Message Passing and MPI

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@rice.edu

Topics for Today

- Principles of message passing
  - —building blocks (send, receive)
- MPI: Message Passing Interface
- Overlapping communication with computation
- Topologies
- Collective communication and computation
- Groups and communicators
- MPI derived data types
- Threading
- Remote Memory Access (RMA)
- Using MPI
- MPI Resources

Message Passing Overview

- The logical view of a message-passing platform
  - —p processes
  - —each with its own exclusive address space
- All data must be explicitly partitioned and placed
- All interactions (read-only or read/write) are two-sided
  - —process that has the data sends it
  - —process that wants the data receives it
- Typically use single program multiple data (SPMD) model
- The bottom line …
  - —strengths
    - – simple performance model: underlying costs are explicit
    - – portable high performance
  - —weakness: two-sided model can be awkward to program

Pros and Cons of Message Passing

- **Advantages**
  - —universality
    - works well on machines with fast or slow network
  - —expressivity
    - useful and complete model for expressing parallel algorithms
  - —lack of data races
    - data races are a pervasive problem for shared memory programs*
  - —performance
    - yields high performance by co-locating data with computation
- **Disadvantages**
  - —managing partitioned address spaces is a hassle
  - —two-sided communication is somewhat awkward to write
  - —debugging multiple is awkward

* MPI is not devoid of race conditions, but they can only occur when non-blocking operations are used

Message Passing Flavors: Blocking, Unbuffered

- **Definition**
  - —send won't return until its data has been transferred
  - —receive won't return until data has arrived
  - —no copy is made of the message data
- **Advantage:**
  - —simple to use
- **Disadvantages**
  - —send and recv may idle, awaiting partner
  - —deadlock is possible since send operations block

```
Processor 0

send x to proc 1

receive y from proc 1
```

```
Processor 1

send y to proc 0

receive x from proc 0
```

Message Passing Flavors: Blocking, Buffered

- **Definition**
  - send won't return until its data may be overwritten
    - may return after data copied into a buffer at sender
    - data may be delivered early into a buffer at receiver
  - receive won't return until the data has arrived
- **Advantages**
  - simple to use
  - avoids deadlock caused by send
- **Disadvantages**
  - receive may idle, awaiting a send
  - deadlock still possible since receive operations block

```
Processor 0

receive y from proc 1

send x to proc 1
```

```
Processor 1

receive x from proc 0

send y to proc 0
```

Buffered Blocking Message Passing

Buffer sizes can have significant impact on performance

```
Processor 0

for (i = 0; i < N; i++)
    produce_data(&a)
    send a to proc 1
```

```
Processor 1

for (i = 0; i < N; i++)
    receive a from proc 0
    consume_data(&a)
```

Larger buffers enable the computation to tolerate asynchrony better

Message Passing Flavors: Non-Blocking

- **Definition**
  - send and receive return before it is safe
    - sender: data can be overwritten before it is sent
    - receiver: can read data out of buffer before it is received
  - ensuring proper usage is the programmer's responsibility
  - status check operation to ascertain completion
- **Advantages**
  - tolerate asynchrony
  - no costly copies or buffer space
  - overlap communication with computation
- **Disadvantage**
  - programming complexity

```
Processor 0
start_send x to proc 1
start_recv y from proc 1
. . .
end_send x to proc 1
end_recv y from proc 1
```

```
Processor 1
start_send y to proc 0
start_recv x from proc 0
. . .
end_send y to proc 0
end_recv x from proc 0
```

MPI: the Message Passing Interface

- **Standard library for message-passing**
  - —portable
  - —almost ubiquitously available
  - —high performance
  - —C and Fortran APIs
- **MPI standard defines**
  - —syntax of library routines
  - —semantics of library routines
- **Details**
  - —MPI routines, data-types, and constants are prefixed by “MPI_”
- **Simple to get started**
  - —fully-functional programs using only six library routines

Scope of the MPI Standards

- Communication contexts
- Datatypes
- Point-to-point communication
- Collective communication (synchronous, non-blocking)
- Process groups
- Process topologies
- Environmental management and inquiry
- The Info object
- Process creation and management
- One-sided communication (refined for MPI-3)
- External interfaces
- Parallel I/O
- Language bindings for Fortran, C and C++
- Profiling interface (PMPI)

MPI
MPI-2
MPI-3

MPI Primitives at a Glance

MPIX_Comm_agree
MPIX_Comm_failure_ack
MPIX_Comm_failure_get_acked
MPIX_Comm_revoke
MPIX_Comm_shrink
MPI_Abort
MPI_Accumulate
MPI_Add_error_class
MPI_Add_error_code
MPI_Add_error_string
MPI_Address
MPI_Aint_add
MPI_Aint_diff
MPI_Allgather
MPI_Allgatherv
MPI_Alloc_mem
MPI_Allreduce
MPI_Alltoall
MPI_Alltoallv
MPI_Alltoallw
MPI_Attr_delete
MPI_Attr_get
MPI_Attr_put
MPI_Barrier
MPI_Bcast
MPI_Bsend
MPI_Bsend_init
MPI_Buffer_attach
MPI_Buffer_detach
MPI_Cancel
MPI_Cart_coords
MPI_Cart_create
MPI_Cart_get
MPI_Cart_map
MPI_Cart_rank
MPI_Cart_shift
MPI_Cart_sub
MPI_Cartdim_get
MPI_Close_port
MPI_Comm_accept
MPI_Comm_call_errhandler
MPI_Comm_compare
MPI_Comm_connect
MPI_Comm_create
MPI_Comm_create_errhandler
MPI_Comm_create_group
MPI_Comm_create_keyval
MPI_Comm_delete_attr
MPI_Comm_disconnect
MPI_Comm_dup
MPI_Comm_dup_with_info

MPI_File_set_errhandler
MPI_File_set_info
MPI_File_set_size
MPI_File_set_view
MPI_File_sync
MPI_File_write
MPI_File_write_all
MPI_File_write_all_begin
MPI_File_write_all_end
MPI_File_write_at
MPI_File_write_at_all
MPI_File_write_at_all_begin
MPI_File_write_at_all_end
MPI_File_write_ordered
MPI_File_write_ordered_begin
MPI_File_write_ordered_end
MPI_File_write_shared
MPI_Finalize
MPI_Finalized
MPI_Free_mem
MPI_Gather
MPI_Gatherv
MPI_Get
MPI_Get_accumulate
MPI_Get_address
MPI_Get_count
MPI_Get_elements
MPI_Get_elements_x
MPI_Get_library_version
MPI_Get_processor_name
MPI_Get_version
MPI_Graph_create
MPI_Graph_get
MPI_Graph_map
MPI_Graph_neighbors
MPI_Graph_neighbors_count
MPI_Graphdims_get
MPI_Grequest_complete
MPI_Grequest_start
MPI_Group_compare
MPI_Group_difference
MPI_Group_excl
MPI_Group_free
MPI_Group_incl
MPI_Group_intersection
MPI_Group_range_excl
MPI_Group_range_incl
MPI_Group_rank
MPI_Group_size
MPI_Group_translate_ranks
MPI_Group_union

MPI_Rsend
MPI_Rsend_init
MPI_Scan
MPI_Scatter
MPI_Scatterv
MPI_Send
MPI_Send_init
MPI_Sendrecv
MPI_Sendrecv_replace
MPI_Ssend
MPI_Ssend_init
MPI_Start
MPI_Startall
MPI_Status_set_cancelled
MPI_Status_set_elements
MPI_Status_set_elements_x
MPI_T_category_changed
MPI_T_category_get_categories
MPI_T_category_get_cvars
MPI_T_category_get_info
MPI_T_category_get_num
MPI_T_category_get_pvars
MPI_T_cvar_get_info
MPI_T_cvar_get_num
MPI_T_cvar_handle_alloc
MPI_T_cvar_handle_free
MPI_T_cvar_read
MPI_T_cvar_write
MPI_T_enum_get_info
MPI_T_enum_get_item
MPI_T_finalize
MPI_T_init_thread
MPI_T_pvar_get_info
MPI_T_pvar_get_num
MPI_T_pvar_handle_alloc
MPI_T_pvar_handle_free
MPI_T_pvar_read
MPI_T_pvar_readreset
MPI_T_pvar_reset
MPI_T_pvar_session_create
MPI_T_pvar_session_free
MPI_T_pvar_start
MPI_T_pvar_stop
MPI_T_pvar_write
MPI_Test
MPI_Test_cancelled
MPI_Testall
MPI_Testany
MPI_Testsome
MPI_Topo_test
MPI_Type_commit

MPI_Comm_free
MPI_Comm_free_keyval
MPI_Comm_get_attr
MPI_Comm_get_errhandler
MPI_Comm_get_info
MPI_Comm_get_name
MPI_Comm_get_parent
MPI_Comm_group
MPI_Comm_idup
MPI_Comm_join
MPI_Comm_rank
MPI_Comm_remote_group
MPI_Comm_remote_size
MPI_Comm_set_attr
MPI_Comm_set_errhandler
MPI_Comm_set_info
MPI_Comm_set_name
MPI_Comm_size
MPI_Comm_spawn
MPI_Comm_spawn_multiple
MPI_Comm_split
MPI_Comm_split_type
MPI_Comm_test_inter
MPI_Compare_and_swap
MPI_Dims_create
MPI_Dist_graph_create
MPI_Dist_graph_create_adjacent
MPI_Dist_graph_neighbors
MPI_Dist_graph_neighbors_count
MPI_Errhandler_create
MPI_Errhandler_free
MPI_Errhandler_get
MPI_Errhandler_set
MPI_Error_class
MPI_Error_string
MPI_Exscan
MPI_Fetch_and_op
MPI_File_c2f
MPI_File_call_errhandler
MPI_File_close
MPI_File_create_errhandler
MPI_File_delete
MPI_File_f2c
MPI_File_get_amode
MPI_File_get_atomicity
MPI_File_get_byte_offset
MPI_File_get_errhandler
MPI_File_get_group
MPI_File_get_info
MPI_File_get_position
MPI_File_get_position_shared
MPI_File_get_size
MPI_File_get_type_extent
MPI_File_get_view
MPI_File_iread
MPI_File_iread_all
MPI_File_iread_at
MPI_File_iread_at_all
MPI_File_iread_shared
MPI_File_iwrite
MPI_File_iwrite_all
MPI_File_iwrite_at

MPI_Iallgather
MPI_Iallgatherv
MPI_Iallreduce
MPI_Ialltoall
MPI_Ialltoallv
MPI_Ialltoallw
MPI_Ibarrier
MPI_Ibcast
MPI_Ibsend
MPI_Iexscan
MPI_Igather
MPI_Igatherv
MPI_Improbe
MPI_Imrecv
MPI_Ineighbor_allgather
MPI_Ineighbor_allgatherv
MPI_Ineighbor_alltoall
MPI_Ineighbor_alltoallv
MPI_Ineighbor_alltoallw
MPI_Info_create
MPI_Info_delete
MPI_Info_dup
MPI_Info_free
MPI_Info_get
MPI_Info_get_nkeys
MPI_Info_get_nthkey
MPI_Info_get_valuelen
MPI_Info_set
MPI_Init
MPI_Init_thread
MPI_Initialized
MPI_Intercomm_create
MPI_Intercomm_merge
MPI_Iprobe
MPI_Irecv
MPI_Ireduce
MPI_Ireduce_scatter
MPI_Ireduce_scatter_block
MPI_Irsend
MPI_Is_thread_main
MPI_Iscan
MPI_Iscatter
MPI_Iscatterv
MPI_Isend
MPI_Issend
MPI_Keyval_create
MPI_Keyval_free
MPI_Lookup_name
MPI_Mprobe
MPI_Mrecv
MPI_Neighbor_allgather
MPI_Neighbor_allgatherv
MPI_Neighbor_alltoall
MPI_Neighbor_alltoallv
MPI_Neighbor_alltoallw
MPI_Op_commute
MPI_Op_create
MPI_Op_free
MPI_Open_port
MPI_Pack
MPI_Pack_external
MPI_Pack_external_size

