Introduction to High Performance Computing

SDS406 – Fall semester, 2024 - 2025

L11: MPI custom types and MPI-I/O (contd. from L10), 9th December 2024

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

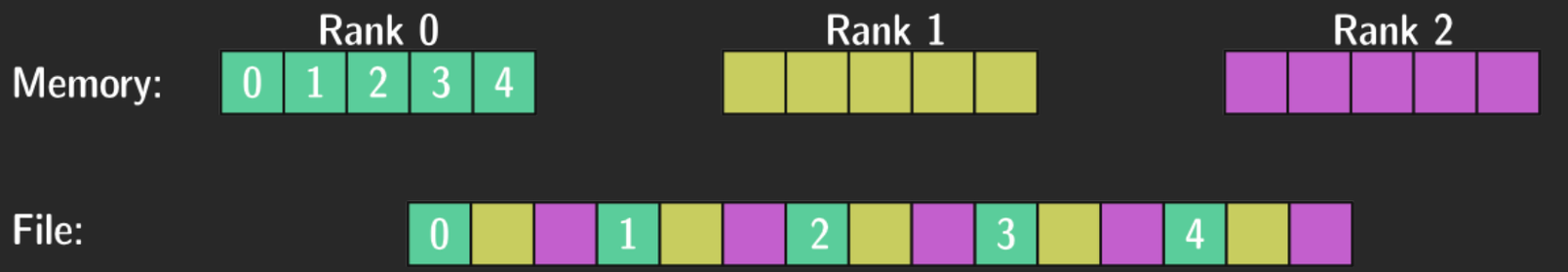
File views

- `l10/ex04` (or `l11/ex01`) 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

- `l10/ex04` (or `l11/ex01`) 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.

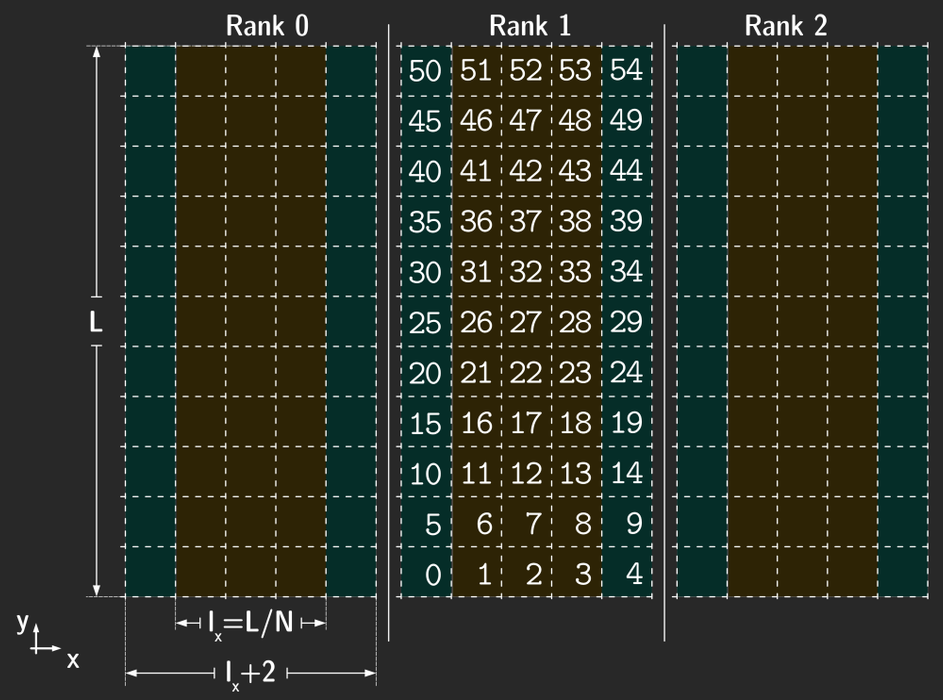


Parallelization of PDEs

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

Exercise `l10/ex05` (or `l11/ex02`), 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

