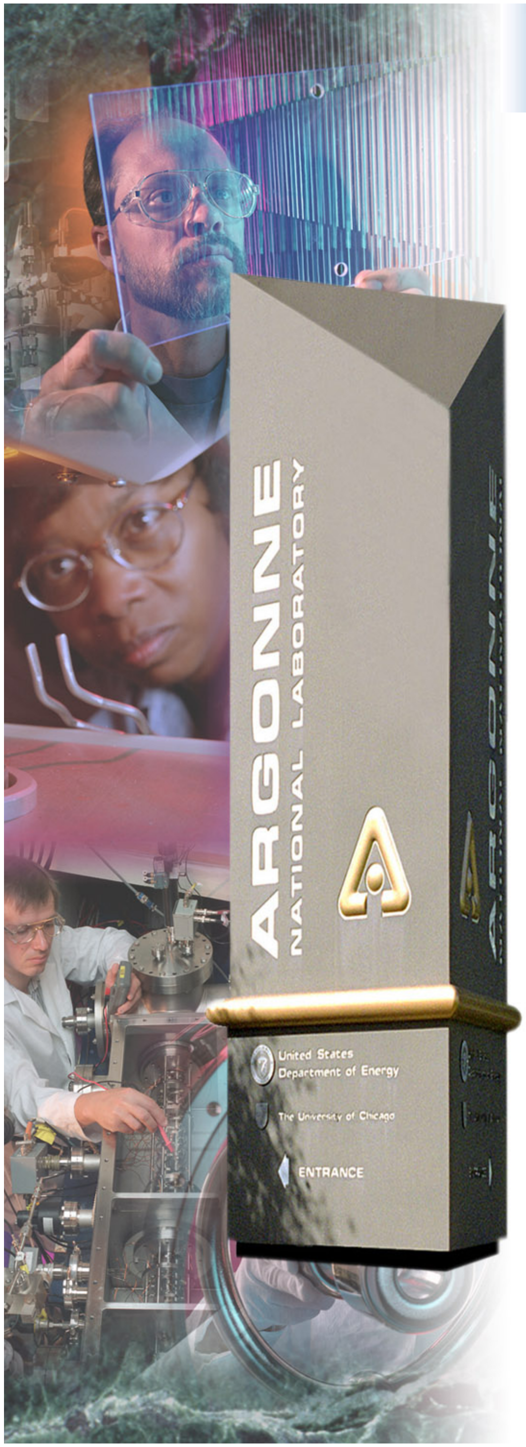
ARGONNE
NATIONAL LABORATORY



Advanced MPI:
I/O and One-Sided Communication

William Gropp, Rusty Lusk, Rob Ross, and Rajeev Thakur

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

Table of Contents

- Conway’s Game of Life – 8
  - Parallel I/O and Life – 19
  - Exchanging Data with RMA – 49
  - Life with 2D Block-Block Decomposition – 72
- Sparse Matrix I/O – 88
  - Writing Sparse Matrices – 117
- pNeo: Modeling the Human Brain – 118
- Passive Target RMA – 128
- Improving Performance – 157
  - Tuning MPI-IO – 158
  - Tuning RMA – 195
- Conclusions - 214

Outline

Before Lunch

- Introduction
  - MPI-1 Status, MPI-2 Status
  - C++ and Fortran90
- Life, 1D Decomposition
  - point-to-point
  - checkpoint/restart
    - *stdout*
    - *MPI-IO*
    - *PnetCDF*
  - RMA
    - *fence*
    - *post/start/complete/wait*

After Lunch

- Life, 2D Decomposition
  - point-to-point
  - RMA
- Sparse Matrix I/O
  - CSR format
  - checkpoint/restart
    - *stdout*
    - *MPI-IO*
- pNeo application
- Passive Target RMA
- Tuning
  - I/O tuning
  - RMA tuning
- Conclusions

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

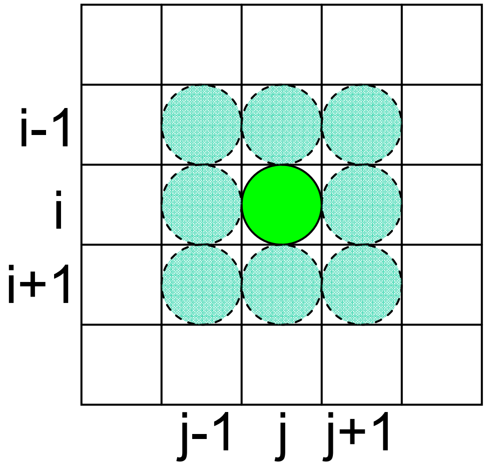
- Example driven
  - Structured data (Life)
  - Unstructured data (Sparse Matrix)
  - Unpredictable communication (pNeo)
  - Passive target RMA (global arrays and MPI mutex)
- Show solutions that use the MPI-2 support for parallel I/O and RMA
  - Walk through actual code
- We assume familiarity with MPI-1

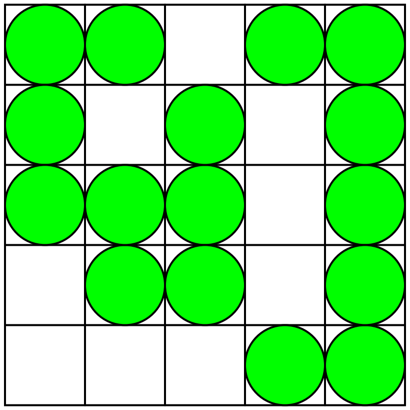
Conway’s Game of Life

- 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

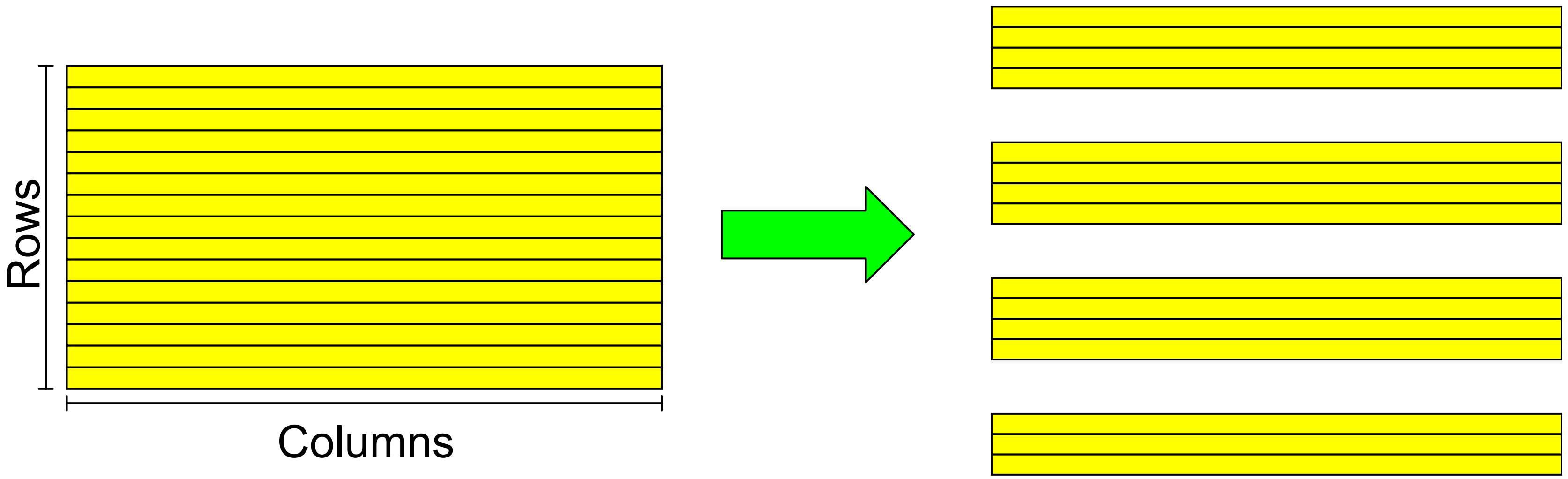
- For the non-parallel version, we:
  - Allocate a 2D matrix to hold state
    - *Actually two matrices, and we will swap them between steps*
  - Initialize the matrix
    - *Force boundaries to be “dead”*
    - *Randomly generate states inside*
  - At each time step:
    - *Calculate each new cell state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

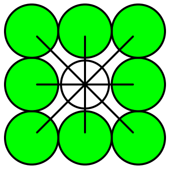
Step 1: Description of Decomposition

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

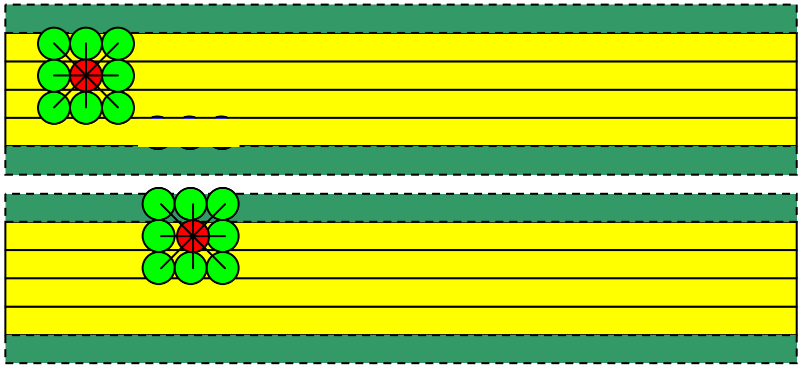


Step 2: Communication

- “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

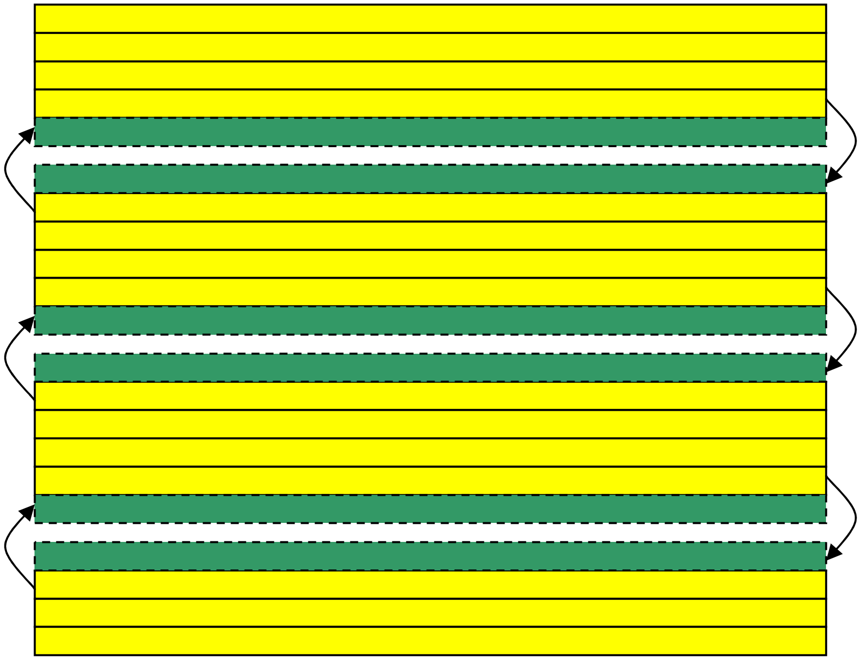


Step 3: Define the Local Arrays

- 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!)

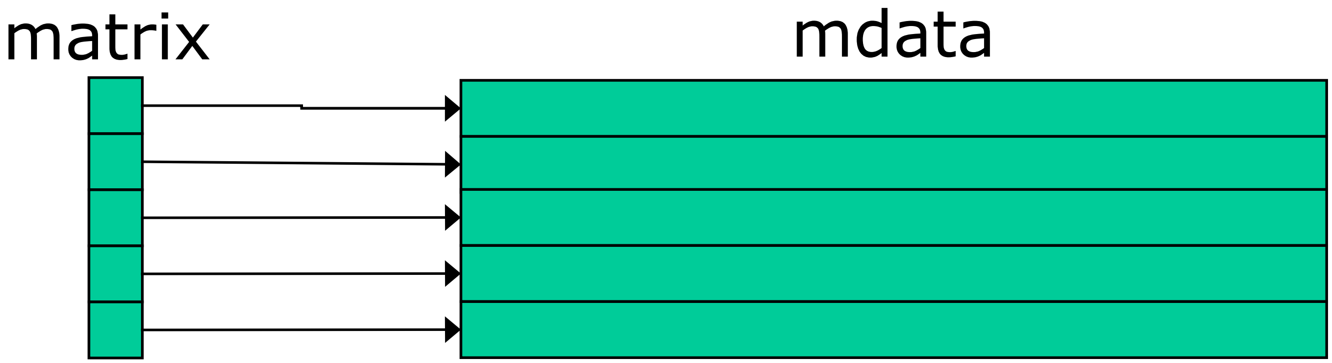
Boundary Regions

- 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*



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 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

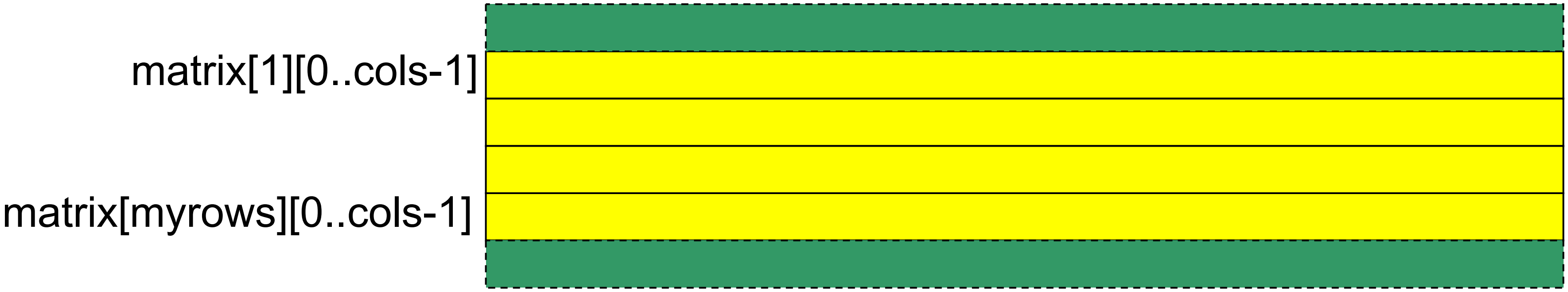
See mlife-io-stdout.c pp. 1-2 for code example.

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



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

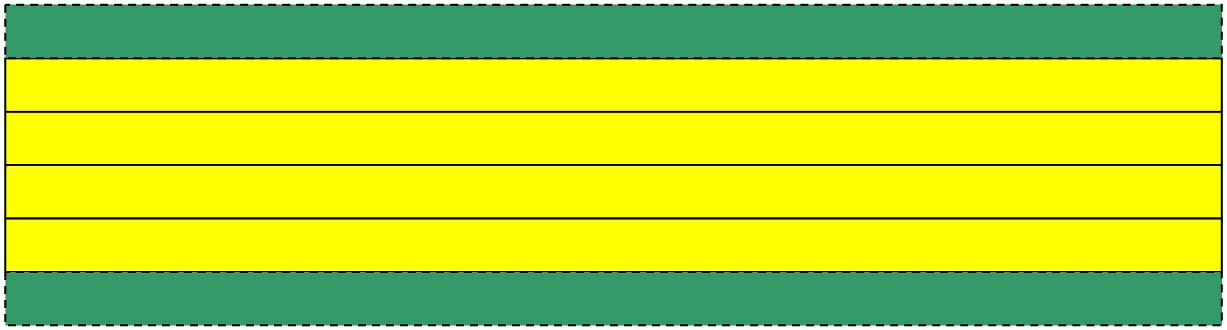
```
MPI_Type_vector(count = myrows,
  blklen = cols, stride = cols+2, MPI_INT, &vectype);
```

  - 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

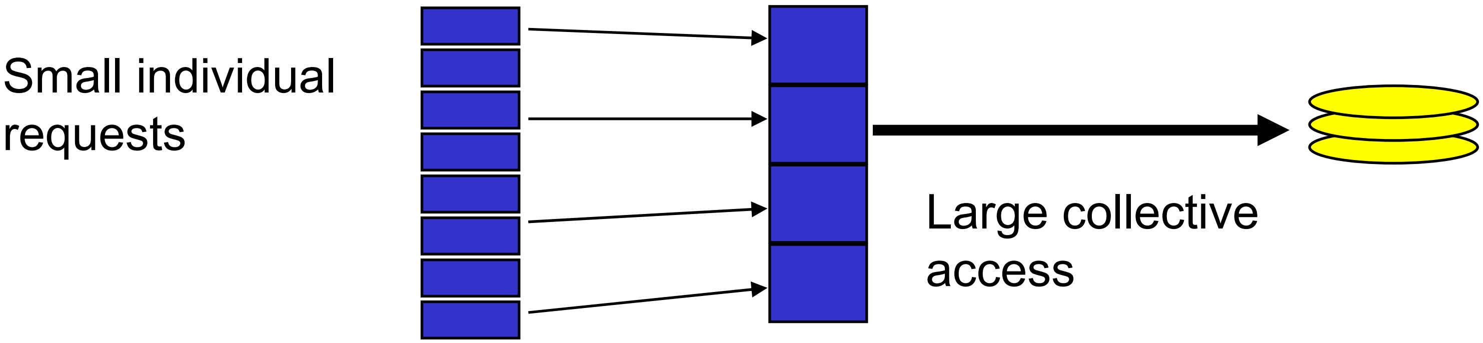
- Writing is like sending and reading is like receiving.
- Any parallel I/O system will need:
  - collective operations
  - user-defined datatypes to describe both memory and file layout
  - communicators to separate application-level message passing from I/O-related message passing
  - non-blocking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

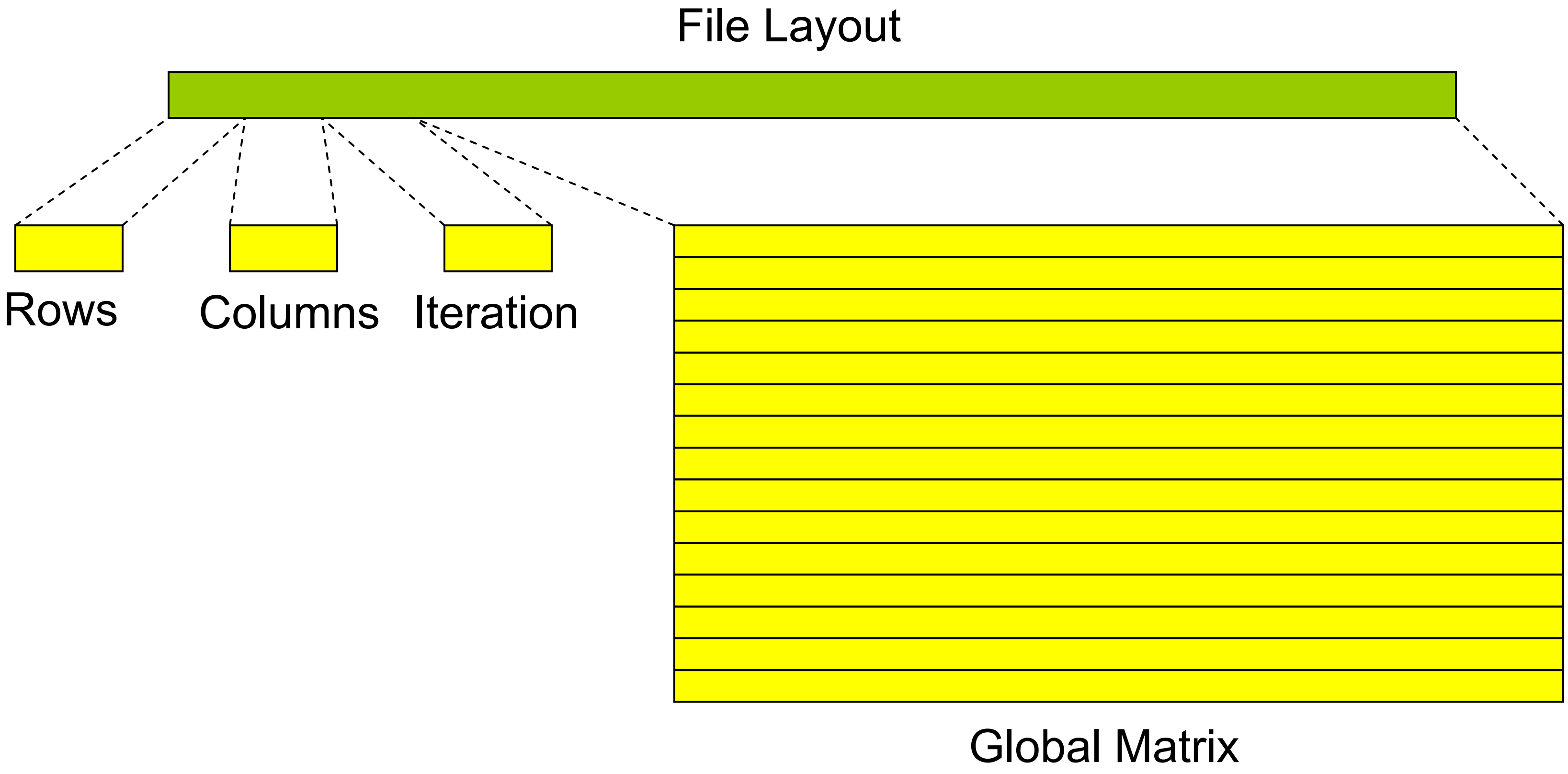
See mlife-io-mpiio.c pp. 3-6 for code example.

