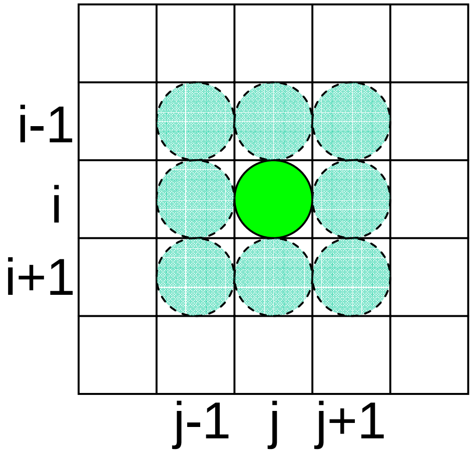
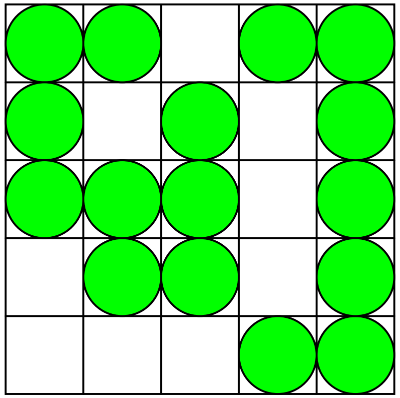
Conway's Game of Life

- A cellular automata
  - Described in 1970 Scientific American
  - Many interesting behaviors; see:
    - *http://www.ibiblio.org/lifepatterns/october1970.html*
- Program issues are very similar to those for codes that use regular meshes, such as PDE solvers
  - Allows us to concentrate on the MPI issues

Rules for Life

- Matrix values A(i,j) initialized to 1 (live) or 0 (dead)
- In each iteration, A(i,j) is set to
  - 1(live) if either
    - *the sum of the values of its 8 neighbors is 3, or*
    - *the value was already 1 and the sum of its 8 neighbors is 2 or 3*
  - 0 (dead) otherwise





Implementing Life

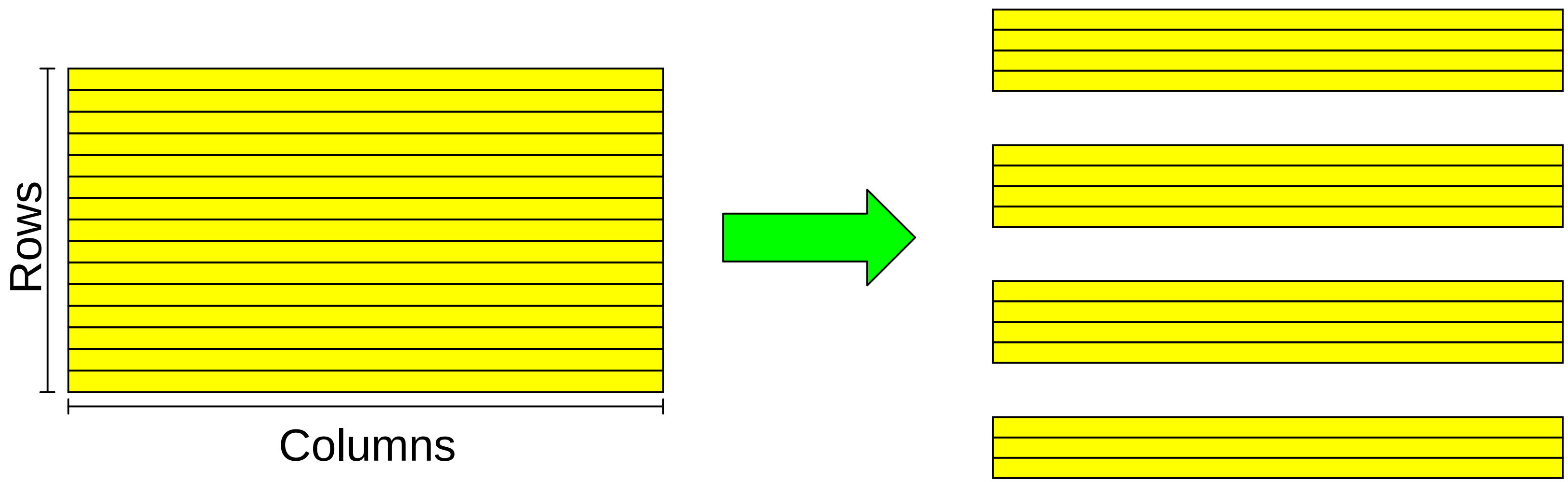
- For the non-parallel version, we:
  - Allocate a 2D matrix to hold state
    - *Actually two matrices, and we will swap them between steps*
  - Initialize the matrix
    - *Force boundaries to be "dead"*
    - *Randomly generate states inside*
  - At each time step:
    - *Calculate each new cell state based on previous cell states (including neighbors)*
    - *Store new states in second matrix*
    - *Swap new and old matrices*

