Unit-II

Parallel Programming on CPU: Vectorization, SIMD overview, Hardware trends for vectorization, Vectorization methods, Programming style for better vectorization,

MPI: The basics for an MPI program , The send and receive commands for process-to-process communication , Collective communication , Data parallel examples , Advanced MPI functionality to simplify code and enable optimizations

Vectorization

Vectorization is a technique used in computer science to perform operations on entire arrays or sequences of data elements simultaneously, instead of processing each element individually. It's commonly used in numerical and scientific computing, as well as in various data analysis and machine learning tasks. In Parallel computing, processors have special vector units that can load and operate on more than one data element at a time.

SIMD overview

Vectorization is an example of single instruction, multiple data (SIMD) processing because it executes a single operation (e.g., addition, division) over a large dataset. A scalar operation, in the context of mathematics and computer science, refers to an operation that is performed on a single scalar value, as opposed to a vector, matrix, or any other data structure. Scalars are single numerical values and can be integers, floating-point numbers, or other numerical types.

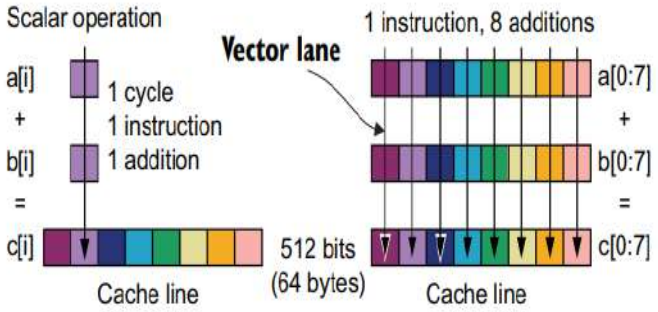


Figure 6.1 A scalar operation does a single double-precision addition in one cycle. It takes eight cycles to process a 64-byte cache line. In comparison, a vector operation on a 512-bit vector unit can process all eight double-precision values in one cycle.

Vectorization Terminology:

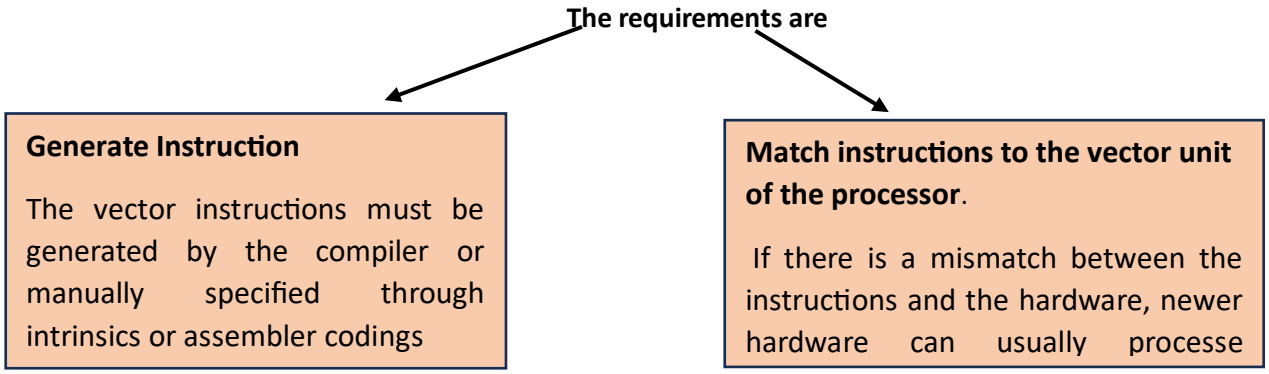
Vector (SIMD) lane : A pathway through a vector operation on vector registers for a single data element much like a lane on a multi-lane free way.

Vector width : The width of the vector unit, usually expressed in bits

Vector length : The number of data elements that can be processed by the vector in one operation.

Vector (SIMD) instruction sets: The set of instructions that extend the regular scalar processor instructions to utilize the vector processor.

Vectorization is produced through both a software and a hardware component.



Hardware trends for vectorization

It is helpful to know the historical dates of hardware and instruction set release for selecting which vector instruction set to use.

Release	Functionality
MMX (trademark with no official meaning)	Targeted towards the graphics market, but GPUs soon took over this function. Vector units shifted their focus to computation rather than graphics. AMD released its version under the name 3DNow! with single-precision support.
SSE (Streaming SIMD Extensions)	First Intel vector unit to offer floating-point operations with single-precision support
SSE2	Double-precision support added
AVX (Advanced Vector Extensions)	Twice the vector length. AMD added a fused multiply-add FMA vector instruction in its competing hardware, effectively doubling the performance for some loops.
AVX2	Intel added a fused multiply-add (FMA) to its vector processor.
AVX512	First offered on the Knights Landing processor; it came to the main-line multi-core processor hardware lineup in 2017. From the years 2018 and on, Intel and AMD (Advanced Micro Devices, Inc.) have created multiple variants of AVX512 as incremental improvements to vector hardware architectures.

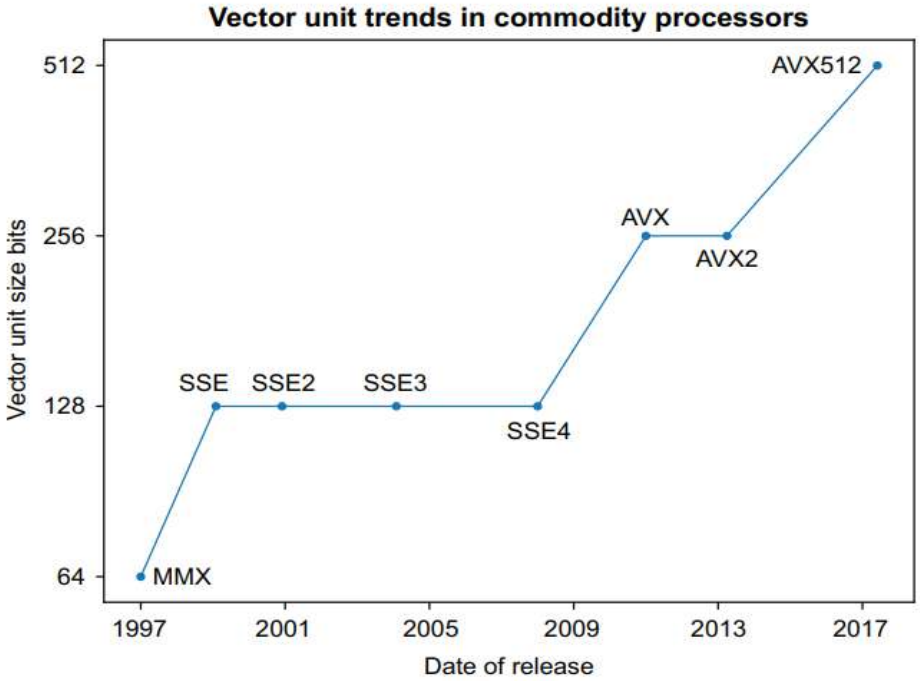


Figure 6.2 The appearance of vector unit hardware for commodity processors began around 1997 and has slowly grown over the last twenty years, both in vector width (size) and in types of operations supported.

Vectorization methods

There are several ways to achieve Vectorization in your program.

- **Optimized libraries :**

Optimized libraries play a crucial role in achieving vectorization and improving the performance of software applications, especially in the context of numerical and scientific computing. These libraries provide pre-implemented and highly optimized functions for common mathematical and linear algebra operations. Some of the most commonly used libraries include

BLAS (Basic Linear Algebra System)—A base component of high-performance linear algebra software

LAPACK—A linear algebra package

SCALAPACK—A scalable linear algebra package

FFT (Fast Fourier transform)—Various implementation packages available

Sparse Solvers—various implementations of sparse solvers available

- **Auto-Vectorization**

Auto-vectorization is a compiler optimization technique that transforms scalar code into vectorized code, taking advantage of SIMD (Single Instruction, Multiple Data) instructions available in modern processors. Most modern compilers provide flags or options to enable or enhance auto-vectorization.

- **Hints to the compiler**

Hints to the compiler are annotations or directives provided by the programmer to guide the compiler's optimization decisions. These hints can inform the compiler about specific optimizations that should be applied to certain parts of the code Different programming languages and compilers have different ways to provide hints to guide vectorization.

1. Pragmas can guide the compiler's vectorization process, helping it identify loops that can be safely and efficiently vectorized.

```
#pragma clang loop vectorize(enable) // Enable loop vectorization
for (int i = 0; i < N; ++i)
{
   // Loop body
}
```

2. In the context of computer programming and compiler optimizations dependencies play a crucial role in determining the order in which instructions or operations can be executed.

