MPI

Jakub Yaghob

Literature and references

- Books
  - Gropp W., Lusk E., Skjellum A.: Using MPI: Portable Parallel Programming with the Message-Passing Interface, ISBN 978-0262527392, MIT Press, 2014
  - Gropp W., Hoefler T., Thakur R., Lusk E.: Using Advanced MPI: Modern Features of the Message-Passing Interface, ISBN 978-0262527637, MIT Press, 2014
- References
  - MPI forum (standard)
    - http://www.mpi-forum.org/docs/docs.html
  - Cornell Virtual Workshop
    - https://www.cac.cornell.edu/VW/topics.aspx

What is MPI?

- Message Passing Interface
- A library of functions
- MPI-1 standard (1994)
    - MPI-1.1 (1995), MPI-1.2 (in MPI 2, 1997), MPI-1.3 (2008)
- MPI-2 standard (1997)
    - MPI-2.1 (2008), MPI-2.2 (2009)
- MPI-3 standard (2012)
    - MPI-3.1 (2015)

MPI-1 standard

- MPI-1 standard (1994)
  - Specifies the names, calling sequences, and results of subroutines and functions to be called from Fortran 77 and C, respectively. All implementations of MPI must conform to these rules, thus ensuring portability. MPI programs should compile and run on any platform that supports the MPI standard
  - The detailed implementation of the library is left to individual vendors, who are thus free to produce optimized versions for their machines
  - Implementations of the MPI-1 standard are available for a wide variety of platforms

MPI-2 standard

- MPI-2 standard
  - Additional features not presented in MPI-1
  - Tools for parallel I/O
  - C++ and Fortran 90 bindings
  - Dynamic process management

MPI-3 standard

- MPI-3 standard
  - Nonblocking collective communication
  - One side communication
  - Removed C++ binding
  - Added Fortran 2008 binding
- MPI-3.1 standard
  - Minor update
  - Portable manipulation with MPI_Aint
  - Nonblocking collective I/O routines

Goals

- The primary goals
  - Provide source code portability
    - MPI programs should compile and run as-is on any platform
  - Allow efficient implementations across a range of architectures
- MPI also offers
  - A great deal of functionality, including a number of different types of communication, special routines for common collective operations, and the ability to handle user-defined data types and topologies
  - Support for heterogeneous parallel architectures

Goals – cont.

- Some things explicitly outside of MPI-1
  - Explicit shared-memory operations
  - The precise mechanism for launching an MPI program
    - Platform dependent
  - Dynamic process management
    - Included in MPI-2
  - Debugging
  - Parallel I/O
    - Included in MPI-2
  - Operations that require more OS support
    - Interrupt-driven receives

Why (not) use MPI?

- You should use MPI when you need to
  - Write portable parallel code
  - Achieve high performance in parallel programming
  - Handle a problem that involves irregular or dynamic data relationship that do not fit well into the data-parallel environment (High-Performance Fortran)
- You should not use MPI when you
  - Can achieve sufficient performance and portability using a data-parallel or shared-memory approach
  - Can use a pre-existing library of parallel routines
  - Don’t need parallelism at all

Library calls

- Library calls classes
  - Initialize, manage, and terminate communications
  - Communication between pairs of processes
  - Communication operations among groups of processes
  - Arbitrary data types

Hello world!

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

void main(int argc, char **argv) {
  int err;
  err = MPI_Init(&argc, &argv);
  printf("Hello world!\n");
  err = MPI_Finalize();
}
```

include

returned
error

naming
convention

Initializing MPI

```
int MPI_Init(int *argc, char ***argv);
```

- Must be called as the first MPI routine
- Establishes the MPI environment

Terminating MPI

