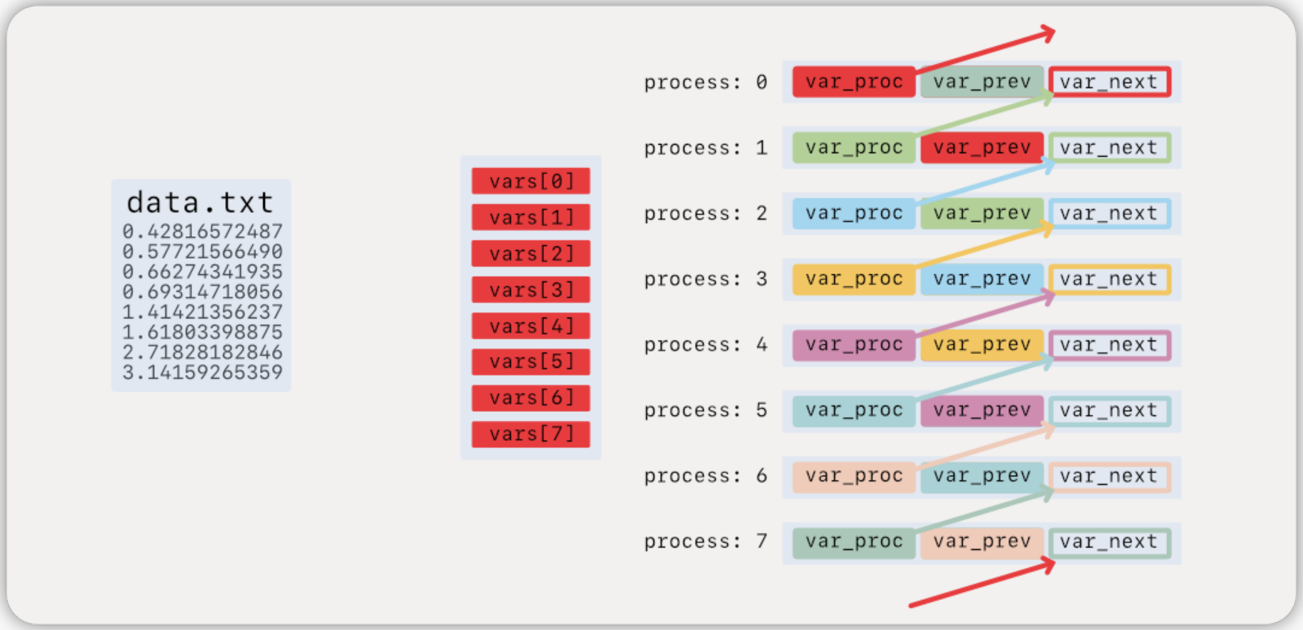
Exercises

Ex08



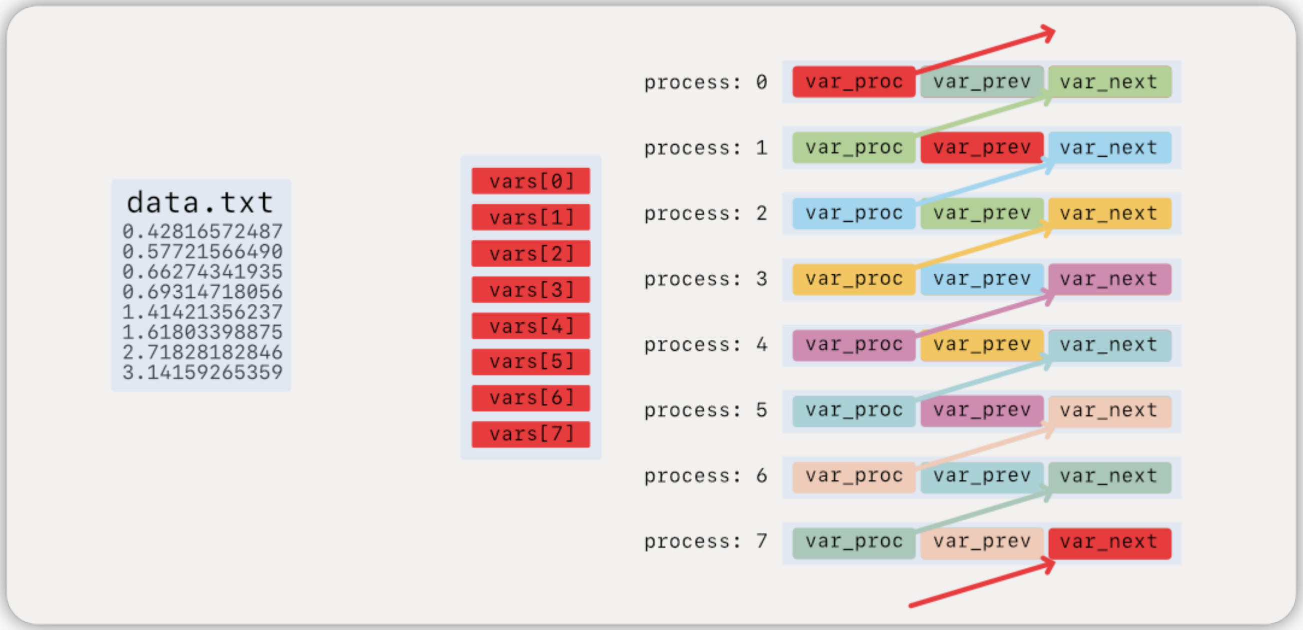
Exercises

Ex08



Exercises

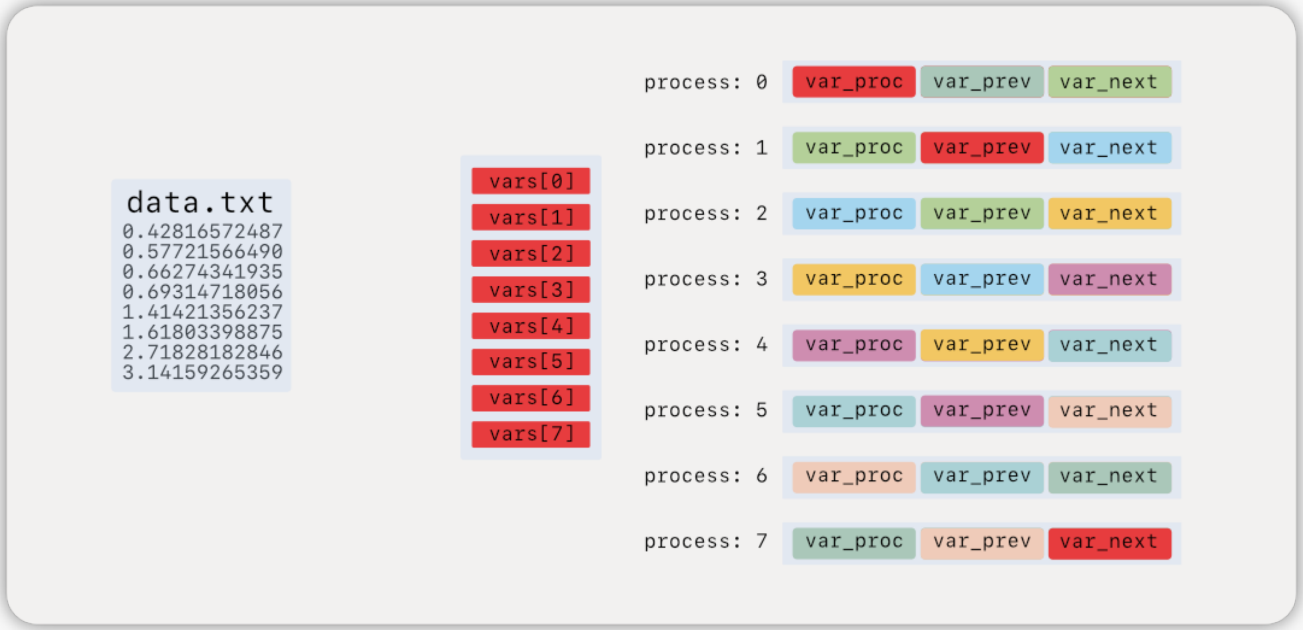
Ex08



data.txt
0.42816572487
0.57721566490
0.66274341935
0.69314718056
1.41421356237
1.61803398875
2.71828182846
3.14159265359
vars[0]
vars[1]
vars[2]
vars[3]
vars[4]
vars[5]
vars[6]
vars[7]
process: 0 var_proc var_prev var_next
process: 1 var_proc var_prev var_next
process: 2 var_proc var_prev var_next
process: 3 var_proc var_prev var_next
process: 4 var_proc var_prev var_next
process: 5 var_proc var_prev var_next
process: 6 var_proc var_prev var_next
process: 7 var_proc var_prev var_next

