



Argonne  
NATIONAL  
LABORATORY

*... for a brighter future*



U.S. Department  
of Energy

UChicago ►  
Argonne<sub>LLC</sub>



Office of  
Science  
U.S. DEPARTMENT OF ENERGY

A U.S. Department of Energy laboratory  
managed by UChicago Argonne, LLC

## *Advanced MPI*

*William Gropp,<sup>\*</sup> Rusty Lusk,<sup>†</sup> Rob Ross,<sup>†</sup> Rajeev Thakur<sup>†</sup>*

<sup>\*</sup>University of Illinois

<sup>†</sup>Argonne National Laboratory

## *Table of Contents*

- Introduction – 4
- Mesh Algorithms and Game of Life – 8
- Parallel I/O and Life – 22
- Portable Checkpointing – 43
- Independent I/O with MPI-IO – 52
- Tuning MPI-IO – 66
- MPI and Multicore – 103
- Exchanging Data with RMA – 127
- pNeo: Modeling the Human Brain – 150
- Passive Target RMA – 160
- Tuning RMA – 189
- Dynamic Process Management – 206
- Recent MPI Forum Efforts – 224
- Conclusions – 233

# Outline

## Morning

- Introduction
  - MPI-1 Status, MPI-2 Status
  - C++ and Fortran90
- Life, 1D Decomposition
  - point-to-point
  - checkpoint/restart
    - *Stdout, MPI-IO, PnetCDF*
- Independent and noncontig I/O
- Tuning MPI-IO
- MPI and Multicore

## Afternoon

- Exchanging data with RMA
  - Fence
  - Post/start/complete/wait
- pNeo application
- Passive Target RMA
- Tuning RMA
- Dynamic Process Management
  - Master/Slave example
- Recent MPI Forum Efforts
  - MPI 2.1, 2.2, 3.0
- Conclusions

# MPI-1

- MPI is a message-passing library interface standard.
  - Specification, not implementation
  - Library, not a language
  - Classical message-passing programming model
- MPI-1 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 and other environments (MPICH2, Open MPI)

## 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 Open MPI, 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)
  - Unpredictable communication (pNeo)
  - Passive target RMA (global arrays and MPI mutex)
  - Dynamic processes (parallel file copy)
- Show solutions that use the MPI-2 support for parallel I/O, RMA, and dynamic processes
  - Walk through actual code
- We assume familiarity with MPI-1

## *Regular Mesh Algorithms*

- Many scientific applications involve the solution of partial differential equations (PDEs)
- Many algorithms for approximating the solution of PDEs rely on forming a set of difference equations
  - Finite difference, finite elements, finite volume
- The exact form of the difference equations depends on the particular method
  - From the point of view of parallel programming for these algorithms, the operations are the same

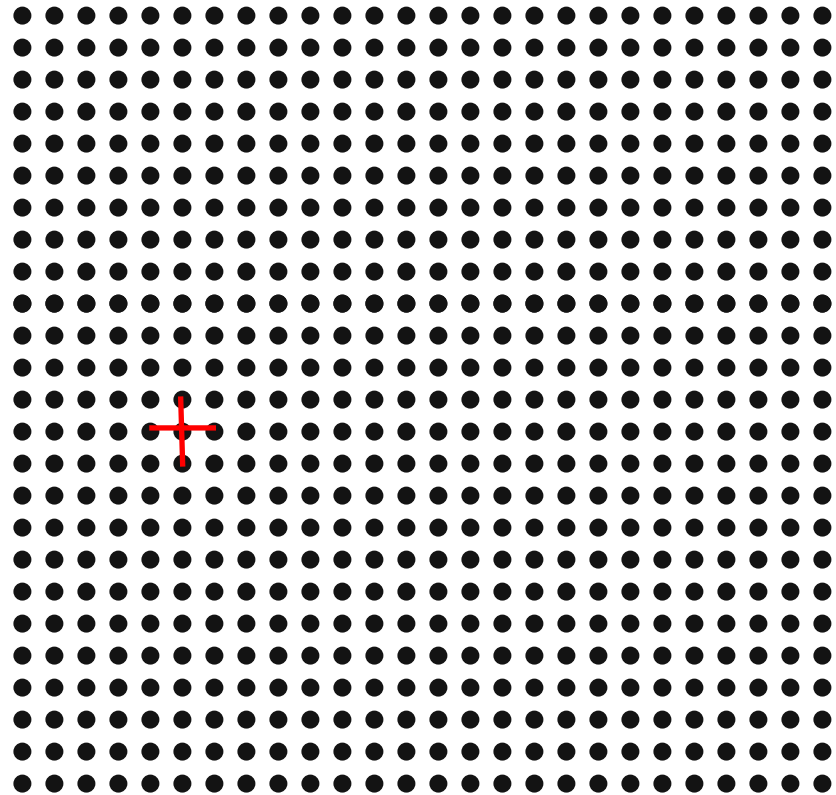


## Poisson Problem

- To approximate the solution of the Poisson Problem  $\nabla^2 u = f$  on the unit square, with  $u$  defined on the boundaries of the domain (Dirichlet boundary conditions), this simple 2nd order difference scheme is often used:
  - $(U(x+h,y) - 2U(x,y) + U(x-h,y)) / h^2 + (U(x,y+h) - 2U(x,y) + U(x,y-h)) / h^2 = f(x,y)$ 
    - *Where the solution  $U$  is approximated on a discrete grid of points  $x=0, h, 2h, 3h, \dots, (1/h)h=1, y=0, h, 2h, 3h, \dots 1$ .*
    - *To simplify the notation,  $U(ih,jh)$  is denoted  $U_{ij}$*
- This is defined on a discrete mesh of points  $(x,y) = (ih,jh)$ , for a mesh spacing “h”

## The Mesh

- Each circle is a mesh point
- Difference equation evaluated at each point involves the four neighbors
- The red “plus” is called the method’s *stencil*
- Good numerical algorithms form a matrix equation  $Au=f$ ; solving this requires computing  $Bv$ , where  $B$  is a matrix derived from  $A$ . These evaluations involve computations with the neighbors on the mesh.

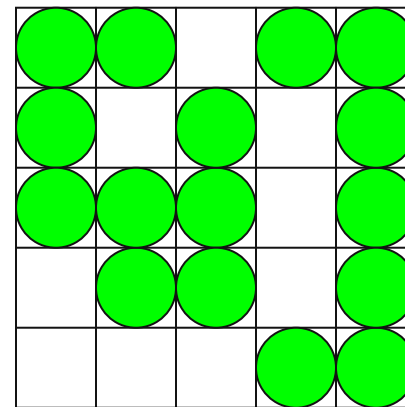
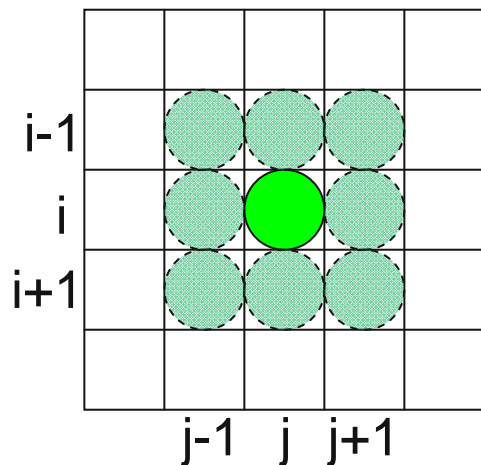


## Conway's Game of Life

- In this tutorial, we use Conway's Game of Life as a simple example to illustrate the program issues common to many codes that use regular meshes, such as PDE solvers
  - Allows us to concentrate on the MPI issues
- Game of Life is a cellular automaton
  - Described in 1970 Scientific American
  - Many interesting behaviors; see:
    - <http://www.ibiblio.org/lifepatterns/october1970.html>

## 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

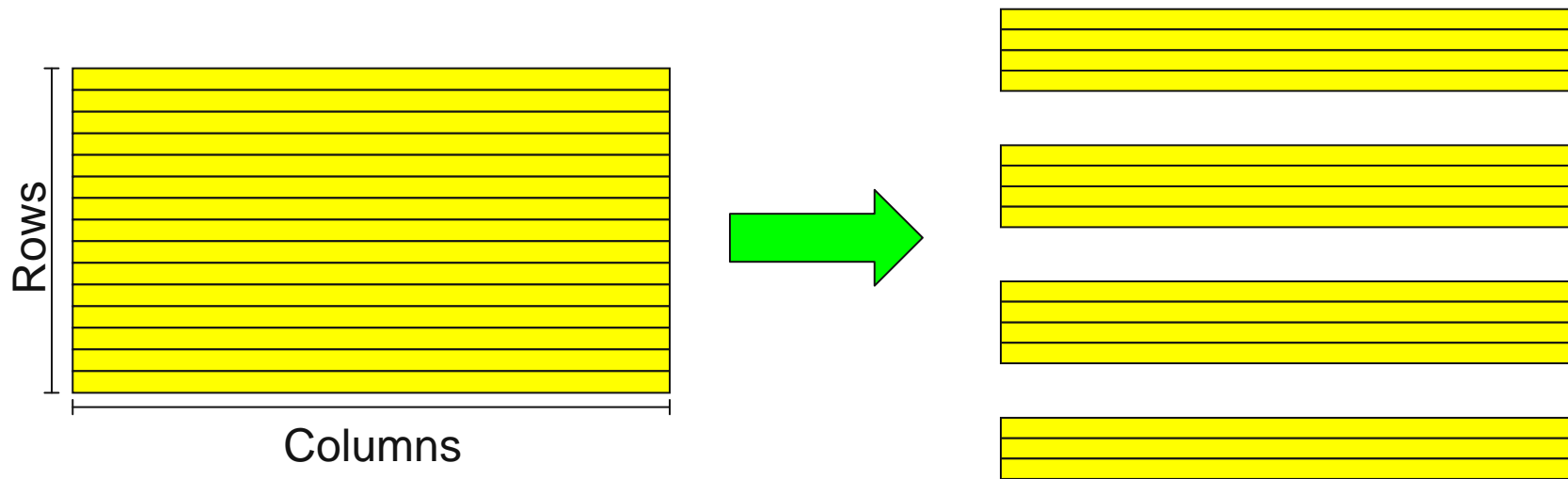
- For the non-parallel version, we:
  - Allocate a 2D matrix to hold state
    - *Actually two matrices, and we will swap them between steps*
  - Initialize the matrix
    - *Force boundaries to be “dead”*
    - *Randomly generate states inside*
  - At each time step:
    - *Calculate each new cell state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

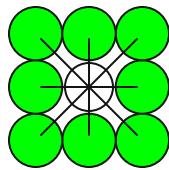
## Step 1: Description of Decomposition

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

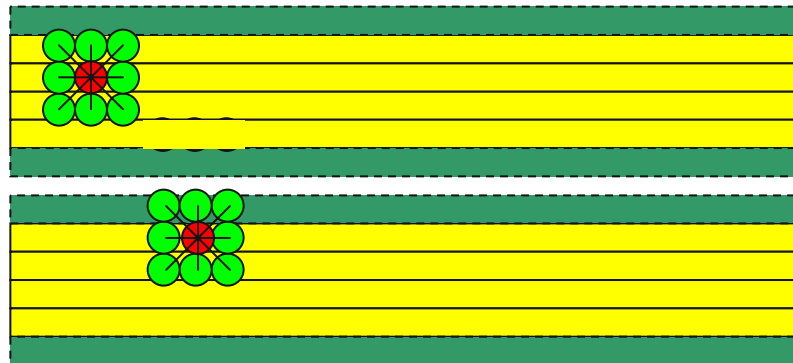


## Step 2: Communication

- “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



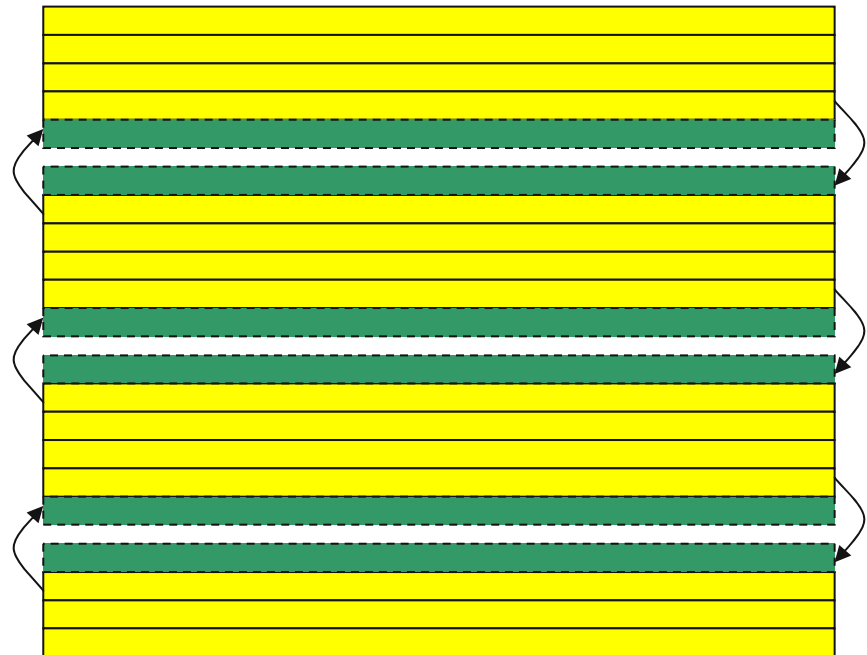


## Step 3: Define the Local Arrays

- 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
  - This is good practice when developing MPI codes, but is not required in this code
  - If this code were made into a component for use in other codes, the duplicate communicator would be required
- 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!)

## *Life stdout “checkpoint”*

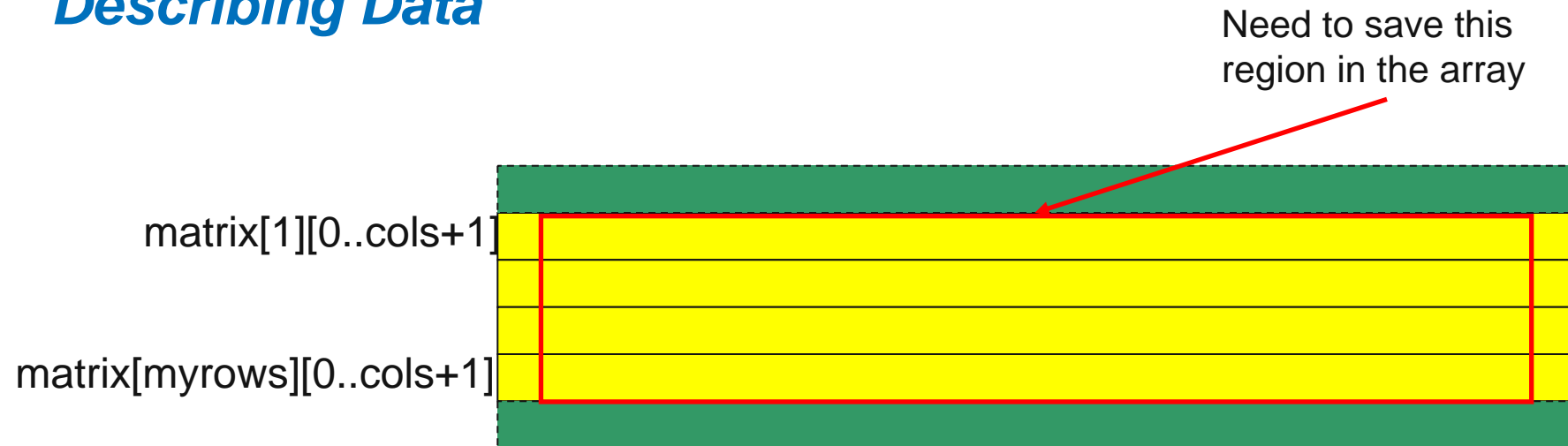
- The first implementation is one that simply prints out the “checkpoint” in an easy-to-read format
- MPI standard does not specify that all stdout will be collected in any particular way
  - Pass data back to rank 0 for printing
  - Portable!
  - Not scalable, but ok for the purpose of stdio

