Lab 1: From Shared Memory to Distributed Memory with MPI

Student Manual

Welcome to Lab 1!

In **Lab 0,** you learned how to write parallel programs using threads (Pthreads) on a shared-memory system. You saw how multiple threads can work together to solve a problem faster, but you also discovered limitations: your program could only run on a single machine, and performance gains were limited by memory bandwidth and other factors.

In this lab, we will take the next big step: **distributed memory programming with MPI**. You will learn how to write programs that can run on multiple computers connected by a network – the foundation of modern supercomputers and cloud computing.

---

1. Learning Objectives

By the end of this lab, you will be able to:

- Explain why shared-memory systems cannot scale indefinitely and why distributed memory is necessary.
- Understand the MPI (Message Passing Interface) programming model: SPMD, ranks, communicators.
- Write, compile, and run basic MPI programs in C.
- Use point-to-point communication (`MPI_Send`, `MPI_Recv`) to exchange messages.
- Use collective communication (`MPI_Bcast`, `MPI_Reduce`) to efficiently distribute data and combine results.
- Implement the counting problem from Lab 0 using MPI and compare its performance with the Pthreads version.
- Measure the cost of communication using a ping-pong program.
- Analyse speedup and efficiency, and understand why MPI programs behave differently from threaded programs.

---

2. Before You Start: What You Need

2.1 Prerequisites

- You should have completed Lab 0 and understand the Pthreads counting program.
- You should be comfortable with C programming (pointers, dynamic memory, compilation).
- You need access to a Linux environment (native, VM, or WSL) with MPI installed.

2.2 Software Installation

If MPI is not already installed on your system, install it now:

```
sudo apt update
sudo apt install openmpi-bin openmpi-common libopenmpi-dev
```

Verify the installation:

```
mpirun --version
```

You should see output similar to: `mpirun (Open MPI) 4.1.4`

2.3 Compiling and Running MPI Programs

- To compile an MPI program, use the `mpicc` wrapper:

  ```
  mpicc -O2 -o myprogram myprogram.c
  ```

- To run your program with 4 processes:

  ```
  mpirun -np 4 ./myprogram
  ```

The `-np` option specifies the number of processes. These processes can run on a single multicore machine (each on a different core) or across a cluster – MPI handles both.

---

3. Review: The Counting Problem with Pthreads

Before we move to MPI, let's revisit the parallel counting program you wrote in Lab 0. This will help us see the key differences.

3.1 Sequential Version (`count_seq.c`)

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

#define N 100000000
#define TARGET 42

int main() {
    int *arr = malloc(N * sizeof(int));
    // Fill array with random numbers 0-99
    srand(time(NULL));
    for (long i = 0; i < N; i++)
        arr[i] = rand() % 100;

    clock_t start = clock();
    long count = 0;
    for (long i = 0; i < N; i++)
        if (arr[i] == TARGET) count++;
    clock_t end = clock();