Various data Depencies are

A flow dependency, also known as a true dependency(Read After Write-RAW), occurs when an instruction depends on the result of a previous instruction. As a result, the second instruction cannot be executed until the first one completes.

For example, if you have the code **b = a + 1** and then **c = b + 2**, there is a flow dependency from the second instruction to the first because it relies on the result of the first instruction.

An anti-flow dependency, also known as an anti-dependency(Write After Read-WAR), occurs when the order of execution of instructions is crucial to avoid incorrect results.

1. **b = a * 2** (Instruction 1)
2. **a = a + 1** (Instruction 2)

In this example, **Instruction 2** modifies the value of the variable **a**, which is used in **Instruction 1**. If **Instruction 2** were to execute before **Instruction 1**, the value of **a** used in **Instruction 1** would

be incorrect, leading to erroneous results. Therefore, the correct order of execution is **Instruction 1** followed by **Instruction 2**.

An output dependency, also known as a write-after-write dependency (WAW dependency), occurs when two instructions write to the same memory location or register.

Consider the following sequence of instructions:

1. **a = 5** (Instruction 1)
2. **a = a + 3** (Instruction 2)

In this example, both **Instruction 1** and **Instruction 2** write to the same variable **a**. If **Instruction 1** and **Instruction 2** are executed out of order, the final value of **a** depends on the order in which the instructions are executed.

- If **Instruction 1** is executed after **Instruction 2**, **a** will be 5.
- If **Instruction 1** is executed before **Instruction 2**, **a** will be 8.

3. Vectorization of Loops:

A "peel loop" is a term used in the context of loop optimization in computer programming and compilers. Loop peeling refers to the process of extracting one or more iterations from the beginning or end of a loop and handling them separately from the main loop.

```
// Original loop
for (int i = 0; i < N; i++) {
   // Loop body
}
// Peeling the first iteration
// Handle the first iteration separately(This can be beneficial, for example, if the first iteration requires special processing, and the remaining iterations start from index 1.)
// Main loop with iterations from 1 to N-1
for (int i = 1; i < N; i++) {
   // Loop body
```

```
}
```

A "remainder loop" refers to a loop that iterates over the remaining elements of a collection or array after a specific condition is met. It is a common programming construct used to process the remaining elements of a data structure once a certain criterion is satisfied within the loop.

For example, if the vectorized loop trip count is 20 and the vector length is 16, it means every time the kernel loop gets executed once, the remainder 4 iterations have to be executed in the remainder- loop.

The peel loop is added to deal with the unaligned data at the start of the loop, and the remainder loop takes care of any extra data at the end of the loop

- **Vector intrinsic:**

  Vector intrinsics are low-level programming constructs used to write explicit vectorized code by directly utilizing the capabilities of SIMD (Single Instruction, Multiple Data) instructions available on modern processors. Vector intrinsics are typically written in assembly or as inline assembly within a higher-level programming language and are often specific to a particular CPU architecture.

  Intrisics provide data types for vectors (i.e. `__m128 a;` would declare the variable `a` to be a vector of 4 floats). They also provide functions that operate directly on vectors (i.e. `_mm_add_ps(a,b)` would add together the two vectors `a` and `b`).

- **Assembler instructions:** Using assembly language for vectorization involves writing low-level code that directly employs SIMD (Single Instruction, Multiple Data) instructions to perform operations on multiple data elements in parallel.

  The example below demonstrates vectorization using x86 assembly with SSE (Streaming SIMD Extensions) instructions:

```
    array1 dd 1, 2, 3, 4    ; First array of integers
    array2 dd 5, 6, 7, 8    ; Second array of integers
    result dd 0, 0, 0, 0    ; Array to store the result
_start:
    movaps xmm0, [array1]   ;     Load 128-bit (4x32-bit) values from array1 to xmm0
```

```
movaps xmm1, [array2]   ;   Load 128-bit (4x32-bit) values from array2 to xmm1
addps xmm0, xmm1        ;   Perform vectorized addition
movaps [result], xmm0   ;   Store the result back to the result array
Exit
```

Programming style for better Vectorization:

Adopting the following programming styles leads to better performance out of the box and less work needed for optimization efforts.

General suggestions:

- Use the restrict attribute on pointers in function arguments and declarations (C and C++).
- Use pragmas or directives where needed to inform the compiler.
- Be careful with optimizing for the compiler with #pragma unroll and other techniques; you might limit the possible options for the compiler transformations.
- Put exceptions and error checks with print statements in a separate loop.

Concerning data structures:

- Try to use a data structure with a long length for the innermost loop
- Use the smallest data type needed (short rather than int).
- Use contiguous memory accesses.
- Use Structure of Arrays (SOA) rather than Array of Structures (AOS).

Array of Structures (AoS)(structure variable is an array)

```
struct person {
char gender;
int age;
} s[5];
```

Structure of Arrays (SoA)(Structure member is an array)

```
struct person {
char name[60];
char gender;
int age;
} ;
```

- Use memory-aligned data structures where possible.

Related to loop structures:

- Use simple loops without special exit conditions.
- Make loop bounds a local variable by copying global values and then using them.
- Use the loop index for array addresses when possible.
- Expose the loop bound size so it is known to the compiler. If the loop is only three iterations long, the compiler might unroll the loop rather than generate a four-wide vector instruction.
- Avoid array syntax in performance-critical loops (FORTRAN).

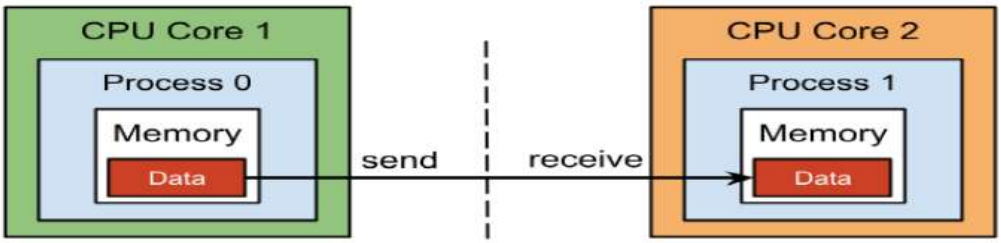
In the loop body:

- Define local variables within a loop so that it is clear that these are not carried to subsequent iterations (C and C++).
- Variables and arrays within a loop should be write-only or read-only (only on the left side of the equal sign or on the right side, except for reductions).
- Don't reuse local variables for a different purpose in the loop—create a new variable. The memory space you waste is far less important than the confusion this creates for the compiler.
- Avoid function calls and inline instead (manually or with the compiler).

MPI: The basics for an MPI program , The send and receive commands for process-to-process communication , Collective communication , Data parallel examples , Advanced MPI functionality to simplify code and enable optimizations

The basics for an MPI program

MPI (Message Passing Interface) is a standard communication protocol used in parallel computing environments to enable processes running on different processors or nodes to communicate and coordinate their actions. It is commonly used in high-performance computing (HPC) and cluster computing applications where multiple computing nodes work together to solve a complex problem.



MPI allows programs to be written in a distributed-memory programming model, where each process has its own local memory space and communicates with other processes using message passing. Processes can send and receive messages, making it possible for them to exchange data and synchronize their execution.

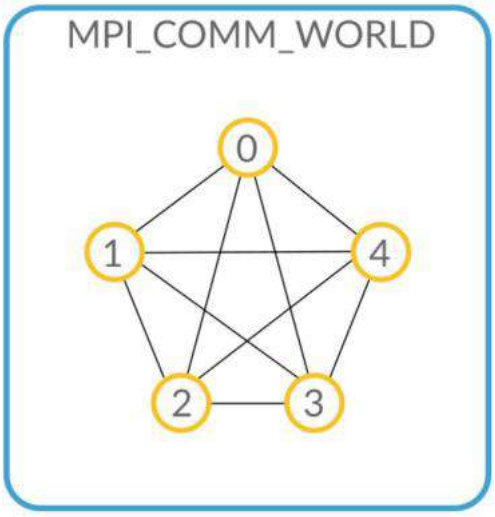
In the context of MPI (Message Passing Interface) programming, the terms "communication world" and "rank" are fundamental concepts used to manage communication and coordination among parallel processes in a parallel computing environment.

1. Communication World:

- A communication world, also known as a communicator, is a group of MPI processes that can communicate with each other.
- MPI_COMM_WORLD is the default communicator that includes all processes created when the MPI application starts.
- Communicators allow processes to be organized into groups, enabling more controlled and specific communication patterns.

