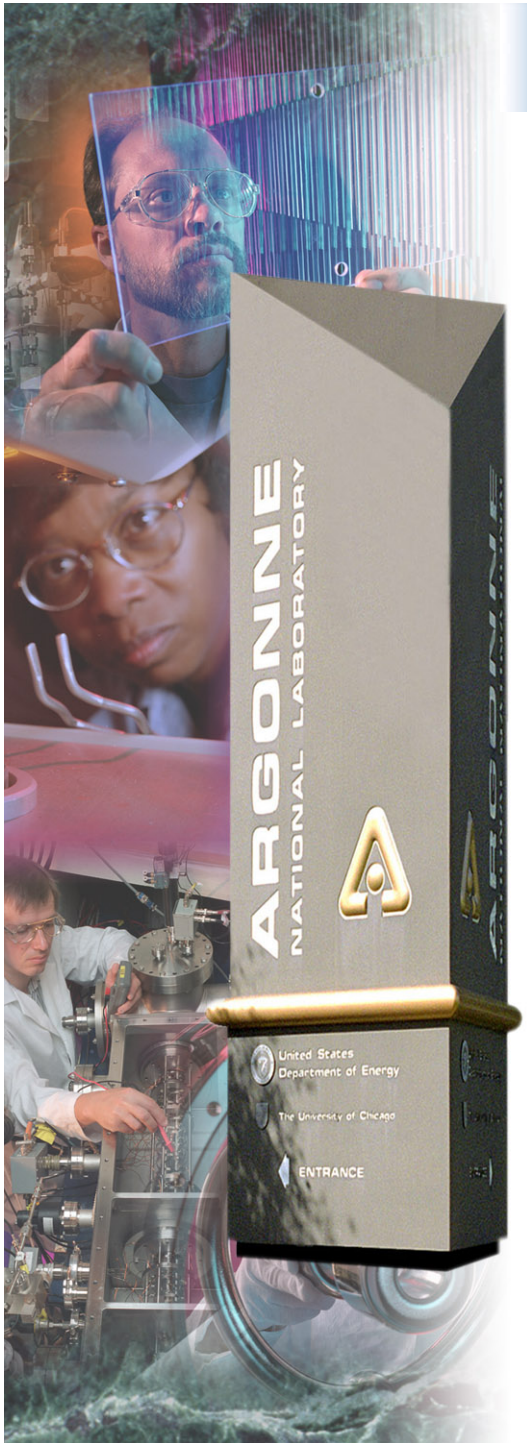


ARGONNE NATIONAL LABORATORY

Using MPI-2: A Problem-Based Approach

William Gropp, Rusty Lusk

Mathematics and Computer Science Division

THE UNIVERSITY OF CHICAGO

Office of Science
U.S. DEPARTMENT OF ENERGY

Argonne National Laboratory is managed by
The University of Chicago for the U.S. Department of Energy

Outline

- Introduction
  - MPI-1 Status, MPI-2 Status
- Life, 1D Decomposition
  - point-to-point
  - checkpoint/restart
    - *stdout*
    - *MPI-IO*
  - RMA
    - *fence*
    - *post/start/complete/wait*
- Life, 2D Decomposition
  - point-to-point
  - RMA

MPI-1

- MPI is a message-passing library interface standard.
  - Specification, not implementation
  - Library, not a language
  - Classical message-passing programming model
- MPI was defined (1994) by a broadly-based group of parallel computer vendors, computer scientists, and applications developers.
  - 2-year intensive process
- Implementations appeared quickly and now MPI is taken for granted as vendor-supported software on any parallel machine.
- Free, portable implementations exist for clusters (MPICH, LAM, OpenMPI) and other environments (MPICH)

MPI-2

- Same process of definition by MPI Forum
- MPI-2 is an extension of MPI
  - Extends the message-passing *model*.
    - *Parallel I/O*
    - *Remote memory operations (one-sided)*
    - *Dynamic process management*
  - Adds other functionality
    - *C++ and Fortran 90 bindings*
      - similar to original C and Fortran-77 bindings
    - *External interfaces*
    - *Language interoperability*
    - *MPI interaction with threads*

MPI-2 Implementation Status

- Most parallel computer vendors now support MPI-2 on their machines
  - Except in some cases for the dynamic process management functions, which require interaction with other system software
- Cluster MPIs, such as MPICH2 and LAM, support most of MPI-2 including dynamic process management
- Our examples here have all been run on MPICH2

Our Approach in this Tutorial

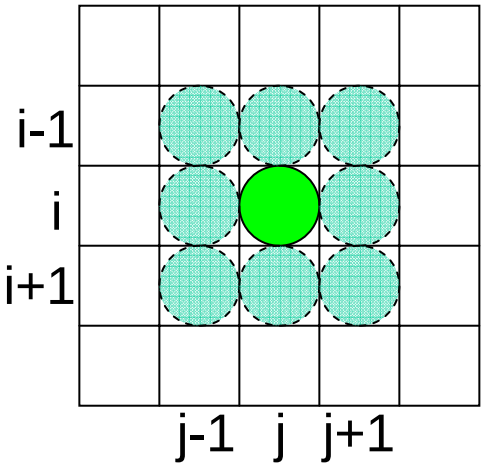
- Example driven
  - Structured data (Life)
- Show solutions that use the MPI-2 support for parallel I/O and RMA
  - Walk through actual code
- We assume familiarity with MPI-1

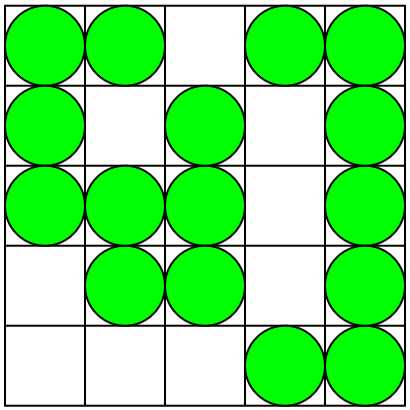
Conway's Game of Life

- 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

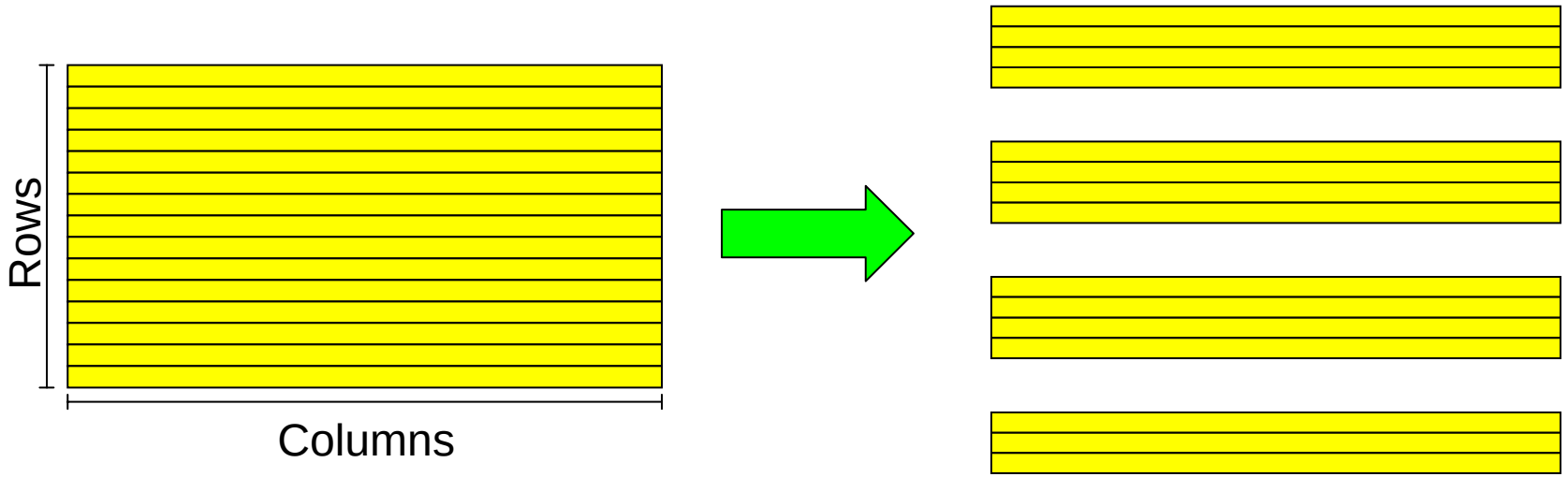
- For the non-parallel version, we:
  - Allocate a 2D matrix to hold state
    - *Actually two matrices, and we will swap them between steps*
  - Initialize the matrix
    - *Force boundaries to be "dead"*
    - *Randomly generate states inside*
  - At each time step:
    - *Calculate each new cell state 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

