Programming with MPI

Parallel Programming

2024-06-12

Prof. Dr. Michael Kuhn
michael.kuhn@ovgu.de

Parallel Computing and I/O
Institute for Intelligent Cooperating Systems
Faculty of Computer Science
Otto von Guericke University Magdeburg
https://parcio.ovgu.de

Outline

Programming with MPI

- Review
- Introduction
- History
- Groups and Communicators
- Point-To-Point Communication
- Collective Communication
- Derived Datatypes
- Summary

- Which aspect is not part of the POSIX Threads standard?
    1. Thread management
    2. Mutexes
    3. Semaphores
    4. Condition variables
    5. Synchronization

- Which thread-to-task mapping does Linux use?
    1. 1:1 mapping (each thread is mapped to a kernel task)
    2. n:1 mapping (all threads are mapped to one kernel task)
    3. m:n mapping (multiple threads are mapped to multiple kernel tasks)

- What happens if a thread is not joined after termination?
  1. The process can crash
  2. Zombie threads remain
  3. Stack memory can overflow

- When will a thread be canceled by `pthread_cancel`?
  1. Before the next function call
  2. After the next function call
  3. After a timeout of 100 ms
  4. After an I/O operation such as `printf`
  5. When a cancellation point function is called

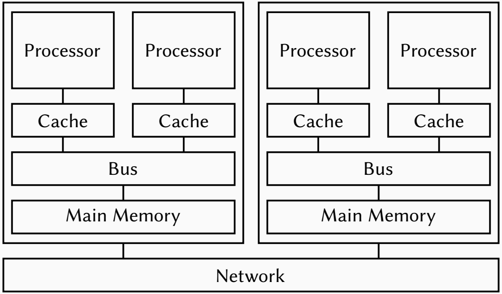
Outline

Programming with MPI

- Review
- Introduction
- History
- Groups and Communicators
- Point-To-Point Communication
- Collective Communication
- Derived Datatypes
- Summary

Motivation

- Shared memory systems have limited scalability
  - Two to four processors with a few dozen cores
- Complex problems require more nodes
  - Distributed memory can be scaled arbitrarily
- Nodes are connected via a network
  - Determines scalability and performance
- Different network technologies and topologies
  - Major competitors: Ethernet and InfiniBand



Motivation…

- OpenMP is a convenient and high-level programming concept
    - It is limited to shared memory systems
- Parallel applications across multiple nodes require message passing
    - Message Passing Interface (MPI) provides necessary functionality
- MPI supports basic and complex operations
    - Sending, receiving, reduction etc.
    - Process groups and synchronization
    - Point-to-point, collective or one-sided communication
- MPI also offers parallel I/O
    - Concurrent access to shared files

Message Passing Interface

- MPI is a standard by the MPI Forum
  - Over 40 participating organizations
  - First standardized and vendor-independent API
  - MPI is not a library but a specification of one
- There are multiple implementations of the standard
  - MPICH, MVAPICH, OpenMPI, Intel MPI etc.
  - Vendors often provide their own implementations

Message Passing Interface…

- MPI implementations are not necessarily binary-compatible
  - They have the same API but different ABIs
  - Compiling an application works with any implementation
- Running compiled application requires original implementation
  - Different implementations might have different constants etc.
  - Way to start processes on different nodes might differ
- Some implementations promise ABI compatibility
  - MPICH ABI Compatibility Initiative for MPICH, Intel MPI, Cray MPT, MVAPICH2, Parastation MPI and RIKEN MPI [MPICH Collaborators, 2023]

Challenges

- Parallel applications now run as independent processes
    - Processes can only access their own data, no shared memory
    - No risk of overwriting other processes' data accidentally
    - Results have to be communicated between processes
- Application code is typically still contained in one file
    - MPI allows us to write a generic version of the application
    - We can determine our rank and the number of processes

Challenges...