    double elapsed = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Count = %ld\n", count);
    printf("Time = %.2f seconds\n", elapsed);
    free(arr);
    return 0;
}
```

3.2 Pthreads Parallel Version (`count_pthread.c`)

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>

#define N 100000000
#define TARGET 42
#define NUM_THREADS 4

int *arr;
long global_count = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

typedef struct {
    long start;
    long end;
} ThreadArgs;

void* count_chunk(void* arg) {
    ThreadArgs *args = (ThreadArgs*) arg;
```

```
    long local_count = 0;
    for (long i = args->start; i < args->end; i++)
        if (arr[i] == TARGET) local_count++;

    pthread_mutex_lock(&mutex);
    global_count += local_count;
    pthread_mutex_unlock(&mutex);
    return NULL;
}

int main() {
    arr = malloc(N * sizeof(int));
    // Fill array
    srand(time(NULL));
    for (long i = 0; i < N; i++)
        arr[i] = rand() % 100;

    pthread_t threads[NUM_THREADS];
    ThreadArgs args[NUM_THREADS];
    long chunk_size = N / NUM_THREADS;

    clock_t start = clock();

    for (int t = 0; t < NUM_THREADS; t++) {
        args[t].start = t * chunk_size;
        args[t].end = (t == NUM_THREADS - 1) ? N : (t + 1) * chunk_size;
        pthread_create(&threads[t], NULL, count_chunk, &args[t]);
    }

    for (int t = 0; t < NUM_THREADS; t++)
        pthread_join(threads[t], NULL);

    clock_t end = clock();
    double elapsed = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Count = %ld\n", global_count);
    printf("Time with %d threads = %.2f seconds\n", NUM_THREADS, elapsed);

    free(arr);
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

3.3 Key Observations from Lab 0

- **Threads share memory:** All threads access the same array `arr` and the same global counter `global_count`.
- **Synchronisation is needed:** A mutex protects the global counter from race conditions.
- **Performance is good but not perfect:** Speedup is limited by:
    - Thread creation overhead
    - Mutex contention (even if minimal)
    - Memory bandwidth saturation (all threads reading from the same memory)

- Cache effects

The big limitation: This program can only run on one machine. To use hundreds or thousands of cores, we need a different approach.

---

4. Why Distributed Memory?

4.1 The Scalability Wall

Imagine you have a problem so large that it doesn't fit in the memory of a single computer. Or you need so much computing power that a single machine cannot provide enough cores. In both cases, you need multiple machines working together.

In a **distributed memory** system:

- Each machine (node) has its own processor(s) and its own private memory.
- Nodes are connected by a network (Ethernet, InfiniBand, etc.).
- Processes on different nodes cannot directly access each other's memory.
- They must cooperate by **explicitly sending messages** (data) to each other.

4.2 The Message Passing Interface (MPI)

MPI is the standard for programming distributed memory systems. It provides:

- A way to start the same program on many processes (SPMD model).
- Functions to send and receive messages.
- Functions to perform collective operations (broadcast, reduce, etc.) efficiently.
- The ability to run on anything from a laptop to the world's largest supercomputer.

4.3 The MPI Model: SPMD

SPMD stands for **Single Program, Multiple Data**. This means:

- You write one program.
- That program is launched on all processes.
- Each process has a unique **rank** (an ID number from 0 to `size-1`).
- Based on its rank, each process can work on a different part of the data or follow a different execution path.

Think of it like a team of workers: they all have the same instruction manual, but each worker knows their own ID and does the part of the job assigned to that ID.

---

5. Essential MPI Concepts and Functions

5.1 Initialisation and Finalisation

Every MPI program must start with `MPI_Init` and end with `MPI_Finalize`.

```
#include <mpi.h>

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);   // Initialize MPI

    // ... your parallel code ...

    MPI_Finalize();            // Clean up MPI
    return 0;
}
```

5.2 Getting Information: Rank and Size

- `MPI_Comm_size` tells you how many processes are in the communicator (usually `MPI_COMM_WORLD`, which includes all processes).
- `MPI_Comm_rank` tells you the rank of the current process.

```
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
```

5.3 Point-to-Point Communication: Sending and Receiving

- `MPI_Send` sends a message to a specific process.
- `MPI_Recv` receives a message from a specific process.

```
MPI_Send(&data, count, datatype, dest, tag, MPI_COMM_WORLD);
MPI_Recv(&data, count, datatype, source, tag, MPI_COMM_WORLD, &status);
```

Parameters:

- `data`: pointer to the data to send/receive
- `count`: number of elements
- `datatype`: type of each element (`MPI_INT`, `MPI_DOUBLE`, etc.)
- `dest` / `source`: rank of the destination/source process
- `tag`: a message identifier (useful for distinguishing different messages)

- `status`: for `MPI_Recv`, information about the received message (can be `MPI_STATUS_IGNORE` if not needed)

Important: These are **blocking** calls. `MPI_Send` does not return until the send buffer can be reused. `MPI_Recv` does not return until the message has been received.

5.4 Collective Communication

Collective operations involve **all** processes in a communicator.

- **Broadcast (`MPI_Bcast`):** One process (the root) sends the same data to all other processes.

```
MPI_Bcast(&data, count, datatype, root, MPI_COMM_WORLD);
```

- **Reduce (`MPI_Reduce`):** All processes contribute data, and the result (sum, max, etc.) is returned to the root.

```
MPI_Reduce(&sendbuf, &recvbuf, count, datatype, op, root, MPI_COMM_WORLD);
```

  Common operations: `MPI_SUM`, `MPI_MAX`, `MPI_MIN`, `MPI_PROD`.

Why use collectives? They are highly optimised (often using tree algorithms) and much more efficient than writing your own with point-to-point sends.

5.5 Timing with MPI

MPI provides a high-resolution timer: `MPI_Wtime()` returns the current wall-clock time in seconds.

