Introduction to High Performance Computing

\\

SDS406 – Fall semester, 2024 - 2025

\\

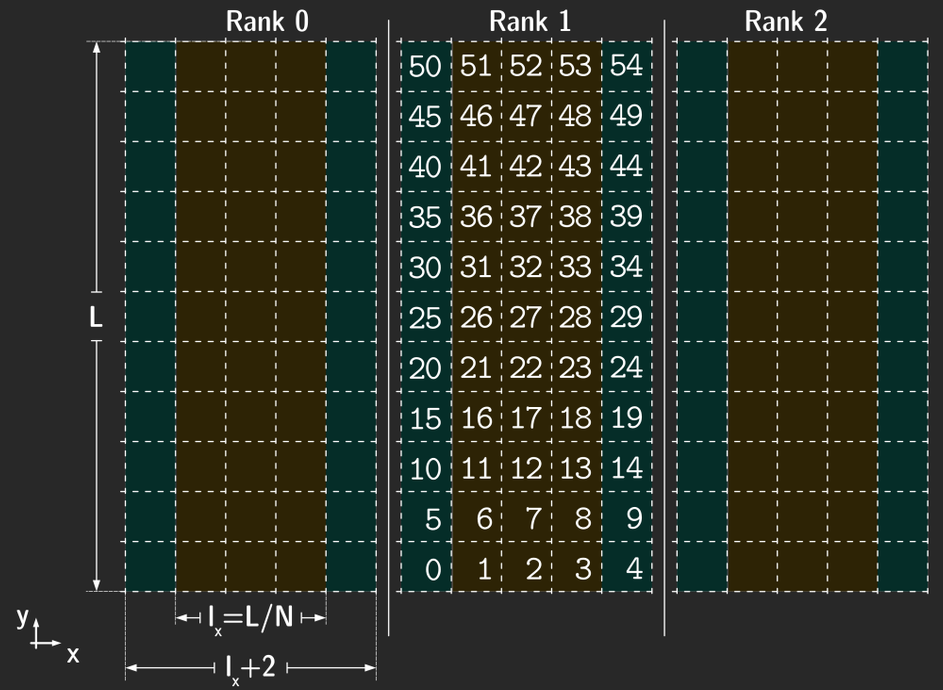
L10: MPI custom types and MPI-I/O, 2nd December 2024

Parallelization of PDEs

- So far, we parallelized in the slowest running index (y) which meant that we needed to communicate continuous elements of the arrays between neighbors.
- Now we will consider parallelizing of the heat equation in the x-dimension
  - We need to communicate the boundaries of y
  - This means we need to communicate *non-contiguous* elements