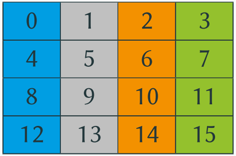
- MPI applications often use SPMD
  - All tasks execute same application but at different points
  - Tasks use different data (domain decomposition)
  - Additional logic to execute only parts of the application
- Decomposition is critical for achievable performance
  - Rows might be faster than columns depending on memory layout
  - Size of sub-domains determines load of each task
- Distribution also determines communication schema
  - Communication might have to be performed at boundaries

Challenges...

- MPI applications often use SPMD
  - All tasks execute same application but at different points
  - Tasks use different data (domain decomposition)
  - Additional logic to execute only parts of the application
- Decomposition is critical for achievable performance
  - Rows might be faster than columns depending on memory layout
  - Size of sub-domains determines load of each task
- Distribution also determines communication schema
  - Communication might have to be performed at boundaries

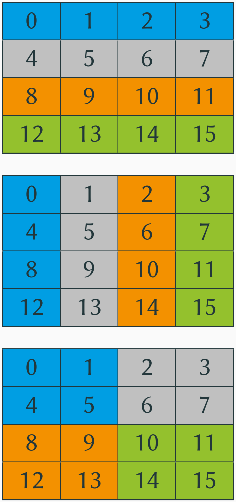
Challenges. . .

- MPI applications often use SPMD
  - All tasks execute same application but at different points
  - Tasks use different data (domain decomposition)
  - Additional logic to execute only parts of the application
- Decomposition is critical for achievable performance
  - Rows might be faster than columns depending on memory layout
  - Size of sub-domains determines load of each task
- Distribution also determines communication schema
  - Communication might have to be performed at boundaries



Challenges...

- MPI applications often use SPMD
  - All tasks execute same application but at different points
  - Tasks use different data (domain decomposition)
  - Additional logic to execute only parts of the application
- Decomposition is critical for achievable performance
  - Rows might be faster than columns depending on memory layout
  - Size of sub-domains determines load of each task
- Distribution also determines communication schema
  - Communication might have to be performed at boundaries



Challenges…

- Application has to be made available on multiple nodes
  - This is normally achieved by using a common file system on all nodes
  - For instance, an NFS file system can be mounted everywhere
- Processes have to be started on participating nodes
  - Many implementations include support for spawning processes via SSH
  - The batch scheduler can also take care of it, requires coordination
- Processes have to locate each other and coordinate
  - Similar to previous point, implementation often takes care of both
  - If the scheduler is involved, it has to pass information to the implementation
  - Process Management Interface (PMI) is typically used to connect components

Outline

Programming with MPI

- Review
- Introduction
- History
- Groups and Communicators
- Point-To-Point Communication
- Collective Communication
- Derived Datatypes
- Summary

- MPI is the current de-facto standard in HPC
  - Previously, Parallel Virtual Machine (PVM) was widely used
- MPI is developed by the MPI Forum, started in 1992
  - MPI-1.0 in 1995: Basic features, communication only
  - MPI-2.0 in 1997: Additional features, including I/O
  - MPI-3.0 in 2012: Better support for one-sided communication
  - MPI-4.0 in 2021: Large-count routines, persistent collectives
- Standard is important for portability across different systems
  - MPI also offers high performance and convenience

Interface

- MPI standard defines an API for C and Fortran
  - C++ used to be available but has been deprecated
  - Bindings are also available for Python, Java etc.
- Abstraction to support efficient communication and I/O
  - Functions have to be high-level enough to be able to apply optimizations
- Standard allows thread-safe implementations but does not require them
  - MPI implementations are typically thread-unsafe by default
  - Thread-safety does have a performance impact due to locking etc.

Interface...