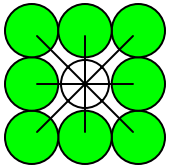
Step 1: Description of Decomposition

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

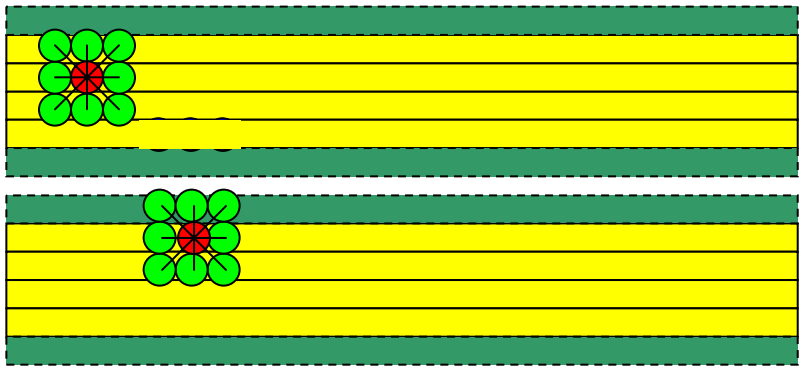


Step 2: Communication

- “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

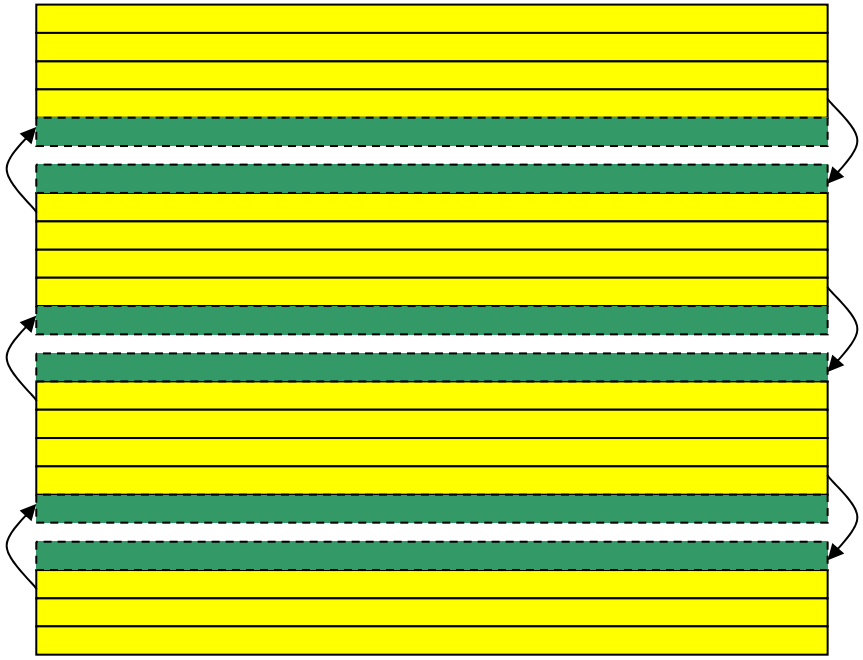


Step 3: Define the Local Arrays

- 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*

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

Note: Parsing Arguments

- MPI standard does not 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 collective
- 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

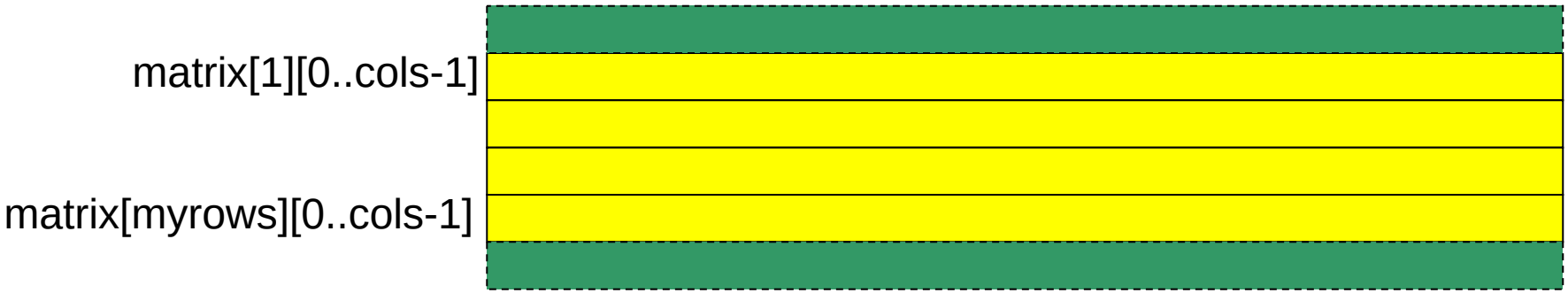
See mlife-io-stdout.c pp. 1-2 for code example.

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



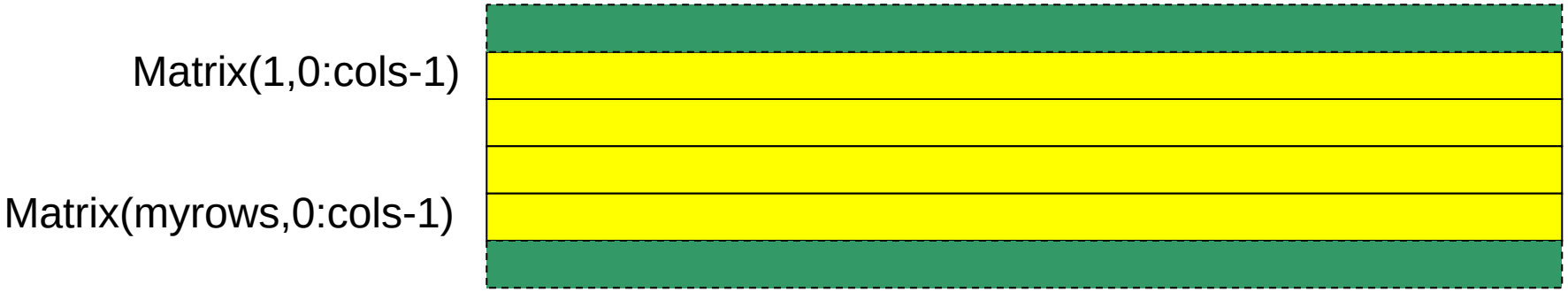
- Lots of rows, all the same size
  - Rows are all allocated as one big block
  - Perfect for MPI_Type_vector

    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)



- 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

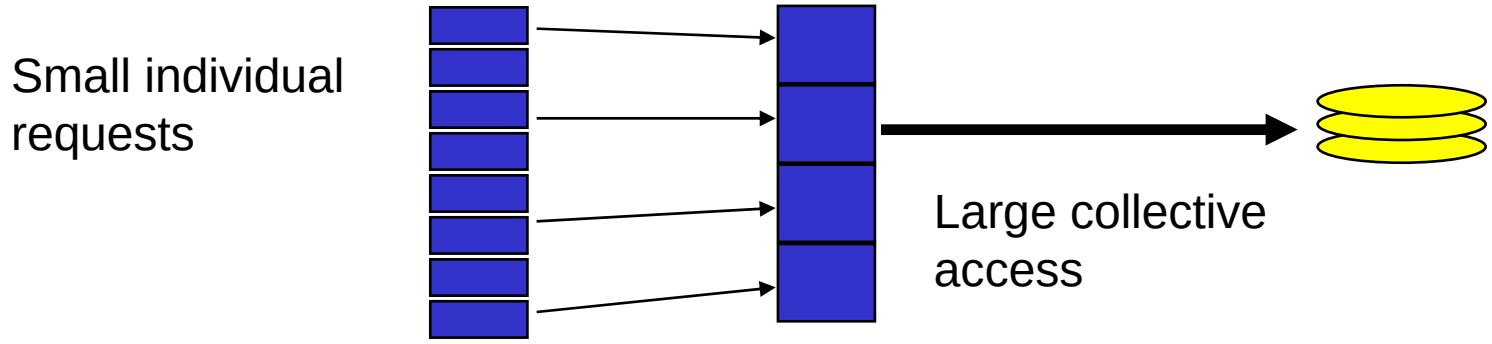
- Writing is like sending and reading is like receiving.
- Any parallel I/O system will need:
  - collective operations
  - user-defined datatypes to describe both memory and file layout
  - communicators to separate application-level message passing from I/O-related message passing
  - non-blocking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

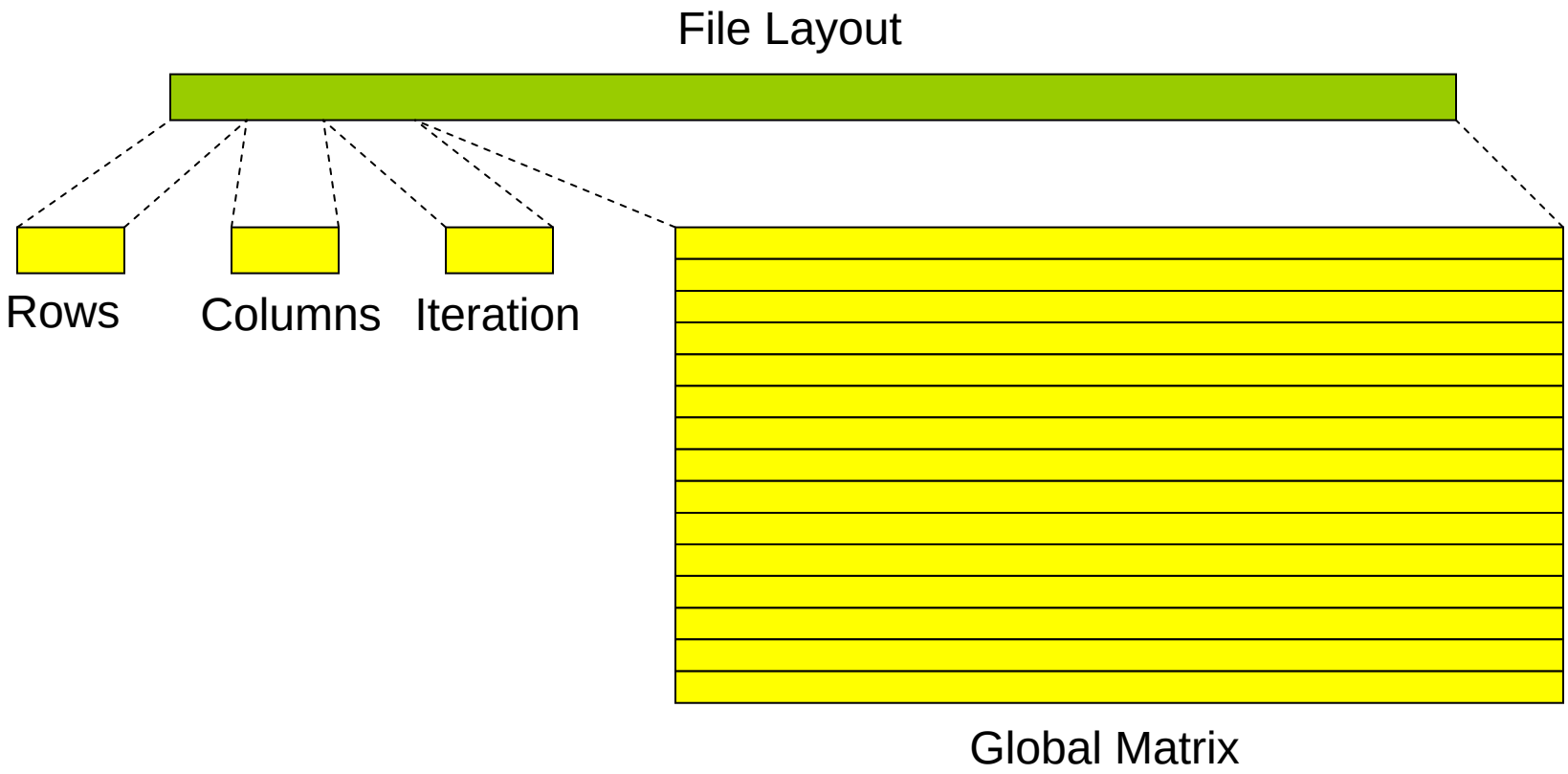
See mlife-io-mpiio.c pp. 3-6 for code example.