```
int MPI_Finalize(void);
```

- The last MPI routine
- Cleans up all MPI data structures, cancels incomplete operations
- Must be called by all processes
  - Otherwise the program will appear to hang

Datatypes

- Variables normally declared as C/Fortran types
- MPI type names used as arguments in MPI routines
- Hides the details of representation
- Automatic translation between representations in a heterogeneous environment
- Arbitrary data types built from the basic types

Basic datatypes

MPI datatype	C type
MPI_CHAR	char (printable)
MPI_SIGNED_CHAR	signed char (integer)
MPI_UNSIGNED_CHAR	unsigned char (integer)
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

Basic datatypes – cont.

MPI datatype	C type
MPI_WCHAR	wchar_t
MPI_INT8_T	int8_t
MPI_UINT8_T	uint8_t
MPI_INT16_T	int16_t
MPI_UINT16_T	uint16_t
MPI_INT32_T	int32_t
MPI_UINT32_T	uint32_t
MPI_INT64_T	int64_t
MPI_UINT64_T	uint64_t
MPI_BYTE	(none)
MPI_PACKED	(none)

Special datatypes

- MPI provides several special datatypes
  - MPI_Comm – a communicator
  - MPI_Status – a structure with several fields of status information for MPI calls
  - MPI_Datatype – a datatype
  - MPI_Request – a nonblocking operation
  - MPI_Aint – an address in a memory

Communicators

- A communicator – a handle representing a group of processes that can communicate with one another
  - Processes can communicate only if they share a communicator
  - There can be many communicators
  - A process can be a member of a number of different communicators
  - Processes numbered sequentially (0-based)
    - Rank of the process
      - Different ranks in different communicators
  - A basic communicator MPI_COMM_WORLD
    - All processes

Getting communicator information

- Rank

```
int MPI_Comm_rank(MPI_Comm comm, int
   *rank);
```

  - Determines a rank for a given communicator
- Size

```
int MPI_Comm_size(MPI_Comm comm, int
   *size);
```

  - A number of processes in a communicator

Point-to-point communication

- One process sends a message and another process receives it
- Active participation from the processes on both sides
- The source and destination processes operate asynchronously
    - The source process may complete sending a message long before the destination process receives the message
    - The destination process may initiate receiving a message that has not yet been sent

Message

- Two parts
  - Envelope
    - Source – the sending process
    - Destination – the receiving process
    - Communicator – a group of processes to which both processes belong
    - Tag – classify messages
  - Message body
    - Buffer – the message data
      - An array datatype[count]
    - Datatype – the type of the message data
    - Count – the number of items of type datatype in buffer

Sending a message

- Blocking send

```
int MPI_Send(void* buf, int count,
  MPI_Datatype datatype, int dest, int tag,
  MPI_Comm comm);
```

  - All arguments are input arguments
  - Returns an error code
  - Possible behaviors
    - The message may be copied into an MPI internal buffer and transferred to its destination later
    - The message may be left where it is, in the program’s variables, until the destination process is ready to receive it
      - Minimizes copying and memory use

Receiving a message

- Blocking receive

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

  - The message envelope arguments determine what messages can be received
    - The source wildcard MPI_ANY_SOURCE
    - The tag wildcard MPI_ANY_TAG
  - It is an error, when the message contains more data than the receiving process is prepared to accept
  - The sender and receiver must use the same message datatype
    - Not checked, undefined behavior
  - Status contains the source, the tag and the actual count

Status

- MPI_Status structure
  - Predefined members MPI_SOURCE, MPI_TAG, MPI_ERROR

```
int MPI_Get_count(const MPI_Status *status,
  MPI_Datatype datatype, int *count)
```

  - Getting the number of elements in the message
    - Datatype should be the same as in MPI_Recv, MPI_Probe, etc.

Derived types

- Constructors
  - MPI_Type_contiguous
    - A contiguous sequence of values in memory
  - MPI_Type_vector
    - Several sequences evenly spaced but not consecutive in memory
  - MPI_Type_hvector
    - Identical to VECTOR, except the distance between successive blocks is in bytes
    - Elements of some other type are interspersed in memory with the elements of interest
  - MPI_Type_indexed
    - Sequences that may vary both in length and in spacing
    - Arbitrary parts of a single array
  - MPI_Type_hindexed
    - Similar to INDEXED, except that the locations are specified in bytes
    - Arbitrary parts of arbitrary arrays, all have the same type

Derived types

- General constructor

```
int MPI_Type_struct(int count, int
  *array_of_blocklengths, MPI_Aint
  *array_of_displacements, MPI_Datatype
  *array_of_types, MPI_Datatype *newtype);
```

Derived types

- Address

```
int MPI_Get_address(const void* location,
  MPI_Aint *address);
```

  - The address of a location in a memory

```
MPI_Aint MPI_Aint_add(MPI_Aint base,
  MPI_Aint disp);
```

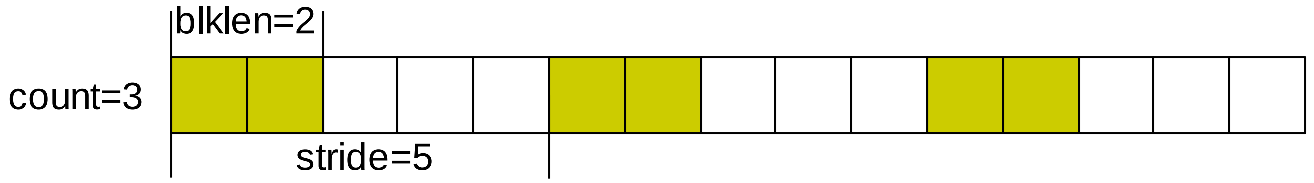
  - Sum of the **`base`** and **`disp`**