Steps in Designing the Parallel Version

- Start with the “global” array as the main object
  - Natural for output – result we’re computing
- Describe decomposition in terms of global array
- Describe communication of data, still in terms of the global array
- Define the “local” arrays and the communication between them by referring to the global array

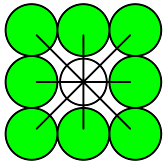
Step 1: Description of Decomposition

- By rows (1D or row-block)
  - Each process gets a group of adjacent rows
- Later we'll show a 2D decomposition

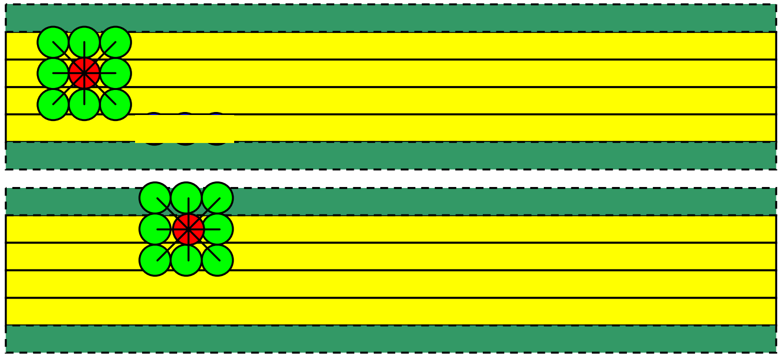


Step 2: Communication

- “Stencil” requires read access to data from neighbor cells



- We allocate extra space on each process to store neighbor cells
- Use send/recv or RMA to update prior to computation

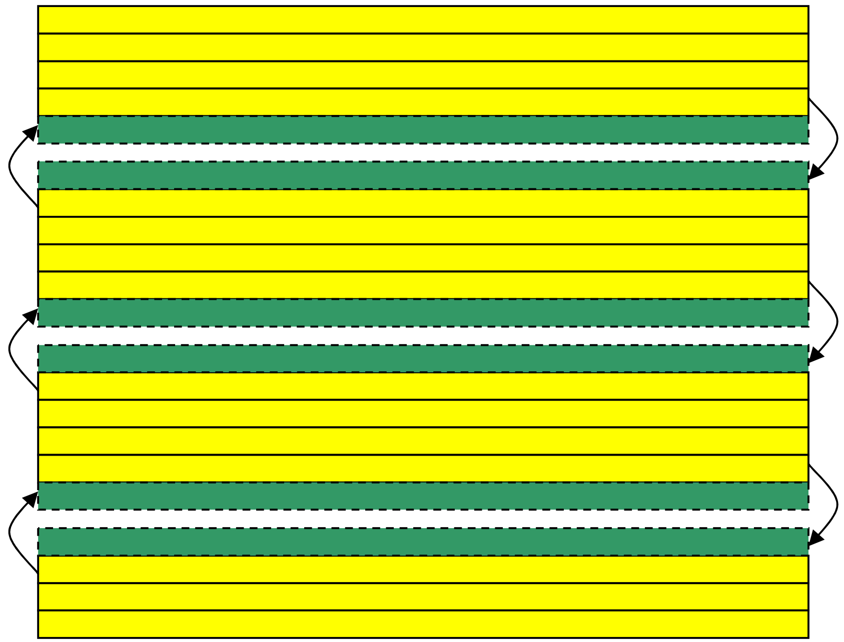


Step 3: Define the Local Arrays

- Correspondence between the local and global array
- “Global” array is an abstraction; there is no one global array allocated anywhere
- Instead, we compute parts of it (the local arrays) on each process
- Provide ways to output the global array by combining the values on each process (parallel I/O!)