MPI_Type_contiguous
MPI_Type_create_darray
MPI_Type_create_hindexed
MPI_Type_create_hindexed_block
MPI_Type_create_hvector
MPI_Type_create_indexed_block
MPI_Type_create_keyval
MPI_Type_create_resized
MPI_Type_create_struct
MPI_Type_create_subarray
MPI_Type_delete_attr
MPI_Type_dup
MPI_Type_extent
MPI_Type_free
MPI_Type_free_keyval
MPI_Type_get_attr
MPI_Type_get_contents
MPI_Type_get_envelope
MPI_Type_get_extent
MPI_Type_get_extent_x
MPI_Type_get_name
MPI_Type_get_true_extent
MPI_Type_get_true_extent_x
MPI_Type_hindexed
MPI_Type_hvector
MPI_Type_indexed
MPI_Type_lb
MPI_Type_match_size
MPI_Type_set_attr
MPI_Type_set_name
MPI_Type_size
MPI_Type_size_x
MPI_Type_struct
MPI_Type_ub
MPI_Type_vector
MPI_Unpack
MPI_Unpack_external
MPI_Unpublish_name
MPI_Wait
MPI_Waitall
MPI_Waitany
MPI_Waitsome
MPI_Win_allocate
MPI_Win_allocate_shared
MPI_Win_attach
MPI_Win_call_errhandler
MPI_Win_complete
MPI_Win_create
MPI_Win_create_dynamic
MPI_Win_create_errhandler
MPI_Win_create_keyval
MPI_Win_delete_attr
MPI_Win_detach
MPI_Win_fence
MPI_Win_flush
MPI_Win_flush_all
MPI_Win_flush_local
MPI_Win_flush_local_all
MPI_Win_free
MPI_Win_free_keyval
MPI_Win_get_attr
MPI_Win_get_errhandler

MPI_File_get_size
MPI_File_get_type_extent
MPI_File_get_view
MPI_File_iread
MPI_File_iread_all
MPI_File_iread_at
MPI_File_iread_at_all
MPI_File_iread_shared
MPI_File_iwrite
MPI_File_iwrite_all
MPI_File_iwrite_at
MPI_File_iwrite_at_all
MPI_File_iwrite_shared
MPI_File_open
MPI_File_preallocate
MPI_File_read
MPI_File_read_all
MPI_File_read_all_begin
MPI_File_read_all_end
MPI_File_read_at
MPI_File_read_at_all
MPI_File_read_at_all_begin
MPI_File_read_at_all_end
MPI_File_read_ordered
MPI_File_read_ordered_begin
MPI_File_read_ordered_end
MPI_File_read_shared
MPI_File_seek
MPI_File_seek_shared
MPI_File_set_atomicity

MPI_Neighbor_allgatherv
MPI_Neighbor_alltoall
MPI_Neighbor_alltoallv
MPI_Neighbor_alltoallw
MPI_Op_commute
MPI_Op_create
MPI_Op_free
MPI_Open_port
MPI_Pack
MPI_Pack_external
MPI_Pack_external_size
MPI_Pack_size
MPI_Pcontrol
MPI_Probe
MPI_Publish_name
MPI_Put
MPI_Query_thread
MPI_Raccumulate
MPI_Recv
MPI_Recv_init
MPI_Reduce
MPI_Reduce_local
MPI_Reduce_scatter
MPI_Reduce_scatter_block
MPI_Register_datarep
MPI_Request_free
MPI_Request_get_status
MPI_Rget
MPI_Rget_accumulate
MPI_Rput

MPI_Win_delete_attr
MPI_Win_detach
MPI_Win_fence
MPI_Win_flush
MPI_Win_flush_all
MPI_Win_flush_local
MPI_Win_flush_local_all
MPI_Win_free
MPI_Win_free_keyval
MPI_Win_get_attr
MPI_Win_get_errhandler
MPI_Win_get_group
MPI_Win_get_info
MPI_Win_get_name
MPI_Win_lock
MPI_Win_lock_all
MPI_Win_post
MPI_Win_set_attr
MPI_Win_set_errhandler
MPI_Win_set_info
MPI_Win_set_name
MPI_Win_shared_query
MPI_Win_start
MPI_Win_sync
MPI_Win_test
MPI_Win_unlock
MPI_Win_unlock_all
MPI_Win_wait
MPI_Wtick
MPI_Wtime

http://www.mpich.org/static/docs/latest/www3

MPI: the Message Passing Interface

Minimal set of MPI routines

`MPI_Init`	initialize MPI
`MPI_Finalize`	terminate MPI
`MPI_Comm_size`	determine number of processes in group
`MPI_Comm_rank`	determine id of calling process in group
`MPI_Send`	send message
`MPI_Recv`	receive message

Starting and Terminating the MPI Programs

- `int MPI_Init(int *argc, char ***argv)`
  - —initialization: must call this prior to other MPI routines
  - —effects
    - – strips off and processes any MPI command-line arguments
    - – initializes MPI environment
- `int MPI_Finalize()`
  - —must call at the end of the computation
  - —effect
    - – performs various clean-up tasks to terminate MPI environment
- Return codes
  - —MPI_SUCCESS
  - —MPI_ERROR

Communicators

- `MPI_Comm`: communicator = communication domain
  —group of processes that can communicate with one another
- Supplied as an argument to all MPI message transfer routines
- Process can belong to multiple communication domains
  —domains may overlap
- `MPI_COMM_WORLD`: root communicator
  — includes all the processes

Communicator Inquiry Functions

- `int MPI_Comm_size(MPI_Comm comm, int *size)`
  —determine the number of processes
- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
  —index of the calling process
  —$0 \le$ rank $<$ communicator size

“Hello World” Using MPI

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
        int npes, myrank;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &npes);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        printf("From process %d out of %d, Hello World!\n",
                myrank, npes);
        MPI_Finalize();
        return 0;
}
```

Sending and Receiving Messages

- ```
  int MPI_Send(void *buf, int count, MPI_Datatype datatype,
               int dest_pe, int tag, MPI_Comm comm)
  ```

- ```
  int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
              int source_pe, int tag, MPI_Comm comm,
              MPI_Status *status)
  ```

- **Message source or destination PE**
  - —index of process in the communicator `comm`
  - —receiver wildcard: `MPI_ANY_SOURCE`
    - any process in the communicator can be source
- **Message-tag: integer values, 0 ≤ tag < MPI_TAG_UB**
  - —receiver tag wildcard: `MPI_ANY_TAG`
    - messages with any tag are accepted
- **Receiver constraint**
  - — message size ≤ buffer length specified

MPI Primitive Data Types

MPI data type	C data type
`MPI_CHAR`	`signed char`
`MPI_SHORT`	`signed short int`
`MPI_INT`	`signed int`
`MPI_LONG`	`signed long int`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_UNSIGNED_SHORT`	`unsigned short int`
`MPI_UNSIGNED`	`unsigned int`
`MPI_UNSIGNED_LONG`	`unsigned long int`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_LONG_DOUBLE`	`long double`
`MPI_BYTE`	`8 bits`
`MPI_PACKED`	`packed sequence of bytes`

Receiver Status Inquiry

- `Mpi_Status`
  - —stores information about an `MPI_Recv` operation
  - —data structure

```
typedef struct MPI_Status {
  int MPI_SOURCE;
  int MPI_TAG;
  int MPI_ERROR;
};
```

- `int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)`
  - —returns the count of data items received
    - – not directly accessible from status variable

Deadlock with MPI_Send/Recv?

destination

tag

```
int a[10], b[10], myrank;
MPI_Status s1, s2;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD, &s1);
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD, &s2);
}
...
```

Definition of MPI_Send says: “This routine may block until the message is received by the destination process”

Deadlock if MPI_Send is blocking

Another Deadlock Pitfall?

Send data to neighbor to your right on a ring ...

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
        MPI_COMM_WORLD);

MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
         MPI_COMM_WORLD, &status);
...
```

Deadlock if MPI_Send is blocking

Avoiding Deadlock with Blocking Sends

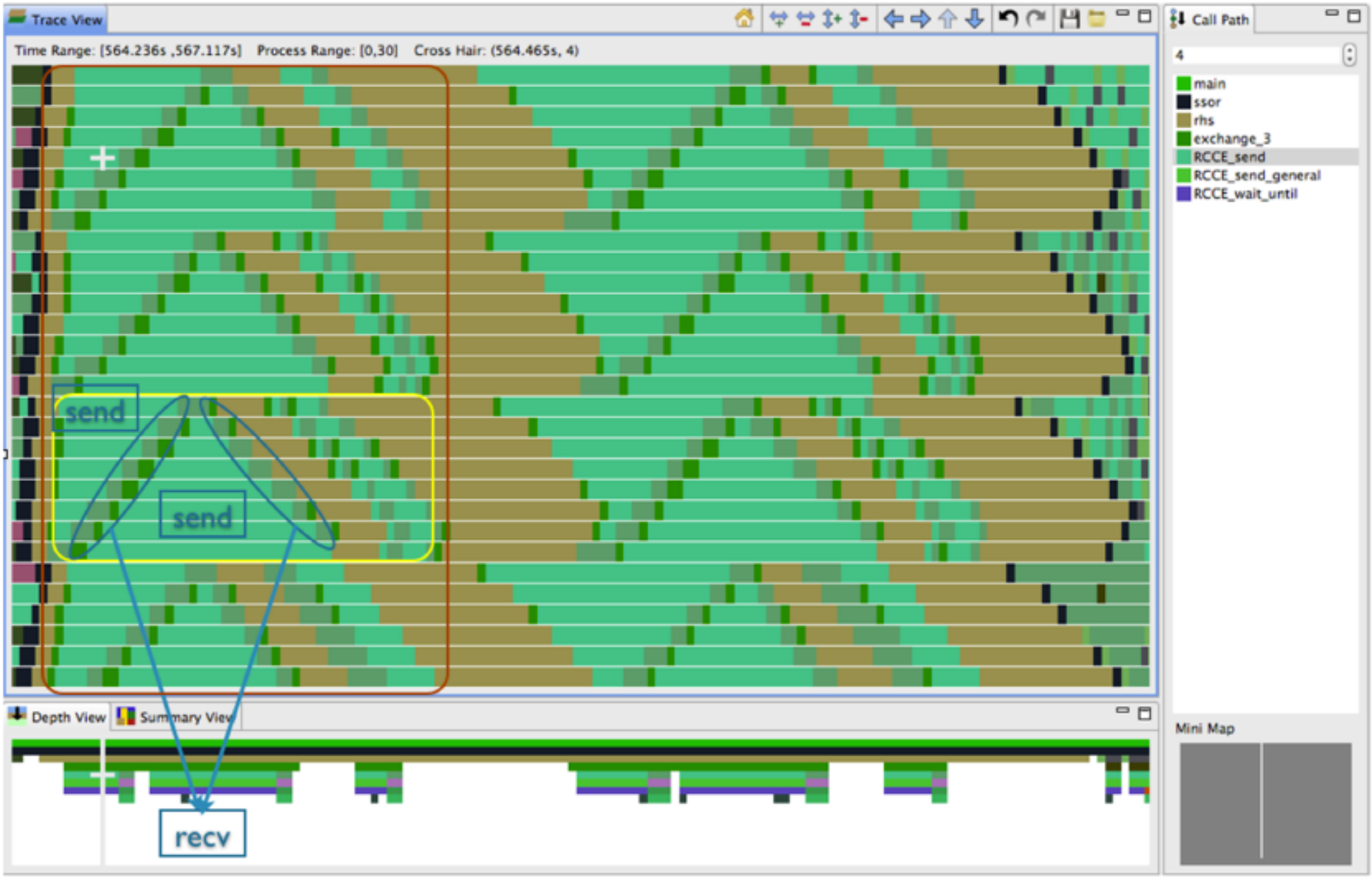
Send data to neighbor to your right on a ring ...