```
MPI_Aint MPI_Aint_diff(MPI_Aint addr1,
  MPI_Aint addr2);
```

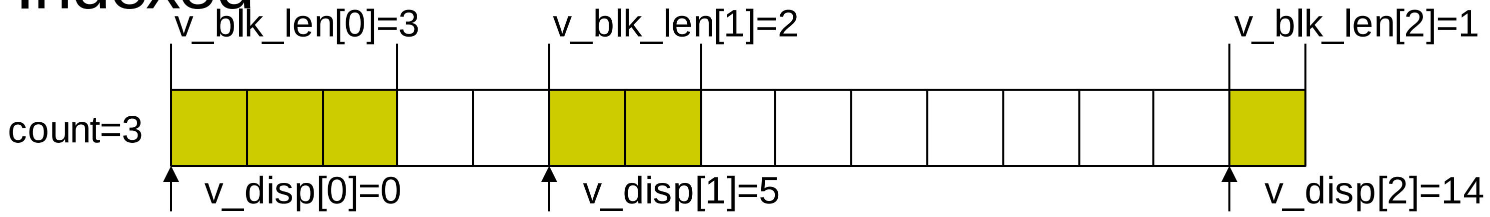
  - Displacement in the same object in the same process

Derived types

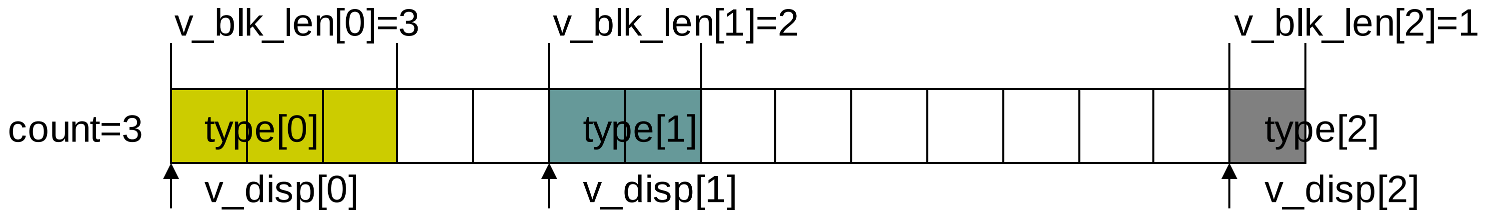
- Derived types in pictures
  - Vector



  - Indexed



  - Struct



Derived types

- Commit

```
int MPI_Type_commit(MPI_Datatype *datatype);
```

  - Must be called before using the datatype in a communication

- Free