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

Portable Checkpointing

Portable File Formats

- Ad-hoc file formats
  - Difficult to collaborate
  - Cannot leverage post-processing tools
- MPI provides external32 data encoding
- High level I/O libraries
  - netCDF and HDF5
  - Better solutions than external32
    - *Define a "container" for data*
      - Describes contents
      - May be queried (self-describing)
    - *Standard format for metadata about the file*
    - *Wide range of post-processing tools available*

File Interoperability in MPI-IO

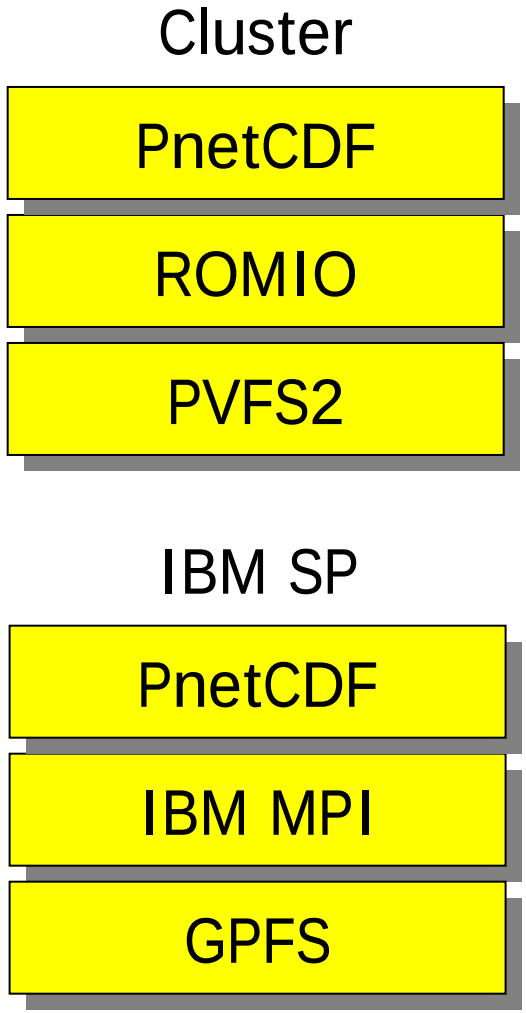
- Users can optionally create files with a portable binary data representation
- "datarep" parameter to **`MPI_File_set_view`**
- **`native`** - default, same as in memory, not portable
- **`external32`** - a specific representation defined in MPI, (basically 32-bit big-endian IEEE format), portable across machines and MPI implementations
- **`internal`** – implementation-defined representation providing an implementation-defined level of portability
  - Not used by anyone we know of…

Higher Level I/O Libraries

- Scientific applications work with structured data and desire more self-describing file formats
- netCDF and HDF5 are two popular “higher level” I/O libraries
  - Abstract away details of file layout
  - Provide standard, portable file formats
  - Include metadata describing contents
- For parallel machines, these should be built on top of MPI-IO
  - HDF5 has an MPI-IO option
    - *http://hdf.ncsa.uiuc.edu/HDF5/*

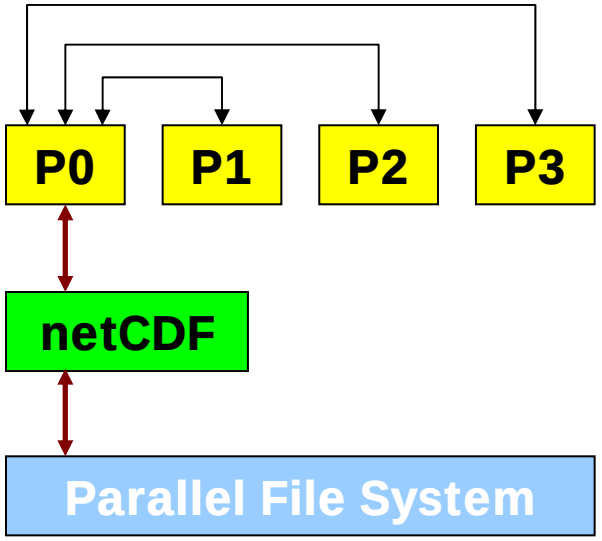
Parallel netCDF (PnetCDF)

- (Serial) netCDF
  - API for accessing multi-dimensional data sets
  - Portable file format
  - Popular in both fusion and climate communities
- Parallel netCDF
  - Very similar API to netCDF
  - Tuned for better performance in today's computing environments
  - Retains the file format so netCDF and PnetCDF applications can share files
  - PnetCDF builds on top of any MPI-IO implementation

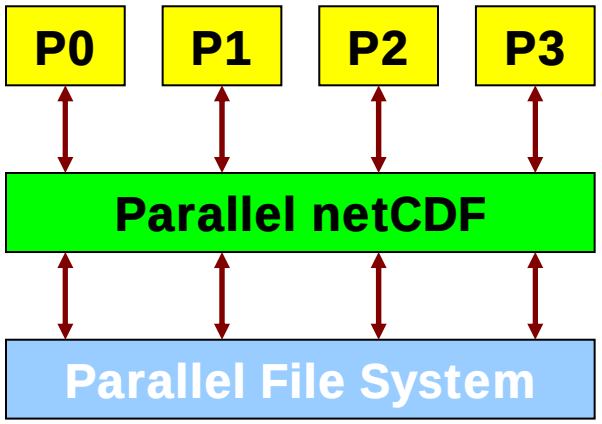


I/O in netCDF and PnetCDF

- (Serial) netCDF
  - Parallel read
    - *All processes read the file independently*
    - *No possibility of collective optimizations*
  - Sequential write
    - *Parallel writes are carried out by shipping data to a single process*
    - *Just like our stdout checkpoint code*



- PnetCDF
  - Parallel read/write to shared netCDF file
  - Built on top of MPI-IO which utilizes optimal I/O facilities of the parallel file system and MPI-IO implementation
  - Allows for MPI-IO hints and datatypes for further optimization



Life PnetCDF Checkpoint/Restart

- Third implementation of MLIFEIO interface
- Stores matrix as a two-dimensional array of integers in the netCDF file format
  - Same canonical ordering as in MPI-IO version
- Iteration number stored as an attribute

PnetCDF Life Checkpoint Code Walkthrough

- Points to observe
  - Creating a netCDF file
  - Defining dimensions
  - Defining variables
  - Storing attributes
  - Discovering dimensions on restart

See mlife-io-pnetcdf.c pp. 1-6 for code example.

Discovering Variable Dimensions

- Because netCDF is self-describing, applications can inquire about data in netCDF files:

```
err = ncmpi_inq_dimlen(ncid,
      dims[0], &coldimsz);
```

- Allows us to discover the dimensions of our matrix at restart time

See mlife-io-pnetcdf.c pp. 7-8 for code example.

