PRACTICAL 1

This

This code is an MPI (Message Passing Interface) program written in C for matrix multiplication using a master-worker approach. Let's break down the code and explain its functionality:

1. **Header Files**:
   ```c
   #include <stdio.h>
   #include <stdlib.h>
   #include <mpi.h>
   ```
   Standard C headers and MPI header are included.

2. **Constants**:
   ```c
   #define SIZE 10
   #define FROM_MASTER 1
   #define FROM_WORKER 2
   #define DEBUG 1
   ```
   These macros define the size of the matrices, message tags for communication between master and workers, and a debug flag.

3. **Global Variables**:
   ```c
   MPI_Status status;
   static double a[SIZE][SIZE];
   static double b[SIZE][SIZE];
   static double c[SIZE][SIZE];
   ```
   `status` is used for receiving status information in MPI communication. `a`, `b`, and `c` are matrices for multiplication.

4. **Helper Functions**:
   - `init_matrix()`: Initializes matrices `a` and `b` with value 1.
   - `print_matrix()`: Prints matrix `c`.

5. **Main Function**:
   - **MPI Initialization**:
     ```c
     MPI_Init(&argc, &argv);
     MPI_Comm_size(MPI_COMM_WORLD, &nproc);
     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
     ```

Initializes MPI environment, retrieves the number of processes (`nproc`), and the rank of the current process (`myrank`).

- **Matrix Initialization and Timing**:
  - Master Process (`myrank == 0`):
    - Initializes matrices `a` and `b`.
    - Starts a timer using `MPI_Wtime()`.
  - Worker Processes (`myrank > 0`):
    - Receive matrix data from the master.

- **Matrix Multiplication**:
  - Master Process:
    - Performs matrix multiplication if there's only one process (`nproc == 1`).
    - If there are multiple processes, divides the matrices into blocks and distributes them to worker processes.
  - Worker Processes:
    - Receive data from the master.
    - Perform matrix multiplication for assigned block.

- **Result Gathering**:
  - Master Process:
    - Collects results from worker processes.
    - Stops the timer and prints execution time.
  - Worker Processes:
    - Send results back to the master.

- **MPI Finalization**:
  ```c
  MPI_Finalize();
  ```
  Finalizes MPI environment before program termination.

This code demonstrates a basic master-worker parallelization strategy for matrix multiplication using MPI. It divides the matrices into blocks and distributes computation among multiple processes, leveraging parallelism for faster execution.

EXE TIME
In the provided code, the execution time is measured using MPI's `MPI_Wtime()` function. Here's how the execution time is calculated and printed:

1. **Timing Initialization**:
   - Before the computation starts, the code initializes a variable `start_time` to record the start time of the computation.

```c
double start_time, end_time;
start_time = MPI_Wtime();
```

2. **Timing Calculation**:
   - After the computation completes, the code records the end time of the computation.
   ```c
   end_time = MPI_Wtime();
   ```

3. **Execution Time Calculation**:
   - The execution time is then calculated by taking the difference between the end time and the start time.
   ```c
   double execution_time = end_time - start_time;
   ```

4. **Printing Execution Time**:
   - Finally, the code prints the execution time.
   ```c
   printf("Execution time on %2d nodes: %f\n", nproc, execution_time);
   ```

This approach provides a simple and accurate way to measure the time taken for the computation to execute. By subtracting the start time from the end time, we obtain the elapsed time in seconds. This timing information helps in understanding the performance characteristics of the program, especially in parallel and distributed computing environments where the execution time may vary depending on factors such as the number of processes and the workload distribution.

MPI_Send:

MPI_Send is used to send a message from one process to another. Its syntax is:

c

Copy code

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

- `buf`: Pointer to the send buffer, i.e., the data that is being sent.

