CEN310 Parallel Programming

Week-4

MPI Programming

Download

- PDF
- DOC
- SLIDE
- PPTX

Outline (1/4)

1. Introduction to MPI
    - What is MPI?
    - Distributed Memory Model
    - MPI Implementation Types
    - Basic Concepts
    - Environment Setup
2. Point-to-Point Communication
    - Blocking Send/Receive
    - Non-blocking Send/Receive
    - Buffering and Synchronization
    - Communication Modes
    - Error Handling

Outline (2/4)

3. Collective Communication
    - Broadcast Operations
    - Scatter/Gather Operations
    - Reduction Operations
    - All-to-All Communication
    - Barrier Synchronization
4. Data Types and Communication
    - Basic Data Types
    - Derived Data Types
    - Pack/Unpack Operations
    - Type Matching
    - Buffer Management

Outline (3/4)

5. Advanced MPI Features
    - Virtual Topologies
    - One-sided Communication
    - Hybrid Programming (MPI + OpenMP)
    - Process Groups
    - Communicators
6. Performance Optimization
    - Communication Patterns
    - Load Balancing
    - Overlapping Communication
    - Memory Management
    - Profiling Tools

Outline (4/4)

7. Best Practices
    - Code Organization
    - Error Handling
    - Debugging Techniques
    - Portability
    - Common Pitfalls
8. Real-World Applications
    - Scientific Computing
    - Data Processing
    - Distributed Algorithms
    - Cluster Computing

1. Introduction to MPI

What is MPI? (1/4)

Message Passing Interface (MPI):

- Standard for distributed memory parallel programming
- Language independent specification
- Portable and scalable
- Extensive functionality

Key Features:

1. Process-based parallelism
2. Explicit message passing
3. Standardized interface
4. Multiple implementations

What is MPI? (2/4)

Basic MPI Program

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

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

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

    // Get process rank and total size
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    printf("Process %d of %d\n", rank, size);

    // Finalize MPI environment
    MPI_Finalize();
    return 0;
}
```

What is MPI? (3/4)

Compilation and Execution

```
# Compile with MPICH
mpicc program.c -o program

# Compile with OpenMPI
mpic++ program.cpp -o program

# Run with 4 processes
mpirun -np 4 ./program

# Run on specific hosts
mpirun -np 4 --hosts node1,node2 ./program
```

What is MPI? (4/4)

Environment Setup

```c
void mpi_environment_example() {
    int thread_support;

    // Initialize with thread support
    MPI_Init_thread(NULL, NULL,
                    MPI_THREAD_MULTIPLE,
                    &thread_support);

    // Check processor name
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Get version information
    int version, subversion;
    MPI_Get_version(&version, &subversion);

    printf("MPI %d.%d on %s\n",
           version, subversion, processor_name);
}
```

2. Point-to-Point Communication

Blocking Communication (1/4)

```c
void blocking_communication_example() {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    const int TAG = 0;

    if(rank == 0) {
        // Sender
        int data = 42;
        MPI_Send(&data, 1, MPI_INT, 1, TAG,
                 MPI_COMM_WORLD);
        printf("Process 0 sent: %d\n", data);
    }
    else if(rank == 1) {
        // Receiver
        int received;
        MPI_Status status;

        MPI_Recv(&received, 1, MPI_INT, 0, TAG,
                 MPI_COMM_WORLD, &status);
        printf("Process 1 received: %d\n", received);
    }
}
```

Blocking Communication (2/4)

Send Modes

```
void demonstrate_send_modes() {
    int rank, data = 42;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if(rank == 0) {
        // Standard send
        MPI_Send(&data, 1, MPI_INT, 1, 0,
                 MPI_COMM_WORLD);

        // Synchronous send
        MPI_Ssend(&data, 1, MPI_INT, 1, 0,
                  MPI_COMM_WORLD);

        // Buffered send
        int buffer_size = MPI_BSEND_OVERHEAD + sizeof(int);
        char* buffer = new char[buffer_size];
        MPI_Buffer_attach(buffer, buffer_size);

        MPI_Bsend(&data, 1, MPI_INT, 1, 0,
                  MPI_COMM_WORLD);

        MPI_Buffer_detach(&buffer, &buffer_size);
        delete[] buffer;

