Today's agenda

- Homework discussion
- Collective Communications: All-with-All
- Derived Datatypes
- Groups, Contexts and Communicators
- Topologies
- Language Binding issues
- The Runtime and Environment Management
- The MPI profiling interface and tracing

Reduce-Scatter

- MPI_Reduce_scatter(void *sendbuf, void *recvbuff, int *revcnt, MPI_Datatype type, MPI_Op op, MPI_Comm comm)
- MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcnt, type, op, comm, ier)
- Can be considered as a

  MPI_Reduce(sendbuf, tmpbuf, cnt, type, op, root, comm);

  MPI_Scatterv(tmpbuf, recvcnt, displs, type, recvbuff, recvcnt[myid], type, root, comm);

  where cnt is the total sum of the recvcnt values and displs[k] is the sum of the recvcnt for up to processor k-1.
- Implementations may use a more optimal approach

How it would work for matvec

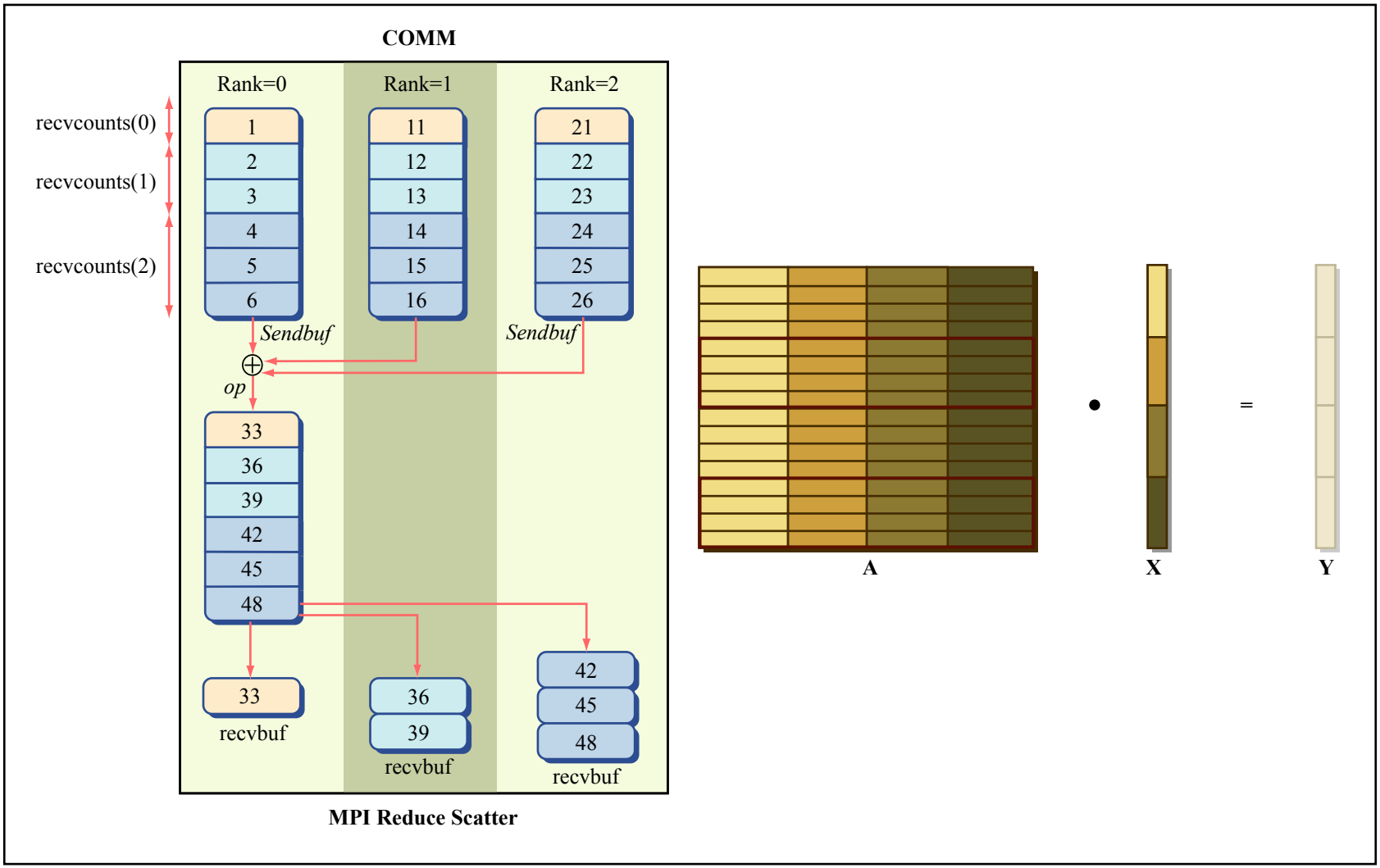


Figure by MIT OpenCourseWare.

3DFFT

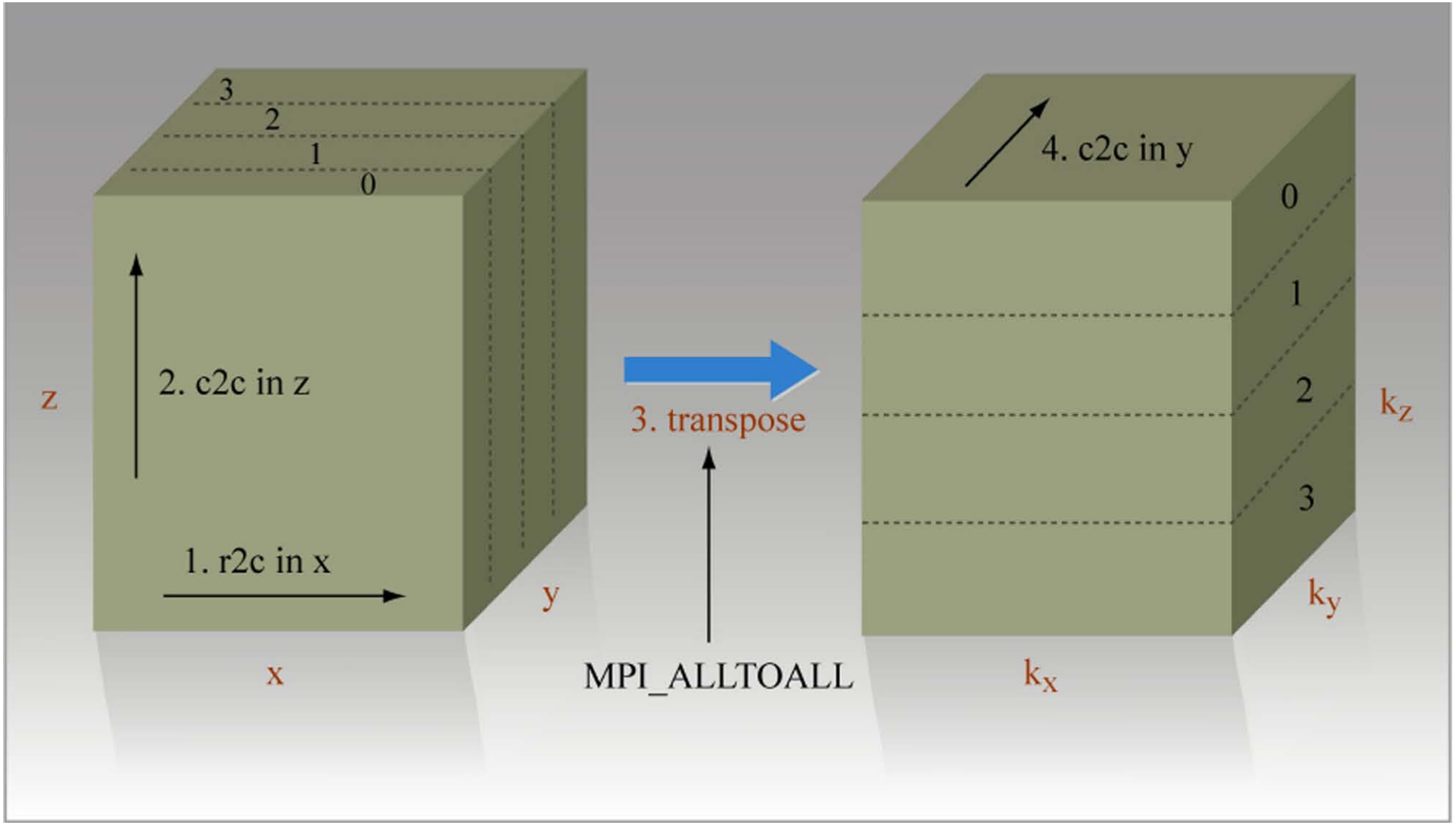


Figure by MIT OpenCourseWare.