2. Rank:

- Rank refers to the unique identifier assigned to each process within a communicator.
- In MPI_COMM_WORLD, ranks range from 0 to (number of processes - 1).
- Ranks are used to distinguish one process from another within the same communicator.
- Processes can communicate with each other using their ranks as identifiers.



The diagram shows a program which runs with five processes. In this example, the size of MPI_COMM_WORLD is 5. The rank of each process is the number inside each circle. The rank of a process always ranges from 0 to 4.

MPI Functions

1. **MPI_Comm_rank(MPI_COMM_WORLD, &rank) :** The rank of a process within a communicator can be obtained using this

Parameters

- MPI_COMM_WORLD: This is a predefined communicator in MPI that includes all processes spawned by the MPI program. It is a communicator for the world of all processes.
- &rank: This is the address of the variable where the rank of the calling process will be stored. The function retrieves the rank and stores it in the memory location pointed to by the &rank variable.

2. **MPI_Comm_size(MPI_COMM_WORLD, &size)** is an MPI function call in that retrieves the total number of processes in the communicator **MPI_COMM_WORLD**.

Parameters

- **MPI_COMM_WORLD**: This is a predefined communicator in MPI that includes all processes spawned by the MPI program. It is a communicator for the world of all processes.

- &size: This is the address of the variable where the total number of processes in the communicator will be stored. The function retrieves the size and stores it in the memory location pointed to by the &size variable.

3. **MPI_Init(&argc, &argv)** is an MPI (Message Passing Interface) function call used to initialize the MPI environment. It is typically the first MPI function called in an MPI program.

Parameters

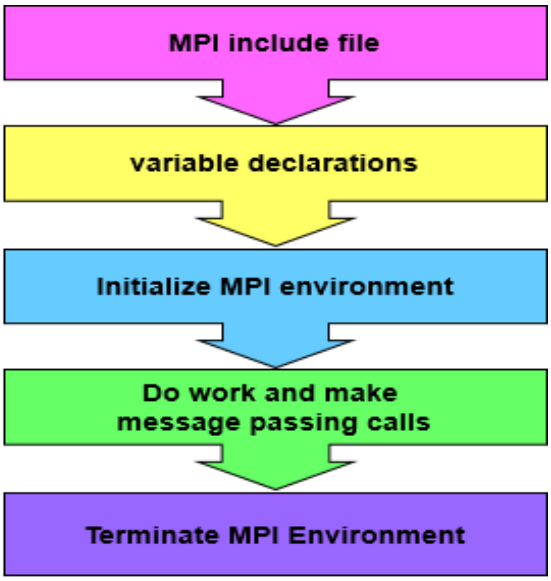
&argc: This passes a pointer to the argc variable to the MPI library. The argc variable holds the number of command-line arguments passed to the program.

&argv: This passes a pointer to the argv variable to the MPI library. The argv variable is an array of strings containing the command-line arguments.

When MPI initializes, it sets up communication channels between the processes, prepares the MPI environment for parallel computation

4. **MPI_Finalize()** is an MPI (Message Passing Interface) function used to finalize the MPI environment. It is typically the last MPI function called in an MPI program, and it performs several important tasks to ensure the proper termination of the MPI application. **MPI_Finalize()** ensures that all communication operations initiated by the program are completed before the program terminates.