Boundary Regions

- In order to calculate next state of cells in edge rows, need data from adjacent rows
- Need to communicate these regions at each step
  - First cut: use isend and irecv
  - Revisit with RMA later



Life Point-to-Point Code Walkthrough

- Points to observe in the code:
  - Handling of command-line arguments
  - Allocation of local arrays
  - Use of a routine to implement halo exchange
    - *Hides details of exchange*

Allows us to use matrix[row][col] to address elements

See mlife.c pp. 1-8 for code example.

Note: Parsing Arguments

- MPI standard does <u>not</u> guarantee that command line arguments will be passed to all processes.
  - Process arguments on rank 0
  - Broadcast options to others
    - *Derived types allow one bcast to handle most args*
  - Two ways to deal with strings
    - *Big, fixed-size buffers*
    - *Two-step approach: size first, data second (what we do in the code)*

See mlife.c pp. 9-10 for code example.

Point-to-Point Exchange

- Duplicate communicator to ensure communications do not conflict
- Non-blocking sends and receives allow implementation greater flexibility in passing messages

See mlife-pt2pt.c pp. 1-3 for code example.

Parallel I/O and Life

Supporting Checkpoint/Restart

- For long-running applications, the cautious user checkpoints
- Application-level checkpoint involves the application saving its own state
  - Portable!
- A canonical representation is preferred
  - Independent of number of processes
- Restarting is then possible
  - Canonical representation aids restarting with a different number of processes

Defining a Checkpoint

- Need enough to restart
  - Header information
    - *Size of problem (e.g. matrix dimensions)*
    - *Description of environment (e.g. input parameters)*
  - Program state
    - *Should represent the global (canonical) view of the data*
- Ideally stored in a convenient container
  - Single file!
- If all processes checkpoint at once, naturally a parallel, collective operation

Life Checkpoint/Restart API

- Define an interface for checkpoint/restart for the row-block distributed Life code
- Five functions:
  - MLIFEIO_Init
  - MLIFEIO_Finalize
  - MLIFEIO_Checkpoint
  - MLIFEIO_Can_restart
  - MLIFEIO_Restart
- All functions are <u>collective</u>
- Once the interface is defined, we can implement it for different back-end formats

Life Checkpoint

- ```
  MLIFEIO_Checkpoint(char      *prefix,
                     int      **matrix,
                     int        rows,
                     int        cols,
                     int        iter,
                     MPI_Info   info);
  ```
- Prefix is used to set filename
- Matrix is a reference to the data to store
- Rows, cols, and iter describe the data (header)
- Info is used for tuning purposes (more later!)
  ```

Life Checkpoint (Fortran)

- ```
  MLIFEIO_Checkpoint(prefix, matrix,
            rows, cols, iter, info )
  character*(*) prefix
  integer       rows, cols, iter
  integer       matrix(rows,cols)
  integer       info
  ```

- Prefix is used to set filename
- Matrix is a reference to the data to store
- Rows, cols, and iter describe the data (header)
- Info is used for tuning purposes (more later!)
  ```

stdio Life Checkpoint Code Walkthrough

- Points to observe
  - All processes call checkpoint routine
    - *Collective I/O from the viewpoint of the program*
  - Interface describes the *global* array
  - Output is independent of the number of processes