- MPI defines syntax and semantics
  - Syntax determines arguments, semantics how a function behaves
- Example: Function for sending data
  - Standard includes description of behavior and rationale
    - "The send call [...] is blocking: it does not return until the message data and envelope have been safely stored away so that the sender is free to modify the send buffer." [Message Passing Interface Forum, 2015]
  - Abstract: `MPI_SEND(buf, count, datatype, dest, tag, comm)`
    - Arguments are annotated as IN/OUT/INOUT and described
  - C: `int MPI_Send(const void* buf, ...)`
    - Return value via normal method
  - Fortran: `MPI_Send(buf, ..., ierror)`
    - Return value via extra argument (`ierror`)

Interface…

- Non-blocking
  - Call returns before operation has been completed
  - User might not be allowed to reuse specified resources (for example, buffers)
- Blocking
  - User is allowed to reuse resources
- Local
  - Completion of a call depends only on the local process
- Non-local
  - Completion of a call might depend on remote processes
  - Communication might be required to happen before completion
- Collective
  - All processes in a communicator have to be involved in a call

History

- 1992: “Standards for Message Passing in a Distributed Memory Environment”
    - Working group established and prepares draft for MPI-1
    - Group consists of 175 people from 40 organizations
- 1994: MPI-1.0 is released
    - MPI-1.1 in 1995, MPI-1.2 in 1997 and MPI-1.3 in 2008
    - Point-to-point and collective communication
    - Groups, communicators and topologies
    - Environment checks
    - Profiling interface

History…

- 1998: MPI-2.0 is released
  - MPI-2.1 in 2008 and MPI-2.2 in 2009
  - One-sided communication
  - Dynamic process management
  - Parallel I/O
- 2012: MPI-3.0 is released
  - MPI-3.1 in 2015
  - Improved one-sided communication
  - Non-blocking collectives

History...

- 2021: MPI-4.0 is released
  - Large-count versions of many routines
  - Persistent collectives
  - Partitioned communication

Compiler Support

- MPI implementations consist of headers and libraries
  - Main header (`mpi.h`) has to be included
  - Applications have to be linked to MPI libraries
- MPI provides own compilers for convenience
  - `mpicc` for C and `mpifort` for Fortran
  - These are usually compiler wrappers around the underlying compiler
- Compiler wrappers take care of linking etc.
  - Compiler flags can usually be extracted if linking should be done manually

Outline

Programming with MPI

- Review
- Introduction
- History
- Groups and Communicators
- Point-To-Point Communication
- Collective Communication
- Derived Datatypes
- Summary

Initialization

- MPI needs to be initialized and finalized
  - Has to be done manually
  - Do as little as possible before and after
- MPI_Init expects application's arguments
  - MPI might parse certain arguments
  - It is possible to pass NULL to ignore

```c
int main(void) {
    MPI_Init(NULL, NULL);
    hello();
    MPI_Finalize();

    return 0;
}
```

Initialization

- MPI uses communicators
  - Basically a group of processes
- We can determine our rank
  - Same as OpemMP's thread ID
- We can query the communicator's size
  - This is the total amount of processes

```
void hello(void) {
    int rank;
    int size;

    MPI_Comm_rank(MPI_COMM_WORLD,
                  &rank);
    MPI_Comm_size(MPI_COMM_WORLD,
                  &size);

    printf("Hello from %d/%d.\n",
           rank, size);
}
```

Initialization

- We can start the application directly
  - It will usually start with one process
- mpiexec allows spawning more processes
  - Optional and specified by the standard
  - There is also often mpirun
- The -n argument is standardized
  - Implementations provide additional ones

```
$ ./hello
Hello from 0/1.

$ mpiexec -n 1 ./hello
Hello from 0/1.

$ mpiexec -n 4 ./hello
Hello from 0/4.
Hello from 3/4.
Hello from 1/4.
Hello from 2/4.
```