## *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. 1-3 for code example.

## Describing Data



- Lots of rows, all the same size

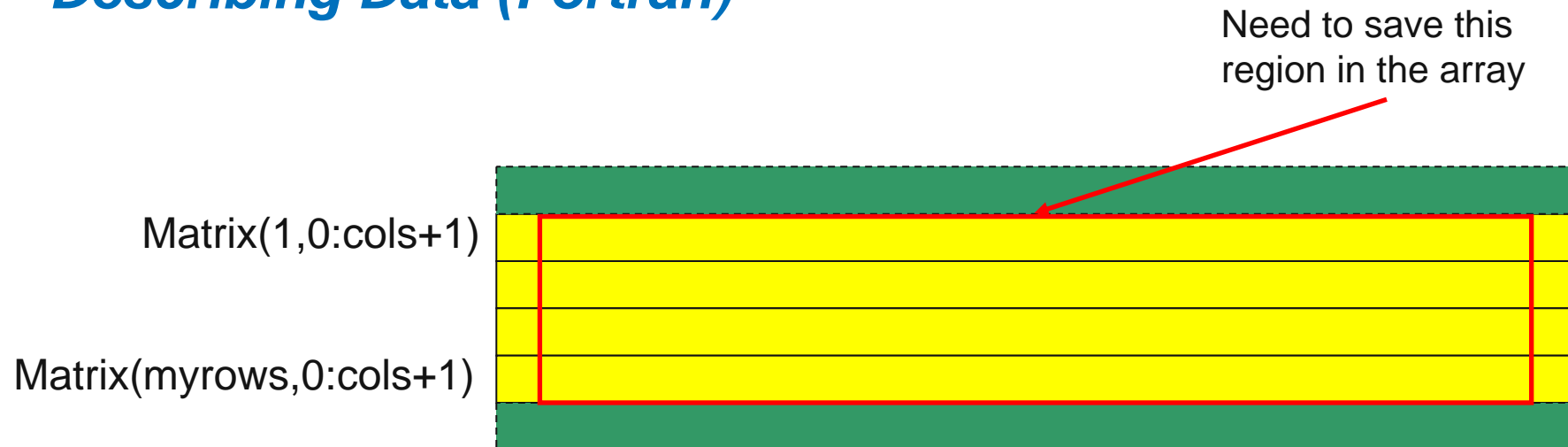
- Rows are all allocated as one big block
- Perfect for MPI\_Type\_vector

```
MPI_Type_vector(count = myrows,
                 blklen = cols, stride = cols+2, MPI_INT, &vectype);
```
- Second type gets memory offset right (allowing use of MPI\_BOTTOM in MPI\_File\_write\_all)

```
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*

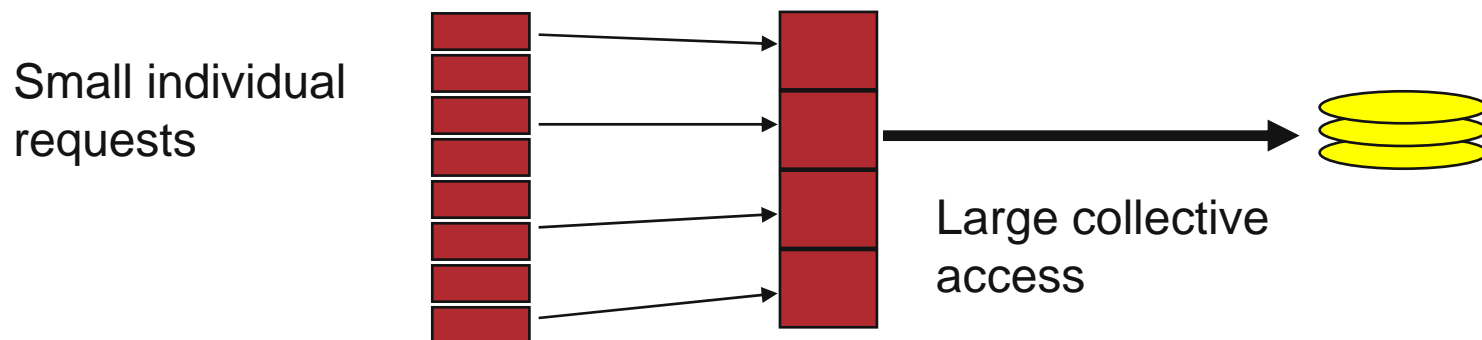
- Writing is like sending and reading is like receiving.
- Any parallel I/O system will need:
  - collective operations
  - user-defined datatypes to describe both memory and file layout
  - communicators to separate application-level message passing from I/O-related message passing
  - non-blocking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

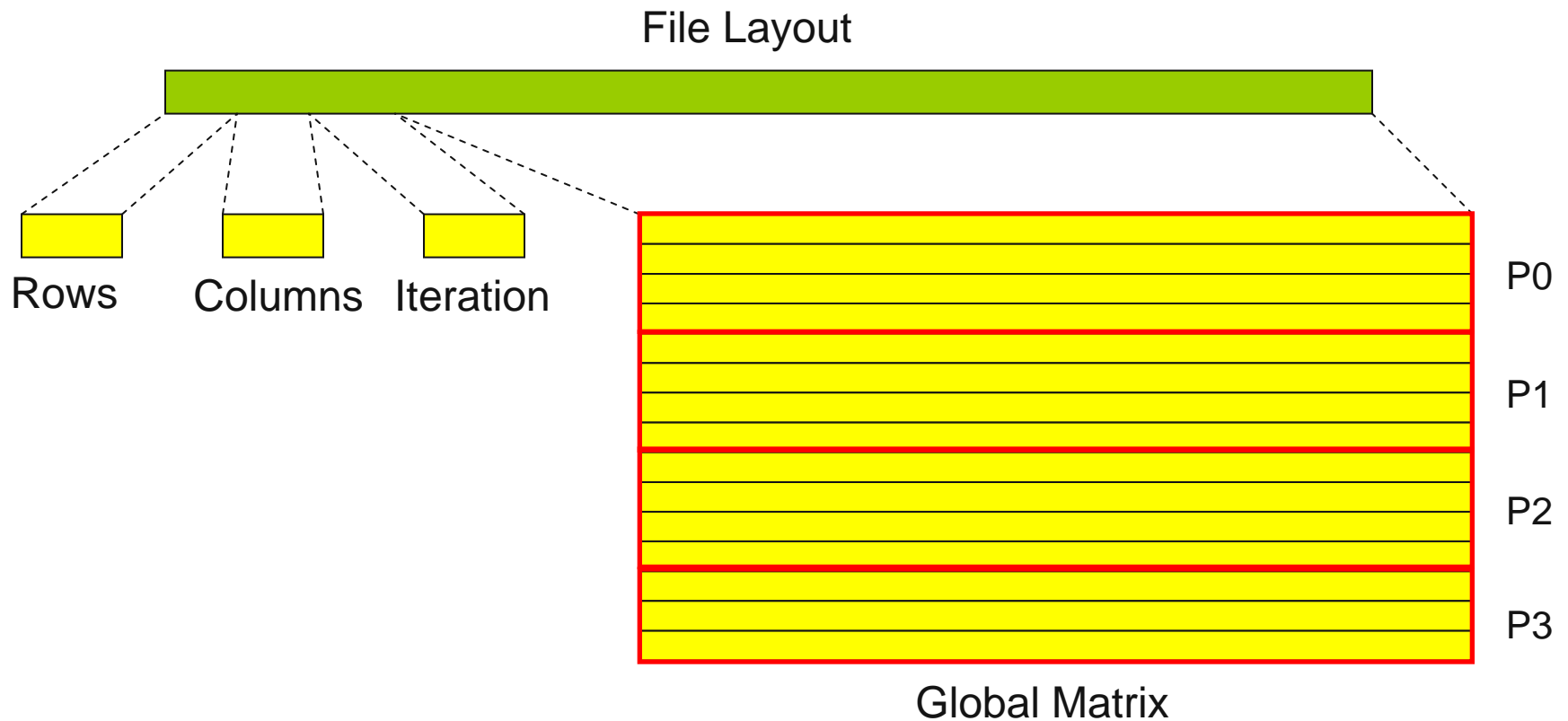
## *Describing Header and Data*

- 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
  - No one uses this
- 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*

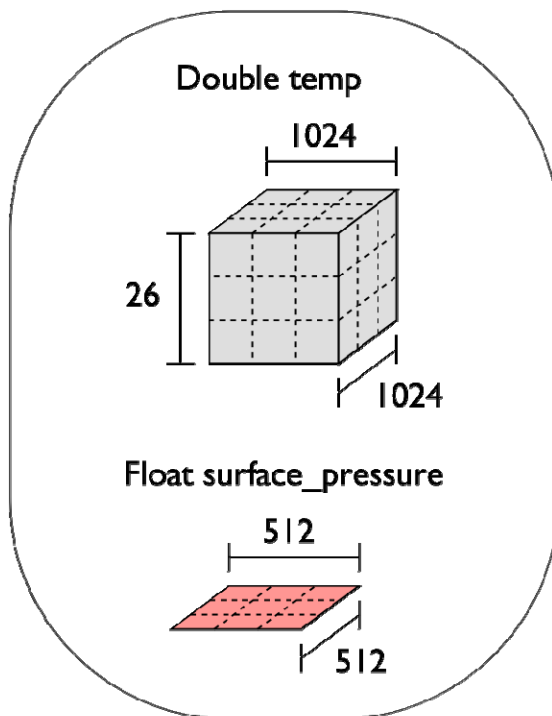
## Higher Level I/O Libraries

- 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://www.hdfgroup.org/HDF5/>

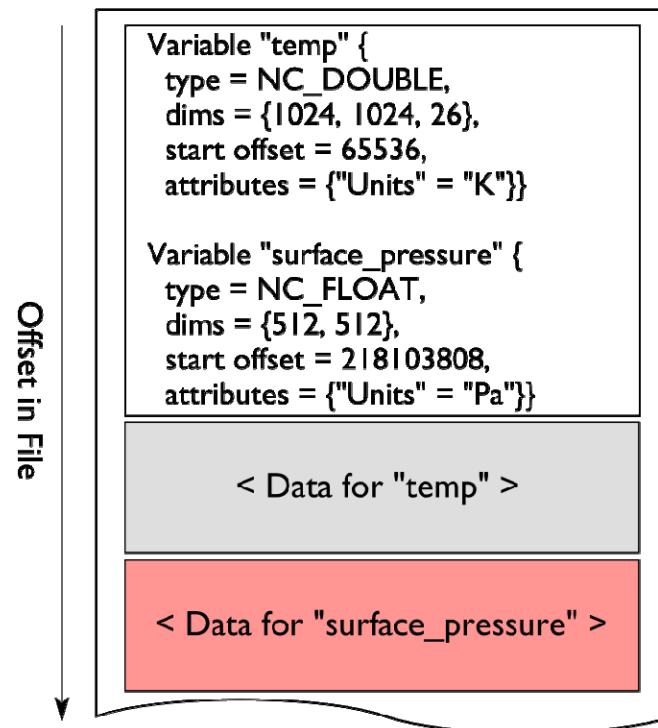
## netCDF Data Model

- netCDF provides a means for storing multiple, multi-dimensional arrays in a single file, along with information about the arrays

### Application Data Structures



### netCDF File "checkpoint07.nc"



netCDF header describes the contents of the file: typed, multi-dimensional variables and attributes on variables or the dataset itself.

Data for variables is stored in contiguous blocks, encoded in a portable binary format according to the variable's type.

## *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

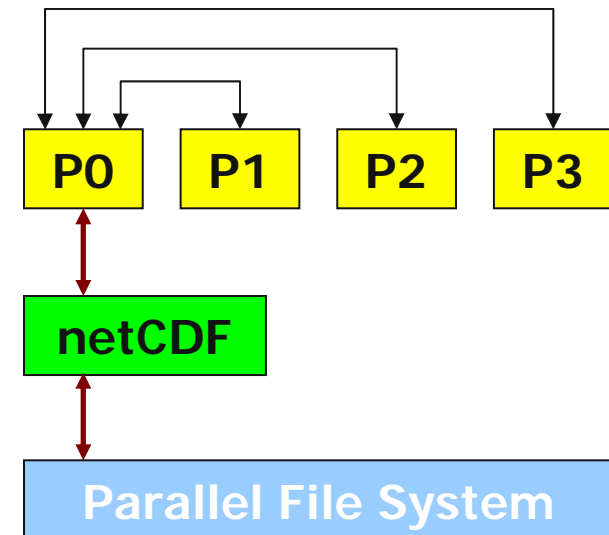
IBM MPI

GPFS

## I/O in netCDF and PnetCDF

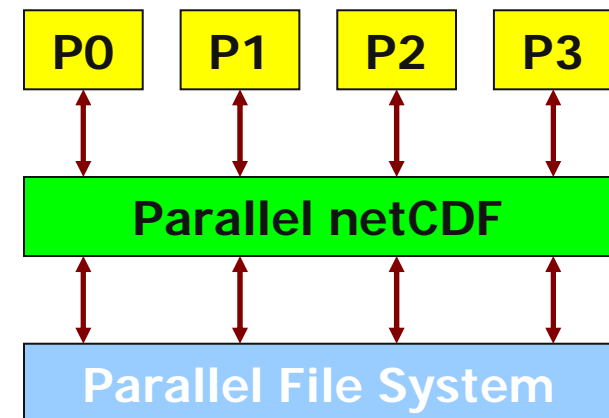
### ■ (Serial) netCDF

- 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. 1-5 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. 6-7 for code example.

## *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`
- `MPI_File_read`
- `MPI_File_write`
- `MPI_File_read_at`
- `MPI_File_write_at`
- `MPI_File_read_shared`
- `MPI_File_write_shared`

} like Unix I/O

} combine seek and I/O  
for thread safety

} use shared file pointer

## 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

## *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

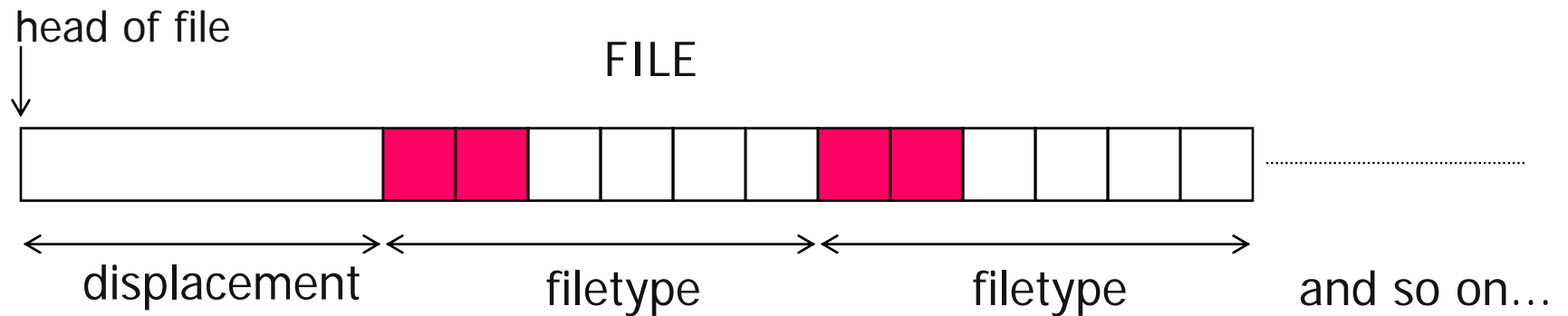
## File Views

- 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



## *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)
```