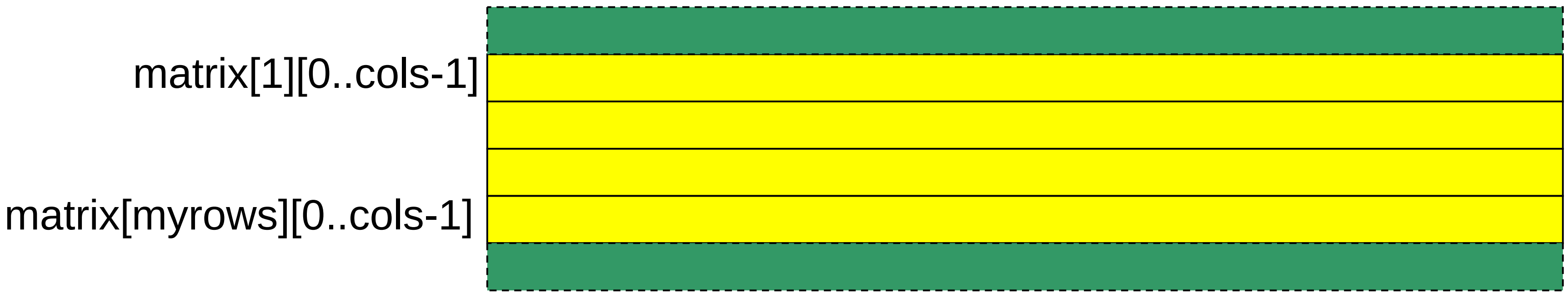
See mlife-io-stdout.c pp. 1-2 for code example.

Life stdout "checkpoint"

- The first implementation is one that simply prints out the "checkpoint" in an easy-to-read format
- MPI standard does not specify that all stdout will be collected in any particular way
  - Pass data back to rank 0 for printing
  - Portable!
  - Not scalable, but ok for the purpose of stdio

See mlife-io-stdout.c pp. 3 for code example.

Describing Data



- Lots of rows, all the same size
  - Rows are all allocated as one big block
  - Perfect for MPI_Type_vector

    MPI_Type_vector(count = myrows,
    blklen = cols, stride = cols+2, MPI_INT, &vectype);
  - Second type gets memory offset right

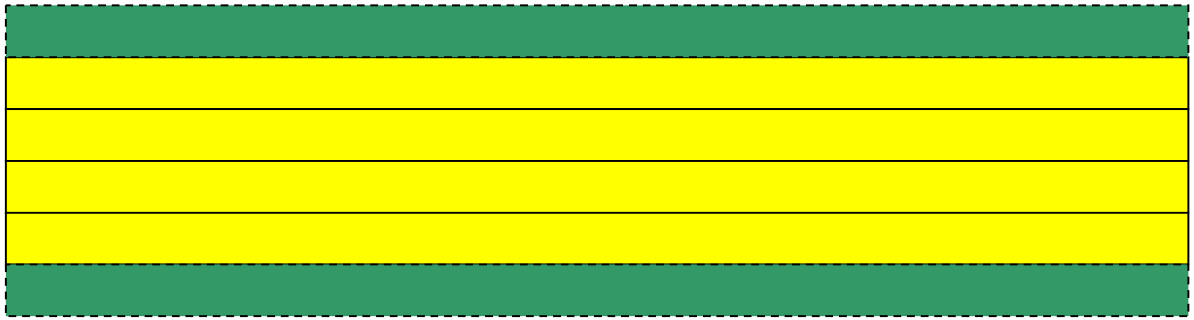
    MPI_Type_hindexed(count = 1, len = 1,
    disp = &matrix[1][1], vectype, &type);

See mlife-io-stdout.c pp. 4-6 for code example.

Describing Data (Fortran)

Matrix(1,0:cols-1)

Matrix(myrows,0:cols-1)



- Lots of rows, all the same size
  - Rows are all allocated as one big block
  - Perfect for MPI_Type_vector

    Call MPI_Type_vector(count = myrows,
    blklen = cols, stride = cols+2, MPI_INTEGER, vectype, ierr)

Life Checkpoint/Restart Notes

- MLIFEIO_Init
  - Duplicates communicator to avoid any collisions with other communication
- MLIFEIO_Finalize
  - Frees the duplicated communicator
- MLIFEIO_Checkpoint and _Restart
  - MPI_Info parameter is used for tuning I/O behavior

Note: Communicator duplication may not always be necessary, but is good practice for safety

See mlife-io-stdout.c pp. 1-8 for code example.