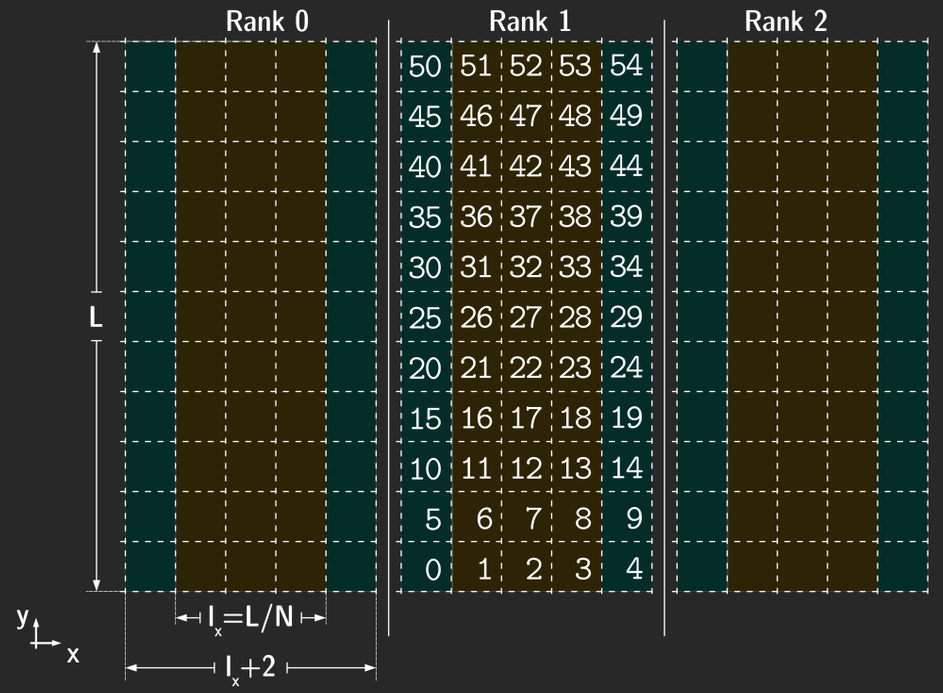
Parallelization of PDEs

- So far, we parallelized in the slowest running index (y) which meant that we needed to communicate continuous elements of the arrays between neighbors.
- Now we will consider parallelizing of the heat equation in the x-dimension
  - We need to communicate the boundaries of y
  - This means we need to communicate *non-contiguous* elements



Parallelization of PDEs

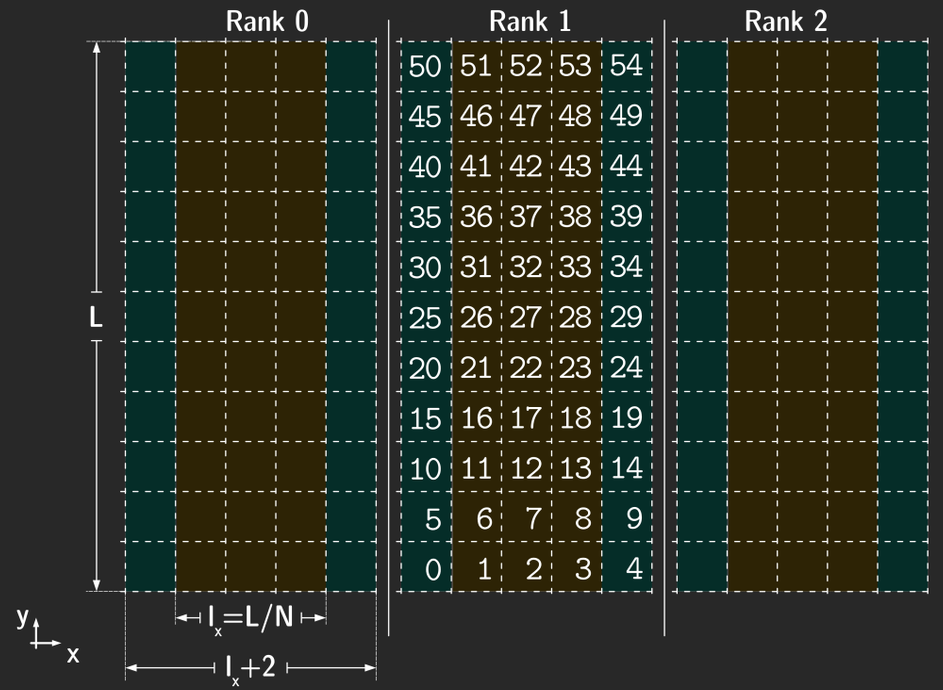
- So far, we parallelized in the slowest running index (y) which meant that we needed to communicate continuous elements of the arrays between neighbors.
- Now we will consider parallelizing of the heat equation in the x-dimension
  - We need to communicate the boundaries of y
  - This means we need to communicate *non-contiguous* elements



- Could use temporary arrays to copy boundaries before communication

Parallelization of PDEs

- So far, we parallelized in the slowest running index (y) which meant that we needed to communicate continuous elements of the arrays between neighbors.
- Now we will consider parallelizing of the heat equation in the x-dimension
  - We need to communicate the boundaries of y
  - This means we need to communicate *non-contiguous* elements