## *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)
```

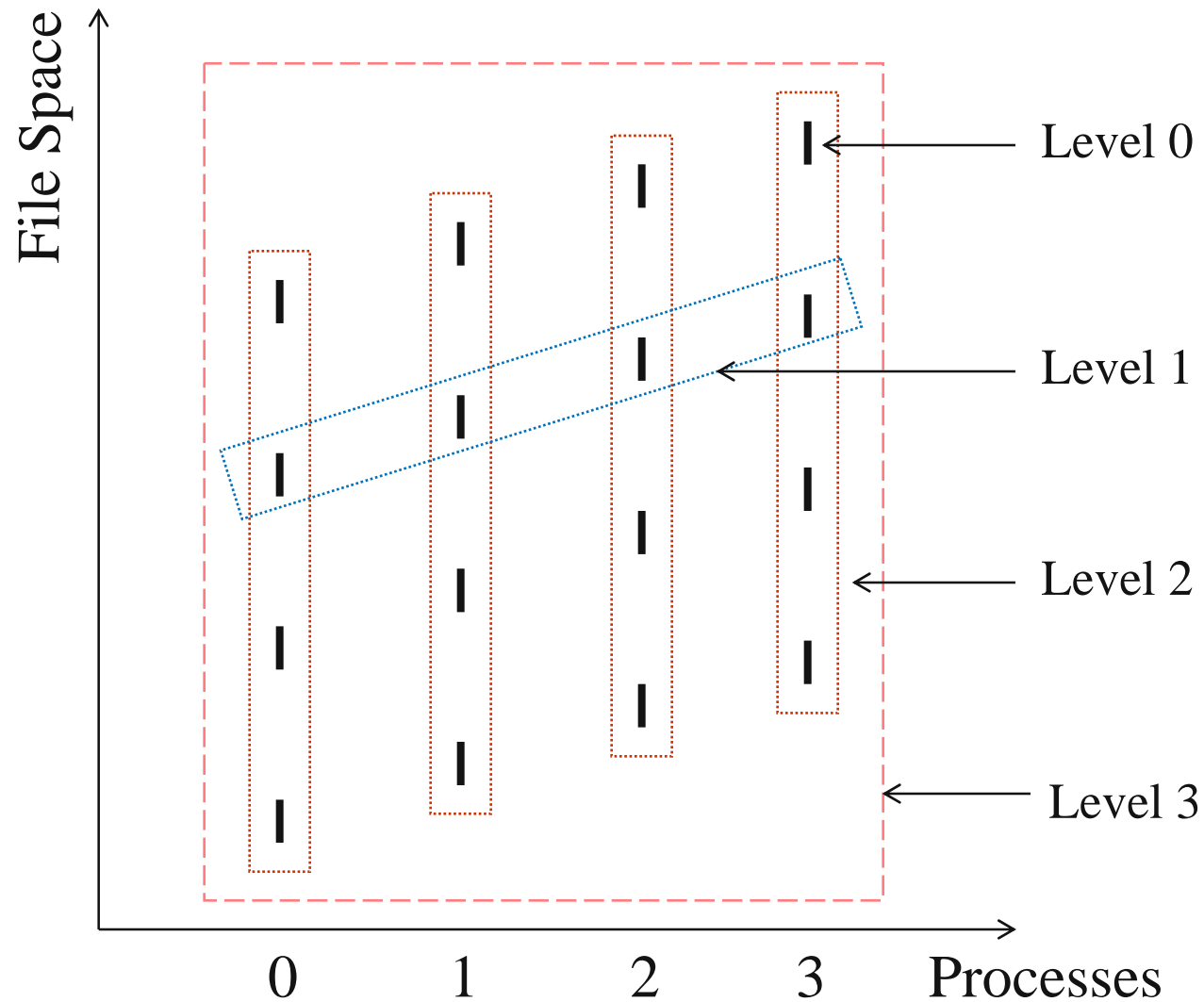


## Level-3 Access (Fortran)

- 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

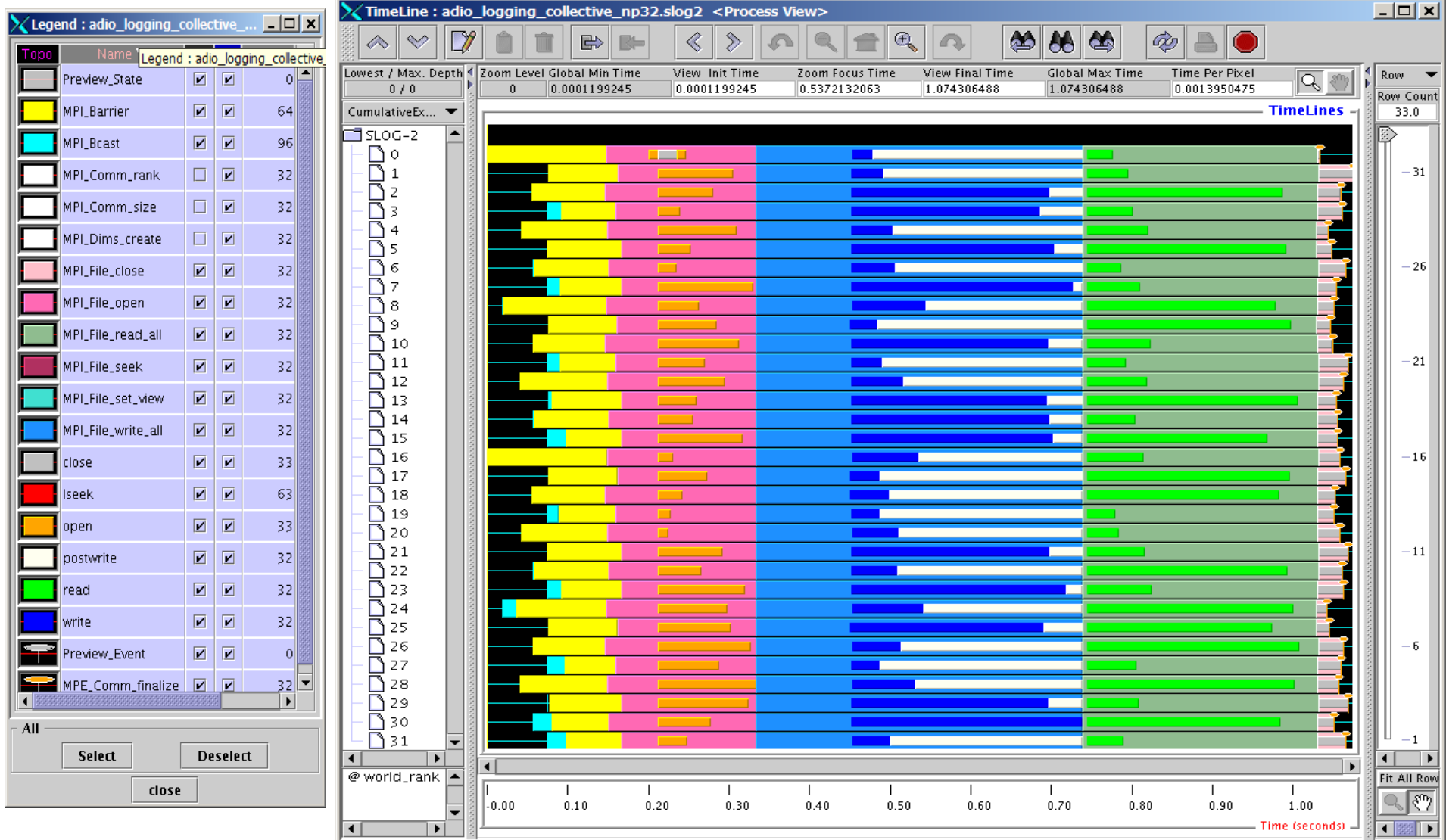
## *Performance Instrumentation*

- We instrumented the source code of our MPI-IO implementation (ROMIO) to log various events (using the MPE toolkit from MPICH2)
- We ran a simple 3D distributed array access code written in three ways:
  - POSIX (level 0)
  - data sieving (level 2)
  - collective I/O (level 3)
- The code was run on 32 nodes of the Jazz cluster at Argonne with PVFS-1 as the file system
- We collected the trace files and visualized them with Jumpshot

## Collective I/O

- The next slide shows the trace for the collective I/O case
- Note that the entire program runs for a little more than 1 sec
- Each process does its entire I/O with a single write or read operation
- Data is exchanged with other processes so that everyone gets what they need
- Very efficient!

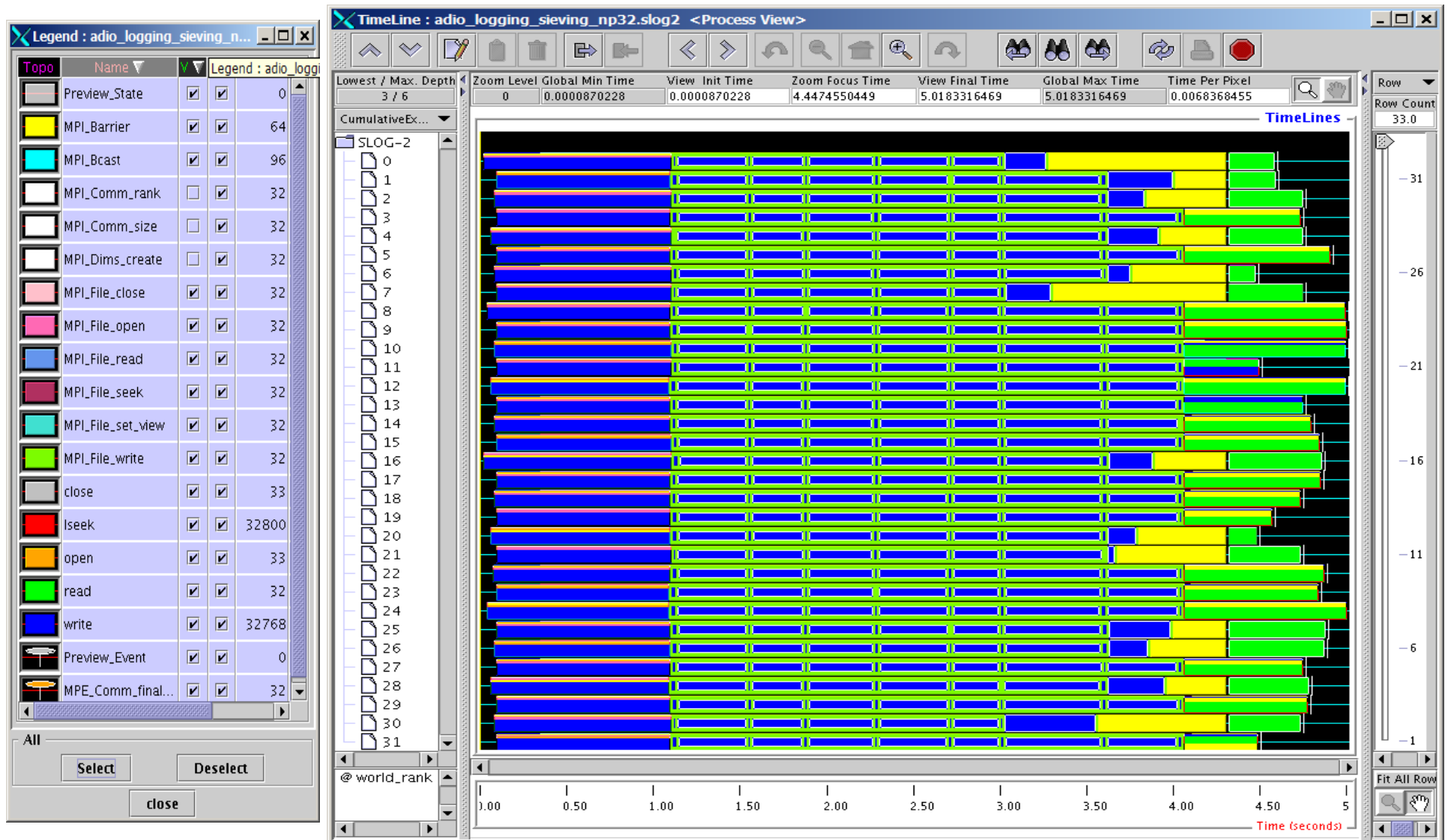
# Collective I/O



## *Data Sieving*

- The next slide shows the trace for the data sieving case
- Note that the program runs for about 5 sec now
- Since the default data sieving buffer size happens to be large enough, each process can read with a single read operation, although more data is read than actually needed (because of holes)
- Since PVFS doesn't support file locking, data sieving cannot be used for writes, resulting in many small writes (1K per process)

# Data Sieving

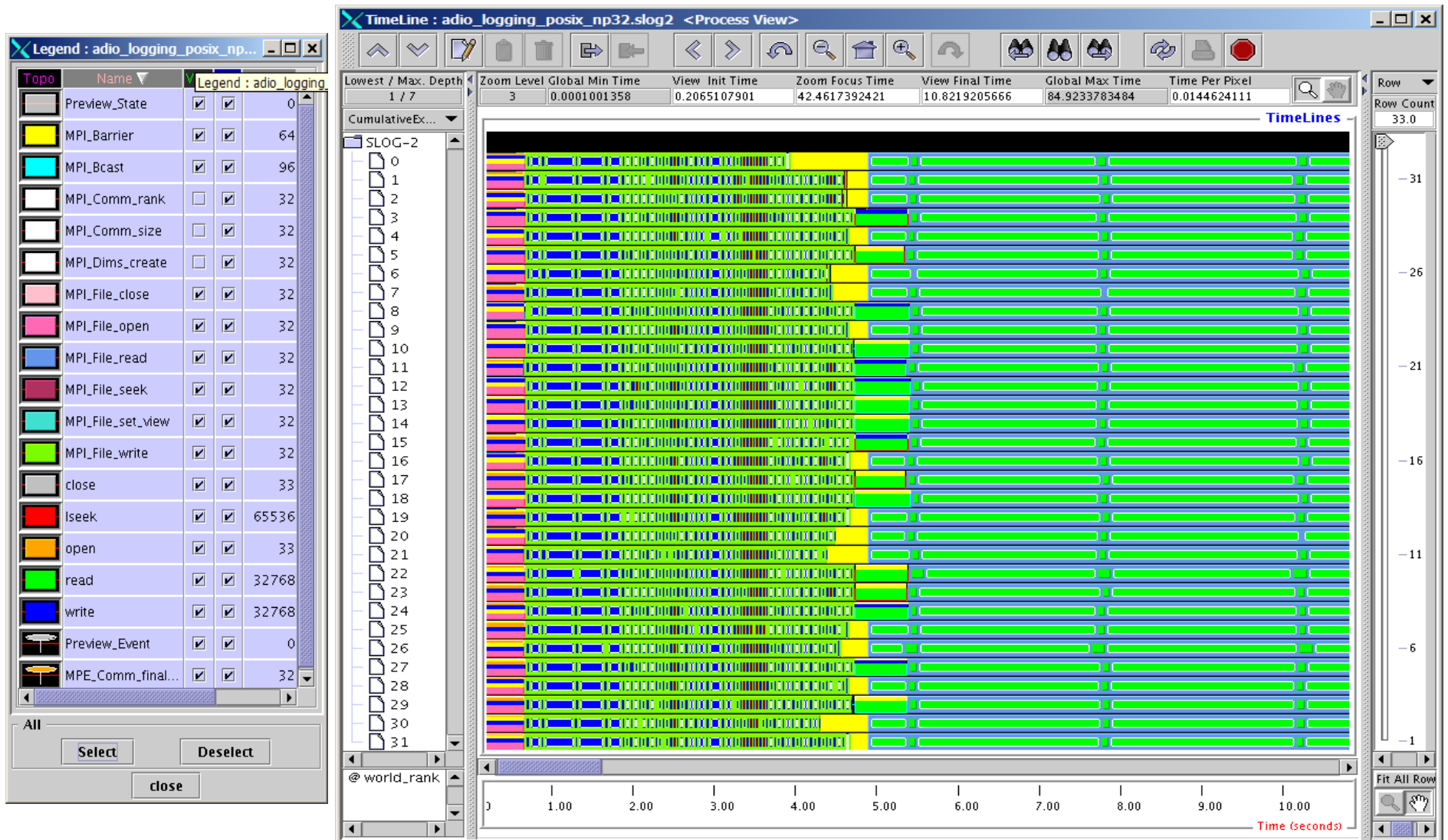




## Posix I/O

- The next slide shows the trace for Posix I/O
- Lots of small reads and writes (1K each per process)
- The reads take much longer than the writes in this case because of a TCP-incast problem happening in the switch
- Total program takes about 80 sec
- Very inefficient!

# Posix I/O

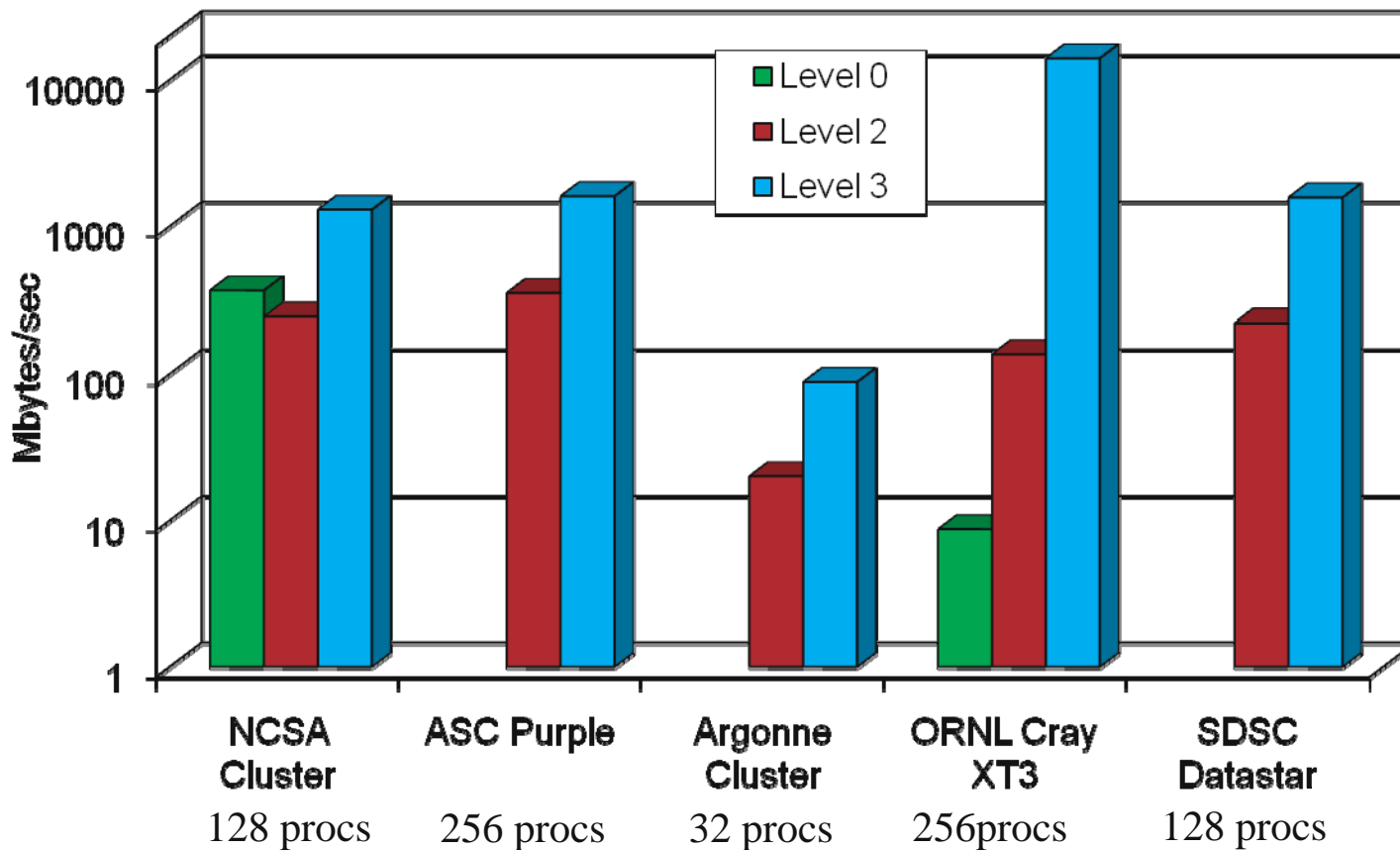


## Bandwidth Results

- 3D distributed array access written as levels 0, 2, 3
- Five different machines
  - NCSA Teragrid IA-64 cluster with GPFS and MPICH2
  - ASC Purple at LLNL with GPFS and IBM's MPI
  - Jazz cluster at Argonne with PVFS and MPICH2
  - Cray XT3 at ORNL with Lustre and Cray's MPI
  - SDSC Datastar with GPFS and IBM's MPI
- *Since these are all different machines with different amounts of I/O hardware, we compare the performance of the different levels of access on a particular machine, not across machines*

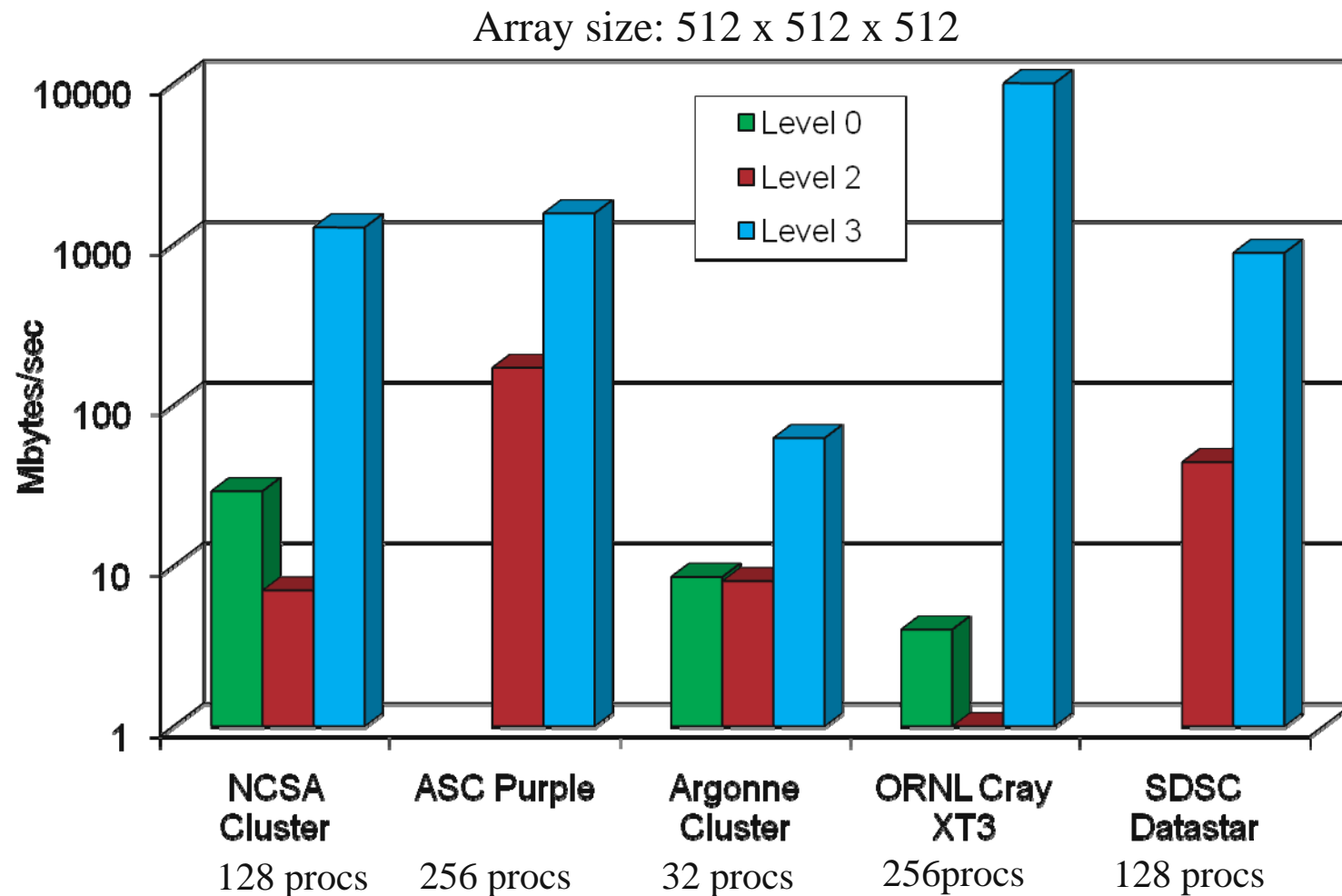
## Distributed Array Access: Read Bandwidth

Array size: 512 x 512 x 512



Thanks to Weikuan Yu, Wei-keng Liao, Bill Loewe, and Anthony Chan for these results.

## *Distributed Array Access: Write Bandwidth*



Thanks to Weikuan Yu, Wei-keng Liao, Bill Loewe, and Anthony Chan for these results.

## *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)*

■ <code>striping_unit</code>	}	MPI-2 predefined hints
■ <code>striping_factor</code>		
■ <code>cb_buffer_size</code>		
■ <code>cb_nodes</code>		
■ <code>ind_rd_buffer_size</code>	}	New Algorithm Parameters
■ <code>ind_wr_buffer_size</code>		
■ <code>start_iodevice</code>	}	Platform-specific hints
■ <code>pfs_svr_buf</code>		
■ <code>direct_read</code>		
■ <code>direct_write</code>		

## *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)

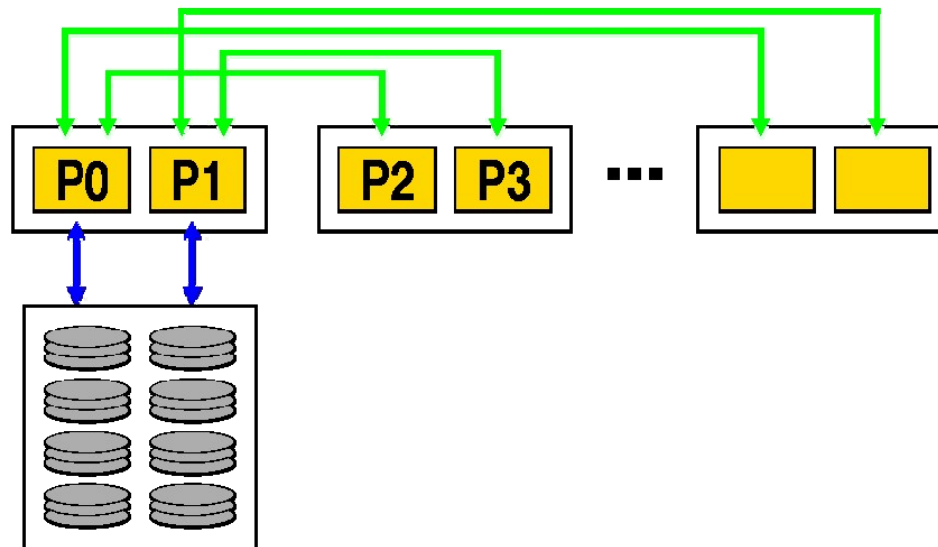
- Tuning ROMIO optimizations

  - `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)

## ***MPI and Multicore***

## MPI and Threads

- MPI describes parallelism between *processes* (with separate address spaces)
- *Thread* parallelism provides a shared-memory model within a process
- OpenMP and Pthreads are common models
  - OpenMP provides convenient features for loop-level parallelism. Threads are created and managed by the compiler, based on user directives.
  - Pthreads provide more complex and dynamic approaches. Threads are created and managed explicitly by the user.



## Programming for Multicore

- Almost all chips are multicore these days
- Today's clusters often comprise multiple CPUs per node sharing memory, and the nodes themselves are connected by a network
- Common options for programming such clusters
  - All MPI
    - *Use MPI to communicate between processes both within a node and across nodes*
    - *MPI implementation internally uses shared memory to communicate within a node*
  - MPI + OpenMP
    - *Use OpenMP within a node and MPI across nodes*
  - MPI + Pthreads
    - *Use Pthreads within a node and MPI across nodes*
- The latter two approaches are known as “hybrid programming”

## *MPI's Four Levels of Thread Safety*

- MPI defines four levels of thread safety. These are in the form of commitments the application makes to the MPI implementation.
  - MPI\_THREAD\_SINGLE: only one thread exists in the application
  - MPI\_THREAD\_FUNNELED: multithreaded, but only the main thread makes MPI calls (the one that called MPI\_Init or MPI\_Init\_thread)
  - MPI\_THREAD\_SERIALIZED: multithreaded, but only one thread *at a time* makes MPI calls
  - MPI\_THREAD\_MULTIPLE: multithreaded and any thread can make MPI calls at any time (with some restrictions to avoid races – see next slide)
- MPI defines an alternative to MPI\_Init
  - MPI\_Init\_thread(requested, provided)
    - Application indicates what level it needs; MPI implementation returns the level it supports

## *Specification of MPI\_THREAD\_MULTIPLE*

- When multiple threads make MPI calls concurrently, the outcome will be as if the calls executed sequentially in some (any) order
- Blocking MPI calls will block only the calling thread and will not prevent other threads from running or executing MPI functions
- It is the user's responsibility to prevent races when threads in the same application post conflicting MPI calls
  - e.g., accessing an info object from one thread and freeing it from another thread
- User must ensure that collective operations on the same communicator, window, or file handle are correctly ordered among threads
  - e.g., cannot call a broadcast on one thread and a reduce on another thread on the same communicator

## *Threads and MPI in MPI-2*

- An implementation is not required to support levels higher than `MPI_THREAD_SINGLE`; that is, an implementation is not required to be thread safe
- A fully thread-safe implementation will support `MPI_THREAD_MULTIPLE`
- A program that calls `MPI_Init` (instead of `MPI_Init_thread`) should assume that only `MPI_THREAD_SINGLE` is supported

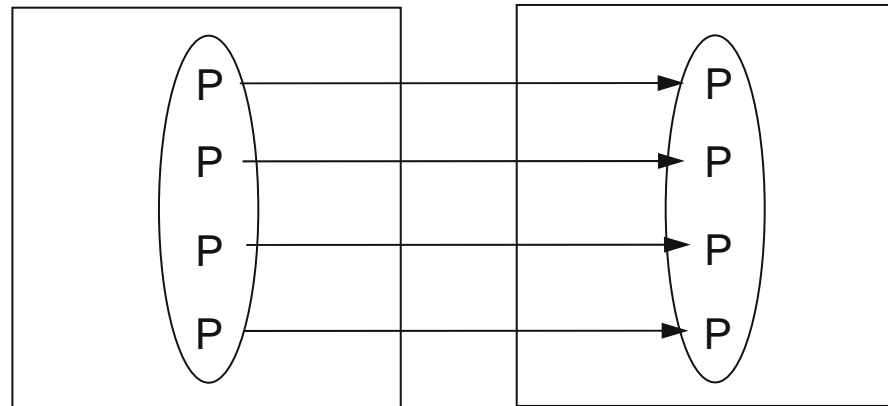
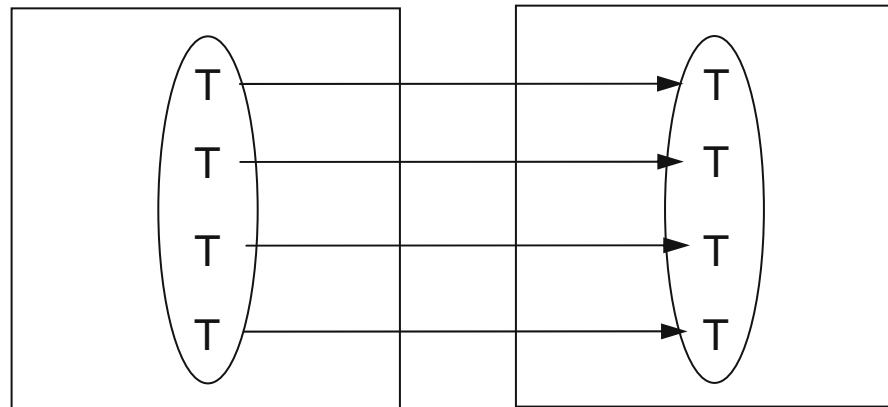
## *The Current Situation*

- All MPI implementations support MPI\_THREAD\_SINGLE (duh).
- They probably support MPI\_THREAD\_FUNNELED even if they don't admit it.
  - Does require thread-safe malloc
  - Probably OK in OpenMP programs
- Many (but not all) implementations support THREAD\_MULTIPLE
  - Hard to implement efficiently though (lock granularity issue)
- “Easy” OpenMP programs (loops parallelized with OpenMP, communication in between loops) only need FUNNELED
  - So don't need “thread-safe” MPI for many hybrid programs
  - But watch out for Amdahl's Law!

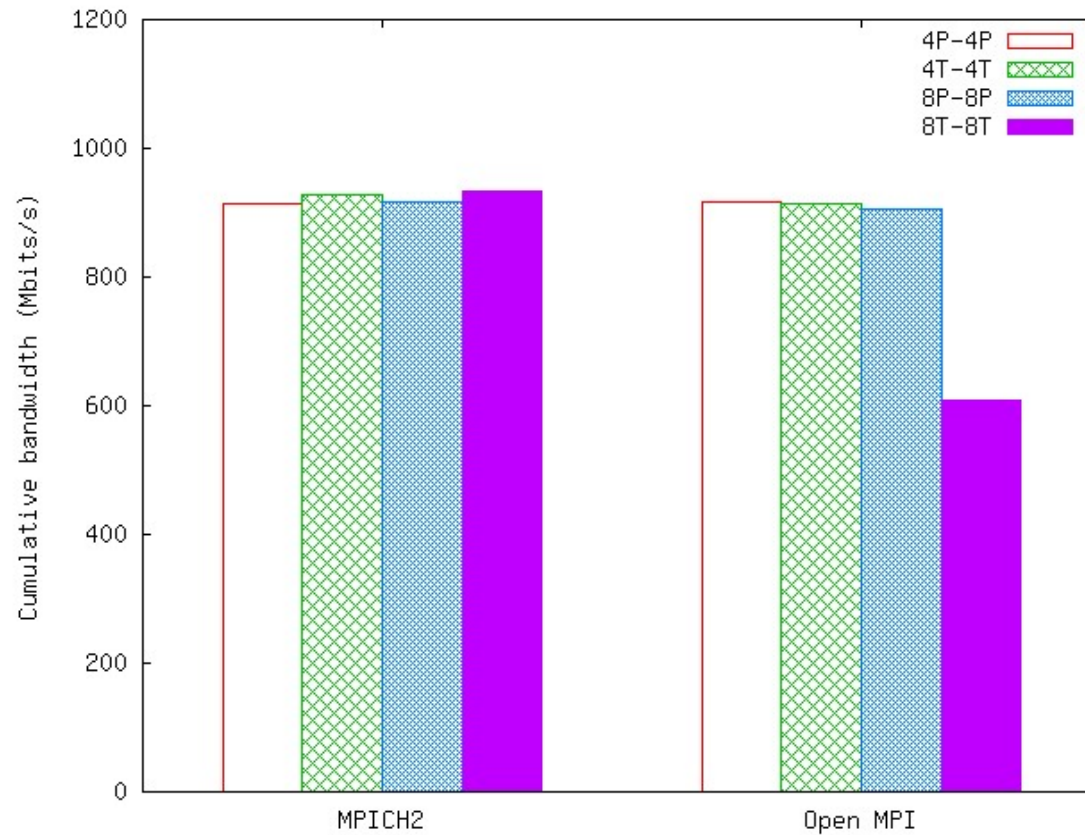
## *Performance with MPI\_THREAD\_MULTIPLE*

- Thread safety does not come for free
- The implementation must protect certain data structures or parts of code with mutexes or critical sections
- To measure the performance impact, we ran tests to measure communication performance when using multiple threads versus multiple processes
  - Details in our Euro PVM/MPI 2007 paper

## *Tests with Multiple Threads versus Processes*



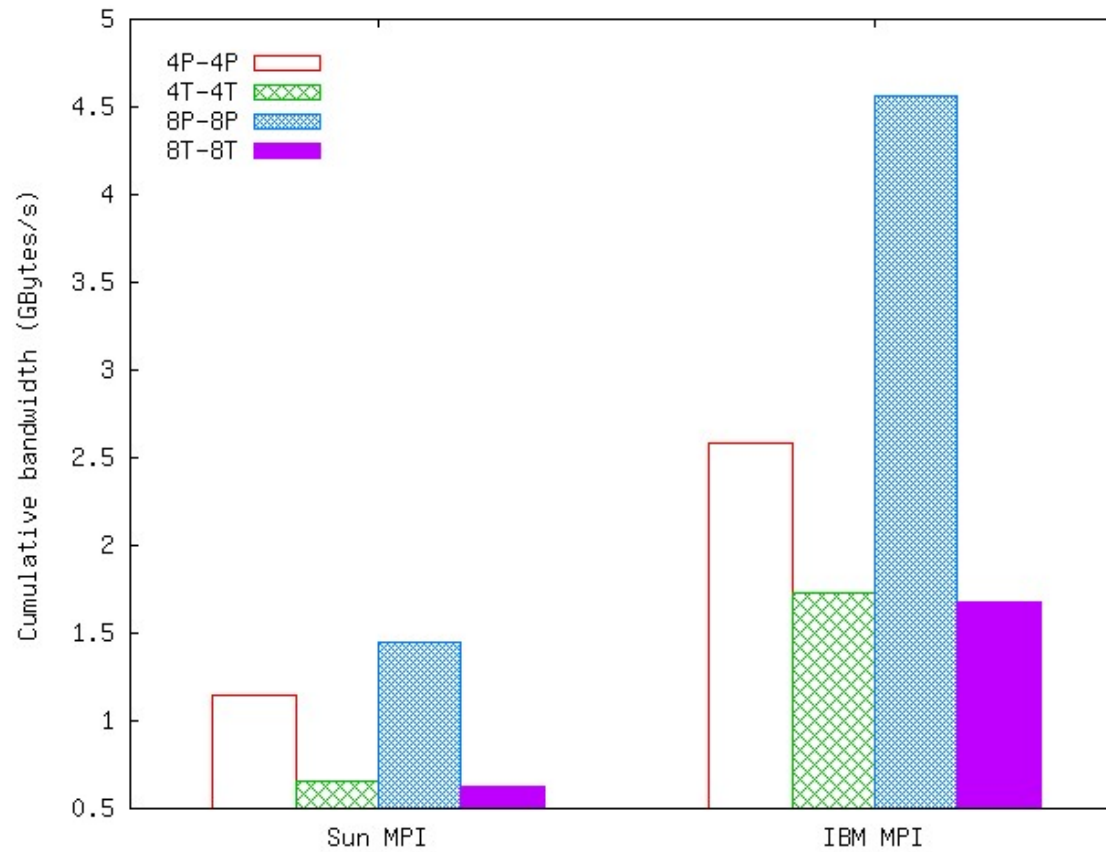
