MPI

What is MPI?
- MPI (Message Passing Interface) is a way for computer programs to talk to each other when they're running on different parts of a big computer or on multiple computers.
- People use MPI to make their computer programs work together and share information.

Key Ideas about MPI:
1. MPI helps make computer programs work together.
2. It's like a language they use to talk to each other.
3. MPI makes it possible for programs to work together on big tasks.
4. It works on different kinds of computers and can even work when computers are different from each other.
5. MPI helps programs work better when they need to do a lot of things at once.

Introduction to MPI:
- MPI is used by separate programs that work on their own.
- It's a way for these programs to talk to each other and help each other.
- Think of it like different people speaking the same language to get things done.
- MPI can work on many different types of computers, making it easy for programs to work together even on different machines.
- MPI helps programs run faster when they have a lot of work to do.

MIMD vs. SPMD:
- MPI programs are organized in two main ways: MIMD (Multiple Instruction, Multiple Data) and SPMD (Single Program, Multiple Data).
- In MIMD, each processor can run different instructions on different data.
- In SPMD, all processors run the same program, but they might work on different pieces of data.

MPI Program Organization:
- MPI programs often use a combination of MIMD and SPMD in a single framework.
- In this framework, processors can have their own local memory that only they can access.
- The number of processes (processors) is fixed when the program starts; you can't add or remove them while the program is running.
- Each of these processes can potentially run a different program, which is sometimes called MPMD (Multiple Program, Multiple Data).

Example:
- Imagine you have a supercomputer with many processors, and you want them to work together to analyze weather data.
- In this case, you might use SPMD, where all processors run the same weather analysis program, but each processor works on a different part of the data (e.g., a different region or time period).
- Alternatively, if you have different tasks, like one processor simulating rainfall and another predicting temperature, you'd be using MIMD within the SPMD framework.
- This combination allows you to efficiently use all the processors and tackle complex tasks.

So, MPI lets you control how processors work together, whether they all do the same thing (SPMD) or different things (MIMD) within a fixed number of processes.

Message Passing Work Allocation:
- In MPI, programs involve two types of processes: Manager Process and Worker Process.
- Message passing programs exchange local data using communication operations.

- MPI consists of a collection of processes.
- Processes in MPI are single-threaded and have separate memory spaces.
- You can start an MPI program from the command line.

General MPI Program Structure:
1. Initialize MPI environment:
  - Include MPI header: `#include <mpi.h>`
  - Initialize MPI: `MPI_Init(&argc, &argv)`
  - Get process rank and size: `MPI_Comm_rank(MPI_COMM_WORLD, &rank)` and `MPI_Comm_size(MPI_COMM_WORLD, &np)`

2. Do work and make message passing calls.

3. Terminate MPI environment:
  - Finalize MPI: `MPI_Finalize()`

MPI Naming Conventions:
- All MPI entities (functions, constants, types) start with `MPI_`.
- C function names use mixed case, like `MPI_Xxxxx`.
- MPI constants are in upper case, such as `MPI_COMM_WORLD`.
- Specially defined types in C follow the same naming convention, e.g., `MPI_Comm` for MPI communicator.

MPI Routines and Return Values:
- MPI routines are implemented as functions in C.
- They return an error code (integer) indicating success or failure.
- `MPI_SUCCESS` is returned if the routine ran successfully.
- An implementation-dependent value is returned if an error occurred.

Example (in simplified terms):
```c
#include <mpi.h>
void main (int argc, char *argv[]) {
  int np, rank, ierr;
  ierr = MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  // Do Some Work

  ierr = MPI_Finalize();
}
```

Special MPI Datatypes:
- MPI in C provides special datatypes like `MPI_Comm` (communicator), `MPI_Status` (status information), and `MPI_Datatype`.
- These datatypes are used for variable declarations and help organize data.

Include Files:
- To use MPI in C, include the header file `mpi.h`.

Initializing MPI:
- In any MPI program, start with `MPI_Init`, which sets up the MPI environment.
- `MPI_Init` takes two arguments: the addresses of `argc` and `argv`.