```
int MPI_Type_free(MPI_Datatype *datatype);
```

  - Deallocates the datatype
  - Any communication using the datatype will complete normally
  - Derived datatypes are not affected

Collective communication

- Communication among all processes in a group
- Set of collective communication routines
  - Hide implementation details
  - The most efficient algorithm
- Collective communication calls do not use tags
  - Associated by order of program execution
  - The programmer must ensure that all processes execute the same collective communication calls and execute them in the same order

Barrier synchronization

- Barrier

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

  - Blocks the calling process until all processes in a group call this function
  - Use it, when some processes cannot proceed until other processes have completed their computation
    - Master process reads the data and transmit them to workers

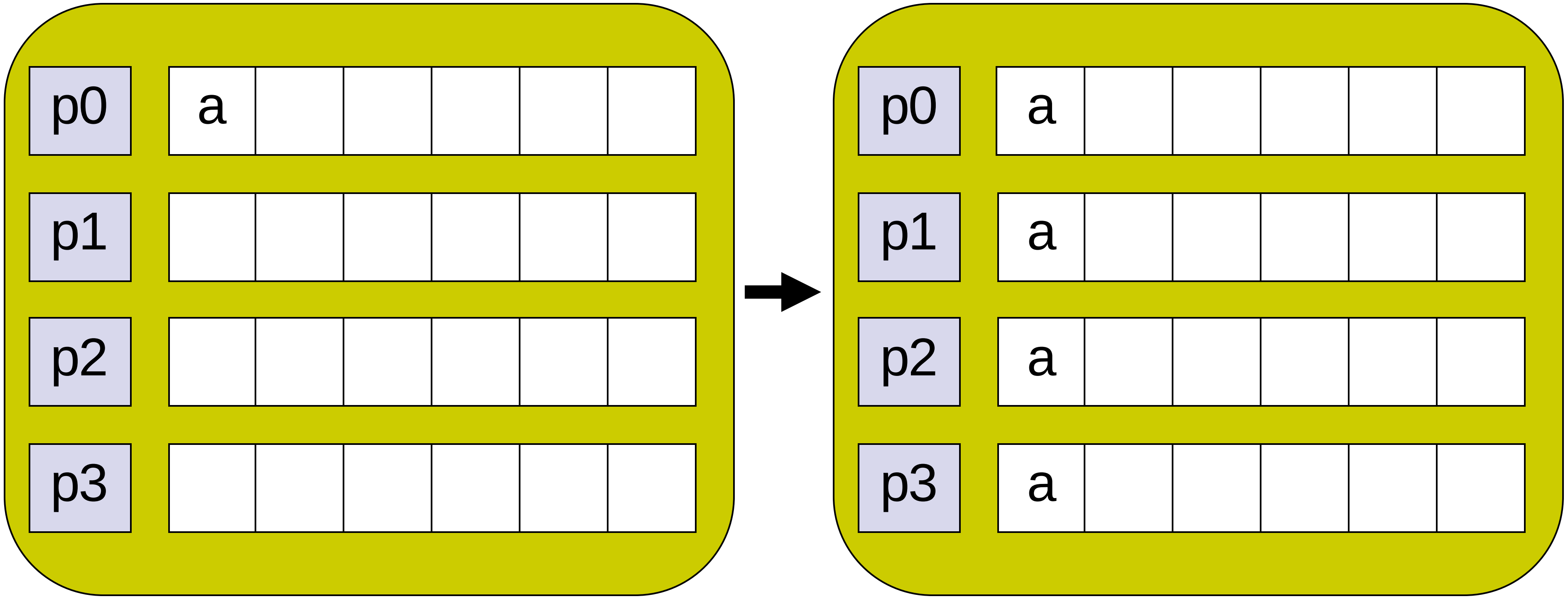
Broadcast

- Broadcast