- Could use temporary arrays to copy boundaries before communication
- *Alternatively*, so-called *custom MPI types*

Custom types

- `MPI_DOUBLE`, `MPI_FLOAT`, `MPI_INT`, etc. are pre-defined MPI types
- User-defined types allow for custom data types
- Using types with *strides* provides for a convenient way to arrange sending and receiving non-continuous data

Custom types

- `MPI_DOUBLE`, `MPI_FLOAT`, `MPI_INT`, etc. are pre-defined MPI types
- User-defined types allow for custom data types
- Using types with *strides* provides for a convenient way to arrange sending and receiving non-continuous data

MPI Vector type

```
int MPI_Type_vector(int count, int blocklength, int stride,
                    MPI_Datatype oldtype, MPI_Datatype *newtype)
```

Custom types

- `MPI_DOUBLE`, `MPI_FLOAT`, `MPI_INT`, etc. are pre-defined MPI types
- User-defined types allow for custom data types
- Using types with *strides* provides for a convenient way to arrange sending and receiving non-continuous data

MPI Vector type

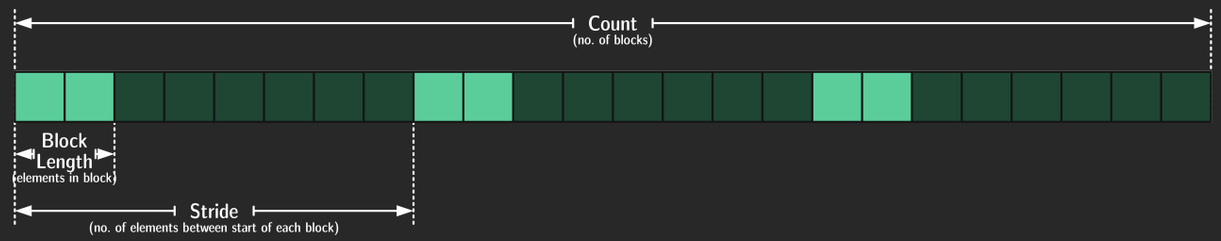
```
int MPI_Type_vector(int count, int blocklength, int stride,
                    MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- One of the most basic ways to create a custom MPI type
- The following corresponds to the figure
  ⟶ pick the 6 highlighted elements from the 24-element array

```
MPI_Datatype dtype;
MPI_Type_vector(3, 2, 8, MPI_DOUBLE, &dtype);
MPI_Type_commit(&dtype);
```

- Can then specify this new type in MPI functions
  ⟶ e.g. in an `MPI_Send()`

```
MPI_Send(&a[0], 1, dtype, dest, tag, MPI_COMM_WORLD);
```



Custom types

MPI Vector type

- See `ex01`:
  - The root process initializes an array of `L` elements, `r[L]`
  - All processes post a non-blocking receive (`MPI_Irecv()`) for `L/nproc` elements
  - The root process will send `L/nproc` elements to each process, one-by-one, in a loop
  - We would like:
    - `rank 0` to receive the elements: `r[0]`, `r[nproc-1]`, `r[2*nproc-1]`, etc.
    - `rank 1` to receive the elements: `r[1]`, `r[ nproc]`, `r[ 2*nproc]`, etc.
    - `rank 2` to receive the elements: `r[2]`, `r[nproc+1]`, `r[2*nproc+1]`, etc.
    - and so on

Custom types

MPI Vector type

- See `ex01`:
  - The root process initializes an array of `L` elements, `r[L]`
  - All processes post a non-blocking receive (`MPI_Irecv()`) for `L/nproc` elements
  - The root process will send `L/nproc` elements to each process, one-by-one, in a loop
  - We would like:
    - `rank 0` to receive the elements: `r[0]`, `r[nproc-1]`, `r[2*nproc-1]`, etc.
    - `rank 1` to receive the elements: `r[1]`, `r[ nproc]`, `r[ 2*nproc]`, etc.
    - `rank 2` to receive the elements: `r[2]`, `r[nproc+1]`, `r[2*nproc+1]`, etc.
    - and so on