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`

like Unix I/O

- `MPI_File_read_at_all`
- `MPI_File_write_at_all`

combine seek and I/O for thread safety

- `MPI_File_read_ordered`
- `MPI_File_write_ordered`

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

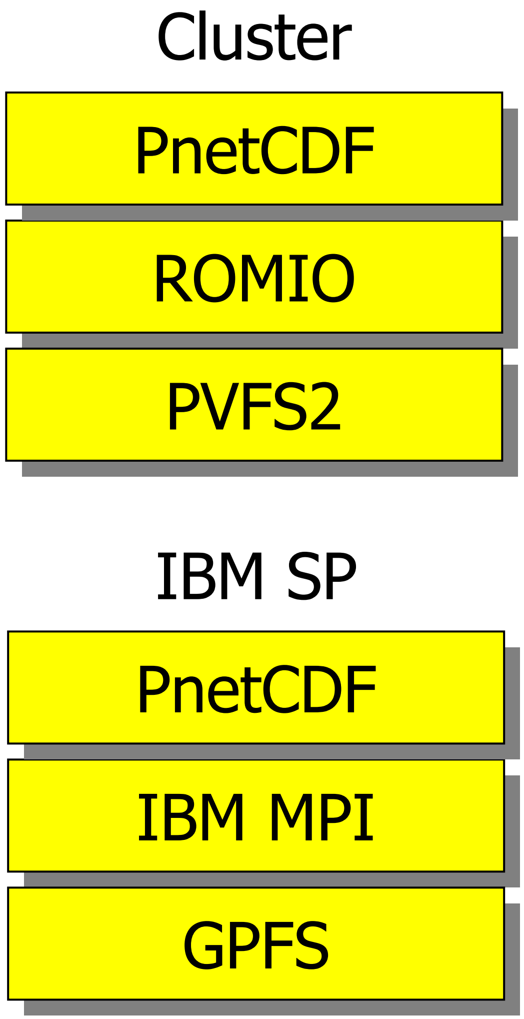
- Users can optionally create files with a portable binary data representation
- "datarep" parameter to **`MPI_File_set_view`**
- **`native`** - default, same as in memory, not portable
- `external32` - a specific representation defined in MPI, (basically 32-bit big-endian IEEE format), portable across 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



Cluster

PnetCDF

ROMIO

PVFS2

IBM SP

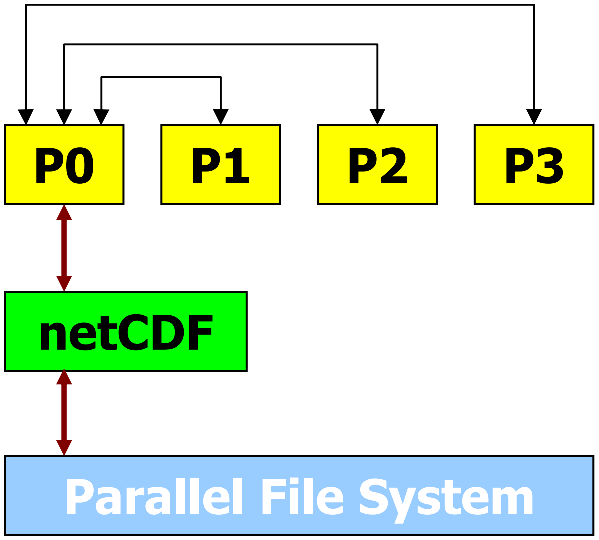
PnetCDF

IBM MPI

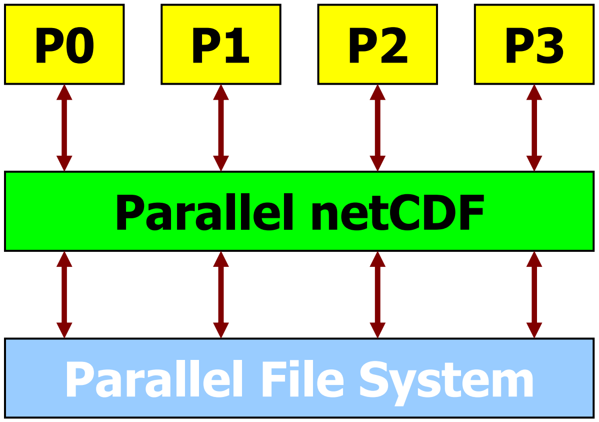
GPFS

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

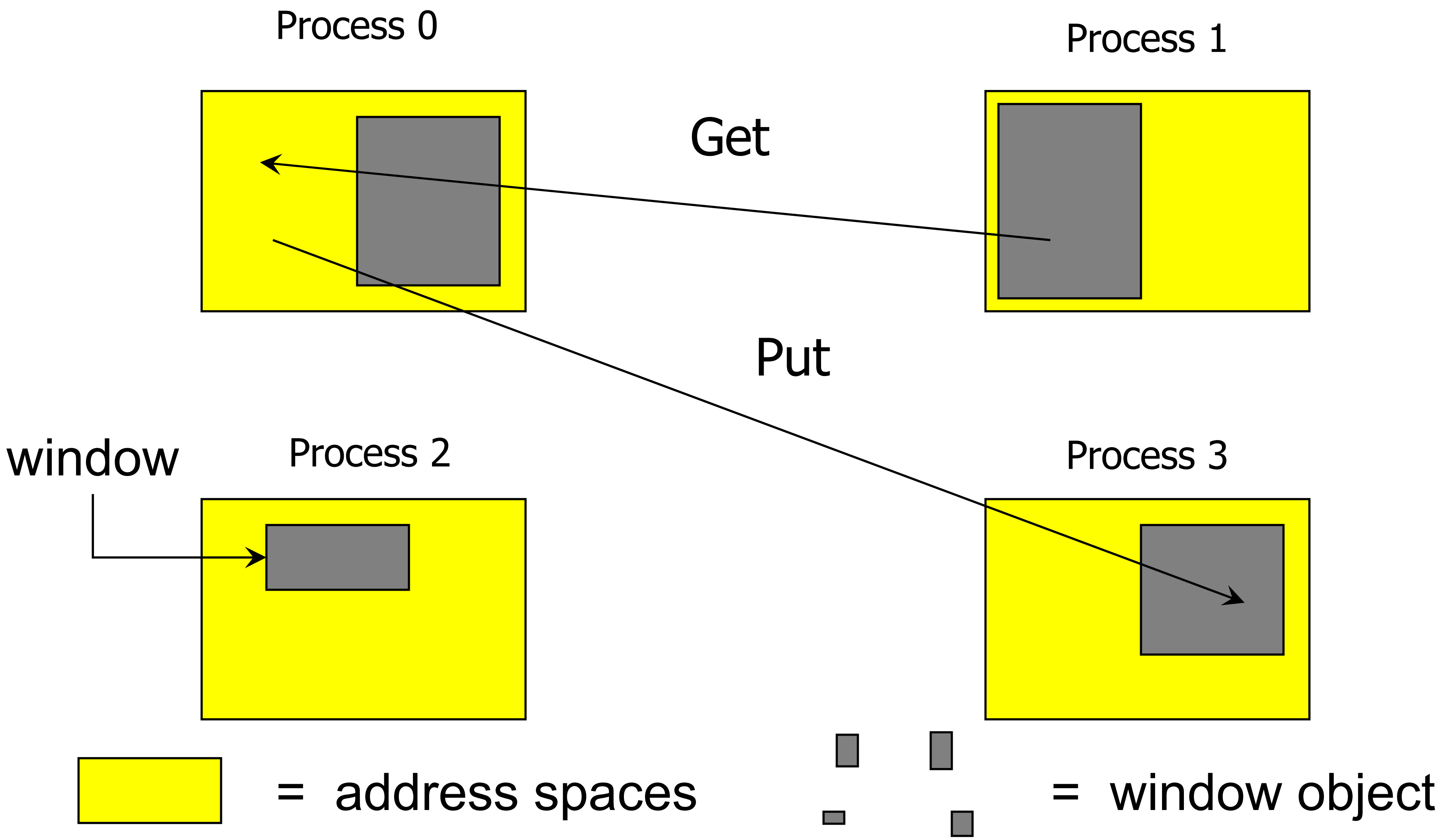
or

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



Process 0

Process 1

Get

Put

window

Process 2

Process 3

= address spaces

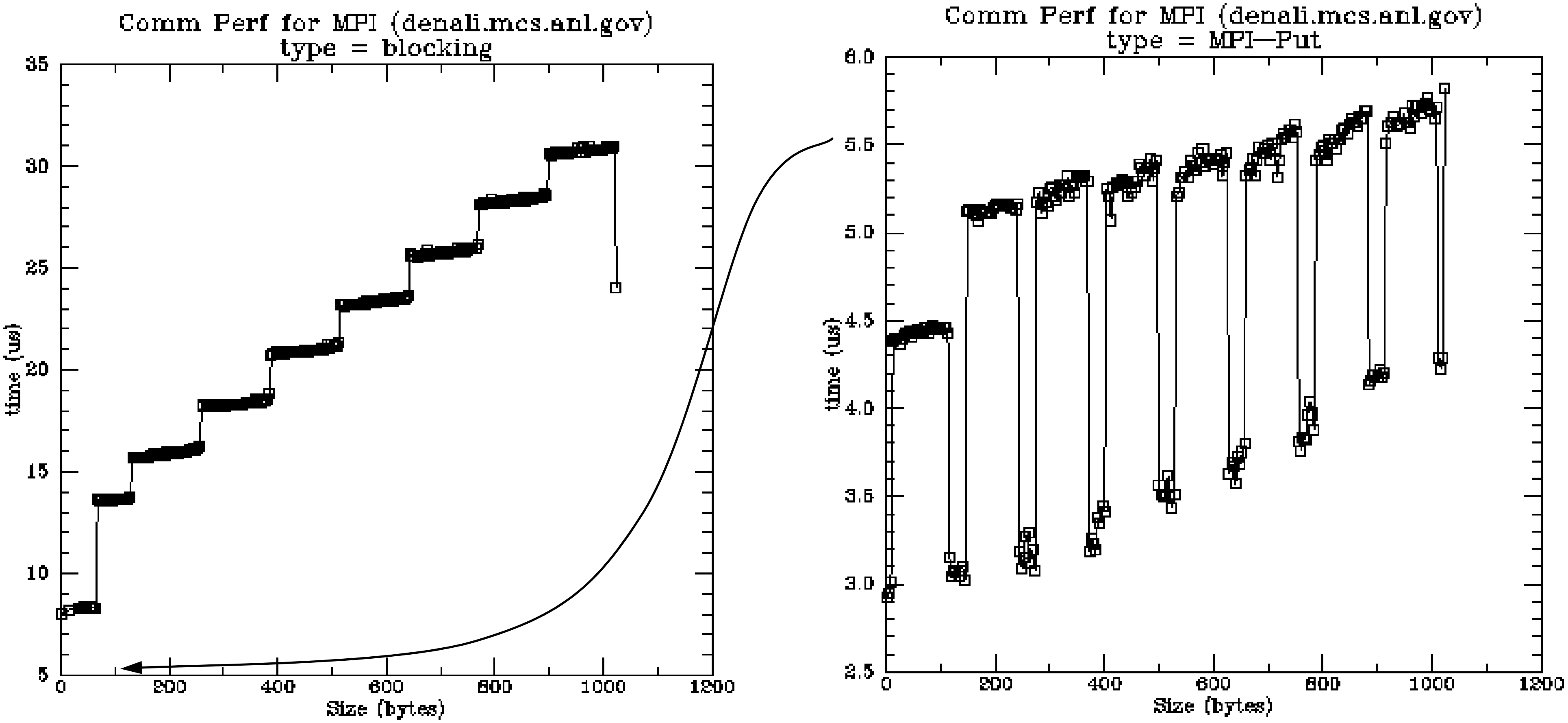
= window object

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_Get`** 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

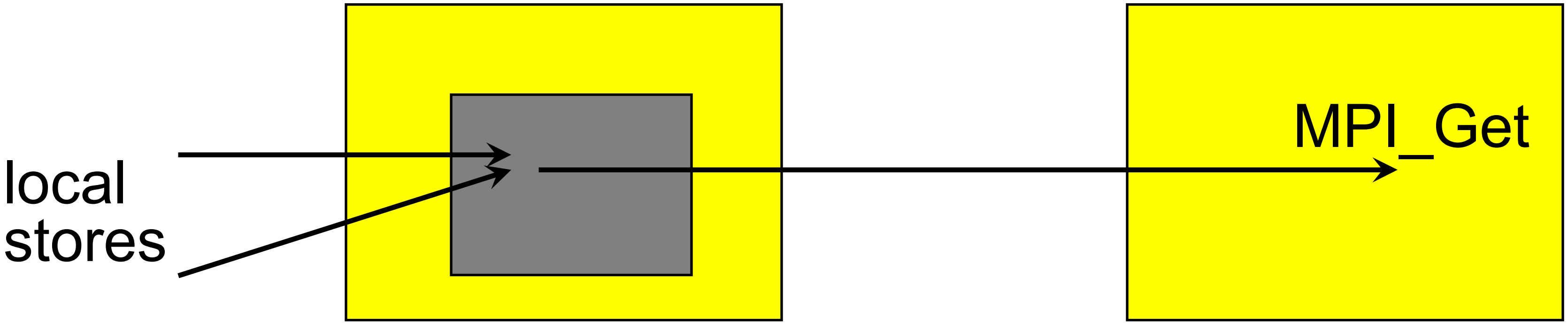
RMA Communication Calls

- **`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

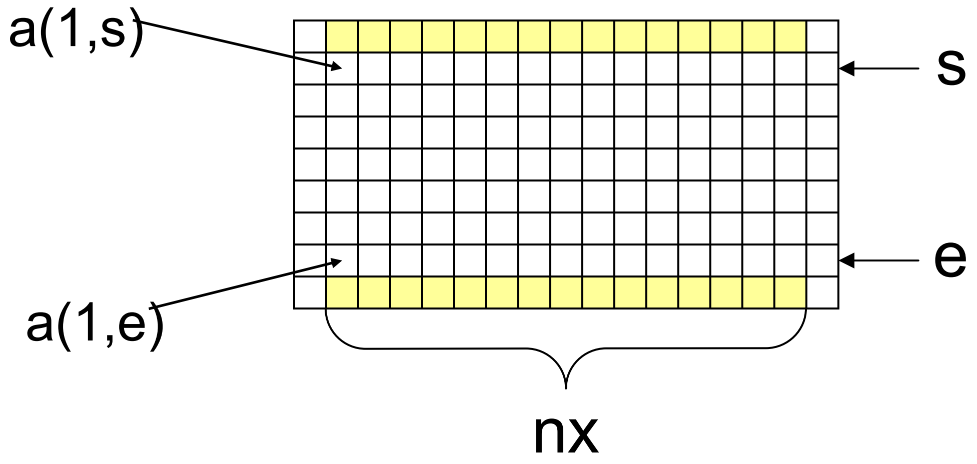
- Define the windows
  - Why – safety, options for performance (later)
- Define the data to move
- Mark the points where RMA can start and where it must complete (e.g., fence/put/put/fence)

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

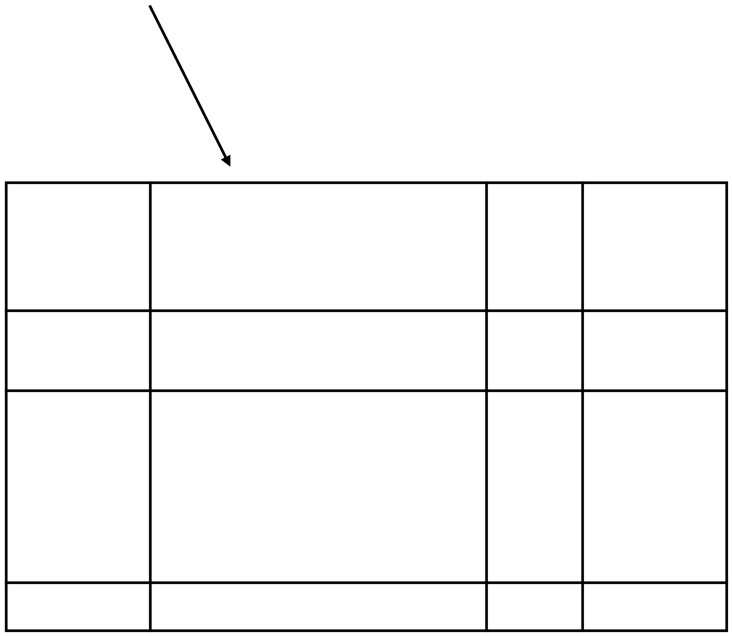
- Go back to global mesh view
- Define decomposition
- Define data to move
- Define local mesh

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

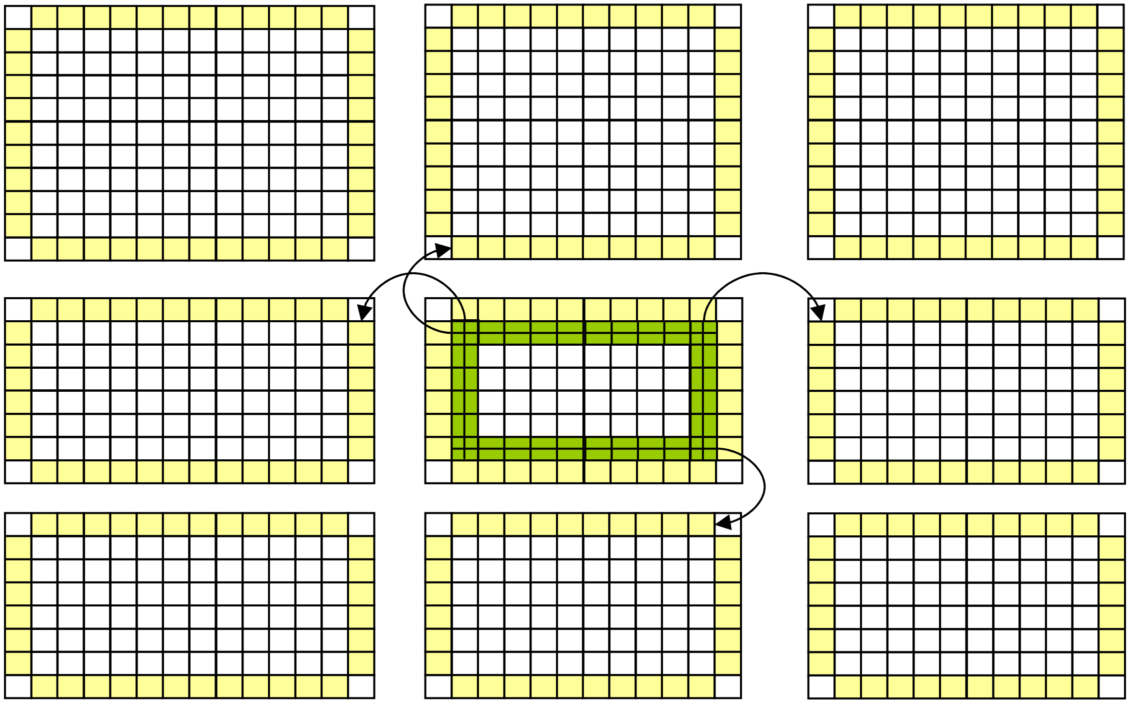
Creating the Window (Fortran)

```
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



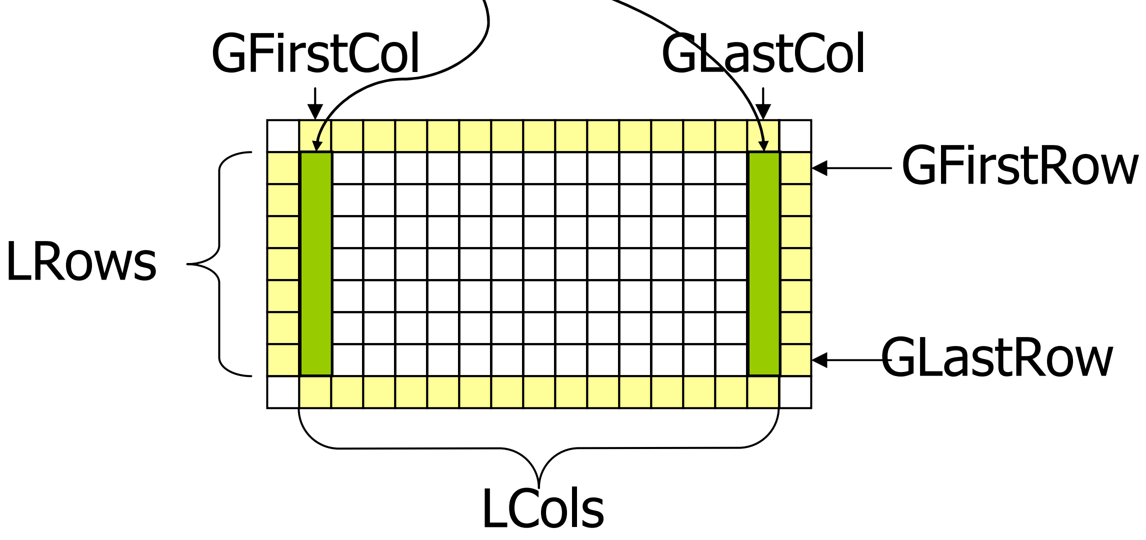
Creating Datatypes for Columns

elements

Stride

```
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



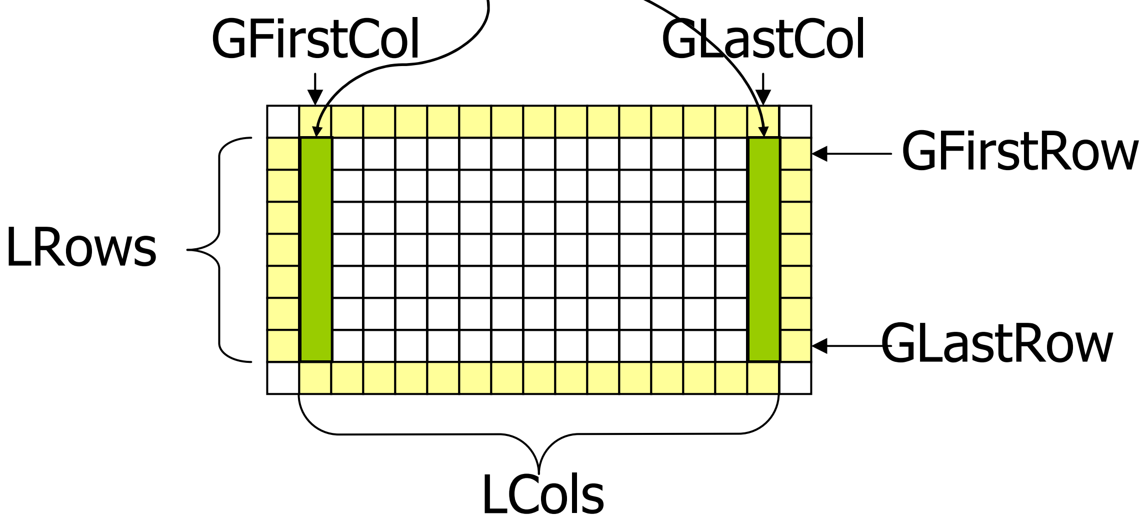
Creating Datatypes for Columns (C++)

elements

Stride

```
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



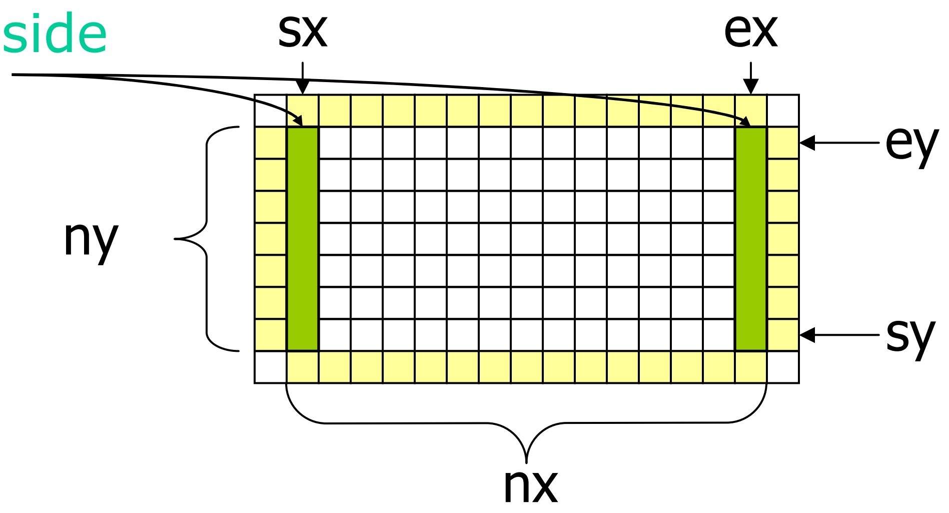
Creating Datatypes for Columns (Fortran)

# elements

Stride

```
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.