Communicators:
- Communicators represent groups of processors that can communicate.
- Multiple communicators can exist, and a processor can belong to several of them.

Getting Communicator Information (Rank and Size):
- Use `MPI_Comm_rank` to find a processor's rank in a communicator.
- Use `MPI_Comm_size` to determine the number of processors in a communicator.

Terminating MPI:
- End an MPI program with `MPI_Finalize`.
- It cleans up MPI-related resources and should be called by all processes. If one process doesn't reach it, the program may hang.

Hello World MPI Program:
- This program is a simple example of using MPI to run parallel code on multiple processors.
- Each processor prints its rank and the total number of processors in the communicator `MPI_COMM_WORLD`.

Sample Program Code:
```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[]) {
   int myrank, size;

   /* Initialize MPI */
   MPI_Init(&argc, &argv);

   /* Get my rank */
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

   /* Get the total number of processors */
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   printf("Processor %d of %d: Hello World!\n", myrank, size);
```

```
    /* Terminate MPI */
    MPI_Finalize();

    return 0;
}
```

Sample Program Output:
- When this code is run on four processors, it will produce output like this:
  ```
  Processor 0 of 4: Hello World!
  Processor 1 of 4: Hello World!
  Processor 2 of 4: Hello World!
  Processor 3 of 4: Hello World!
  ```

Summary:
- MPI is used for creating parallel programs based on message passing.
- Typically, the same program is run on multiple processors.
- The program involves six basic MPI calls, including initialization and finalization, obtaining ranks and sizes, and sending/receiving messages.

Point-to-Point Communication:
- MPI allows processes in a communicator (like `MPI_COMM_WORLD`) to exchange messages.
- Point-to-point communication is the basic form of data exchange between processes.

Sending Messages:
- To send a message, a process uses `MPI_Send` with the following parameters:
  - `smessage`: A pointer to the message data.
  - `count`: The number of data items being sent.
  - `datatype`: The type of data being sent (e.g., `MPI_INT`, `MPI_DOUBLE`).
  - `dest`: The rank of the destination process.
  - `tag`: A unique tag to identify the message.
  - `comm`: The communicator to use (usually `MPI_COMM_WORLD`).

Receiving Messages:
- To receive a message, a process uses `MPI_Recv` with these parameters:
  - `rmessage`: A pointer to where the received data will be stored.
  - `count`: The number of data items expected.
  - `datatype`: The expected data type (must match the sender).
  - `source`: The rank of the source process (or `MPI_ANY_SOURCE` to accept from any source).
  - `tag`: The tag to match with the incoming message.
  - `comm`: The communicator (usually `MPI_COMM_WORLD`).
  - `status`: A pointer to an `MPI_Status` structure to get status information.

Predefined Data Types for MPI:
- MPI provides predefined data types that match C data types. For example:
  - `MPI_INT` corresponds to `int`.

- `MPI_DOUBLE` corresponds to `double`.
- There are types for various data sizes and types.

These functions and data types are used for processes to send and receive data in parallel MPI programs.

Blocking Operation:
- A blocking operation is one where a process waits for the operation to complete before continuing with other tasks.
- In MPI, common blocking operations include `MPI_Send` and `MPI_Recv`.
- When you use blocking operations, the process pauses until the data is sent or received, synchronizing sender and receiver.

Non-blocking Operation:
- A non-blocking operation allows a process to continue with other tasks without waiting for the operation to complete.
- In MPI, common non-blocking operations include `MPI_ISend` and `MPI_IRecv`.
- Non-blocking operations are asynchronous, meaning the sender and receiver can continue with other work while data is being sent or received.

Synchronous and Asynchronous Communications:
- These terms describe whether an operation is synchronized with other processes or not.
- Synchronous communication, like `MPI_Ssend`, means the sender and receiver are synchronized. The sender waits for the receiver to acknowledge the message.
- Asynchronous communication, like `MPI_ISend`, means the sender and receiver can work independently without synchronization.