## Concurrent Bandwidth Test on Linux Cluster



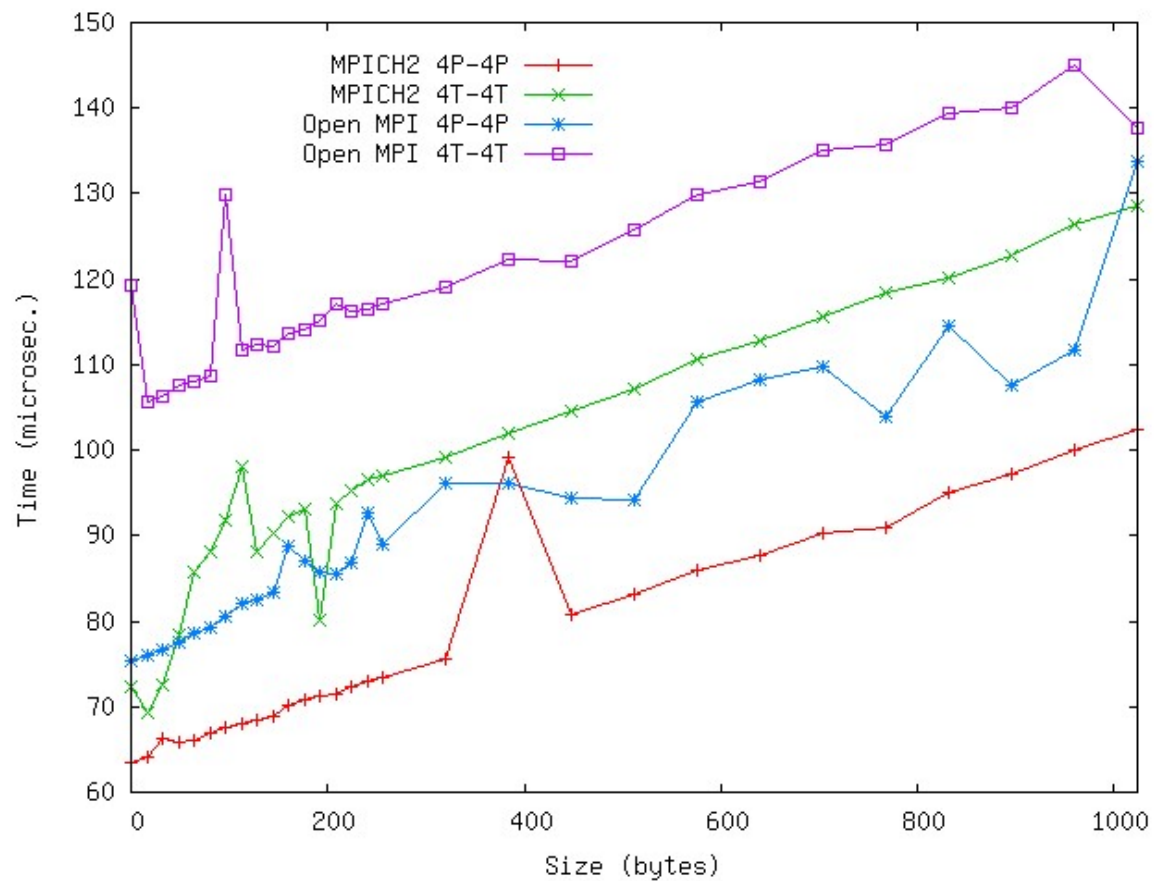
MPICH2 version 1.0.5  
Open MPI version 1.2.1



## Concurrent Bandwidth Test on a single SMP (Sun and IBM)

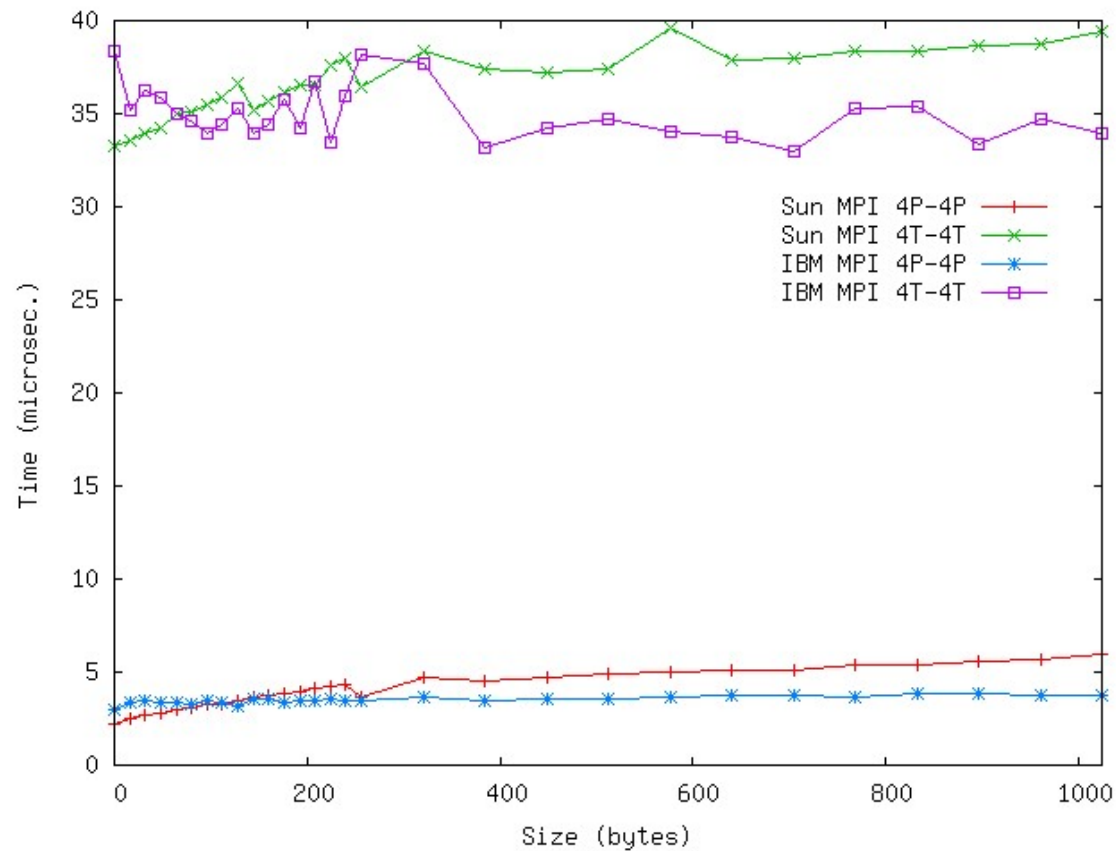


## Concurrent Latency Test on Linux Cluster



MPICH2 version 1.0.5  
Open MPI version 1.2.1

## Concurrent Latency Test on a single SMP (Sun and IBM)



## *What MPI's Thread Safety Means in the Hybrid MPI+OpenMP Context*

### ■ MPI\_THREAD\_SINGLE

- There is no OpenMP multithreading in the program.

### ■ MPI\_THREAD\_FUNNELED

- All of the MPI calls are made by the master thread. i.e. all MPI calls are
  - Outside OpenMP parallel regions, or
  - Inside OpenMP master regions, or
  - Guarded by call to MPI\_Is\_thread\_main MPI call.
    - (same thread that called MPI\_Init\_thread)

### ■ MPI\_THREAD\_SERIALIZED

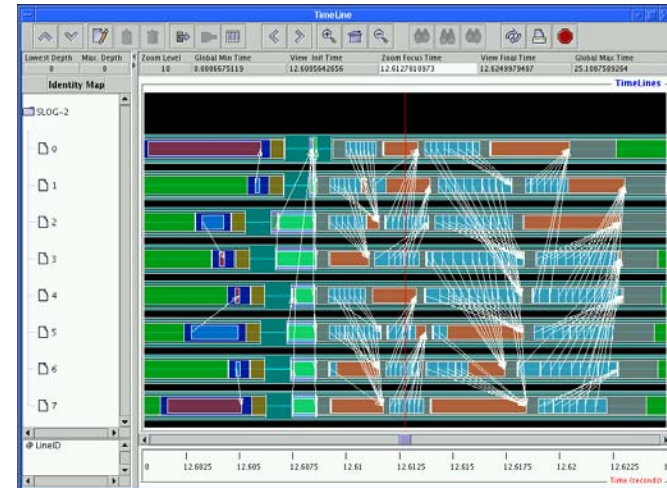
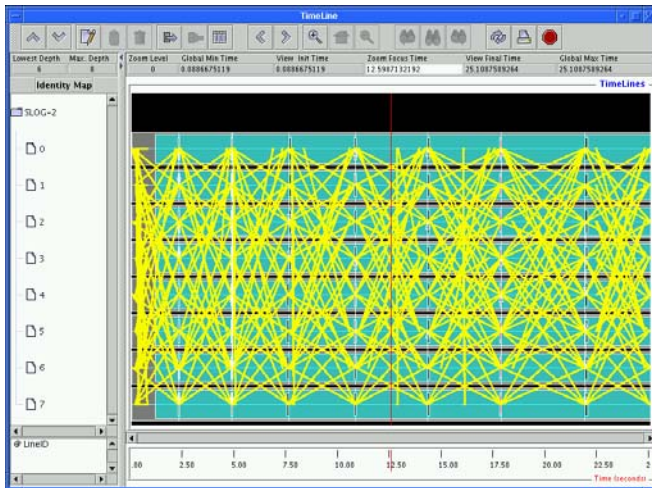
```
#pragma omp parallel
...
#pragma omp atomic
{
    ...MPI calls allowed here...
}
```

### ■ MPI\_THREAD\_MULTIPLE

- Any thread may make an MPI call at any time

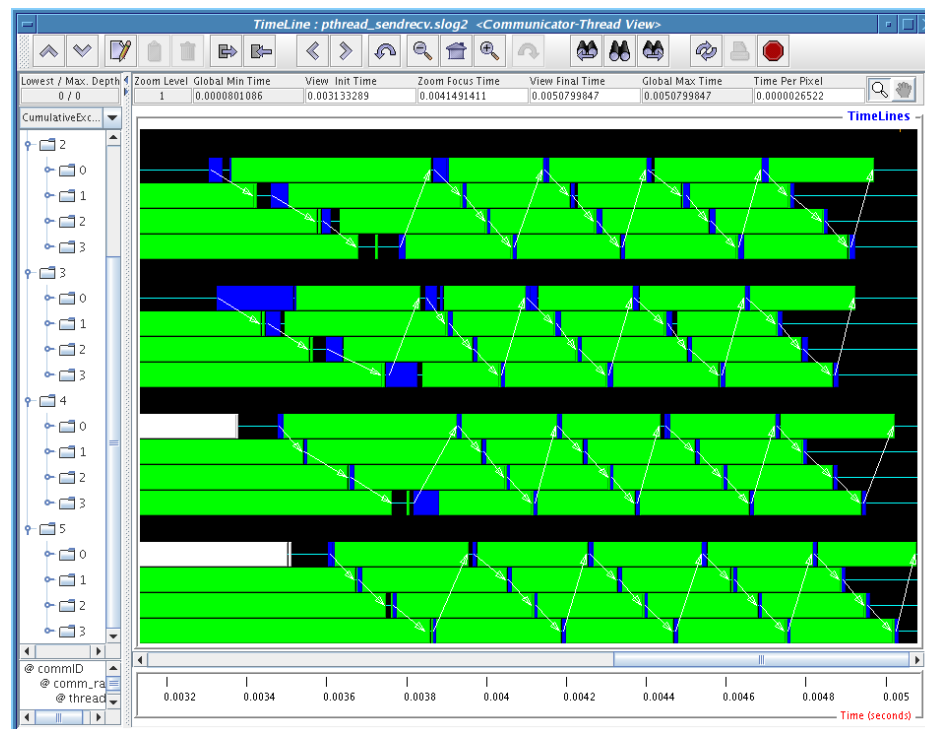
## Visualizing the Behavior of Hybrid Programs

- Jumpshot is a logfile-based parallel program visualizer. Uses MPI profiling interface.
- Recently it has been augmented in two ways to improve scalability.
  - Summary states and messages are shown as well as individual states and messages.
    - Provides a high-level view of a long run.
    - SLOG2 logfile structure allows fast interactive access (jumping, scrolling, and zooming) for large logfiles.



## Jumpshot and Multithreading

- Newest additions are for multithreaded and hybrid programs that use Pthreads
  - Separate timelines for each thread id
  - Support for grouping threads by communicator as well as by process



## *Using Jumpshot with Hybrid MPI+OpenMP Programs*

- SLOG2/Jumpshot needs two properties of the OpenMP implementation that are not guaranteed by the OpenMP standard
  - OpenMP threads must be Pthreads
    - Otherwise, the locking in the logging library (which uses Pthread locks) necessary to preserve exclusive access to the logging buffers would need to be modified
  - These Pthread ids must be reused (threads are “parked” when not in use)
    - Otherwise Jumpshot would need zillions of time lines

## *Three Platforms for Hybrid Programming Experiments*

### ■ Linux cluster

- 24 nodes, each with two Opteron dual-core processors, 2.8 Ghz each
- Intel 9.1 Fortran compiler
- MPICH2-1.0.6, which has MPI\_THREAD\_MULTIPLE
- Multiple networks; we used GigE

### ■ IBM Blue Gene/P

- 40,960 nodes, each consisting of four PowerPC 850 MHz cores
- XLF 11.1 Fortran cross-compiler
- IBM's MPI V1R1M2 (based on MPICH2), has MPI\_THREAD\_MULTIPLE
- 3D Torus and tree networks

### ■ SiCortex SC5832

- 972 nodes, each consisting of six MIPS 500 MHz cores
- Pathscale 3.0.99 Fortran cross-compiler
- SiCortex MPI implementation based on MPICH2, has MPI\_THREAD\_FUNNELED
- Kautz graph network



# Experiments

## ■ Basic

- Proved that necessary assumptions for our tools hold
  - OpenMP threads are Pthreads
  - Thread id's are reused

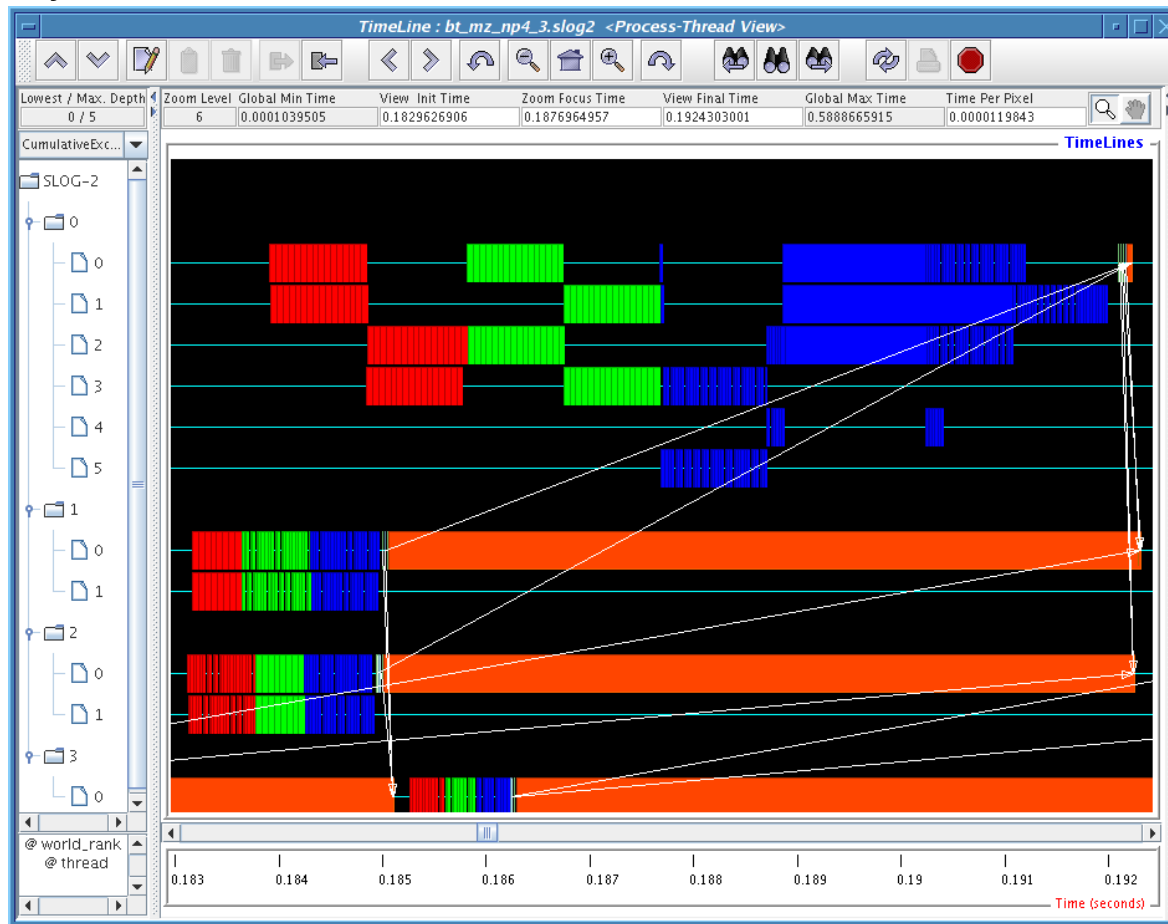
## ■ NAS Parallel Benchmarks

- NPB-MZ-MPI, version 3.1
- Both BT and SP
- Two different sizes (W and B)
- Two different modes (“MPI everywhere” and OpenMP/MPI)
  - With four nodes on each machine

- Demonstrated satisfying level of portability of programs and tools across three quite different hardware/software environments

# *It Might Not Be Doing What You Think*

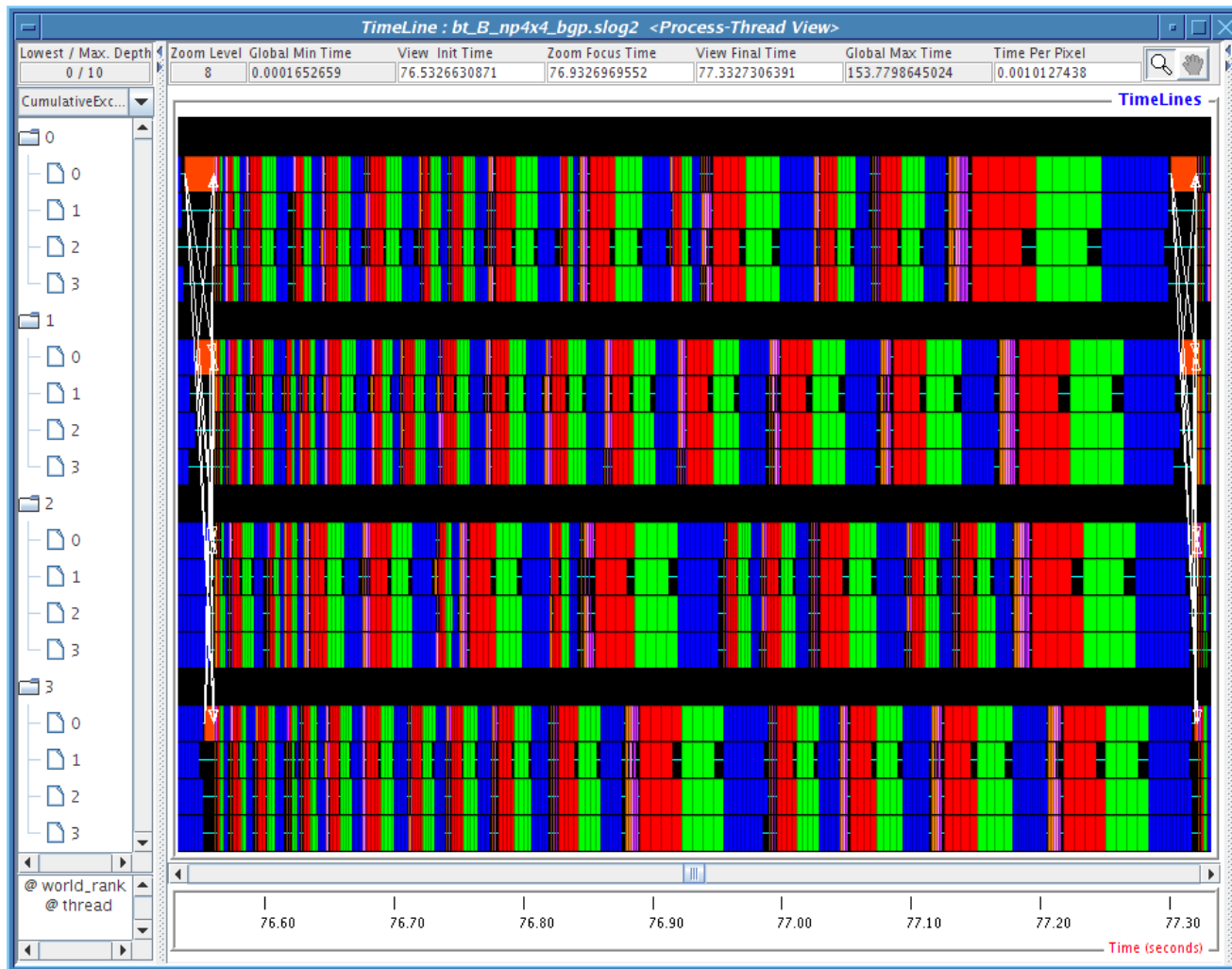
- An early run:



- Nasty interaction between the environment variables OMP\_NUM\_THREADS and NPB\_MAX\_THREADS

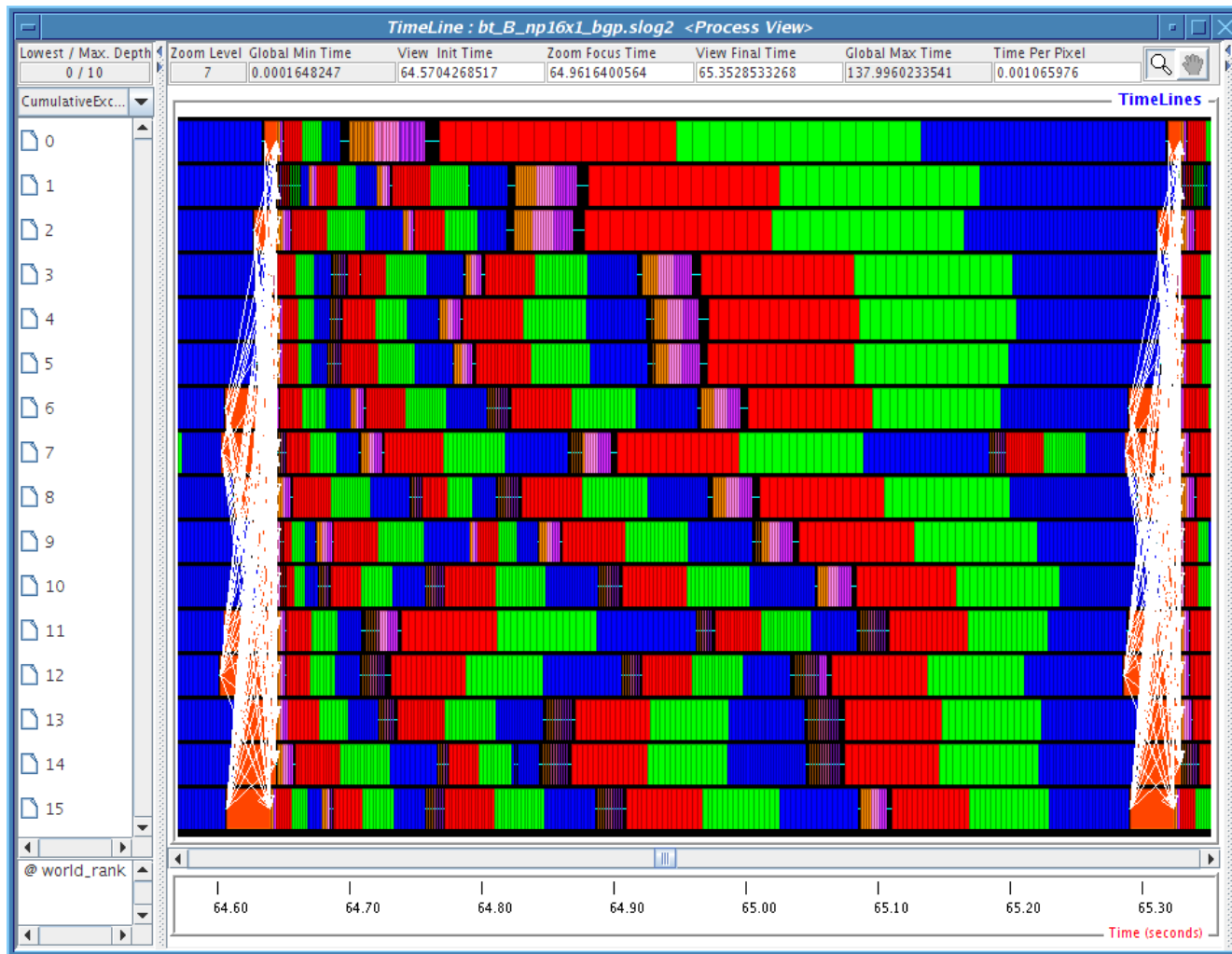
## More Like What You Expect

- BT class B on 4 BG/P nodes, using OpenMP on each node



## MPI Everywhere

- BT class B on 4 BG/P nodes, using 16 MPI processes



## Observations on Experiments

Experiment	Cluster	BG/P	SiCortex
Bt-mz.W.16x1	1.84	9.46	20.60
Bt-mz-W.4x4	0.82	3.74	11.26
Sp-mz.W.16x1	0.42	1.79	3.72
Sp-mz.W.4x4	0.78	3.00	7.98
Bt-mz.B.16.1	24.87	113.31	257.67
Bt-mz.B.4x4	27.96	124.60	399.23
Sp-mz.B.16x1	21.19	70.69	165.82
Sp-mz.B.4x4	24.03	81.47	246.76
Bt-mz.B.24x1			241.85
Bt-mz.B.4x6			337.86
Sp-mz.B.24x1			127.28
Sp-mz.B.4x6			211.78

- Time in seconds
- On the small version of BT (W), hybrid was better
- For SP and size B problems, MPI everywhere is better
- On SiCortex, more processes or threads are better than fewer

## Observations

- This particular benchmark has been studied much more deeply elsewhere
  - Rolf Rabenseifner, “Hybrid parallel programming on HPC platforms,” *Proceedings of EWOMP’03*.
  - Barbara Chapman, Gabriele Jost, and Ruud van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*, MIT Press, 2008.
- Adding “hybridness” to a well-tuned MPI application is not going to speed it up. So this NPB study doesn’t tell us much.
- More work is needed to understand the behavior of hybrid programs and what is needed for future application development.

## *Exchanging Data with RMA*

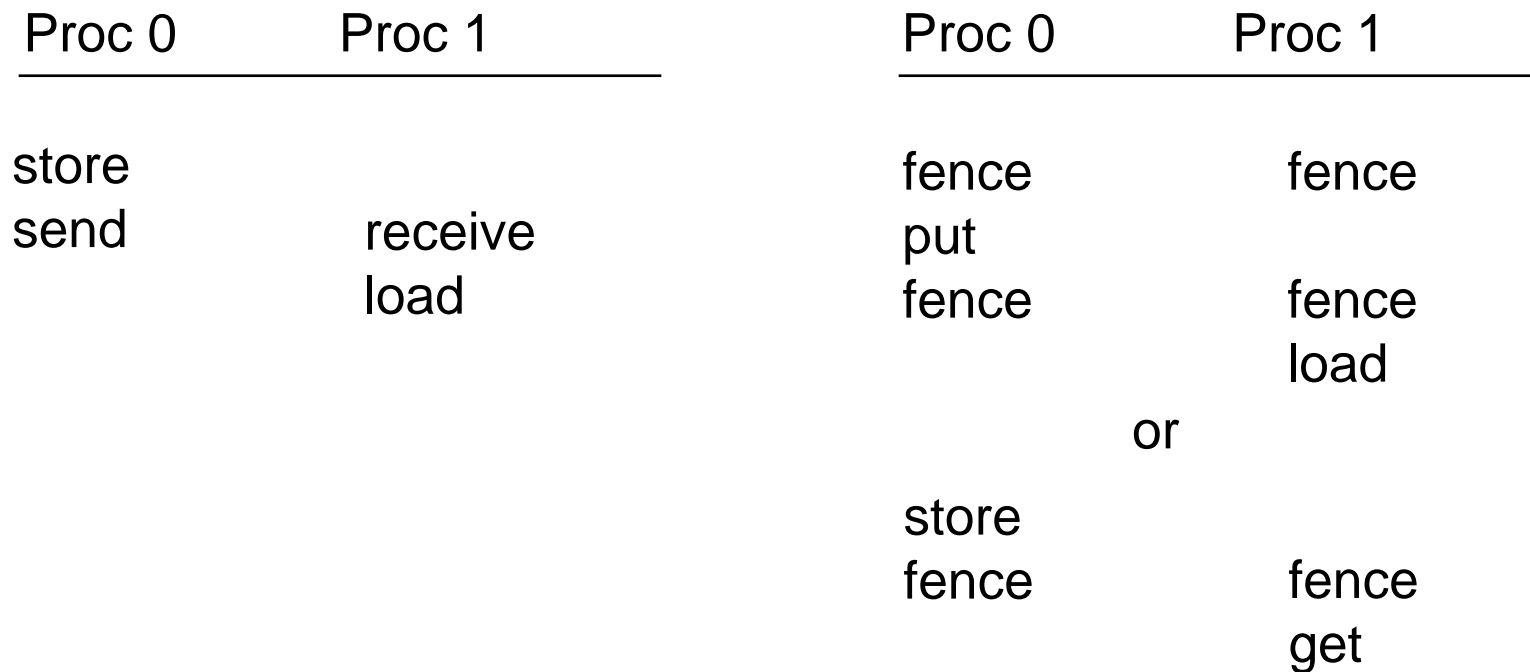
## Revisiting Mesh Communication

- 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

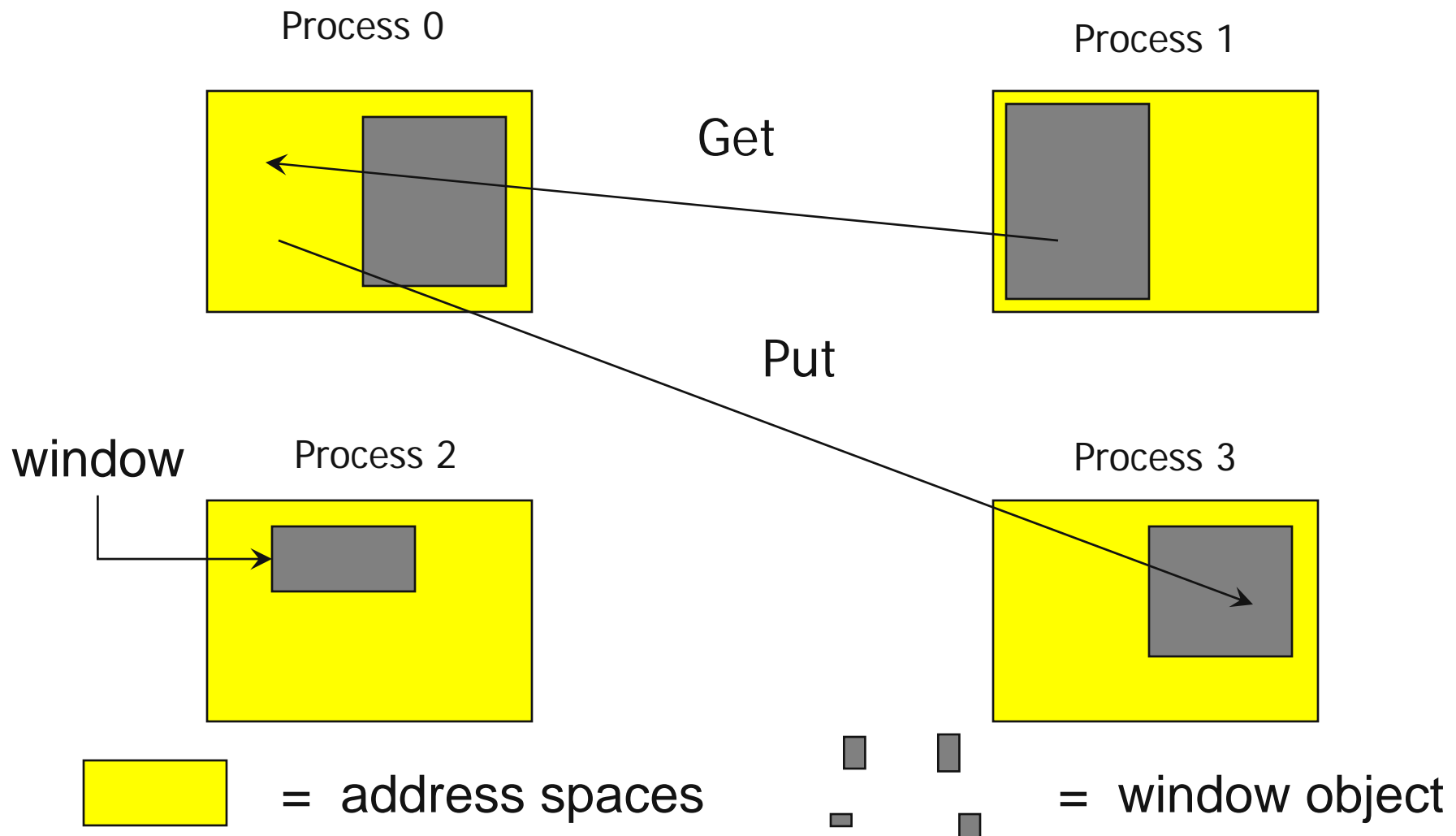


## Remote Memory Access in MPI-2 (also called One-Sided Operations)

### ■ 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

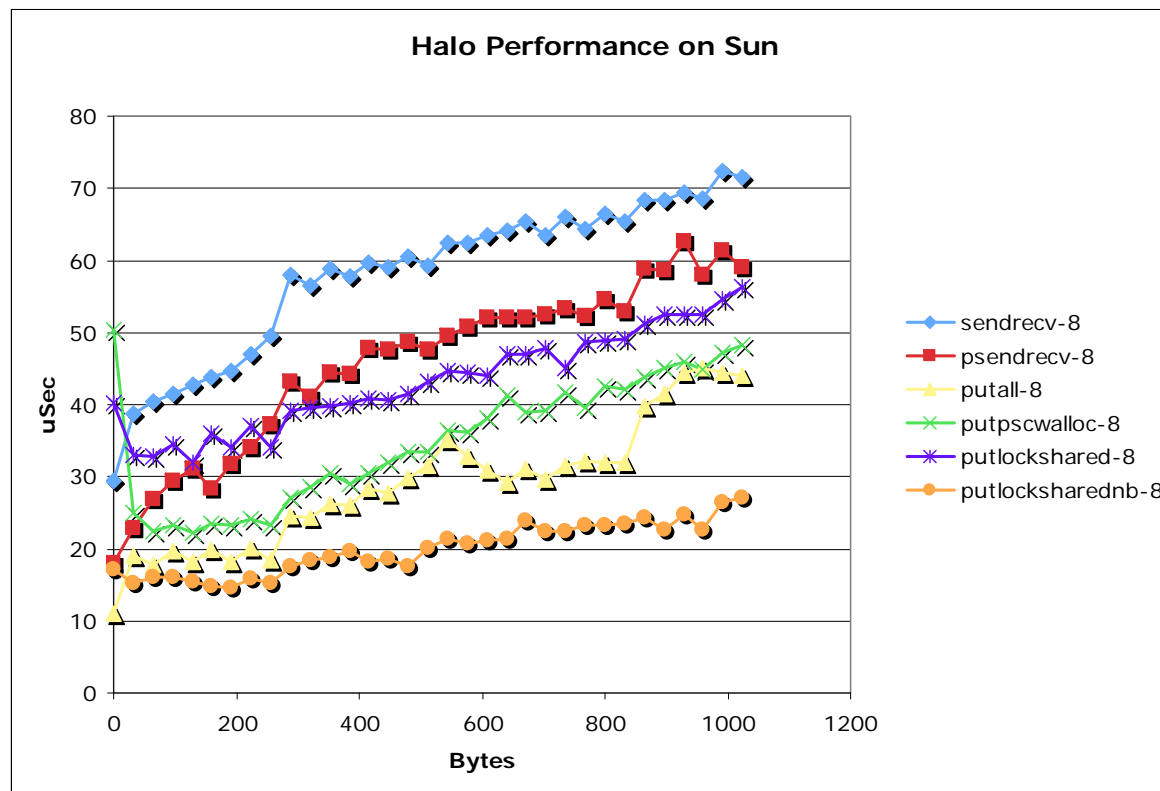


## *Basic RMA Functions for Communication*

- **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\_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**

## Why Use RMA?

- Potentially higher performance on some platforms, e.g., SMPs
- Details later



## *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



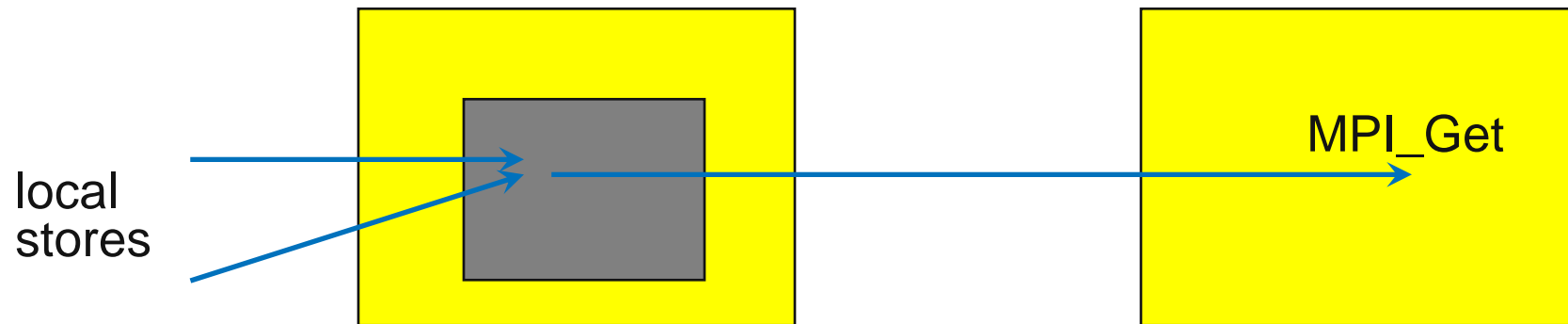
## *RMA Communication Calls*

- **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

MPI\_Win\_fence(win)

MPI\_Put

MPI\_Put

MPI\_Win\_fence(win)

Process 1

MPI\_Win\_fence(win)

MPI\_Win\_fence(win)

## *Mesh Exchange Using MPI RMA*

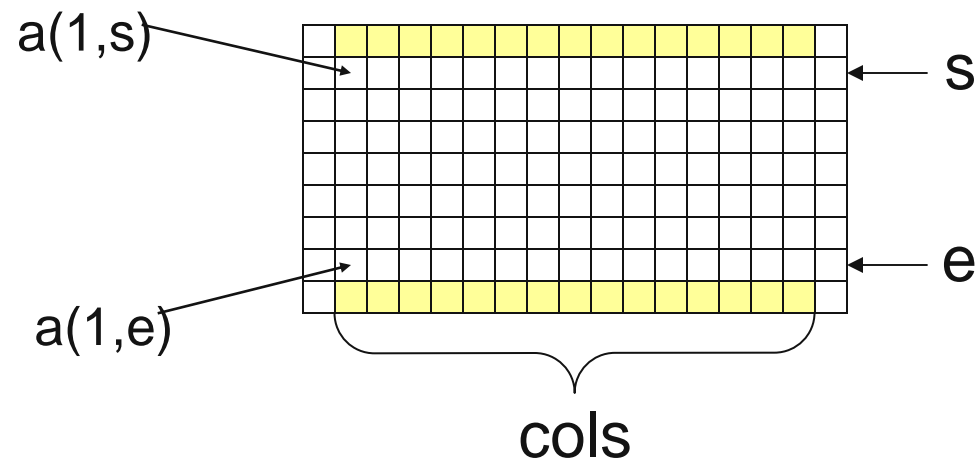
- Define the windows
  - Why – safety, options for performance (later)
- Define the data to move
- Mark the points where RMA can start and where it must complete (e.g., fence/put/put/fence)

## *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
  - $1 + (\# \text{ cells in a row}) * (\# \text{ of rows} - 1)$
  - $= 1 + (\text{cols} + 2) * ((e+1) - (s-1) + 1 - 1)$
  - $= 1 + (\text{cols} + 2) * (e - s + 2)$



## 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
- Two MPI\_Win objects are used because there are two arrays, matrix and temp

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

MPI\_Win\_start(target\_grp)

MPI\_Put

MPI\_Put

MPI\_Win\_complete(target\_grp)

### Process 1

MPI\_Win\_post(origin\_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.

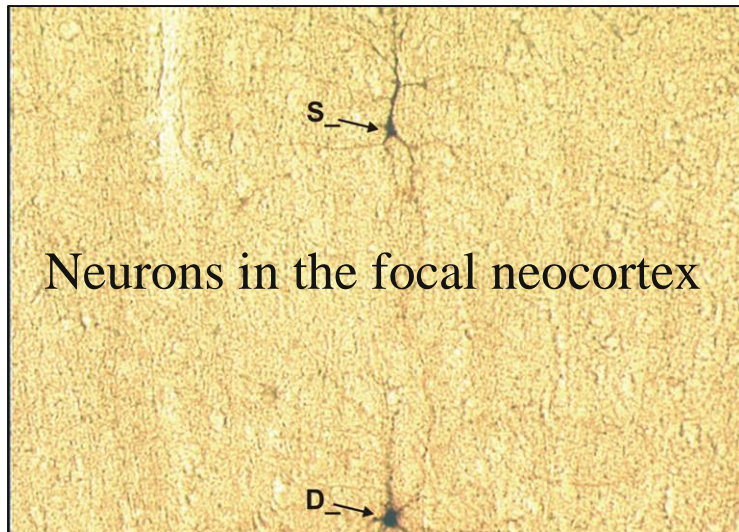
## *pNeo - Modeling the Human Brain*



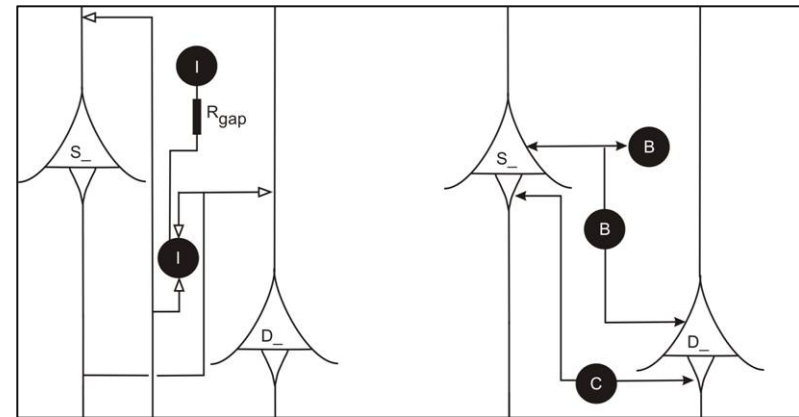
## Science Driver

- 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

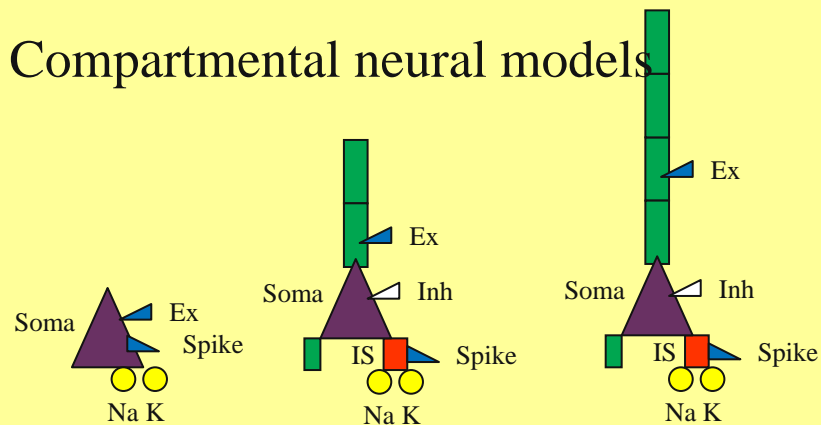


Neurons in the focal neocortex



Excitatory and inhibitory  
signal wiring between neurons

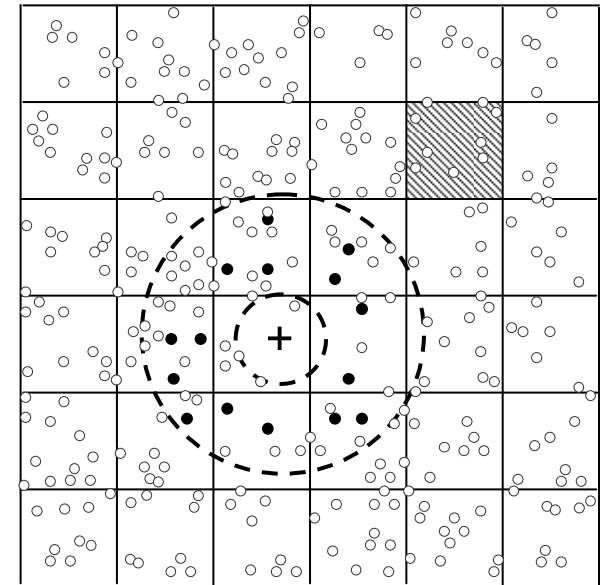
## Compartmental neural models





## 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 not 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) {  
    wi n. Lock( MPI::LOCK_EXCLUSIVE, 1, 0 );  
    wi n. Put(outbuf, n, MPI::INT, 1, 0, n, MPI::INT);  
    wi n. Unl ock(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, zero, n, MPI_INTEGER, win, ierr)
  call MPI_Win_unlock(1, win, ierr)
endif
```