        // Ready send
        MPI_Rsend(&data, 1, MPI_INT, 1, 0,
                  MPI_COMM_WORLD);
    }
}
```

Blocking Communication (3/4)

Error Handling

```
void error_handling_example() {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Create error handler
    MPI_Errhandler errhandler;
    MPI_Comm_create_errhandler(error_handler_function,
                              &errhandler);

    // Set error handler
    MPI_Comm_set_errhandler(MPI_COMM_WORLD,
                           errhandler);

    // Example operation that might fail
    int* data = nullptr;
    int result = MPI_Send(data, 1, MPI_INT,
                         rank+1, 0, MPI_COMM_WORLD);

    if(result != MPI_SUCCESS) {
        char error_string[MPI_MAX_ERROR_STRING];
        int length;
        MPI_Error_string(result, error_string, &length);
        printf("Error: %s\n", error_string);
    }

    MPI_Errhandler_free(&errhandler);
}
```

Blocking Communication (4/4)

Status Information

```c
void check_message_status() {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if(rank == 0) {
        int data[100];
        MPI_Send(data, 100, MPI_INT, 1, 0,
                MPI_COMM_WORLD);
    }
    else if(rank == 1) {
        MPI_Status status;
        int received[100];

        MPI_Recv(received, 100, MPI_INT, 0, 0,
                 MPI_COMM_WORLD, &status);

        // Check source
        printf("Received from process %d\n",
               status.MPI_SOURCE);

        // Check tag
        printf("With tag %d\n",
               status.MPI_TAG);

        // Get count of received elements
        int count;
        MPI_Get_count(&status, MPI_INT, &count);
        printf("Received %d elements\n", count);
    }
}
```

3. Non-blocking Communication

Non-blocking Operations (1/3)

```
void non_blocking_example() {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    const int SIZE = 1000000;
    std::vector<double> send_buf(SIZE);
    std::vector<double> recv_buf(SIZE);

    MPI_Request send_request, recv_request;
    MPI_Status status;

    // Start non-blocking send and receive
    if(rank == 0) {
        MPI_Isend(send_buf.data(), SIZE, MPI_DOUBLE,
                  1, 0, MPI_COMM_WORLD, &send_request);

        // Do other work while communication progresses
        do_computation();

        // Wait for send to complete
        MPI_Wait(&send_request, &status);
    }
    else if(rank == 1) {
        MPI_Irecv(recv_buf.data(), SIZE, MPI_DOUBLE,
                  0, 0, MPI_COMM_WORLD, &recv_request);

        // Do other work while waiting
        do_computation();

        // Wait for receive to complete
        MPI_Wait(&recv_request, &status);
    }
}
```

Non-blocking Operations (2/3)

Multiple Requests

```
void multiple_requests_example() {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    const int NUM_REQUESTS = 4;
    std::vector<MPI_Request> requests(NUM_REQUESTS);
    std::vector<MPI_Status> statuses(NUM_REQUESTS);

    // Start multiple non-blocking operations
    for(int i = 0; i < NUM_REQUESTS; i++) {
        int next = (rank + 1) % size;
        int prev = (rank - 1 + size) % size;

        MPI_Isend(&data[i], 1, MPI_INT, next, i,
                  MPI_COMM_WORLD, &requests[i*2]);
        MPI_Irecv(&data[i], 1, MPI_INT, prev, i,
                  MPI_COMM_WORLD, &requests[i*2+1]);
    }

    // Wait for all requests to complete
    MPI_Waitall(NUM_REQUESTS, requests.data(),
                statuses.data());
}
```

Non-blocking Operations (3/3)

Testing for Completion

```
void test_completion_example() {
    MPI_Request request;
    MPI_Status status;
    int flag;

    // Start non-blocking operation
    MPI_Isend(&data, 1, MPI_INT, dest, tag,
              MPI_COMM_WORLD, &request);

    // Test if operation is complete
    do {
        MPI_Test(&request, &flag, &status);
        if(!flag) {
            // Do useful work while waiting
            do_other_work();
        }
    } while(!flag);

    // Alternative: Wait with timeout
    double timeout = 1.0; // seconds
    double start = MPI_Wtime();

    while(MPI_Wtime() - start < timeout) {
        MPI_Test(&request, &flag, &status);
        if(flag) break;
        do_other_work();
    }
}
```

4. Collective Communication

Broadcast Operations (1/4)