Alternative decomposition

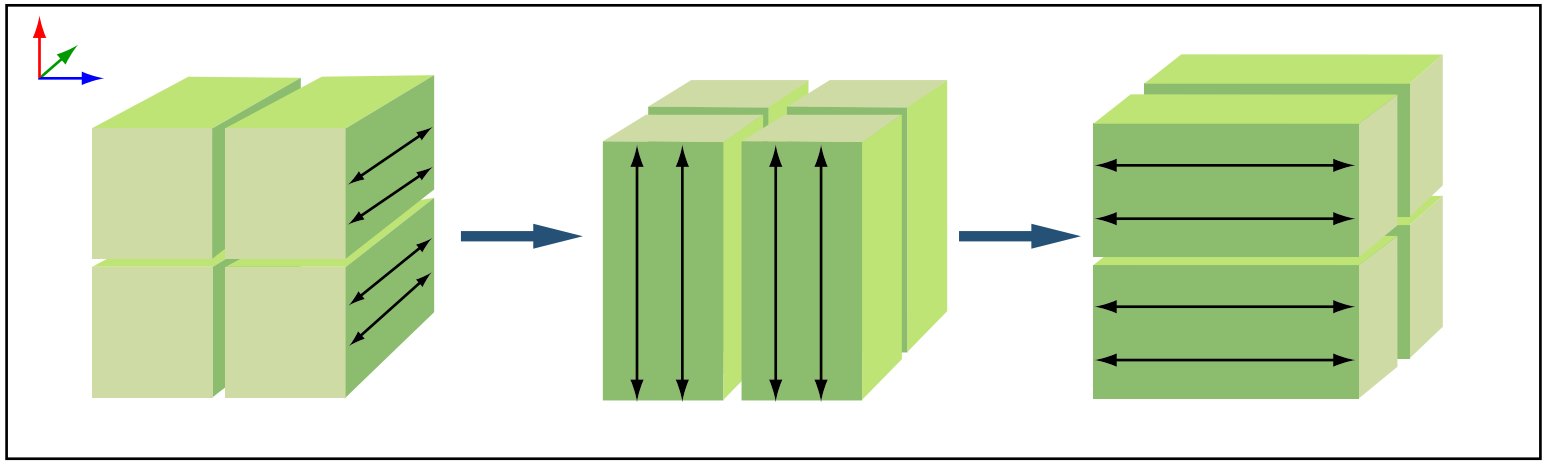


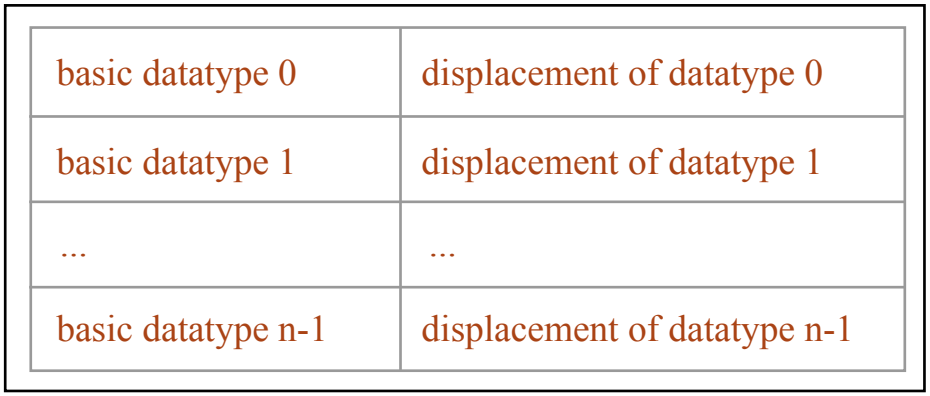
Figure by MIT OpenCourseWare.

Why Derived Datatypes?

- So far all MPI operations seen operate on 1D arrays of predefined datatypes.
  - Multidimensional arrays (linearly stored in memory) can be sent as the equivalent 1D array
  - Contiguous sections of arrays need to be copied (implicitly in Fortran 90/95) to one big chunk to sent over
  - Edges, vertices etc. of 2D/3D arrays need to be sent separately or packed to a sent buffer on the sent side and unpacked from the receive buffer on the receive side, at the programmer's effort
  - Strided data needs to be packed/sent/received/unpacked as above.
- Message aggregation: int & double in same message

What is a Derived Datatype?

- A general datatype is an opaque object specifying:
  - A sequence of basic datatypes
  - A sequence of integer (byte) displacements
- Type signature:
  - {type1, type2, ..., typeN}
- Type map:
  - {(type1,disp1), (type2,disp2)



basic datatype 0	displacement of datatype 0
basic datatype 1	displacement of datatype 1
...	...
basic datatype n-1	displacement of datatype n-1

Figure by MIT OpenCourseWare.

- The displacements are not required to be positive, distinct, or in increasing order. Therefore, the order of items need not coincide with their order in store, and an item may appear more than once.

A derived datatype

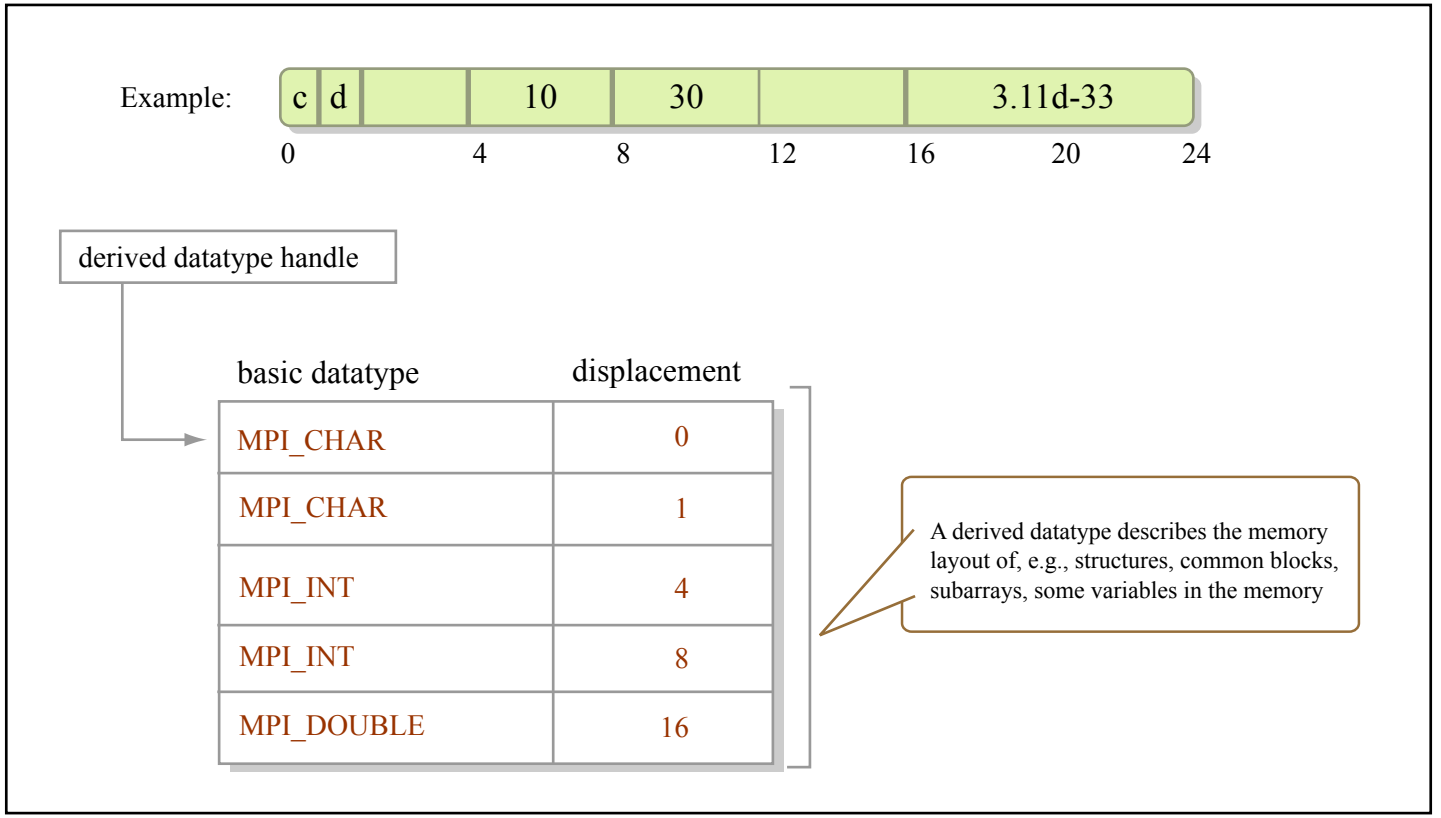


Figure by MIT OpenCourseWare.

More details

- Basic (predefined) MPI datatypes are in fact defined in the same way, based on base language datatypes
- (User) derived datatypes can be defined in terms of basic as well as other defined datatypes.
  - This level of recursive definition can be repeated to construct very complicated datatypes
- Just like basic datatypes, defined datatypes can be used as arguments to communication routines.
- An efficient implementation of communication events when working with such complicated datatypes is left to the implementation
  - May use optimizations known to work on architecture

Size and Extent

- Size: length of "useful" part == data to be transferred
- Extent: the span from the first byte to the last byte occupied by entries in this datatype, rounded up to satisfy alignment requirements.
- Alignment is architecture/language/compiler specific

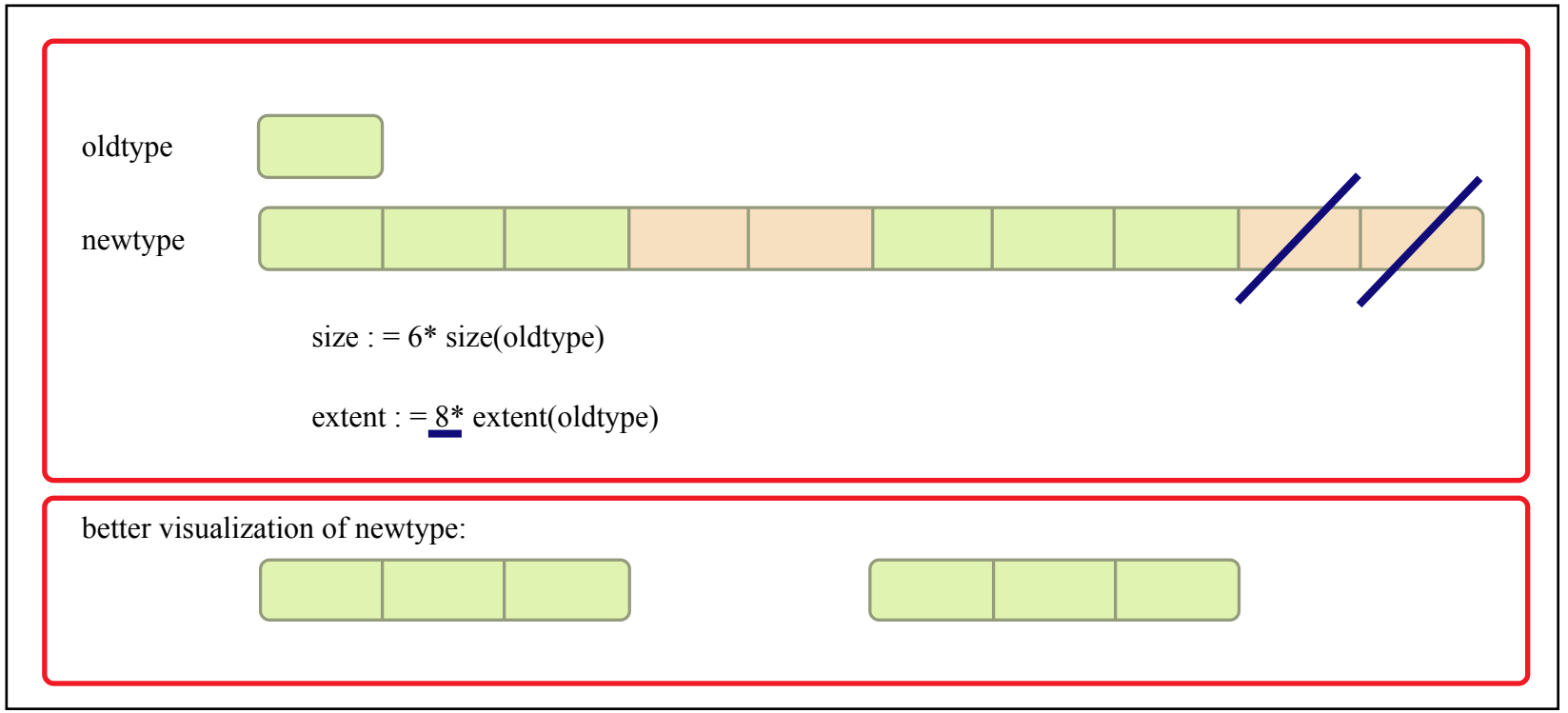


Figure by MIT OpenCourseWare.