Break the circular wait

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

if (myrank%2 == 1) {    // odd processes send first, receive second
        MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
                MPI_COMM_WORLD);
        MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
                MPI_COMM_WORLD, &status);
}
else {                  // even processes receive first, send second
        MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
                MPI_COMM_WORLD, &status);
        MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
                MPI_COMM_WORLD);
}
...
```

Serialization in NAS LU on Intel SCC



Primitives for Non-blocking Communication

- Non-blocking send and receive return before they complete

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
          int dest, int tag, MPI_Comm comm,
          MPI_Request *request)
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
          int source, int tag, MPI_Comm comm,
          MPI_Request *request)
```

- `MPI_Test`: has a particular non-blocking request finished?

```
int MPI_Test(MPI_Request *request, int *flag,
      MPI_Status *status)
```

- `MPI_Waitany`: block until some request in a set completes

```
int MPI_Wait_any(int req_cnt, MPI_Request *req_array,
                 int *req_index, MPI_Status *status)
```

- `MPI_Wait`: block until a particular request completes

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

Avoiding Deadlocks with NB Primitives

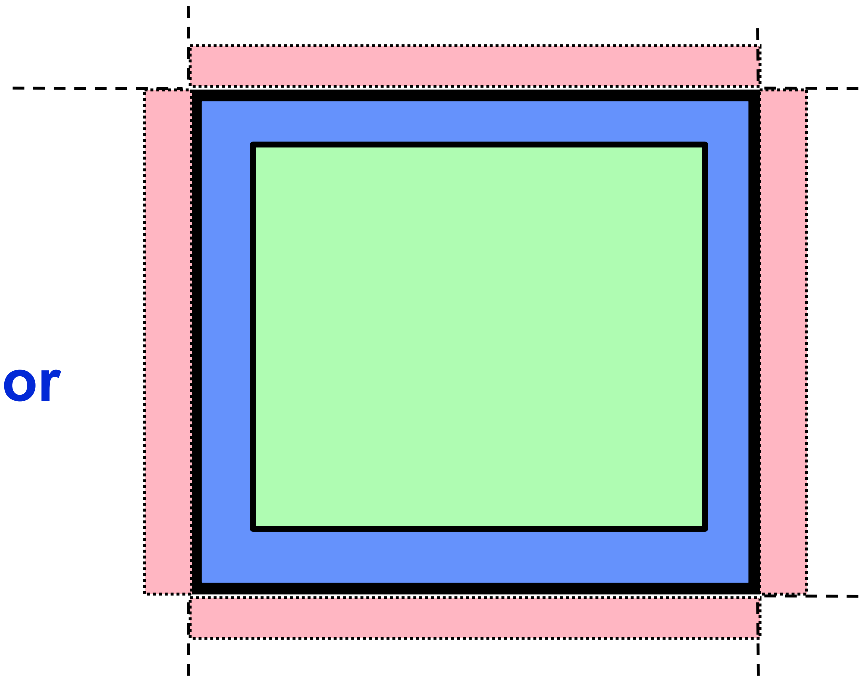
Using non-blocking operations avoids most deadlocks

tag

```
int a[10], b[10], myrank;
MPI_Request r1, r2;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
   MPI_ISend(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD, &r1);
   MPI_ISend(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD, &r2);
}
else if (myrank == 1) {
   MPI_IRecv(b, 10, MPI_INT, 0, 2, &status, MPI_COMM_WORLD, &r1);
   MPI_IRecv(a, 10, MPI_INT, 0, 1, &status, MPI_COMM_WORLD, &r2);
}
...
```

Overlapping Communication Example

- Original
  - —send boundary layer (blue) to neighbors with blocking send
  - —receive boundary layer (pink) from neighbors
  - —compute data volume (green + blue)
- Overlapped
  - —send boundary layer (blue) to neighbor with non-blocking send
  - —compute interior region (green)
  - —receive boundary layer (pink)
  - —wait for non-blocking sends to complete (blue)
  - —compute boundary layer (blue)



Message Exchange

To exchange messages in a single call (both send and receive)

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
  MPI_Datatype senddatatype, int dest, int sendtag,
  void *recvbuf, int recvcount, MPI_Datatype recvdatatype,
  int source, int recvtag, MPI_Comm comm,
  MPI_Status *status)
```

Requires both send and receive arguments

Why Sendrecv?

Sendrecv is useful for executing a shift operation along a chain of processes. If blocking send and recv are used for such a shift, then one needs to avoid deadlock with an odd/even scheme. When Sendrecv is used, MPI handles these issues.

To use same buffer for both send and receive

```
int MPI_Sendrecv_replace(void *buf, int count,
  MPI_Datatype datatype, int dest, int sendtag,
  int source, int recvtag, MPI_Comm comm,
  MPI_Status *status)
```

Collective Communication in MPI

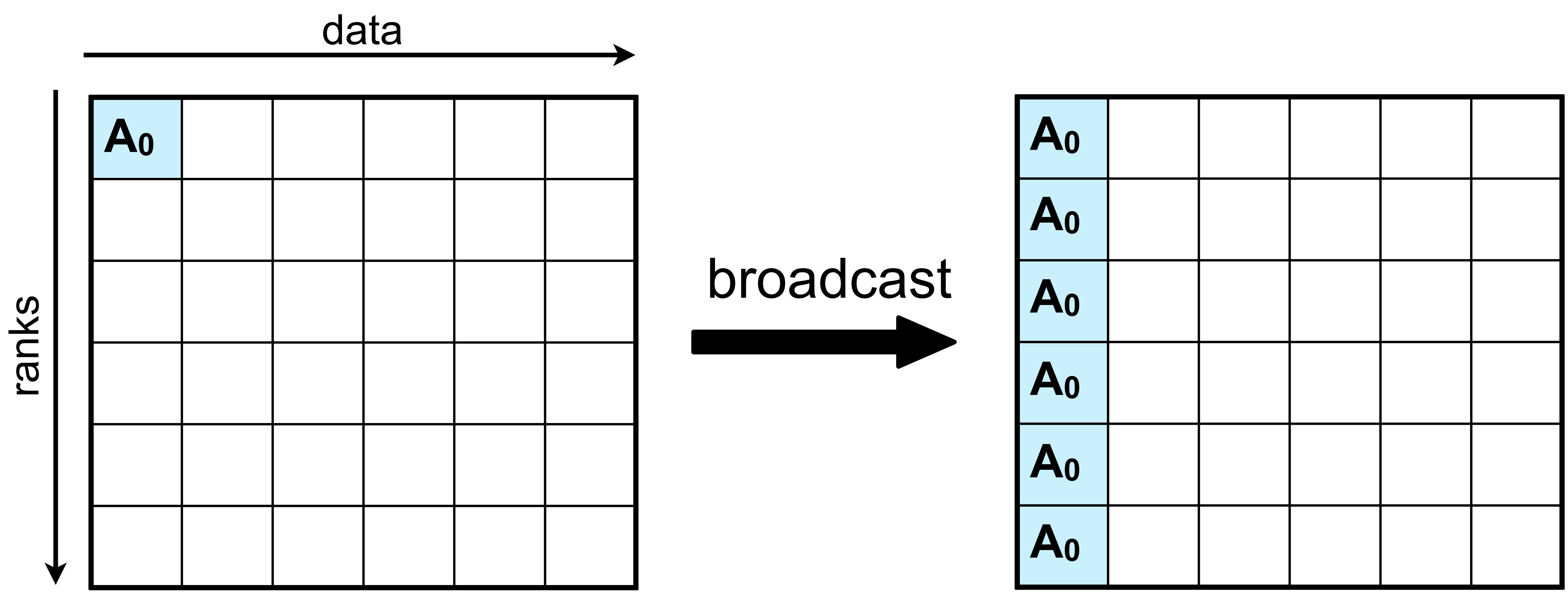
- MPI provides an extensive set of collective operations
- Operations defined over a communicator's processes
- All processes in a communicator must call the same collective operation
  - —e.g. all participants in a one-to-all broadcast call the broadcast primitive, even though all but the root are conceptually just "receivers"
- Simplest collective: barrier synchronization

```
int MPI_Barrier(MPI_Comm comm)
```

  – wait until all processes arrive

One-to-all Broadcast

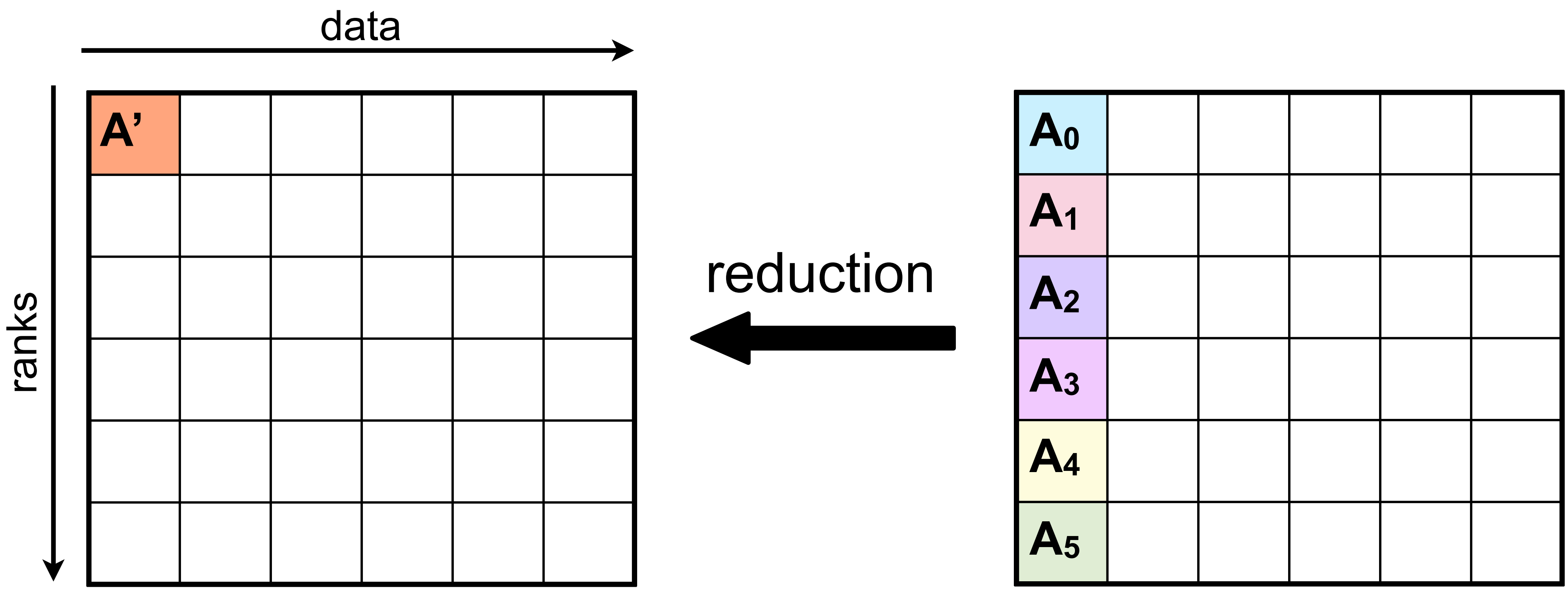
```
int MPI_Bcast(void *buf, int count,
              MPI_Datatype datatype, int source,
              MPI_Comm comm)
```



All-to-one Reduction