Exchanging Data with RMA

Revisiting Mesh Communication

- Recall how we designed the parallel implementation
  - Determine source and destination data
- Do not need full generality of send/receive
  - Each process can completely define what data needs to be moved to itself, relative to each processes local mesh
    - *Each process can "get" data from its neighbors*
  - Alternately, each can define what data is needed by the neighbor processes
    - *Each process can "put" data to its neighbors*

Remote Memory Access

- Separates data transfer from indication of completion (synchronization)
- In message-passing, they are combined

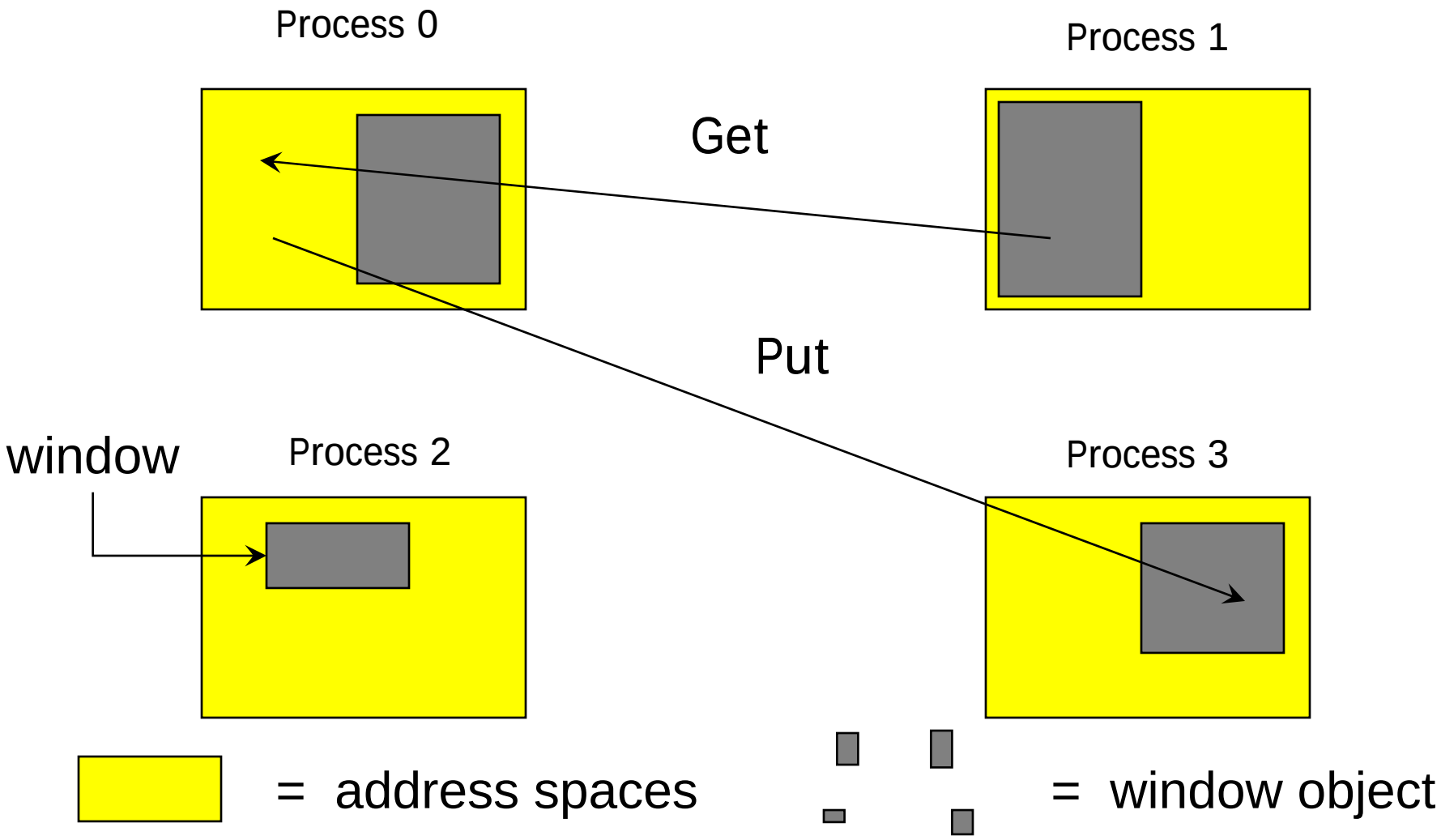
Proc 0	Proc 1
store	
send	receive
	load

Proc 0	Proc 1
fence	fence
put	
fence	fence
	load
or	
store	
fence	fence
	get

Remote Memory Access in MPI-2 (also called One-Sided Operations)

- Goals of MPI-2 RMA Design
  - Balancing efficiency and portability across a wide class of architectures
    - *shared-memory multiprocessors*
    - *NUMA architectures*
    - *distributed-memory MPP's, clusters*
    - *Workstation networks*
  - Retaining "look and feel" of MPI-1
  - Dealing with subtle memory behavior issues: cache coherence, sequential consistency

Remote Memory Access Windows and Window Objects

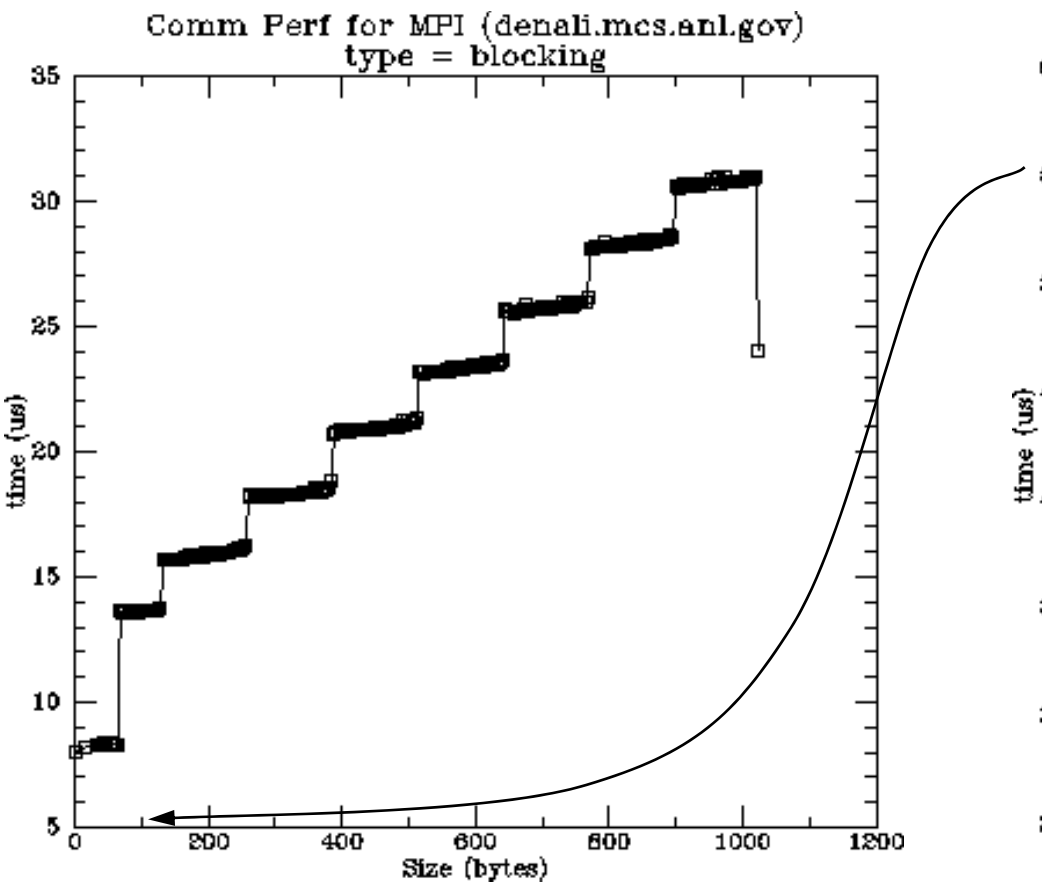


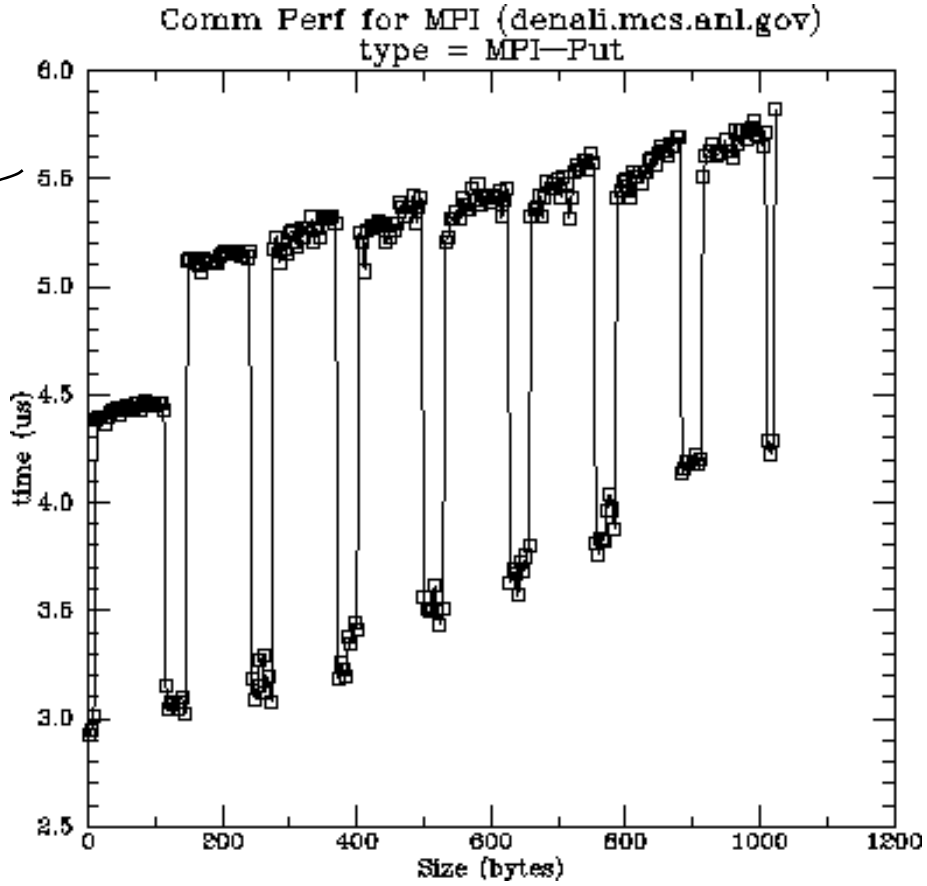
Basic RMA Functions for Communication

- **`MPI_Win_create`** exposes local memory to RMA operation by other processes in a communicator
  - Collective operation
  - Creates window object
- **`MPI_Win_free`** deallocates window object