```
/***
 * TODO: Define the new datatype that strides by `nproc' elements
 ***/
MPI_Datatype dtype;
MPI_Type_vector(/* TODO */);
MPI_Type_commit(&dtype);
```

Parallel I/O

Binary vs text // Binary I/O // Parallel I/O with MPI

Binary file I/O

- So far we have been writing files as *formatted text*
- This requires *converting* the internal, binary representation of floating point numbers into *human readable* representations, e.g. as characters, using a specific encoding, usually *ASCII*
- However, the more general and flexible way to store data is in *binary*. This allows storing data of multiple types, multidimensional data, etc.
- Think of it as storing the same binary data as in a memory buffer to a file

Binary file I/O

- So far we have been writing files as *formatted text*
- This requires *converting* the internal, binary representation of floating point numbers into *human readable* representations, e.g. as characters, using a specific encoding, usually *ASCII*
- However, the more general and flexible way to store data is in *binary*. This allows storing data of multiple types, multidimensional data, etc.
- Think of it as storing the same binary data as in a memory buffer to a file

```
double x[L];
/* Fill x[] */
FILE *fp = fopen("filename", "w");
fwrite(&x[0], sizeof(double), L, fp);
fclose(fp);
```

```
double y[L];
FILE *fp = fopen("filename", "r");
fread(&y[0], sizeof(double), L, fp);
fclose(fp);
/* Do something with y[] */
```

Binary file I/O

- In Python, the `struct` module can be used to convert to/from a byte string
- Alternatively you can use `numpy` and the `numpy.fromfile()` function

```
import struct

fp = open("out", "br")
r = fp.read()
fp.close()
n = len(r)//8
r = struct.unpack(n*"d", r)
```

```
import numpy as np

r = np.fromfile("out", dtype=np.float64)
```

Binary file I/O

- In Python, the `struct` module can be used to convert to/from a byte string
- Alternatively you can use `numpy` and the `numpy.fromfile()` function

```
import struct

fp = open("out", "br")
r = fp.read()
fp.close()
n = len(r)//8
r = struct.unpack(n*"d", r)
```

```
import numpy as np

r = np.fromfile("out", dtype=np.float64)
```

- Inspecting files on the command line:
  - The Linux commands `more` and `less` interpret files as *text*. A text file is still a binary file, but its contents are intended to be interpreted as characters
  - The Linux command `od` can be used to interpret files in other formats, e.g. octal, hexadecimal, floating point, etc. in addition to text. `od` stands for "Octal dump".

Binary file I/O

- In Python, the `struct` module can be used to convert to/from a byte string
- Alternatively you can use `numpy` and the `numpy.fromfile()` function

```
import struct

fp = open("out", "br")
r = fp.read()
fp.close()
n = len(r)//8
r = struct.unpack(n*"d", r)
```