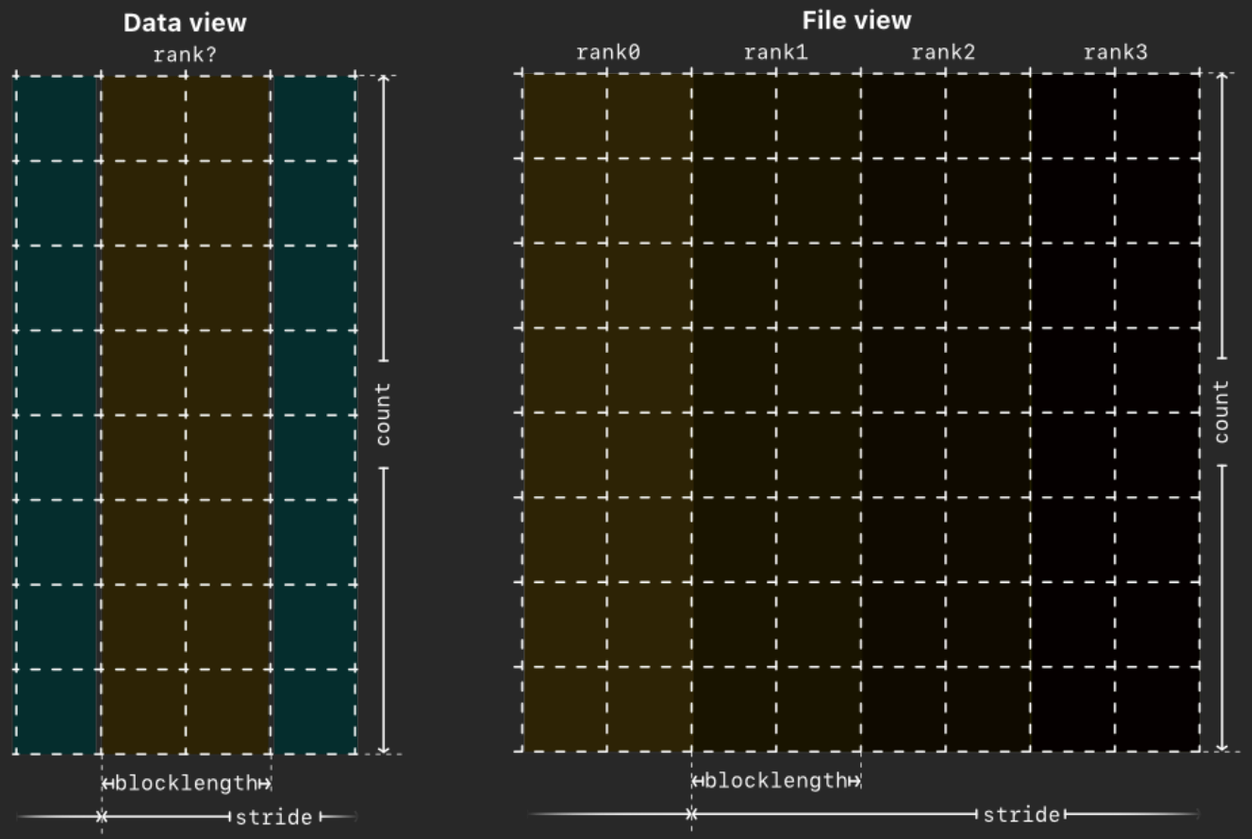
MPI-I/O and combining two custom types

- 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"



Strong scaling of the MPI-parallelized heat equation solution

$\Rightarrow$ Use at most two nodes, 32 processes per node

$\Rightarrow$ Plot against inverse time-to-solution rather than speed-up

$\Rightarrow$ Scale both parallelizations over x and y separately

$\Rightarrow$ Include in the plot the scalar version of at $n_{\text{proc}} = 1$