Examples of Blocking (P2P - Point-to-Point) Send/Recv:
- Blocking Send (`MPI_Send`) waits until the message is received by the destination process.
- Blocking Receive (`MPI_Recv`) waits until the message data is stored in the receive buffer.
- The sender can start the send operation whether or not the receiver has initiated the receive.
- Examples illustrate situations where blocking operations succeed, deadlock, or require buffering.

Standard Send/Recv:
- The order of sending and receiving is maintained.
- Order of receive at the target process is not necessarily the same as the order of send from the source process.
- Deadlock can occur if not properly managed.

In summary, blocking operations wait for completion, non-blocking operations allow concurrent work, and synchronous operations synchronize sender and receiver. Careful use of these operations is essential to prevent deadlocks and ensure efficient communication in MPI programs.

Certainly, let's go through the examples you mentioned:

Example 1: Always Succeeds
```c
if (rank == 0) {
   MPI_Send(data, count, MPI_INT, 1, tag, MPI_COMM_WORLD);
   MPI_Recv(data, count, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);
} else if (rank == 1) {
   MPI_Recv(data, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
   MPI_Send(data, count, MPI_INT, 0, tag, MPI_COMM_WORLD);
}
```
- In this example, both processes (rank 0 and rank 1) perform a send and a receive operation.
- The order of send and receive is synchronized, and it ensures that both processes communicate successfully.

Example 2: Always Deadlocks
```c
if (rank == 0) {
   MPI_Recv(data, count, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);
   MPI_Send(data, count, MPI_INT, 1, tag, MPI_COMM_WORLD);
} else if (rank == 1) {
   MPI_Recv(data, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
   MPI_Send(data, count, MPI_INT, 0, tag, MPI_COMM_WORLD);
}
```
- In this example, both processes perform a receive and a send operation in the opposite order.
- This leads to a deadlock because each process is waiting for the other to perform the opposite operation, but neither can proceed.

Example 3: Succeeds with Sufficient Buffering (Unsafe)
```c
if (rank == 0) {
   MPI_Send(data, count, MPI_INT, 1, tag, MPI_COMM_WORLD);
   MPI_Recv(data, count, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);
} else if (rank == 1) {
   MPI_Send(data, count, MPI_INT, 0, tag, MPI_COMM_WORLD);
   MPI_Recv(data, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
}
```
- In this example, both processes perform a send and a receive operation in a synchronized manner.
- However, this code only succeeds if there is sufficient buffering in the MPI implementation.
- It's considered unsafe because it relies on buffering behavior that may not always be present.

These examples illustrate the importance of careful synchronization and ordering in MPI communication to prevent deadlocks and ensure successful data exchange.

Buffering in MPI:
- In MPI, buffering refers to how the implementation stores messages during communication.
- MPI implementations can buffer messages on the sending process, receiving process, both, or neither, depending on the situation.
- Typically, small messages are buffered on the receiving process for efficiency.

Buffered Send Mode:
- MPI has a buffered send mode that allows you to attach and detach a buffer for sending messages.
- Functions like `MPI_Buffer_attach` and `MPI_Buffer_detach` are used to manage buffered sends.
- Buffered send operation: `MPI_Bsend`.

Avoiding Deadlocks:
- Deadlocks can occur when processes wait for each other to perform certain operations.
- Using non-blocking operations (e.g., `MPI_ISend` and `MPI_IRecv`) can help avoid deadlocks.
- In the example provided, replacing either the send or receive operation with non-blocking counterparts would prevent a potential deadlock.

Collective Communication and Computation Operations:
- MPI provides a set of functions for collective communication operations, which involve multiple processes.
- These operations are defined over a group corresponding to the communicator.
- All processes within a communicator must participate in these operations.

MPI_Barrier:
- `MPI_Barrier` is a barrier synchronization operation.
- It blocks the caller until all processes within a communicator have called it.
- It synchronizes all processes within a group to ensure they reach a known point before continuing.
- Barriers are rarely required in a parallel program but can be useful for measuring performance and load balancing.