- `count`: Number of elements in the send buffer.
- `datatype`: MPI datatype of each element in the send buffer.
- `dest`: Rank of the destination process to which the message will be sent.
- `tag`: Message tag, which can be used by the receiver to identify different types of messages.
- `comm`: MPI communicator that defines the group of processes involved in the communication.

MPI_Recv:

MPI_Recv is used to receive a message sent by another process. Its syntax is:

c

Copy code

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

- `buf`: Pointer to the receive buffer, where the received data will be stored.
- `count`: Maximum number of elements that can be received in the receive buffer.
- `datatype`: MPI datatype of each element in the receive buffer.
- `source`: Rank of the source process from which the message will be received.
- `tag`: Message tag to match with the tag of the message being received.
- `comm`: MPI communicator that defines the group of processes involved in the communication.
- `status`: Pointer to an MPI_Status object, which provides information about the received message (e.g., source, tag, etc.).

PRACTICAL 2

This code is a simple MPI program written in C that demonstrates the initialization of the MPI environment, obtaining the rank and size of the MPI communicator, and printing a "Hello" message from each process.

Let's break down the code step by step:

1. **Include Headers**:
   ```c
   #include <stdio.h>
   #include "mpi.h"
   ```
   These lines include the necessary headers. `<stdio.h>` is for standard input/output functions, and `"mpi.h"` is the header file provided by the MPI library.

2. **Main Function**:
   ```c
   int main(int argc, char *argv[])
   ```
   The `main` function is the entry point of the program, which takes command-line arguments `argc` (argument count) and `argv` (argument vector).

3. **MPI Initialization**:
   ```c
   MPI_Init(&argc, &argv);
   ```
   This line initializes the MPI environment. It should be called before any other MPI function. The `argc` and `argv` parameters are used to initialize MPI and may be modified by the MPI library.

4. **Obtaining Rank and Size**:
   ```c
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   ```
   - `MPI_Comm_rank` retrieves the rank (process identifier) of the calling process within the specified communicator (`MPI_COMM_WORLD` is the default communicator that includes all processes).
   - `MPI_Comm_size` retrieves the total number of processes in the communicator.

5. **Printing Process Information**:
   ```c
   printf("Hello from process %d of %d\n", rank, size);
   ```

This line prints a "Hello" message from each process. `rank` is the rank of the current process, and `size` is the total number of processes.

6. **MPI Finalization**:
```c
MPI_Finalize();
```
This line finalizes the MPI environment, freeing any resources associated with it. It should be called before the program exits.

7. **Return Statement**:
```c
return 0;
```
This statement returns an integer value (`0`) to the operating system to indicate successful program execution.

Overall, this code initializes the MPI environment, retrieves the rank and size of each process, prints a "Hello" message from each process indicating its rank and the total number of processes, and then finalizes the MPI environment before exiting.

PRACTICAL 3

Sure, let's break down each segment of the code:

1. **Header Files and Definitions**:
```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define n 10
int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int a2[1000];
```
- Includes necessary header files for MPI and standard C libraries.
- Defines the size of the array `n` as 10 and initializes an array `a` with some values.
- Declares an array `a2` which will be used by the slave processes to store received array segments.

2. **Main Function**:
```c
int main(int argc, char* argv[])
```
- The entry point of the program.

3. **MPI Initialization**:
   ```c
   MPI_Init(&argc, &argv);
   ```
   - Initializes the MPI environment.

4. **Retrieve Process Information**:
   ```c
   int pid, np;
   MPI_Status status;
   MPI_Comm_rank(MPI_COMM_WORLD, &pid);
   MPI_Comm_size(MPI_COMM_WORLD, &np);
   ```
   - Declares variables `pid` and `np` to store the process ID and the total number of processes respectively.
   - Retrieves the process ID (`pid`) and the number of processes (`np`) using `MPI_Comm_rank` and `MPI_Comm_size`.

5. **Master Process (pid == 0)**:
   ```c
   if (pid == 0) {
      // Master process code
   }
   ```
   - Checks if the process ID is 0, indicating the master process.

6. **Divide and Distribute Array**:
   ```c
   elements_per_process = n / np;
   ```
   - Calculates the number of elements each process will receive.

7. **Send Array Segments to Worker Processes**:
   ```c
   MPI_Send(&elements_per_process, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
   MPI_Send(&a[index], elements_per_process, MPI_INT, i, 0, MPI_COMM_WORLD);
   ```
   - Sends the number of elements and the corresponding array segment to each worker process.

8. **Calculate Partial Sum in Master Process**:
   ```c
   // Master process calculates its own partial sum
   ```