LUNCH

I/O for General Distributed Data

Handling Irregular Data Structures

- One strength of MPI is that you can handle any kind of situation (because you have to do much of the work yourself)
- Example: sparse matrix operations, such as used in PDE codes

Sparse Matrix I/O

- We have seen how to use MPI-I/O with regular data structures. What about irregular data structures?
  - Each process has a different amount of data
- For a simple example, we look at I/O for sparse matrices
  - Similar code can be used for unstructured meshes
- First more on I/O, then the example

Sparse Matrix I/O Characteristics

- Local to global data mapping not known by each process
  - Depends on number of nonzeros on previous ranks!
- Will need to communicate to determine relative positions before performing I/O
- Will use <u>independent I/O</u> in some cases
- Will read <u>noncontiguous</u> regions from file

Independent I/O with MPI-IO

Writing to a File

- Use **`MPI_File_write`** or **`MPI_File_write_at`**
- Use **`MPI_MODE_WRONLY`** or **`MPI_MODE_RDWR`** as the flags to **`MPI_File_open`**
- If the file doesn't exist previously, the flag **`MPI_MODE_CREATE`** must also be passed to **`MPI_File_open`**
- We can pass multiple flags by using bitwise-or '|' in C, or addition '+" in Fortran

Ways to Access a Shared File

• **`MPI_File_seek`**	like Unix I/O
• **`MPI_File_read`**	
• **`MPI_File_write`**	
• **`MPI_File_read_at`**	combine seek and I/O for thread safety
• **`MPI_File_write_at`**	
• **`MPI_File_read_shared`**	use shared file pointer
• **`MPI_File_write_shared`**	

Using Explicit Offsets

```
#include “mpi.h”
MPI_Status status;
MPI_File fh;
MPI_Offset offset;

MPI_File_open(MPI_COMM_WORLD, “/pfs/datafile”,
            MPI_MODE_RDONLY, MPI_INFO_NULL, &fh)
nints = FILESIZE / (nprocs*INTSIZE);
offset = rank * nints * INTSIZE;
MPI_File_read_at(fh, offset, buf, nints, MPI_INT,
                 &status);
MPI_Get_count(&status, MPI_INT, &count );
printf( “process %d read %d ints\n”, rank, count );

MPI_File_close(&fh);
```

Using Explicit Offsets (C++)

```
#include “mpi.h”
MPI::Status status;
MPI::Offset offset;

fh = MPI::FILE::Open(MPI::COMM_WORLD,“/pfs/datafile”,
                     MPI::MODE_RDONLY, MPI::INFO_NULL);
nints = FILESIZE / (nprocs*sizeof(int));
offset = rank * nints * sizeof(int);
fh.Read_at(offset, buf, nints, MPI::INT, status);
count = status.Get_count( MPI::INT );
cout << “process “ << rank << “read “ << count <<
     “ints” << “\n”;
fh.Close();
```

Using Explicit Offsets (Fortran)

```
  include 'mpif.h'

  integer status(MPI_STATUS_SIZE)
  integer (kind=MPI_OFFSET_KIND) offset
C in F77, see implementation notes (might be integer*8)

  call MPI_FILE_OPEN(MPI_COMM_WORLD, '/pfs/datafile', &
            MPI_MODE_RDONLY, MPI_INFO_NULL, fh, ierr)
  nints = FILESIZE / (nprocs*INTSIZE)
  offset = rank * nints * INTSIZE
  call MPI_FILE_READ_AT(fh, offset, buf, nints,
                        MPI_INTEGER, status, ierr)
  call MPI_GET_COUNT(status, MPI_INTEGER, count, ierr)
  print *, 'process ', rank, 'read ', count, 'integers'

  call MPI_FILE_CLOSE(fh, ierr)
```

Why Use Independent I/O?

- Sometimes the synchronization of collective calls is not natural
- Sometimes the overhead of collective calls outweighs their benefits
  - Example: very small I/O during header reads

Sparse Matrix Operations

- A typical operation is a matrix-vector multiply
- Consider an example where the sparse matrix is produced by one application and you wish to use a parallel program to solve the linear system

Sparse Matrix Format

n – number of rows/cols (matrix dimensions)
nz – number of nonzero elements
ia[0..n] – index into data for first element in row i
ja[0..nz-1] – column location for element j
a[0..nz-1] – actual data

(0, 0, 0, 0, 4
1, 0, 3, 0, 0
5, 2, 0, 0, 8
0, 6, 7, 0, 0
0, 0, 0, 9, 0)

n = 5
nz = 9
ia[] = (0, 1, 3, 6, 8, 9)
ja[] = (4, 0, 2, 0, 1, 4, 1, 2, 3)
a[] = (4, 1, 3, 5, 2, 8, 6, 7, 9)

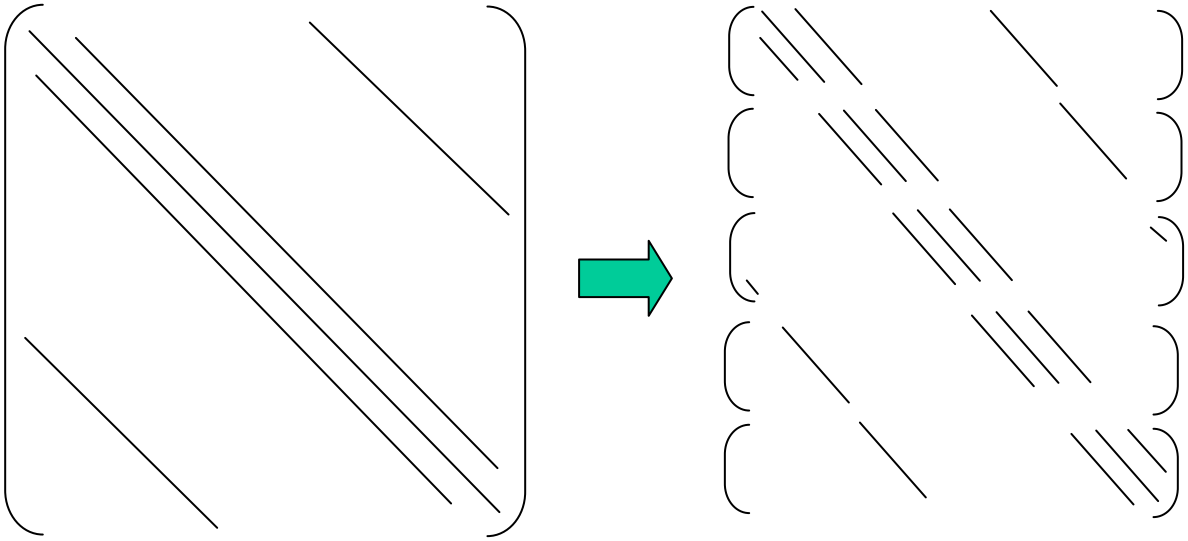
(known as CSR or AIJ format)
Note: Format isn't a win for a matrix of this size and density.

Steps in Designing the Parallel Version

- Same as our other examples:
  - Decomposition
  - Communication (for the matrix-vector product)
  - Define the local representation

Step 1: Description of Decomposition

- By rows (matches equations)
- In practice, the diagonal block and off-diagonal block are stored separately
  - For simplicity, we will ignore this



Step 2: Communication

- For matrix-vector product, we would need
  - Elements of vector (also distributed in the same way as the matrix) from other processes corresponding to columns in which there are non-zero entries
- Can be implemented with send and receive or with RMA
  - For simplicity, we will not show this part of the code

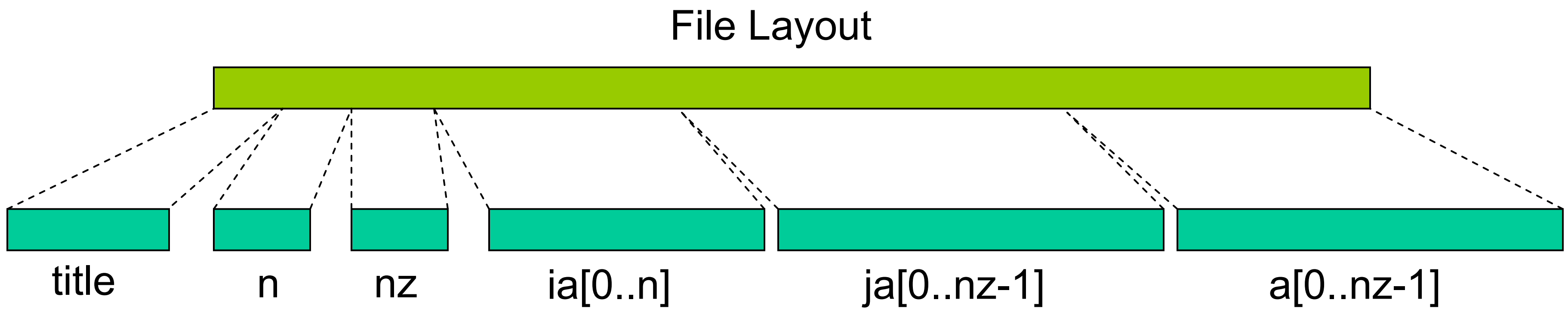
Step 3: Define the Local Arrays

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

I/O in Sparse Matrix Codes

- Define the file format
- We want the file to be independent of the number of processes
- File requires:
  - Header information
    - *Size of matrix, number of non-zeros*
    - *Name of matrix*
  - ia, ja, and A vectors

Placing Data in Checkpoint



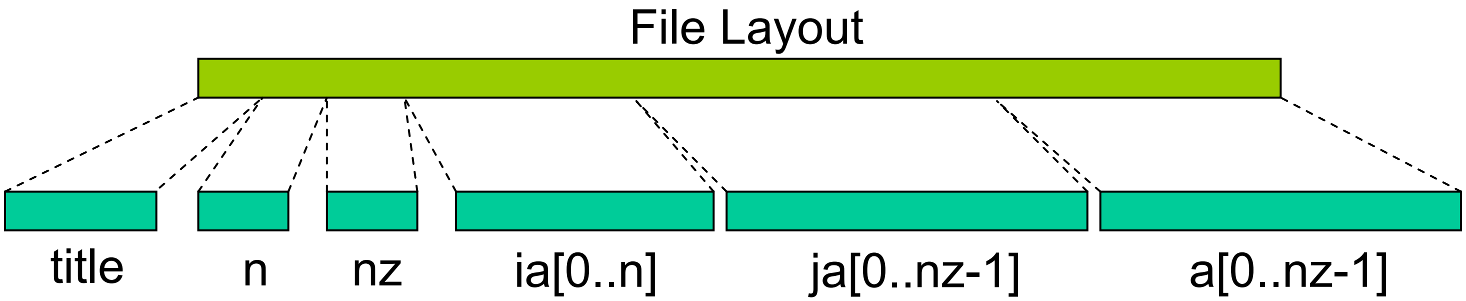
- Unlike data layout in the Life case, positioning of data for a given process depends on the values held by other processes (number of nonzero values)!
- Each process has pieces that are spread out in the file (noncontiguous!)

stdio CSRIO Code Walkthrough

- Points to observe
  - MPI_Exscan and MPI_Allreduce to discover starting locations and complete sizes of vectors
  - Passing data to rank 0 for printing
  - Converting ia from local to global references

See csrio-stdout.c pp. 1-2 for code example.

Writing Sparse Matrices (stdout)



- Steps:
  - MPI_Exscan to get count of nonzeros from all previous processes
    - *gives starting offset in ja[] and a[] arrays and value to add to ia[] elements*
  - MPI_Allreduce to get total count of nonzeros (nz) – gives size of ja[] and a[] arrays
  - Process zero writes header (title, n, nz)
  - Copy ia[] and adjust to refer to global matrix locations
  - Pass data back to rank zero for printing

See csrio-stdout.c pp. 3-8 for code example.

Noncontiguous I/O in File

- Each process describes the part of the file that it is responsible for
  - This is the "file view"
  - Described in MPI with an offset (useful for headers) and an MPI_Datatype
- Only the part of the file described by the file view is visible to the process; reads and writes access these locations
- This provides an efficient way to perform *noncontiguous accesses*

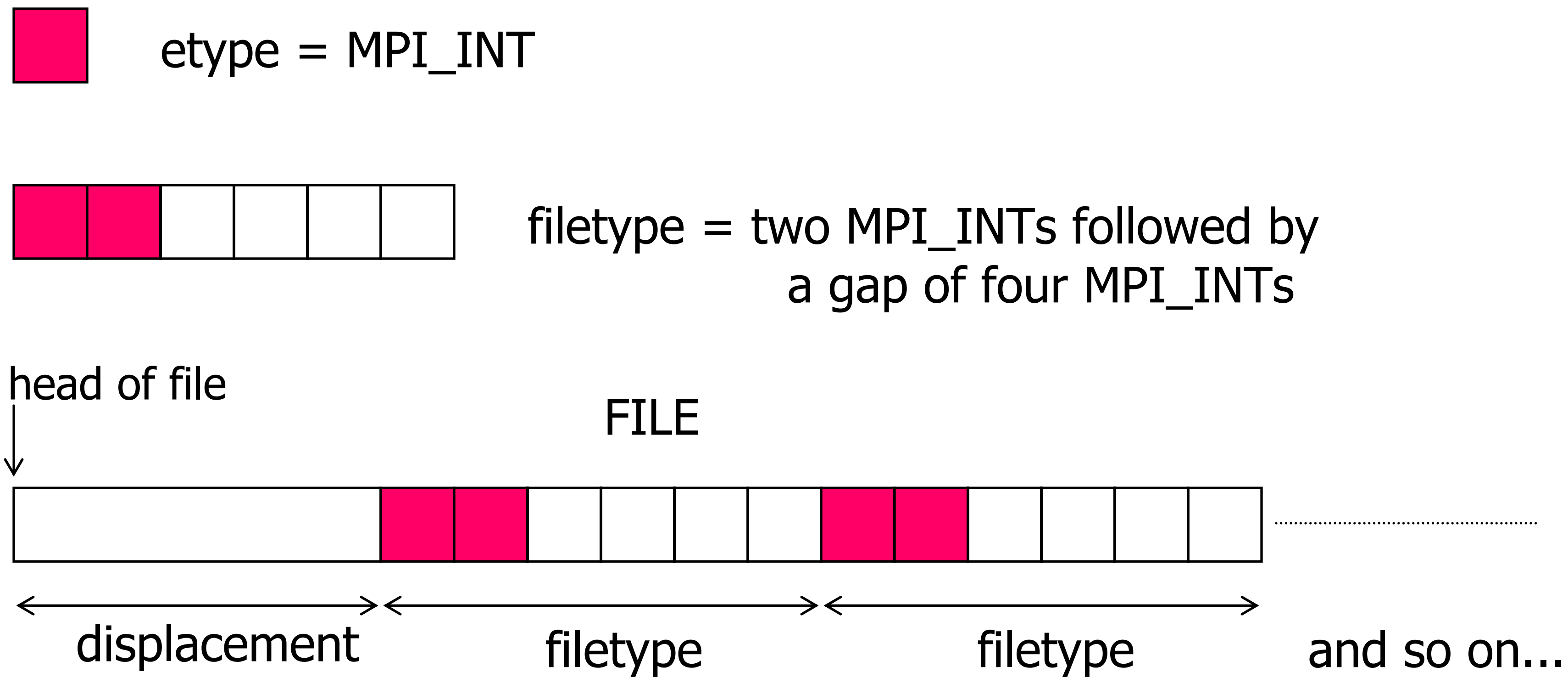
Noncontiguous Accesses

- Common in parallel applications
- Example: distributed arrays stored in files
- A big advantage of MPI I/O over Unix I/O is the ability to specify noncontiguous accesses in memory and file within a single function call by using derived datatypes
- Allows implementation to optimize the access
- Collective I/O combined with noncontiguous accesses yields the highest performance

File Views

- Specified by a triplet (*displacement*, *etype*, and *filetype*) passed to **`MPI_File_set_view`**
- *displacement* = number of bytes to be skipped from the start of the file
  - E.g., to skip a file header
- *etype* = basic unit of data access (can be any basic or derived datatype)
- *filetype* = specifies which portion of the file is visible to the process

A Simple Noncontiguous File View Example



etype = MPI_INT

filetype = two MPI_INTs followed by
a gap of four MPI_INTs

head of file

FILE

displacement

filetype

filetype

and so on...

Noncontiguous File View Code

```
MPI_Aint lb, extent;
MPI_Datatype etype, filetype, contig;
MPI_Offset disp;

MPI_Type_contiguous(2, MPI_INT, &contig);
lb = 0; extent = 6 * sizeof(int);
MPI_Type_create_resized(contig, lb, extent, &filetype);
MPI_Type_commit(&filetype);
disp = 5 * sizeof(int); etype = MPI_INT;

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
     MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, etype, filetype, "native",
                  MPI_INFO_NULL);
MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
```

Noncontiguous File View Code (C++)

```
MPI::Aint lb, extent;
MPI::Datatype etype, filetype, contig;
MPI::Offset disp;

contig = MPI::Type::Contiguous(2, MPI::INT);
lb = 0; extent = 6 * sizeof(int);
filetype = MPI::Type::Create_resized(contig, lb, extent);
filetype.Commit();
disp = 5 * sizeof(int); etype = MPI::INT;

fh = MPI::File::Open(MPI::COMM_WORLD, "/pfs/datafile",
     MPI::MODE_CREATE | MPI::MODE_RDWR, MPI::INFO_NULL );
fh.Set_view( disp, etype, filetype, "native",
                  MPI::INFO_NULL);
fh.Write(buf, 1000, MPI::INT);
```

Noncontiguous File View Code (Fortran)

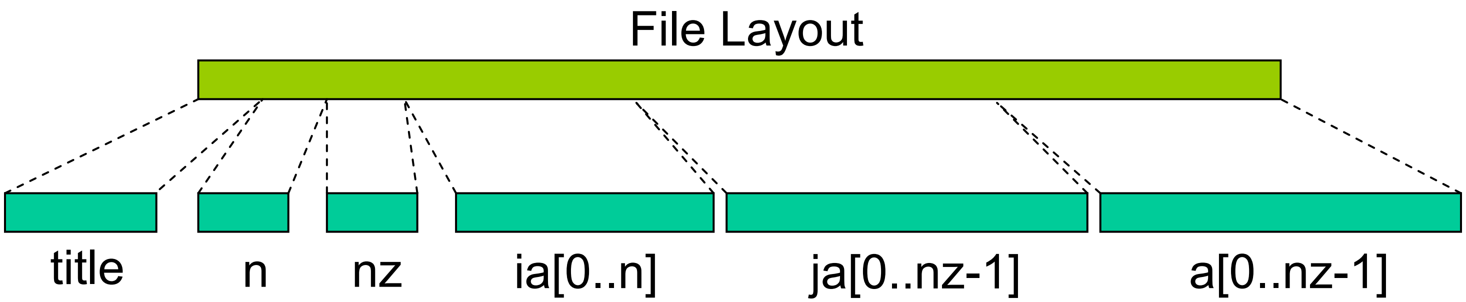
```
integer (kind=MPI_ADDRESS_KIND) lb, extent;
integer etype, filetype, contig;
integer (kind=MPI_OFFSET_KIND) disp;

call MPI_Type_contiguous(2, MPI_INTEGER, contig, ierr)
call MPI_Type_size( MPI_INTEGER, sizeofint, ierr )
lb = 0
extent = 6 * sizeofint
call MPI_Type_create_resized(contig, lb, extent, filetype, ierr)
call MPI_Type_commit(filetype, ierr);
disp = 5 * sizeof(int); etype = MPI_INTEGER

call MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",  &
     MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, fh, ierr)
call MPI_File_set_view(fh, disp, etype, filetype, "native", &
                  MPI_INFO_NULL, ierr)
call MPI_File_write(fh, buf, 1000, MPI_INTEGER, MPI_STATUS_IGNORE)
```

MPI-IO CSRIO Code Walkthrough

- Points to observe
  - Independent I/O when reading or writing the header
  - Use of file views when reading or writing data

See csrio-mpiio.c pp. 1-2 for code example.

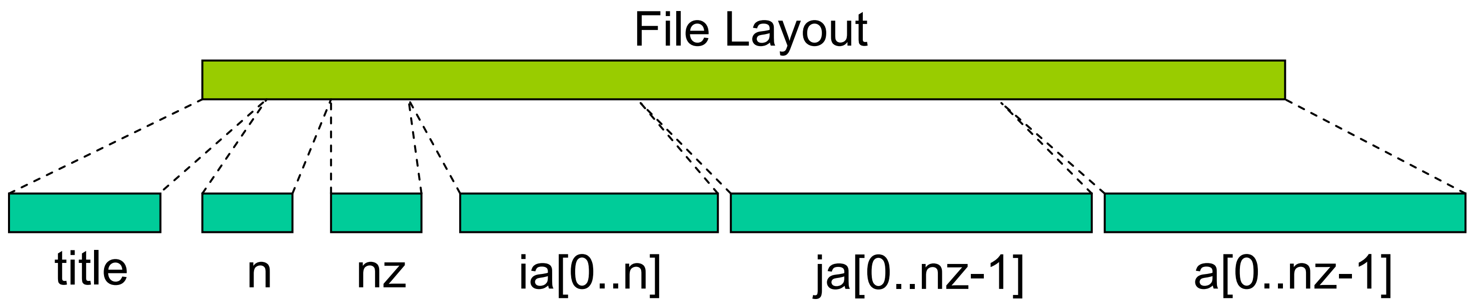
Reading Sparse Matrix Header



- Steps:
  - Process 0 reads the title, n, and nz independently (i.e., using independent I/O)
    - *Collective open times can be very large*
  - MPI_Bcast values to everyone
    - *MPI_Type_struct used to combine data into a single broadcast*

See csrio-mpiio.c pp. 3-5 for code example.