```
double start = MPI_Wtime();
// ... work ...
double end = MPI_Wtime();
printf("Time = %f seconds\n", end - start);
```

5.6 Common MPI Datatypes

C type	MPI datatype
`int`	`MPI_INT`
`long`	`MPI_LONG`
`float`	`MPI_FLOAT`
`double`	`MPI_DOUBLE`
`char`	`MPI_CHAR`

6. Hands-On Exercises

Now it's time to write some MPI programs! Follow each exercise carefully. The first two are warm-ups; Exercise 3 is the main event where you'll reimplement the counting problem.

Exercise 1: Hello World with MPI

Goal: Write a simple MPI program where each process prints its rank and the total number of processes.

Steps:

1. Create a file named `hello_mpi.c`.
2. Include the MPI header: `#include <mpi.h>`.
3. In `main`, call `MPI_Init`.
4. Get the world size and rank using `MPI_Comm_size` and `MPI_Comm_rank`.
5. Print a message: `"Hello from process X of Y"`.
6. Call `MPI_Finalize`.

Code:

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    printf("Hello from process %d of %d\n", world_rank, world_size);

    MPI_Finalize();
    return 0;
}
```

Compile and run:

```
mpicc -o hello_mpi hello_mpi.c
mpirun -np 4 ./hello_mpi
```

Expected output (order may vary):

```
Hello from process 0 of 4
Hello from process 1 of 4
Hello from process 2 of 4
Hello from process 3 of 4
```

Questions:

- Why might the output appear in a different order each time?
- What happens if you run with `-np 1`? With `-np 8`?

Exercise 2: Ping-Pong – Measuring Communication Cost

Goal: Two processes (rank 0 and rank 1) send a message back and forth many times. This measures the round-trip time and helps you understand the cost of communication.

Steps:

1. Create `pingpong.c`.
2. Include MPI header and `stdio.h`.
3. Define a constant `PING_PONG_LIMIT` (e.g., 10000).
4. Initialise MPI, get rank and size.
5. Ensure exactly 2 processes are used; if not, print an error and abort.
6. Determine the partner rank (if you're 0, partner is 1; if you're 1, partner is 0).
7. Record the start time with `MPI_Wtime()`.
8. In a loop that runs `PING_PONG_LIMIT` times:
    - If it's your turn to send (based on the counter parity), increment the counter and send it to your partner.
    - Otherwise, receive the counter from your partner.
9. After the loop, record the end time.
10. Only process 0 prints the total time.

Code:

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

#define PING_PONG_LIMIT 10000

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
```

```
    if (world_size != 2) {
        fprintf(stderr, "This program requires exactly 2 processes.\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    int ping_pong_count = 0;
    int partner_rank = (world_rank == 0) ? 1 : 0;

    double start_time = MPI_Wtime();

    while (ping_pong_count < PING_PONG_LIMIT) {
        if (world_rank == ping_pong_count % 2) {
            ping_pong_count++;
            MPI_Send(&ping_pong_count, 1, MPI_INT, partner_rank, 0, MPI_COMM_WORLD);
            printf("Process %d sent %d to process %d\n", world_rank, ping_pong_count, partner_rank);
        } else {
            MPI_Recv(&ping_pong_count, 1, MPI_INT, partner_rank, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("Process %d received %d from process %d\n", world_rank, ping_pong_count, partner_rank);
        }
    }

    double end_time = MPI_Wtime();
    if (world_rank == 0) {
        printf("Ping-pong completed in %f seconds\n", end_time - start_time);
        printf("Average round-trip time: %f microseconds\n",
               (end_time - start_time) * 1e6 / PING_PONG_LIMIT);
    }

    MPI_Finalize();
    return 0;
}
```

Compile and run:

```
mpicc -o pingpong pingpong.c
mpirun -np 2 ./pingpong
```

Analysis:

- The time includes both the increment and the communication. To measure pure communication latency, modify the code to send a fixed value (e.g., `0`) without incrementing.
- Try increasing the message size: send an array of 10, 100, 1000 integers. What happens to the round-trip time?

Compare with Pthreads: In the Pthreads version, updating a shared counter was essentially free (a memory operation). Here, even a single integer takes microseconds. This is the cost of distributed memory.

Exercise 3: Parallel Counting with MPI

Goal: Reimplement the counting problem from Lab 0 using MPI. Each process will generate its own portion of the array, count occurrences of the target locally, and then the partial counts will be summed using `MPI_Reduce`.

Key differences from Pthreads:

- No shared array – each process has its own private chunk.
- No mutex – each process maintains its own local count.
- Combining results is done via message passing (`MPI_Reduce`).

Steps:

1. Create `count_mpi.c`.
2. Include `mpi.h`, `stdio.h`, `stdlib.h`, `time.h`.
3. Define constants: `N` (total elements) and `TARGET` (42).
4. In `main`, initialise MPI and get rank and size.
5. **Partition the data:** Compute how many elements each process should handle. Distribute any remainder so that the first `remainder` processes get one extra element.
    - `elements_per_proc = N / size`
    - `remainder = N % size`
    - If `rank < remainder`, this process gets `elements_per_proc + 1` elements.
    - Otherwise, it gets `elements_per_proc` elements.
6. Allocate a local array of the appropriate size.
7. Seed the random number generator with `time(NULL) + rank` so each process gets a different sequence.
8. Fill the local array with random numbers 0–99.
9. Synchronise all processes with `MPI_Barrier` (to ensure they start timing together).
10. Record start time with `MPI_Wtime()`.
11. Loop through the local array and count occurrences of `TARGET` into `local_count`.
12. Use `MPI_Reduce` to sum all `local_count` values into `global_count` on process 0.
13. Record end time.
14. Process 0 prints the total count and the execution time.
15. Free the local array and call `MPI_Finalize`.

Code:

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

#define N 100000000      // total number of elements
#define TARGET 42

int main(int argc, char** argv) {
    int rank, size;
```

```
long local_count = 0;
long global_count = 0;
double start, end;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

// Determine how many elements each process gets
long elements_per_proc = N / size;
long remainder = N % size;
long local_n;

if (rank < remainder) {
    local_n = elements_per_proc + 1;
} else {
    local_n = elements_per_proc;
}

// Allocate local array
int *local_arr = (int*) malloc(local_n * sizeof(int));
if (!local_arr) {
    fprintf(stderr, "Process %d: malloc failed\n", rank);
    MPI_Abort(MPI_COMM_WORLD, 1);
}

// Seed random number generator differently for each process
srand(time(NULL) + rank);

// Fill local array with random numbers 0-99
for (long i = 0; i < local_n; i++) {
    local_arr[i] = rand() % 100;
}

// Synchronise before timing
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();

// Count occurrences of TARGET in local chunk
for (long i = 0; i < local_n; i++) {
    if (local_arr[i] == TARGET)
        local_count++;
}

// Reduce all local counts to global count on rank 0
MPI_Reduce(&local_count, &global_count, 1, MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD);

end = MPI_Wtime();

if (rank == 0) {
    printf("Total count = %ld\n", global_count);
    printf("Time with %d processes = %f seconds\n", size, end - start);
}

free(local_arr);
MPI_Finalize();
```

```
    return 0;
}
```

Compile and run:

```
mpicc -O2 -o count_mpi count_mpi.c
mpirun -np 4 ./count_mpi
```

Expected output: The total count should be approximately `N * 0.01` (about 1,000,000 for `N=100M`). The time will vary with the number of processes.

Experiments:

1. Run with 1, 2, 4, and 8 processes (if available). Record the execution times in a table.
2. Compute speedup = T(1) / T(N) for each N.
3. Plot speedup vs. number of processes. Add a line for ideal speedup.
4. Compare your results with the Pthreads version from Lab 0. Which is faster for the same number of cores? Why?

Questions to think about:

- Why does each process generate its own random data instead of having process 0 generate everything and distribute it?
- What would happen if you forgot to call `MPI_Reduce` and instead tried to send all partial counts to process 0 using individual `MPI_Send` calls? Which would be more efficient?
- How does the performance scale as you increase the number of processes? Is it linear? Why or why not?

Exercise 4: Parallel Estimation of π (Optional but Highly Recommended)

Goal: Use MPI to estimate π by numerical integration. This exercise reinforces the same patterns as the counting problem but with floating-point arithmetic.

The value of π can be approximated by:

$$\pi = \int_0^1 \frac{4}{1+x^2} dx \approx \frac{1}{n} \sum_{i=0}^{n-1} \frac{4}{1+x_i^2}$$

where x_i is the midpoint of each interval.

Steps:

1. Create `pi_mpi.c`.
2. Define a function `f(x) = 4.0 / (1.0 + x * x)`.
3. Set `N` to a large number (e.g., 1,000,000,000).
4. Follow the same pattern as Exercise 3:
    - Broadcast `N` from rank 0 to all processes.
    - Compute each process's range of intervals.
    - Each process computes its partial sum.