MPI Program Structure

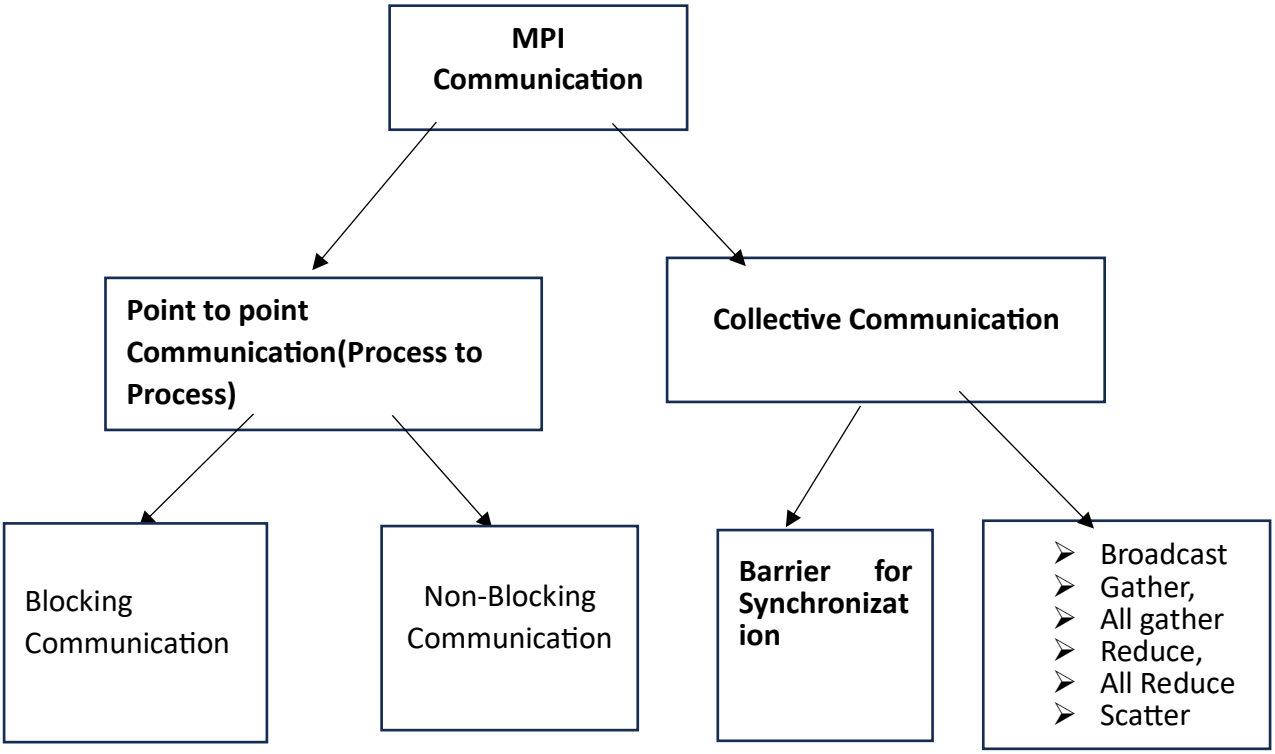


MPI Program

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char** argv)
 {
   MPI_Init(&argc, &argv);
   int rank, size;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Get the rank of the current process
   MPI_Comm_size(MPI_COMM_WORLD, &size); // Get the total number of processes in the
communicator
   printf("Hello from process %d of %d in MPI_COMM_WORLD\n", rank, size);
   MPI_Finalize();
   return 0;
}
```

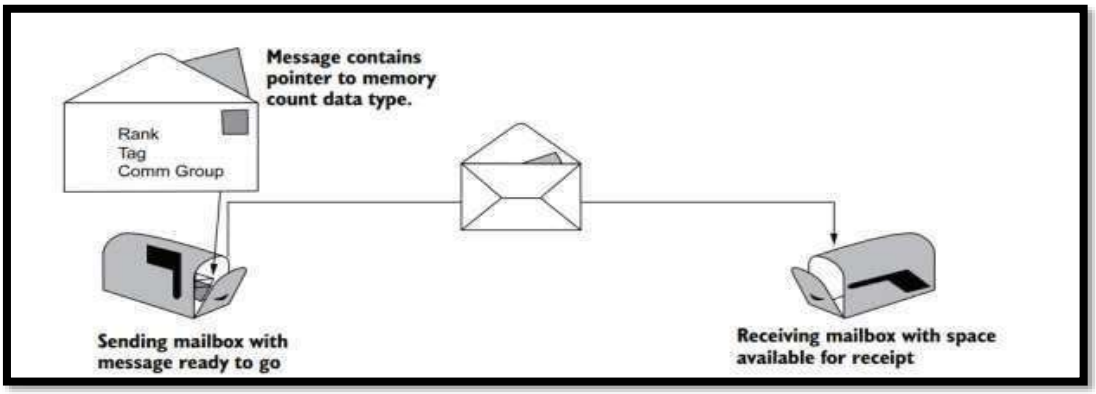
MPI Datatypes: MPI provides its own reference data types corresponding to the various elementarydata types in C.

MPI Datatype	C 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	(none)
MPI_PACKED	(none)



The send and receive commands for process-to-process communication

The core of the message-passing approach is to send a message from point-to-point or, perhaps more precisely, process-to-process. The whole point of parallel processing is to coordinate work.



The Figure shows the three components for process to process communication

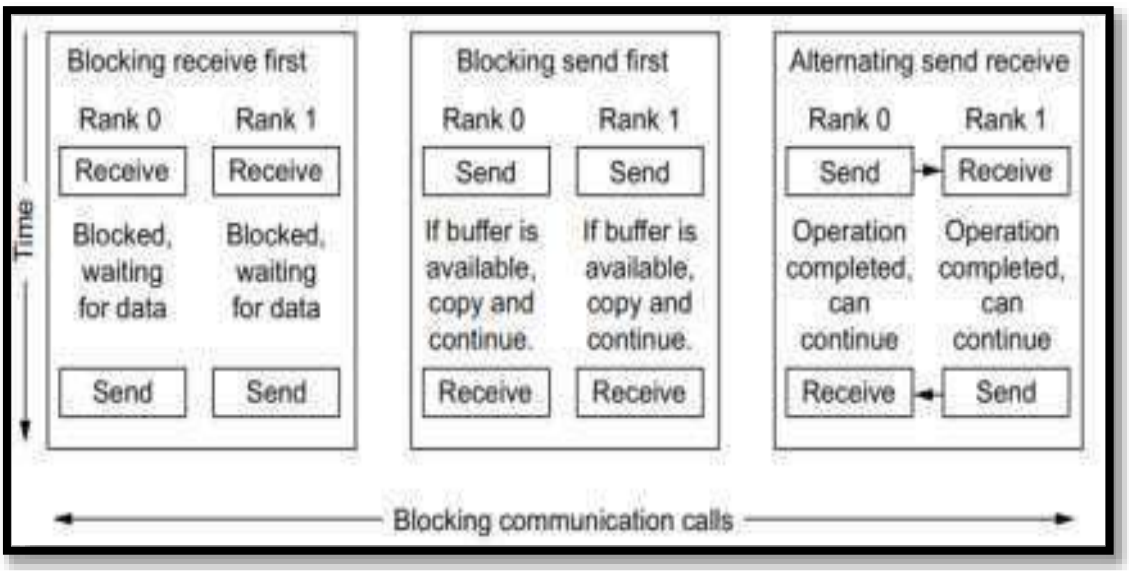
- **Mail Box:** There must be a mailbox at either end of the system. The size of the mailbox is important. The sending side knows the size of the message, but the receiving side does not. To make sure there is a place for the message to be stored, it is usually better to post the receive first. This avoids delaying the message by the receiving process having to allocate a temporary space to store the message until a receive is posted and it can copy it to the right

location. For an analogy, if the receive (mailbox) is not posted (not there), the postman has to hangout until someone puts one up. Posting the receive first avoids the possibility of insufficient memory space on the receiving end to allocate a temporary buffer to store the message.

- **Message :**The message itself is always composed of a triplet at both ends: a pointer to a memory buffer, a count, and a type. The type sent and type received can be different types and counts. The rationale for using types and counts is that it allows the conversion of types between the processes at the source and at the destination. This permits a message to be converted to a different form at the receiving end. In a heterogeneous environment, this might mean converting lower-endian to big-endian, a lowlevel difference in the byte order of data stored on different hardware vendors. Also, the receive size can be greater than the amount sent. This permits the receiver to query how much data is sent so it can properly handle the message. But the receiving size cannot be smaller than the sending size because it would cause a write past the end of the buffer.
- **Envelope:** The envelope also is composed of a triplet. It defines who the message is from, who it is sent to, and a message identifier to keep from getting multiple messages confused. The triplet consists of the rank, tag, and communication group. The rank is for the specified communication group. The tag helps the programmer and MPI distinguish which message goes to which receive. In MPI, the tag is a convenience. It can be set to MPI_ANY_TAG if an explicit tag number is not desired.

We have two types of process to process communication:Blocking and non blocking

Blocking communication in MPI refers to the type of communication where a process halts its execution until a specific communication operation is completed. This means that the sending and receiving processes are synchronized also refered as Synchronuous Communication. The sender blocks until the receiver is ready to receive the message, and vice versa. The two most common blocking communication operations in MPI are **MPI_Send** and **MPI_Recv**.



MPI_Send is a blocking communication function in MPI (Message Passing Interface) used for sending messages from one process to another. It sends a message from the sender process to the specified destination process. Here is the syntax for **MPI_Send**

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

- **buf**: A pointer to the send buffer (the data you want to send).
- **count**: The number of elements in the send buffer.
- **datatype**: The data type of the elements in the send buffer.
- **dest**: The rank of the destination process.
- **tag**: A message tag, which can be used by the receiver to distinguish different kinds of messages.
- **comm**: The communicator (usually **MPI_COMM_WORLD** for communication among all processes).

MPI_Recv is a blocking communication function in MPI (Message Passing Interface) used for receiving messages from other processes. It receives a message from a specified source process. Here is the syntax for **MPI_Recv**

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

- **buf**: A pointer to the receive buffer (where the received data will be stored).
- **count**: The number of elements in the receive buffer.
- **datatype**: The data type of the elements in the receive buffer.

- **source**: The rank of the source process from which you want to receive the message. Use **MPI_ANY_SOURCE** if you want to receive a message from any source.
- **tag**: A message tag. If you used tags in **MPI_Send**, you can use the same tag here to filter messages. MPI_ANY_TAG matches any tag comm
- **comm**: The communicator (usually **MPI_COMM_WORLD** for communication among all processes).
- **status**: A pointer to an **MPI_Status** structure that will hold information about the received message, such as the source, tag, and error codes.

Program

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
   int rank, size;
   int data_send = 42;
   int data_recv;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   if (size < 2) {
      printf("This program requires at least 2 processes.\n");
      MPI_Finalize();
      return 1;
   }
   // Blocking Send from process 0 to process 1
   if (rank == 0) {
      MPI_Send(&data_send, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
      printf("Process %d sent data: %d\n", rank, data_send);
   }
   // Blocking Receive at process 1
   else if (rank == 1) {
      MPI_Recv(&data_recv, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
        printf("Process %d received data: %d\n", rank, data_recv);
    }
    MPI_Finalize();
    return 0;
}
```

Problems with blocking communication:

- A deadlock occurs when a set of processes are blocked because each is waiting for the other to release a resource. For example, if two processes are waiting for each other to send a message before they can receive, they will be deadlocked.
- Processes that are blocked waiting for communication can waste computational resources, such as CPU time and memory, as they are not performing useful work during that time.so we go for non blocking communication

Non blocking communication

Non-blocking communication in MPI allows processes to initiate communication operations and continue their execution without waiting for the communication to complete. This is often referred to as asynchronous or non-blocking calls. Asynchronous means that the call initiates the operation but does not wait for the completion of the work.

MPI provides non-blocking communication functions like **MPI_Isend(I means Immediate)**, **MPI_Irecv**, **MPI_Test**, **MPI_Wait**, and others to facilitate non-blocking communication. **Completion of a non-blocking send operation means that the sender is now free to update the send buffer "message".**

Completion of a non-blocking receive operation means that the receive buffer "message" contains the received data.

- **MPI_Isend** is a non-blocking communication function used for sending messages from one process to another. Unlike **MPI_Send**, which is a blocking operation, **MPI_Isend** returns immediately after initiating the send operation, allowing the sender process to continue its execution without waiting for the message to be delivered.

MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request);

- **buf**: A pointer to the send buffer (the data you want to send).
- **count**: The number of elements in the send buffer.

- **datatype**: The data type of the elements in the send buffer.
- **dest**: The rank of the destination process.
- **tag**: A message tag, which can be used by the receiver to distinguish different kinds of messages.
- **comm**: The communicator (usually **MPI_COMM_WORLD** for communication among all processes).
- **request**: A pointer to an **MPI_Request** variable that will be used to identify the send request. You can later use this request to check the status of the communication or wait for its completion.

➢ **MPI_Irecv** is a non-blocking communication function used for receiving messages from other processes. Unlike **MPI_Recv**, which is a blocking operation, **MPI_Irecv** returns immediately after initiating the receive operation, allowing the receiving process to continue its execution without waiting for a message to arrive.

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

- **buf**: A pointer to the receive buffer (where the received data will be stored).
- **count**: The number of elements in the receive buffer.
- **datatype**: The data type of the elements in the receive buffer.
- **source**: The rank of the source process from which you want to receive the message. Use **MPI_ANY_SOURCE** if you want to receive a message from any source.
- **tag**: A message tag. If you used tags in **MPI_Send**, you can use the same tag here to filter messages.
- **comm**: The communicator (usually **MPI_COMM_WORLD** for communication among all processes).
- **request**: A pointer to an **MPI_Request** variable that will be used to identify the receive request. You can later use this request to check the status of the communication or wait for its completion.