Parallel I/O and MPI

- The stdio checkpoint routine works but is not parallel
  - One process is responsible for all I/O
  - Wouldn't want to use this approach for real
- How can we get the full benefit of a parallel file system?
  - We first look at how parallel I/O works in MPI
  - We then implement a fully parallel checkpoint routine
    - *Because it will use the same interface, we can use it without changing the rest of the parallel life code*

Why MPI is a Good Setting for Parallel I/O

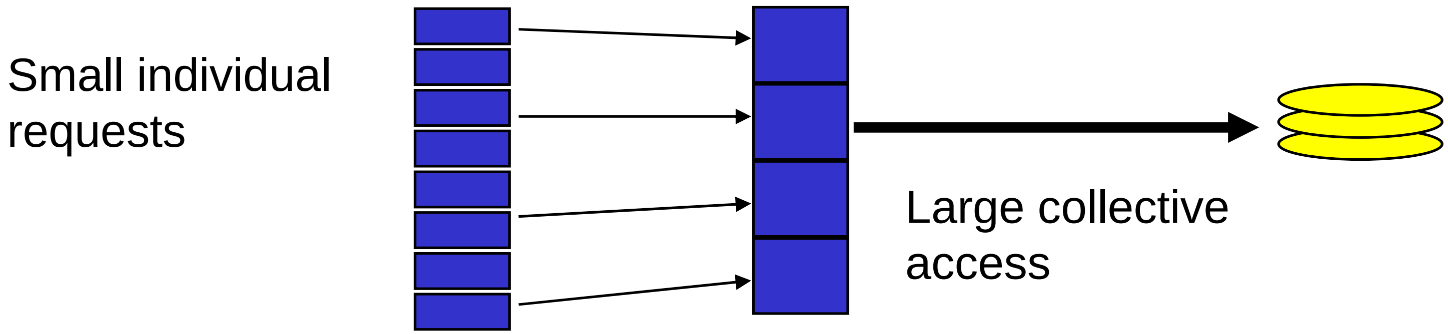
- Writing is like sending and reading is like receiving.
- Any parallel I/O system will need:
  - collective operations
  - user-defined datatypes to describe both memory and file layout
  - communicators to separate application-level message passing from I/O-related message passing
  - non-blocking operations
- I.e., lots of MPI-like machinery

What does Parallel I/O Mean?

- At the program level:
  - Concurrent reads or writes from multiple processes to a common file
- At the system level:
  - A parallel file system and hardware that support such concurrent access

Collective I/O and MPI

- A critical optimization in parallel I/O
- All processes (in the communicator) must call the collective I/O function
- Allows communication of “big picture” to file system
  - Framework for I/O optimizations at the MPI-IO layer
- Basic idea: build large blocks, so that reads/writes in I/O system will be large
  - Requests from different processes may be merged together
  - Particularly effective when the accesses of different processes are noncontiguous and interleaved



Collective I/O Functions

- **`MPI_File_write_at_all`**, etc.
  - **`_all`** indicates that all processes in the group specified by the communicator passed to **`MPI_File_open`** will call this function
  - **`_at`** indicates that the position in the file is specified as part of the call; this provides thread-safety and clearer code than using a separate "seek" call
- Each process specifies only its own access information — the argument list is the same as for the non-collective functions

MPI-IO Life Checkpoint Code Walkthrough

- Points to observe
  - Use of a user-defined MPI datatype to handle the local array
  - Use of MPI_Offset for the offset into the file
    - *“Automatically” supports files larger than 2GB if the underlying file system supports large files*
  - Collective I/O calls
    - *Extra data on process 0*

See mlife-io-mpiio.c pp. 1-2 for code example.

Life MPI-IO Checkpoint/Restart

- We can map our collective checkpoint directly to a single collective MPI-IO file write: MPI_File_write_at_all
  - Process 0 writes a little extra (the header)
- On restart, two steps are performed:
  - Everyone reads the number of rows and columns from the header in the file with MPI_File_read_at_all
    - *Sometimes faster to read individually and bcast (see later example)*
  - If they match those in current run, a second collective call used to read the actual data
    - *Number of processors can be different*