```
int MPI_Reduce(void *sendbuf, void *recvbuf,
               int count, MPI_Datatype datatype,
               MPI_Op op, int target, MPI_Comm comm)
```

MPI_Op examples: sum, product, min, max, ... (see next page)



$A' = op(A_0, A_1, \ldots, A_{p-1})$

MPI_Op Predefined Reduction Operations

Operation	Meaning	Datatypes
`MPI_MAX`	Maximum	integers and floating point
`MPI_MIN`	Minimum	integers and floating point
`MPI_SUM`	Sum	integers and floating point
`MPI_PROD`	Product	integers and floating point
`MPI_LAND`	Logical AND	integers
`MPI_BAND`	Bit-wise AND	integers and byte
`MPI_LOR`	Logical OR	integers
`MPI_BOR`	Bit-wise OR	integers and byte
`MPI_LXOR`	Logical XOR	integers
`MPI_BXOR`	Bit-wise XOR	integers and byte
`MPI_MAXLOC`	Max value-location	Data-pairs
`MPI_MINLOC`	Min value-location	Data-pairs

MPI_MAXLOC and MPI_MINLOC

- **Combine pairs of (value, index)**

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$

- **`MPI_MAXLOC`**

$$w = \mathtt{max}(u, v) \qquad k = \begin{cases} i & \mathtt{if}\ u > v \\ \mathtt{min}(i, j) & \mathtt{if}\ u = v \\ j & \mathtt{if}\ u < v \end{cases}$$

- **`MPI_MINLOC`**

$$w = \mathtt{min}(u, v) \qquad k = \begin{cases} i & \mathtt{if}\ u < v \\ \mathtt{min}(i, j) & \mathtt{if}\ u = v \\ j & \mathtt{if}\ u > v \end{cases}$$

value	9	12	14	27	9	27
index	0	1	2	3	4	5

MAXLOC(value,index) = (27, 3)   MINLOC(value,index) = (9, 0)

Data Types for MINLOC and MAXLOC Reductions

`MPI_MAXLOC` and `MPI_MINLOC` reductions operate on data pairs

MPI Datatype	C Datatype
`MPI_2INT`	**pair of** ints
`MPI_SHORT_INT`	short **and** int
`MPI_LONG_INT`	long **and** int
`MPI_LONG_DOUBLE_INT`	long double **and** int
`MPI_FLOAT_INT`	float **and** int
`MPI_DOUBLE_INT`	double **and** int

All-to-All Reduction and Prefix Scan

- **All-to-all reduction - every process gets a copy of the result**

```
int MPI_Allreduce(void *sendbuf, void *recvbuf,
            int count, MPI_Datatype datatype,
            MPI_Op op, MPI_Comm comm)
```

  —semantically equivalent to MPI_Reduce + MPI_Bcast

- **Parallel prefix operations**

  —inclusive scan: processor i result = op(v_0, ... v_i)

```
int MPI_Scan(void *sendbuf, void *recvbuf, int count,
            MPI_Datatype datatype, MPI_Op op,
            MPI_Comm comm)
```

  —exclusive scan: processor i result = op(v_0, ... v_{i-1})

```
int MPI_Exscan(void *sendbuf, void *recvbuf, int count,
            MPI_Datatype datatype, MPI_Op op,
            MPI_Comm comm)
```

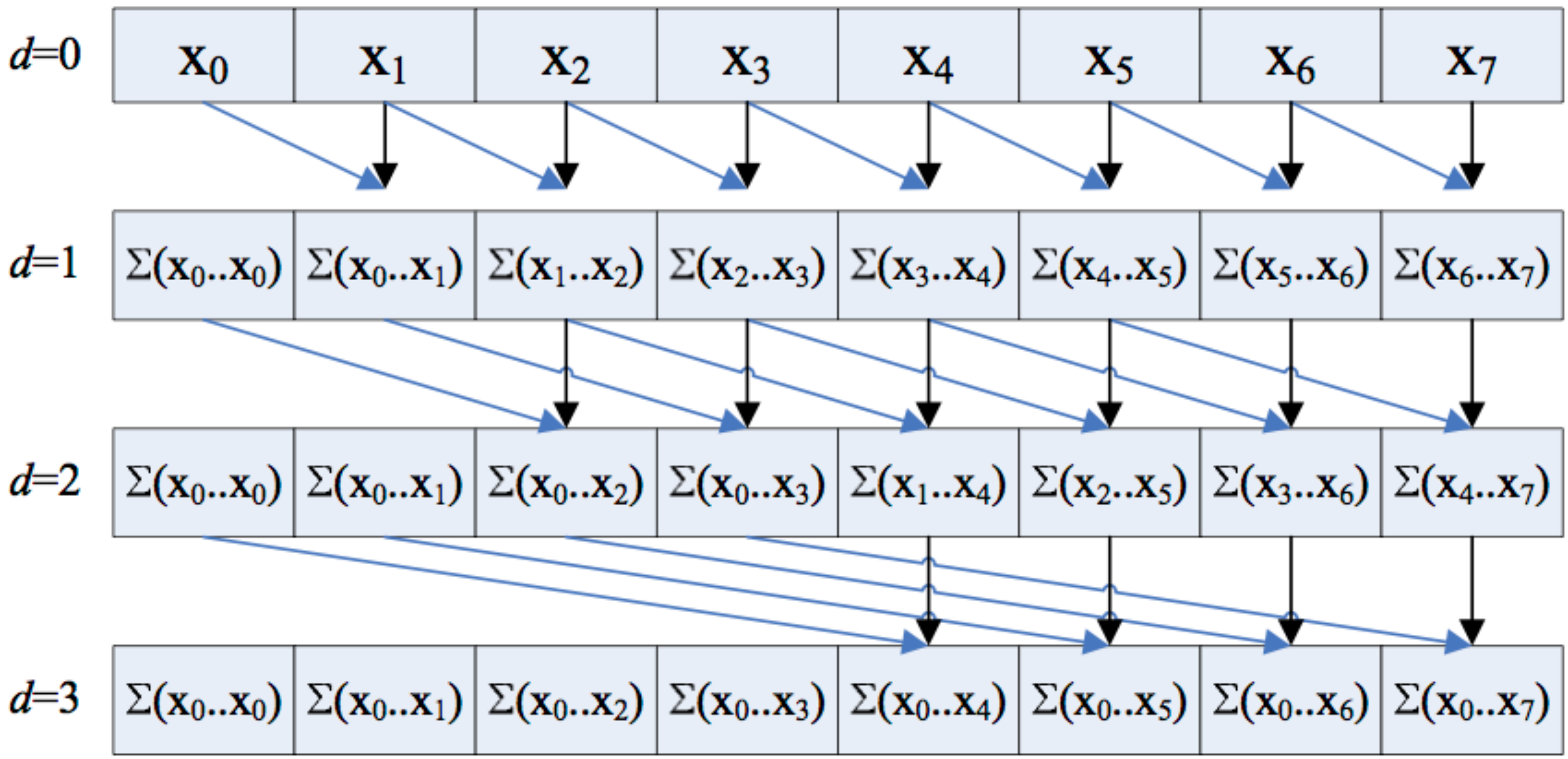
Parallel Prefix Scan

- Inclusive scan: processor i result = op(v_0, ... v_i)
- Exclusive scan: processor i result = op(v_0, ... v_{i-1})
- Examples for scans using sum for op

```
input    [2  4  1  1  0  1 -3  2  0  6  1  5]
scan     [2  6  7  8  8  9  6  8  8 14 15 20]
exscan   [0  2  6  7  8  8  9  6  8  8 14 15]
```

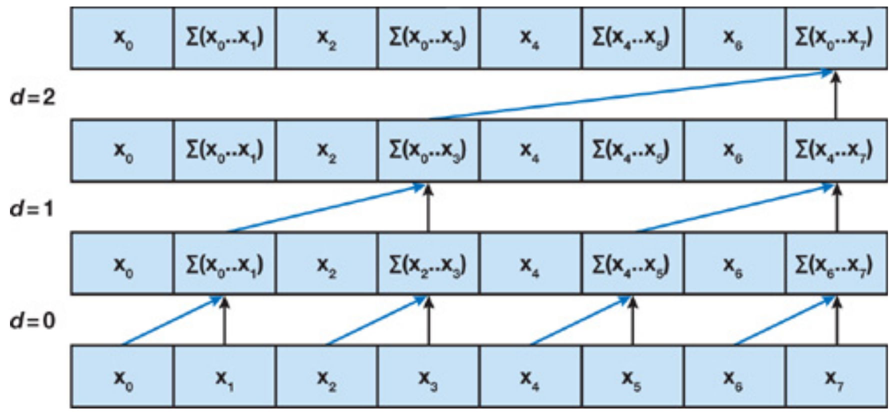
Inclusive Sum Scan with MPI_Scan

```
input    [2   4   1   1   0   1  -3   2   0   6   1   5]
scan     [2   6   7   8   8   9   6   8   8  14  15  20]
```

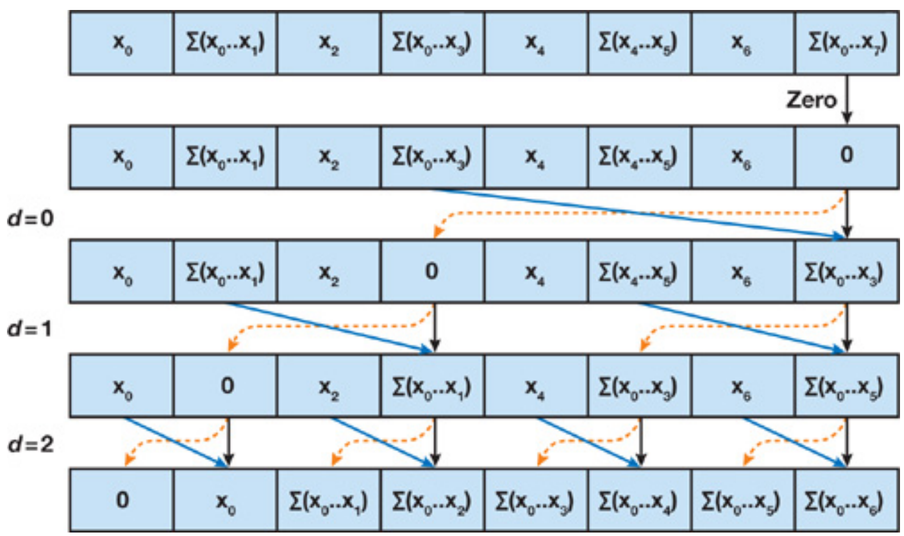


Exclusive Sum Scan with MPI_Exscan