Datatype construction: Contiguous

- MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
- The simplest possible derived datatype
- Concatenation of *count* copies of oldtype variables
- Call with 2, MPI_DOUBLE_PRECISION to get your own MPI_DOUBLE_COMPLEX in Fortran if absent.

Figure by MIT OpenCourseWare.

Datatype construction: Vector

- MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)
- Concatenation of *count* copies of blocks of oldtype variables of size *blocklength* positioned *stride* blocks apart. Strides (displacements) can be negative.

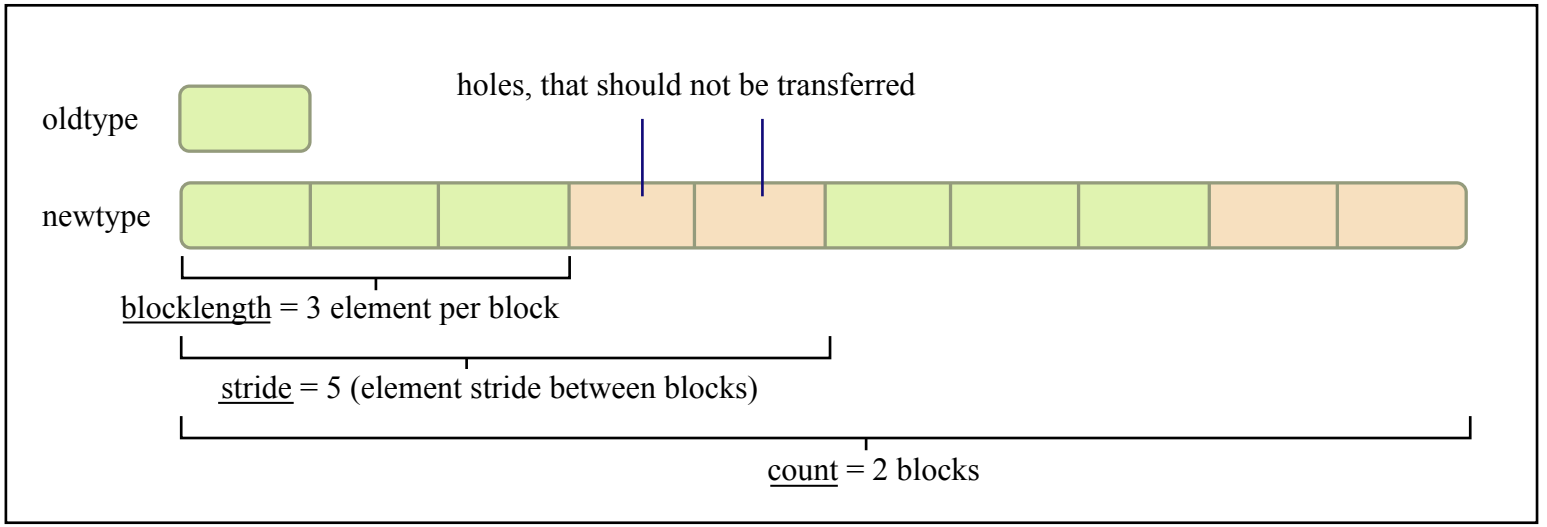


Figure by MIT OpenCourseWare.

More about vector types

- Type before described as
  - MPI_Type_vector(2, 3, 5, oldtype, newtype)
- MPI_Type_contiguous(n, oldtype, newtype) same as:
  - MPI_Type_vector(n, 1, 1, oldtype, newtype)
  - MPI_Type_vector(1, n, k, oldtype, newtype) for any k
- MPI_Type_hvector() requires stride to be in bytes, instead of *oldtype* units. Type is MPI_Aint.
- MPI_Type_indexed(int count, int *array_of_blocklen, int *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype); *MPI_Type_hindexed()*
  - For vectors with variably sized blocks, variable strides

Datatype construction: Structure

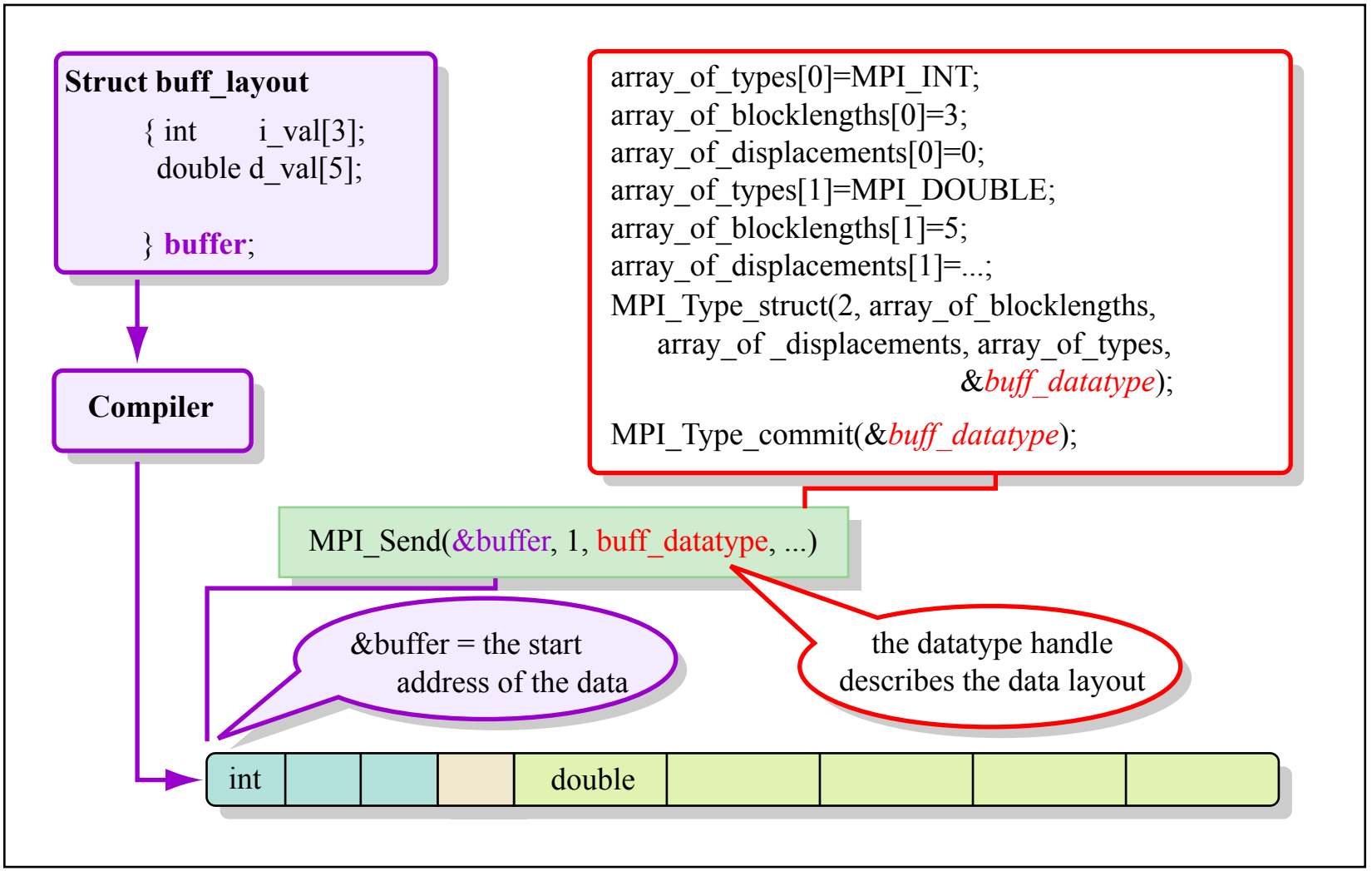


Figure by MIT OpenCourseWare.

Alignment, gaps and addresses

- MPI_Type_struct(int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_oldtypes, MPI_Datatype *newtype);
- Alignment restrictions may require the presence of gaps in your structure.
- count=2, array_of_blocklenghts=[3,5], array_of_types=[MPI_INT,MPI_DOUBLE]
- What about array_of_displacements ? [0,addr1-addr0]

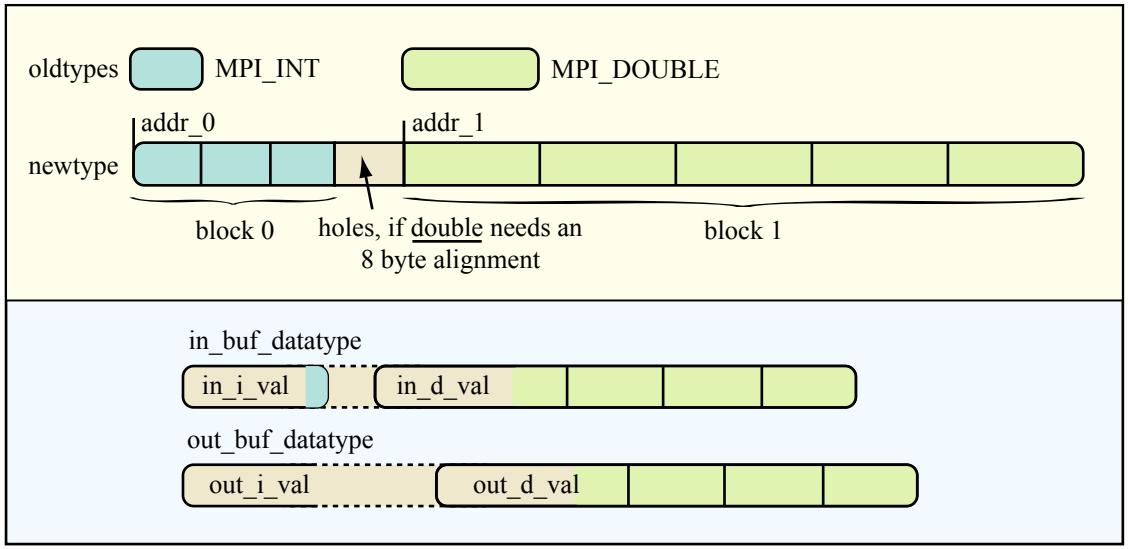


Figure by MIT OpenCourseWare.