```
import numpy as np

r = np.fromfile("out", dtype=np.float64)
```

- Inspecting files on the command line:
  - The Linux commands `more` and `less` interpret files as *text*. A text file is still a binary file, but its contents are intended to be interpreted as characters
  - The Linux command `od` can be used to interpret files in other formats, e.g. octal, hexadecimal, floating point, etc. in addition to text. `od` stands for "Octal dump".
- See `ex02` which contains some examples demonstrating binary read/write

Parallel I/O

Variants:

- "Non-parallel" I/O: all send to one rank, which carries out I/O ⟵ our approach so far, with the exception of the heat equation example
- Independent parallel I/O: each rank writes/reads to/from its own file
- "Cooperative" parallel I/O: each rank writes/reads to/from different parts of the same file

Parallel I/O

Variants:

- "Non-parallel" I/O: all send to one rank, which carries out I/O ⟵ our approach so far, with the exception of the heat equation example
- Independent parallel I/O: each rank writes/reads to/from its own file
- "Cooperative" parallel I/O: each rank writes/reads to/from different parts of the same file

Cooperative parallel I/O

- Requires parallelism aware software layer (MPI-I/O)
- Performance depends on underlying hardware and filesystem
- In MPI context
  - Writing to a file ⟺ sending a buffer
  - Reading from a file ⟺ receiving a buffer

Parallel I/O

Variants:

- "Non-parallel" I/O: all send to one rank, which carries out I/O ⟵ our approach so far, with the exception of the heat equation example
- Independent parallel I/O: each rank writes/reads to/from its own file
- "Cooperative" parallel I/O: each rank writes/reads to/from different parts of the same file

Cooperative parallel I/O

- Requires parallelism aware software layer (MPI-I/O)
- Performance depends on underlying hardware and filesystem
- In MPI context
  - Writing to a file ⇔ sending a buffer
  - Reading from a file ⇔ receiving a buffer
- This means custom types can be used in the same way as in `MPI_Send()` and `MPI_Recv`
  - When writing a custom type can be used to pick out specific elements of the array to be written
  - When reading it can be used to pick the specific elements of the array to be filled
- Extensive use of custom MPI types is also typically used to set "file views". These determine which parts of the file will be picked to be read or written.

MPI-I/O

- `MPI_File_open`, `MPI_File_write_at`, `MPI_File_write_all`, etc.

```
int MPI_File_open(MPI_Comm comm, const char *filename,
                  int amode, MPI_Info info,
                  MPI_File *fh)
```

Example:

```
MPI_File fh;
MPI_File_open(MPI_COMM_WORLD, "out", MPI_WRONLY | MPI_CREATE, MPI_INFO_NULL, &fh)
```

- Open file with filename `out` in write-only mode; create if it does not exist
- Options for `amode`: `MPI_MODE_APPEND`, `MPI_MODE_CREATE`, `MPI_MODE_DELETE_ON_CLOSE`, `MPI_MODE_EXCL`, `MPI_MODE_RDONLY`, `MPI_MODE_RDWR`, `MPI_MODE_SEQUENTIAL`, `MPI_MODE_WRONLY`, `MPI_MODE_UNIQUE_OPEN`

MPI-I/O

Use `MPI_File_write_at()` and `MPI_File_read_at()` write or read different parts of a file according to an offset.

```
int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf,
                      int count, MPI_Datatype datatype, MPI_Status *status)
```

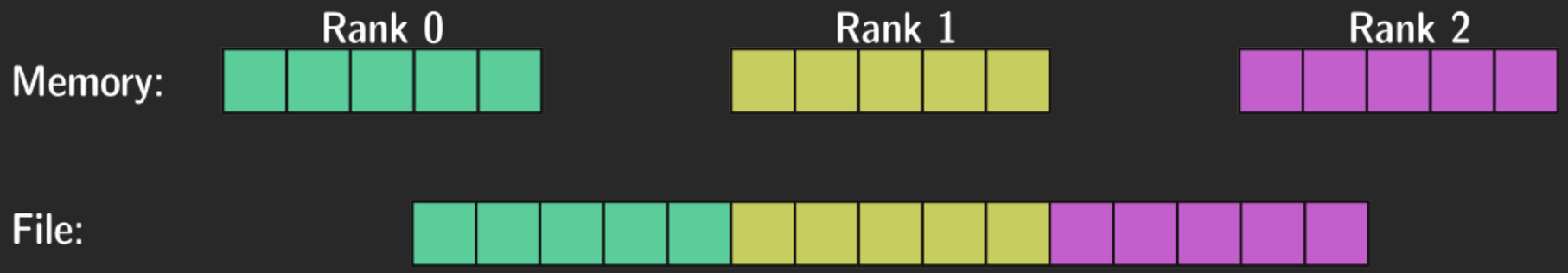
- E.g. see `ex03`
  - Each rank allocates and fills an array of `N` random numbers
  - `MPI_File_write_at()` is used to write the data into a file, in order of the ranks

MPI-I/O

Use `MPI_File_write_at()` and `MPI_File_read_at()` write or read different parts of a file according to an offset.