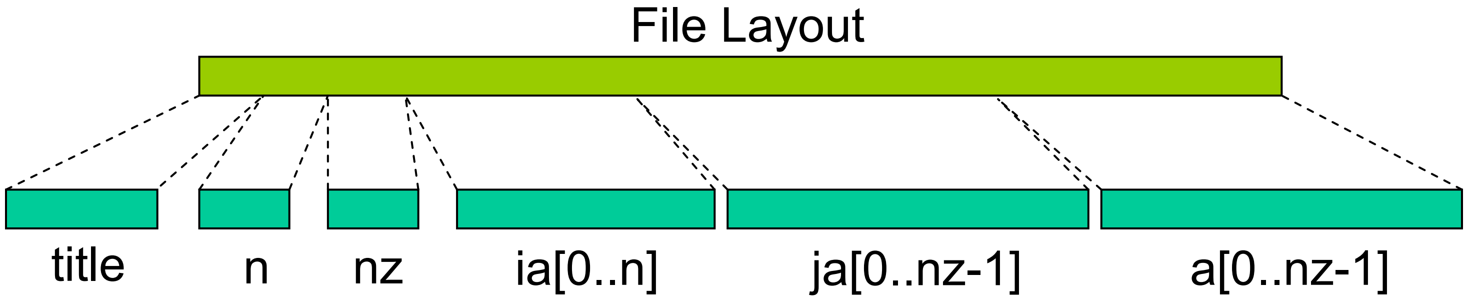
Reading Sparse Matrix Data



- Steps:
  - Everyone reads the portion of ia[] for their rows
  - MPI_Allreduce to verify that everyone successfully allocated memory
    - *Avoids potential deadlocks if one process were to return an error*
  - Collectively read data
  - Convert ia[] to refer to local matrix

See csrio-mpiio.c pp. 6-9 for code example.

Writing Sparse Matrices



- Steps:
  - MPI_Scan to get count of nonzeros from all previous processes
    - *gives starting offset in ja[] and a[] arrays and value to add to ia[] elements*
  - MPI_Allreduce to get total count of nonzeros (nz) – gives size of ja[] and a[] arrays
  - Process zero writes header (title, n, nz)
  - Copy ia[] and adjust to refer to global matrix locations
  - All processes write ia, ja, a collectively

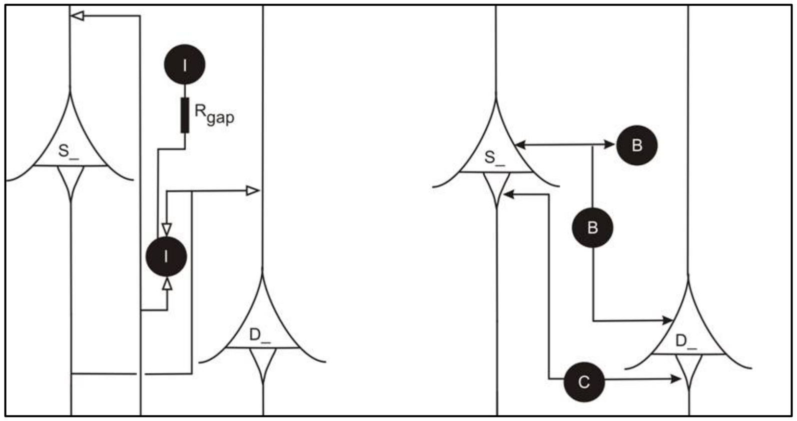
See csrio-mpiio.c pp. 10-13 for code example.

pNeo - Modeling the Human Brain

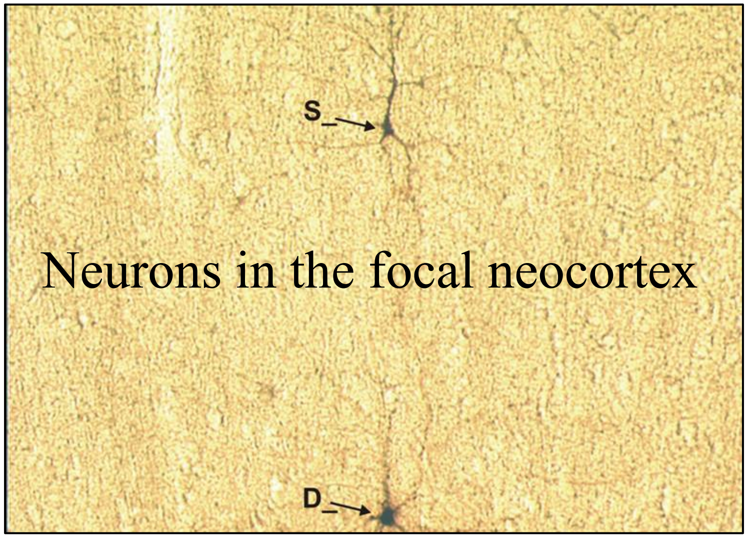
Science Driver

- Goal: Understand conditions, causes, and possible corrections for epilepsy
- Approach: Study the onset and progression of epileptiform activity in the neocortex
- Technique: Create a model of neurons and their interconnection network, based on models combining wet lab measurements of resected tissue samples and *in vivo* studies
- Computation: Develop a simulation program that can be used for detailed parameter studies

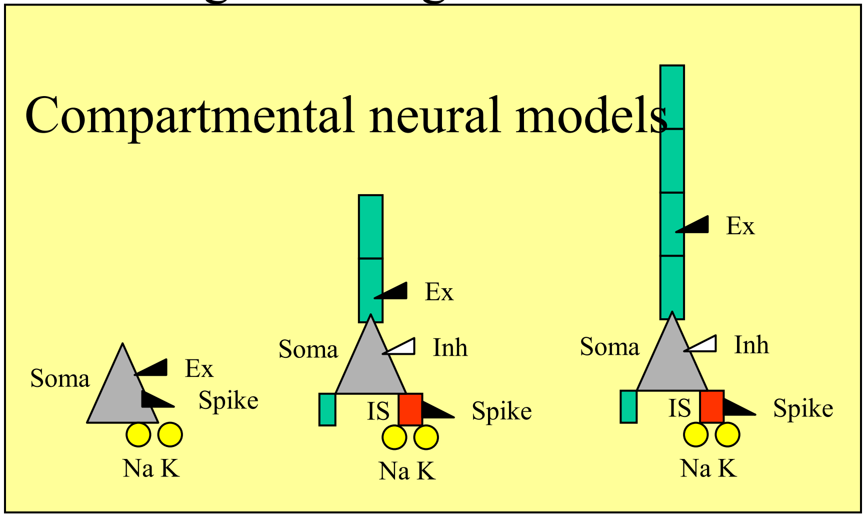
Model Neurons



Excitatory and inhibitory signal wiring between neurons

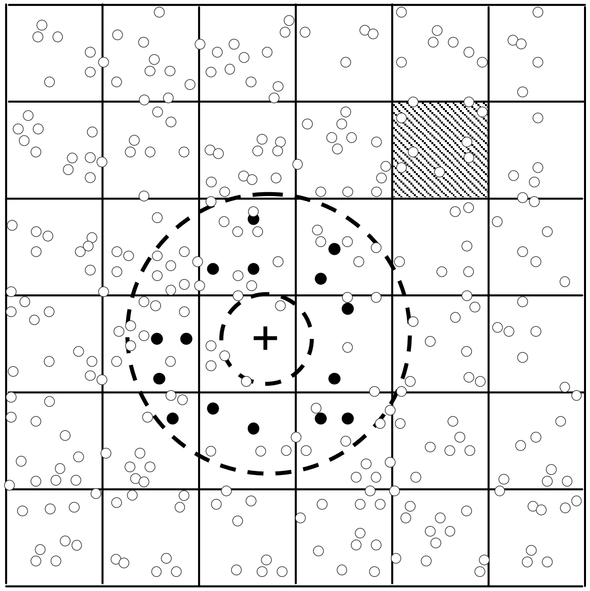


Neurons in the focal neocortex


Compartmental neural models

Modeling Approach

- Individual neurons are modeled using electrical analogs to parameters measured in the laboratory
- Differential equations describe evolution of the neuron state variables
- Neuron spiking output is wired to thousands of cells in a neighborhood
- Wiring diagram is based on wiring patterns observed in neocortex tissue samples
- Computation is divided among available processors



Schematic of a two dimensional patch of neurons showing communication neighborhood for one of the cells in the simulation and partitioning of the patch among processors.

Abstract pNeo for Tutorial Example

- “Simulate the simulation” of the evolution of neuron state instead of solving the differential equations
- Focus on how to code the interactions between cells in MPI
- Assume one cell per process for simplicity
  - Real code multiplexes many individual neurons onto one MPI process

What Happens In Real Life

- Each cell has a fixed number of connections to some other cells
- Cell “state” evolves continuously
- From time to time “spikes” arrive from connected cells.
- Spikes influence the evolution of cell state
- From time to time the cell state causes spikes to be sent to other connected cells

What Happens In Existing pNeo Code

- In pNeo, each cell is connected to about 1000 cells
  - Large runs have 73,000 cells
  - Brain has ~100 billion cells
- Connections are derived from neuro-anatomical data
- There is a global clock marking time steps
- The state evolves according to a set of differential equations
- About 10 or more time steps between spikes
  - I.e., communication is unpredictable and sparse
- Possible MPI-1 solutions
  - Redundant communication of communication pattern before communication itself, to tell each process how many receives to do
  - Redundant "no spikes this time step" messages
- MPI-2 solution: straightforward use of Put, Fence

What Happens in Tutorial Example

- There is a global clock marking time steps
- At the beginning of a time step, a cell notes spikes from connected cells (put by them in a previous time step).
- A dummy evolution algorithm is used in place of the differential equation solver.
- This evolution computes which new spikes are to be sent to connected cells.
- Those spikes are sent (put), and the time step ends.
- We show both a Fence and a Post/Start/Complete/Wait version.

Two Examples Using RMA

- Global synchronization
  - Global synchronization of all processes at each step
  - Illustrates Put, Get, Fence
- Local synchronization
  - Synchronization across connected cells, for improved scalability (synchronization is local)
  - Illustrates Start, Complete, Post, Wait

pNeo Code Walkthrough

- Points to observe
  - Data structures can be the same for multiple synchronization approaches
- Code is simple compared to what a send/receive version would look like
  - Processes do no need to know which other processes will send them spikes at each step

See pneo_fence.c and pneo_pscw.c for code examples.

Passive Target RMA

Active vs. Passive Target RMA

- *Active target* RMA requires participation from the target process in the form of synchronization calls (fence or P/S/C/W)
- In *passive target* RMA, target process makes no synchronization call

Passive Target RMA

- We need to indicate the beginning and ending of RMA calls by the process performing the RMA
  - This process is called the *origin* process
  - The process being accessed is the *target* process
- For passive target, the begin/end calls are
  - MPI_Win_lock, MPI_Win_unlock

Synchronization for Passive Target RMA

- **`MPI_Win_lock(locktype, rank, assert, win)`**
  - Locktype is
    - *MPI_LOCK_EXCLUSIVE*
      - One process at a time may access
      - Use when modifying the window
    - *MPI_LOCK_SHARED*
      - Multiple processes (as long as none hold MPI_LOCK_EXCLUSIVE)
      - Consider using when using MPI_Get (only) on the window
  - Assert is either 0 or MPI_MODE_NOCHECK
- **`MPI_Win_unlock(rank, win)`**
- Lock is not a real lock but means begin-RMA; unlock is end-RMA, not real unlock

Put with Lock

```
if (rank == 0) {
   MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 1,
                   0, win);
   MPI_Put(outbuf, n, MPI_INT, 1,
           0, n, MPI_INT, win);
   MPI_Win_unlock(1, win);
 }
```

- Only process performing MPI_Put makes MPI RMA calls
  - Process with memory need not make any MPI calls; it is “passive”
- Similarly for MPI_Get, MPI_Accumulate

Put with Lock (C++)

```
if (rank == 0) {
   win.Lock( MPI::LOCK_EXCLUSIVE, 1,
                 0 );
   win.Put(outbuf, n, MPI::INT, 1,
           0, n, MPI::INT);
   win.Unlock(1);
 }
```

- Only process performing MPI_Put makes MPI RMA calls
  - Process with memory need not make any MPI calls; it is “passive”
- Similarly for MPI_Get, MPI_Accumulate

Put with Lock (Fortran)

```
if (rank .eq. 0) then
   call MPI_Win_lock(MPI_LOCK_EXCLUSIVE,&
                     1, 0, win, ierr )
   call MPI_Put(outbuf, n, MPI_INTEGER, &
        1, 0, n, MPI_INTEGER, win, ierr)
   call MPI_Win_unlock(1, win, ierr)
 endif
```

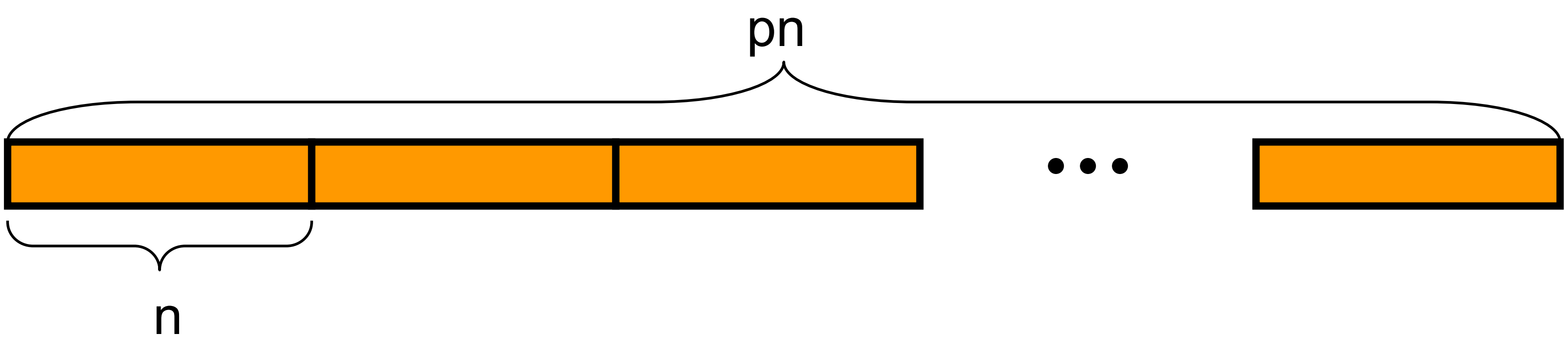
- Only process performing MPI_Put makes MPI RMA calls
  - Process with memory need not make any MPI calls; it is “passive”
- Similarly for MPI_Get, MPI_Accumulate

Global Arrays

- Lets look at updating a single array, distributed across a group of processes

A Global Distributed Array

- Problem: Application needs a single, 1-dimensional array that any process can update or read
- Solution: Create a window object describing local parts of the array, and use MPI_Put and MPI_Get to access



- Each process has alocal[n]
- We must provide access to a[pn]
- We cannot use MPI_Win_fence; we must use MPI_Win_lock and MPI_Win_unlock

Creating the Global Array

```
double *locala;
...
MPI_Alloc_mem(n * sizeof(double), MPI_INFO_NULL,
              &locala);
MPI_Win_create(locala, n * sizeof(double),
               sizeof(double),
               MPI_INFO_NULL, comm, &win);
```

Creating the Global Array (C++)

```
Volatile double *locala;
...
locala = MPI::Alloc_mem(n * sizeof(double),
                        MPI::INFO_NULL );
win = MPI::Win::Create(locala, n * sizeof(double),
                    sizeof(double),
MPI::INFO_NULL,
                    comm);
```

Comments

- MPI-2 allows “global” to be relative to a communicator, enabling hierarchical algorithms
  - i.e., “global” does not have to refer to MPI_COMM_WORLD
- MPI_Alloc_mem is required for greatest portability
  - Some MPI implementations may allow memory not allocated with MPI_Alloc_mem in passive target RMA operations

Accessing the Global Array From a Remote Process

- *To update:*

```
rank = i / n;
offset = i % n;
MPI_Win_lock(MPI_LOCK_EXCLUSIVE, rank, 0, win);
MPI_Put(&value, 1, MPI_DOUBLE,
        rank, offset, 1,MPI_DOUBLE, win);
MPI_Win_unlock(rank, win);
```

- *To read:*

```
rank = i / n;
offset = i % n;
MPI_Win_lock(MPI_LOCK_SHARED, rank, 0, win);
MPI_Get(&value, 1, MPI_DOUBLE,
        rank, offset, 1, MPI_DOUBLE, win);
MPI_Win_unlock(rank, win);
```

Accessing the Global Array From a Remote Process (C++)

- *To update:*

```
rank = i / n;
offset = i % n;
win.Lock(MPI_LOCK_EXCLUSIVE, rank, 0);
win.Put(&value, 1, MPI::DOUBLE,
        rank, offset, 1, MPI::DOUBLE);
win.Unlock(rank);
```

- *To read:*

```
rank = i / n;
offset = i % n;
win.Lock(MPI::LOCK_SHARED, rank, 0);
win.Get(&value, 1, MPI_DOUBLE,
        rank, offset, 1, MPI_DOUBLE);
win.Unlock(rank);
```

Accessing the Global Array From a Remote Process (Fortran)

- *To update:*

```
rank = i / n
offset = mod(i,n)
call MPI_Win_lock(MPI_LOCK_EXCLUSIVE, rank, 0, &
                  win, ierr)
call MPI_Put(value, 1, MPI_DOUBLE_PRECISION, &
        rank, offset, 1, MPI_DOUBLE_PRECISION, &
        win, ierr )
call MPI_Win_unlock(rank, win, ierr )
```

- *To read:*

```
rank = i / n
offset = mod(i,n)
call MPI_Win_lock(MPI_LOCK_SHARED, rank, 0, &
                  win, ierr )
call MPI_Get(value, 1, MPI_DOUBLE_PRECISION, &
        rank, offset, 1, MPI_DOUBLE_PRECISION, &
        win, ierr )
call MPI_Win_unlock(rank, win, ierr )
```

Accessing the Global Array From a Local Process

- The issues
  - Cache coherence (if no hardware)
  - Data in register

- *To read:*

```
volatile double *locala;
rank = i / n;
offset = i % n;
MPI_Win_lock(MPI_LOCK_SHARED, rank, 0, win);
if (rank == myrank) {
    value = locala[offset];
}
else {
    MPI_Get(&value, 1, MPI_DOUBLE,
            rank, offset, 1, MPI_DOUBLE, win);
}
MPI_Win_unlock(rank, win);
```

Accessing the Global Array From a Local Process (C++)

- The issues
  - Cache coherence (if no hardware)
  - Data in register
- *To read:*

```
volatile double *locala;
rank = i / n;
offset = i % n;
win.Lock(MPI::LOCK_SHARED, rank, 0);
if (rank == myrank) {
    value = locala[offset];
}
else {
    win.Get(&value, 1, MPI::DOUBLE,
            rank, offset, 1, MPI::DOUBLE);
}
win.Unlock(rank);
```

Accessing the Global Array From a Local Process (Fortran)

- The issues
  - Cache coherence (if no hardware)
  - Data in register
  - (We’ll come back to this case)
- *To read:*

```
double precision locala(0:mysize-1)
rank = i / n;
offset = mod(i,n)
call MPI_Win_lock(MPI_LOCK_SHARED, rank, 0, win, ierr)
if (rank .eq. myrank) then
    value = locala(offset);
else
    call MPI_Get(&value, 1, MPI_DOUBLE_PRECISION, &
            rank, offset, 1, MPI_DOUBLE_PRECISION, &
            win, ierr )
endif
call MPI_Win_unlock(rank, win, ierr )
```

Memory for Passive Target RMA

- Passive target operations are *harder* to implement
  - Hardware support helps
- MPI *allows* (but does not require) an implementation to require that windows objects used for passive target RMA use local windows allocated with MPI_Alloc_mem

Allocating Memory

- MPI_Alloc_mem, MPI_Free_mem
- Special Issue: Checking for no memory available:
  - e.g., the Alloc_mem equivalent of a null return from malloc
  - Default error behavior of MPI is to abort
- Solution:
  - Change the error handler on MPI_COMM_WORLD to MPI_ERRORS_RETURN, using MPI_COMM_SET_ERRHANDLER (in MPI-1, MPI_ERRHANDLER_SET)
  - Check error *class* with MPI_ERROR_CLASS
    - *Error **codes** are not error **classes***

Using MPI_Alloc_mem from Fortran

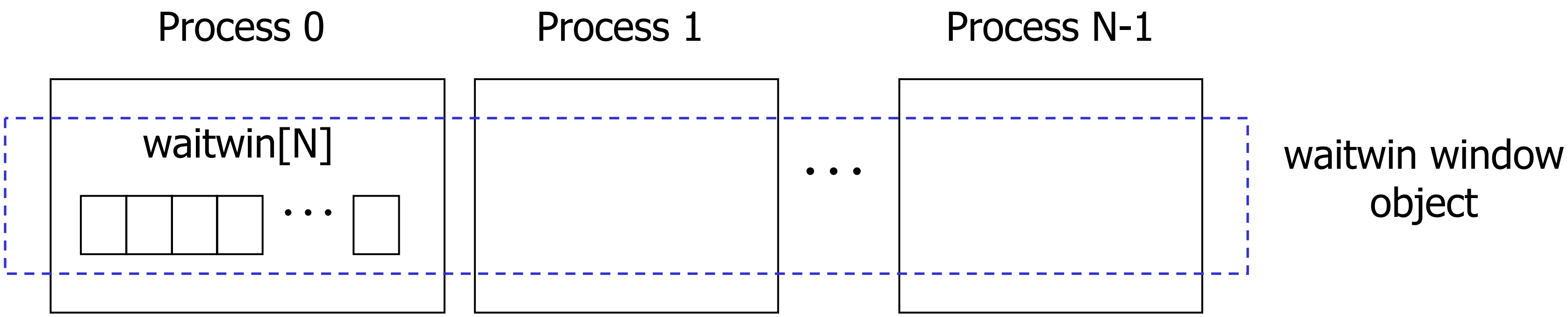
- No general solution, but some Fortran extensions allow the following:

```
double precision u
pointer (p, u(0:50,0:20))
integer (kind=MPI_ADDRESS_KIND) size
integer sizeofdouble, ierror
! careful with size (must be MPI_ADDRESS_KIND)
call MPI_SIZEOF(u, sizeofdouble, ierror)
size = 51 * 21 * sizeofdouble
call MPI_ALLOC_MEM(size, MPI_INFO_NULL, p, ierror)
...
... program may now refer to u, including passing it
... to MPI_WIN_CREATE
...
call MPI_FREE_MEM(u, ierror)  ! not p!
```