➢ **MPI_Test** is a non-blocking communication function used to check the completion status of a communication request initiated by non-blocking send (**MPI_Isend**) or receive (**MPI_Irecv**) operations. It allows you to query whether a non-blocking operation has been completed without waiting for its completion. Here is the syntax for **MPI_Test**:

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

- **request**: A pointer to an **MPI_Request** variable that identifies the communication request.
- **flag**: A pointer to an integer variable that will be set to true (non-zero) if the communication operation associated with the request has completed, and false (zero) otherwise.
- **status**: A pointer to an **MPI_Status** structure that will hold information about the completed communication, such as the source, tag, and error codes.

MPI_Test returns **MPI_SUCCESS** if the operation associated with the request has completed and **flag** is set to true. Otherwise, it returns **MPI_ERR_PENDING** if the operation is still pending, meaning it has not yet completed.

- **MPI_Wait** is a blocking function used to wait for the completion of a specific communication request, such as non-blocking send (**MPI_Isend**) or receive (**MPI_Irecv**) operations. It suspends the execution of the calling process until the specified communication operation is completed. Here is the syntax for **MPI_Wait**:

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

- **request**: A pointer to an **MPI_Request** variable that identifies the communication request.
- **status**: A pointer to an **MPI_Status** structure that will hold information about the completed communication, such as the source, tag, and error codes. You can pass **MPI_STATUS_IGNORE** if you don't need this information.

MPI_Wait blocks until the communication associated with the specified request is complete. Once the operation has completed, you can use the information in the **status** object if needed.

Program

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
   int rank, size;
   int data_send = 42;
```

```
    int data_recv;
    MPI_Request request;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (size < 2) {
        printf("This program requires at least 2 processes.\n");
        MPI_Finalize();
        return 1;
    }
    // Non-blocking Send from process 0 to process 1
    if (rank == 0) {
        MPI_Isend(&data_send, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
        printf("Process %d initiated non-blocking send with data: %d\n", rank, data_send);
    }
    // Non-blocking Receive at process 1
    else if (rank == 1) {
        MPI_Irecv(&data_recv, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
        printf("Process %d initiated non-blocking receive.\n", rank);
    }
    // Wait for the non-blocking communication to complete
    MPI_Wait(&request, MPI_STATUS_IGNORE);
    if (rank == 1) {
        printf("Process %d received data: %d\n", rank, data_recv);
    }
    MPI_Finalize();
    return 0;
}
```

Advantages of Non Blocking Communication:

1. **Overlapping of Computation and Communication:** Non-blocking operations allow computation and communication to occur concurrently. Processes can initiate communication

operations and then continue with other computations without waiting for the communication to complete. This overlap of computation and communication can lead to improved overall performance and better utilization of resources.

2. **Reduced Synchronization Overheads:** Non-blocking operations reduce the need for synchronization points in the code. With blocking communication, processes often need to synchronize at communication points, leading to potential idle time for some processes. Non-blocking communication reduces these synchronization overheads and can lead to more balanced workloads among processes.

3. **Minimized Potential for Deadlocks:** Non-blocking communication reduces the likelihood of encountering deadlocks, which can occur in scenarios where processes are waiting for each other to release resources. Non-blocking operations allow processes to progress independently, reducing the chances of deadlocks.

4. **Better Load Balancing:** Non-blocking operations enable dynamic load balancing techniques. Processes can continue with computation tasks even when waiting for communication, allowing load balancing algorithms to adjust the workload dynamically based on the actual computational needs of the processes.

5. **Optimized Network Utilization:** Overlapping computation and communication can lead to more efficient use of network resources. Processes can perform useful work while waiting for messages, reducing idle time and maximizing the utilization of the communication network.

Other variants of send/receive might be useful in special situations.

The modes are indicated by a one- or two-letter prefix, similar to that seen in the immediate variant, as listed here:

B (buffered)

S (synchronous)

R (ready)

IB (immediate buffered)

IS (immediate synchronous)

IR (immediate ready)

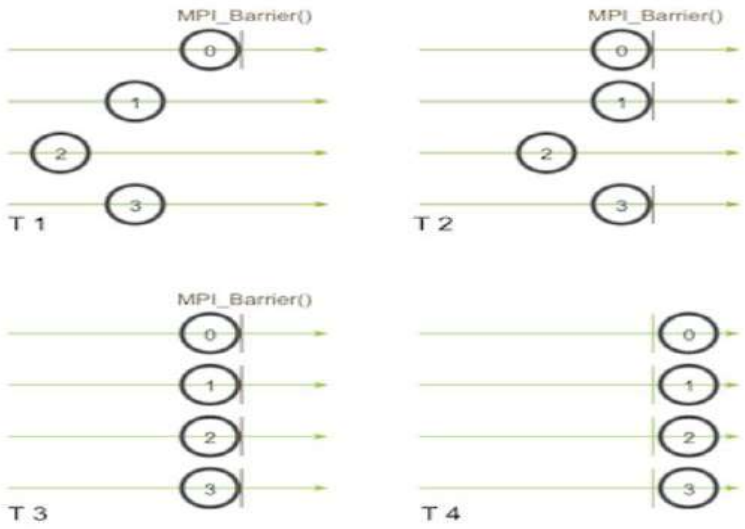
Collective communication

Collective communication refers to a type of communication pattern in parallel and distributed computing, where multiple processes or nodes collaborate to exchange information among themselves. These communication patterns are essential in high-performance computing and distributed systems to efficiently solve problems that require coordinated efforts among multiple participants.

Types of collective communication

Collective communication operations are made of the following types:

1. **Barrier:** A barrier is a synchronization point that forces all processes in a group to wait until they have all reached the barrier before continuing. Barriers are often used to ensure that all processes are at the same point in their execution.



Syntax:

int MPI_Barrier(MPI_Comm communicator);

communicator: The communicator that defines the group of processes that synchronize at the barrier.

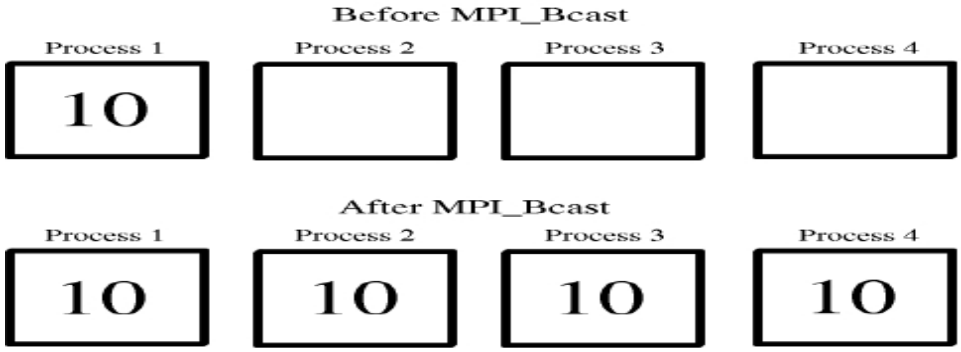
The **MPI_Barrier** function is often used to coordinate the execution of processes in a parallel program. For example, if different processes are performing different parts of a computation and need to ensure that they all reach a certain point before proceeding.

Program

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
   int rank, size;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   // Some computation before the barrier
   printf("Process %d reached the barrier.\n", rank);
   MPI_Barrier(MPI_COMM_WORLD); // All processes wait here until everyone reaches this point
   // Code after the barrier
   MPI_Finalize();
   return 0;
}
```

2. **Data Movement (**or Global Communication):

    - **Broadcast:** In a broadcast operation, one process sends the same data to all other processes in a group. It is often used to distribute information from one process to all others.



Syntax

MPI_Bcast(void* data, int count, MPI_Datatype datatype, int root, MPI_Comm communicator)

- **data:** A pointer to the data that the root process wants to broadcast. This data is sent by the root process and received by all other processes.
- **count**: The number of data elements in the buffer.
- **datatype**: The datatype of the elements in the buffer.
- **root**: The rank of the process within the communicator that is broadcasting the data.
- **communicator**: The communicator that defines the group of processes over which the broadcast operation is performed.