Exercises

Ex08


data.txt
0.42816572487
0.57721566490
0.66274341935
0.69314718056
1.41421356237
1.61803398875
2.71828182846
3.14159265359
vars[0]
vars[1]
vars[2]
vars[3]
vars[4]
vars[5]
vars[6]
vars[7]
process: 0 var_proc var_prev var_next
process: 1 var_proc var_prev var_next
process: 2 var_proc var_prev var_next
process: 3 var_proc var_prev var_next
process: 4 var_proc var_prev var_next
process: 5 var_proc var_prev var_next
process: 6 var_proc var_prev var_next
process: 7 var_proc var_prev var_next

Exercises

Ex08

- The TODOs 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]$
```

Hybrid MPI/OpenMP

Combining OpenMP and MPI

- OpenMP parallelism within node
- MPI parallelism between nodes

Why?

- Better control of granularity
- Easier to parallelize across domains not divisible by number of processes available
- Allows for controlling parallelism in less parallelizable regions (e.g. I/O)

Hybrid MPI/OpenMP

Types of hybrid parallelism

- Only master thread calls MPI
    - No MPI calls within OpenMP parallel regions
    - May be permitted in an OpenMP Master region
- Any thread calls MPI
    - Serialized
        - If multiple threads call MPI, there are mechanisms to serialize the calls
    - Concurrently
        - MPI thread-safety level permits concurrently calling MPI

Hybrid MPI/OpenMP

Thread awareness in MPI