Mutex with Passive Target RMA

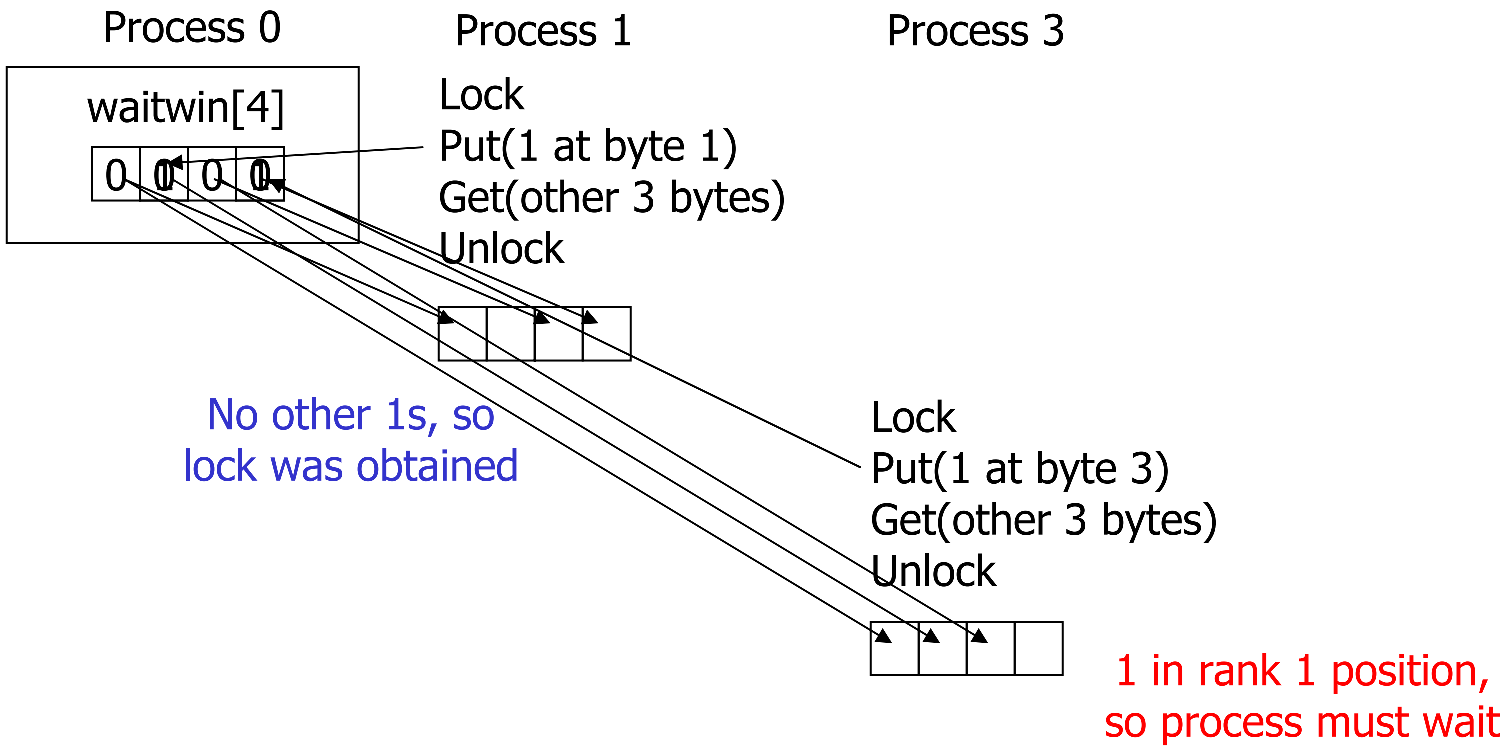
- MPI_Win_lock/unlock DO NOT define a critical section
- One has to implement a distributed locking algorithm using passive target RMA operations in order to achieve the equivalent of a mutex
- Example follows

Implementing Mutex



- Create “waitwin” window object
  - One process has N-byte array (byte per process)
- One access epoch to try to lock
  - Put “1” into corresponding byte
  - Get copy of all other values
- If all other values are zero, obtained lock
- Otherwise must wait

Attempting to lock



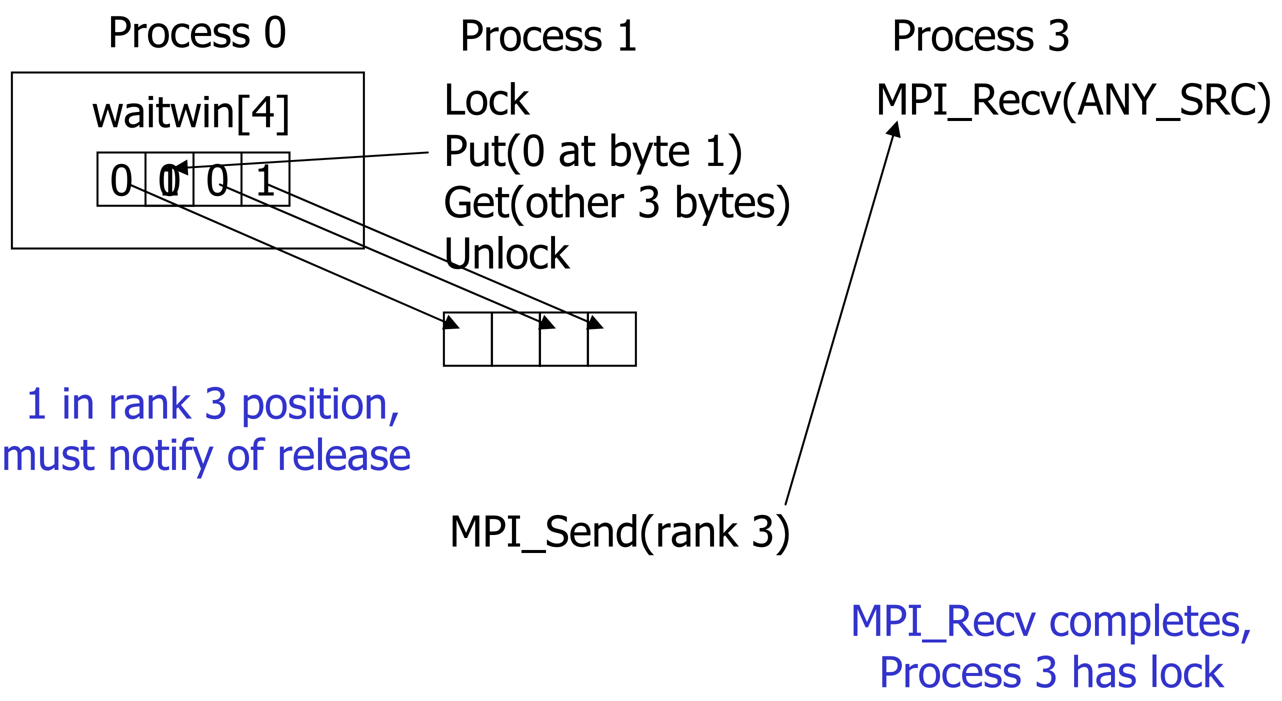
- Processes use one access epoch to attempt to obtain the lock
- Process 1 succeeds, but process 3 must wait

Waiting for the lock

- Naïve approach: simply MPI_Get the other bytes over and over
  - Lots of extra remote memory access
  - Better approach is to somehow notify waiting processes
  - Using RMA, set up a second window object with a byte on each process, spin-wait on local memory
    - *This approach is like MCS locks*
    - *Lots of wasted CPU cycles spinning*

- Better approach: Using MPI-1 point-to-point, send a zero-byte message to the waiting process to notify it that it has the lock
    - *Let MPI implementation handle checking for message arrival*

Releasing the Lock



Process 0

waitwin[4]

0 0 0 1

Process 1

Lock
Put(0 at byte 1)
Get(other 3 bytes)
Unlock

Process 3

MPI_Recv(ANY_SRC)

1 in rank 3 position,
must notify of release

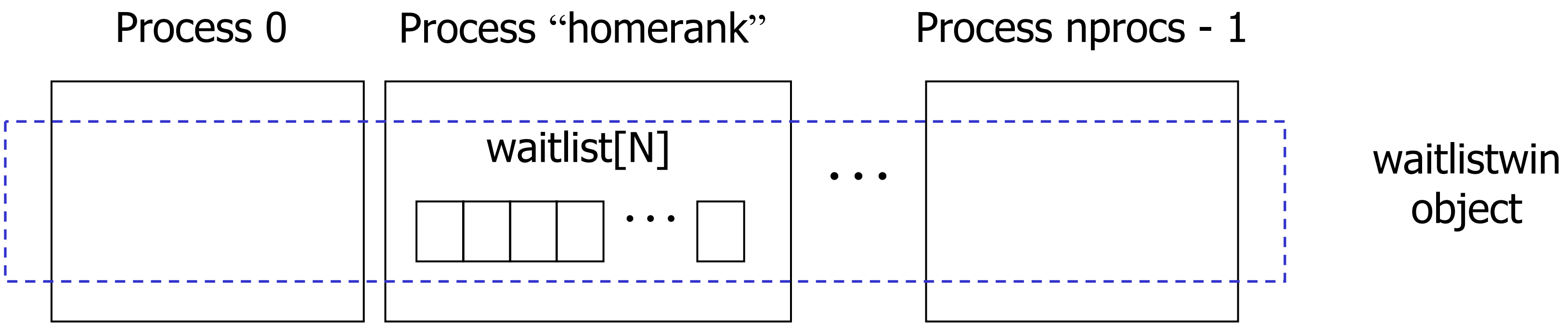
MPI_Send(rank 3)

MPI_Recv completes,
Process 3 has lock

- Process 1 uses one access epoch to release the lock
- Because process 3 is waiting, process 1 must send a message to notify process 3 that it now owns the lock

Mutex Code Walkthrough

- Code allows any process to be the "home" of the array:



- mpimutex_t type, for reference:

```
typedef struct mpimutex {
        int nprocs, myrank, homerank;
        MPI_Comm comm;
        MPI_Win waitlistwin;
        MPI_Datatype waitlisttype;
        unsigned char *waitlist;
} *mpimutex_t;
```

See mpimutex.c for code example.

Comments on Local Access

- Volatile:
  - Tells compiler that some other agent (such as another thread or process) may change the value
  - In practice, rarely necessary for arrays but *usually necessary for scalars*
  - Volatile is *not* just for MPI-2. Any shared-memory program needs to worry about this (even for cache-coherent shared-memory systems)
- Fortran users don't have volatile (yet):
  - But they can use the following evil trick

Simulating Volatile for Fortran

- Replace MPI_Win_unlock with
  ```
  subroutine My_Win_unlock(rank, win, var,
  ierr)
  integer rank, win, ierr
  double precision var
  call MPI_Win_unlock(rank, win)
  return
  ```
- When used in Fortran code, the compiler only sees
  ```
  call My_Win_unlock(rank, win, var, ierr)
  ```
  and assumes that `var` might be changed, causing the compiler to reload `var` from memory rather than using a value in register

Improving Performance

- MPI provides ways to tune for performance
- I/O
  - Using the right functions the right way
  - Providing Hints
- RMA
  - Asserts and info

Tuning MPI-IO

General Guidelines for Achieving High I/O Performance

- Buy sufficient I/O hardware for the machine
- Use fast file systems, not NFS-mounted home directories
- Do not perform I/O from one process only
- Make large requests wherever possible
- For noncontiguous requests, use derived datatypes and a single collective I/O call

Using the Right MPI-IO Function

- Any application as a particular “I/O access pattern” based on its I/O needs
- The same access pattern can be presented to the I/O system in different ways depending on what I/O functions are used and how
- In our SC98 paper, we classify the different ways of expressing I/O access patterns in MPI-IO into four *levels*: level 0 – level 3
- We demonstrate how the user’s choice of *level* affects performance

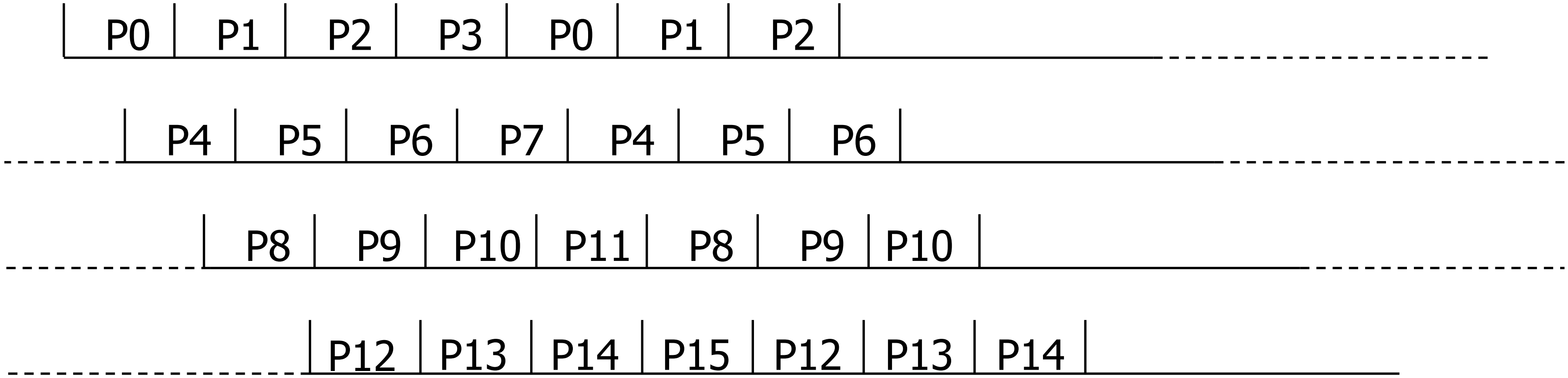
Example: Distributed Array Access

Large array distributed among 16 processes

P0	P1	P2	P3
P4	P5	P6	P7
P8	P9	P10	P11
P12	P13	P14	P15

Each square represents a subarray in the memory of a single process

Access Pattern in the file



P0 P1 P2 P3 P0 P1 P2

P4 P5 P6 P7 P4 P5 P6

P8 P9 P10 P11 P8 P9 P10

P12 P13 P14 P15 P12 P13 P14

Level-0 Access

- Each process makes one independent read request for each row in the local array (as in Unix)

```
MPI_File_open(..., file, ..., &fh)
for (i=0; i<n_local_rows; i++) {
    MPI_File_seek(fh, ...);
    MPI_File_read(fh, &(A[i][0]), ...);
}
MPI_File_close(&fh);
```

Level-1 Access

- Similar to level 0, but each process uses collective I/O functions

```
MPI_File_open(MPI_COMM_WORLD, file, ..., &fh);
for (i=0; i<n_local_rows; i++) {
    MPI_File_seek(fh, ...);
    MPI_File_read_all(fh, &(A[i][0]), ...);
}
MPI_File_close(&fh);
```

Level-2 Access

- Each process creates a derived datatype to describe the noncontiguous access pattern, defines a file view, and calls independent I/O functions

```
MPI_Type_create_subarray(..., &subarray, ...);
MPI_Type_commit(&subarray);
MPI_File_open(..., file, ..., &fh);
MPI_File_set_view(fh, ..., subarray, ...);
MPI_File_read(fh, A, ...);
MPI_File_close(&fh);
```

Level-3 Access

- Similar to level 2, except that each process uses collective I/O functions

```
 MPI_Type_create_subarray(..., &subarray,  ...);
MPI_Type_commit(&subarray);
MPI_File_open(MPI_COMM_WORLD, file,..., &fh);
MPI_File_set_view(fh, ..., subarray, ...);
MPI_File_read_all(fh, A, ...);
MPI_File_close(&fh);
```

Level-0 Access (C++)

- Each process makes one independent read request for each row in the local array (as in Unix)

```
fh = MPI::File::Open(..., file, ...)
for (i=0; i<n_local_rows; i++) {
    fh.Seek(...);
    fh.Read(&(A[i][0]), ...);
}
fh.Close();
```

Level-1 Access (C++)

- Similar to level 0, but each process uses collective I/O functions

```
fh = MPI::File::Open(MPI::COMM_WORLD,
                     file, ...);
for (i=0; i<n_local_rows; i++) {
    fh.Seek(...);
    fh.Read_all(&(A[i][0]), ...);
}
fh.Close();
```

Level-2 Access (C++)

- Each process creates a derived datatype to describe the noncontiguous access pattern, defines a file view, and calls independent I/O functions

```
subarray = MPI::Type_create_subarray(...);
subarray.Commit();
fh = MPI::File::Open(..., file, ...);
fh.Set_view(..., subarray, ...);
fh.Read(A, ...);
fh.Close();
```

Level-3 Access (C++)

- Similar to level 2, except that each process uses collective I/O functions

```
subarray = MPI::Type::Create_subarray(...);
subarray.Commit();
fh = MPI::File::Open(MPI::COMM_WORLD,
                     file, ...);
fh.Set_view(..., subarray,...);
fh.Read_all(A, ...);
fh.Close();
```

Level-0 Access (Fortran)

- Each process makes one independent read request for each row in the local array (as in Unix)

```
call MPI_File_open(..., file, ...,fh,ierr)
do i=1, n_local_rows
    call MPI_File_seek(fh, ..., ierr)
    call MPI_File_read(fh, a(i,0),...,ierr)
enddo
call MPI_File_close(fh, ierr)
```

Level-1 Access (Fortran)

- Similar to level 0, but each process uses collective I/O functions

```
call MPI_File_open(MPI_COMM_WORLD, file,&
               ..., fh, ierr)
do i=1,n_local_rows
    call MPI_File_seek(fh, ..., ierr)
    call MPI_File_read_all(fh, a(i,0), ...,&
                           ierr)
enddo
call MPI_File_close(fh,ierr)
```

Level-2 Access (Fortran)

- Each process creates a derived datatype to describe the noncontiguous access pattern, defines a file view, and calls independent I/O functions

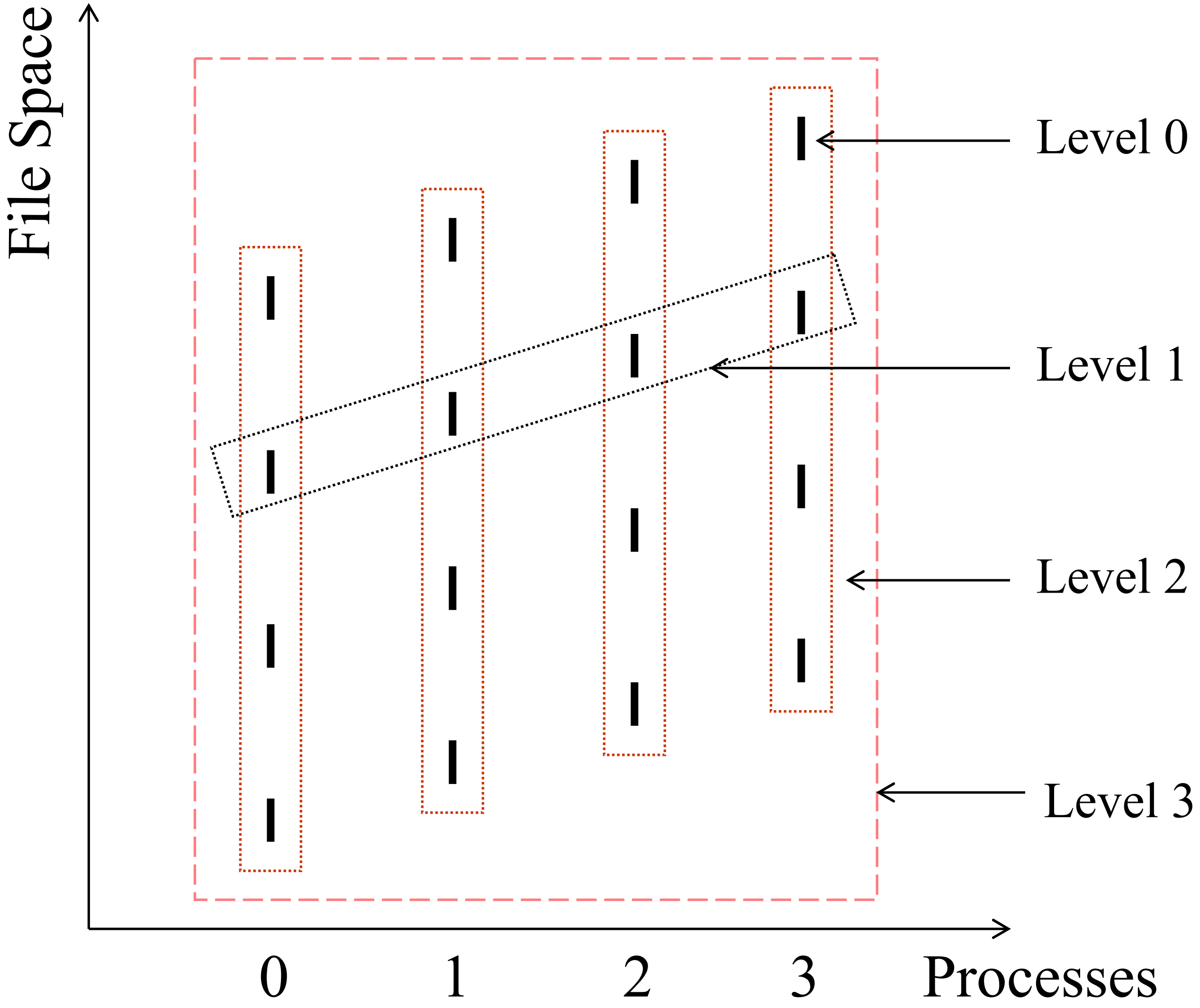
```
call MPI_Type_create_subarray(..., &
                       subarray, ..., ierr)
call MPI_Type_commit(subarray, ierr)
call MPI_File_open(..., file,..., fh, ierr)
call MPI_File_set_view(fh, ..., subarray,&
         ..., ierr)
call MPI_File_read(fh, A, ..., ierr)
call MPI_File_close(fh, ierr )
```

Level-3 Access (Fortran)

- Similar to level 2, except that each process uses collective I/O functions

```
call MPI_Type_create_subarray(..., &
              subarray, ierr )
call MPI_Type_commit(subarray, ierr )
call MPI_File_open(MPI_COMM_WORLD, file,&
               ..., fh, ierr )
call MPI_File_set_view(fh, ..., subarray,&
                    ..., ierr )
call MPI_File_read_all(fh, A, ..., ierr)
call MPI_File_close(fh,ierr)
```

The Four Levels of Access



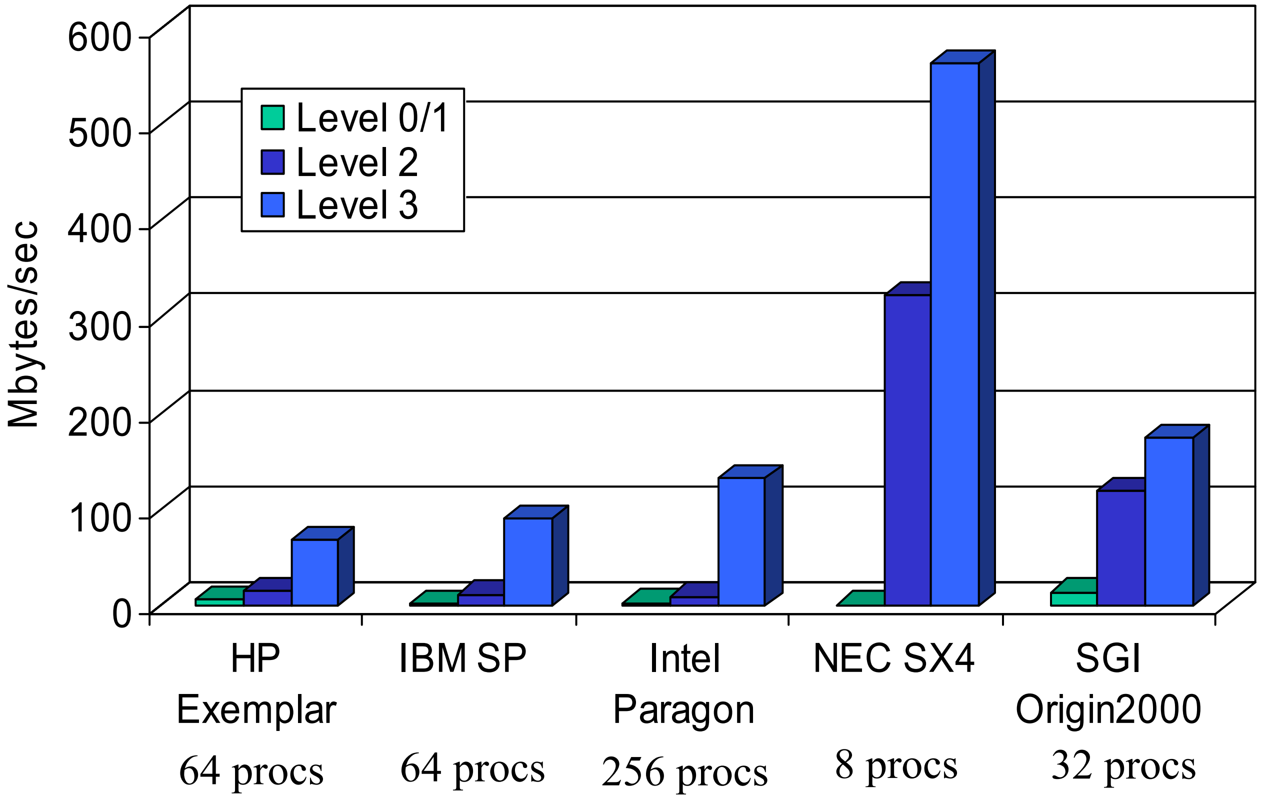
Optimizations

- Given complete access information, an implementation can perform optimizations such as:
  - Data Sieving: Read large chunks and extract what is really needed
  - Collective I/O: Merge requests of different processes into larger requests
  - Improved prefetching and caching