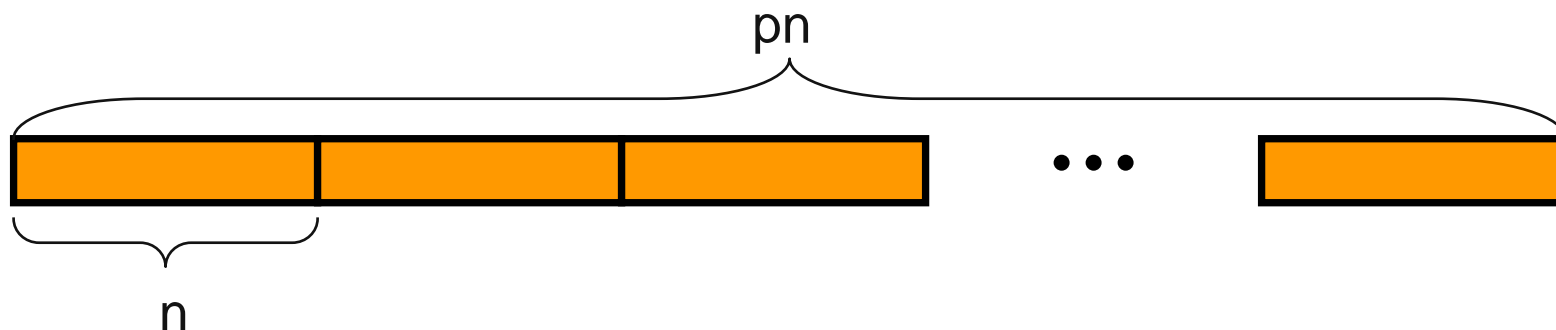
- Only process performing MPI\_Put makes MPI RMA calls
  - Process with memory need not make any MPI calls; it is “passive”
- Similarly for MPI\_Get, MPI\_Accumulate
- zero must be a variable of type
  - Integer (kind=MPI\_ADDRESS\_KIND) zero