```
int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf,
                      int count, MPI_Datatype datatype, MPI_Status *status)
```

- E.g. see `ex03`
  - Each rank allocates and fills an array of `N` random numbers
  - `MPI_File_write_at()` is used to write the data into a file, in order of the ranks



MPI-I/O

File views

- File views determine which part of the file each process can see
- Same machinery as in sending/receiving custom types, e.g. using vector types with strides, etc.

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
                      MPI_Datatype filetype, const char *datarep, MPI_Info info)
```

  - `etype`: the element type (e.g. `MPI_DOUBLE`)
  - `filetype`: the type used for the file view. Determines which part of the file the rank can "view"
  - `datarep`: use `"native"` unless there is a need to explicitly set a different data representation

- In case we are writing to a file:
  - Think of the `filetype` as the custom type on the buffer of the receiving rank (which is now the file we are writing to)
  - Think of the `MPI_File_write()` call as the same as an `MPI_Send()`

MPI-I/O

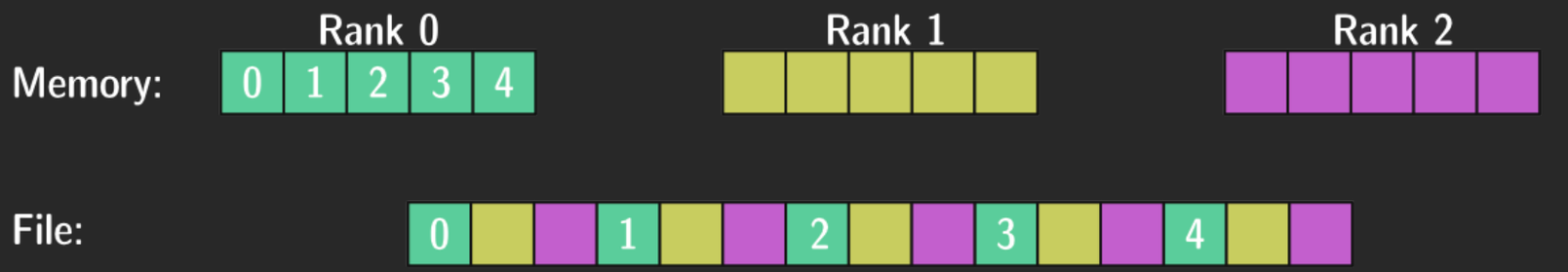
File views

- `ex04` demonstrate the basic use of a "file view"
- Start `nproc` processes
- Each rank allocates and fills `N` double precision numbers
  - Rank 0 fills it with `1.0`, `2.0`, ..., `N-1`,
  - Rank 1 fills it with `N`, `N+1`, ..., `2*N-1`
  - etc.
- We want the file to be written such that:
  - The first `nproc` elements in the file are the first elements of all processes
  - The next `nproc` elements in the file are the second elements of all processes
  - etc.

MPI-I/O

File views

- `ex04` demonstrate the basic use of a "file view"
- Start `nproc` processes
- Each rank allocates and fills `N` double precision numbers
  - Rank 0 fills it with `1.0`, `2.0`, ..., `N-1`,
  - Rank 1 fills it with `N`, `N+1`, ..., `2*N-1`
  - etc.
- We want the file to be written such that:
  - The first `nproc` elements in the file are the first elements of all processes
  - The next `nproc` elements in the file are the second elements of all processes
  - etc.