Address, size and extent

- MPI_Address(void* location, MPI_Aint *address)
- MPI_BOTTOM for the start of the address space
  - Use MPI_Address to get absolute addresses for your constituent parts and calculate the correct displacement, with the gaps the the compiler requires
  - MPI_Type_lb/ub() & MPI_LB/UB for endbounds
- MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
  - Will calculate the proper extent in bytes of the datatype
- MPI_Type_size(MPI_Datatype datatype, int *size)
  - Will calculate the proper size in bytes ("useful" part that gets communicated) of the datatype.

Correct usage of addresses (std)

- Successively declared variables in C or Fortran are not necessarily stored at contiguous locations. Thus, care must be exercised that displacements do not cross from one variable to another. Also, in machines with a segmented address space, pointers arithmetic has some peculiar properties. Thus, the use of pointer addresses should instead be replaced by the use of absolute addresses, ie. displacements relative to the start address MPI_BOTTOM.
- Variables belong to the same sequential storage if they belong to the same array, to the same COMMON block in Fortran, or to the same structure in C. Beware of unions! **Look up the rules in the standard!**

Creating & destroying datatypes

- MPI_Type_commit(MPI_Datatype *datatype)
  - You can now go ahead and use the datatype in any communication operation that makes sense.
  - A datatype may specify overlapping entries. The use of such a datatype in a receive operation is erroneous. (This is erroneous even if the actual message received is short enough not to write any entry more than once.)
- MPI_Type_free(MPI_Datatype *datatype)
  - Freeing a datatype does not affect any other datatype that was built from the freed datatype. The system behaves as if input datatype arguments to derived datatype constructors are passed by value. Any communication operations using this datatype that are still pending will complete fine.

Creating equivalent types

- Create types:
  - CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, type2, ...)
  - CALL MPI_TYPE_CONTIGUOUS(4, MPI_REAL, type4, ...)
  - CALL MPI_TYPE_CONTIGUOUS(2, type2, type22, ...)
- With proper care, any of the above can be used to accomplish the same end. Which is to be used is a matter of programming clarity and performance. While in principle complex types composed of complex types should not be slower, implementations may not really manage the indirection well.

Matching sends & receives

- Sends:
  - CALL MPI_SEND(a, 4, MPI_REAL, ...)
  - CALL MPI_SEND(a, 2, type2, ...)
  - CALL MPI_SEND(a, 1, type22, ...)
  - CALL MPI_SEND(a, 1, type4, ...)
- Receives:
  - CALL MPI_RECV(a, 4, MPI_REAL, ...)
  - CALL MPI_RECV(a, 2, type2, ...)
  - CALL MPI_RECV(a, 1, type22, ...)
  - CALL MPI_RECV(a, 1, type4, ...)
- Each of the sends matches any of the receives.

Counting

- MPI_Get_elements(MPI_Status *status, MPI_Datatype datatype, int *count)
- MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
- Define a derived datatype

CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, Type2, ierr)

CALL MPI_TYPE_COMMIT(Type2, ierr)

- One processors sends consecutively:

CALL MPI_SEND(a, 2, MPI_REAL, 1, 0, comm, ierr)

CALL MPI_SEND(a, 3, MPI_REAL, 1, 0, comm, ierr)

Counting example

- The other process receives

```
CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
CALL MPI_GET_COUNT(stat, Type2, i, ierr)  !i=1
CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr) !i=2
CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
CALL MPI_GET_COUNT(stat, Type2, i, ierr)
! returns i=MPI_UNDEFINED
CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr) !i=3
```

Datatyping array sections

```
REAL a(100,100,100), e(9,9,9) ! e=a(1:17:2, 3:11, 2:10)

CALL MPI_TYPE_VECTOR( 9, 1, 2, MPI_REAL, oneslice,
    ierr)

CALL MPI_TYPE_HVECTOR(9, 1, 100*sizeofreal, oneslice,
    twoslice, ierr)

CALL MPI_TYPE_HVECTOR( 9, 1, 100*100*sizeofreal,
    twoslice, 1, threeslice, ierr)

CALL MPI_TYPE_COMMIT( threeslice, ierr)

CALL MPI_SENDRECV(a(1,3,2), 1, threeslice, myrank, 0, e,
    9*9*9, MPI_REAL, myrank, 0, MPI_COMM_WORLD,
    status, ierr)
```

Groups, Contexts, Communicators

- Group: An ordered set of processes, each associated with a rank (within a continuous range). Part of a communicator.
  - Predefined: MPI_GROUP_EMPTY,MPI_GROUP_NULL
- Context: A property of a communicator that partitions the communication space. Not externally visible.
  - Contexts allow Pt2Pt and collective calls not to interfere with each other; same with calls belonging to different communicators.
- Communicator: Group+Context+cached info
  - Predefined: MPI_COMM_WORLD,MPI_COMM_SELF
- Intra- and Inter-communicators

Group Constructors

- MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
- MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
- MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
- MPI_Group_difference(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
- MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
- MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
- MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup)
- MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup)

More group functions

- MPI_Group_free(MPI_Group *group)
- MPI_Group_size(MPI_Group group, int *size)
- MPI_Group_rank(MPI_Group group, int *rank)
- MPI_Group_translate_ranks (MPI_Group group1, int n, int *ranks1, MPI_Group group2, int *ranks2)
- MPI_Group_compare(MPI_Group group1,MPI_Group group2, int *result)
- MPI_IDENT results if the group members and group order is exactly the same in both groups. This happens for instance if group1 and group2 are the same handle. MPI_SIMILAR results if the group members are the same but the order is different. MPI_UNEQUAL results otherwise.

Communicator Functions

- MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
- MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
- MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
- MPI_Comm_compare(MPI_Comm comm1,MPI_Comm comm2, int *result)
- MPI_Comm_free(MPI_Comm *comm)
- And the MPI_Comm_size,MPI_Comm_rank we have already met.

Inter-Communicators

- So far all communications have been between processes belonging to the same communicator.
- MPI allows for communications between different communicators.
  - They can only be Pt2Pt and not collective
  - They require the generation of inter-communicator objects.
  - For more look at the material on the Web and the standard.

Virtual Topologies

- Employing the information cached in communicators we can map an (intra-)communicator's processes to an underlying topology (cartesian or graph) that better reflects the communication requirements of our algorithm.
- This has possible performance advantages: The process to hardware mapping *could* be thus more optimal. *In practice this is rare.*
- The notational power of this approach however allows code to be far more readable and maintainable.