## *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 `local[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*

```
volatile 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 *local a;  
...  
local a = MPI::Allloc_mem(n * sizeof(double),  
                           MPI::INFO_NULL);  
win = MPI::Win::Create(local a, 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 *local a;  
rank = i / n;  
offset = i % n;  
MPI_Win_lock(MPI_LOCK_SHARED, rank, 0, win);  
if (rank == myrank) {  
    value = local a[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 *local a;  
rank = i / n;  
offset = i % n;  
win.Lock(MPI::LOCK_SHARED, rank, 0);  
if (rank == myrank) {  
    value = local a[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:

```
doubl e preci si on l ocal a(0:mysi ze-1)
rank = i / n
offset = mod(i , n)
call MPI_Wi n_l ock(MPI_LOCK_SHARED, rank, 0, wi n, i err)
if (rank .eq. myrank) then
    val ue = l ocal a(offset)
el se
    call MPI_Get(&val ue, 1, MPI_DOUBLE_PRECI SI ON, &
                rank, offset, 1, MPI_DOUBLE_PRECI SI ON, &
                wi n, i err)
endi f
call MPI_Wi n_unl ock(rank, wi n, i err)
```

## *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

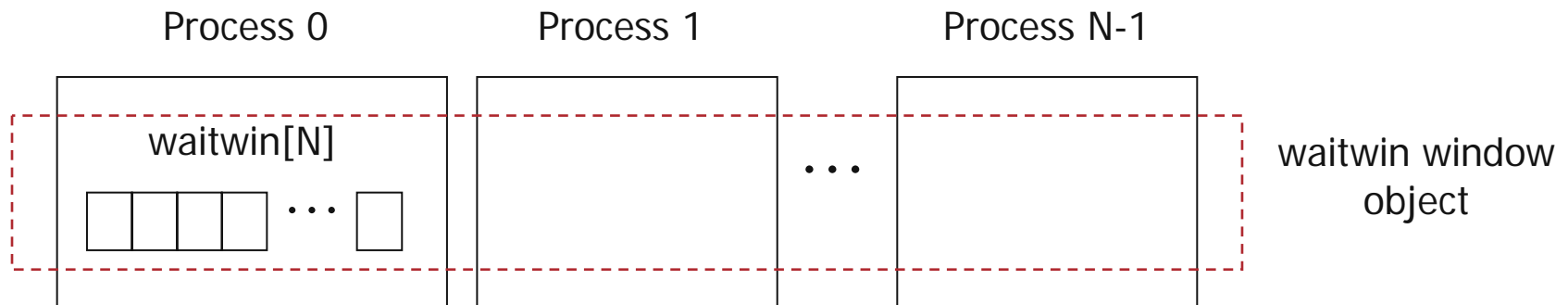
...

call MPI\_FREE\_MEM(u, ierror) ! not p!

## *Mutex with Passive Target RMA*

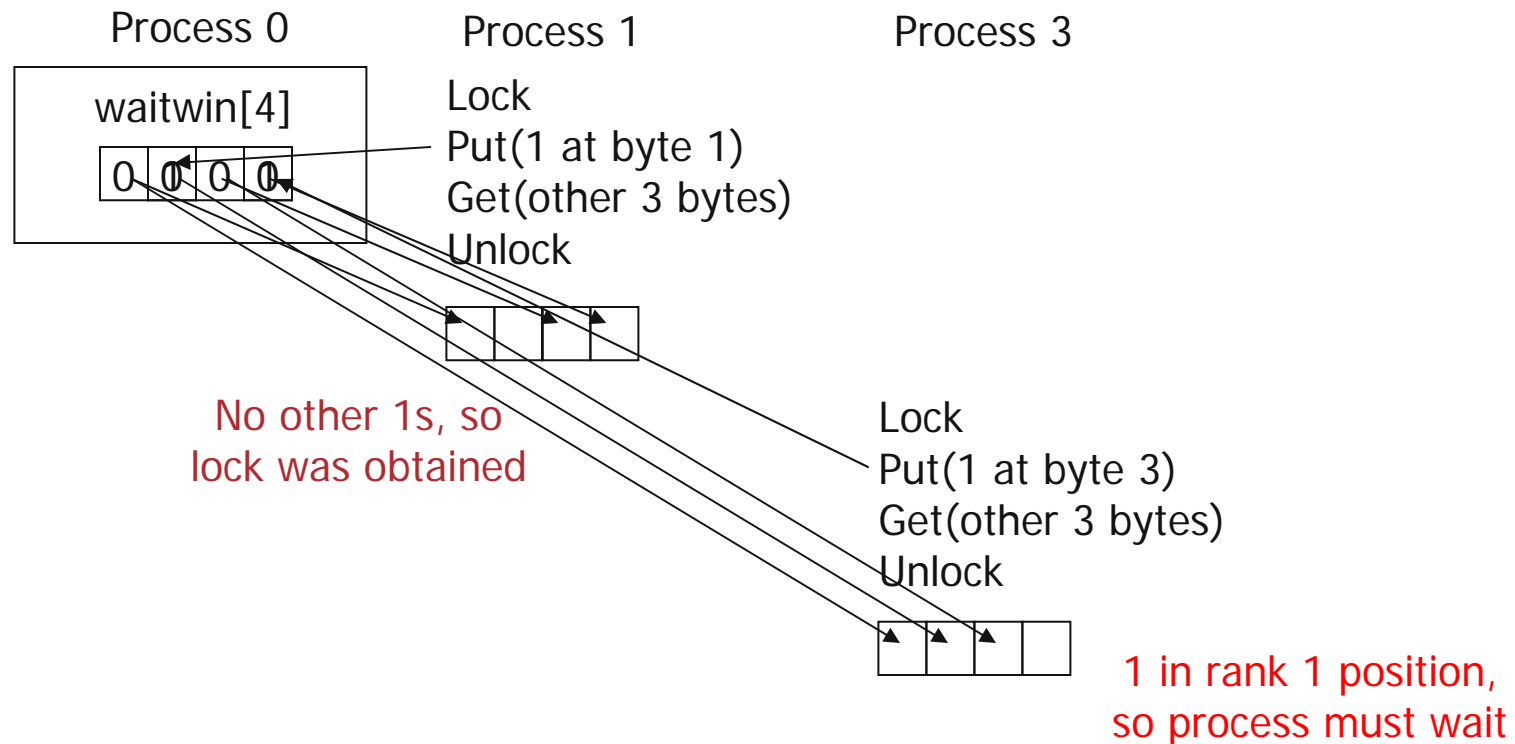
- MPI\_Win\_lock/unlock DO NOT define a critical section
- One has to implement a distributed locking algorithm using passive target RMA operations in order to achieve the equivalent of a mutex
- Example follows

## 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



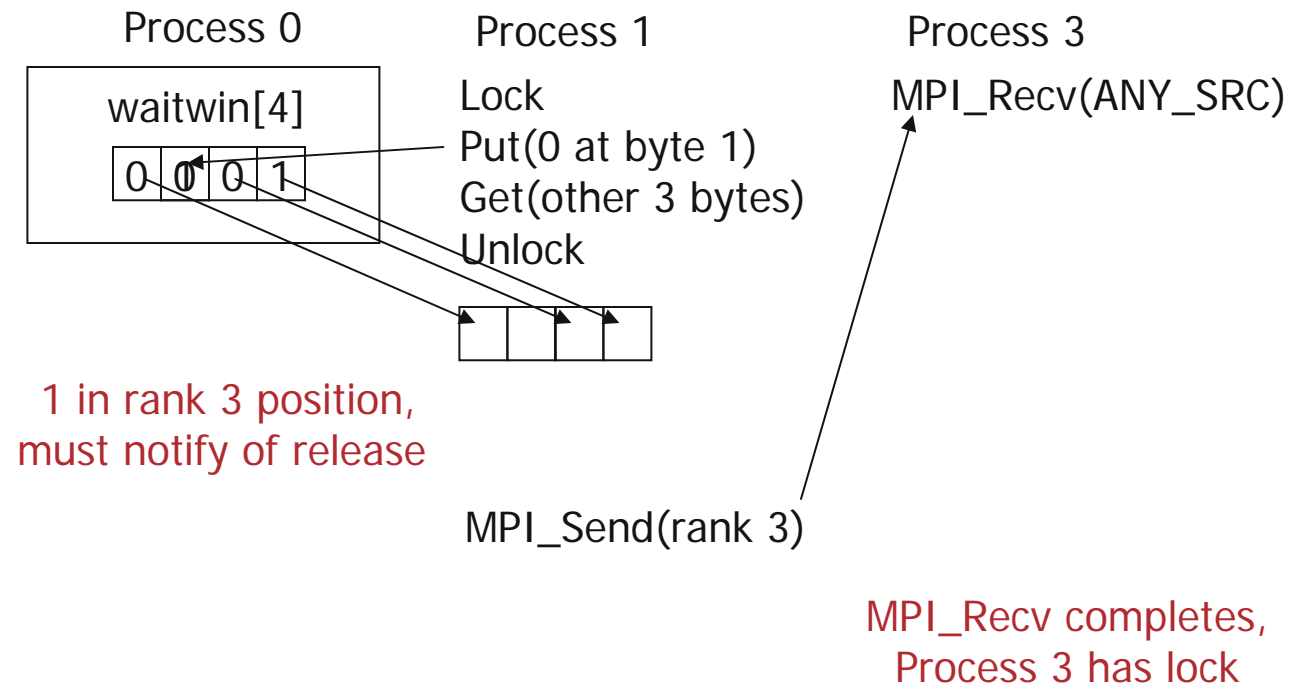
- Processes use one access epoch to attempt to obtain the lock
- Process 1 succeeds, but process 3 must wait

## 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*



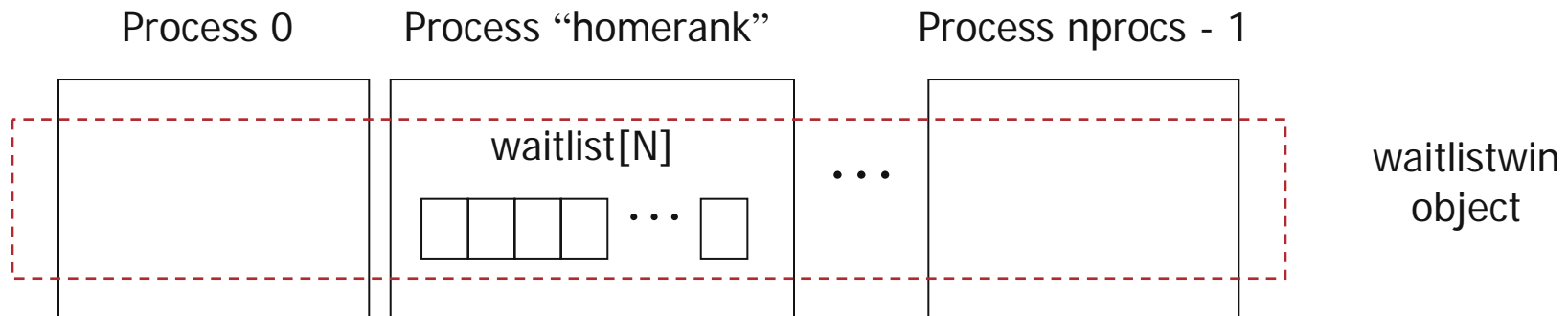
## Releasing the Lock



- Process 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

# *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”

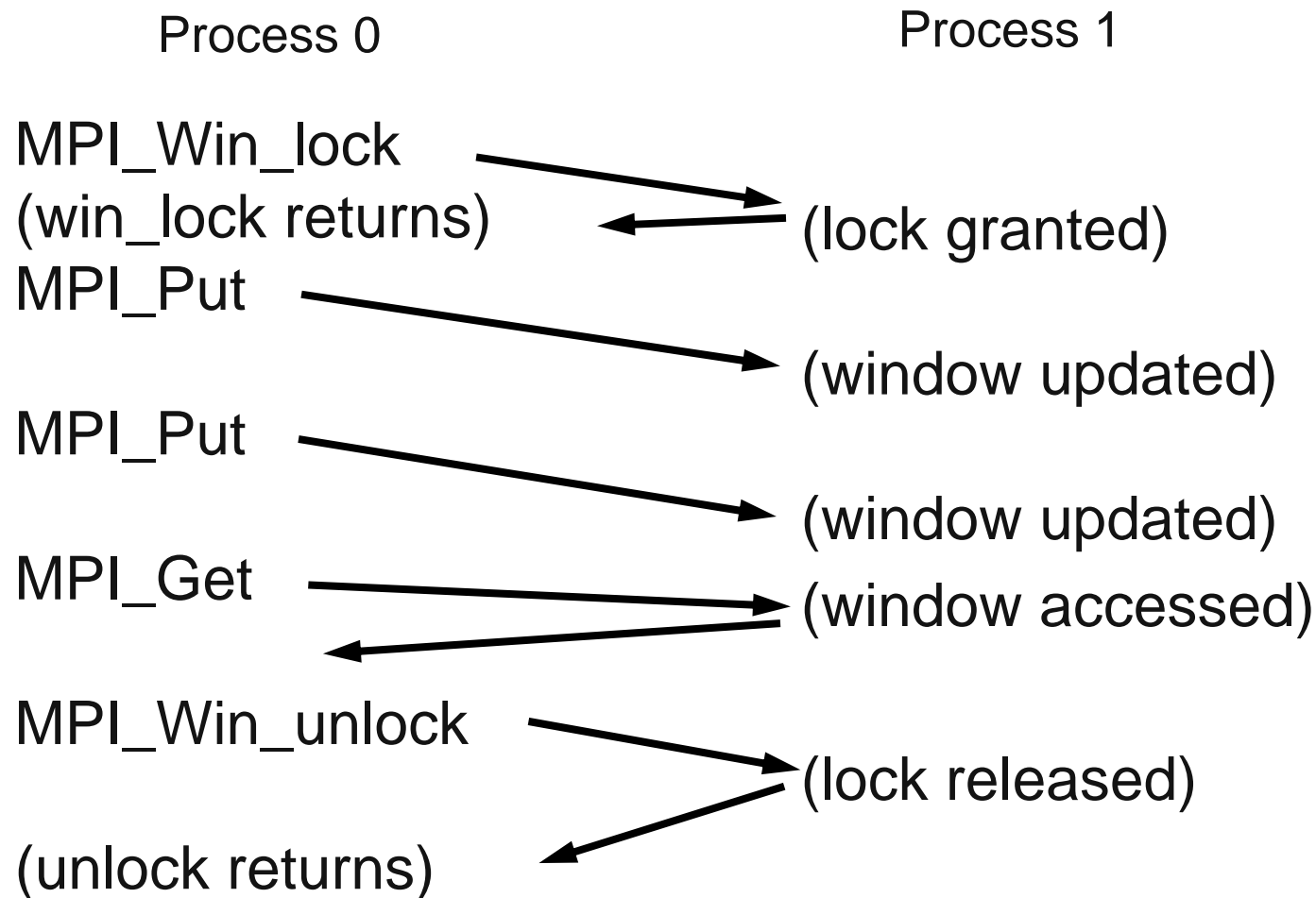
```
integer info;  
call MPI_Info_create( info, ierr )  
call MPI_Info_set( info, "no_locks", "true", ierr )  
call MPI_Win_create( ..., info, ... , ierr )  
call MPI_Info_free( info, ierr )
```

## *Understanding the MPI-2 Completion Model*

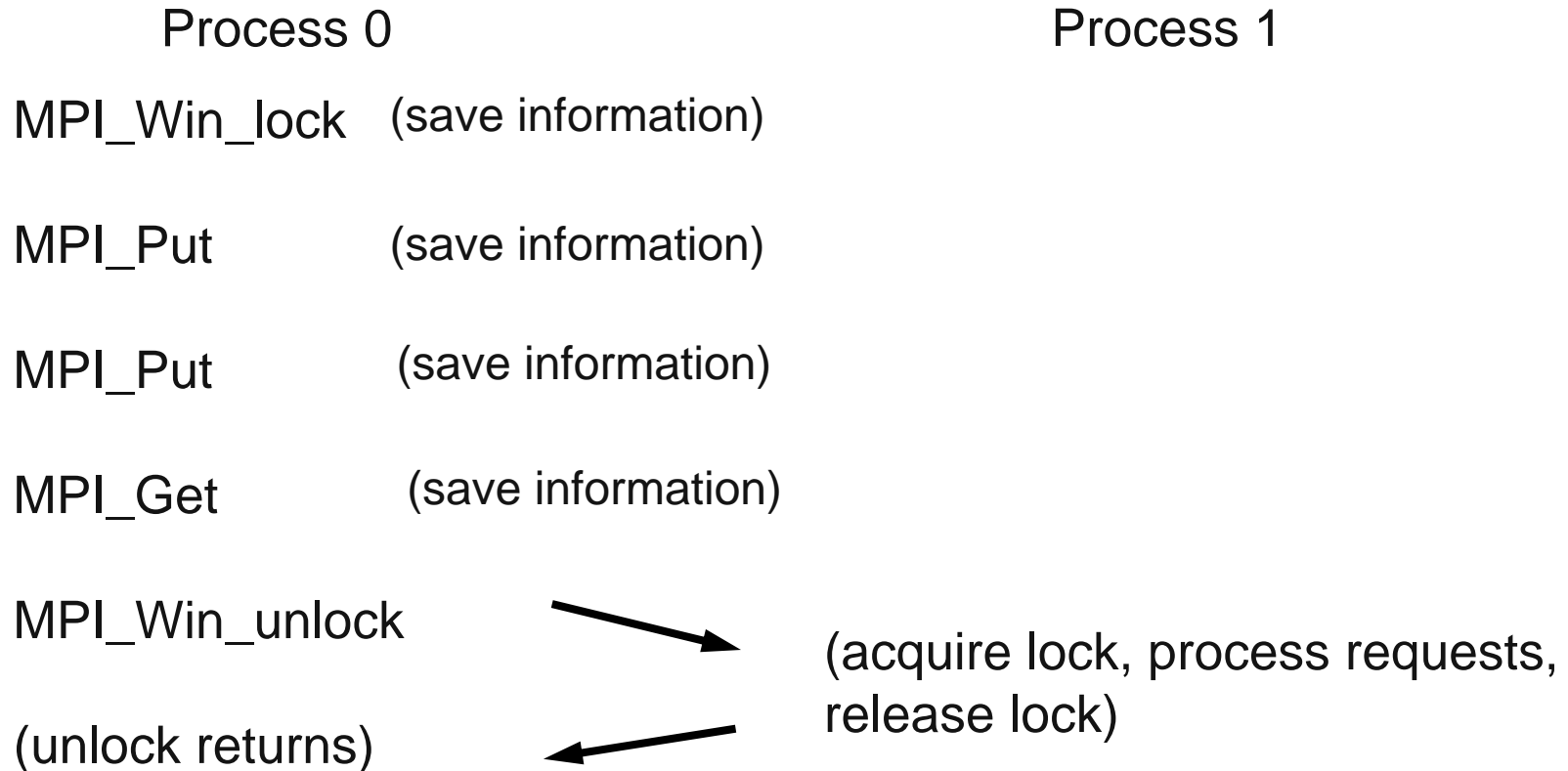
- Very relaxed
  - To give the implementer the greatest flexibility
  - Describing this relaxed model precisely is difficult
    - *Implementer only needs to obey the rules*
  - But it doesn't matter; simple rules work for most programmers
- When does the data actually *move*?



## *Data Moves Early*

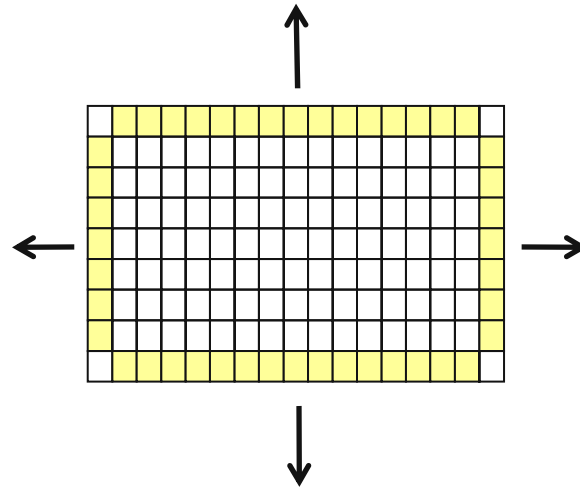


## *Data Moves Late*



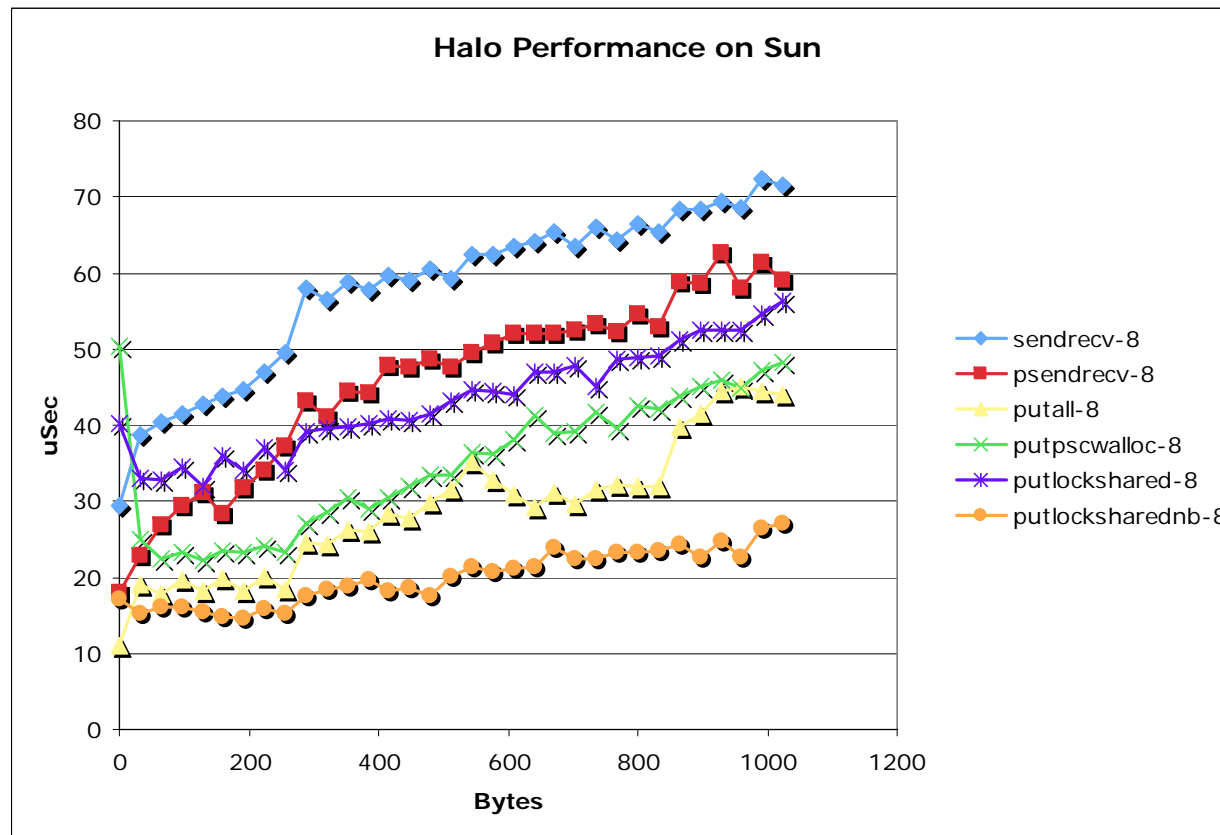
## Performance Tests

- “Halo” exchange or ghost-cell exchange operation
  - Each process exchanges data with its nearest neighbors
  - Part of mpptest benchmark
  - One-sided version uses all 3 synchronization methods

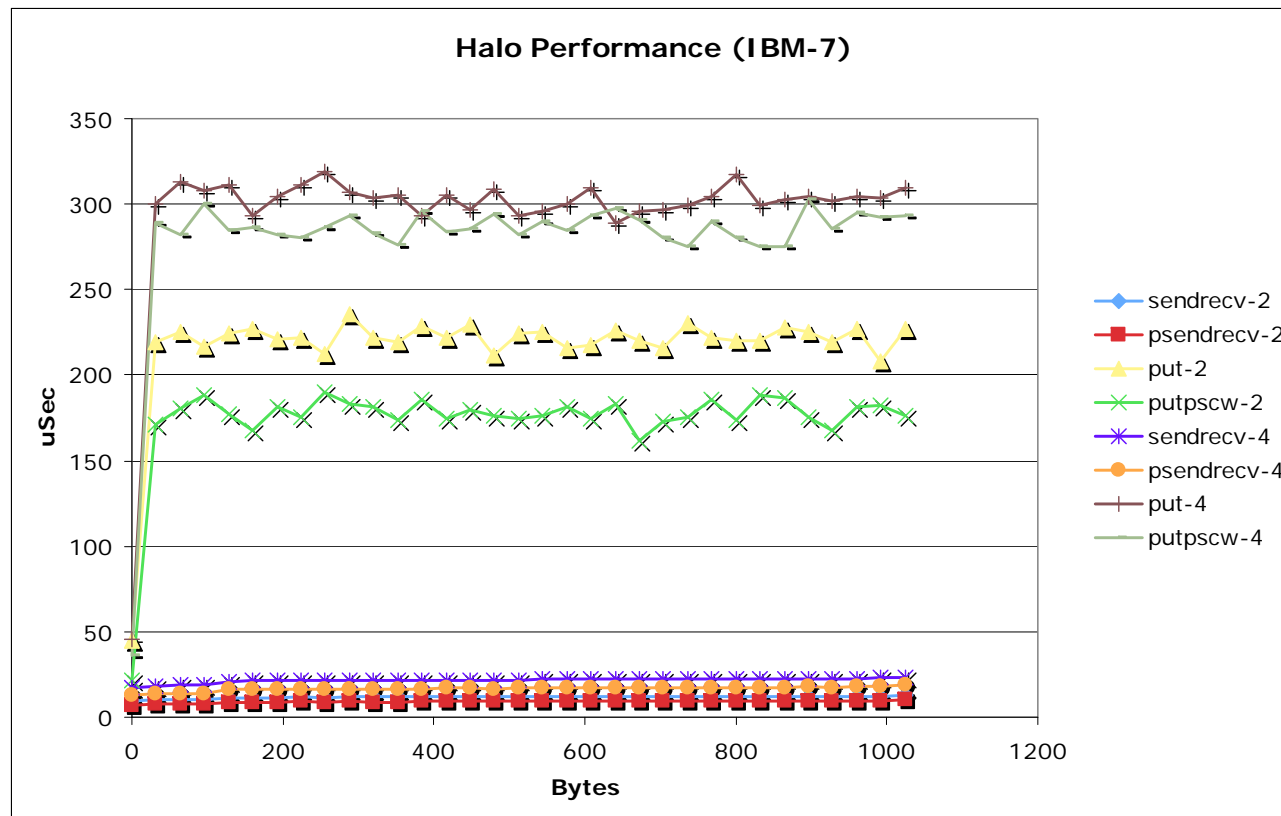


- Ran on
  - Sun Fire SMP at Univ. of Aachen, Germany
  - IBM p655+ SMP at San Diego Supercomputer Center

## One-Sided Communication on Sun SMP with Sun MPI



## One-Sided Communication on IBM SMP with IBM MPI

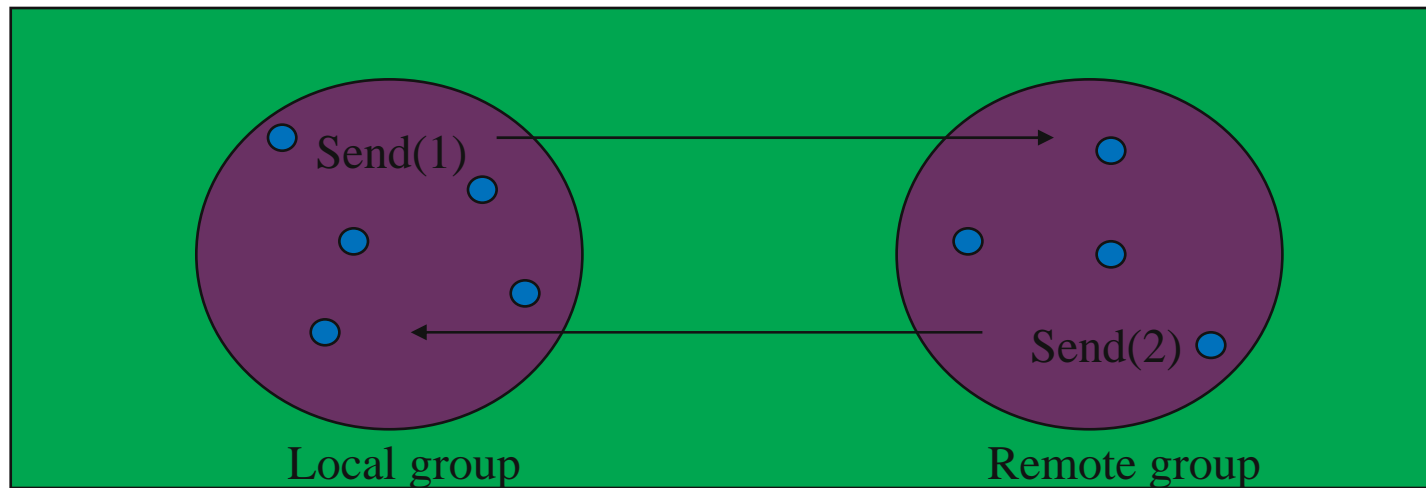


## *Dynamic Process Management*

## Intercommunicators

- MPI has two kinds of communicators: *intra* and *inter* communicators
- Intracommunicators
  - Contain a single group of processes
  - MPI\_COMM\_WORLD is an intracommunicator
- Intercommunicators
  - Contain a *local* group and a *remote* group
  - Point-to-point communication is between a process in one group and a process in the other.
  - Can be merged into a normal (intra) communicator

## Intercommunicators



- In MPI-2, the local and remote groups may come from different MPI\_COMM\_WORLDs



# Dynamic Process Management

## ■ Issues

- maintaining simplicity, flexibility, and correctness
- interaction with operating system, resource manager, and process manager
- connecting independently started processes

## ■ Spawning new processes is *collective*, returning an intercommunicator.

- Local group is group of spawning processes.
- Remote group is group of new processes.
- New processes have own **MPI\_COMM\_WORLD**.
- **MPI\_Comm\_get\_parent** lets new processes find parent communicator.

## Spawning New Processes