- **`MPI_Put`** moves data from local memory to remote memory
- **`MPI_Ge`**t retrieves data from remote memory into local memory
- **`MPI_Accumulate`** updates remote memory using local values
- Data movement operations are non-blocking
- **Subsequent synchronization on window object needed to ensure operation is complete**

Performance of RMA





Caveats: On SGI, MPI_Put uses specially allocated memory

Advantages of RMA Operations

- Can do multiple data transfers with a single synchronization operation
  - like BSP model
- Bypass tag matching
  - effectively precomputed as part of remote offset
- Some irregular communication patterns can be more economically expressed
- Can be significantly faster than send/receive on systems with hardware support for remote memory access, such as shared memory systems

Irregular Communication Patterns with RMA

- If communication pattern is not known *a priori*, the send-recv model requires an extra step to determine how many sends-recvs to issue
- RMA, however, can handle it easily because only the origin or target process needs to issue the put or get call
- This makes dynamic communication easier to code in RMA

RMA Window Objects

```
MPI_Win_create(base, size, disp_unit, info,
               comm, win)
```

- Exposes memory given by **`(base, size)`** to RMA operations by other processes in **`comm`**
- **`win`** is window object used in RMA operations
- **`disp_unit`** scales displacements:
  - 1 (no scaling) or **`sizeof(type)`**, where window is an array of elements of type **`type`**
  - Allows use of array indices
  - Allows heterogeneity

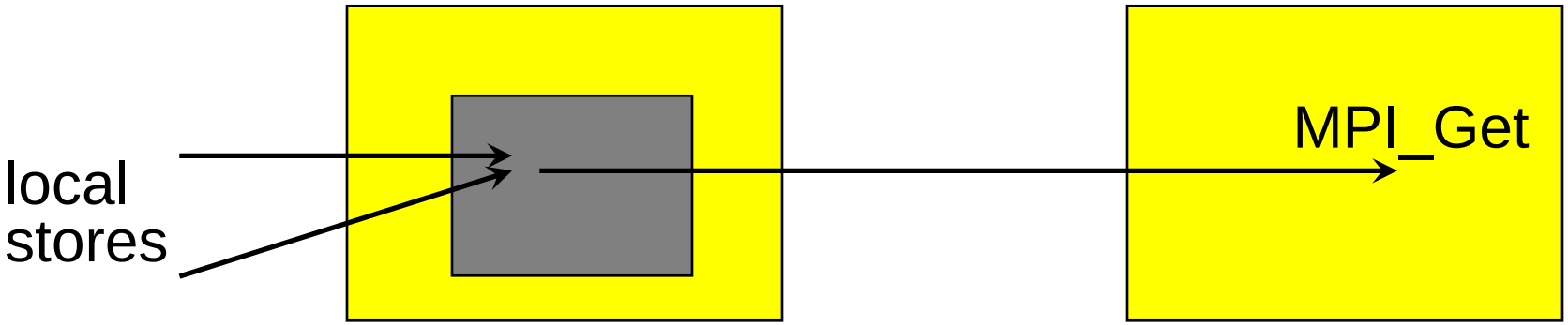
RMA Communication Calls

- **`MPI_Put`** - stores into remote memory
- **`MPI_Get`** - reads from remote memory
- **`MPI_Accumulate`** - updates remote memory
- All are non-blocking: data transfer is described, maybe even initiated, but may continue after call returns
- Subsequent synchronization on window object is needed to ensure operations are complete

Put, Get, and Accumulate

- **`MPI_Put(origin_addr, origin_count,`**
  `origin_datatype,`
  `target_rank, target_offset,`
  `target_count, target_datatype,`
  `window)`
- **`MPI_Get( ... )`**
- **`MPI_Accumulate( ..., op, ... )`**
- **`op`** is as in **`MPI_Reduce`**, but no user-defined operations are allowed

The Synchronization Issue



- Issue: Which value is retrieved?
  - Some form of synchronization is required between local load/stores and remote get/put/accumulates
- MPI provides multiple forms

Synchronization with Fence

Simplest methods for synchronizing on window objects:

- **`MPI_Win_fence`** - like barrier, supports BSP model

Process 0	Process 1
MPI_Win_fence(win)	MPI_Win_fence(win)
MPI_Put MPI_Put	
MPI_Win_fence(win)	MPI_Win_fence(win)

Mesh Exchange Using MPI RMA

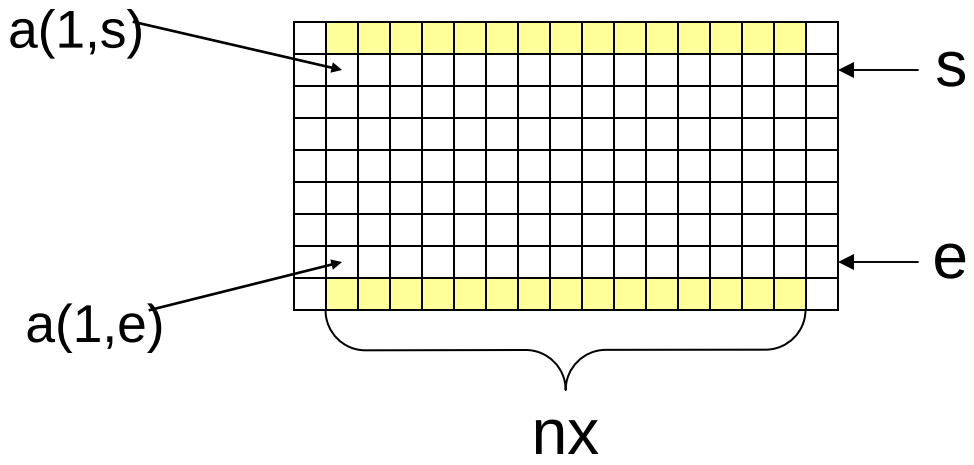
- Define the windows
  - Why – safety, options for performance (later)
- Define the data to move
- Mark the points where RMA can start and where it must complete (e.g., fence/put/put/fence)

Outline of 1D RMA Exchange

- Create Window object
- Computing target offsets
- Exchange operation

Computing the Offsets

- Offset to top ghost row
  - 1
- Offset to bottom ghost row
  - 1 + (# cells in a row)*(# of rows – 1)
  - = 1 + (nx + 2)*(e – s + 2)



Fence Life Exchange Code Walkthrough

- Points to observe
  - MPI_Win_fence is used to separate RMA accesses from non-RMA accesses
    - *Both starts and ends data movement phase*
  - Any memory may be used
    - *No special malloc or restrictions on arrays*
  - Uses same exchange interface as the point-to-point version

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

Comments on Window Creation

- MPI-2 provides MPI_SIZEOF for Fortran users
  - Not universally implemented
  - Use MPI_Type_size for portability
- Using a displacement size corresponding to a basic type allows use of put/get/accumulate on heterogeneous systems
  - Even when the sizes of basic types differ
- Displacement size also allows easier computation of offsets in terms of array index instead of byte offset

More on Fence

- MPI_Win_fence is collective over the group of the window object
- MPI_Win_fence is used to *separate*, not just complete, RMA and local memory operations
  - That is why there are *two* fence calls
- Why?
  - MPI RMA is designed to be portable to a wide variety of machines, including those without cache coherent hardware (including some of the fastest machines made)
  - See performance tuning for more info

Scalable Synchronization with Post/Start/Complete/Wait

- Fence synchronization is not scalable because it is collective over the group in the window object
- MPI provides a second synchronization mode: *Scalable Synchronization*
  - Uses four routines instead of the single MPI_Win_fence:
    - *2 routines to mark the begin and end of calls to RMA routines*
      - MPI_Win_start, MPI_Win_complete
    - *2 routines to mark the begin and end of access to the memory window*
      - MPI_Win_post, MPI_Win_wait
- P/S/C/W allows synchronization to be performed only among communicating processes

Synchronization with P/S/C/W

- Origin process calls MPI_Win_start and MPI_Win_complete
- Target process calls MPI_Win_post and MPI_Win_wait

Process 0	Process 1
MPI_Win_start(target_grp)	MPI_Win_post(origin_grp)
MPI_Put MPI_Put	
MPI_Win_complete(target_grp)	MPI_Win_wait(origin_grp)

P/S/C/W Life Exchange Code Walkthrough

- Points to Observe
  - Use of MPI group routines to describe neighboring processes
  - No change to MPI_Put calls
    - *You can start with MPI_Win_fence, then switch to P/S/C/W calls if necessary to improve performance*

See mlife-pscw.c pp. 1-4 for code example.

Life with 2D Block-Block Decomposition

Why Use a 2D Decomposition?