In summary, buffering in MPI affects how messages are stored during communication, non-blocking operations can help prevent deadlocks, and collective communication operations enable synchronized actions among multiple processes. `MPI_Barrier` is a synchronization tool used to ensure that all processes in a communicator reach a known point before proceeding.

MPI: Global Communications:
- Global communication operations in MPI are blocking calls, meaning processes wait for these calls to complete.
- When no tag is provided, messages are matched based on the order of execution within the group.

- Intercommunicators (communication between two distinct groups of processes) are not allowed.
- You cannot match global communication calls with point-to-point (P2P) receives.

Collective Message Passing with MPI:

1. **MPI_Bcast (Broadcast):**
   - Broadcasts data from a root process to all other processes in the same communicator.
   - All processes must call it with the same arguments.

2. **MPI_Reduce (Reduction):**
   - Performs global reduce operations across all members of a group.
   - Many predefined operations are available, such as MPI_SUM, MPI_MIN, MPI_MAX, MPI_PROD, and logical operations.
   - You can also define custom operations.
   - The result is returned in the output buffer.

3. **Predefined Reduction Operations:**
   - MPI provides various predefined reduction operations, such as MPI_MAX (maximum), MPI_MIN (minimum), MPI_SUM (sum), MPI_PROD (product), logical operations, and more.
   - These operations work with different data types.

Examples of Global Communications:

MPI_Bcast (Broadcast):
```c
MPI_Bcast(buffer, count, MPI_Datatype, root, MPI_COMM_WORLD);
```
- The root process sends data in `buffer` to all other processes in `MPI_COMM_WORLD`.

MPI_Reduce (Reduction):
```c
MPI_Reduce(sendbuf, recvbuf, count, MPI_Datatype, MPI_Op, root, MPI_COMM_WORLD);
```
- The root process collects data from all other processes in `MPI_COMM_WORLD`, performs the operation specified by `MPI_Op`, and stores the result in `recvbuf`.

These collective operations enable synchronization and data exchange among multiple processes in MPI programs.

MPI_Gather:
- `MPI_Gather` is used to gather data from all processes within a communicator and collect it to the root process.
- The root process receives the messages in rank order.
- All processes, including the root, must call `MPI_Gather` with the same arguments.
- This operation is useful when you want to collect data from multiple processes to one process.

MPI_Scatter:
- `MPI_Scatter` is the inverse of `MPI_Gather`.
- It scatters data from the root process to all other processes within a communicator.
- Non-root processes ignore the `sendbuf`.
- This operation is useful when you want to distribute data from one process to multiple processes.

MPI_Allgather:
- `MPI_Allgather` is similar to `MPI_Gather`, but with a twist: it gathers data from all processes to all other processes within a communicator.
- Each process collects data from all others, and the result is stored in `recvbuf`.
- Unlike `MPI_Gather`, `recvbuf` is not ignored.
- This operation is helpful when all processes need data from all others.

MPI_Allreduce:
- `MPI_Allreduce` performs a reduction operation on data across all processes within a communicator.
- All processes collect data from all others and perform an operation (e.g., sum, min, max) on the data.
- The result is available in `recvbuf`.
- This operation is used when you need to compute a collective result from data across all processes.

MPI_Alltoall:
- `MPI_Alltoall` is similar to `MPI_Allgather`, but each process sends distinct data to each of the receivers.
- The `sendbuf` contains data to be sent to all other processes, and the `recvbuf` stores data received from all others.
- It's used when processes need to exchange data with every other process in the communicator.

In summary, these collective communication operations in MPI allow for efficient data exchange, gathering, and reduction across multiple processes within a communicator. The choice of operation depends on your specific communication needs in a parallel program.

Approximation of Pi:
This code calculates the approximate value of Pi (π) using the Monte Carlo method in a parallel manner with MPI (Message Passing Interface). Here's a breakdown of the code:

1. **Include Headers:** The code includes necessary header files for MPI and other libraries.