A cartesian topology

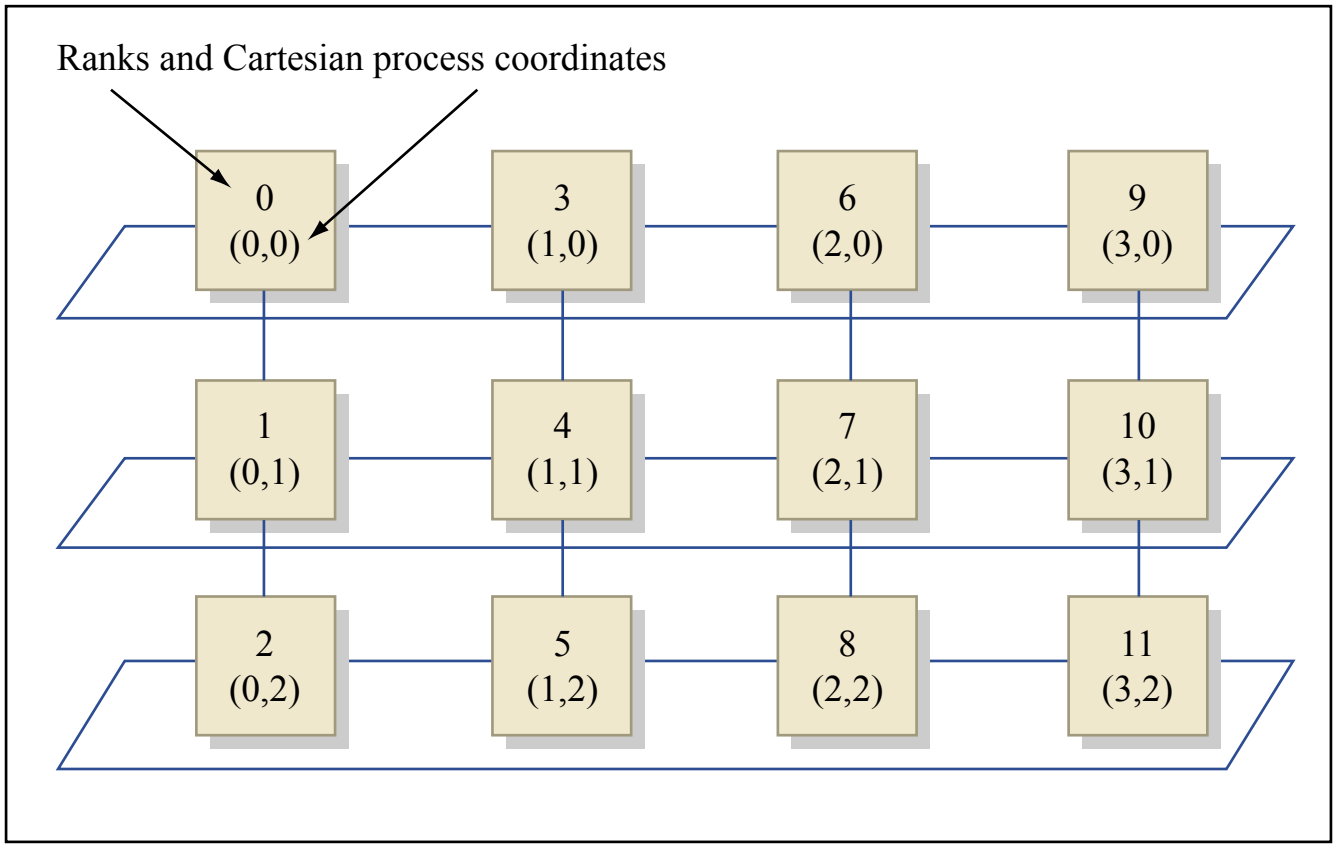


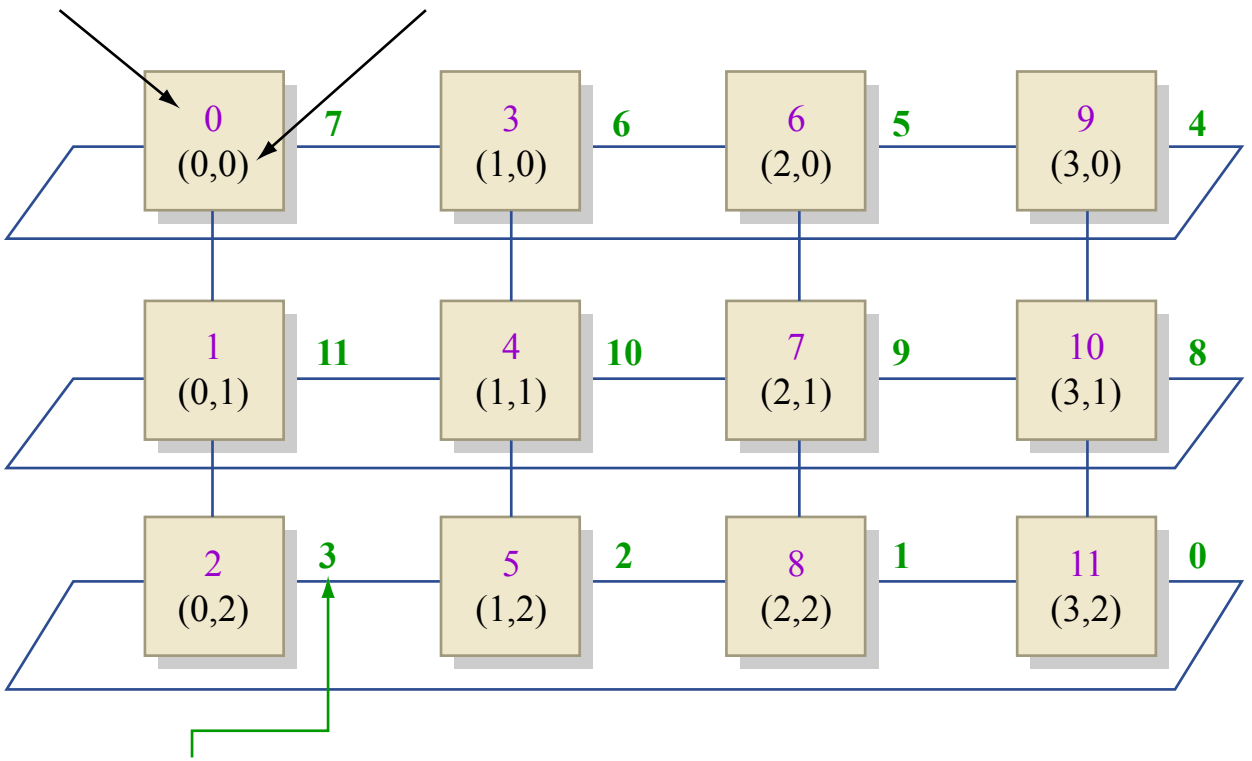
Figure by MIT OpenCourseWare.

Cartesian topology calls

- MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)
  - Extra processes get MPI_COMM_NULL for comm_cart
- MPI_Dims_create(int nnodes, int ndims, int *dims)
  - If ndims(k) is set, this is a constraint
- For graphs, MPI_Graph_create(), same rules
- MPI_Topo_test(MPI_Comm comm, int *status)
  - Returns MPI_CART, MPI_GRAPH, MPI_UNDEFINED
- MPI_Cartdim_get, MPI_Cart_get etc. for cartesian topologies
- MPI_Graphdim_get, MPI_Graph_get etc. for graphs

Ranks in cartesian communicators

- Ranks and Cartesian process coordinates in comm_cart



- Ranks in comm and comm_cart may differ, if **recorder = 1** or **.TRUE.**
- This reordering can allow MPI to optimize communications

Figure by MIT OpenCourseWare.

Cartesian rank/coordinate functions

- MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank); out of range values get shifted (periodic topos)
- MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)

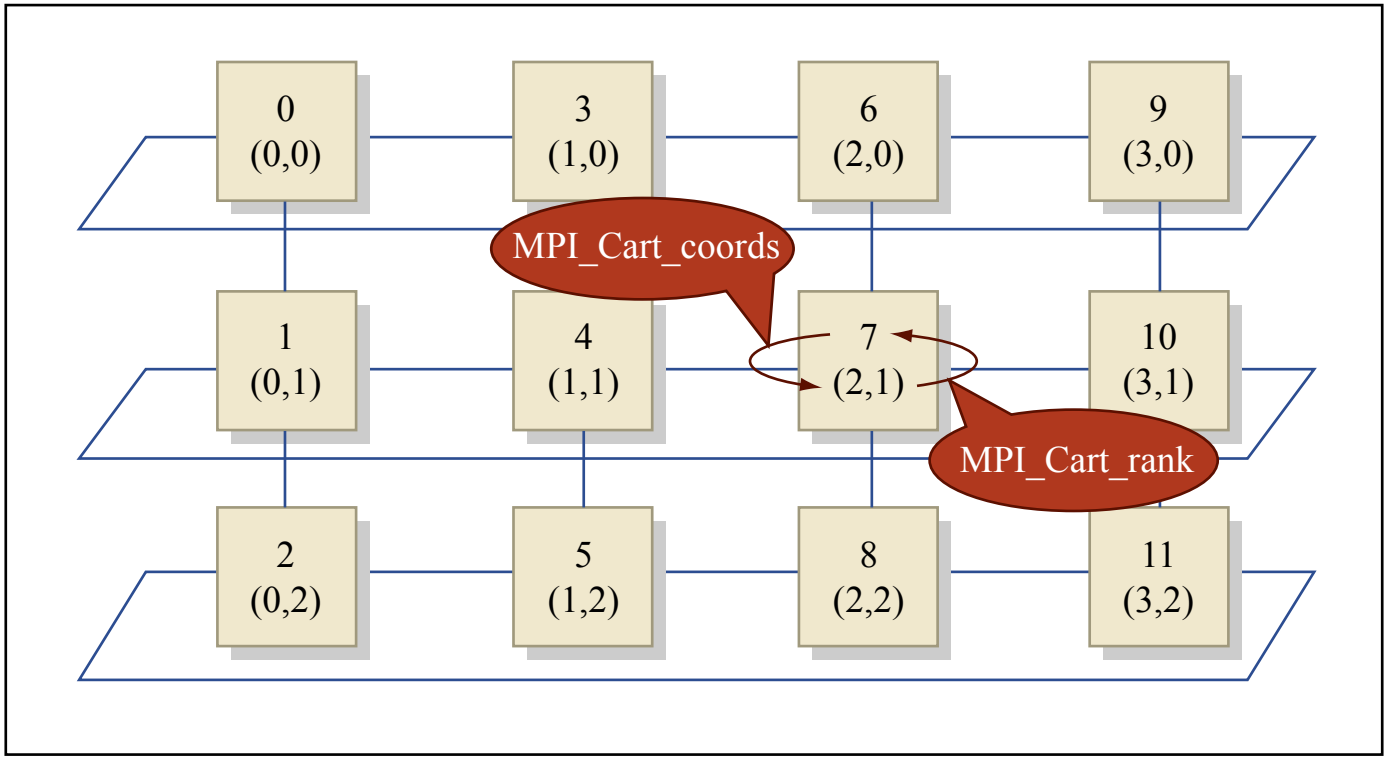


Figure by MIT OpenCourseWare.

Cartesian shift

- MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)
  - MPI_PROC_NULL for shifts at non-periodic boundaries

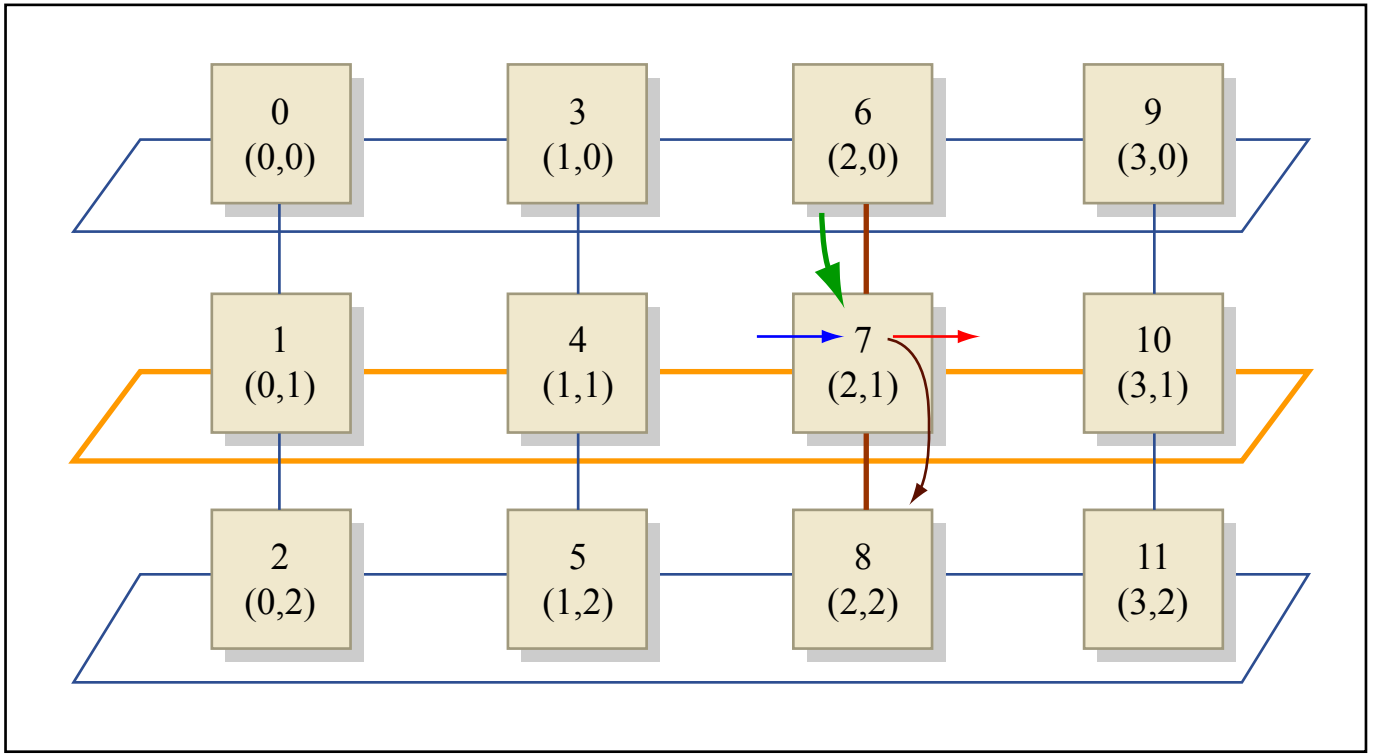


Figure by MIT OpenCourseWare.