```
MPI_Comm_spawn(command, argv, numprocs, info, root,  
               comm, intercomm, errcodes)
```

- Tries to start `numprocs` process running `command`, passing them command-line arguments `argv`
- The operation is collective over `comm`
- Spawnees are in remote group of `intercomm`
- Errors are reported on a per-process basis in `errcodes`
- `Info` used to optionally specify hostname, archname, wdir, path, file, softness.

## *Spawning Multiple Executables*

- `MPI_Comm_spawn_multiple( ... )`
- Arguments `command`, `argv`, `numprocs`, `info` all become arrays
- Still collective

## *In the Children*

- **MPI\_Init** (only MPI programs can be spawned)
- **MPI\_COMM\_WORLD** comprises all child processes spawned with one call to **MPI\_Comm\_spawn**
- **MPI\_Comm\_get\_parent** obtains parent intercommunicator
  - Same as intercommunicator returned by **MPI\_Comm\_spawn** in parents
  - Remote group is spawners
  - Local group is those spawned

## *Manager-Worker Example*

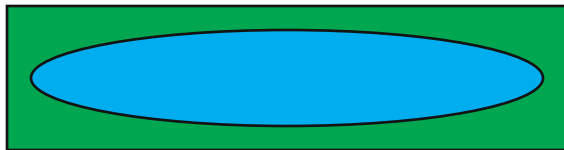
- Single manager process decides how many workers to create and which executable they should run
- Manager spawns  $n$  workers, and addresses them as 0, 1, 2, ...,  $n-1$  in new intercomm
- Workers address each other as 0, 1, ...,  $n-1$  in `MPI_COMM_WORLD`, address manager as 0 in parent intercomm
- One can find out how many processes can usefully be spawned

## *Establishing Connection Between Separately Started MPI Jobs*

- Two sets of MPI processes may wish to establish connections, e.g.,
  - Two parts of an application started separately
  - A visualization tool wishes to attach to an application
  - A server wishes to accept connections from multiple clientsBoth server and client may be parallel programs
- Establishing connections is collective but asymmetric (“Client”/“Server”)
- Connection results in an intercommunicator

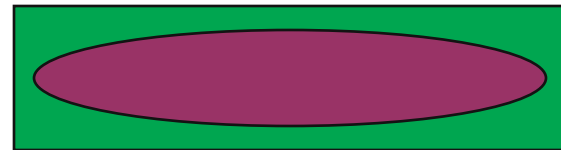
## *Establishing Connection Between Parallel Programs*

In server



MPI\_Comm\_accept

In client



MPI\_Comm\_connect



New intercommunicator

## Connecting Processes

### ■ Server:

- `MPI_Open_port( info, port_name )`
  - *system supplies port\_name*
  - *might be host:num; might be low-level switch #*
- `MPI_Comm_accept( port_name, info, root, comm, intercomm )`
  - *collective over comm*
  - *returns intercomm; remote group is clients*

### ■ Client:

- `MPI_Comm_connect( port_name, info, root, comm, intercomm )`
  - *remote group is server*



## *Optional Name Service*

`MPI_Publish_name( service_name, info, port_name )`

`MPI_Lookup_name( service_name, info, port_name )`

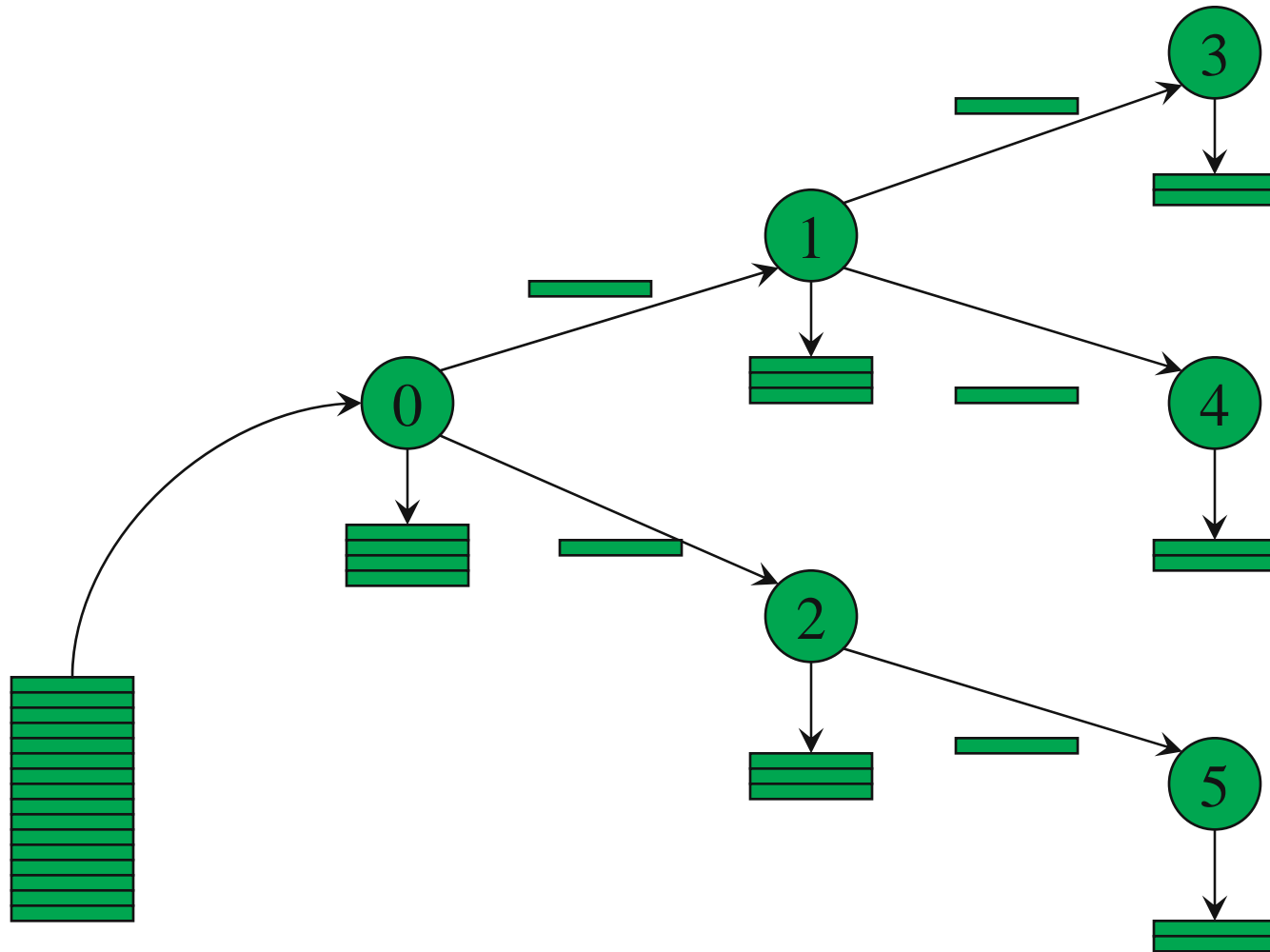
- allow connection between `service_name` known to users and system-supplied `port_name`
- MPI implementations are allowed to ignore this service

## Bootstrapping

- `MPI_Join( fd, intercomm )`
- collective over two processes connected by a socket.
- `fd` is a file descriptor for an open, quiescent socket.
- `intercomm` is a new intercommunicator.
- Can be used to build up full MPI communication.
- `fd` is *not* used for MPI communication.

## *Parallel File Copy Example*

## Parallel File Copy



See `pcp-spawn-master.c` and `pcp-spawn-slave.c` for code examples.

## *Parallelism in Distributing Files*

- All processes writing in parallel
- Message-passing in parallel (assume scalable implementation of MPI\_Bcast)
- Pipeline parallelism with blocks from the file
- Use syntax adopted from cp:

```
pcp 0-63 /home/progs/cpi /tmp/cpi
```

## *Common MPI User Errors*

## 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

## *Recent Efforts of the MPI Forum*



## *Introduction*

- The MPI Forum has resumed meeting since January 2008
- Meetings held every 6–8 weeks in Chicago, SF Bay Area, Portland (with SC09), or in Europe (with Euro PVM/MPI)
- About 30 attendees per meeting
  - All major vendors represented; several attendees from Europe and Japan
- Full schedule: [http://meetings.mpi-forum.org/Meeting\\_details.php](http://meetings.mpi-forum.org/Meeting_details.php)

## *MPI 2.1*

- First official action of the new MPI Forum
- Led by Rolf Rabenseifner, University of Stuttgart
- Minor clarifications and bug fixes to MPI 2.0
- No new features or changes to the API
- Merging of MPI 1.2, MPI 2.0, and errata from the web site into a single document
  - No references to MPI-1 or MPI-2 in the text, just MPI
  - Turned out to be a huge effort to fix the text to make it sound like one cohesive document
- Officially approved by the MPI Forum at the Sept 2008 meeting
- Supported by open source MPI implementations (MPICH2, Open MPI) soon thereafter

## ***MPI 2.2 (Today's Official MPI Standard)***

- Led by Bill Gropp
- Officially approved by the MPI Forum at the Sept 2009 meeting
- Small updates to the standard
  - Does not break backward compatibility
- Spec can be downloaded from the MPI Forum web site  
[www.mpi-forum.org](http://www.mpi-forum.org)
- Also available for purchase as a book from  
<https://fs.hlr.de/projects/par/mpi/mpi22/>
- Supported by MPICH2 1.2

## *New Features in MPI 2.2*

- Scalable graph topology interface
  - Existing interface requires the entire graph to be specified on all processes, which requires too much memory on large numbers of processes
  - New functions allow the graph to be specified in a distributed fashion (MPI\_Dist\_graph\_create, MPI\_Dist\_graph\_create\_adjacent)
- A local reduction function
  - MPI\_Reduce\_local(inbuf, inoutbuf, count, datatype, op)
  - Needed for libraries to implement user-defined reductions
- MPI\_Comm\_create extended to enable creation of multiple disjoint communicators
- Regular (non-vector) version of MPI\_Reduce\_scatter called MPI\_Reduce\_scatter\_block

## *New Features in MPI 2.2*

- MPI\_IN\_PLACE option added to MPI\_Alltoall, Alltoallv, Alltoallw, and Exscan
- The restriction on the user not being allowed to access the contents of the buffer passed to MPI\_Isend before the send is completed by a test or wait has been lifted
- C++ bindings have been deprecated
  - Not used much
  - Difficult to maintain
  - Users prefer to use Boost.MPI
  - *May* be removed in a future version of the MPI specification

## MPI 3

- New features or substantial additions
- Several working Groups
  - Hybrid Programming                      Lead: Pavan Balaji
  - RMA    Leads: Bill Gropp, Rajeev Thakur
  - Fault Tolerance                              Lead: Rich Graham
  - Tools    Lead: Adam Moody
  - Collectives                                      Lead: Torsten Hoefler
  - Fortran bindings                              Lead: Craig Rasmussen
- Nonblocking collectives have already been voted in
  - MPI\_Ibcast, MPI\_Ireduce, MPI\_Ibarrier, etc

## *Mailing Lists and Archives*

■ Web site: <http://lists.mpi-forum.org/>

■ Lists

- mpi-forum
- mpi-22, mpi-3
- mpi3-coll
- mpi3-rma
- mpi3-ft
- mpi3-fortran
- mpi3-tools
- mpi3-hybridpm

## ***For Further Information***

- <http://meetings.mpi-forum.org/>



# *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>
  - Two-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* also available in German from Oldenbourg
    - *Using MPI-2* also available in Japanese, from Pearson Education Japan

## Conclusions

- MPI is a proven, effective, portable parallel programming model
- 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  
<http://www.mcs.anl.gov/mpi/mpich2>
- More Information on PnetCDF
  - Parallel netCDF web site:  
<http://www.mcs.anl.gov/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/>

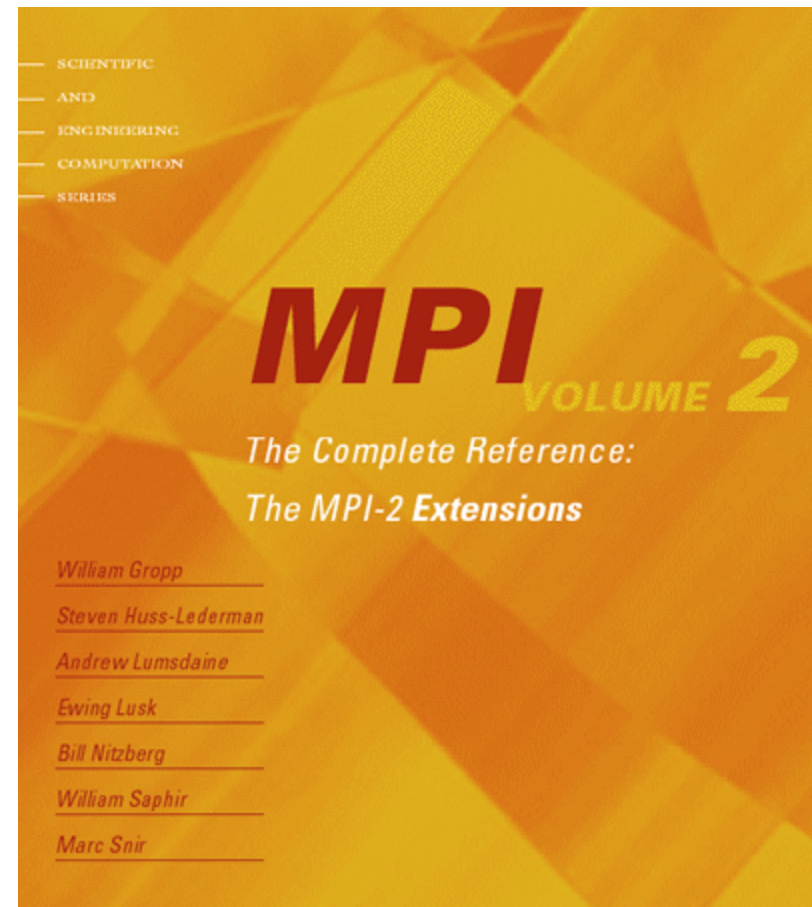
## MPICH2

- 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

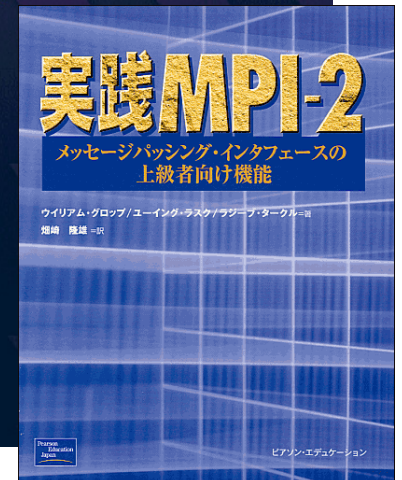
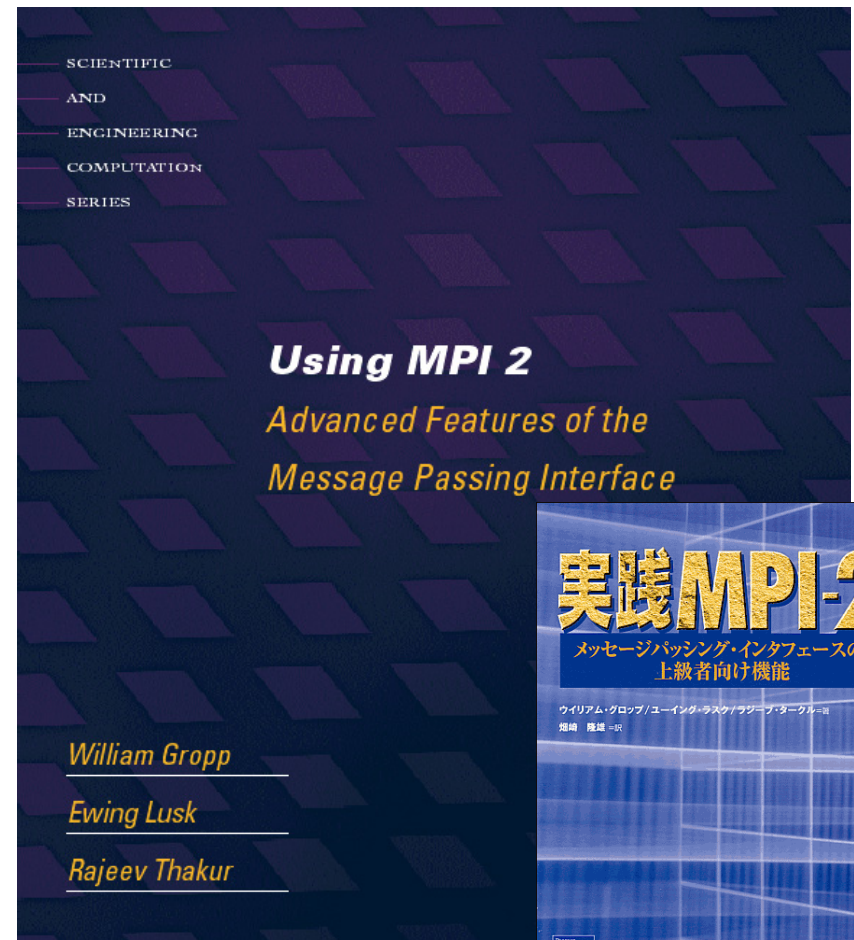
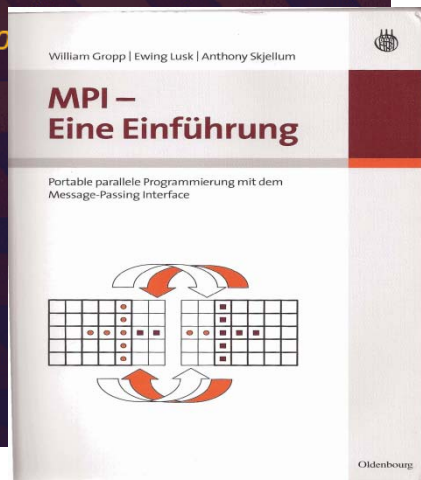
- Incorporates latest research into MPI implementation
  - Our own
    - *Collective operations*
    - *Optimizations for one-sided ops*
    - *Optimized datatype handling*
    - *I/O*
  - Others
    - *Collectives, for example*
- See recent Euro PVM/MPI and Cluster Proceedings
- In use by vendors
  - IBM on BG/P
  - Cray on Red Storm, XT5
  - Intel, Microsoft, Myricom, SiCortex,...
  - 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}>

## *Source code of examples*

- The code examples can be downloaded from [www.mcs.anl.gov/mpi/tutorial/advmpi/mpi2tutorial.tar.gz](http://www.mcs.anl.gov/mpi/tutorial/advmpi/mpi2tutorial.tar.gz)