Multi-Threaded Initialization

- MPI_Init only allows serial processes
  - That is, no threads are allowed to run
- Thread-safety requires locks
  - MPI is tuned for high performance
  - Locking overhead should be avoided
- MPI_Init_thread allows requesting a thread-safety level
  - Implementations may not support all

```
int main(void) {
    int thread_level;

    MPI_Init_thread(NULL, NULL,
        MPI_THREAD_MULTIPLE,
        &thread_level);

    printf("thread_level=%d\n",
        thread_level);

    MPI_Finalize();
    return 0;
}
```

Multi-Threaded Initialization

- MPI_THREAD_SINGLE
  - Only one thread will run
- MPI_THREAD_FUNNELED
  - Process can be multi-threaded but only the main thread will make MPI calls
- MPI_THREAD_SERIALIZED
  - All threads can make MPI calls but not at the same time
- MPI_THREAD_MULTIPLE
  - Threads can make MPI calls in parallel

```c
int main(void) {
    int thread_level;

    MPI_Init_thread(NULL, NULL,
        MPI_THREAD_MULTIPLE,
        &thread_level);

    printf("thread_level=%d\n",
        thread_level);

    MPI_Finalize();
    return 0;
}
```

Multi-Threaded Initialization

- `MPI_THREAD_SINGLE`
  - Only one thread will run
- `MPI_THREAD_FUNNELED`
  - Process can be multi-threaded but only the main thread will make MPI calls
- `MPI_THREAD_SERIALIZED`
  - All threads can make MPI calls but not at the same time
- `MPI_THREAD_MULTIPLE`
  - Threads can make MPI calls in parallel

```
$ ./init_thread
thread_level=3

$ mpiexec -n 4 ./init_thread
thread_level=3
thread_level=3
thread_level=3
thread_level=3
```

Miscellaneous

- `MPI_Get_processor_name`
  - Returns an implementation-defined processor name
  - This typically returns the hostname of the current node
- `MPI_Initialized`
  - Checks whether MPI has been initialized
  - Useful if libraries want to check for MPI support
- `MPI_Wtime`
  - Returns wall-clock time for time measurements
- `MPI_Wtick`
  - Returns resolution of `MPI_Wtime`

Rationale

- Communicators allow separating different sets of processes
  - Groups contain processes
  - Communicators are based on groups
- All processes are available by default (MPI_COMM_WORLD)
  - Ranks are numbered from 0 to n-1
- Communicators can be used to define independent contexts
  - For instance, MPI-aware library should not interfere with application
- Some operations should only be performed by the local process
  - If they require a communicator, MPI_COMM_SELF can be used

Communicators

- Chicken and egg problem
    - Creating new communicator requires an existing communicator
    - MPI_COMM_WORLD can be used
- Processes can have multiple ranks
    - Rank only valid in a communicator
    - Processes can belong to multiple groups and communicators

```
void comm(void) {
    MPI_Comm new_comm;
    MPI_Group new_group;
    MPI_Group world_group;

    MPI_Comm_group(MPI_COMM_WORLD,
        &world_group);
    MPI_Group_incl(world_group,
        size, reverse_ranks,
        &new_group);
    MPI_Comm_create(MPI_COMM_WORLD,
        new_group, &new_comm);

    print_rank(new_comm);
}
```

Communicators

- Chicken and egg problem
  - Creating new communicator requires an existing communicator
  - MPI_COMM_WORLD can be used
- Processes can have multiple ranks
  - Rank only valid in a communicator
  - Processes can belong to multiple groups and communicators

```c
void print_rank(MPI_Comm comm) {
    int new_rank;

    MPI_Comm_rank(comm, &new_rank);
    printf("rank=%d (world=%d)\n",
        new_rank, rank);
}
```

```
$ mpiexec -n 4 ./comm
rank=3 (world=0)
rank=2 (world=1)
rank=1 (world=2)
rank=0 (world=3)
```

Outline

Programming with MPI

- Review
- Introduction
- History
- Groups and Communicators
- Point-To-Point Communication
- Collective Communication
- Derived Datatypes
- Summary