```c
int sum = 0;
for (i = 0; i < elements_per_process; i++)
    sum += a[i];
```
- Calculates the partial sum of the array segment owned by the master process.

9. **Receive Partial Sums from Worker Processes**:
```c
MPI_Recv(&tmp, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
```
- Receives partial sums from all worker processes.

10. **Print Final Sum**:
```c
printf("Sum of array is : %d\n", sum);
```
- Prints the total sum of the array.

11. **Slave Processes (pid > 0)**:
```c
else {
    // Slave process code
}
```
- Handles code execution for slave processes.

12. **Receive Array Segments**:
```c
MPI_Recv(&n_elements_recieved, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
MPI_Recv(&a2, n_elements_recieved, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
```
- Receives the number of elements and the array segment from the master process.

13. **Calculate Partial Sum in Slave Process**:
```c
for (int i = 0; i < n_elements_recieved; i++)
    partial_sum += a2[i];
```
- Calculates the partial sum of the received array segment.

14. **Send Partial Sum to Master Process**:
```c
MPI_Send(&partial_sum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
```

- Sends the partial sum back to the master process.

15. **MPI Finalization**:
```c
MPI_Finalize();
```
- Cleans up MPI state before process exit.

This breakdown explains the functionality of each segment of the code, including MPI initialization, array distribution, partial sum calculation, communication between processes, and finalization.

PRACTICAL 6

This code is an MPI (Message Passing Interface) parallel implementation of the quicksort algorithm to sort an array of integers. Let's break down each segment of the code:

1. **Header Files and Function Definitions**:
   - The code includes necessary header files such as `mpi.h`, `stdio.h`, `stdlib.h`, `time.h`, and `unistd.h`.
   - It defines two functions: `swap()` for swapping two elements in an array, and `quicksort()` for performing the quicksort algorithm recursively on an array.

2. **Merge Function**:
   - The `merge()` function merges two sorted arrays into a single sorted array. This function is used in the merge step of the parallel quicksort algorithm.

3. **Main Function**:
   - The main function initializes MPI environment using `MPI_Init()` and retrieves the process ID (`rank_of_process`) and the number of processes (`number_of_process`) using `MPI_Comm_rank` and `MPI_Comm_size` respectively.
   - It checks if the correct number of arguments is provided, expecting two filenames: input file and output file.
   - The master process (rank 0) reads input data from a file, distributes chunks of the array to worker processes using MPI collective communication functions (`MPI_Bcast` and `MPI_Scatter`), and then sorts its own chunk using quicksort.
   - Worker processes receive their chunks of the array using `MPI_Scatter`, sort their chunks using quicksort, and then participate in the merge step by sending and receiving chunks to/from other processes.
   - After sorting and merging, the master process gathers the sorted chunks from worker processes and writes the sorted array to an output file.
   - Timing of the computation is performed using `MPI_Wtime()` to measure the total execution time.
   - Finally, the program prints the sorted array and the total execution time.

4. **MPI Initialization and Finalization**:
   - The code initializes MPI environment using `MPI_Init()` and finalizes it using `MPI_Finalize()` before program termination.

This code demonstrates a parallel implementation of the quicksort algorithm using MPI, where the array is divided into chunks and sorted concurrently by multiple processes. The sorted chunks are then merged together to produce the final sorted array.

PRACTICAL 7

When you execute a program that uses pthreads (POSIX Threads), several steps occur in the execution process:

1. **Program Startup**:
   - The program starts executing from the `main` function.

2. **Thread Creation**:
   - When a pthread function like `pthread_create()` is called, a new thread is created within the process.
   - The new thread begins execution by calling the function passed as an argument to `pthread_create()`. In the provided code, this function is `generate_fibonacci()`.

3. **Thread Execution**:
   - Each thread executes independently, sharing the same memory space as other threads in the process.
   - In the example code, multiple threads execute the `generate_fibonacci()` function concurrently to calculate Fibonacci numbers.

4. **Thread Synchronization** (if applicable):