Performance Results

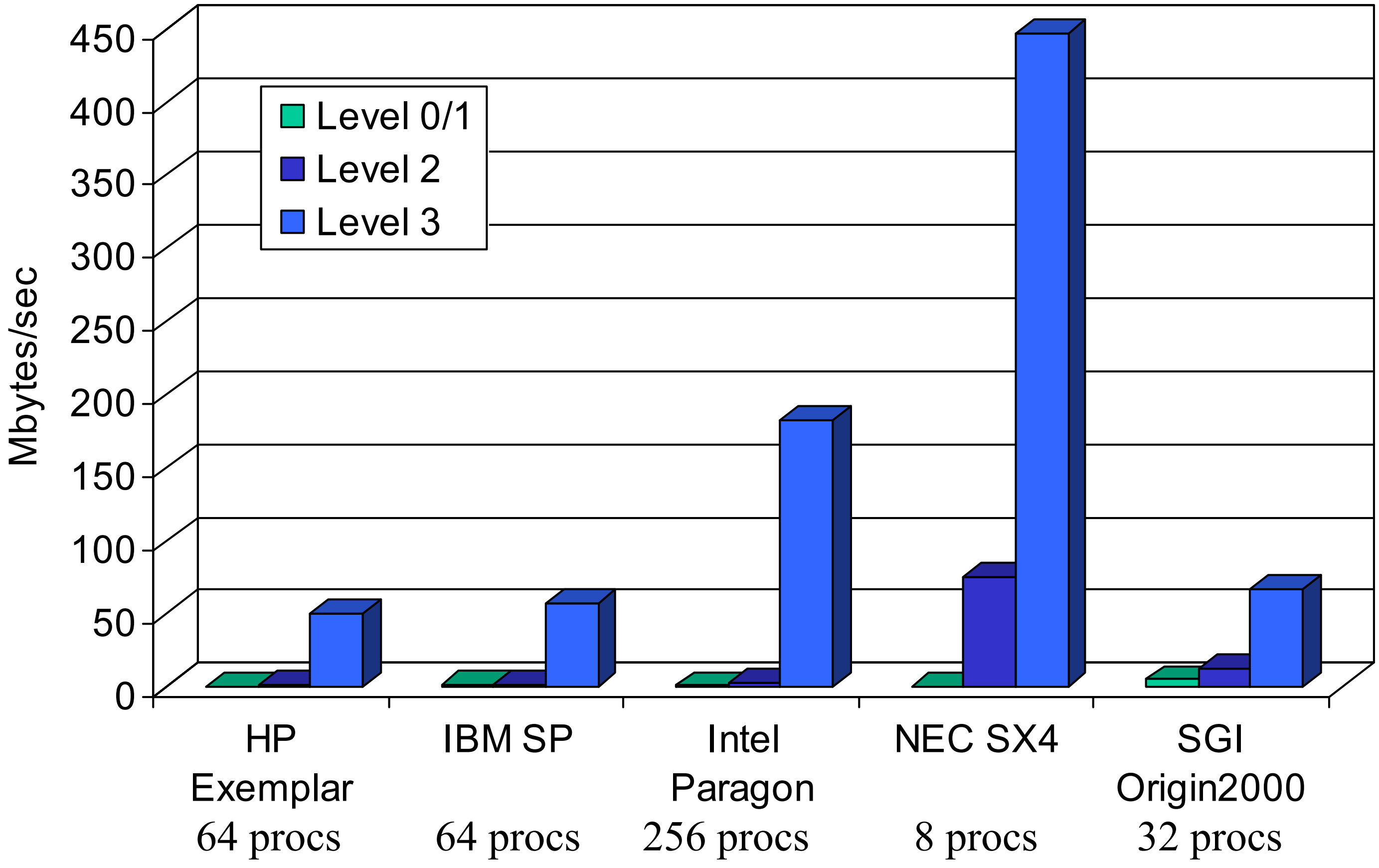
- Distributed array access
- Unstructured code from Sandia
- On five different parallel machines:
  - HP Exemplar
  - IBM SP
  - Intel Paragon
  - NEC SX-4
  - SGI Origin2000

Distributed Array Access: Read Bandwidth



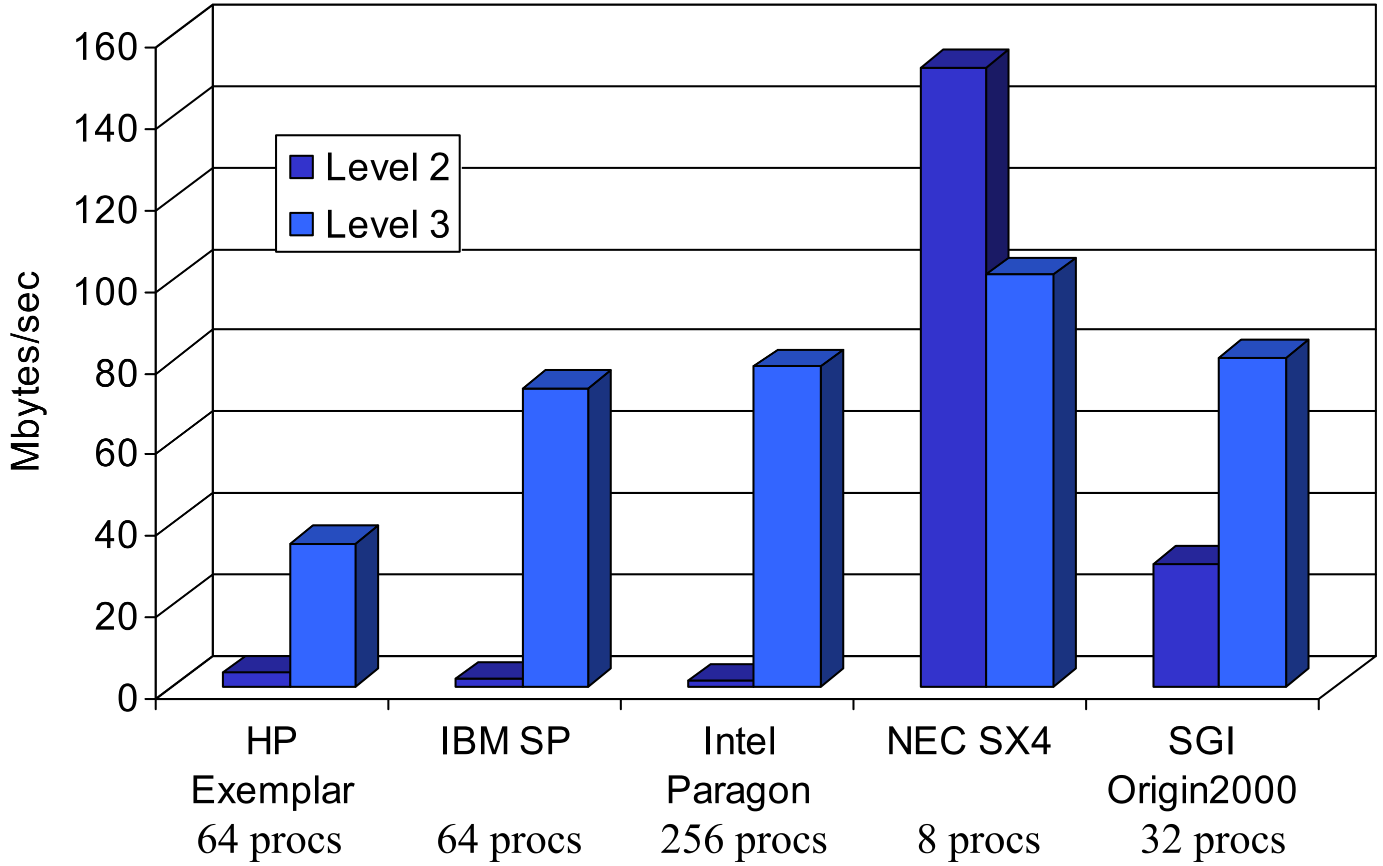
Array size: 512 x 512 x 512

Distributed Array Access: Write Bandwidth

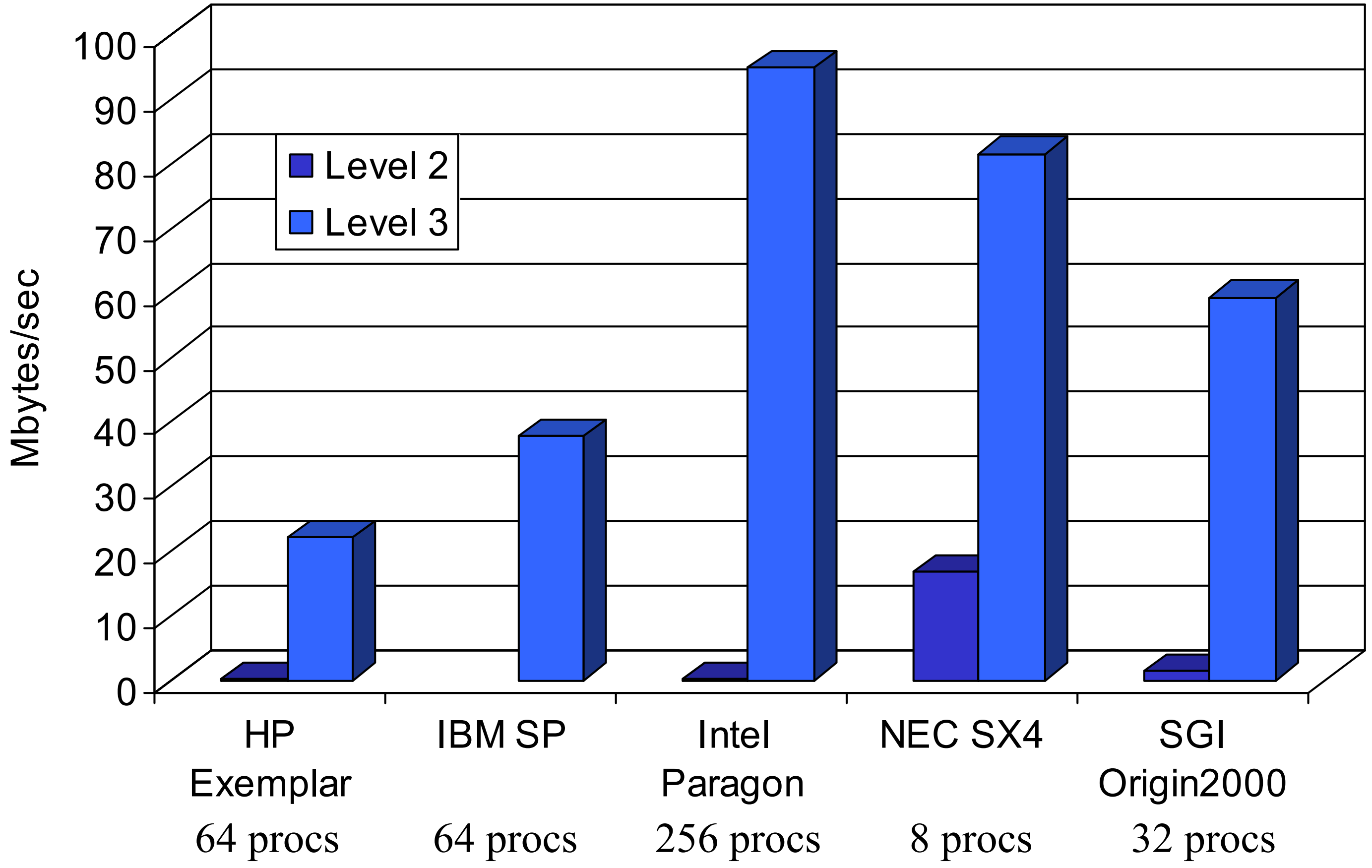


Array size: 512 x 512 x 512

Unstructured Code: Read Bandwidth

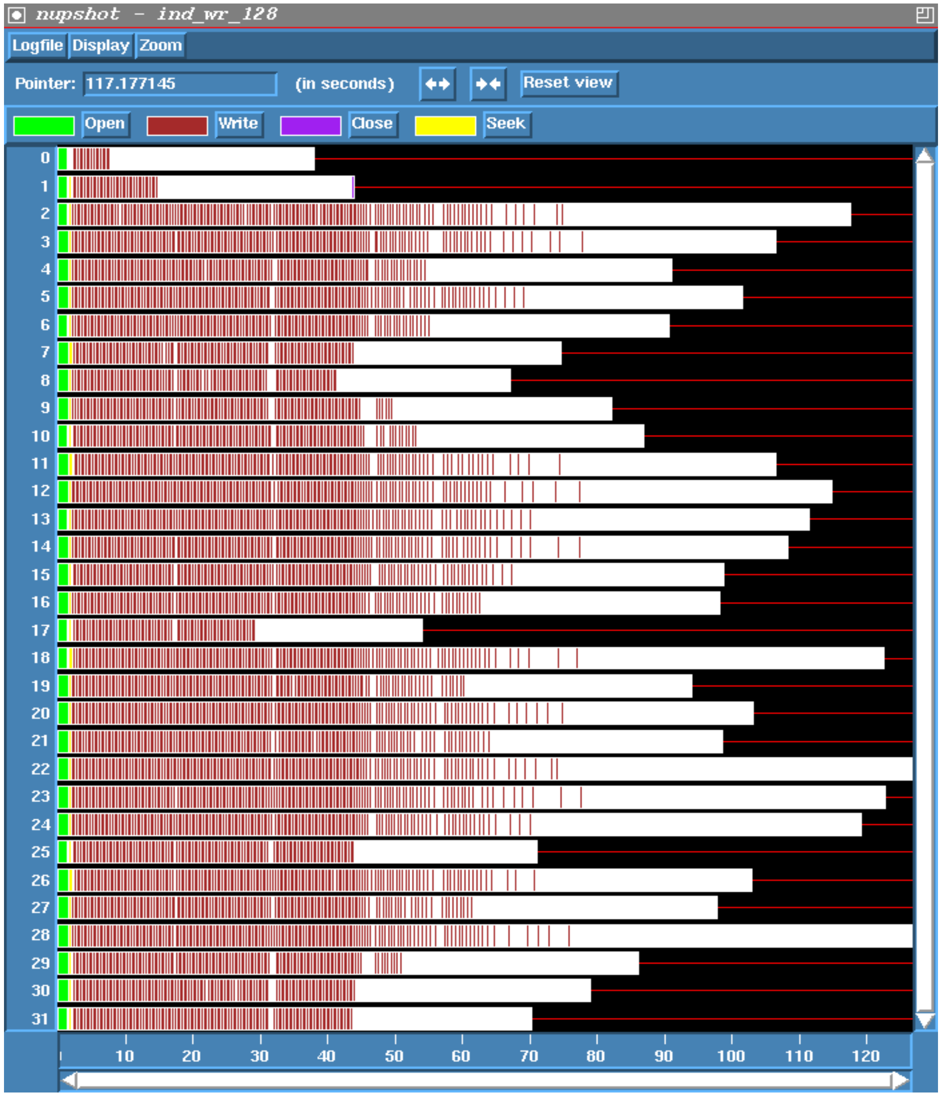


Unstructured Code: Write Bandwidth



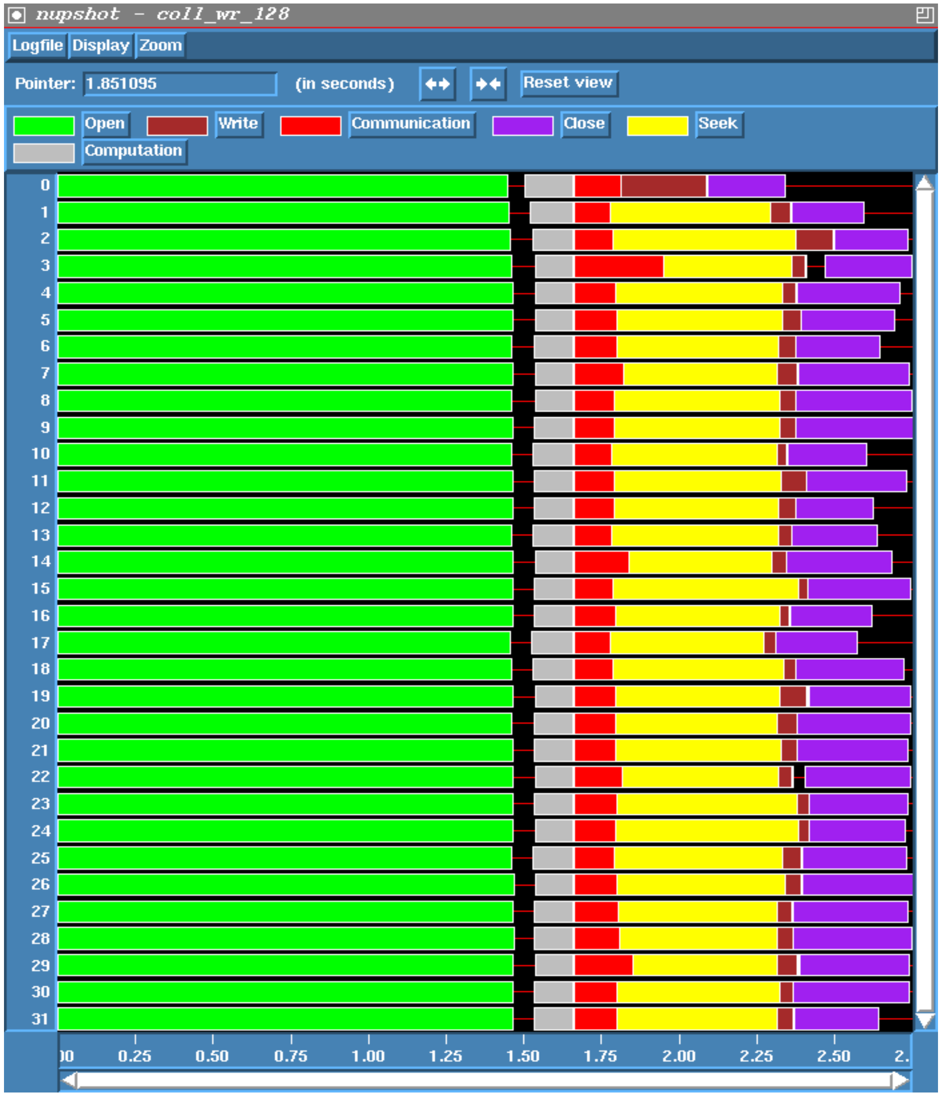
Independent Writes

- On Paragon
- Lots of seeks and small writes
- Time shown = 130 sec.



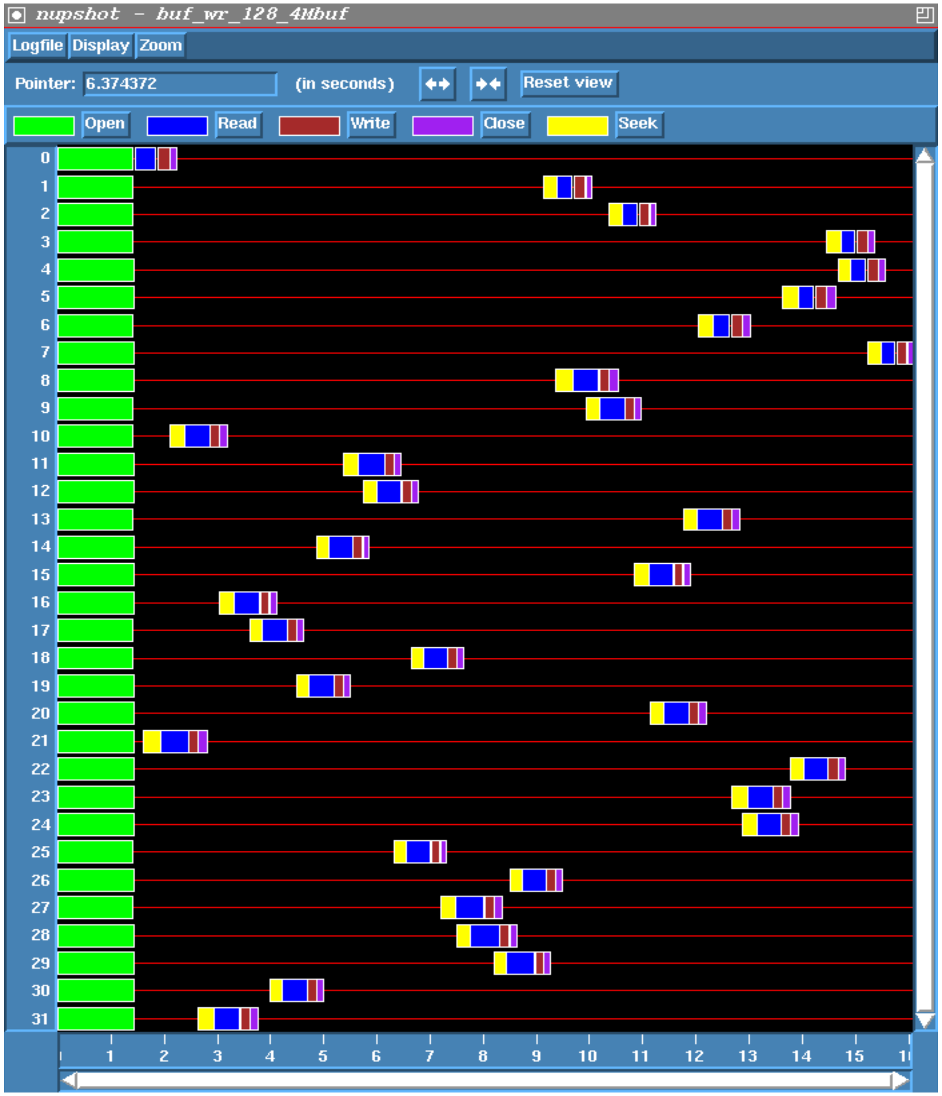
Collective Write

- On Paragon
- Computation and communication precede seek and write
- Time shown = 2.75 sec.