Guarantees

- Message order is guaranteed
  - If a process sends two messages, the first one will be received first
  - If a process posts two receives, the first one will get the message
- Rules do not apply when multi-threaded
  - If two threads send one message each, their order is undefined
  - Would require coordinating threads, that is, introduce overhead
- There are no fairness guarantees
  - A message might never be received because of other matching messages

Sending and Receiving

- Point-to-point between two processes
- Sending
  - Buffer: Data to send
  - Count: Number of elements
  - Datatype: Type of elements
  - Destination: Target rank
  - Tag: Distinguish messages
  - Communicator: Process mapping

```
void mysend(void) {
    char str[100];
    snprintf(str, 100,
        "Hello from %d\n", rank);

    MPI_Send(str, 100, MPI_CHAR,
        (rank + 1) % size,
        0, MPI_COMM_WORLD);
    MPI_Recv(str, 100, MPI_CHAR,
        (size + rank - 1) % size,
        0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);

    printf("%d: %s", rank, str);
}
```

Sending and Receiving

- Point-to-point between two processes
- Receiving
    - Buffer: Where to receive data
    - Count: Number of elements
    - Datatype: Type of elements
    - Source: Source rank
    - Tag: Distinguish messages
    - Communicator: Process mapping
    - Status: Query information

```c
void mysend(void) {
    char str[100];
    snprintf(str, 100,
        "Hello from %d\n", rank);

    MPI_Send(str, 100, MPI_CHAR,
        (rank + 1) % size,
        0, MPI_COMM_WORLD);
    MPI_Recv(str, 100, MPI_CHAR,
        (size + rank - 1) % size,
        0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);

    printf("%d: %s", rank, str);
}
```

- Point-to-point between two processes
- Ring communication
  - Send to next process
  - Receive from previous process
  - Output order might be mixed

```
$ mpiexec -n 4 ./send
1: Hello from 0
0: Hello from 3
3: Hello from 2
2: Hello from 1
```

Sending and Receiving…

- Might not be clear from which process to receive
  - Functions require specifying a source rank and tag
- Wildcards allow matching any source or any tag
  - `MPI_ANY_SOURCE` instead of actual source rank
  - `MPI_ANY_TAG` instead of actual source tag
- We still might be interested to know which rank and tag a message came from
  - Can be queried via `MPI_Status`'s `MPI_SOURCE` and `MPI_TAG` members
- `MPI_Get_count` returns the number of received elements

- What happens if we send 100,000 bytes?
  1. The same as with 100
  2. Application deadlocks
  3. Crash due to stack overflow
  4. MPI warns about too many elements

```c
void mysend(void) {
    char str[100];
    snprintf(str, 100,
        "Hello from %d\n", rank);

    MPI_Send(str, 100, MPI_CHAR,
        (rank + 1) % size,
        0, MPI_COMM_WORLD);
    MPI_Recv(str, 100, MPI_CHAR,
        (size + rank - 1) % size,
        0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);

    printf("%d: %s", rank, str);
}
```

Blocking Send

- MPI_Send is the default blocking send function
  - Standard allows using a buffer but does not mandate it
  - "The send call [...] uses the standard communication mode. In this mode, **it is up to MPI to decide whether outgoing messages will be buffered**. [...] In such a case, the send call may complete before a matching receive is invoked. On the other hand, [...] MPI may choose not to buffer outgoing messages, for performance reasons. In this case, the send call will not complete until a matching receive has been posted, and the data has been moved to the receiver. [...] The standard mode send is non-local: successful completion of the send operation may depend on the occurrence of a matching receive." [Message Passing Interface Forum, 2015]
- Buffering is typically only used for small messages
  - Larger messages make the send operation synchronous