   - If synchronization mechanisms like mutexes, condition variables, or barriers are used, threads may synchronize their execution to avoid race conditions or ensure certain conditions are met.
   - In the provided code, a mutex is used to synchronize access to the `fib` array to prevent concurrent writes by multiple threads.

5. **Thread Termination**:
   - Each thread executes until it reaches the end of its function or explicitly calls `pthread_exit()` to terminate itself.
   - In the example code, each thread exits after finishing the calculation of Fibonacci numbers.

6. **Main Thread Wait**:
   - The main thread may wait for other threads to complete their execution using `pthread_join()`.

- In the provided code, the main thread waits for the Fibonacci generation thread to finish before proceeding to print the result.

7. **Program Termination**:
   - Once all threads have completed their execution, the program continues executing the remaining code in the `main` function or any other functions called from it.
   - After reaching the end of the `main` function or encountering a `return` statement, the program terminates.

Throughout this execution process, the operating system scheduler manages the execution of threads, deciding when each thread should run and for how long. The use of pthreads allows for concurrent execution of multiple tasks within a single process, leveraging the capabilities of modern multi-core processors for improved performance.

PROGRAM
This program generates the Fibonacci series using multithreading in C. Let's break down the code segment by segment:

1. **Header Files and Definitions**:
   - The code includes the necessary header files `stdio.h` and `pthread.h`.
   - It defines `MAX_TERM` to limit the number of terms in the Fibonacci series.

2. **Global Variables**:
   - `fib` is an array used to store Fibonacci terms.
   - `mutex` is a mutex variable used to synchronize access to the `fib` array when multiple threads are writing to it concurrently.

3. **Function to Generate Fibonacci Series**:
   - The `generate_fibonacci` function is executed by each thread to generate the Fibonacci series up to the specified number of terms (`n`).
   - It uses a loop to calculate Fibonacci terms and stores them in the `fib` array.
   - Access to the `fib` array is synchronized using the mutex to prevent race conditions when multiple threads are writing to it simultaneously.

4. **Main Function**:
   - The `main` function initializes the mutex using `pthread_mutex_init`.
   - It prompts the user to enter the number of terms for the Fibonacci series, ensuring it does not exceed the maximum limit.
   - It creates a thread (`tid`) using `pthread_create` to generate the Fibonacci series concurrently.
   - The main thread waits for the Fibonacci generation thread to finish using `pthread_join`.
   - After the thread completes, it prints the generated Fibonacci series.

- Finally, it destroys the mutex using `pthread_mutex_destroy`.

5. **Explanation of Mutex Usage**:
   - The mutex ensures that only one thread can modify the `fib` array at a time. This prevents race conditions where multiple threads might try to write to the same array element simultaneously, leading to incorrect results.
   - The mutex is locked using `pthread_mutex_lock` before accessing the `fib` array and unlocked using `pthread_mutex_unlock` afterward to allow other threads to access it.

Overall, this program demonstrates how to use multithreading and mutexes in C to generate the Fibonacci series concurrently, improving performance by utilizing multiple CPU cores.

Function Purpose	C Function Call
Initialize MPI	`int MPI_Init(int *argc, char **argv)`
Determine number of processes within a communicator	`int MPI_Comm_size(MPI_Comm comm, int *size)`
Determine processor rank within a communicator	`int MPI_Comm_rank(MPI_Comm comm, int *rank)`
Exit MPI (must be called last by all processors)	`int MPI_Finalize()`
Send a message	`int MPI_Send (void *buf,int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
Receive a message	`int MPI_Recv (void *buf,int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`

PRACTICAL 7

Sure, let's go through the code line by line and explain each part:

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```
These lines include necessary header files. `pthread.h` provides the functionality for working with POSIX threads, `stdio.h` for standard input/output operations, `stdlib.h` for memory allocation and deallocation functions, `string.h` for string manipulation functions, and `unistd.h` for standard symbolic constants and types.