Independent Writes with Data Sieving

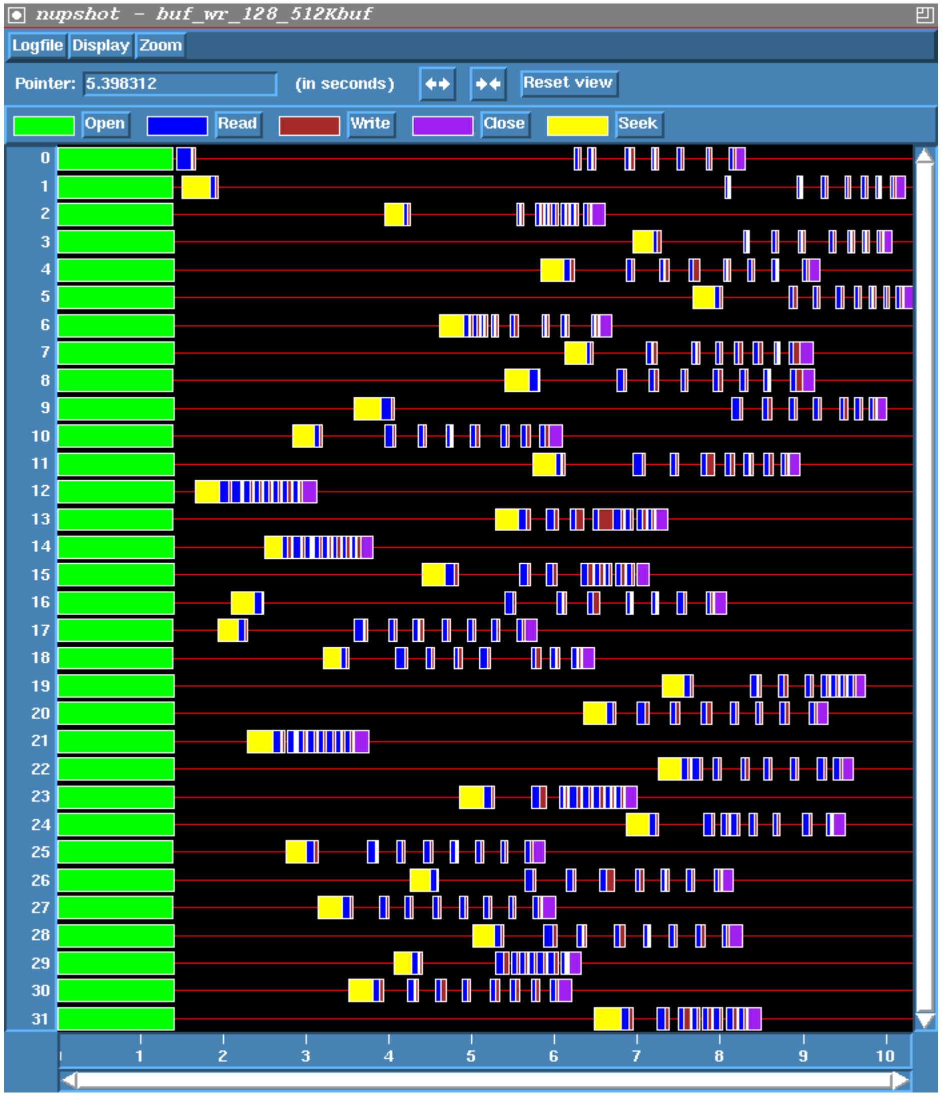
- On Paragon
- Access data in large “blocks” and extract needed data
- Requires lock, read, modify, write, unlock for writes
- 4 MB blocks
- Time = 16 sec.



Pointer: 6.374372 (in seconds)

Changing the Block Size

- Smaller blocks mean less contention, therefore more parallelism
- 512 KB blocks
- Time = 10.2 sec.

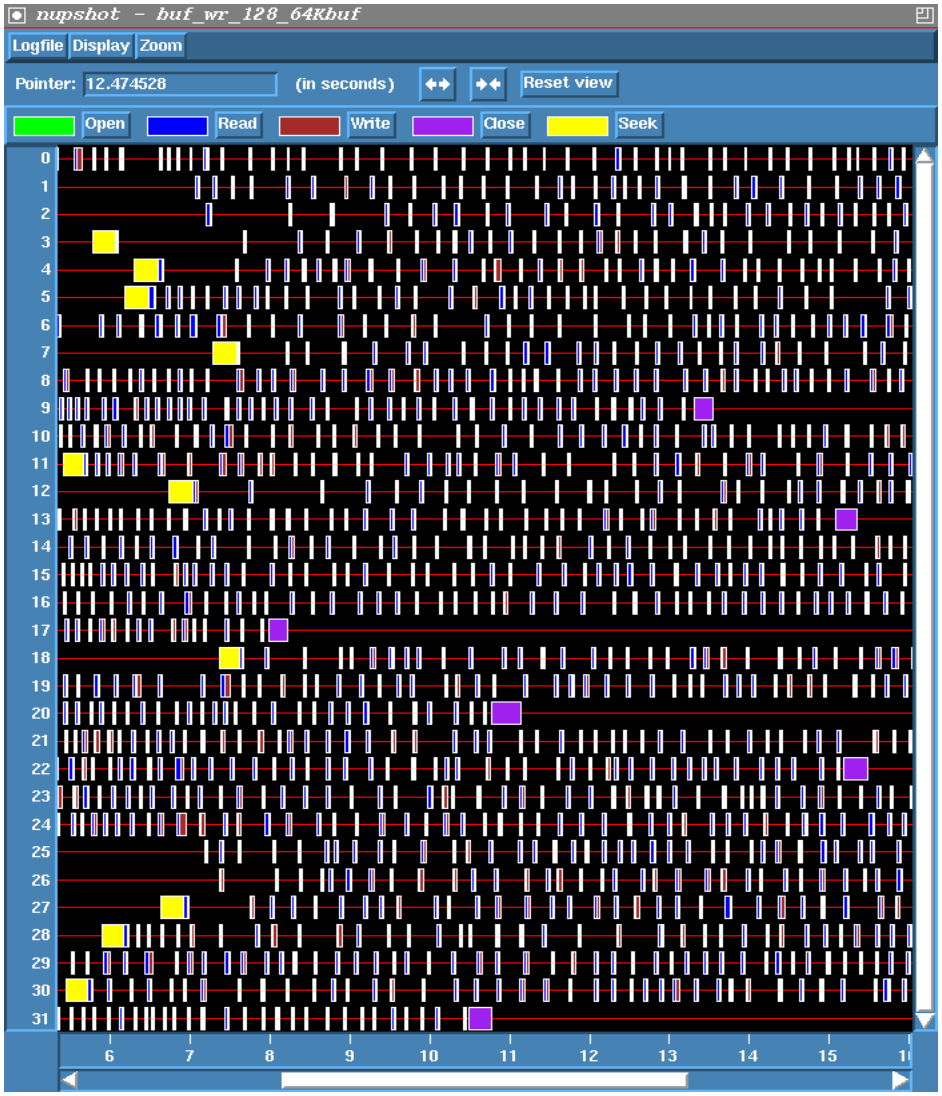


nupshot - buf_wr_128_512Kbuf

Pointer: 5.398312 (in seconds)

Data Sieving with Small Blocks

- If the block size is too small, however, the increased parallelism doesn’t make up for the many small writes
- 64 KB blocks
- Time = 21.5 sec.



Pointer: 12.474528

Passing Hints

- MPI-2 defines a new object, **`MPI_Info`**
- Provides an extensible list of key=value pairs
- Used in I/O, One-sided, and Dynamic to package variable, optional types of arguments that may not be standard

Passing Hints to MPI-IO

```
MPI_Info info;

MPI_Info_create(&info);

/* no. of I/O devices to be used for file striping */
MPI_Info_set(info, "striping_factor", "4");

/* the striping unit in bytes */
MPI_Info_set(info, "striping_unit", "65536");

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_CREATE | MPI_MODE_RDWR, info, &fh);

MPI_Info_free(&info);
```

Passing Hints to MPI-IO (C++)

```
MPI::Info info;

info = MPI::Info_create();

// no. of I/O devices to be used for file striping
info.Set("striping_factor", "4");

// the striping unit in bytes
info.Set("striping_unit", "65536");

fh = MPI::File::Open(MPI::COMM_WORLD, "/pfs/datafile",
              MPI::MODE_CREATE | MPI::MODE_RDWR, info);

info.Free();
```

Passing Hints to MPI-IO (Fortran)

```
integer info

call MPI_Info_create(info, ierr)

! no. of I/O devices to be used for file striping
call MPI_Info_set(info, "striping_factor", "4”, ierr )

! the striping unit in bytes
call MPI_Info_set(info, "striping_unit", "65536”, ierr )

call MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile", &
              MPI_MODE_CREATE + MPI_MODE_RDWR, info, &
              fh, ierr )

call MPI_Info_free( info, ierr )
```

Examples of Hints (used in ROMIO)

- `striping_unit`
- `striping_factor`
- `cb_buffer_size`
- `cb_nodes`

MPI-2 predefined hints

- `ind_rd_buffer_size`
- `ind_wr_buffer_size`

New Algorithm Parameters

- `start_iodevice`
- `pfs_svr_buf`
- `direct_read`
- `direct_write`

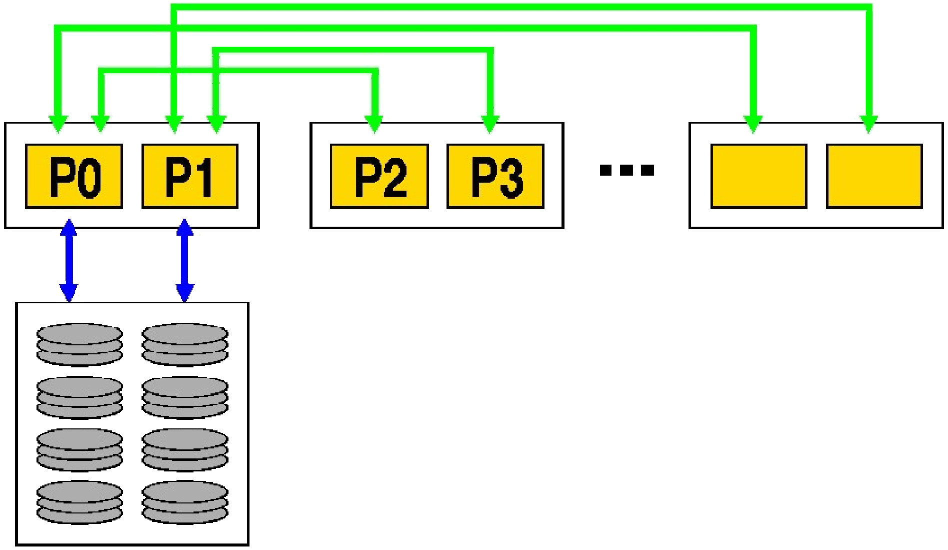
Platform-specific hints

ROMIO Hints and PVFS

- Controlling PVFS
  - `striping_factor` - size of “strips” on I/O servers
  - `striping_unit` - number of I/O servers to stripe across
  - `start_iodevice` - which I/O server to start with
- Controlling aggregation
  - `cb_config_list` - list of aggregators
  - `cb_nodes` - number of aggregators (upper bound)
- Tuning ROMIO optimizations
  - `romio_cb_read, romio_cb_write` - aggregation on/off
  - `romio_ds_read, romio_ds_write` - data sieving on/off

Aggregation Example

- Cluster of SMPs
- One SMP box has fast connection to disks
- Data is aggregated to processes on single box
- Processes on that box perform I/O on behalf of the others



Summary of I/O Tuning

- MPI I/O has many features that can help users achieve high performance
- The most important of these features are the ability to specify noncontiguous accesses, the collective I/O functions, and the ability to pass hints to the implementation
- Users must use the above features!
- In particular, when accesses are noncontiguous, users must create derived datatypes, define file views, and use the collective I/O functions

Common Errors in Using MPI-IO

- Not defining file offsets as `MPI_Offset` in C and `integer (kind=MPI_OFFSET_KIND)` in Fortran (or perhaps `integer*8` in Fortran 77)
- In Fortran, passing the offset or displacement directly as a constant (e.g., 0) in the absence of function prototypes (F90 mpi module)
- Using darray datatype for a block distribution other than the one defined in darray (e.g., floor division)
- filetype defined using offsets that are not monotonically nondecreasing, e.g., 0, 3, 8, 4, 6.
(can occur in irregular applications)

Tuning RMA

Performance Tuning RMA

- MPI provides *generality* and *correctness*
- Special cases may allow performance optimizations
  - MPI provides two ways to identify special cases:
    - *Assertion flags for MPI_Win_fence, etc.*
    - *Info values for MPI_Win_create and MPI_Alloc_mem*

Tuning Fence

- Asserts for fence
  - Note that these rely on understanding the "global/collective" use of the RMA calls in the code.

MPI_Win_fence Assert Values

- MPI_MODE_NOSTORE
  - No update to the local window was made by the local process (using assignments, e.g., **stores**) since the last call to MPI_Win_fence
- MPI_MODE_NOPUT
  - There will be no RMA (**Put** or Accumulate) to the local window before the next MPI_Win_fence
- MPI_MODE_NOPRECEDE
  - This MPI_Win_fence will not complete any RMA calls made by this process (no **preceding** RMA calls)
- MPI_MODE_NOSUCCEED
  - No RMA calls will be made on this window before the next MPI_Win_fence call (no **succeeding** (as in coming after) RMA calls)

Assert Values in Life Exchange

```
MPI_Win_fence(MPI_MODE_NOPRECEDE, win);

MPI_Put(&matrix[myrows][0], cols+2, MPI_INT,
         exch_next, 0, cols+2,  MPI_INT, win);

MPI_Put(&matrix[1][0], cols+2, MPI_INT, exch_prev,
     (nrows_prev+1)*(cols+2), cols+2, MPI_INT, win);

MPI_Win_fence(MPI_MODE_NOSTORE | MPI_MODE_NOPUT |
              MPI_MODE_NOSUCCEED, win);
```

Assert Values in Life Exchange (C++)

```
win.Fence(MPI::MODE_NOPRECEDE);
win.Put(&matrix[myrows][0], cols+2, MPI::INT,
         exch_next, 0, cols+2,  MPI::INT);

win.Put(&matrix[1][0], cols+2, MPI::INT,
        exch_prev, (nrows_prev+1)*(cols+2),
        cols+2, MPI::INT);

win.Fence(MPI_MODE_NOSTORE | MPI_MODE_NOPUT |
          MPI_MODE_NOSUCCEED);
```

Assert Values in Life Exchange (Fortran)

```
call MPI_Win_fence(MPI_MODE_NOPRECEDE, win, ierr )

call MPI_Put( matrix(myrows,0), cols+2, MPI_INTEGER, &
         exch_next, 0, cols+2,  MPI_INTEGER, win, &
         ierr)

call MPI_Put( matrix(1,0), cols+2, MPI_INTEGER, &
              exch_prev, &
     (nrows_prev+1)*(cols+2), cols+2, MPI_INT, win, &
              ierr )

call MPI_Win_fence(MPI_MODE_NOSTORE + &
          MPI_MODE_NOPUT + MPI_MODE_NOSUCCEED, win, &
          ierr )
```

Tuning P/S/C/W

- Asserts for MPI_Win_start and MPI_Win_post
- Start
  - MPI_MODE_NOCHECK
    - *Guarantees that the matching calls to MPI_Win_post have already been made*
- Post
  - MPI_MODE_NOSTORE, MPI_MODE_NOPUT
    - *Same meaning as for MPI_Win_fence*
  - MPI_MODE_NOCHECK
    - *Nocheck means that the matching calls to MPI_Win_start have* not *yet occurred*

MPI_Win_create

- If only active-target RMA will be used, pass an info object to MPI_Win_create with key “**`no_locks`**” set to “**`true`**”

```
MPI_Info info;
MPI_Info_create( &info );
MPI_Info_set( info, "no_locks", "true" );
MPI_Win_create( ..., info, ... );
MPI_Info_free( &info );
```

MPI_Win_create (C++)

- If only active-target RMA will be used, pass an info object to MPI_Win_create with key “**no_locks**” set to “**true**”

```
MPI::Info info;
info = MPI::Info::Create();
info.Set("no_locks", "true" );
win = MPI::Win::Create( ..., info, ... );
info.Free();
```

MPI_Win_create (Fortran)

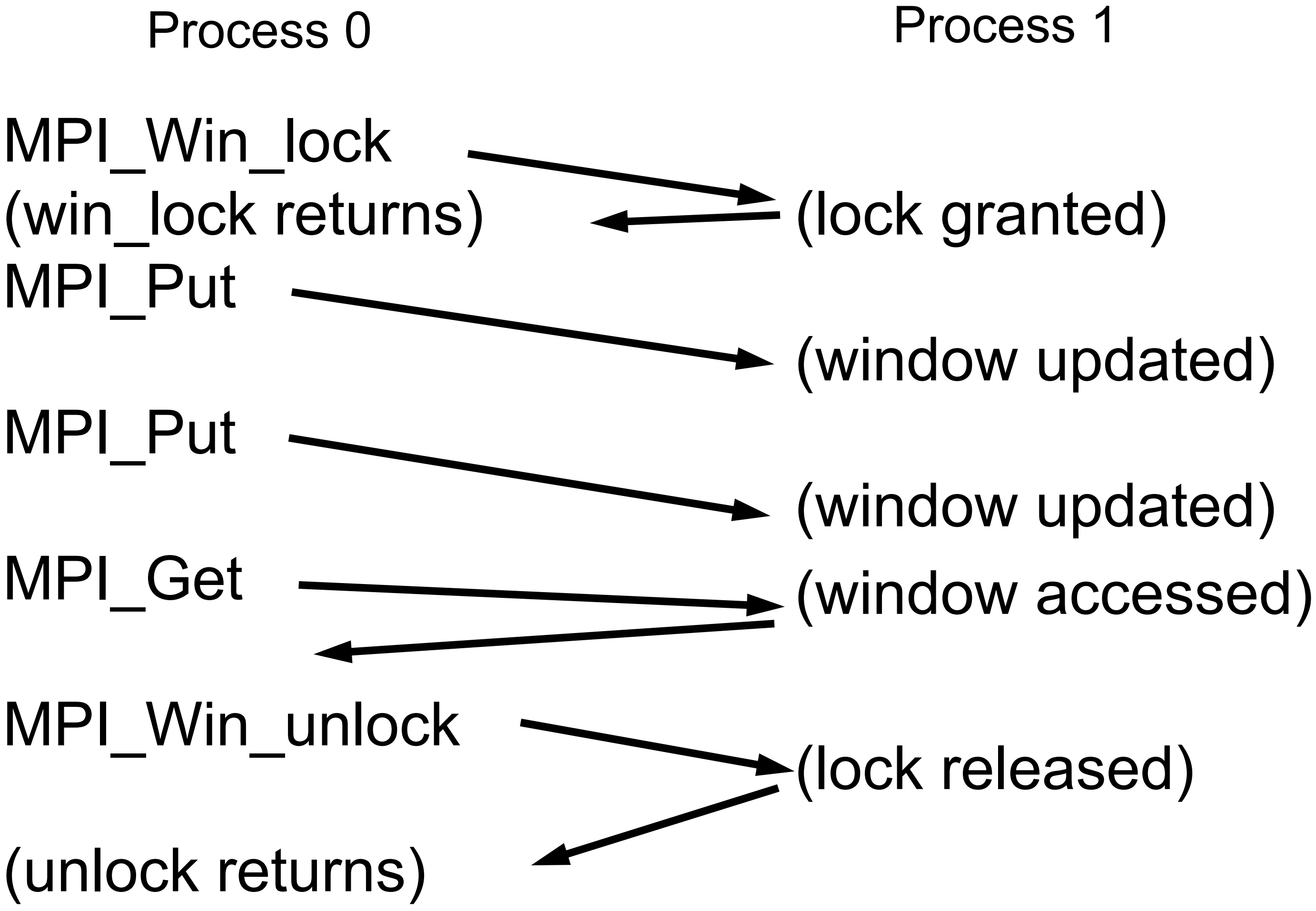
- If only active-target RMA will be used, pass an info object to MPI_Win_create with key “**no_locks**” set to “**true**”

```
Integer info;
call MPI_Info_create( info, ierr )
call MPI_Info_set( info, "no_locks", "true",
  ierr )
call MPI_Win_create( ..., info, ... , ierr)
call MPI_Info_free( info, ierr )
```

Understanding the MPI-2 Completion Model

- Very relaxed
  - To give the implementer the greatest flexibility
  - Describing this relaxed model precisely is difficult
    - *Implementer only needs to obey the rules*
  - But it doesn't matter; simple rules work for most programmers
- When does the data actually *move*?