Blocking Send…

- There are a number of different send/receive variants
    - Synchronous send (MPI_Ssend)
        - Blocks until the destination process has started to receive the message
        - Behaves like MPI_Send for large messages
    - Blocking and non-blocking (MPI_Send and MPI_Isend)
        - Blocking behavior specifies when calls return and buffers can be reused
        - Non-blocking allows overlapping communication with computation
    - Buffered (MPI_Bsend)
        - Data is explicitly buffered, buffers have to be provided manually
        - Behaves like MPI_Send for small messages
    - Ready send (MPI_Rsend)
        - Requires matching receive operation to be started already, otherwise undefined
    - Combined blocking send and receive (MPI_Sendrecv)
        - Avoids deadlocks due to blocking sends waiting for receives to be posted

Non-Blocking Send

- Non-blocking send does not deadlock
  - I stands for immediate
- MPI_Wait blocks until completion
  - Functions to wait for multiple requests (all, any or some)
  - It is an error not to wait or access the buffer before the send has finished
- Alternatively, MPI_Test or MPI_Probe

```c
void mysend(char* str, char* buf) {
    MPI_Request req;

    MPI_Isend(str, 100000, MPI_CHAR,
        (rank + 1) % size,
        0, MPI_COMM_WORLD, &req);
    MPI_Recv(buf, 100000, MPI_CHAR,
        (size + rank - 1) % size,
        0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
    MPI_Wait(&req,
        MPI_STATUS_IGNORE);

    printf("%d: %s", rank, buf);
}
```

Non-Blocking Send

- Non-blocking send does not deadlock
  - I stands for immediate
- MPI_Wait blocks until completion
  - Functions to wait for multiple requests (all, any or some)
  - It is an error not to wait or access the buffer before the send has finished
- Alternatively, MPI_Test or MPI_Probe

```
$ mpiexec -n 4 ./isend
2: Hello from 1.
0: Hello from 3.
1: Hello from 0.
3: Hello from 2.
```

Combined Send and Receive

- Combined blocking send and receive
  - Still blocking but avoids deadlock
- Abstraction to achieve typical use case
  - For example, send to and receive from neighboring processes
  - Implementation can handle this specific use case efficiently and correctly

```c
void mysend(void) {
    char str[100000];
    char buf[100000];
    snprintf(str, 100000,
        "Hello from %d.\n", rank);

    MPI_Sendrecv(str, 100000,
        MPI_CHAR, (rank + 1) % size,
        0, buf, 100000, MPI_CHAR,
        (size + rank - 1) % size,
        0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);

    printf("%d: %s", rank, buf);
}
```

Combined Send and Receive

- Combined blocking send and receive
  - Still blocking but avoids deadlock
- Abstraction to achieve typical use case
  - For example, send to and receive from neighboring processes
  - Implementation can handle this specific use case efficiently and correctly

```
$ mpiexec -n 4 ./sendrecv
0: Hello from 3.
3: Hello from 2.
1: Hello from 0.
2: Hello from 1.
```

Overview

	Send	Receive	Testing
Blocking	MPI_Send MPI_Ssend MPI_Rsend	MPI_Recv	MPI_Probe MPI_Wait
	MPI_Sendrecv		
Non-blocking	MPI_Isend MPI_Issend	MPI_Irecv	MPI_Iprobe MPI_Test

- Most functions are available as blocking and non-blocking versions
    - There are also non-blocking synchronous functions
- Blocking is easier to use, non-blocking is more efficient

Outline

Programming with MPI

- Review
- Introduction
- History
- Groups and Communicators
- Point-To-Point Communication
- Collective Communication
- Derived Datatypes
- Summary

Overview

- Point-to-point communication happens between two ranks
  - Collective communication happens between all ranks
- Which ranks are involved depends on communicator
  - By default, we only have `MPI_COMM_WORLD` and `MPI_COMM_SELF`
- MPI contains a wide range of collective communication functions
  - Broadcast
  - Barrier
  - Distributing or collecting data
- One collective call is often more efficient than many point-to-point calls
  - InfiniBand hardware typically has support for efficient collectives

Overview…

- 1:1 communication
  - Traditional point-to-point communication such as send and receive
- 1:n communication
  - Collective communication such as broadcast
- n:1 communication
  - Collective communication such as reduction
- n:n communication
  - Collective communication such as reduction to all

Overview…

P0	A	B	C
P1			
P2			

Broadcast

$\rightarrow$

P0	A	B	C
P1	A	B	C
P2	A	B	C

Overview...