- Use `MPI_Reduce` to sum all partial sums into `global_sum` on rank 0.
- Multiply the global sum by the interval width `h = 1.0/N`.
- Print the result and timing.

Code:

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

#define N 1000000000  // 1 billion intervals

double f(double x) {
    return 4.0 / (1.0 + x * x);
}

int main(int argc, char** argv) {
    int rank, size;
    double local_sum = 0.0, global_sum = 0.0;
    double start, end;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

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

    double h = 1.0 / n;
    int intervals_per_proc = n / size;
    int remainder = n % size;
    int start_index, end_index;

    if (rank < remainder) {
        start_index = rank * (intervals_per_proc + 1);
        end_index = start_index + intervals_per_proc + 1;
    } else {
        start_index = rank * intervals_per_proc + remainder;
        end_index = start_index + intervals_per_proc;
    }

    start = MPI_Wtime();
    for (int i = start_index; i < end_index; i++) {
        double x = (i + 0.5) * h;
        local_sum += f(x);
    }
    local_sum *= h;

    MPI_Reduce(&local_sum, &global_sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    end = MPI_Wtime();

    if (rank == 0) {
        printf("Pi = %.16f\n", global_sum);
        printf("Time with %d processes = %f seconds\n", size, end - start);
    }
```

```
    MPI_Finalize();
    return 0;
}
```

Compile and run:

```
mpicc -O2 -o pi_mpi pi_mpi.c -lm
mpirun -np 4 ./pi_mpi
```

Compare with counting program: Which one scales better? Why?

Exercise 5: Manual Reduction (Optional Challenge)

Goal: Implement the reduction yourself using `MPI_Send` and `MPI_Recv` in a tree pattern, and compare its performance with `MPI_Reduce`.

Hint: Design a binary tree where:

- Processes with even ranks send their partial sum to a parent.
- The parent adds the received sum to its own and forwards the result up the tree.
- At the end, rank 0 has the total sum.

This exercise will deepen your understanding of point-to-point communication and show you why collective operations are so valuable.

---

7. Comparing Pthreads and MPI

Now that you have both versions of the counting program, create a comparison table:

Feature	Pthreads	MPI
Memory model	Shared	Distributed
Data placement	All threads access same array	Each process has own chunk
Synchronisation	Mutexes, condition variables	Message passing
Communication	Implicit (via shared memory)	Explicit (send/receive)
Scalability	Limited to one node	Can scale to thousands of nodes
Overhead	Low for small core counts	Higher due to message passing
Ease of programming	Simpler for shared data	More complex, but more control

Write a paragraph in your report discussing:

- Which version is faster on a single multicore machine for the same number of cores?
- What are the advantages of MPI if you had access to a cluster of 100 machines?
- What limits the scalability of the Pthreads version?

---

8. Performance Analysis

Run the MPI counting program with 1, 2, 4, and 8 processes. Record your results in a table:

Processes	Time (s)	Speedup	Efficiency (Speedup / Processes)
1	T1	1.0	1.0
2	T2	T1/T2	(T1/T2)/2
4	T4	T1/T4	(T1/T4)/4
8	T8	T1/T8	(T1/T8)/8

Plot speedup vs. number of processes. Add a line for ideal speedup (speedup = processes).

Answer these questions:

1. Is the speedup linear? If not, why?
2. What is the efficiency with 8 processes? What causes the drop in efficiency?
3. How do these results compare with the Pthreads version from Lab 0? Which achieves higher efficiency? Why?

---

9. Common Pitfalls and Debugging Tips

9.1 Deadlocks

- **Symptom:** Your program hangs and does not complete.
- **Common cause:** Two processes both call `MPI_Send` before calling `MPI_Recv`. For small messages, MPI may buffer them, but for large messages, this will deadlock.
- **Solution:** Ensure a proper order (e.g., even ranks send first, odd ranks receive first) or use `MPI_Sendrecv`.

9.2 Mismatched Tags or Ranks

- If the tag in `MPI_Send` does not match the tag in `MPI_Recv`, the message won't be received.
- If the destination rank in `MPI_Send` is incorrect, the message may go to the wrong process or cause a deadlock.

9.3 Collective Operations Must Be Called by All

- All processes in a communicator **must** call a collective function like `MPI_Bcast` or `MPI_Reduce`. If one process misses the call, the program will hang indefinitely.

9.4 Integer Overflow

- When `N` is very large, ensure that `N * sizeof(int)` does not overflow. Use `long` for counts and indices if necessary.

9.5 Compiling with `-lm`

- If you use math functions like `sqrt` or `pow`, you must link the math library with `-lm`. In the π program, we used `-lm`.

9.6 Forgetting to Initialise MPI

- Every MPI function (except `MPI_Init` itself) must be called after `MPI_Init`. Calling an MPI function before initialisation will cause a crash.

9.7 Using `printf` Excessively

- In the ping-pong program, printing every message slows down the program significantly. For accurate timing, you may want to remove the `printf` statements inside the loop.

---

10. Self-Assessment Questions

Test your understanding by answering these questions:

1. What are the main limitations of shared-memory parallelism that motivate the use of distributed memory?
2. Explain the SPMD model and how it is used in MPI. What does "Single Program, Multiple Data" mean in practice?

3. In the MPI counting program (Exercise 3), why does each process generate its own random data instead of having process 0 generate the whole array and distribute it? What are the trade-offs?
4. What would happen if you forgot to call `MPI_Bcast` for `n` in the π program? Would the program still work? Why or why not?
5. Compare the time taken for a single `MPI_Send`/`MPI_Recv` pair (for a small message) with the time to update a shared variable in Pthreads. Why is there such a large difference?
6. Run the ping-pong program with different message sizes (1 integer, 10 integers, 100 integers, 1000 integers). Plot message size vs. round-trip time. What can you conclude about latency and bandwidth?
7. According to Amdahl's law, if 95% of the counting problem is parallelisable, what is the maximum theoretical speedup with 8 processors? How does your measured speedup compare? Explain the discrepancy.
8. Why might the MPI version show lower efficiency than the Pthreads version for the same number of cores on a single machine? What are the sources of overhead in MPI that are not present in Pthreads?
9. Modify the MPI counting program so that process 0 generates the entire array and uses `MPI_Scatter` to distribute it. Compare the performance with the version where each process generates its own data. Discuss the trade-offs in terms of memory usage, communication overhead, and scalability.
10. Imagine you have access to a cluster with 100 nodes, each with 16 cores. Which programming model (Pthreads or MPI) would you use to utilise all cores? Why? Could you combine them? How?

---

11. Deliverables

Prepare a single PDF report containing:

1. **Screenshots** of the terminal showing successful compilation and runs for:
   - Exercise 1: Hello World with 4 processes
   - Exercise 2: Ping-Pong with 2 processes
   - Exercise 3: MPI Counting with 4 processes
2. **A table** of execution times and speedup for the MPI counting program (Exercise 3) with 1, 2, 4, and 8 processes (if 8 are available; otherwise 1, 2, 4).
3. **A speedup graph** (plot speedup vs. number of processes) for the MPI counting program. Include a line for ideal speedup.
4. **A comparison** (table or paragraph) contrasting the Pthreads and MPI versions of the counting program.
5. **Answers** to the self-assessment questions (Section 10).
6. **A brief reflection** (10–15 sentences) on the challenges and insights you gained from moving from shared-memory to distributed-memory programming. What was the most surprising thing you learned? What was the most difficult part?

---

12. Further Reading and Resources

- MPI Tutorial – Excellent step-by-step tutorials with examples.
- LLNL MPI Tutorial – In-depth coverage from Lawrence Livermore National Laboratory.
- OpenMPI Documentation – Official documentation for OpenMPI.

- Using MPI – 3rd Edition by Gropp, Lusk, and Skjellum – The classic textbook on MPI programming.

---

13. Glossary

Term	Definition
MPI	Message Passing Interface – a standard for distributed memory programming
Rank	A unique identifier for each process in an MPI communicator
Communicator	A group of processes that can communicate with each other
SPMD	Single Program, Multiple Data – the programming model used by MPI
Blocking communication	A communication call that does not return until the operation is complete
Collective communication	Communication involving all processes in a communicator (e.g., broadcast, reduce)
Latency	The time to send a message of zero size (the overhead per message)
Bandwidth	The rate at which data can be sent (bytes per second)

Congratulations! You have now taken your first steps into the world of distributed memory programming. The skills you've learned in this lab are the foundation for programming the largest supercomputers in the world. Keep experimenting, keep asking questions, and enjoy the journey!