Cartesian subspaces

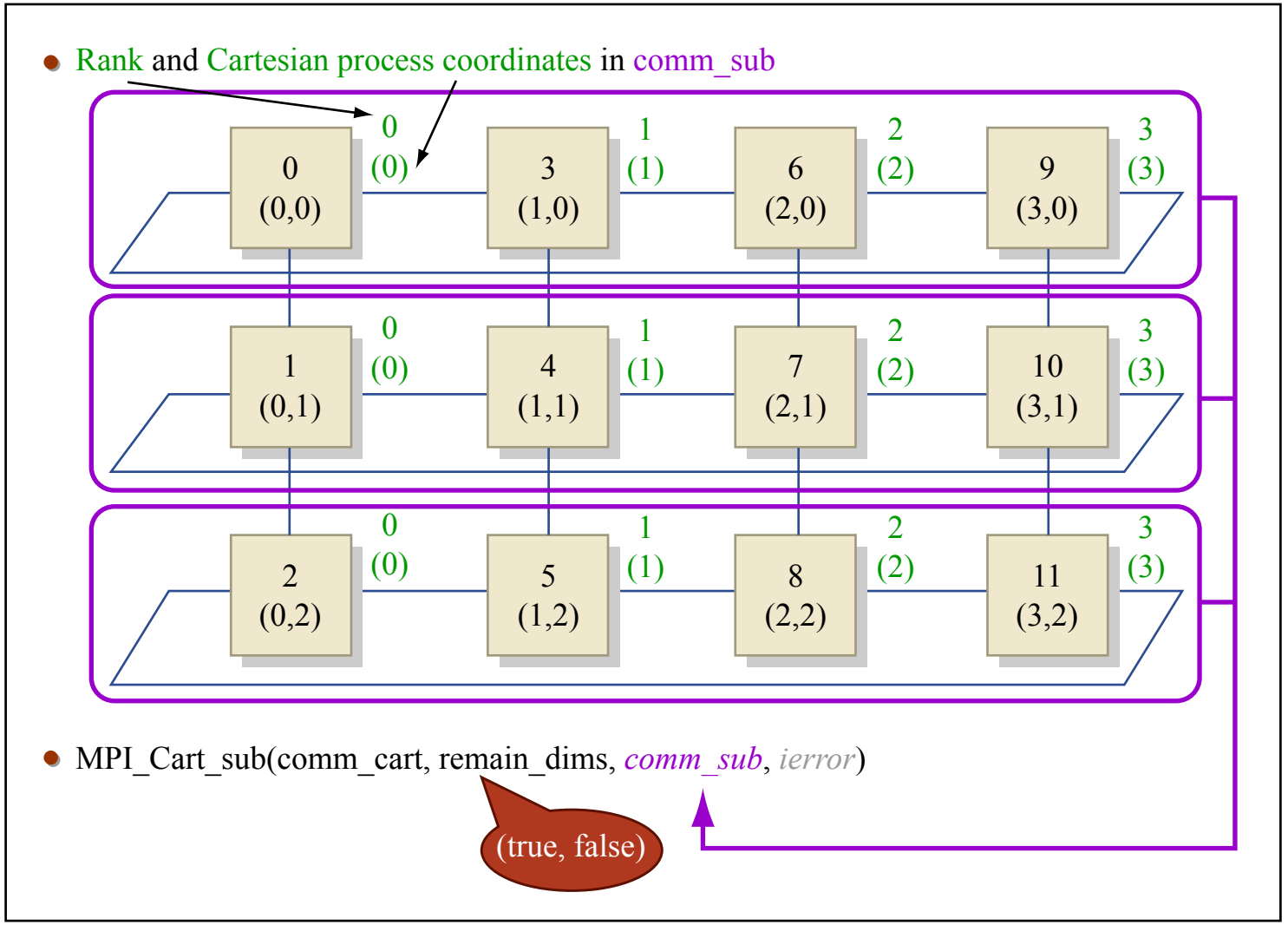


Figure by MIT OpenCourseWare.

More functions

- MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)
- MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors, int *neighbors)
- Used in that order to get the neighbors of a process in a graph.

Fortran binding issues

call MPI_GET_ADDRESS(buf,bufaddr, ierror) ditto...

call MPI_TYPE_CREATE_STRUCT(1,1, bufaddr,MPI_REAL,type,ierror) ditto...

call MPI_TYPE_COMMIT(type,ierror) ditto...

val_old = buf **register = buf**

val_old = register

call MPI_RECV(MPI_BOTTOM,1,type,...) ditto...

val_new = buf **val_new = register**

call MPI_IRECV(buf,..req) call MPI_IRECV(buf,..req) call MPI_IRECV(buf,..req)

register = buf **b1 = buf**

call MPI_WAIT(req,..) call MPI_WAIT(req,..) call MPI_WAIT(req,..)

b1 = buf **b1 = register**

Further Fortran issues

Basic vs. Extended Fortran Support
Strong typing in F90 a problem with choice args
A scalar should not be passed instead of a vector.
Extra work to code with KIND numerical types
MPI_IRECV(buf(a:b:c), ...)
Fortran derived datatypes require MPI equivalents
Problems with input arguments that are copied...

- e.g. MPI_Recv with a buffer that was passed to the parent subroutine as a section or an assumed shape array argument that is associated with such a section.

The MPI runtime

- Provides for process placement, execution & handling
- Handles signals (SIGKILL, SIGSUSP, SIGUSR1/2)
- Usually collects stdout and stderr, may propagate stdin
- May propagate environment variables
- May provide support for debugging, profiling, tracing
- May interface with a queuing system for better process placement
- MPI-2 specifies *(but doesn't require)* standardized mpirun clone: mpiexec. Others: *poe*, *mpprun*, *prun*...
- Command line arguments and/or environment variables allow for different behavior/performance

MPI environment

- Initialize, Finalize and Abort functionality
- Error (exception) handling
- Other inquiry functions:
  - double MPI_Wtime(void), double MPI_Wtick(void)
    - MPI_WTIME_IS_GLOBAL
  - MPI_Get_processor_name(char *name, int *resultlen)
- MPI communicator inquiry (size, rank) for MPI_COMM_WORLD

Exceptions

- Use exceptions and MPI return codes!
- Default error handler: MPI_ERRORS_ARE_FATAL
  - The handler, when called, causes the program to abort on all executing processes. This has the same effect as if MPI_ABORT was called by the process that invoked the handler.
- Alternative: MPI_ERRORS_RETURN
  - The handler has no effect other than returning the error code to the user. Put checks for the error codes in your source!
  - MPICH provides two more:
    - MPE_Errors_call_dbx_in_xterm, MPE_Signals_call_debugger

Error Handling

- Environment Error handling routines:
  - MPI_Errhandler_create(MPI_Handler_function *function, MPI_Errhandler *errhandler)
  - MPI_Errhandler_set(MPI_Comm comm, MPI_Errhandler errhandler)
  - MPI_Errhandler_get(MPI_Comm comm, MPI_Errhandler *errhandler)
  - MPI_Errhandler_free(MPI_Errhandler *errhandler)
  - MPI_Error_string(int errorcode, char *string, int *resultlen)
  - MPI_Error_class(int errorcode, int *errorclass)

The MPI Profiling Interface

- The MPI standard takes great pains to offer a specification for a useful profiling interface that does has minimum overhead and high flexibility.
- All MPI calls have a shifted name of PMPI_... instead of MPI_...
- A profiling library can write it's own MPI_... call, calling the corresponding PMPI_... call to actually do the message passing.
- This provides a way to trace as well as profile in terms of cost in time a parallel program's execution for performance or debugging reasons.