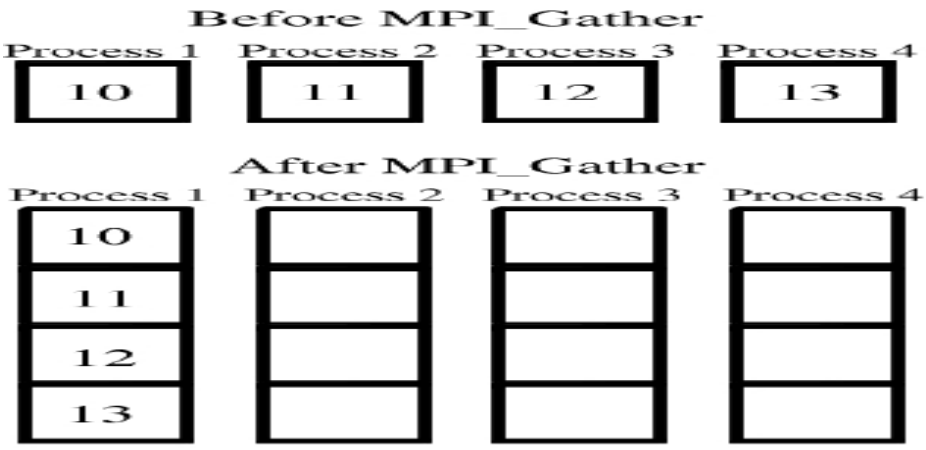
Program

```
#include<stdio.h>
#include<mpi.h>
int main(int argc, char* argv[])
{
int a = 10, r, s;
MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &s);
MPI_Comm_rank(MPI_COMM_WORLD, &r);
printf("\n data in process %d before bcast=%d", r, a);
MPI_Bcast(&a, 1, MPI_INT, 0, MPI_COMM_WORLD);
printf("\n data in process %d after bcast=%d",r,a);
MPI_Finalize(); return 0;
}
```

Output

```
C:\Users\akshith's\source\repos\Project3\x64\Debug>mpiexec -n 4 ./project3.exe

 data in process 0 before bcast=10
 data in process 0 after bcast=10

 data in process 1 before bcast=0
 data in process 1 after bcast=10

 data in process 3 before bcast=0
 data in process 3 after bcast=10

 data in process 2 before bcast=0
 data in process 2 after bcast=10
```

- **Gather:** The gather operation collects data from all processes in a group and sends it to a designated process. This is useful when you want to aggregate data from multiple sources.



Syntax

MPI_Gather(void* sendbuf, int send_count, MPI_Datatype sendtype, void * recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm communicator)

sendbuf: A pointer to the send buffer (data to be sent) on each process.

sendcount: The number of elements to send from the send buffer.

sendtype: The datatype of the elements in the send buffer.

recvbuf: A pointer to the receive buffer on the root process. This is where the gathered data will be stored.

recvcount: The number of elements to receive from each process.

recvtype: The datatype of the elements in the receive buffer.

root: The rank of the root process, which will receive the gathered data.

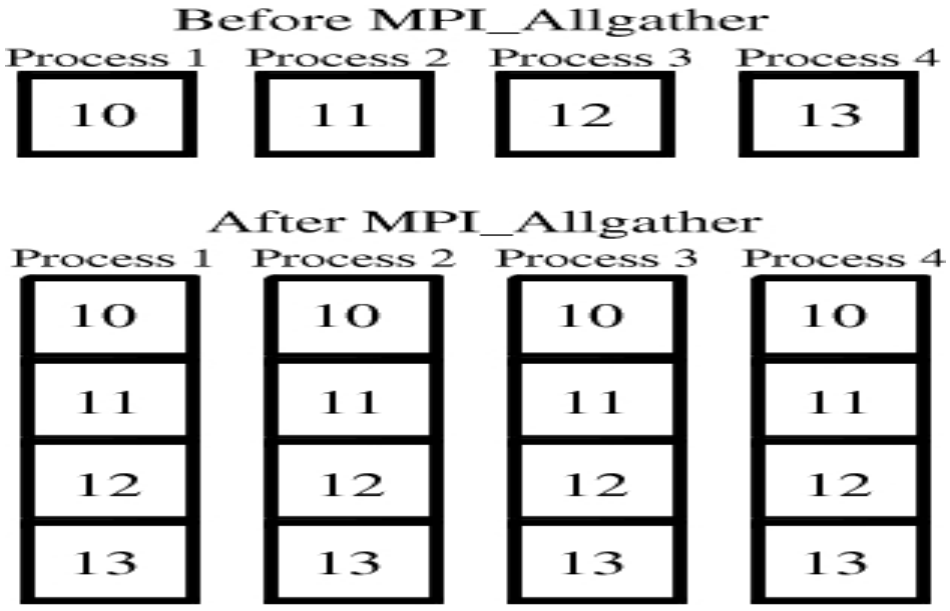
communicator: The communicator that defines the group of processes.

Program

```
#include<stdio.h>
#include<mpi.h>
int main(int argc, char* argv[])
{
int d = 0, r, s, a[5];
MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD,&s);
MPI_Comm_rank(MPI_COMM_WORLD, &r);
d = r * 2;
MPI_Gather(&d, 1, MPI_INT, &a, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (r == 0)
```

```
{
printf("data received by process o=");
for (int i = 0;i < s;i++)
printf("%d\t", a[i]);
}
MPI_Finalize();return 0;
}
```

- **Allgather : MPI_Allgather** distributes the gathered data to all processes in the communicator, not just to the root process. Each process receives the entire gathered dataset.



Syntax

MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype senddatatype, void*recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm communicator)

sendbuf: A pointer to the send buffer (data to be sent) on each process.

sendcount: The number of elements to send from the send buffer on each process.

sendtype: The datatype of the elements in the send buffer.

recvbuf: A pointer to the receive buffer on each process. This is where the gathered data will be stored.

recvcount: The number of elements to receive from each process.

recvtype: The datatype of the elements in the receive buffer.

communicator: The communicator that defines the group of processes

Here's how MPI_Allgather works:

Each process provides data in its send buffer (sendbuf).

The data from the send buffers of all processes is gathered.

The gathered data is distributed to the receive buffers of all processes (recvbuf).

Program

```
#include<stdio.h>
#include<mpi.h>
int main(int argc, char* argv[])
{
int d = 0, r, s, a[5];
MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD,    &s);
MPI_Comm_rank(MPI_COMM_WORLD, &r);
d = r * 2;
MPI_Allgather (&d, 1, MPI_INT, &a, 1, MPI_INT, MPI_COMM_WORLD);
printf("data received by process %d=",r);
for (int i = 0;i < s;i++)
printf("%d\t", a[i]);
MPI_Finalize(); return 0;
}
```

```
C:\Users\akshith's\source\repos\Project3\x64\Debug>mpiexec -n 5 ./project3.exe
data received by process 2=0    2       4       6       8
data received by process 1=0    2       4       6       8
data received by process 3=0    2       4       6       8
data received by process 4=0    2       4       6       8
data received by process 0=0    2       4       6       8
```

- **Reduce:** In a reduce operation, data from all processes is combined using an associative and commutative operation (e.g., addition, multiplication) to produce a single result. This is often used for aggregating data or finding global statistics. There are many operations that can be done during the reduction. The most common are

MPI_MAX (maximum value in an array)

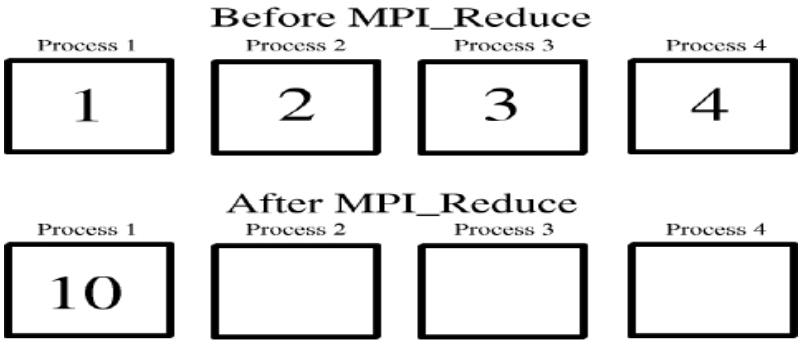
MPI_MIN (minimum value in an array)

MPI_SUM (sum of an array)

MPI_MINLOC (index of minimum value)

MPI_MAXLOC (index of maximum value

Example:



Syntax

• MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm communicator)

- **sendbuf**: A pointer to the send buffer (data to be reduced) on each process.
- **recvbuf**: A pointer to the receive buffer on the root process. This is where the reduced result will be stored.
- **count**: The number of elements in the send buffer.
- **datatype**: The datatype of the elements in the send buffer.
- **op**: The reduction operation to be performed (e.g., **MPI_SUM**, **MPI_MAX**, **MPI_MIN**, **MPI_PROD**,etc.).
- **root**: The rank of the root process, where the reduced result will be stored.
- **communicator**: The communicator that defines the group of processes