2. **MPI Initialization:** The program starts by initializing the MPI environment using `MPI_Init`.

3. **Process Information:** It gets information about the number of processes (`numprocs`) and the rank of the current process (`myid`) within the MPI communicator `MPI_COMM_WORLD`.

4. **Calculation Loop:** The main computation loop continues until `done` is set to 1. It prompts the user to enter the number of intervals (n) to use for the Pi calculation.

5. **Broadcast n:** The value of `n` is broadcasted from the root process (rank 0) to all other processes using `MPI_Bcast`. This ensures that all processes have the same value of `n`.

6. **Calculation of Pi:** Each process calculates its part of the Pi approximation using the Monte Carlo method. It divides the interval [0, 1] into `n` subintervals and calculates the sum of areas of rectangles under the curve.

7. **Reduce Operation:** After the calculation, the partial result (`mypi`) from each process is reduced to the root process (rank 0) using `MPI_Reduce` with the `MPI_SUM` operation. This results in the final value of Pi being available on the root process.

8. **Printing Results:** On the root process, the calculated Pi value is printed along with the absolute error compared to the known value of Pi (`PI25DT`).

9. **MPI Finalization:** Finally, the MPI environment is finalized using `MPI_Finalize`.

Handling MPI Errors:

In this section, error handling in MPI is discussed. The code demonstrates how to handle errors gracefully using custom error handlers. Here's an explanation of the key points:

1. **Setting Error Handler:** The code sets a custom error handler using `MPI_Errhandler_set`. It specifies `MPI_ERRORS_RETURN` as the error handler, which means that errors will be returned as error codes instead of causing the program to abort.

2. **Error Handling Function:** The custom error handler function `ErrorHandler` is defined. It can handle MPI errors gracefully.

3. **Handling Errors:** When an MPI error occurs, the error code is captured, and the custom error handler function (`ErrorHandler`) is called.

4. **Error Codes and Messages:** Error codes are converted to comprehensible error messages using `MPI_Error_string`. This allows for better error reporting.

5. **MPI Finalization:** The MPI environment is finalized using `MPI_Finalize`.

This code demonstrates how to compute Pi using parallel processing with MPI and how to handle MPI errors without program termination. It provides a robust way to perform parallel computations while handling potential errors effectively.

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

void ErrorHandler(int error_code) {
   if (error_code != MPI_SUCCESS) {
      char error_string[MPI_MAX_ERROR_STRING];
      int length_of_error_string, error_class;

      MPI_Error_class(error_code, &error_class);
      MPI_Error_string(error_class, error_string, &length_of_error_string);

      fprintf(stderr, "Error class: %s\n", error_string);

      MPI_Error_string(error_code, error_string, &length_of_error_string);
      fprintf(stderr, "Error message: %s\n", error_string);
   }
}

int main(int argc, char *argv[]) {
   int done = 0, n, myid, numprocs, i, error_code;
   double PI25DT = 3.14159265358979323846264338;
```

```
    double mypi, pi, h, sum, x;

    error_code = MPI_Init(&argc, &argv);
    MPI_Errhandler_set(MPI_COMM_WORLD, MPI_ERRORS_RETURN);

    error_code = MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    error_code = MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    while (!done) {
      if (myid == 0) {
        printf("Enter the number of intervals: (0 quits)\n");
        scanf("%d", &n);
      }

      error_code = MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

      if (n == 0) break;

      h = 1.0 / (double)n;
      sum = 0.0;

      for (i = myid + 1; i <= n; i += numprocs) {
        x = h * ((double)i - 0.5);
        sum += 4.0 / (1.0 + x * x);
      }

      mypi = h * sum;

      error_code = MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

      if (myid == 0) {
        printf("Pi is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
      }
    }