P0	A	B	C
P1			
P2			

Broadcast →

P0	A	B	C
P1	A	B	C
P2	A	B	C

P0	A	B	C
P1			
P2			

Scatter →

P0	A	B	C
P1	B		
P2	C		

Overview...

P0	A	B	C
P1			
P2			

Broadcast →

P0	A	B	C
P1	A	B	C
P2	A	B	C

P0	A	B	C
P1			
P2			

Scatter →

P0	A	B	C
P1	B		
P2	C		

P0	A		
P1	B		
P2	C		

Gather →

P0	A	B	C
P1	B		
P2	C		

Reduction

- Reducing
  - Send buffer: Data to reduce
  - Receive buffer: Root needs separate buffer
  - Count: Number of elements
  - Datatype: Type of elements
  - Operation: Reduction to perform
  - Root: Rank to reduce at
  - Communicator: Process mapping
- Reduction operations known from OpenMP
  - Apply a given function to multiple buffers, reducing it to one buffer
- Ordering is arbitrary, might influence result

```c
void reduce(void) {
    int buf = 42;

    MPI_Reduce(&rank, &buf, 1,
        MPI_INT, MPI_MAX,
        0, MPI_COMM_WORLD);

    printf("%d: %d\n", rank, buf);
}
```

Reduction

- Reducing
  - Send buffer: Data to reduce
  - Receive buffer: Root needs separate buffer
  - Count: Number of elements
  - Datatype: Type of elements
  - Operation: Reduction to perform
  - Root: Rank to reduce at
  - Communicator: Process mapping
- Reduction operations known from OpenMP
  - Apply a given function to multiple buffers, reducing it to one buffer
- Ordering is arbitrary, might influence result

```
$ mpiexec -n 4 ./reduce
0: 3
1: 42
2: 42
3: 42
```

- Reducing to all
  - Send buffer: Data to reduce
  - Receive buffer: Needs separate buffer
  - Count: Number of elements
  - Datatype: Type of elements
  - Operation: Reduction to perform
  - Communicator: Process mapping
- No root rank specified anymore
  - Reduced buffer is available for all ranks

```c
void reduce(void) {
    int buf = 42;

    MPI_Allreduce(&rank, &buf, 1,
        MPI_INT, MPI_MAX,
        MPI_COMM_WORLD);

    printf("%d: %d\n", rank, buf);
}
```

Reduction to All

- Reducing to all
  - Send buffer: Data to reduce
  - Receive buffer: Needs separate buffer
  - Count: Number of elements
  - Datatype: Type of elements
  - Operation: Reduction to perform
  - Communicator: Process mapping
- No root rank specified anymore
  - Reduced buffer is available for all ranks

```
$ mpiexec -n 4 ./allreduce
0: 3
1: 3
2: 3
3: 3
```

- Why not use MPI_Reduce followed by MPI_Broadcast?
  1. More optimization potential
  2. Two collectives could deadlock
  3. Data could be broadcasted before reduction is finished

```
void reduce(void) {
    int buf = 42;

    MPI_Allreduce(&rank, &buf, 1,
        MPI_INT, MPI_MAX,
        MPI_COMM_WORLD);

    printf("%d: %d\n", rank, buf);
}
```

Barrier

- Barrier
  - Communicator: Process mapping
- Waits for all processes
  - Can cause significant overhead
  - Often not necessary due to implicit synchronization via messages

```
void barrier(void) {
    printf("%d: before barrier\n",
        rank);

    MPI_Barrier(MPI_COMM_WORLD);

    printf("%d: after barrier\n",
        rank);
}
```