```
input    [2  4  1  1  0  1 -3  2  0  6  1  5]
exscan   [0  2  6  7  8  8  9  6  8  8 14 15]
```



up sweep



down sweep

Scatter/Gather

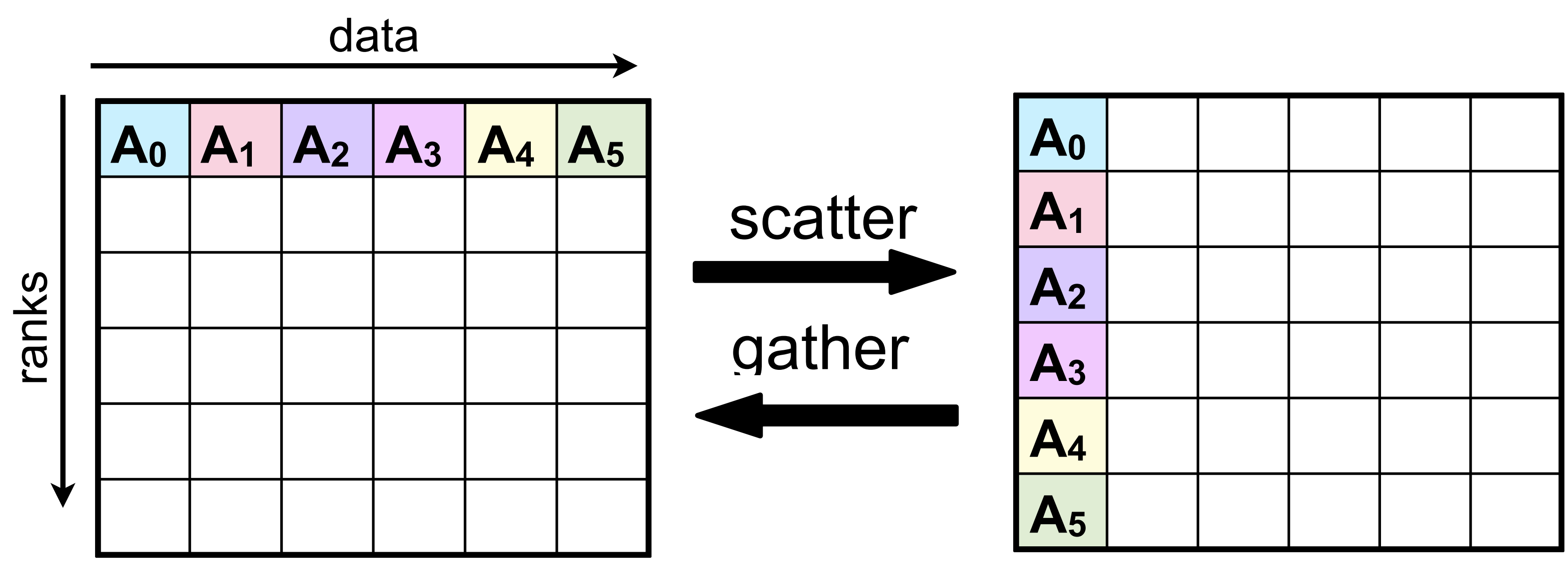
- **Scatter data p-1 blocks from root process delivering one to each other**

```
int MPI_Scatter(void *sendbuf, int sendcount,
          MPI_Datatype senddatatype, void *recvbuf,
          int recvcount, MPI_Datatype recvdatatype,
          int source, MPI_Comm comm)
```

sendcount = number sent to each

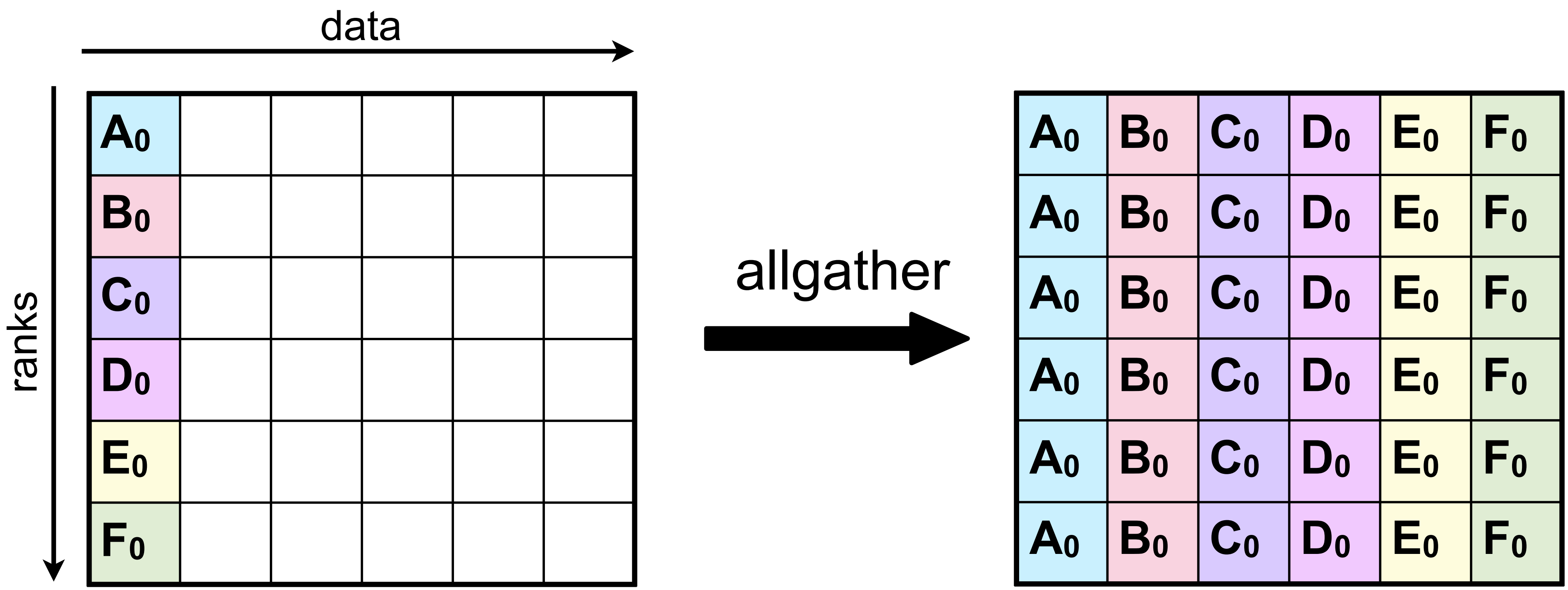
- **Gather data at one process**

```
int MPI_Gather(void *sendbuf, int sendcount,
          MPI_Datatype senddatatype, void *recvbuf,
          int recvcount, MPI_Datatype recvdatatype,
          int target, MPI_Comm comm)
```



Allgather

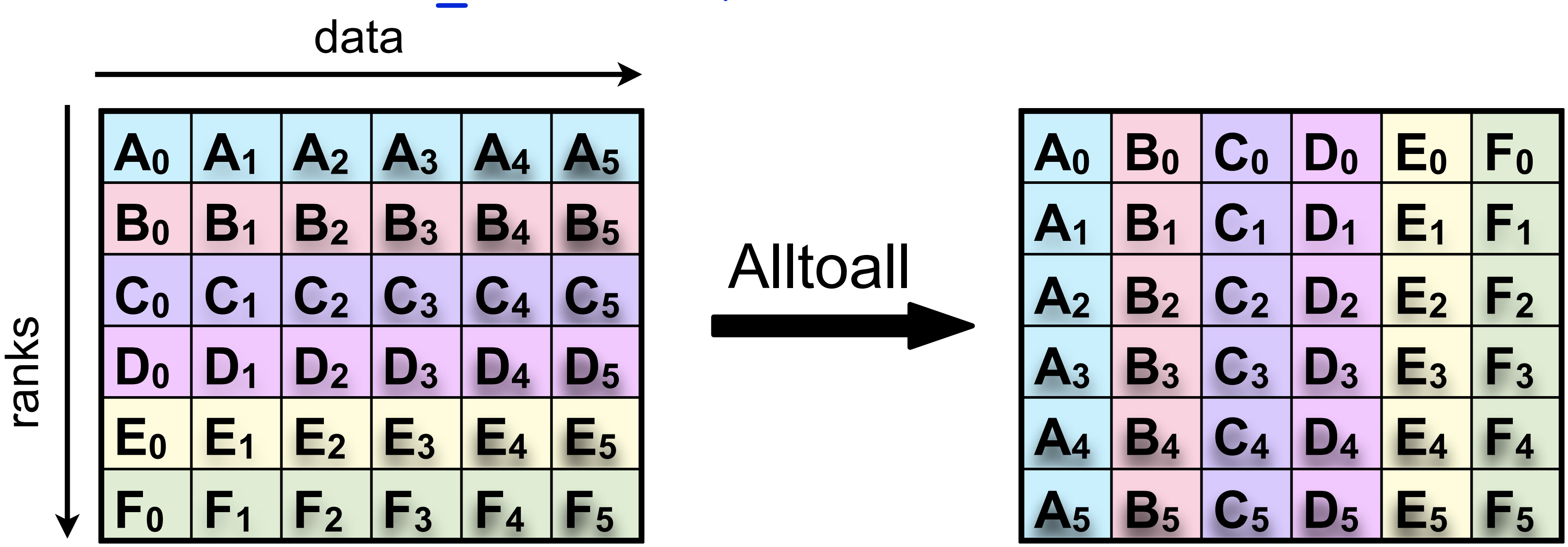
```
int MPI_AllGather(void *sendbuf, int sendcount,
                  MPI_Datatype senddatatype, void *recvbuf,
                  int recvcount, MPI_Datatype recvdatatype,
                  MPI_Comm comm)
```



All-to-All Personalized Communication

- Each process starts with its own set of blocks, one destined for each process
- Each process finishes with all blocks destined for itself
- Analogous to a matrix transpose

```
int MPI_Alltoall(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf,
        int recvcount, MPI_Datatype recvdatatype,
        MPI_Comm comm)
```



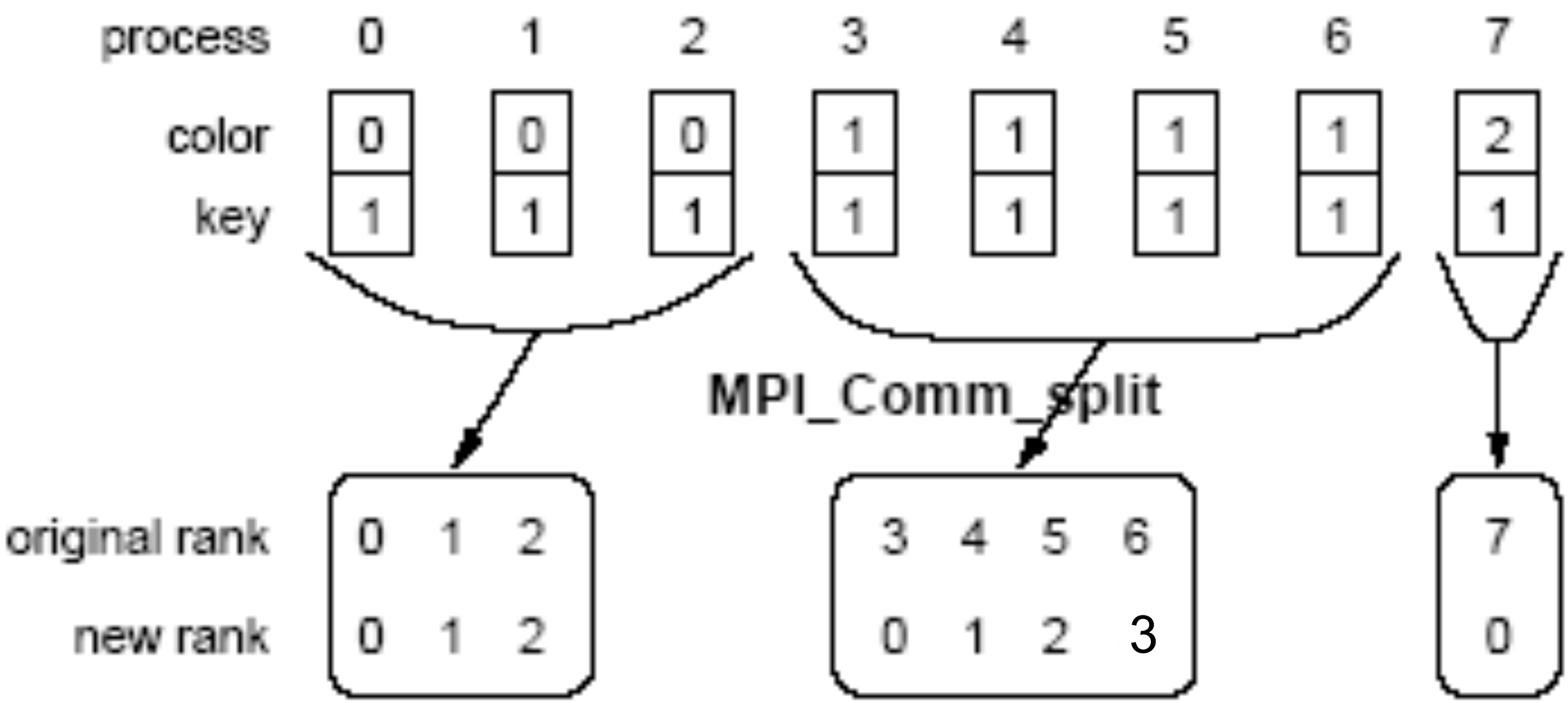
Splitting Communicators

- Useful to partition communication among process subsets
- MPI provides mechanism for partitioning a process group
  - —splitting communicators
- Simplest such mechanism

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
                   MPI_Comm *newcomm)
```