```
int MPI_Bcast(void* buffer, int count,
  MPI_Datatype datatype, int root,
  MPI_Comm comm);
```

  - Broadcasts a message from the process with rank root to all processes of the group, itself included
  - Called by all members of group using the same arguments for comm, root

Broadcast

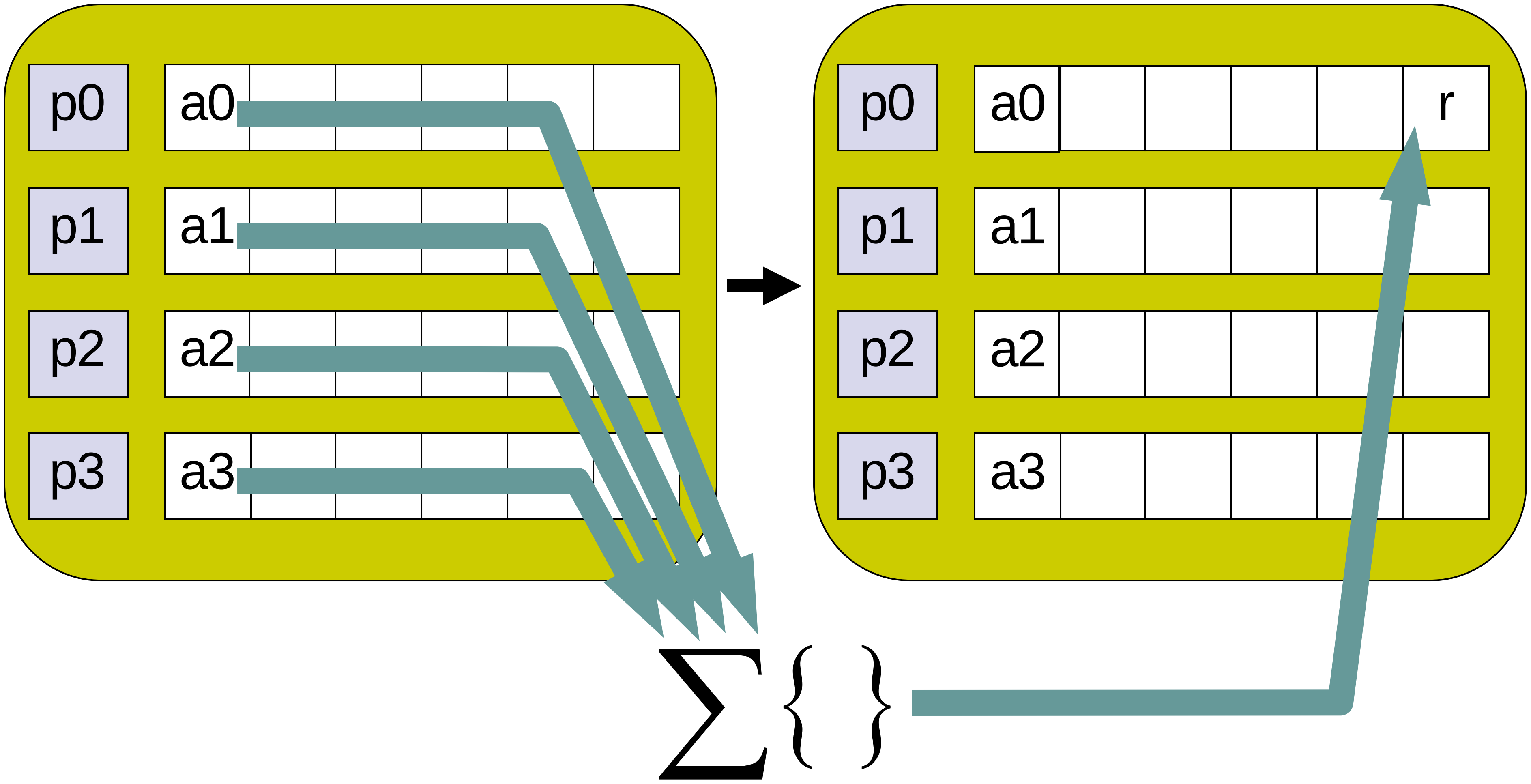


Reduction

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

- Collects data from each process
- Reduces these data to a single value using an operation
- Stores the reduced result on the root process
- A set of predefined operations or user defined
  - MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR, MPI_MINLOC, MPI_MAXLOC

Reduction



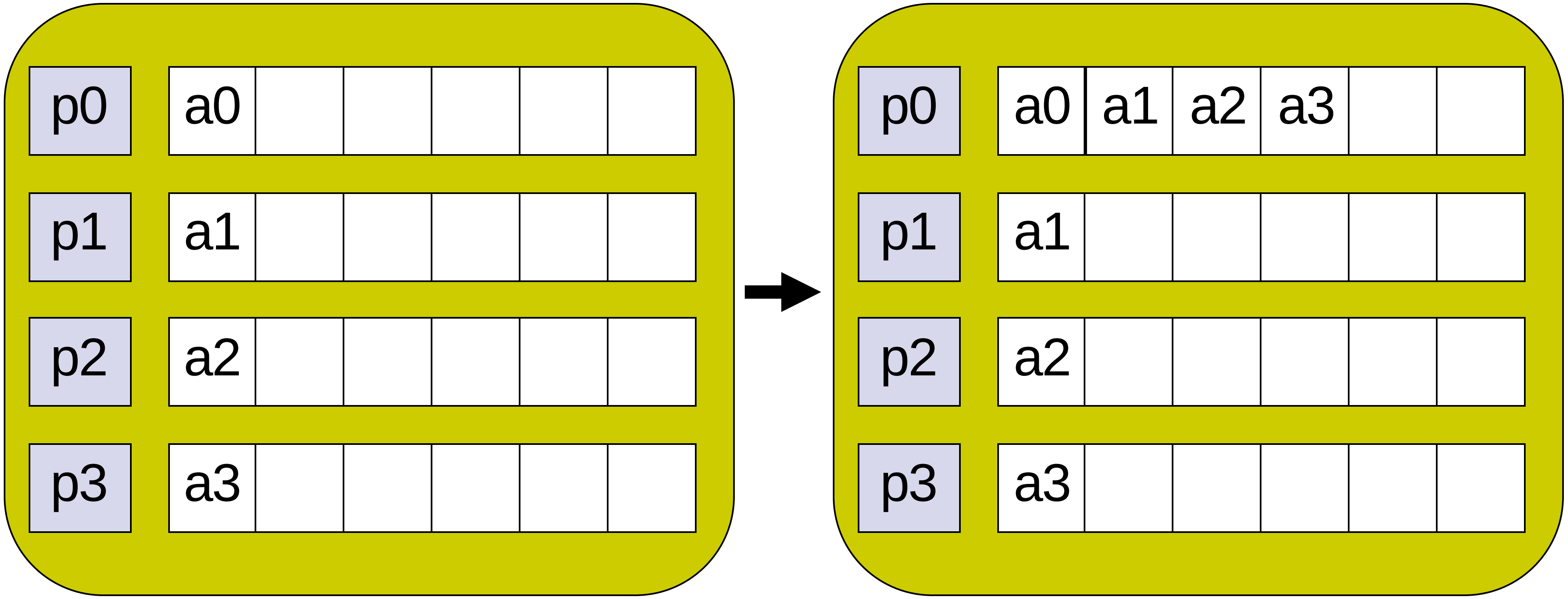
Gather

- Gather