    MPI_Finalize();
    return 0;
}
```

Sure, I'd be happy to explain how this code approximates Pi using the Monte Carlo method and some math.

Monte Carlo Method:
The Monte Carlo method is a statistical technique used to estimate numerical results through random sampling. In this case, we are using it to estimate the value of Pi.

Math Behind the Monte Carlo Method for Pi:
1. Imagine a unit square (side length = 1) with a quarter-circle of radius 1 inscribed within it. The center of the circle is at the origin (0,0), and it touches the square at points (0,1) and (1,0).

2. The area of the quarter-circle is one-fourth of the area of the whole circle, which is Pi times the radius squared (Pi * 1 * 1 = Pi).

3. We randomly generate points within the unit square.

4. The probability of a random point falling within the quarter-circle (i.e., inside the circle) is proportional to the ratio of the area of the quarter-circle to the area of the unit square. This ratio is Pi/4.

5. If we generate a large number of random points and count how many fall inside the quarter-circle, we can use the ratio of points inside the circle to the total number of points generated to estimate the value of Pi.

Now, let's look at how the code implements this method:

Code Explanation:
1. We initialize MPI and set an error handler for MPI operations.

2. The code reads the number of intervals (n) for random point generation from the user. More intervals result in a more accurate approximation.

3. Using MPI_Bcast, the value of 'n' is broadcasted to all processes.

4. Each process calculates its part of the approximation. It divides the interval [1, n] into smaller parts based on its process ID (myid) and calculates the sum of f(x) = 4 / (1 + x^2) for the points it generates within its interval.

5. The local approximation (mypi) is computed by each process.

6. Using MPI_Reduce with MPI_SUM operation, all processes send their local approximations to process 0, where the final approximation (pi) is calculated by summing up all the local approximations.

7. Finally, process 0 prints the estimated value of Pi along with the error compared to the known value of Pi (PI25DT).

By repeating this process with different values of 'n' and taking the average, you can get increasingly accurate estimates of Pi using more computational resources.

Two versions of the compare-and-exchange methods:

Version 1 (Direct Exchange):
1. Imagine you have two people, Person 1 (P1) and Person 2 (P2).
2. P1 wants to send a number (let's call it "A") to P2.
3. P2 also has a number (let's call it "B").
4. P2 compares the numbers A and B.
5. If A is bigger than B, P2 sends back A to P1. Otherwise, if B is bigger or they're the same, P2 sends back B to P1.

So, in Version 1, P1 sends a number to P2, and P2 decides which number is bigger and sends it back.

Version 2 (Mutual Comparison):
1. Again, you have two people, P1 and P2.
2. P1 sends a number (A) to P2.
3. P2 sends its number (B) to P1.
4. Now, both P1 and P2 compare the numbers they received.
5. P1 keeps the bigger number, and P2 keeps the smaller number.

In Version 2, both P1 and P2 send their numbers to each other, compare them, and each ends up with one of the numbers based on the comparison.

Use Cases:
These methods are used when two or more people or processes need to make sure they have the same data or decide which data to keep based on a comparison. This is common in computer programs that run on multiple computers or processors to work together effectively and efficiently. It helps them synchronize their actions and share data properly.

Parallel Mergesort

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

void merge(int arr[], int left[], int right[], int l, int r) {
   // Implement merge as before
}

void parallel_merge_sort(int arr[], int size) {
   // Implement parallel merge sort as before
}

int main(int argc, char* argv[]) {
   MPI_Init(&argc, &argv);
   int my_rank;
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   int arr[] = {38, 27, 43, 3, 9, 82, 10};
   int size = sizeof(arr) / sizeof(arr[0]);

   parallel_merge_sort(arr, size);

   if (my_rank == 0) {
      printf("Sorted Array: ");
      for (int i = 0; i < size; i++) {
         printf("%d ", arr[i]);
      }
      printf("\n");
   }

   MPI_Finalize();
   return 0;
}
```

Matrix Multiplication using MPI.