```c
#define jobs 10
```
This line defines a macro named `jobs` with a value of `10`. This macro will be used later to specify the number of threads to be created.

```c
pthread_t tid[jobs];
```
This declares an array `tid` of type `pthread_t` with a size of `jobs`. `pthread_t` is the data type used to represent threads in POSIX systems. This array will store the thread IDs of the threads created.

```c
void *trythis(void *arg) {
```
This line defines the thread function named `trythis`. It takes a `void` pointer `arg` as an argument and returns a `void` pointer. This is a typical signature for a thread function in pthreads.

```c
int counter = *((int *)arg);
```
This line retrieves the integer argument passed to the thread function and stores it in the variable `counter`. The argument is passed as a void pointer and needs to be dereferenced and cast to an integer pointer before being dereferenced again to get the actual integer value.

```c
```

```
printf("\n Job %d has started\n", counter);
```
This line prints a message indicating that the thread has started. It includes the value of `counter` which represents the job number of the thread.

```c
for (unsigned long long i = 0; i < 90000000; i++) {
    // Introduce a side effect to prevent optimization
    volatile unsigned long long temp = i * 2; // Example computation
}
```
This loop introduces a computational workload to simulate some processing that the thread is performing. It iterates 90 million times and performs a simple computation `i * 2` in each iteration. The `volatile` keyword is used to prevent the compiler from optimizing away the loop.

```c
printf("\n Job %d has finished\n", counter);
```
This line prints a message indicating that the thread has finished its job. Again, it includes the value of `counter` to identify the job number of the thread.

```c
int main(void) {
```
This line defines the `main` function, which is the entry point of the program.

```c
int *thread_args = malloc(jobs * sizeof(int));
```
This line dynamically allocates memory to store arguments for each thread. It allocates memory for an array of integers with a size equal to the number of jobs multiplied by the size of an integer.

```c
if (thread_args == NULL) {
    fprintf(stderr, "Memory allocation failed\n");
    return 1;
}
```
This block of code checks if memory allocation was successful. If `malloc` returns `NULL`, it means that memory allocation failed. In that case, an error message is printed to the standard error stream (`stderr`), and the program returns with an exit status of `1`, indicating an error.

```c
```

```
for (int i = 0; i < jobs; i++) {
    thread_args[i] = i;
    int error = pthread_create(&(tid[i]), NULL, &trythis, &(thread_args[i]));
    if (error != 0)
        printf("\nThread can't be created : [%s]", strerror(error));
}
```

This loop creates threads using the `pthread_create` function. For each thread, it sets up the argument (`thread_args[i] = i`) and passes it to the `trythis` function. If an error occurs during thread creation, an error message is printed.

```c
for (int i = 0; i < jobs; i++) {
    pthread_join(tid[i], NULL);
}
```

This loop waits for each thread to finish execution using the `pthread_join` function. It ensures that the main thread waits for all created threads to complete before proceeding further.

```c
free(thread_args);
```

This line frees the dynamically allocated memory for `thread_args` using the `free` function. It's important to free dynamically allocated memory to avoid memory leaks.

```c
return 0;
```

This line returns `0` from the `main` function, indicating successful execution of the program.