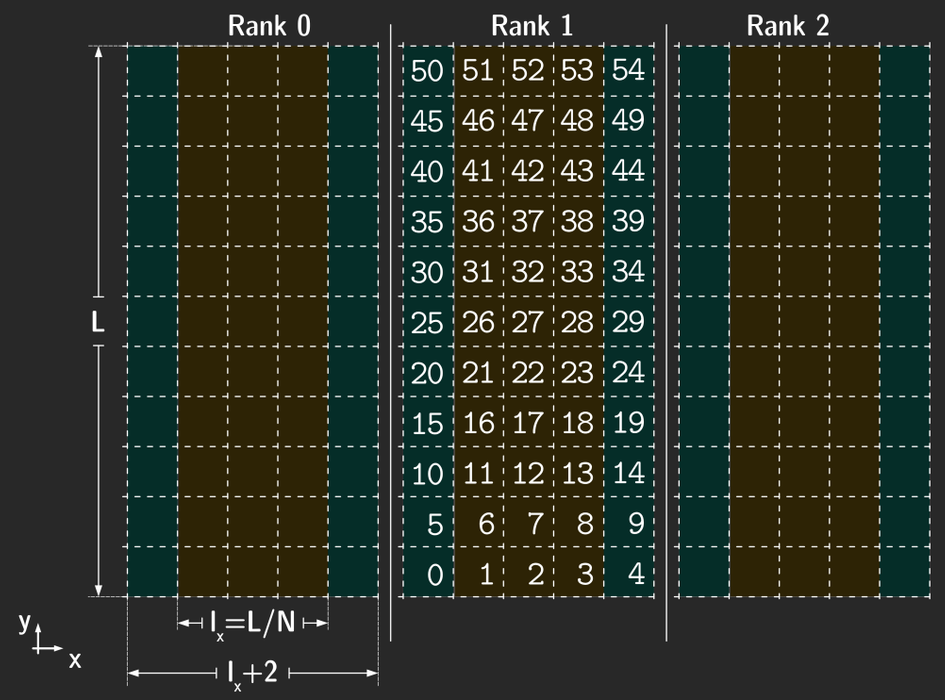


Parallelization of PDEs

The heat equation program with parallelization over the χ-coordinate and MPI-I/O

Exercise `ex05`, with missing parts marked as `TODO`s

- Communication of the non-contiguous elements can be achieved using a vector custom data type
- Writing the non-contiguous elements to a binary file in parallel using MPI-I/O and file views
- The included `plot.py` takes care of reading from a binary file but is otherwise the same as before



Parallelization of PDEs

```
/*
 * TODO_1
 */
#define IDX(y, x)
```

Parallelization of PDEs

```
/*
 * TODO_1
 */
#define IDX(y, x)
```

```
/***
 * Set the boundary condition for v[L*(lx+2)]
 ***/
void
boundary_condition(double *v)
{
  int size, rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  /*
   * TODO_2
   */

  /*
   * Which rank has x == 0?
   */
  if(rank == /* */) {
    /* ... */
  }

  /*
   * Which rank has x == L-1?
   */
  if(rank == /* */) {
    /* ... */
  }

  /*
   * Set y = L/2 to 1
   */
...
```

Parallelization of PDEs

```c
/***
 * Update the boundary of v[L*(lx+2)], by exchanging "halos"
 ***/
void
update_boundary(double *v)
{
  int size, rank;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  /*
   * TODO_3
   */

  MPI_Datatype dtype;
  MPI_Type_vector(/* ... */);
  MPI_Type_commit(&dtype);

  /* Send x = 0 boundary to lx+1 of backward neighbor */
  MPI_Sendrecv(/* ... */);

  /* Send x = lx-1 boundary to -1 of forward neighbor */
  MPI_Sendrecv(/* ... */);
  ...
```

Parallelization of PDEs

MPI-I/O and combining two custom types

- Use MPI-I/O to write the resulting array in parallel
- Need two custom types: one for the "file view" and one for the "data view"

Parallelization of PDEs

MPI-I/O and combining two custom types

- Use MPI-I/O to write the resulting array in parallel
- Need two custom types: one for the "file view" and one for the "data view"