```
int MPI_Gather(const void* sendbuf, int sendcount,
   MPI_Datatype sendtype, void* recvbuf, int
   recvcount, MPI_Datatype recvtype, int root,
   MPI_Comm comm)
int MPI_Gatherv(const void* sendbuf, int
   sendcount, MPI_Datatype sendtype, void*
   recvbuf, const int recvcounts[], const int
   displs[], MPI_Datatype recvtype, int root,
   MPI_Comm comm)
```

  - All-to-one communication
  - The receive arguments are only meaningful to the root process
  - Each process (including the root) sends the contents of the send buffer to the root
  - The root process receives the messages and stores them in rank order

Gather



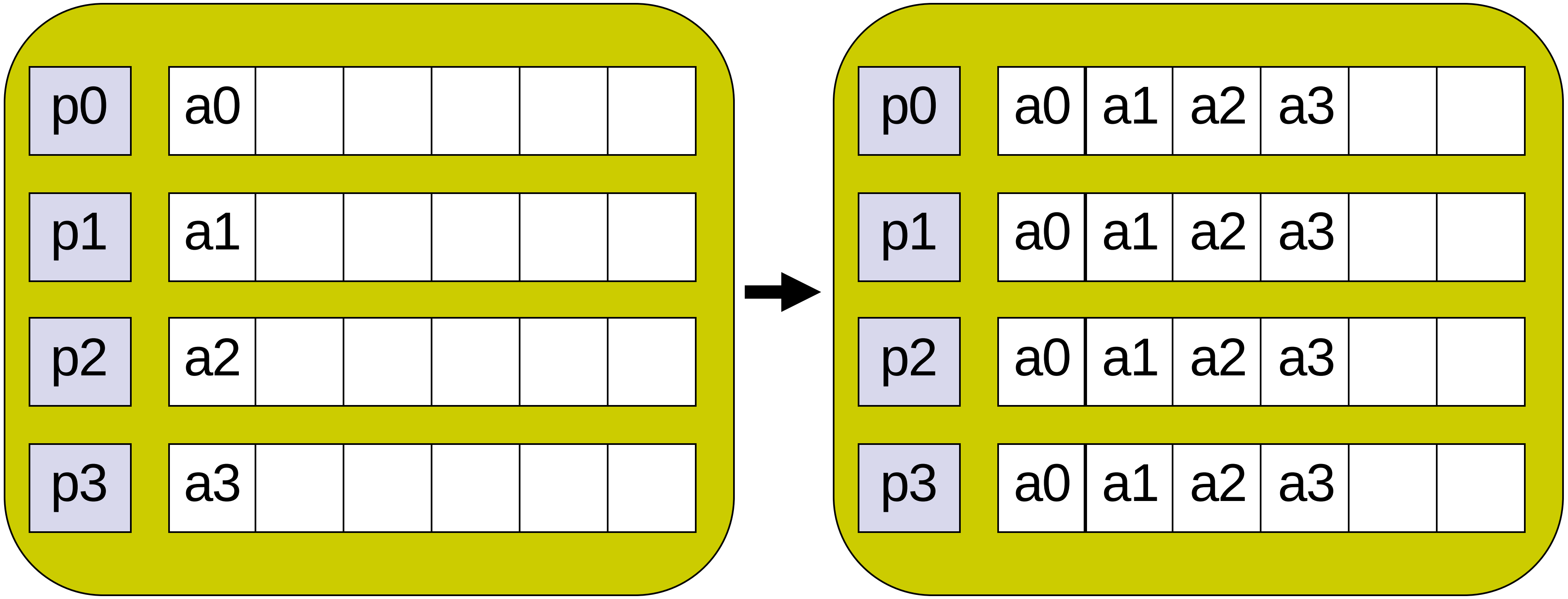
Allgather

- Allgather