- More scalable due to reduced communication requirements
  - We can see why with a simple communication model.
  - Let the time to move n words from one process to another be $T_c = s + rn$
  - 1D decomposition time on p processes is
    - $T = 2(s+rn) + T_1/p$
  - 2D decomposition time on p processes is
    - $T = 4(s + r(n/\sqrt{p})) + T_1/p$
  - For large n, 2D decomposition has much smaller communication time
  - (Even stronger effect for 3D decompositions of 3D problems)

Designing the 2D Decomposition

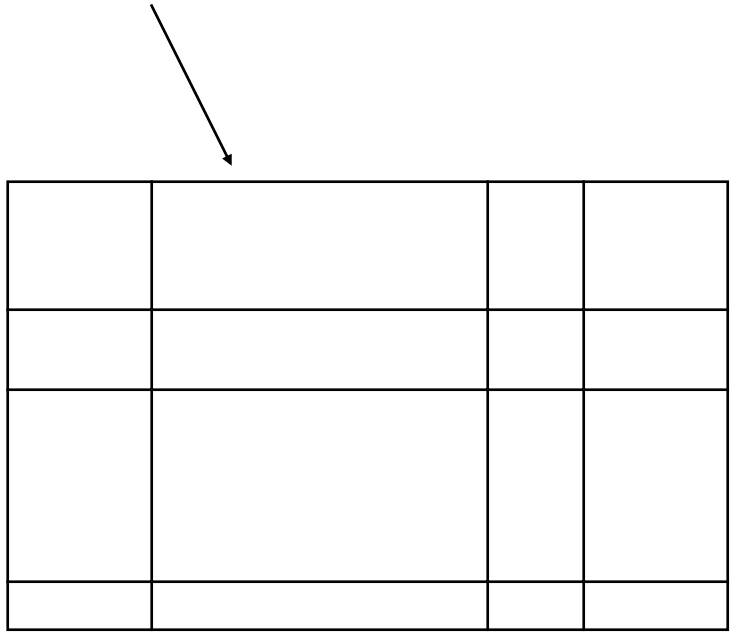
- Go back to global mesh view
- Define decomposition
- Define data to move
- Define local mesh

Mesh Exchange for 2D Decomposition

- Creating the datatypes
- Using fence
- Using scalable synchronization

Outline of 2D RMA Exchange

- Create Window Object
- Computing target offsets
  - Even for less regular decompositions
- Creating Datatypes
- Exchange Operation



Creating the Window

```
MPI_Win win;
int *localMesh;

/* nx is the number of (non-ghost) values in x, ny
 in y */
nx = ex - sx + 1;
ny = ey - sy + 1;
MPI_Win_create(localMesh,
               (ex-sx+3)*(ey-sy+3)*sizeof(int),
                sizeof(int), MPI_INFO_NULL,
                MPI_COMM_WORLD, &win);
```

- Nothing new here

Creating the Window (C++)

```
MPI::Win win;
int *localMesh;

// nx is the number of (non-ghost) values in x,
// ny in y
nx = ex - sx + 1;
ny = ey - sy + 1;
win = MPI::Win::Create(localMesh,
                       (ex-sx+3)*(ey-sy+3)*sizeof(int),
                        sizeof(int), MPI::INFO_NULL,
                        MPI::COMM_WORLD);
```

- Nothing new here

Creating the Window (Fortran)

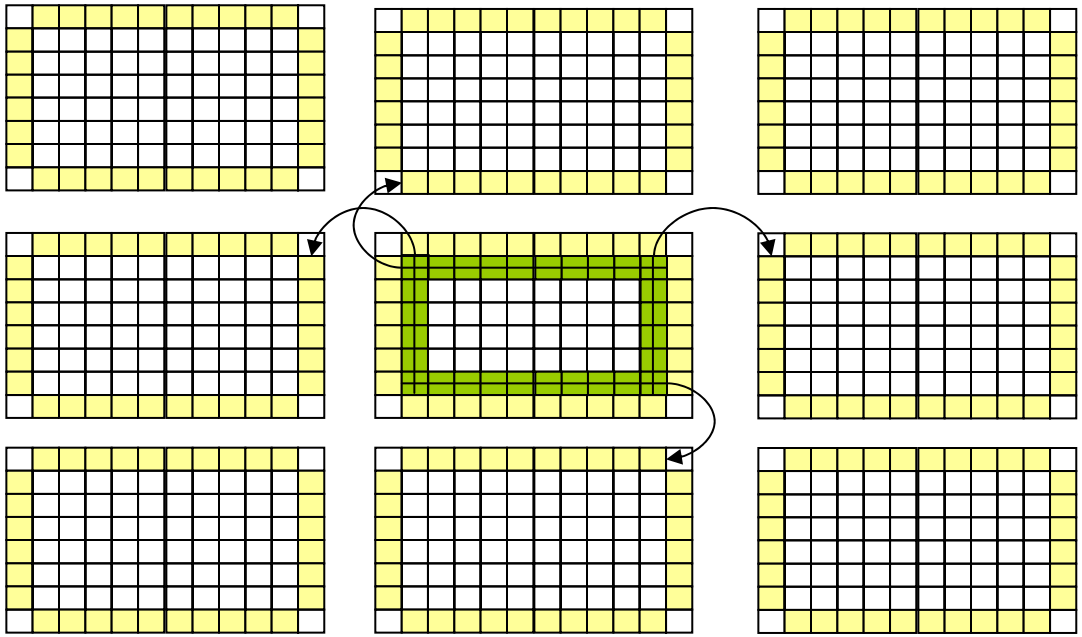
```
integer win, sizedouble, ierr
double precision a(sx-1:ex+1,sy-1:ey+1)

! nx is the number of (non-ghost) values in x, ny in y
nx = ex - sx + 1
ny = ey - sy + 1
call MPI_TYPE_SIZE(MPI_DOUBLE_PRECISION, sizedouble,&
                   ierr)
call MPI_WIN_CREATE(a, (ex-sx+3)*(ey-sy+3)*sizedouble, &
                    sizedouble, MPI_INFO_NULL, &
                    MPI_COMM_WORLD, win, ierr)
```

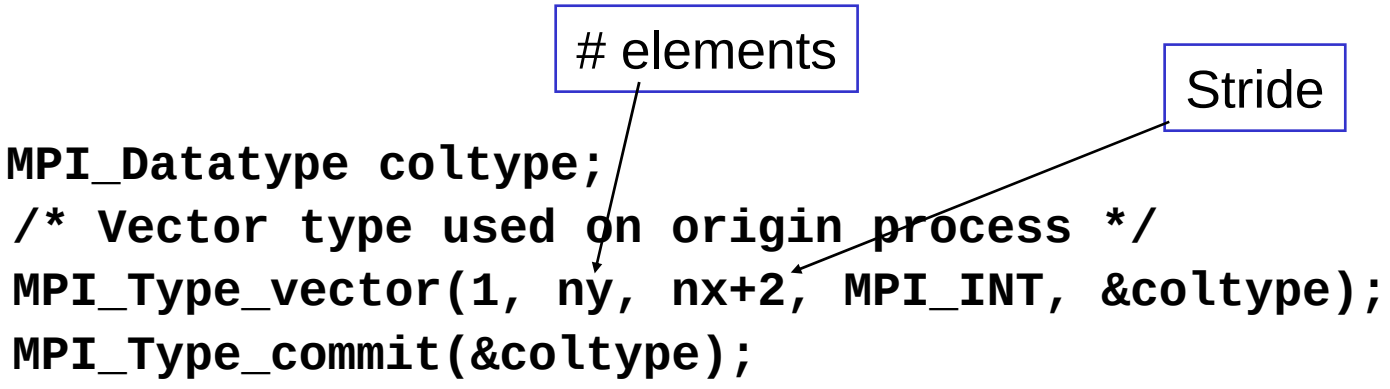
- Nothing new here

Computing Target Offsets

- Similar to 1D, but may include some computation since neighbor with shared boundary still needs to know the size of the other dimension as that is needed to compute the offsets

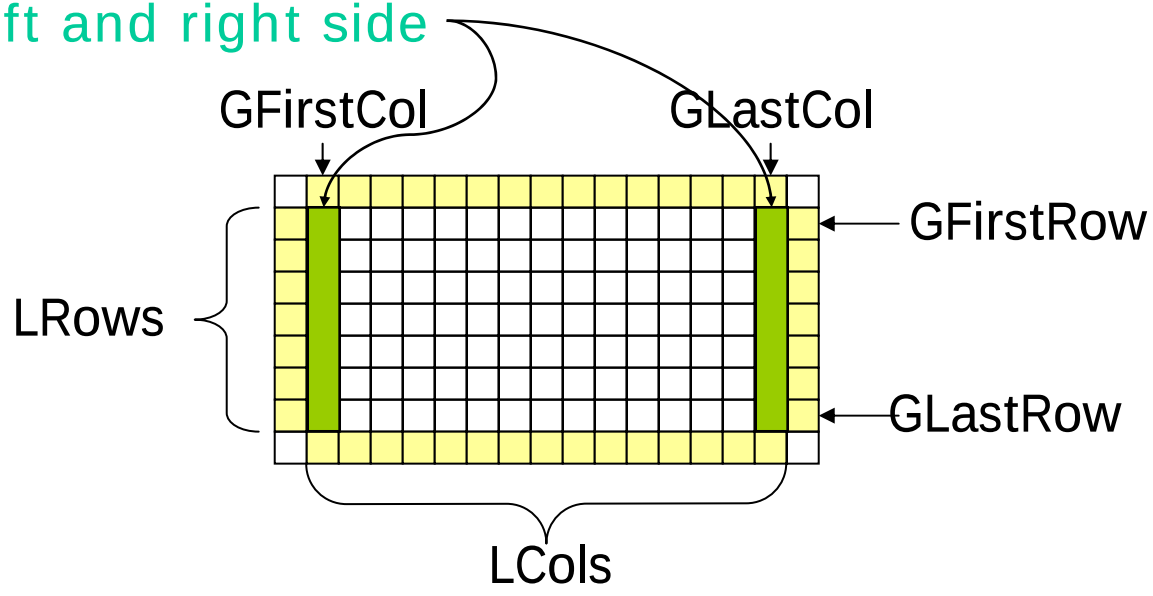


Creating Datatypes for Columns

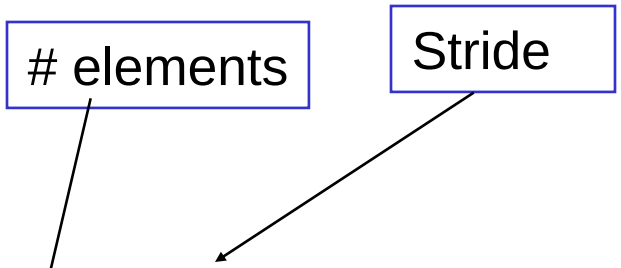



```
MPI_Datatype coltype;
/* Vector type used on origin process */
MPI_Type_vector(1, ny, nx+2, MPI_INT, &coltype);
MPI_Type_commit(&coltype);
```