Program

```
int main(int argc, char* argv[])
{
int size, rank;
MPI_Init(NULL,NULL);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
int localsum ;
int globalsum;
```

```
localsum = 10 + rank;
printf("process %d value=%d", rank, localsum);
MPI_Reduce(&localsum, &globalsum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
if (rank == 0)
{
printf("\n globalsum = %d", globalsum);
}
MPI_Finalize();
return (0);
}
```

Output for 3 processes: global sum=13

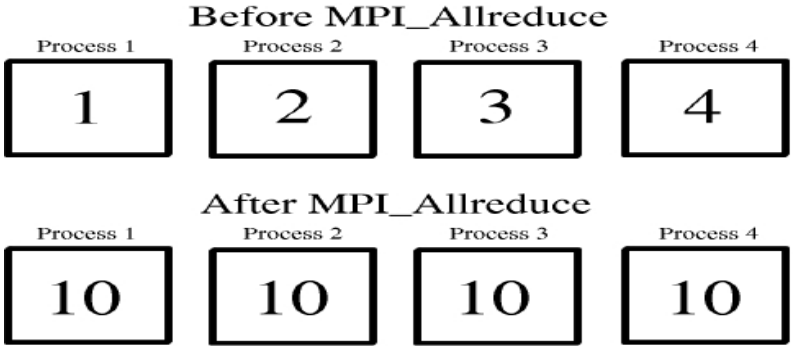
First process local sum=10+0(rank)=10

Second process local sum=10+1(rank)=11

Third process local sum=11+2(rank)=13

- **AllReduce: MPI_Allreduce** is a collective communication function in the MPI (Message Passing Interface) standard. It is similar to **MPI_Reduce** in that it performs a reduction operation across all processes in a communicator. However, unlike **MPI_Reduce**, the result of the reduction operation is distributed to all processes, not just the root process. Every process receives the reduced result.

Example



Syntax

- MPI_Allreduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm communicator);

sendbuf: A pointer to the send buffer (data to be reduced) on each process.

recvbuf: A pointer to the receive buffer on each process. This is where the reduced result will be stored.

count: The number of elements in the send buffer.

datatype: The datatype of the elements in the send buffer.

op: The reduction operation to be performed (e.g., **MPI_SUM**, **MPI_MAX**, **MPI_MIN**, **MPI_PROD**, **MPI_LAND**, **MPI_BAND**, **MPI_LOR**, etc.).

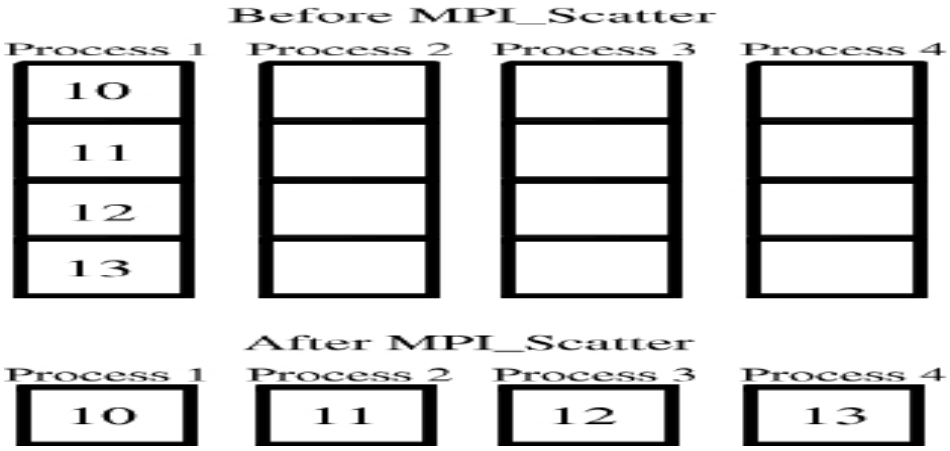
communicator: The communicator that defines the group of processes.

Program

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main(int argc, char* argv[])
{
int size, rank;
MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
int globalsum;
printf("process %d value=%d", rank, localsum);
MPI_Allreduce(&rank, &globalsum, 1, MPI_INT, MPI_SUM,
MPI_COMM_WORLD);
printf("\n globalsum = %d", globalsum);
MPI_Finalize();
return (0);
}
```

Output for 3 processes: global sum=13 ,3 times for 3 process

- **Scatter:** Scatter is the opposite of gather. It takes data from one process and distributes it to all other processes in a group. Each process receives a different portion of the data.



Syntax

MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype, void*recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm communicator);

sendbuf: A pointer to the send buffer (data to be scattered) on the root process.

sendcount: The number of elements to send from the send buffer on the root process.

sendtype: The datatype of the elements in the send buffer.

recvbuf: A pointer to the receive buffer on each process. This is where the scattered data will be stored.

recvcount: The number of elements to receive on each process.

recvtype: The datatype of the elements in the receive buffer.

root: The rank of the root process, which is the source of the scattered data.

communicator: The communicator that defines the group of processes.

Program

```
#include<stdio.h>
#include<mpi.h>
int main(int argc, char* argv[])
{
int d = 0, r, s, *
buf=NULL;
MPI_Init(NULL,
NULL);
MPI_Comm_size(MPI_COMM_WORLD,      &s);
MPI_Comm_rank(MPI_COMM_WORLD,      &r);
if(r == 0)
{
int a[5] = { 1,2,3,4,5 };
```

```
buf = a;
}
printf("\n data in process %d before scatter=%d", r, d);
MPI_Scatter(buf, 1, MPI_INT, &d, 1, MPI_INT, 0, MPI_COMM_WORLD);
printf("\n data in process %d after
scatter=%d", r, d);MPI_Finalize();
return 0;
}
```

Output

```
:\Users\akshith's\source\repos\Project3\x64\Debug>mpiexec -n 5 ./project3.exe

data in process 4 before scatter=0
data in process 4 after scatter=5

data in process 2 before scatter=0
data in process 2 after scatter=3

data in process 1 before scatter=0
data in process 1 after scatter=2

data in process 3 before scatter=0
data in process 3 after scatter=4

data in process 0 before scatter=0
data in process 0 after scatter=1
```

Data Parallel Examples

The data parallel strategy is the most common approach in parallel applications.

First, a simple caseof the stream triad where no communication is necessary.

The Stream Triad benchmark measures the memory bandwidth of a computing system. It is a simple yet effective benchmark to assess the memory performance of a node. The Stream Triad benchmark calculates the memory bandwidth by performing a simple operation on arrays in memory.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define ARRAY_SIZE (1 << 20) // 1 million elements (adjust based on your system)
#define SCALAR 2.0
int main() {
```

```
double *a, *b, *c;
int i;
double start_time, end_time;
double bandwidth;

// Allocate memory for arrays
a = (double*)malloc(ARRAY_SIZE * sizeof(double));
b = (double*)malloc(ARRAY_SIZE * sizeof(double));
c = (double*)malloc(ARRAY_SIZE * sizeof(double));
// Initialize arrays
#pragma omp parallel for
for (i = 0; i < ARRAY_SIZE; i++) {
   a[i] = 1.0;
   b[i] = 2.0;
   c[i] = 0.0;
}
// Measure start time
start_time = omp_get_wtime();
// Perform Stream Triad operation
#pragma omp parallel for
for (i = 0; i < ARRAY_SIZE; i++) {
   c[i] = a[i] + b[i] * SCALAR;
}
// Measure end time
end_time = omp_get_wtime();
// Calculate bandwidth in GB/s
bandwidth = (3 * ARRAY_SIZE * sizeof(double)) / ((end_time - start_time) * 1e9);
// Print bandwidth
printf("Memory Bandwidth: %f GB/s\n", bandwidth);
// Free allocated memory
free(a);
free(b);
```

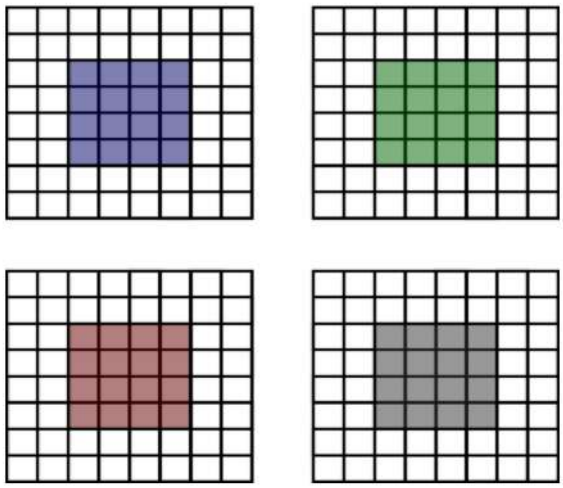
```
    free(c);
    return 0;
}
```

Second the more typical ghost cell exchange techniques used to link together the subdivided domains distributed to each process

Ghost cell exchange/updates are a common technique in computational science and high-performance computing, particularly in the context of numerical simulations, to manage boundary conditions and ensure accurate results when performing computations on a grid or mesh. Ghost cells, also known as halo cells or boundary cells, are additional grid cells that surround the main computational domain.