Data Moves Early



Process 0

Process 1

MPI_Win_lock

(lock granted)

(win_lock returns)

MPI_Put

(window updated)

MPI_Put

(window updated)

MPI_Get

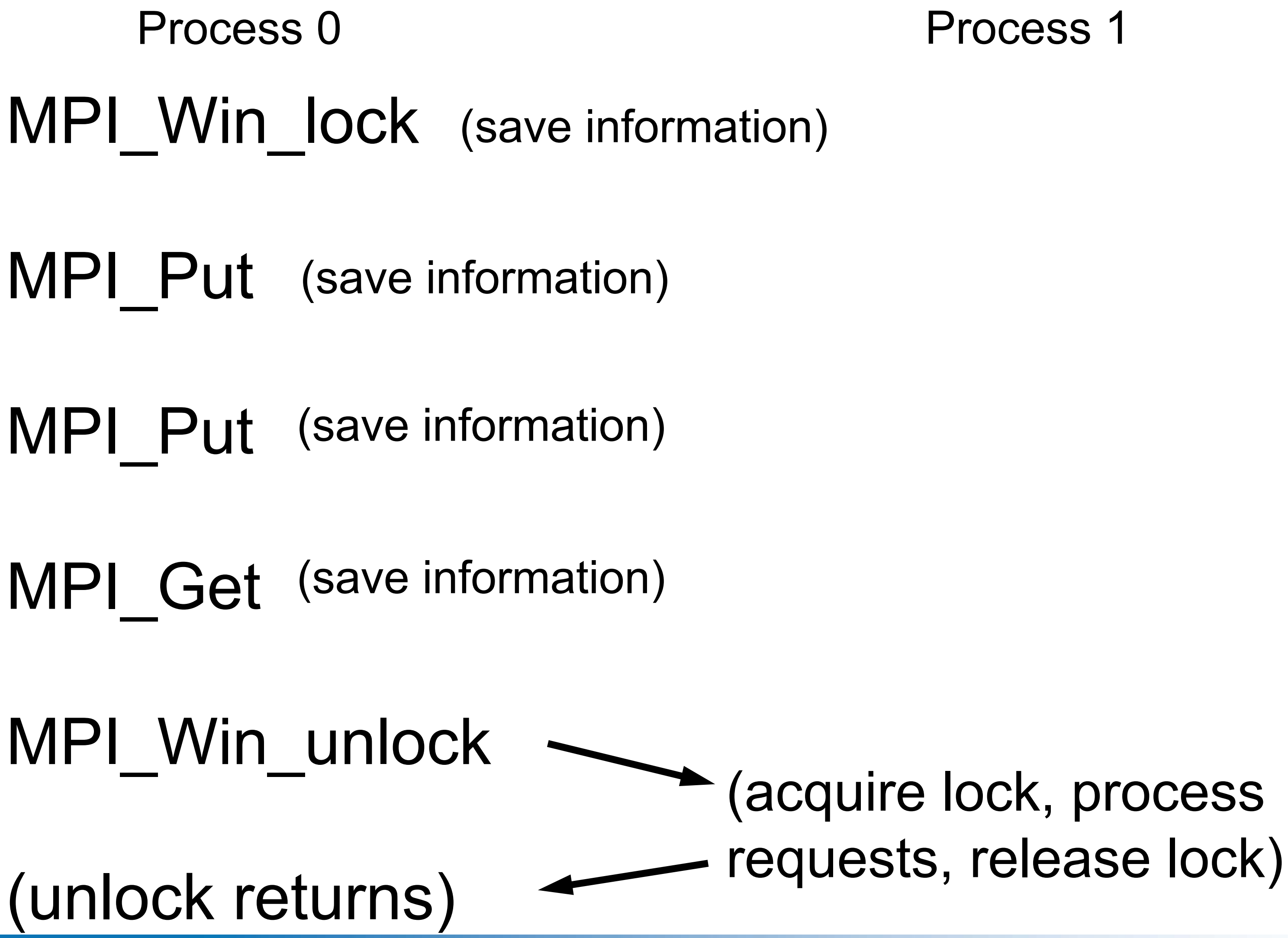
(window accessed)

MPI_Win_unlock

(lock released)

(unlock returns)

Data Moves Late



Process 0

Process 1

MPI_Win_lock (save information)

MPI_Put (save information)

MPI_Put (save information)

MPI_Get (save information)

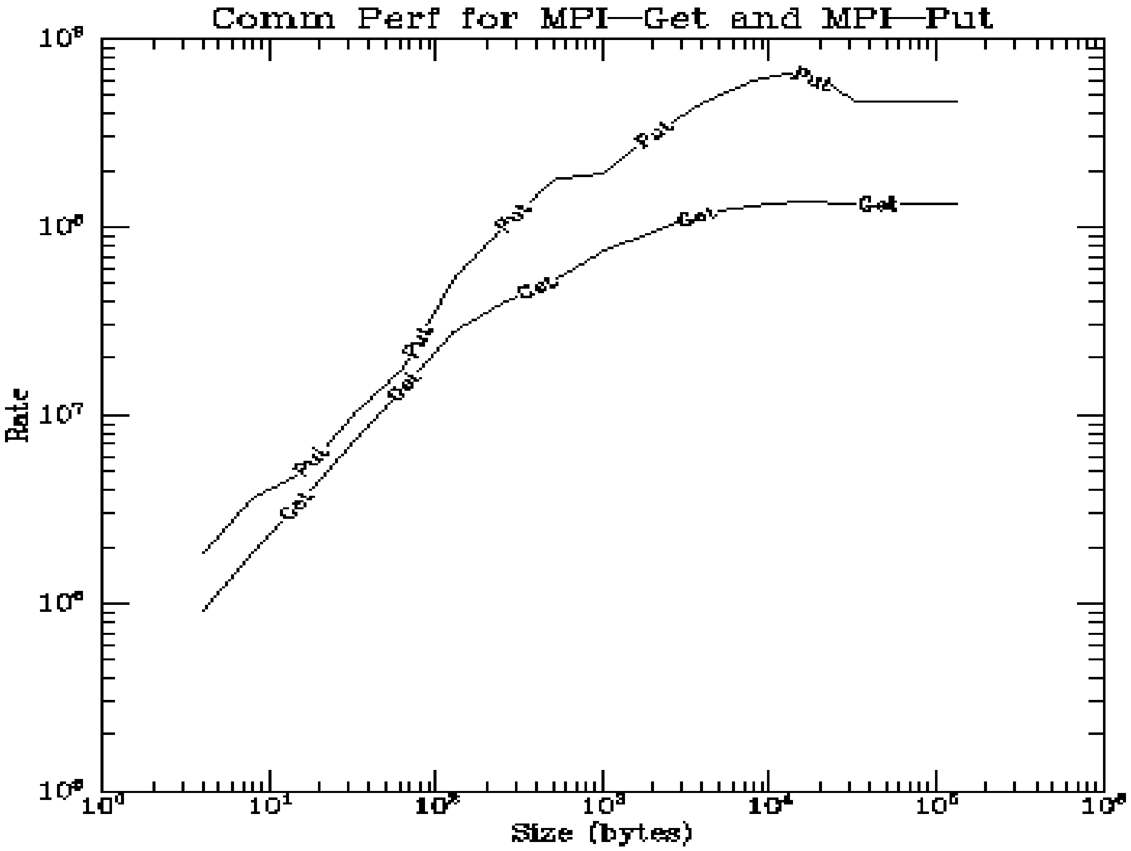
MPI_Win_unlock

(acquire lock, process requests, release lock)

(unlock returns)

Relative Performance of Put and Get

- Put and get may not give the same performance
- mpptest (http://www.mcs.anl.gov/mpi/mpptest) has performance tests for MPI_Put, MPI_Get, and other RMA operations
- Earth Simulator applications use the MPI RMA operations



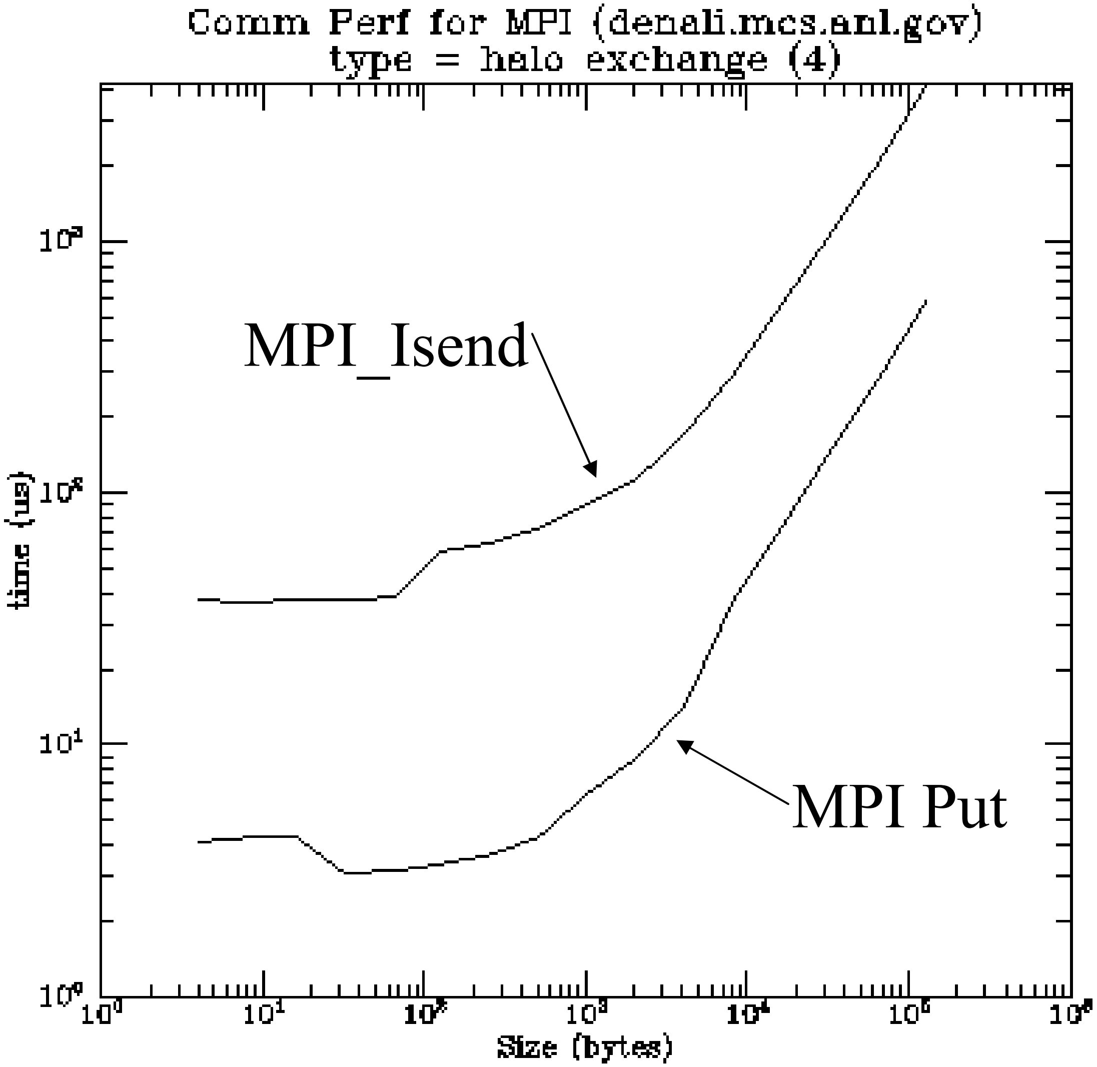
Caveat: Results for SGI implementation. Others may differ

Implementation Limitations

- MPI_PROC_NULL not recognized
  - MPI-Forum clarified standard that MPI_PROC_NULL is valid in Put/Get/Accumulate
  - Most implementations now correct
- Some implementations require special memory for (local) windows
  - MPI standard requires that *any* memory can be used with active-target RMA
- Both of these are bugs, but ones that you can work around

Comments on Exchange

- MPI_ADDRESS_KIND needed for offsets in Fortran
- Datatypes aren't always fast; consider using contiguous regions
- MPI_Put can be much faster that MPI Point-to-point
  - 4 neighbor exchange on SGI Origin



Implementation Limitations

- Not all MPI implementations include Passive target operations (e.g. not in LAM)
  - Passive send of contiguous datatypes safest
  - Passive target MPI_Accumulate and noncontiguous datatypes hardest to implement

Top MPI Errors

- Fortran: missing ierr argument
- Fortran: missing MPI_STATUS_SIZE on status
- Fortran: Using integers where MPI_OFFSET_KIND or MPI_ADDRESS_KIND integers are required (particularly in I/O)
- Fortran 90: Using array sections to nonblocking routines (e.g., MPI_Isend)
- All: MPI_Bcast not called collectively (e.g., sender bcasts, receivers use MPI_Recv)
- All: Failure to wait (or test for completion) on MPI_Request
- All: Reusing buffers on nonblocking operations
- All: Using a single process for all file I/O
- All: Using MPI_Pack/Unpack instead of Datatypes
- All: Unsafe use of blocking sends/receives
- All: Using MPI_COMM_WORLD instead of comm in libraries
- All: Not understanding implementation performance settings
- All: Failing to install and use the MPI implementation according to its documentation.

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/

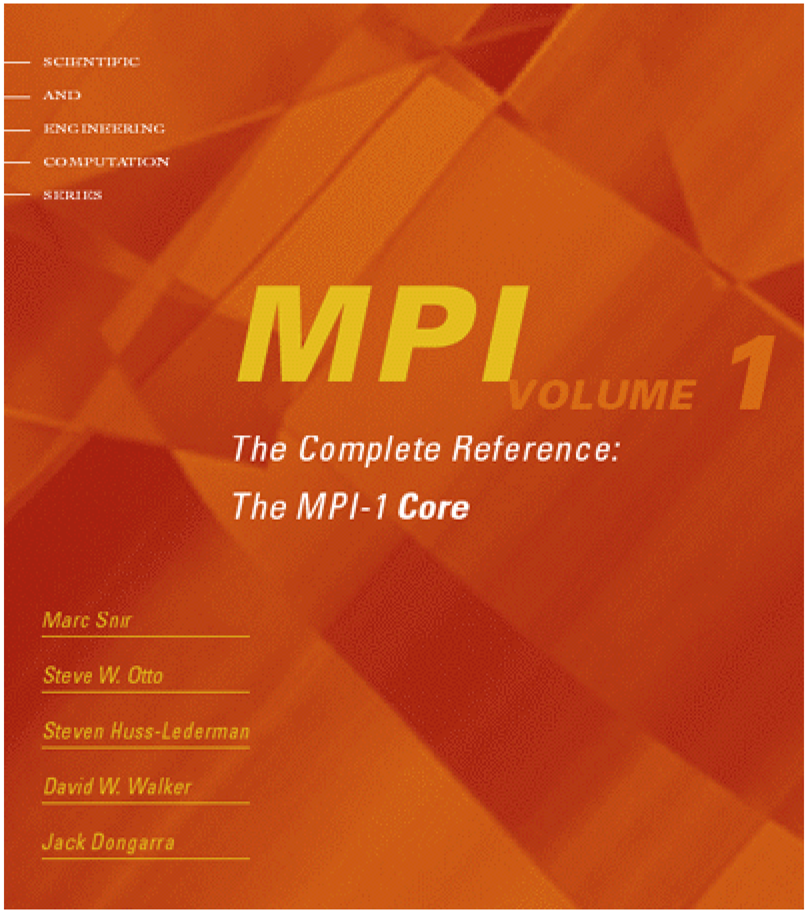
MPICH2

- 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
  - Coming soon from another major vendor
  - Having vendors adapt MPICH2 into their products has helped make it efficient and robust

The MPI Standard (1 & 2)

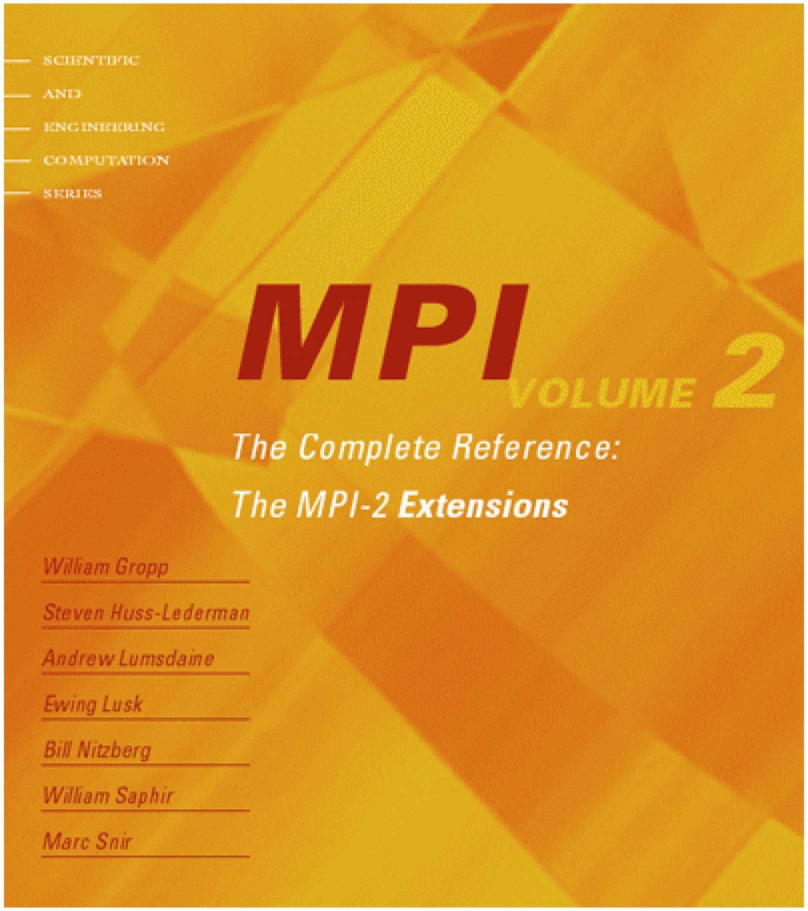


SCIENTIFIC
AND
ENGINEERING
COMPUTATION
SERIES

MPI VOLUME 1

The Complete Reference:
The MPI-1 **Core**

Marc Snir
Steve W. Otto
Steven Huss-Lederman
David W. Walker
Jack Dongarra



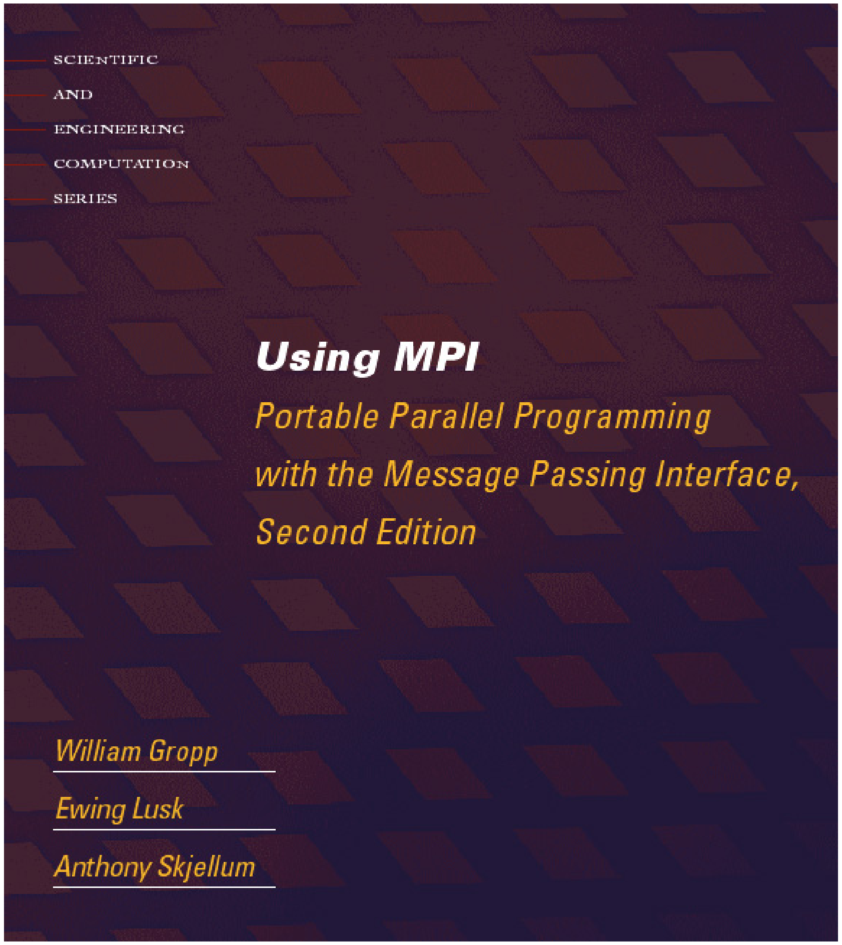
SCIENTIFIC
AND
ENGINEERING
COMPUTATION
SERIES

MPI VOLUME 2

The Complete Reference:
The MPI-2 **Extensions**

William Gropp
Steven Huss-Lederman
Andrew Lumsdaine
Ewing Lusk
Bill Nitzberg
William Saphir
Marc Snir

Tutorial Material on MPI, MPI-2

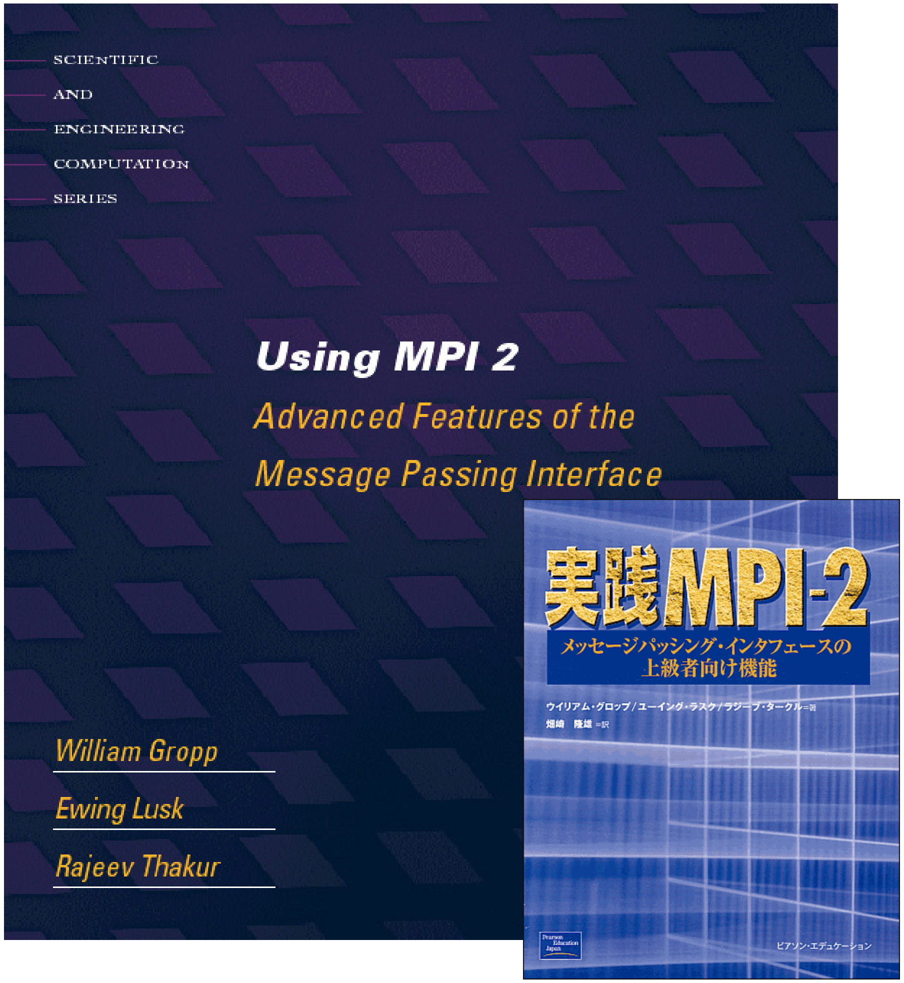


SCIENTIFIC
AND
ENGINEERING
COMPUTATION
SERIES

Using MPI

Portable Parallel Programming with the Message Passing Interface, Second Edition

William Gropp
Ewing Lusk
Anthony Skjellum



SCIENTIFIC
AND
ENGINEERING
COMPUTATION
SERIES

Using MPI 2

Advanced Features of the Message Passing Interface

William Gropp
Ewing Lusk
Rajeev Thakur

実践MPI-2

メッセージパッシング・インタフェースの
上級者向け機能

ウイリアム・グロップ／ユーイング・ラスク／ラジーブ・ターケル＝著
畑崎　隆雄＝訳

Pearson Education Japan

ピアソン・エデュケーション

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