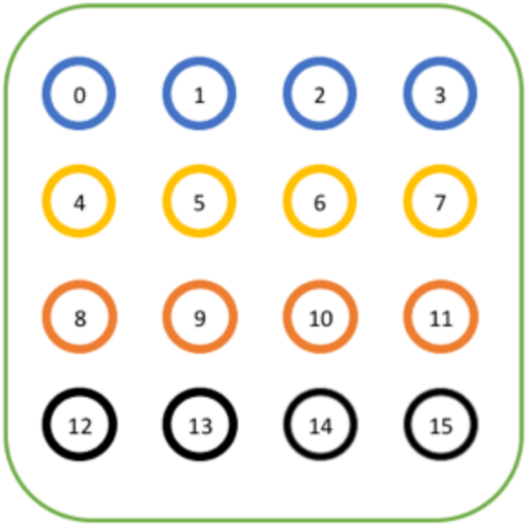
  - —effect
    - group processes by color
    - sort resulting groups by key

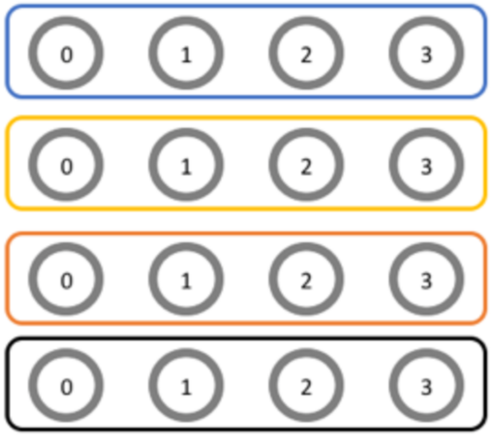
Splitting Communicators



Using `MPI_Comm_split` to split a group of processes in a communicator into subgroups

Splitting a Communicator






```c
// Get the rank and size in the original communicator
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int color = world_rank / 4; // Determine color based on row

// Split the communicator based on the color and use the
// original rank for ordering
MPI_Comm row_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &row_comm);
```

Figure Credit: http://mpitutorial.com/tutorials/introduction-to-groups-and-communicators

Cartesian Topologies

- For regular problems a multidimensional mesh organization of processes can be convenient
- Creating a new communicator augmented with a mesh view

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims,
                    int *dims, int *periods, int reorder,
                    MPI_Comm *comm_cart)
```

- Map processes into a mesh
  - `ndims    = number of dimensions`
  - `dims     = vector with length of each dimension`
  - `periods  = vector indicating which dims are periodic`
  - `reorder  = flag - ranking may be reordered`
- Processor coordinate in cartesian topology
  - a vector of length `ndims`

Using Cartesian Topologies

- **Sending and receiving still requires 1-D ranks**
- **Map Cartesian coordinates ⇔ rank**

```
int MPI_Cart_coord(MPI_Comm comm_cart, int rank, int maxdims,
                   int *coords)
```

```
int MPI_Cart_rank(MPI_Comm comm_cart, int *coords, int *rank)
```

- **Most common operation on cartesian topologies is a shift**
- **Determine the rank of source and destination of a shift**

```
int MPI_Cart_shift(MPI_Comm comm_cart, int dir, int s_step,
                   int *rank_source, int *rank_dest)
```

Splitting Cartesian Topologies

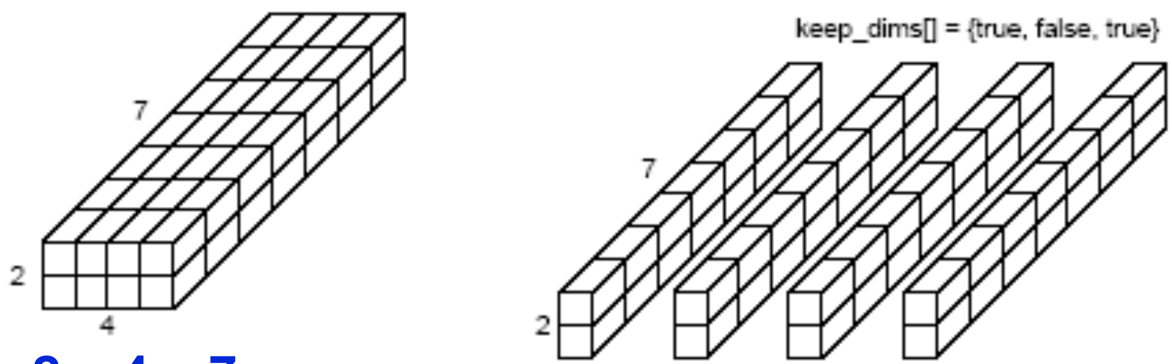
- **Processes arranged in a virtual grid using Cartesian topology**
- **May need to restrict communication to a subset of the grid**
- **Partition a Cartesian topology to form lower-dimensional grids**

```
int MPI_Cart_sub(MPI_Comm comm_cart, int *keep_dims,
                 MPI_Comm *comm_subcart)
```

- **If `keep_dims[i]` is true (i.e. non-zero in C)**
  - `i`**th dimension is retained in the new sub-topology**
- **Process coordinates in a sub-topology**
  - **derived from coordinate in the original topology**
  - **disregard coordinates for dimensions that were dropped**

Splitting Cartesian Topologies

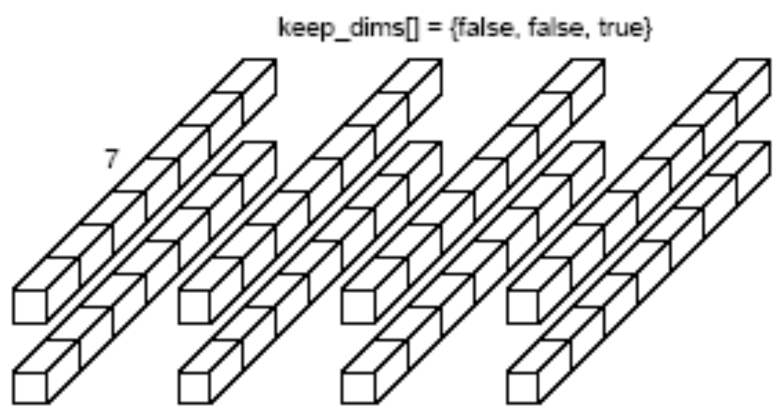
```
int MPI_Cart_sub(MPI_Comm comm_cart, int *keep_dims,
                 MPI_Comm *comm_subcart)
```



2 x 4 x 7

(a)

4 @ 2 x 1 x 7



(b)

8 @ 1 x 1 x 7

Graph Toplogies

- For irregular problems a graph organization of processes can be convenient

```
int MPI_Graph_create(MPI_Comm comm_old, int nnodes,
                     int *index, int *edges,
                     int reorder, MPI_Comm *cgraph)
```

- Map processes into a graph
  - nnodes = number of nodes
  - index = vector of integers describing node degrees
  - edges = vector of integers describing edges
  - reorder = flag indicating ranking may be reordered

process	neighbors
0	1, 3
1	0
2	3
3	0, 2

nnodes =	4
index =	2, 3, 4, 6
edges =	1, 3, 0, 3, 0, 2

difference between current and previous index indicates node degree

Operations on Graph Topologies

- **Interrogating a graph topology with MPI_Graphdims_get**

```
int MPI_Graphdims_get(MPI_Comm comm, int *nnodes,
                      int *nedges)
```

  - inquire about length of node and edge vectors

- **Extracting a graph topology with MPI_Graph_get**

```
int MPI_Graph_get(MPI_Comm comm, int maxindex,
                  int maxedges, int *index,
                  int *edges)
```

  - read out the adjacency list structure in index and edges

MPI Derived Data Types

- A **general datatype** is an opaque object that specifies 2 things
  - —a sequence of basic data types
  - —a sequence of integer (byte) displacements
    - – not required to be positive, distinct, or in increasing order
- Some properties of general data types
  - —order of items need not coincide with their order in memory
  - —an item may appear more than once
- **Type map** = pair of type & displacement sequences (equivalently, a sequence of pairs)
- **Type signature** = sequence of basic data types

Building an MPI Data Type

```
int MPI_Type_struct(int count, int blocklens[],
   MPI_Aint indices[], MPI_Datatype old_types[],
   MPI_Datatype *newtype )
```

if you define a structure datatype and wish to send or receive multiple items, you should explicitly include an MPI_UB entry as the last member of the structure.

Example

struct { int a; char b; } foo;

```
blen[0]=1; indices[0] = 0;                  // offset of a
oldtypes[0]=MPI_INT;
blen[1]=1; indices[1] = &foo.b - &foo.a; // offset of b
oldtypes[1]=MPI_CHAR;
blen[2]=1; indices[2] = sizeof(foo);     // offset of UB
oldtypes[2]= MPI_UB;
MPI_Type_struct( 3, blen, indices, oldtypes, &newtype );
```

MPI Data Type Constructor Example 1

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
                        MPI_Datatype *newtype)
```

- newtype is the datatype obtained by concatenating count copies of oldtype

- Example
  - consider constructing newtype from the following
    - oldtype with type map *{ (double, 0), (char, 8) }* , with extent 16
    - let count = 3
  - type map of newtype is
    - *{ (double, 0), (char, 8),*
      (double, 16), (char, 24),
      (double, 32), (char, 40) }
    - namely, alternating double and char elements, with displacements *0*, *8*, *16*, *24*, *32*, *40*

MPI Data Type Constructor Example 2

```
int MPI_Type_vector(int count, int blocklength, int stride,
                    MPI_Datatype oldtype,
                    MPI_Datatype *newtype)
```

- **Let oldtype have type map**

  { (double, 0), (char, 8) } with extent 16

- **A call to `MPI_Type_vector`(2, 3, 4, oldtype, newtype) will create the datatype with type map**
  - two blocks with three copies each of the old type, with a stride of 4 elements (*4* x *16* bytes) between the blocks

{ (double, 0), (char, 8), (double, 16), (char, 24), (double, 32), (char, 40),
(double, 64), (char, 72), (double, 80), (char, 88), (double, 96), (char, 104) }

Threads and MPI

- **MPI-2 Specification**
  - —does not mandate thread support
  - —specifies what a thread-compliant MPI should do
  - —specifies four levels of thread support
- **Threads are not addressable**
  - —MPI_Send(... thread_id ...) is not possible

Initializing MPI for Threading

```
int MPI_Init_thread(int *argc, char ***argv,
                    int required, int *provided)
```

Used instead of MPI_Init; MPI_Init_thread has a provision to request a certain level of thread support in required