- For both the left and right side

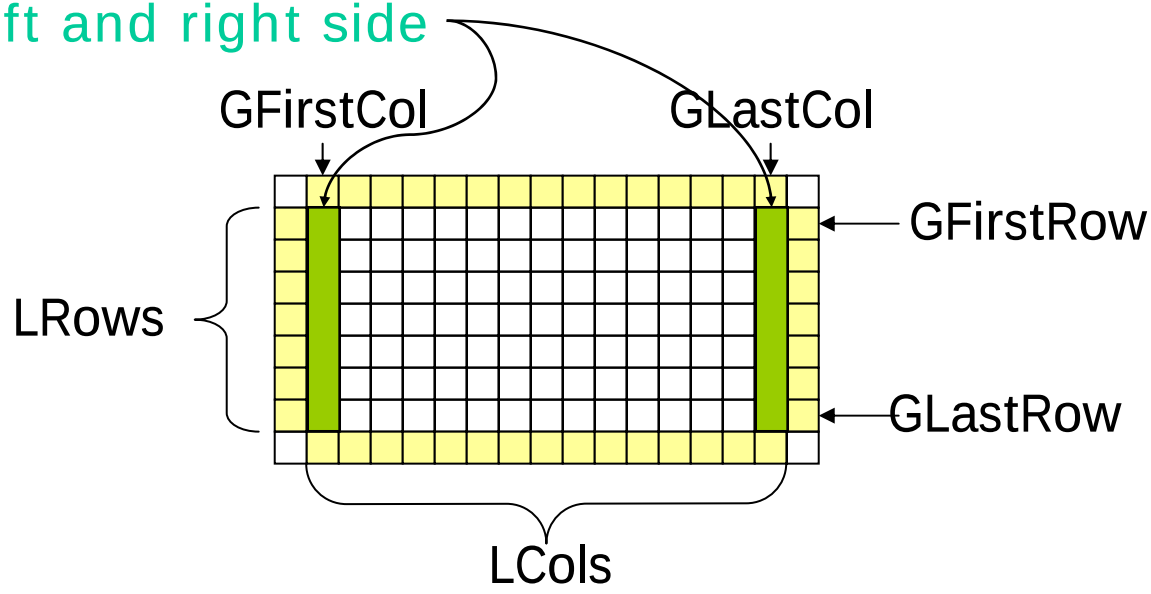


Creating Datatypes for Columns (C++)



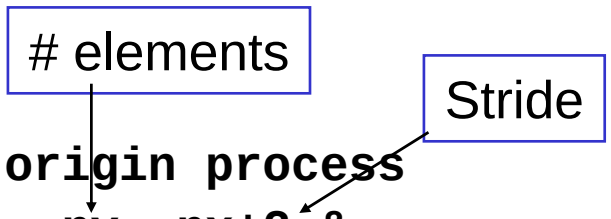

```
MPI::Datatype coltype;
// Vector type used on origin process
coltype = MPI::Type::Create_vector(1, ny, nx+2, MPI::INT );
coltype.Commit();
```

- For both the left and right side

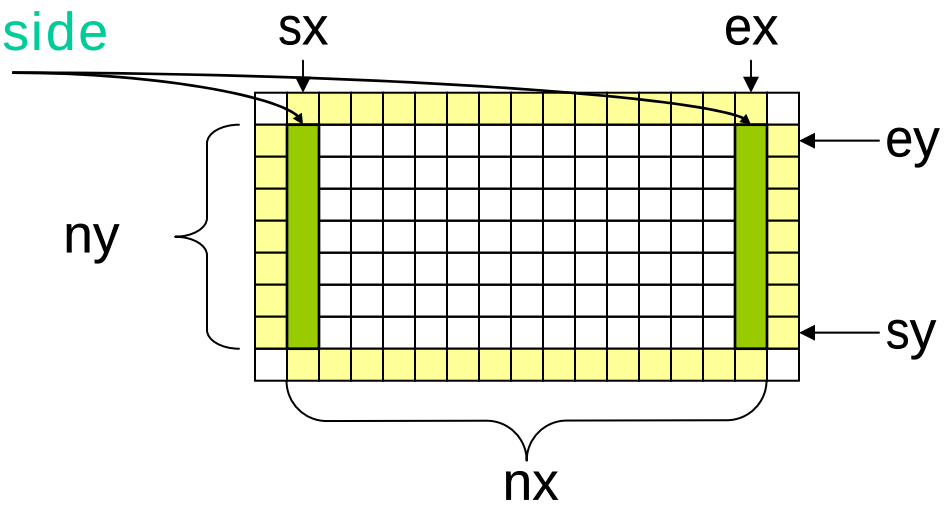


Creating Datatypes for Columns (Fortran)

```
integer coltype
! Vector type used on origin process
call MPI_TYPE_VECTOR(1, ny, nx+2,&
                     MPI_DOUBLE_PRECISION, &
                     coltype, ierr)
call MPI_TYPE_COMMIT(coltype, ierr)
```



- For both the left and right side



2D Life Code Walkthrough

- Points to observe
  - More complicated than 1D!
  - Communication of noncontiguous regions uses derived datatypes
- For the RMA version (mlife2d-fence)
  - Be careful in determining the datatype for the target process
  - Be careful in determining the offset
  - MPI_Win_fence must return before data may be used on target

See mlife2d.c, mlife2d-pt2pt.c, mlife2d-fence.c for code examples.

Conclusions

Designing Parallel Programs

- Common theme – think about the “global” object, then see how MPI can help you
- Also specify the largest amount of communication or I/O between “synchronization points”
  - Collective and noncontiguous I/O
  - RMA

Summary

- MPI-2 provides major extensions to the original message-passing model targeted by MPI-1.
- MPI-2 can deliver to libraries and applications portability across a diverse set of environments.
- Implementations are here now.
- Sources:
  - The MPI standard documents are available at
    `http://www.mpi-forum.org`
  - 2-volume book: *MPI - The Complete Reference*, available from MIT Press
  - *Using MPI* (Gropp, Lusk, and Skjellum) and *Using MPI-2* (Gropp, Lusk, and Thakur), MIT Press.
    - *Using MPI-2 also available in Japanese, from Pearson Education Japan*

Conclusions

- MPI is a proven, effective, portable parallel programming model
  - 26TF application on the Earth Simulator
- MPI has succeeded because
  - features are orthogonal (complexity is the product of the number of *features*, not routines)
  - programmer can control memory motion (critical in high-performance computing)
  - complex programs are no harder than easy ones
  - open process for defining MPI led to a solid design

More Information on Software

- MPICH2
  - Latest version available from www.mcs.anl.gov/mpi/mpich2
- More Information on PnetCDF
  - Parallel netCDF web site:
    http://www.mcs.anl.gov/parallel-netcdf/
  - Parallel netCDF mailing list:
    Mail to majordomo@mcs.anl.gov with the body "subscribe parallel-netcdf"
  - The SDM SciDAC web site:
    http://sdm.lbl.gov/sdmcenter/
- PETSc
  - http://www.mcs.anl.gov/petsc
- HDF5
  - http://hdf.ncsa.uiuc.edu/HDF5/