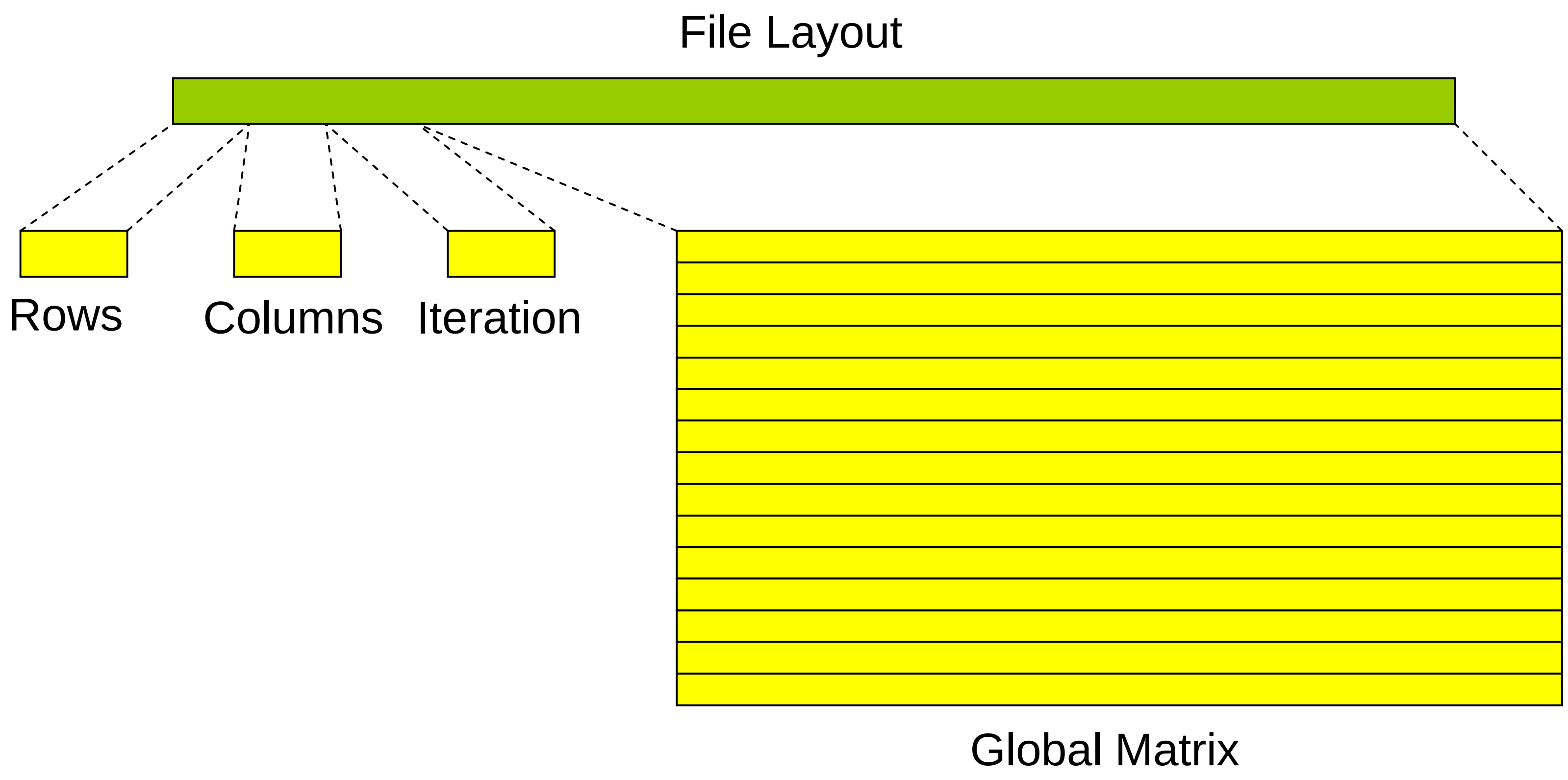
See mlife-io-mpiio.c pp. 3-6 for code example.

Describing Header and Data

- Data is described just as before
- Create a struct wrapped around this to describe the header as well:
  - no. of rows
  - no. of columns
  - Iteration no.
  - data (using previous type)

See mlife-io-mpiio.c pp. 7 for code example.

Placing Data in Checkpoint



Note: We store the matrix in global, canonical order with no ghost cells.

See mlife-io-mpiio.c pp. 9 for code example.

The Other Collective I/O Calls

- `MPI_File_seek`
- `MPI_File_read_all`
- `MPI_File_write_all`
- `MPI_File_read_at_all`
- `MPI_File_write_at_all`
- `MPI_File_read_ordered`
- `MPI_File_write_ordered`

like Unix I/O

combine seek and I/O for thread safety

use shared file pointer