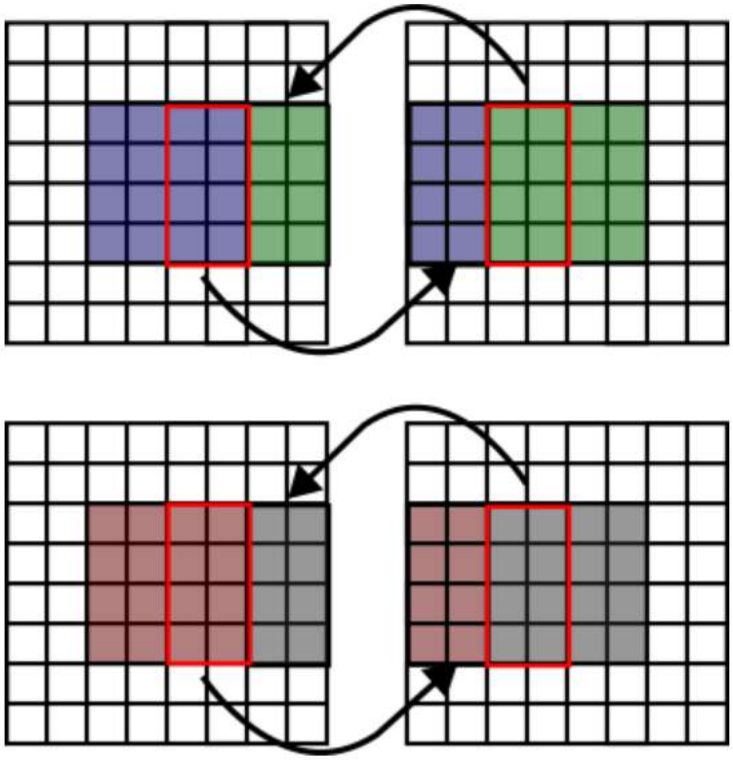
The update of ghost cells involves exchanging data between neighbouring chunks to ensure that each chunk has the correct information about its boundary cells. This communication is necessary because the computation in one chunk often depends on the values of neighbouring cells.

To exchange halo, first we have four block like this:



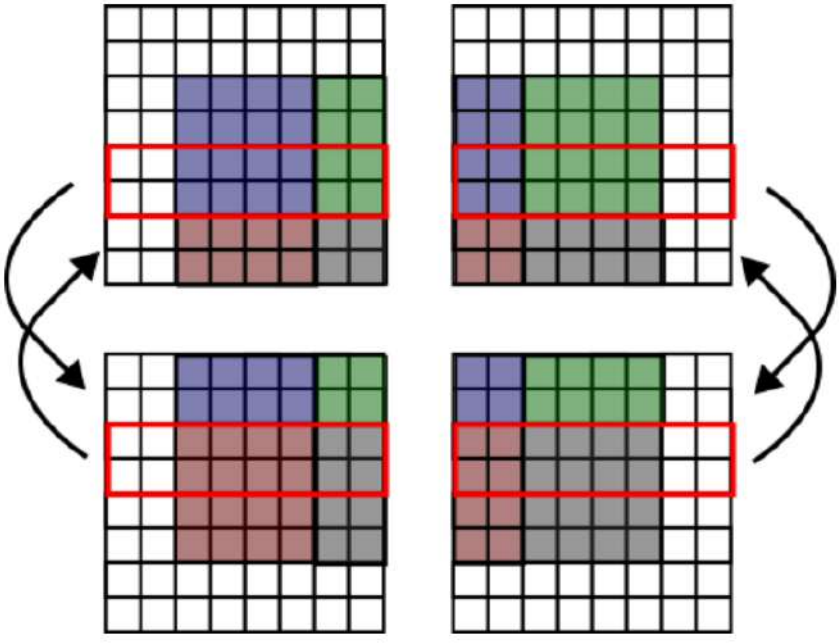
The white cells represeent the halo cells which can be used for temporary storage of values while exchanging the data.Each mesh is assigned to different processors for performing operations.

The first communication happens along x-axis



We can see how the cells are exchanged from one mesh to another with the help of halo cells.(Colors are swapped).

The second communcation happens along y-axis. Note that here the communicated boundary extended to ghosts, this is necessary to have corners transfered correctly:



Advanced MPI functionality to simplify code and enable optimizations

The advanced functions that are useful in common data parallel applications.

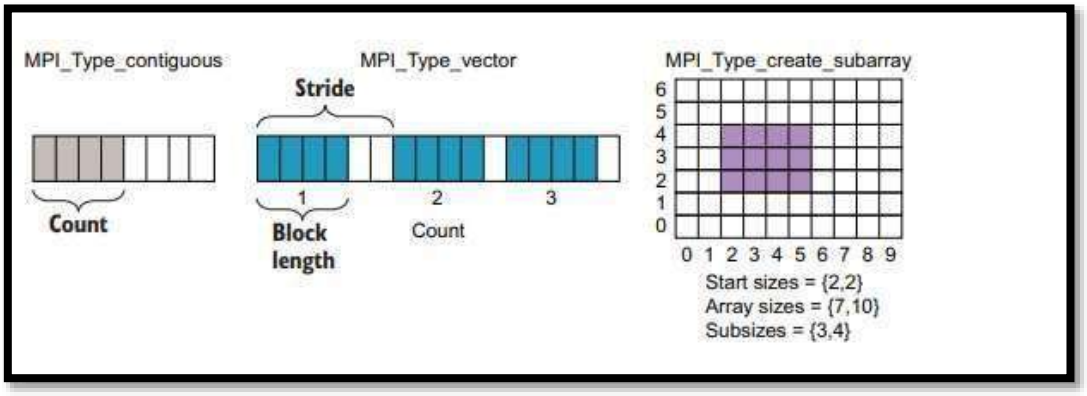
- **MPI custom data types**
- **Topology support**

- **Custom MPI Data Types:**

MPI has a rich set of functions to create new, custom MPI data types from the basic MPI types

- MPI_Type_contiguous—makes a block of contiguous data into a type.
- MPI_Type_vector—creates a type out of blocks of strided data. (elements in strided data are not necessarily contiguous in memory; there are gaps between them).
- MPI_Type_create_subarray—creates a rectangular subset of a larger array.
- MPI_Type_create_struct—Creates a data type encapsulating the data items in a structure in a portable way that accounts for padding by the compiler

Three MPI custom data types with illustrations of the arguments used in their creation



A type must be committed before use and it must be freed to avoid a memory leak. The routines include

- MPI_Type_Commit—Initializes the new custom type with needed memory allocation or other setup
- MPI_Type_Free—Frees any memory or data structure entries from the creation of the data type

- **Topology support**

Cartesian topology support in MPI (Message Passing Interface) allows you to define a logical, multi-dimensional grid or mesh of processes to facilitate communication and coordination in parallel applications. This is particularly useful for simulations, numerical computations, and other scientific computing tasks where data is organized in multi-dimensional arrays or grids.

To create a Cartesian topology in MPI, you typically follow these steps:

1. Initialize MPI and determine the size and rank of your MPI communicator.

```
#include <mpi.h>

int main(int argc, char** argv) {

   MPI_Init(&argc, &argv);

   int size, rank;

   MPI_Comm_size(MPI_COMM_WORLD, &size);

   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   MPI_Finalize();

   return 0;

}
```

2. Define the dimensions and periodicity of the grid using an integer array, and create the Cartesian communicator using **MPI_Cart_create**

```
int dims[ndims];  // Array specifying the number of processes in each dimension

int periods[ndims];  // Array indicating whether the grid is periodic in each dimension

MPI_Comm cart_comm;

MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periods, 0, &cart_comm);
```

3. Retrieve the coordinates of each process in the Cartesian grid using **MPI_Cart_coords**.

```
int coords[ndims];

MPI_Cart_coords(cart_comm, rank, ndims, coords);
```

4. Perform point-to-point communication or collective operations specific to your application's grid structure, often using functions like **MPI_Send**, **MPI_Recv**, and collective operations like **MPI_Allreduce**, **MPI_Gather**, or **MPI_Scatter**.
5. After you're done, free the Cartesian communicator using **MPI_Comm_free**

```
MPI_Comm_free(&cart_comm);
```