Barrier

- Barrier
  - Communicator: Process mapping
- Waits for all processes
  - Can cause significant overhead
  - Often not necessary due to implicit synchronization via messages
- Does not work for everything
  - Terminal output might be buffered
  - Output has to be collected from nodes

```
$ mpiexec -n 4 ./barrier
1: before barrier
0: before barrier
2: before barrier
2: after barrier
3: before barrier
3: after barrier
0: after barrier
1: after barrier
```

Outline

Programming with MPI

- Review
- Introduction
- History
- Groups and Communicators
- Point-To-Point Communication
- Collective Communication
- Derived Datatypes
- Summary

Rationale

- MPI supports most basic data types out of the box
  - `char`, `int`, `long`, `float`, `double` etc.
- Applications often use their own data types
  - For instance, structures containing multiple values
- MPI allows handling these data types directly
  - Developers have to replicate the data types for MPI
  - MPI might be able to handle them more efficiently then
- Data types can then be specified like normal ones
  - Every function that accepts a data type also accepts derived ones

Non-Contiguous Data Types

- Example: Diagonal of a 3×3 matrix
  - For instance, within a function doing parallel matrix calculations

Non-Contiguous Data Types

- Example: Diagonal of a 3×3 matrix
  - For instance, within a function doing parallel matrix calculations
- MPI supports a vector data type
  - Count: Number of blocks
  - Block length: Elements per block
  - Stride: Elements between blocks
  - Old type: Old data type
  - New type: New data type

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

Non-Contiguous Data Types

- Example: Diagonal of a 3×3 matrix
  - For instance, within a function doing parallel matrix calculations
- MPI supports a vector data type
  - Count: Number of blocks
  - Block length: Elements per block
  - Stride: Elements between blocks
  - Old type: Old data type
  - New type: New data type

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

```
MPI_Type_vector(3, 1, 4,
    MPI_DOUBLE, &newtype);
MPI_Type_commit(&newtype);
MPI_Send(matrix, 1, newtype,
    rank, 0, MPI_COMM_WORLD);
```

Non-Contiguous Data Types

- Matrix is stored in row- or column-major order
  - 3×3 matrix has three diagonal elements
  - Each diagonal element is a double value
  - Diagonal elements are four values apart
- Can be generalized for arbitrary dimensions
  - Sender and receiver have to agree on data type
- There are many more data type constructors
  - Interactive tools can help create own derived data types [RookieHPC, 2022]

```
MPI_Type_vector(3, 1, 4,
    MPI_DOUBLE, &newtype);
```

1	2	3
4	5	6
7	8	9

Outline

Programming with MPI

Review

Introduction

History

Groups and Communicators

Point-To-Point Communication

Collective Communication

Derived Datatypes

Summary

Summary

- MPI is a standard for parallel programming on distributed memory systems
  - It supports communication, synchronization, I/O and much more
- Groups of processes can be assigned to communicators
  - Allows separating different parts of an application or library
- Point-to-point communication allows sending messages between two processes
  - There are various versions of basic send and receive functions
- Collective communication involves all processes in a communicator
  - This includes actual communication as well as synchronization functionality
- Derived data types allow MPI to handle application-specific data types directly
  - Allows the MPI implementation to make access more convenient and efficient

References

[Barney, 2023] Barney, B. (2023). **Message Passing Interface (MPI).** https://hpc-tutorials.llnl.gov/mpi/.

[Message Passing Interface Forum, 2015] Message Passing Interface Forum (2015). **MPI: A Message-Passing Interface Standard Version 3.1.** https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report/mpi31-report.htm.

[MPICH Collaborators, 2023] MPICH Collaborators (2023). **MPICH ABI Compatibility Initiative.** https://www.mpich.org/abi/.

[RookieHPC, 2022] RookieHPC (2022). **MPI datatype creator.** https://rookiehpc.github.io/mpi/tools/datatype_creator/.