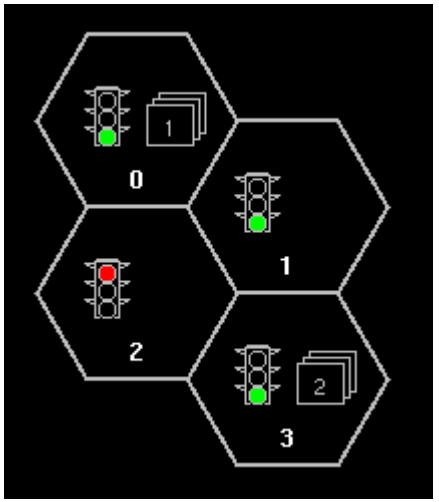
MPI Profiling & Performance Tools

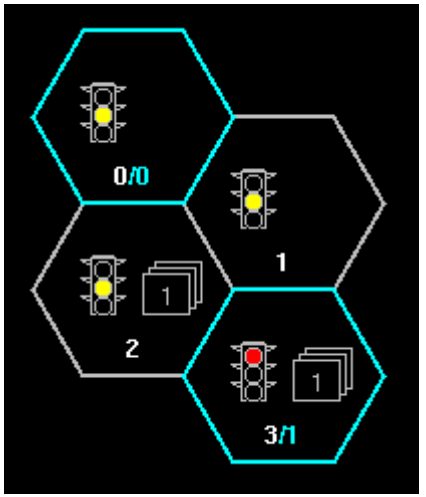
- An extremely wide choice of tools:
  - Research codes:
    - AIMS (NASA Ames)
    - *(sv)Pablo (UIUC)*
    - *Paradyn/Dyninst (University of Wisconsin)*
    - *TAU (University of Oregon)*
    - XMPI (Indiana University)
    - MPE/Jumpshot (ANL)
    - Paragraph/MPICL
    - FPMPI
    - Also lightweight statistics tools: mpiP, ipm
  - Commercial tools (VT, speedshop, Intel Trace A/C, **VAMPIR**)

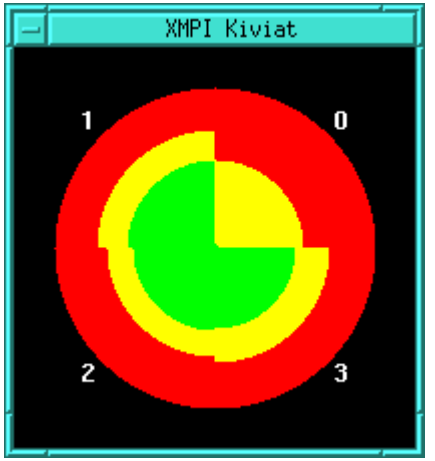
XMPI

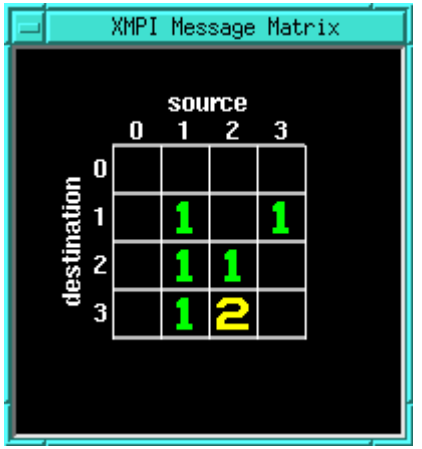
- Works with LAM/MPI, could work with other implementations.
- A GUI for launching MPI parallel programs, monitoring them in real time and also do post-mortem analysis on them.
- Uses the slower "daemon" mode of LAM, provides individual message detail and has multiple views. The daemon mode allows cmdline tracing tools mpimsg and mpitask to be more informative
- Very easy to use but non-daemon mode is required for performance tuning. Launch with **-ton** and collect tracefile with **lamtrace**

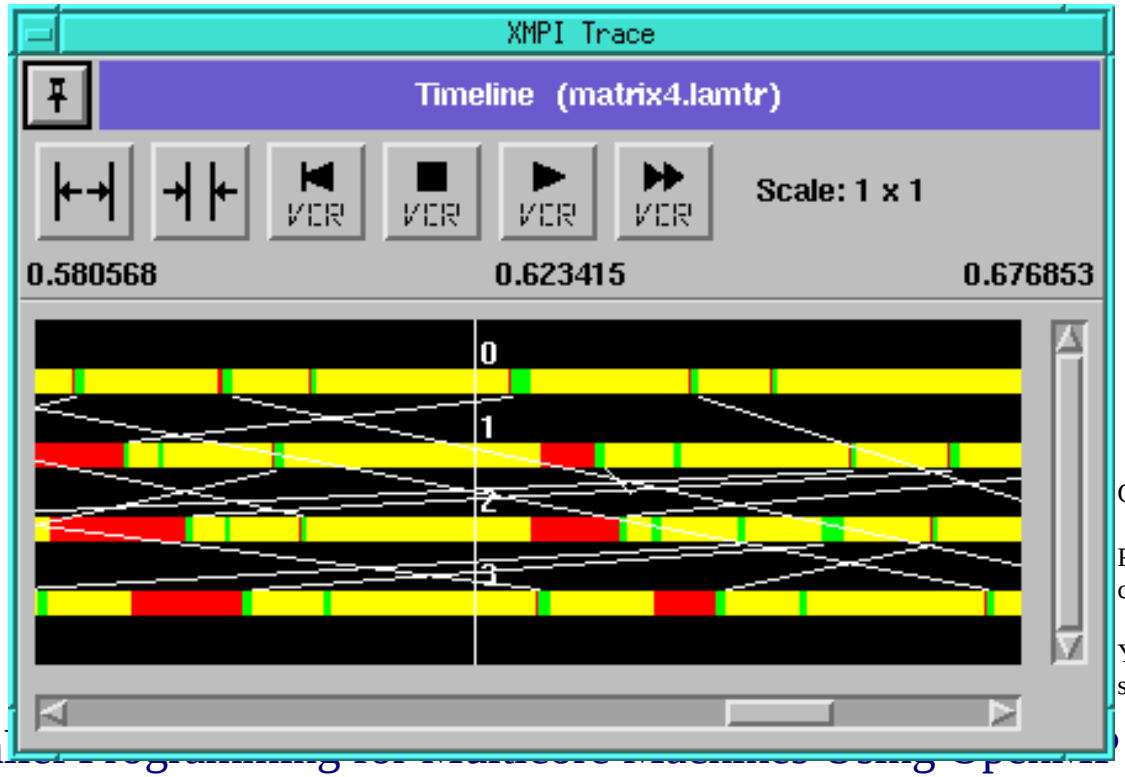
XMPI in action

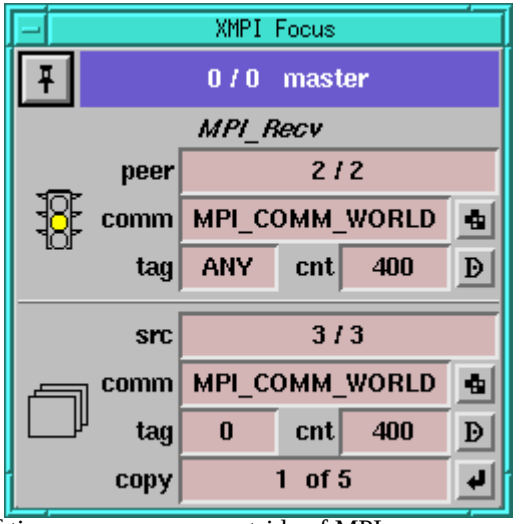












Green: Represents the length of time a process runs outside of MPI.

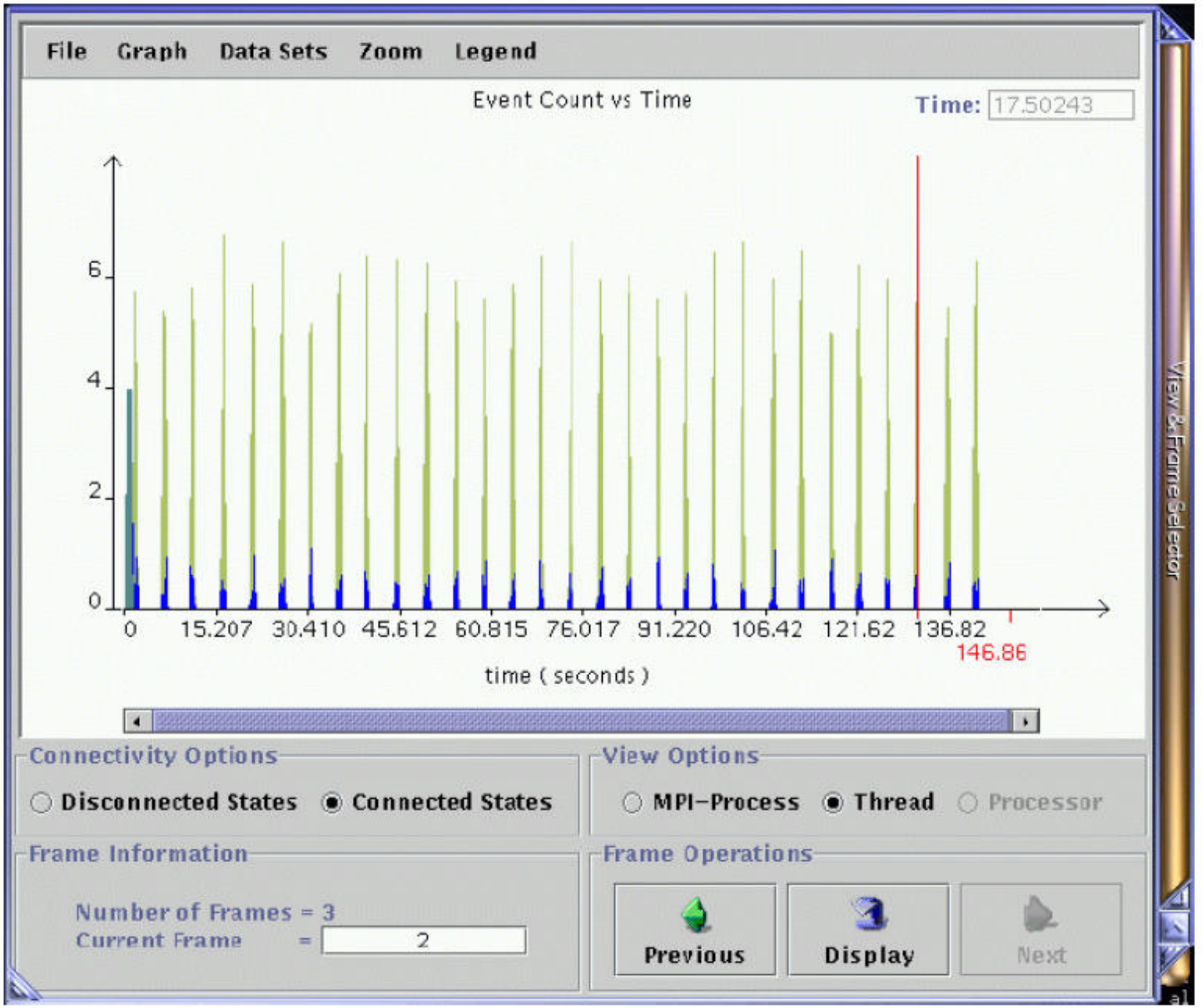
Red: Represents the length of time a process is blocked, waiting for communication to finish before the process resumes execution.

Yellow: Represents a process's overhead time inside MPI (for example, time spent doing message packing).