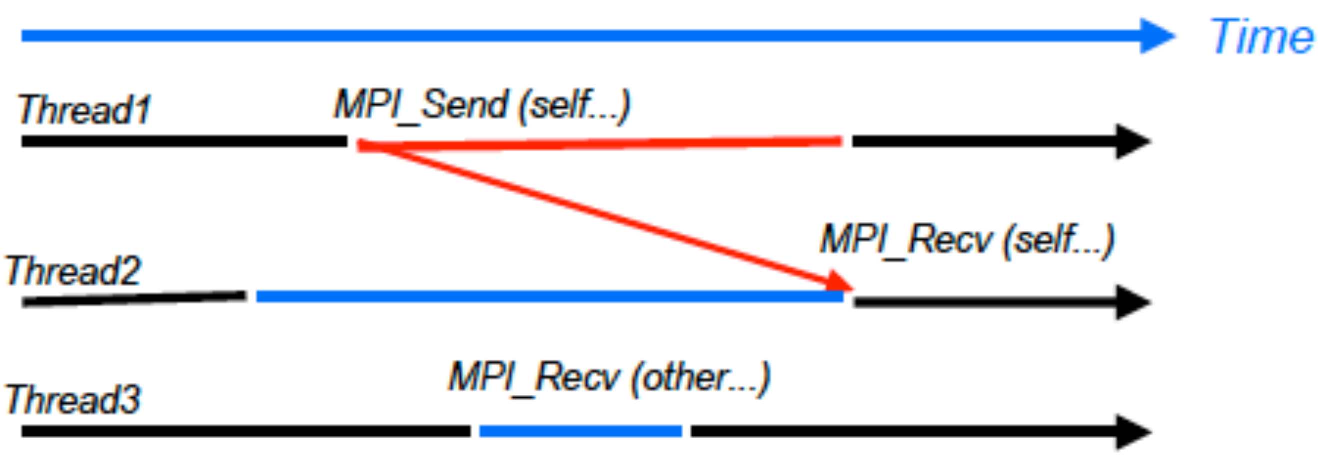
- **MPI_THREAD_SINGLE: only one thread will execute**
- **MPI_THREAD_FUNNELED: if the process is multithreaded, only the thread that called MPI_Init_thread will make MPI calls**
- **MPI_THREAD_SERIALIZED: if the process is multithreaded, only one thread will make MPI library calls at one time**
- **MPI_THREAD_MULTIPLE: if the process is multithreaded, multiple threads may call MPI at once with no restrictions**

Require the lowest level that you need

MPI_Init is equivalent to supplying MPI_THREAD_SINGLE to MPI_Init_thread

Thread-compliant MPI

- **All MPI library calls are thread safe**
- **Blocking calls block the calling thread only**
  - **—other threads can continue executing**



MPI Threading Inquiry Primitives

- Inquire about what kind of thread support MPI has provided to your application

```
int MPI_Query_thread(int *provided)
```

- Inquire whether this thread called MPI_Init or MPI_Init_thread

```
int MPI_Is_thread_main(int *flag)
```

MPI + Threading Example

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int errs = 0;
    int provided, flag, claimed;
    pthread_t thread;

    MPI_Init_thread( 0, 0, MPI_THREAD_MULTIPLE, &provided );

    MPI_Is_thread_main( &flag );
    if (!flag) {
        errs++;
        printf( "This thread called init_thread but Is_thread_main gave false\n" );
        fflush(stdout);
    }
    MPI_Query_thread( &claimed );
    if (claimed != provided) {
        errs++;
        printf( "Query thread gave thread level %d but Init_thread gave %d\n", claimed, provided );
        fflush(stdout);
    }
    pthread_create(&thread, NULL, mythread_function, NULL);
    ...

    MPI_Finalize();
    return errs;
}
```

One-Sided vs. Two-Sided Communication

- **Two-sided: data transfer and synchronization are conjoined**
  - —message passing communication is two-sided
    - – sender and receiver issue explicit send or receive operations to engage in a communication
- **One-sided: data transfer and synchronization are separate**
  - —a process or thread of control can read or modify remote data without explicit pairing with another process
  - —terms
    - – origin process: process performing remote memory access
    - – target process: process whose data is being accessed

Why One-Sided Communication?

- If communication pattern is not known a priori, using a two-sided (send/recv) model requires an extra step to determine how many sends-recvs to issue on each processor

Consider the communication associated with acquiring information about neighboring vertices in a partitioned graph

- Easier to code using one-sided communication because only the origin or target process needs to issue the put or get call
- Expose hardware shared memory
  —more direct mapping of communication onto HW using load/store
    – avoid SW overhead of message passing; let the HW do its thing!

Figure credit: "Introduction to Parallel Computing" A. Grama, A. Gupta, G. Karypis, and V. Kumar. Addison Wesley, 2003

One-Sided Communication in MPI-2

- MPI-2 Remote Memory Access (RMA)
  - —processes in a communicator can read, write, and accumulate values in a region of “shared” memory
- Two aspects of RMA-based communication
  - —data transfer, synchronization
- RMA advantages
  - —multiple data transfers with a single synchronization operation
  - —can be significantly faster than send/recv on some platforms
    - – e.g. systems with hardware support for shared memory

MPI-2 RMA Operation Overview

- **MPI_Win_create**
  - collective operation to create new window object
  - exposes memory to RMA by other processes in a communicator
- **MPI_Win_free**
  - deallocates window object
- **Non-blocking data movement operations**
  - 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
- **Synchronization operations**

Active Target vs. Passive Target RMA

- Passive target RMA
  - target process makes no synchronization call
- Active target RMA
  - requires participation from the target process in the form of synchronization calls (fence or post/wait, start/complete)
- Illegal to have overlapping active and passive RMA epochs

Synchronization for Passive Target RMA

- **MPI_Win_lock(locktype, target_rank, assert, win) “beginning RMA”**
  - —locktype values
    - – 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)
      - useful when accessing window only with MPI_Get
  - —assert values
    - – 0
    - – MPI_MODE_NOCHECK
- **MPI_Win_unlock(target_rank, win) “ending RMA”**

Active Target Synchronization

- **MPI_Win_start**
  —begins an RMA epoch on origin process
- **MPI_Win_post**
  —starts RMA exposure for a local window on a target process
- **MPI_Win_wait/test**
  —end RMA exposure on local window on a target process
- **MPI_Win_complete**
  —forces local completion an RMA epoch on origin
- **MPI_Win_fence**
  —collective forces remote completion of put/get/acc before fence

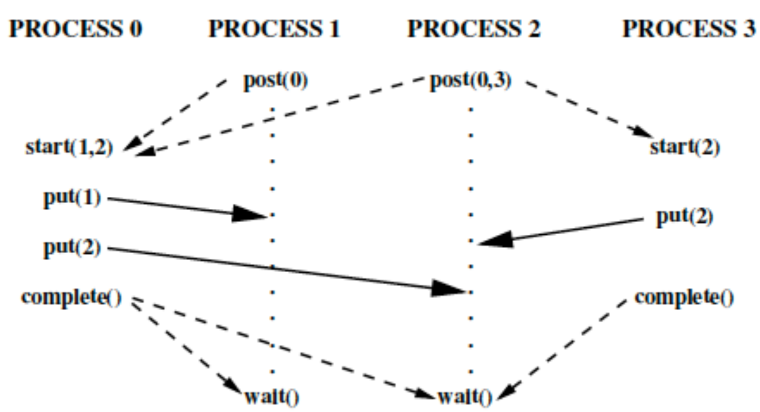


Figure credit: MPI-3 draft specification, Nov. 2010.

MPI RMA Active Target Example 1

Generic loosely synchronous, iterative code, using fence synchronization

The window at each process consists of array A, which contains the origin and target buffers of the Get calls

```
...
while(!converged(A)){
   update_boundary(A);
   MPI_Win_fence(MPI_MODE_NOPRECEDE, win);
   for(i=0; i < toneighbors; i++)
      MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
                 fromdisp[i], 1, fromtype[i], win);
   update_core(A);
   MPI_Win_fence(MPI_MODE_NOSUCCEED,win);
 }
```

Similar code could be written with Put rather than Get

MPI RMA Active Target Example 2

Generic loosely synchronous, iterative code, using fence synchronization

The window at each process consists of array A, which contains the origin and target buffers of the Get calls

```
...
while(!converged(A)){
   update_boundary(A);
   MPI_Win_post(togroup, win);
   MPI_Win_start(fromgroup, win);
   for(i=0; i < toneighbors; i++)
      MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
                  fromdisp[i], 1, fromtype[i], win);
   update_core(A);
   MPI_Win_complete(win);
 }
```

Similar code could be written with Put rather than Get

MPI-1 Profiling Interface - PMPI

- To support tools, MPI implementations define two interfaces to every MPI function
  - —MPI_xxx
  - —PMPI_xxx
- One can "wrap" MPI functions with a tool library to observe execution of an MPI program