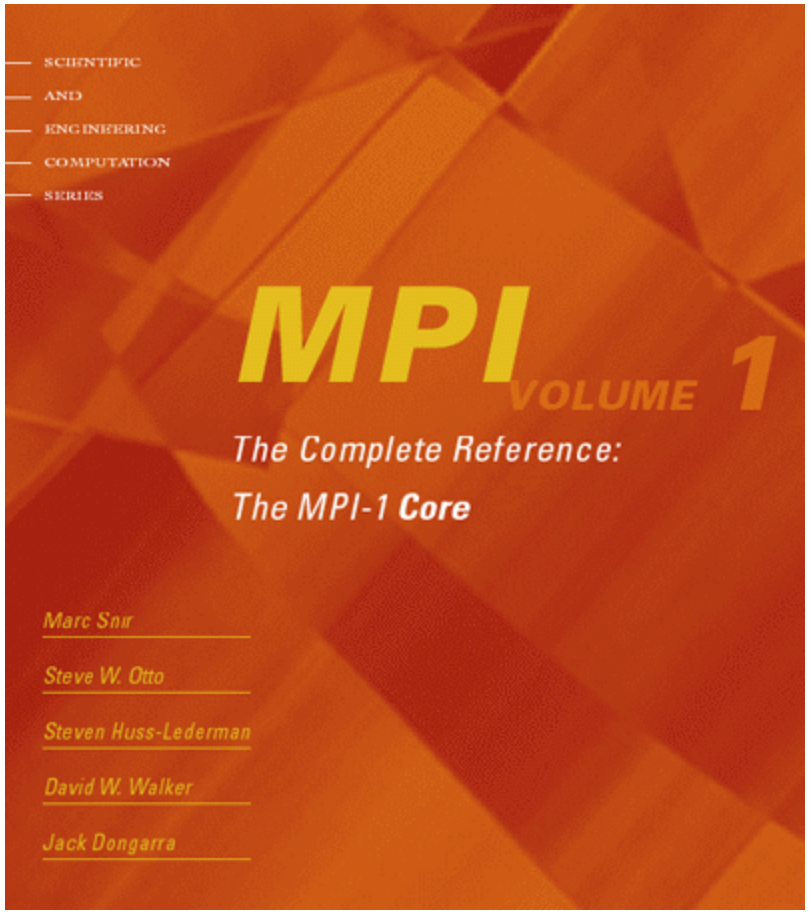
MPICH2

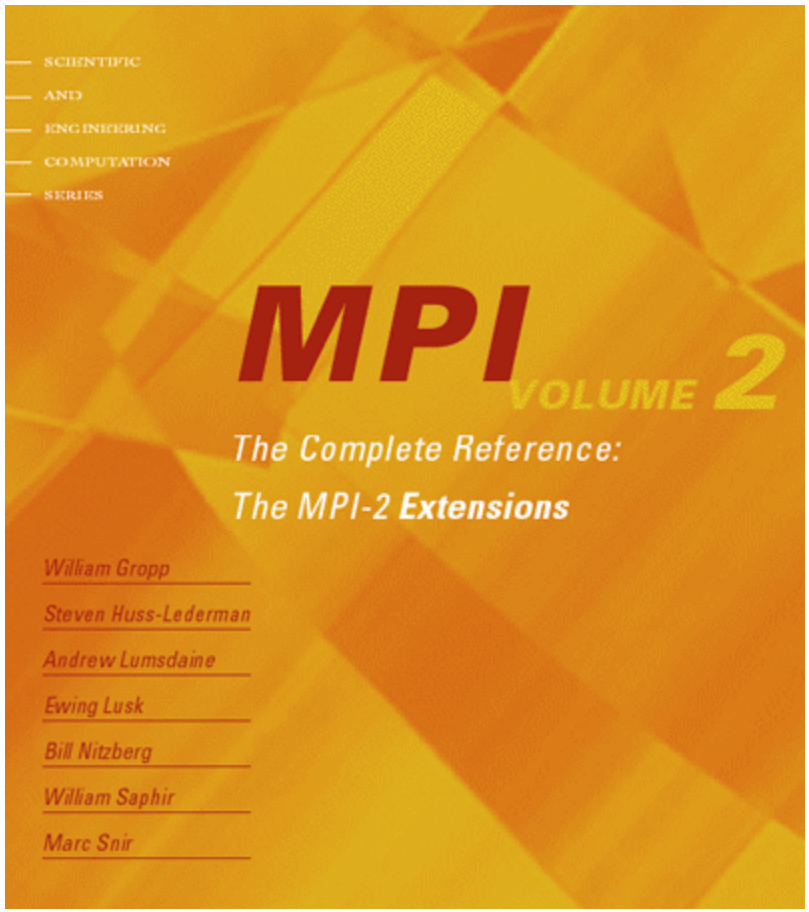
- Goals: same as MPICH
  - Research project, to explore scalability and performance, incorporate and test research results
  - Software project, to encourage use of MPI-2
- Scope: all of MPI-2
  - I/O
  - Dynamic
  - One-sided
  - All the obscure parts, too
  - Useful optional features recommended by the Standard (full mpiexec, singleton-init, thread safety)
  - Other useful features (debugging, profiling libraries, tools)

MPICH2

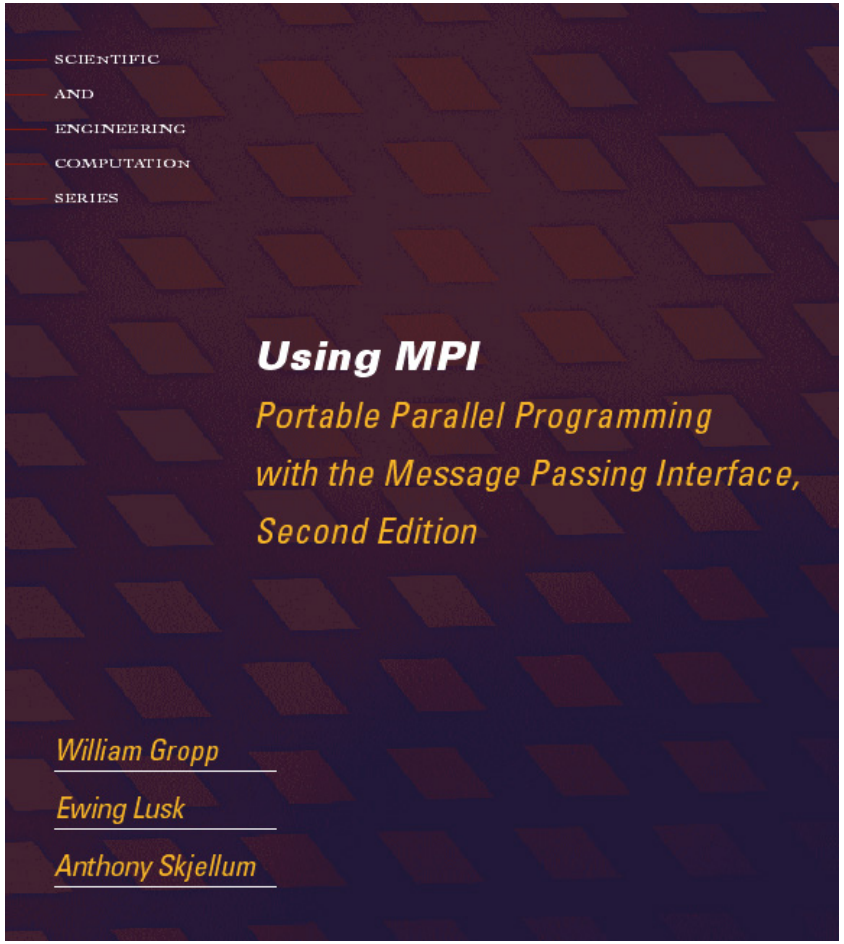
- Incorporates latest research into MPI implementation
  - Our own
    - *Collective operations*
    - *Optimizations for one-sided ops*
    - *Optimized datatype handling*
    - *I/O*
  - Others
    - *Collectives, for example*
- See recent EuroPVM and Cluster Proceedings
- In use by vendors
  - IBM on BG/L
  - Cray on Red Storm/XT3
  - Microsoft, Intel
  - Having vendors adapt MPICH2 into their products has helped make it efficient and robust

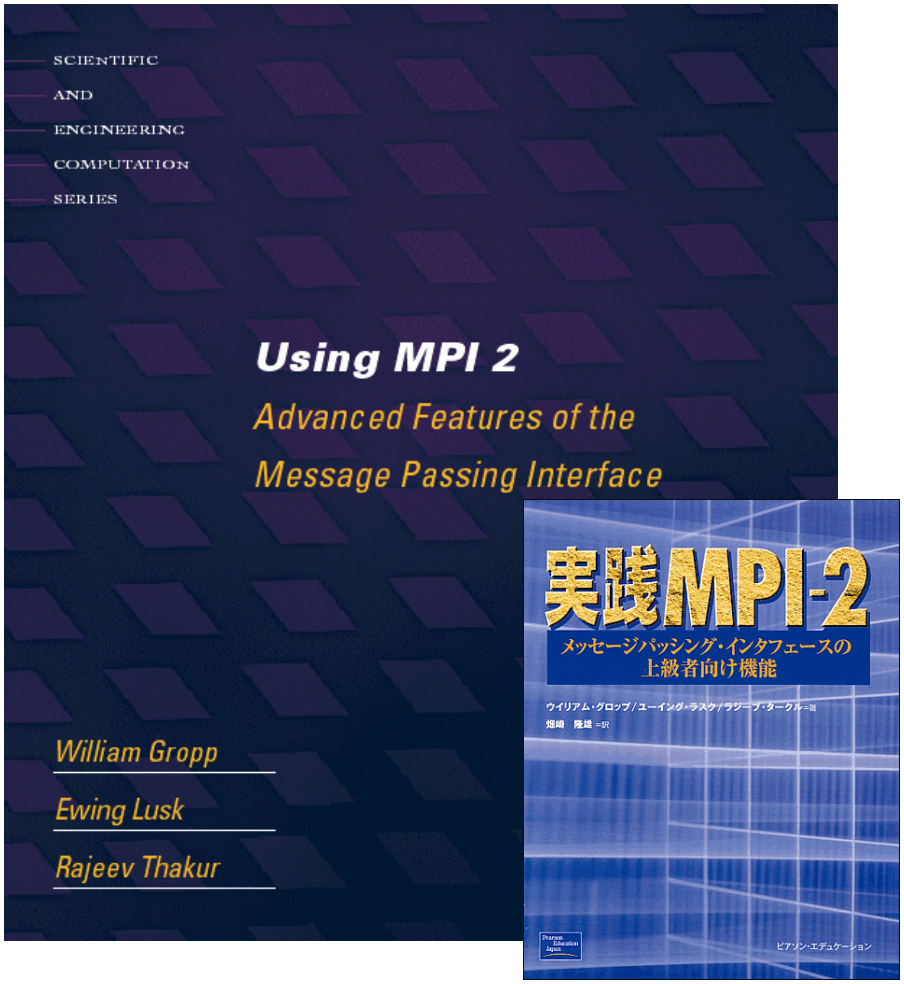
The MPI Standard (1 & 2)





Tutorial Material on MPI, MPI-2





http://www.mcs.anl.gov/mpi/{usingmpi,usingmpi2}