```
int MPI_Init_thread(int *argc, char ***argv, int required, int *provided)
```

- `MPI_Init_thread()` instead of `MPI_Init()`
- `required` is the level of thread-safety required (input)
- `provided` is the level of thread-safety this implementation of MPI can provide (output)
- One of:
    - `MPI_THREAD_SINGLE`, no thread-safety assumed. Equivalent to `MPI_Init()`
    - `MPI_THREAD_FUNNELED`, it is assumed only one thread will call MPI functions
    - `MPI_THREAD_SERIALIZED`, multiple threads might call MPI, but serialized
    - `MPI_THREAD_MULTIPLE`, any thread can call MPI, even concurrently with other threads

Hybrid MPI/OpenMP

Single mode

- We will cover using *single mode*
    - Parallel regions can exist in a program
    - MPI is called outside of the parallel regions

Hybrid MPI/OpenMP

Single mode

- See `ex09`
- `MPI_Comm_rank()` and `MPI_Comm_size()` are called outside parallel regions
- In the `omp parallel` region we call `omp_get_num_threads()` and `omp_get_thread_num()`

Hybrid MPI/OpenMP

Single mode

- See `ex09`
- `MPI_Comm_rank()` and `MPI_Comm_size()` are called outside parallel regions
- In the `omp parallel` region we call `omp_get_num_threads()` and `omp_get_thread_num()`
- Try changing `OMP_NUM_THREADS` and the `-N` and `-npernode` arguments in `sub-09.sh` and observe the output

Hybrid MPI/OpenMP

Single mode

- See `ex09`
- `MPI_Comm_rank()` and `MPI_Comm_size()` are called outside parallel regions
- In the `omp parallel` region we call `omp_get_num_threads()` and `omp_get_thread_num()`
- Try changing `OMP_NUM_THREADS` and the `-N` and `-npernode` arguments in `sub-09.sh` and observe the output
- Each MPI process spawns its own OpenMP region

Hybrid MPI/OpenMP

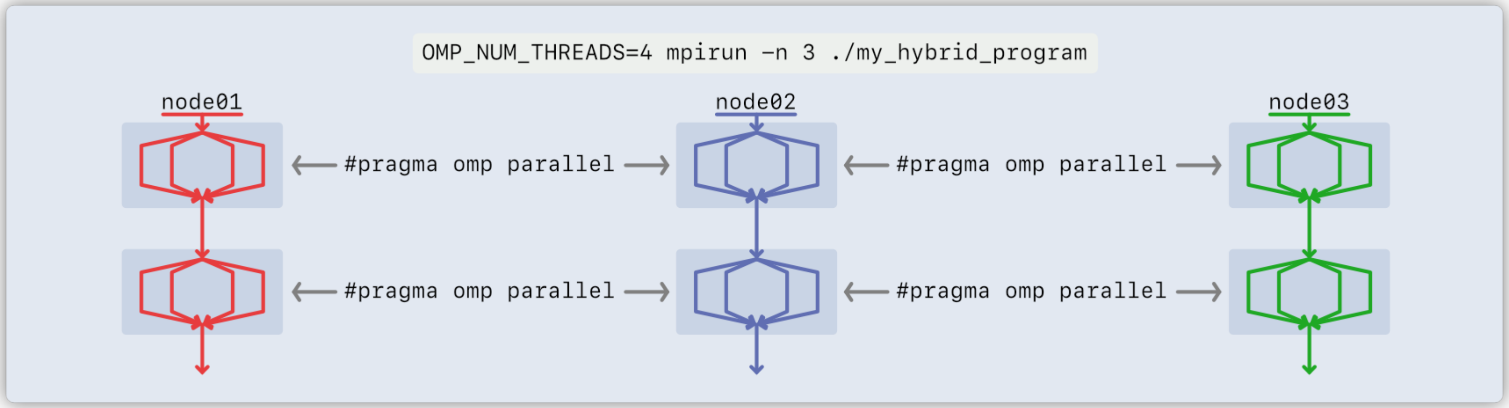
Single mode

- See `ex09`
- `MPI_Comm_rank()` and `MPI_Comm_size()` are called outside parallel regions
- In the `omp parallel` region we call `omp_get_num_threads()` and `omp_get_thread_num()`
- Try changing `OMP_NUM_THREADS` and the `-N` and `-npernode` arguments in `sub-09.sh` and observe the output
- Each MPI process spawns its own OpenMP region
- I.e., there is a hierarchy, in which MPI processes are at the top level and OpenMP threads at the lower level

Hybrid MPI/OpenMP

Single mode

- See `ex09`
- `MPI_Comm_rank()` and `MPI_Comm_size()` are called outside parallel regions
- In the `omp parallel` region we call `omp_get_num_threads()` and `omp_get_thread_num()`
- Try changing `OMP_NUM_THREADS` and the `-N` and `-npernode` arguments in `sub-09.sh` and observe the output
- Each MPI process spawns its own OpenMP region
- I.e., there is a hierarchy, in which MPI processes are at the top level and OpenMP threads at the lower level


```
OMP_NUM_THREADS=4 mpirun -n 3 ./my_hybrid_program
```

Hybrid MPI/OpenMP

Dot product using hybrid MPI/OpenMP

- `ex10` implements a vector dot-product using MPI and OpenMP
- Investigate the timing as you vary the number of threads per process and total number of processes