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

#define MATRIX_SIZE 3

void matrix_multiply(int A[MATRIX_SIZE][MATRIX_SIZE], int B[MATRIX_SIZE][MATRIX_SIZE], int C[MATRIX_SIZE][MATRIX_SIZE], int rows_per_process) {
   // Implement matrix multiplication as before
}

int main(int argc, char* argv[]) {
   MPI_Init(&argc, &argv);
   int my_rank, num_procs;
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

   int A[MATRIX_SIZE][MATRIX_SIZE] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
   int B[MATRIX_SIZE][MATRIX_SIZE] = {{9, 8, 7}, {6, 5, 4}, {3, 2, 1}};
   int C[MATRIX_SIZE][MATRIX_SIZE] = {{0}};

   int rows_per_process = MATRIX_SIZE / num_procs;
   int start_row = my_rank * rows_per_process;

   matrix_multiply(&A[start_row], B, &C[start_row], rows_per_process);

   if (my_rank == 0) {
      for (int i = 0; i < MATRIX_SIZE; i++) {
         for (int j = 0; j < MATRIX_SIZE; j++) {
            printf("%d ", C[i][j]);
         }
         printf("\n");
      }
   }

   MPI_Finalize();
   return 0;
}
```

These shorter versions of the code examples maintain the functionality of parallel Mergesort and Matrix Multiplication using MPI while removing some of the redundant details.

Example of parallel mergesort and matrix multiplication
Parallel Mergesort Example (Mathematics):

Suppose we have an unsorted array of numbers: `[8, 3, 5, 2, 9, 1, 6, 4, 7]`.

1. **Parallel Division:** We divide the array into two parts, assigning half of the array to one process and the other half to another process. For simplicity, let's say Process 1 gets `[8, 3, 5]`, and Process 2 gets `[2, 9, 1, 6, 4, 7]`.

2. **Local Sorting:** Each process sorts its portion of the array. Process 1 sorts `[8, 3, 5]` into `[3, 5, 8]`, and Process 2 sorts `[2, 9, 1, 6, 4, 7]` into `[1, 2, 4, 6, 7, 9]`.

3. **Parallel Merge:** Now, the two sorted subarrays need to be merged together in parallel. Process 1 has `[3, 5, 8]`, and Process 2 has `[1, 2, 4, 6, 7, 9]`.

   - First, both processes compare the smallest elements from their subarrays. Process 1 compares `3` with `1`. `1` is smaller, so it sends `1` to Process 1.

   - Next, they compare `3` (from Process 1) with `2` (from Process 2). `2` is smaller, so it sends `2` to Process 1.

   - This process continues, and they exchange elements in a way that the merged subarray on Process 1 becomes `[1, 2, 3, 4, 5, 6, 7, 8, 9]`.

4. **Final Result:** Process 1 now has the entire sorted array `[1, 2, 3, 4, 5, 6, 7, 8, 9]`.

This demonstrates the basic idea of parallel mergesort, where parallel processes work on sorting and merging parts of the array, ultimately resulting in a sorted array.

Matrix Multiplication Example (Mathematics):

Suppose we have two matrices:

Matrix A (2x2):
```
| 1  2 |
| 3  4 |
```

Matrix B (2x2):
```
| 5  6 |
| 7  8 |
```

We want to perform matrix multiplication C = A * B.

1. **Matrix Multiplication:** To compute C, we take the dot product of rows from A and columns from B.

   - C[0][0] = Row 1 of A * Column 1 of B = (1*5) + (2*7) = 19

- C[0][1] = Row 1 of A * Column 2 of B = (1*6) + (2*8) = 22
- C[1][0] = Row 2 of A * Column 1 of B = (3*5) + (4*7) = 43
- C[1][1] = Row 2 of A * Column 2 of B = (3*6) + (4*8) = 50

The resulting matrix C is:

Matrix C (2x2):
```
| 19  22 |
| 43  50 |
```

This demonstrates a basic matrix multiplication operation where we calculate the values of C by taking dot products of rows and columns from A and B.

These examples provide a mathematical understanding of how parallel mergesort and matrix multiplication work without going into the actual code implementation.