```
int MPI_Allgather(const void* sendbuf, int
   sendcount, MPI_Datatype sendtype, void*
   recvbuf, int recvcount, MPI_Datatype
   recvtype, MPI_Comm comm)
int MPI_Allgatherv(const void* sendbuf, int
   sendcount, MPI_Datatype sendtype, void*
   recvbuf, const int recvcounts[], const int
   displs[], MPI_Datatype recvtype, MPI_Comm
   comm)
```

  - After the data are gathered into root process, they are broadcasted to all processes
  - No root process specified
  - Send and receive arguments meaningful to all processes

Allgather


p0
a0
p1
a1
p2
a2
p3
a3
p0
a0 a1 a2 a3
p1
a0 a1 a2 a3
p2
a0 a1 a2 a3
p3
a0 a1 a2 a3

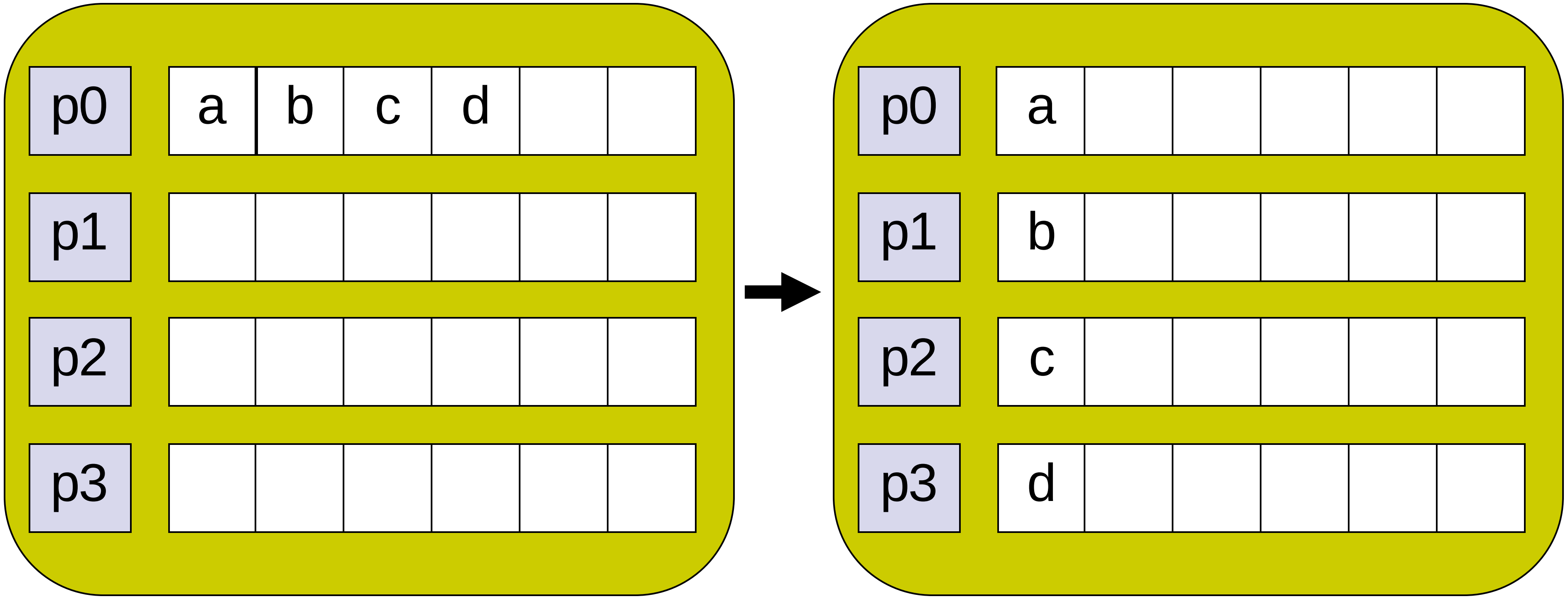
Scatter

- Scatter