MPE (from MPICH/MPICH2)

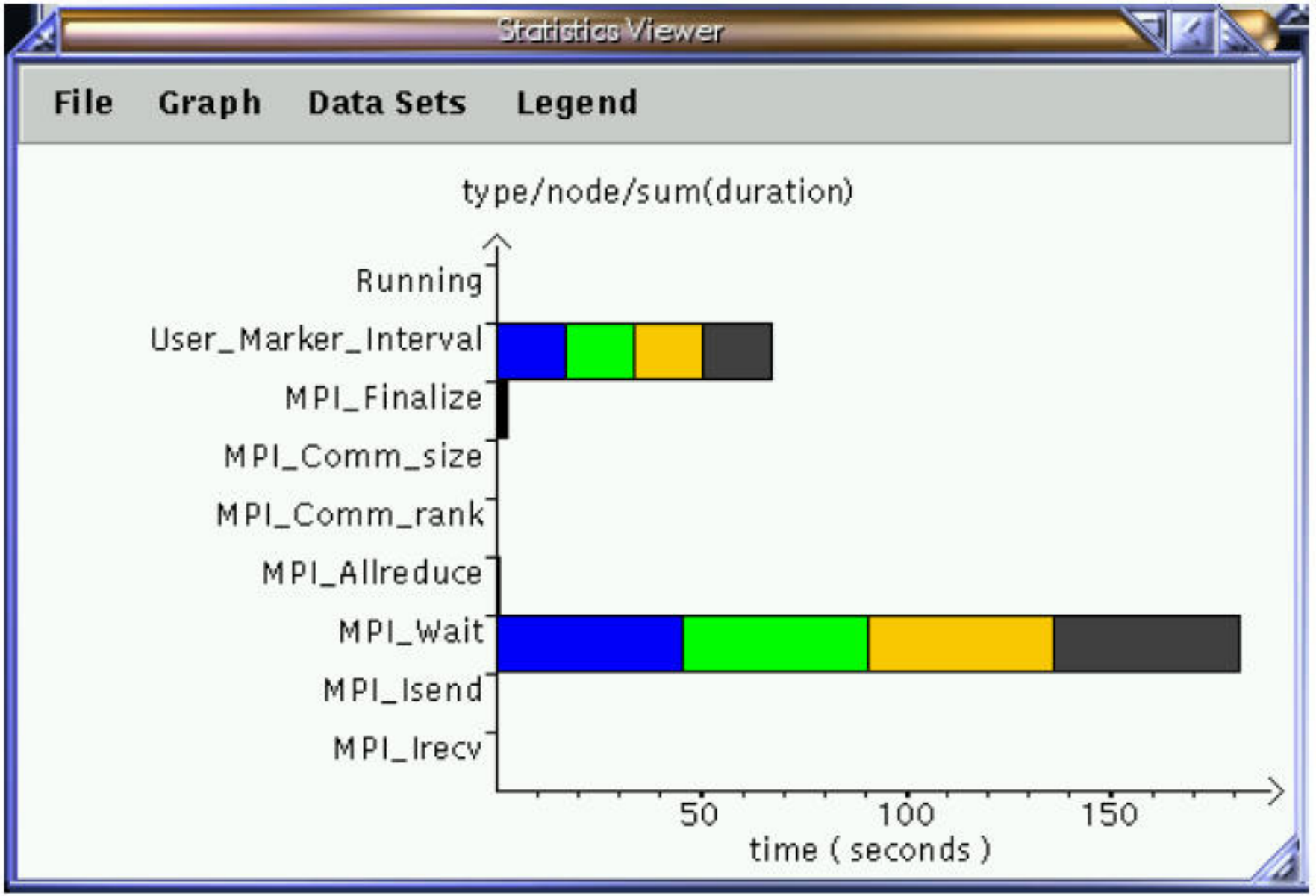
- Set of utility routines, including graphics
- Graphical viewing of traces with (n)upshot, jumpshot
- Compile with -mpe=mpitrace to enable basic tracing
  - A message printed to stdout at every entry and exit
- Compile with -mpe=mpilog to enable logging
  - ALOG, CLOG, CLOG2, UTE, SLOG and SLOG2 format
  - Converters between formats (eg. Clog2slog2)
- SLOG2 is the newest and most scalable
- Jumpshot-4 is needed for looking at SLOG2 files

Jumpshot

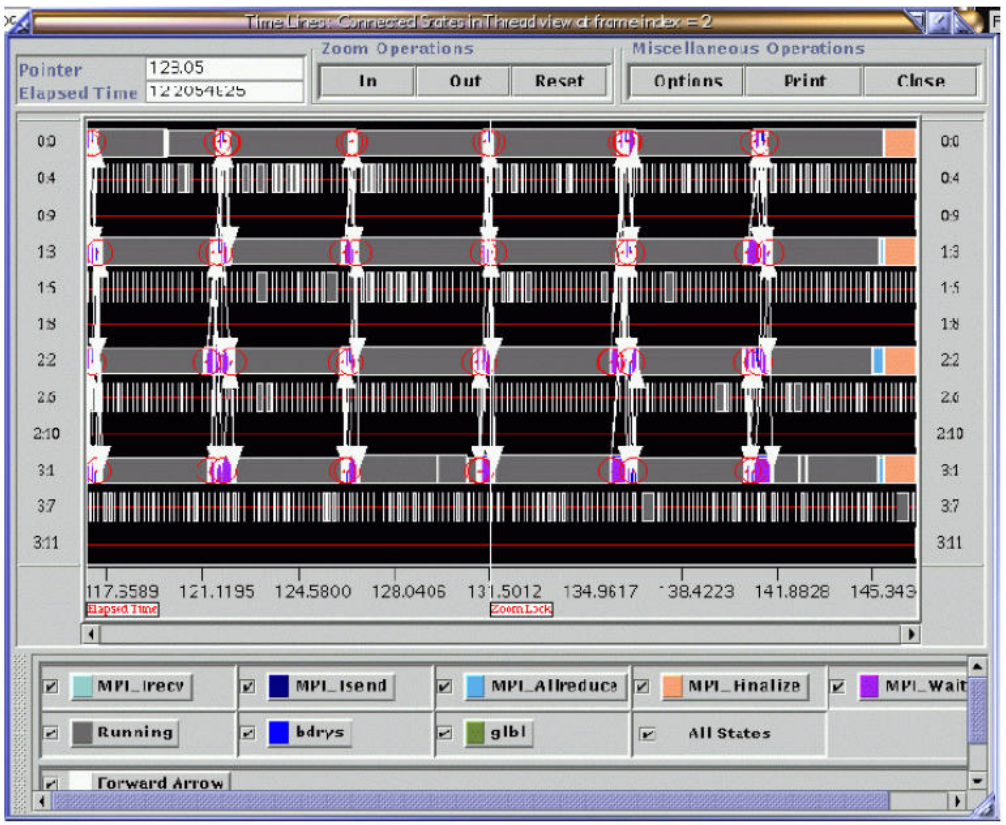

File
Graph
Data Sets
Zoom
Legend
Event Count vs Time
Time: 17.50243
6
4
2
0
0
15.207
30.410
45.612
60.815
76.017
91.220
106.42
121.62
136.82
146.86
time (seconds)
View & Frame Selector
Connectivity Options
Disconnected States
Connected States
View Options
MPI-Process
Thread
Processor
Frame Information
Number of Frames = 3
Current Frame = 2
Frame Operations
Previous
Display
Next

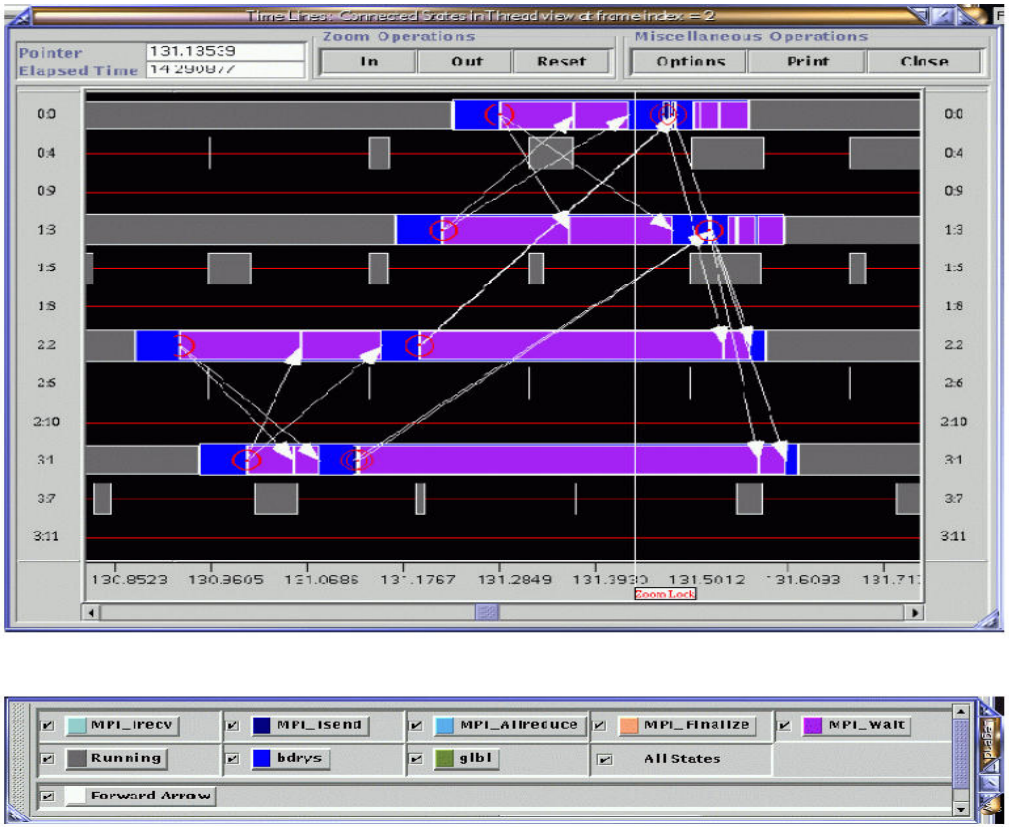

MPI_Irecv
MPI_Isend
MPI_Allreduce
MPI_Comm_r...
MPI_Comm_s...
MPI_Finalize
MPI_Wait
Running
layout
setup
bdrys
glbl
Forward Arrow
Select/Deselect
All
None
Change Color
Legend

More Jumpshot



Even more Jumpshot





12.950 Parallel Programming for Multicore Machines Using OpenMP and MPI

IAP 2010