```
int MPI_Send(void* buffer, int count, MPI_Datatype dtype,
                int dest, int tag, MPI_Comm comm)
   {
     double tstart = MPI_Wtime(); /* Pass on all arguments */
     int extent;
     int result = PMPI_Send(buffer,count,dtype,dest,tag,comm);
     MPI_Type_size(datatype, &extent); /* Compute size */
     totalBytes += count*extent;
     totalTime += MPI_Wtime() - tstart; /* and time */
     return result;
   }
```

Some MPI Tools

- MPICH MPI implementation
  - —MPI tracing library
  - —Jumpshot trace visualization tool
- Vampir: MPI trace analysis tools
  - —http://www.vampir.eu/
- MPIp library for profiling MPI operations
  - —http://mpip.sourceforge.net
- memcheck
  - —OpenMPI + valgrind checks correct use of comm buffers
  - —http://www.open-mpi.org
- marmot
  - —checks usage of MPI routines
  - —http://www.hlrs.de/organization/av/amt/projects/marmot

Vampir displays

MPI Libraries

- **SCALAPACK - dense linear algebra using block-cyclic tilings**
  - http://www.netlib.org/scalapack/scalapack_home.html
- **PetSC - Portable Extensible, Toolkit for Scientific Computation**
  - data structures and routines for solution of scientific applications modeled by partial differential equations
  - http://www.mcs.anl.gov/petsc/petsc-as
- **Trilinos - software framework for solving large-scale, complex multi-physics engineering and scientific problems**
  - http://trilinos.sandia.gov

MPI-3 Additions

Nonblocking collective operations

- —barrier synchronization
- —broadcast
- —gather
- —scatter
- —gather-to-all
- —all-to-all scatter/gather
- —reduce
- —reduce-scatter
- —inclusive scan
- —exclusive scan

Building MPI Programs

- **Each MPI installation defines compilation scripts**
  - **—mpicc: C**
  - **—mpif90: Fortran 90**
  - **—mpif77: Fortran 77**
  - **—mpicxx, mpiCC: C++**
- **Benefits of using these scripts**
  - **—they supply the appropriate paths**
    - **for MPI include files**
    - **for MPI library files**
  - **—they link appropriate libraries into your executable**

Common Errors and Misunderstandings

- Expecting argc and argv to be passed to all processes
  - —some MPI implementations pass them to all processes, but the MPI standard does not require it
- Doing things before MPI_Init or after MPI_Finalize
  - —the MPI standard says nothing about the state of an execution outside this interval
- Matching MPI_Bcast with MPI_Recv; all should use MPI_Bcast
- Assuming your MPI implementation is thread safe

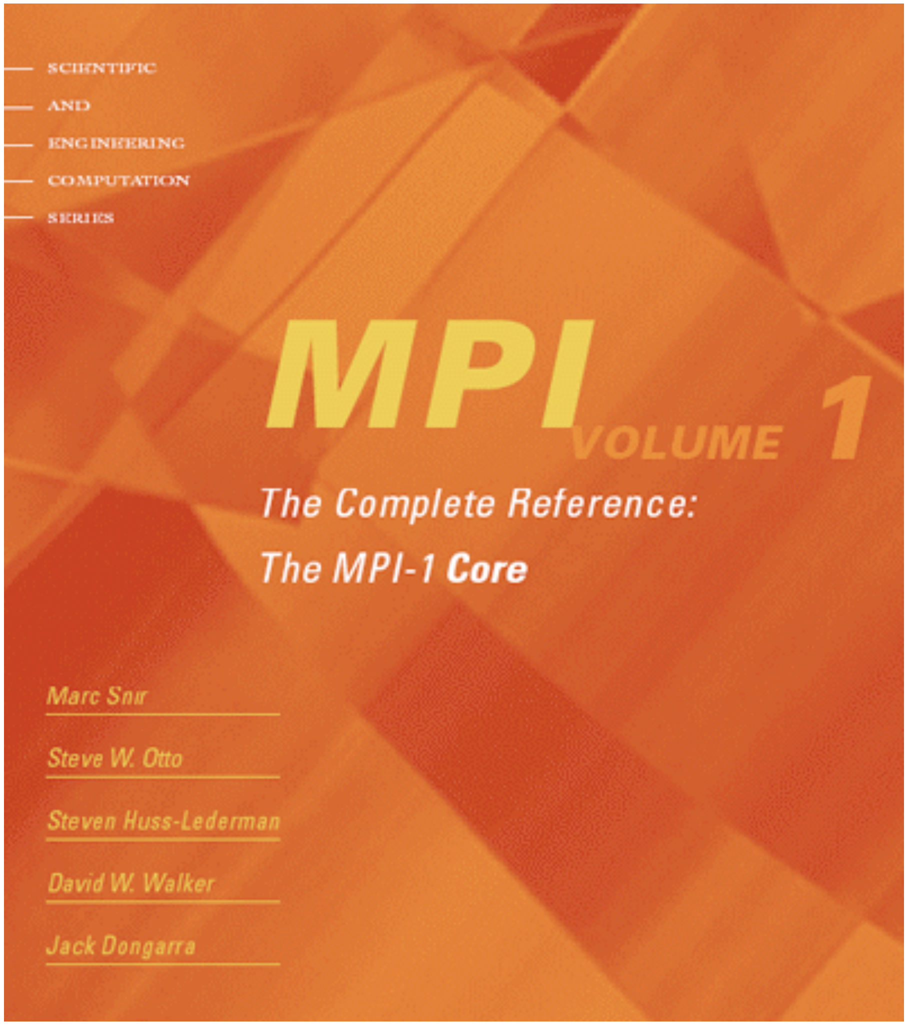
Running MPI Programs

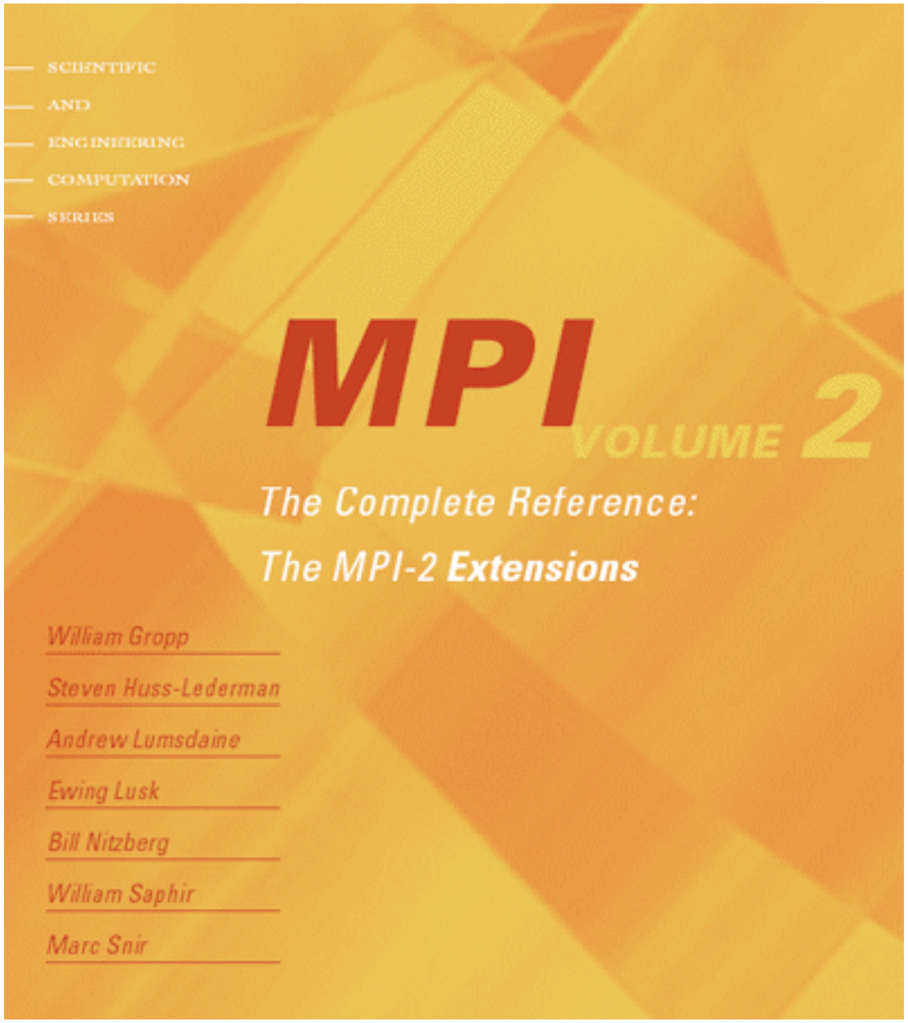
- Each MPI installation provides one or more launch scripts
  - mpirun
  - mpiexec
- On networks of workstations, launch MPI as follows
  - mpirun [-np PE] [--hostfile <filename>] <pgm>
    - mpirun will use rsh or ssh to launch jobs on machines in hostfile
    - without a hostfile, it will run all jobs on the local node
- If running under a resource manager (e.g. PBS)
  - `mpirun [-np ncores] yourprogram`
  - `mpiexec [-np ncores] yourprogram`

MPI Online Resources

- http://www.mpi-forum.org
  - http://www.mpi-forum.org/docs/docs.html
    - MPI standards documents (all official releases)
- http://www.mcs.anl.gov/research/projects/mpi/
  - tutorials http://www.mcs.anl.gov/research/projects/mpi/learning.html
  - MPICH and MPICH2 implementations by ANL

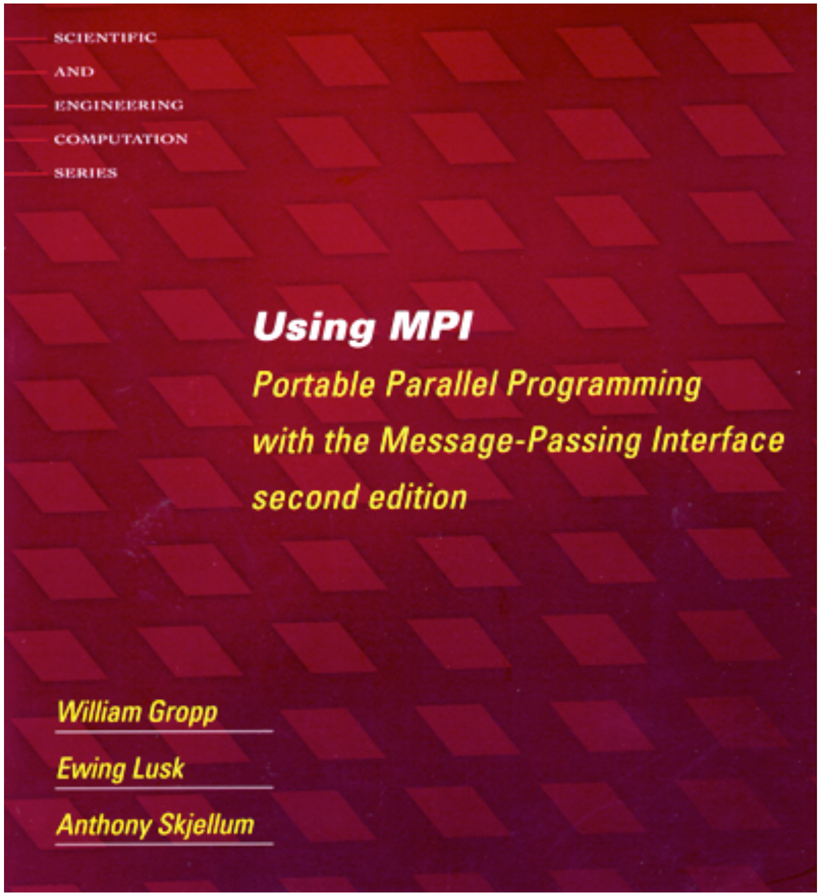
The MPI and MPI-2 Standards

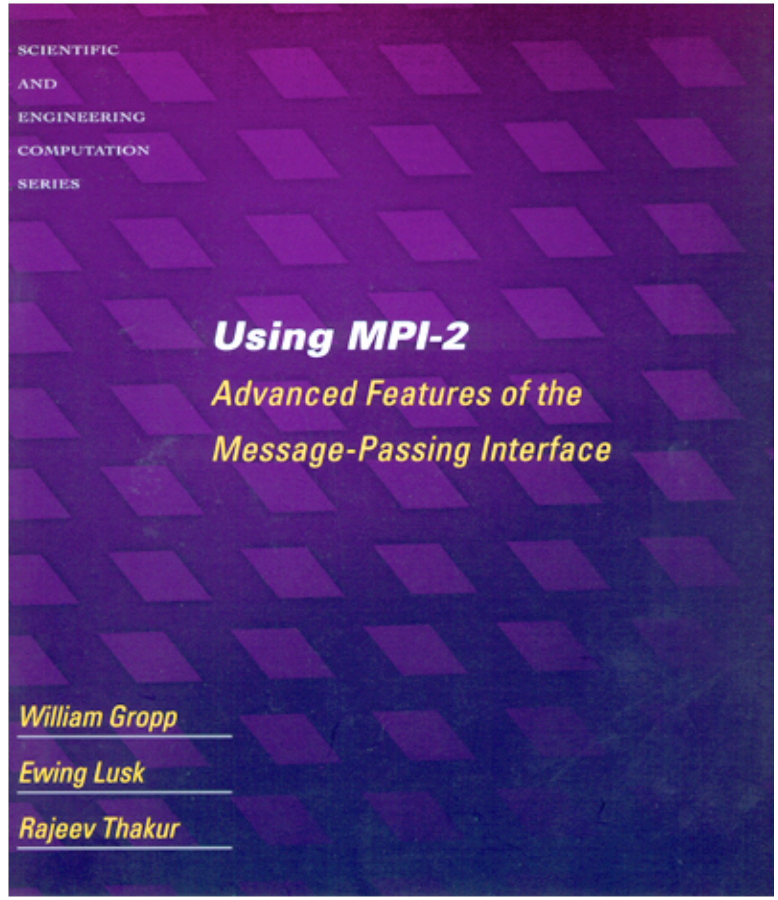




- **MPI: The Complete Reference, Volume 1 (2nd Edition) - The MPI Core, MIT Press, 1998.**
- **MPI: The Complete Reference, Volume 2 - The MPI-2 Extensions, MIT Press, 1998.**

Guides to MPI and MPI-2 Programming





- **Using MPI: http://www.mcs.anl.gov/mpi/usingmpi**
- **Using MPI-2: http://www.mcs.anl.gov/mpi/usingmpi2**

References

- William Gropp, Ewing Lusk and Anthony Skjellum. Using MPI, 2nd Edition Portable Parallel Programming with the Message Passing Interface. MIT Press, 1999; ISBN 0-262-57132-3
- Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. “Introduction to Parallel Computing,” Chapter 6. Addison Wesley, 2003.
- Athas, W. C. and Seitz, C. L. 1988. Multicomputers: Message-Passing Concurrent Computers. Computer 21, 8 (Aug. 1988), 9-24. DOI= http://dx.doi.org/10.1109/2.73