```
int MPI_Scatter(const void* sendbuf, int
   sendcount, MPI_Datatype sendtype, void*
   recvbuf, int recvcount, MPI_Datatype
   recvtype, int root, MPI_Comm comm)
int MPI_Scatterv(const void* sendbuf, const
   int sendcounts[], const int displs[],
   MPI_Datatype sendtype, void* recvbuf, int
   recvcount, MPI_Datatype recvtype, int root,
   MPI_Comm comm)
```

  - One-to-all communication
  - Different data are sent from the root process to each process in rank order
  - The send arguments are only meaningful to the root process

Scatter



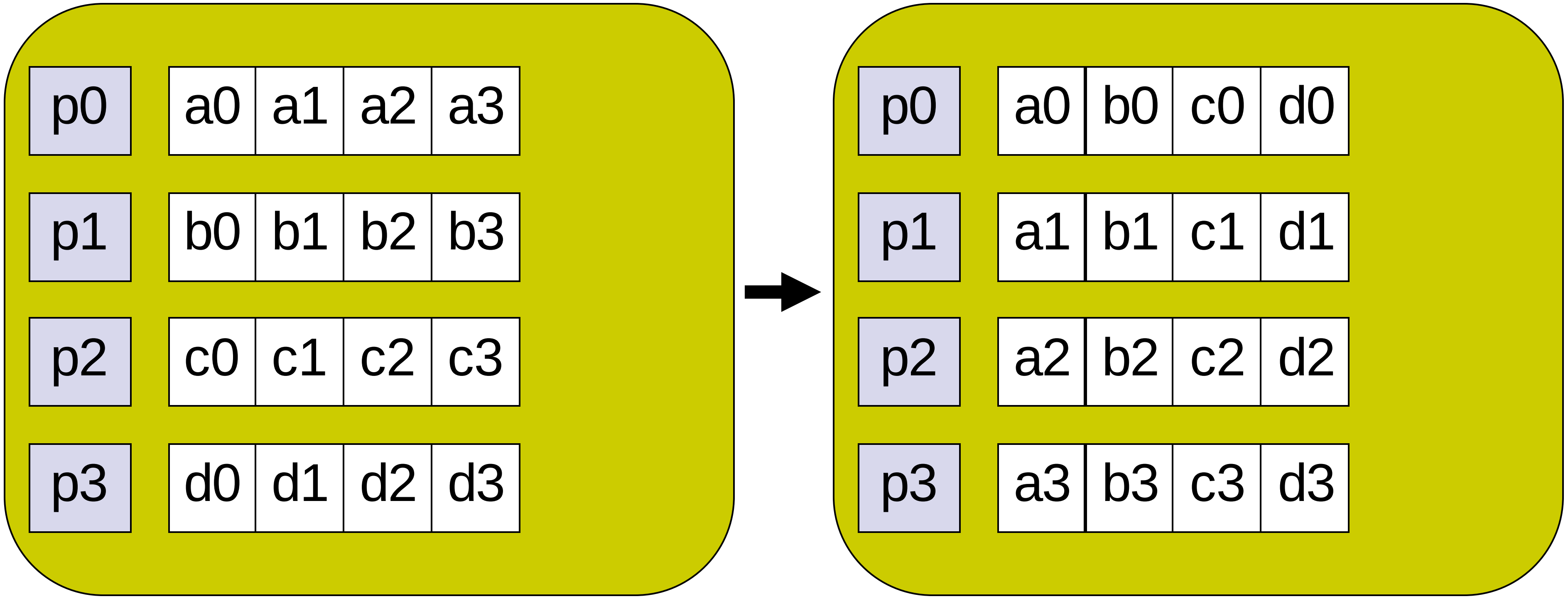
Alltoall

- Alltoall

```
int MPI_Alltoall(const void* sendbuf, int
   sendcount, MPI_Datatype sendtype, void*
   recvbuf, int recvcount, MPI_Datatype
   recvtype, MPI_Comm comm)
int MPI_Alltoallv(const void* sendbuf, const
   int sendcounts[], const int sdispls[],
   MPI_Datatype sendtype, void* recvbuf, const
   int recvcounts[], const int rdispls[],
   MPI_Datatype recvtype, MPI_Comm comm)
int MPI_Alltoallw(const void* sendbuf, const
   int sendcounts[], const int sdispls[],
   const MPI_Datatype sendtypes[], void*
   recvbuf, const int recvcounts[], const int
   rdispls[], const MPI_Datatype recvtypes[],
   MPI_Comm comm)
```

  - Scatters data to other processes, gather data from them
  - Matrix transposition

Alltoall


p0
a0 a1 a2 a3
p1
b0 b1 b2 b3
p2
c0 c1 c2 c3
p3
d0 d1 d2 d3
p0
a0 b0 c0 d0
p1
a1 b1 c1 d1
p2
a2 b2 c2 d2
p3
a3 b3 c3 d3

Other collective operations

- MPI_Allreduce
  - Combine the elements of each process’s input buffer
  - Stores the combined value on the receive buffer of all group members
- MPI_Scan, MPI_Excscan
  - A prefix reduction on data distributed across the group
- MPI_Reduce_scatter
  - Combines MPI_Reduce and MPI_Scatter