```
void broadcast_example() {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    const int SIZE = 1000;
    std::vector<double> data(SIZE);

    if(rank == 0) {
        // Root process initializes data
        for(int i = 0; i < SIZE; i++)
            data[i] = i;
    }

    // Broadcast data to all processes
    MPI_Bcast(data.data(), SIZE, MPI_DOUBLE,
              0, MPI_COMM_WORLD);

    printf("Process %d received data[0] = %f\n",
           rank, data[0]);
}
```

Broadcast Operations (2/4)

Scatter Operation

```
void scatter_example() {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    const int ELEMENTS_PER_PROC = 100;
    std::vector<double> send_data;
    std::vector<double> recv_data(ELEMENTS_PER_PROC);

    if(rank == 0) {
        send_data.resize(ELEMENTS_PER_PROC * size);
        for(int i = 0; i < send_data.size(); i++)
            send_data[i] = i;
    }

    // Scatter data to all processes
    MPI_Scatter(send_data.data(), ELEMENTS_PER_PROC,
                MPI_DOUBLE, recv_data.data(),
                ELEMENTS_PER_PROC, MPI_DOUBLE,
                0, MPI_COMM_WORLD);

    // Process local data
    double local_sum = 0;
    for(int i = 0; i < ELEMENTS_PER_PROC; i++)
        local_sum += recv_data[i];

    printf("Process %d local sum: %f\n",
           rank, local_sum);
}
```

Broadcast Operations (3/4)

Gather Operation

```
void gather_example() {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Local data
    double local_value = rank * 2.0;
    std::vector<double> gathered_data;

    if(rank == 0)
        gathered_data.resize(size);

    // Gather data to root process
    MPI_Gather(&local_value, 1, MPI_DOUBLE,
               gathered_data.data(), 1, MPI_DOUBLE,
               0, MPI_COMM_WORLD);

    if(rank == 0) {
        printf("Gathered values: ");
        for(int i = 0; i < size; i++)
            printf("%f ", gathered_data[i]);
        printf("\n");
    }
}
```

Broadcast Operations (4/4)

All-to-All Communication

```
void alltoall_example() {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Send and receive buffers
    std::vector<double> send_buf(size);
    std::vector<double> recv_buf(size);

    // Initialize send buffer
    for(int i = 0; i < size; i++)
        send_buf[i] = rank * size + i;

    // All-to-all communication
    MPI_Alltoall(send_buf.data(), 1, MPI_DOUBLE,
                 recv_buf.data(), 1, MPI_DOUBLE,
                 MPI_COMM_WORLD);

    // Print received data
    printf("Process %d received: ", rank);
    for(int i = 0; i < size; i++)
        printf("%f ", recv_buf[i]);
    printf("\n");
}
```

5. Advanced MPI Features

Virtual Topologies (1/3)

```
void cartesian_topology_example() {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Create 2D cartesian topology
    int dims[2] = {0, 0};
    int periods[2] = {1, 1}; // Periodic boundaries
    int reorder = 1;

    MPI_Dims_create(size, 2, dims);

    MPI_Comm cart_comm;
    MPI_Cart_create(MPI_COMM_WORLD, 2, dims,
                    periods, reorder, &cart_comm);

    // Get coordinates
    int coords[2];
    MPI_Cart_coords(cart_comm, rank, 2, coords);

    printf("Process %d coordinates: (%d,%d)\n",
           rank, coords[0], coords[1]);

    // Find neighbors
    int left, right, up, down;
    MPI_Cart_shift(cart_comm, 0, 1, &left, &right);
    MPI_Cart_shift(cart_comm, 1, 1, &up, &down);

    printf("Process %d neighbors: left=%d right=%d up=%d down=%d\n",
           rank, left, right, up, down);
}
```

Virtual Topologies (2/3)

Graph Topology

```
void graph_topology_example() {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Define graph connectivity
    std::vector<int> index = {2, 4, 6, 8};  // Cumulative degrees
    std::vector<int> edges = {1, 2, 0, 3, 0, 3, 1, 2};

    // Create graph topology
    MPI_Comm graph_comm;
    MPI_Graph_create(MPI_COMM_WORLD, size,
                    index.data(), edges.data(),
                    1, &graph_comm);

    // Get neighbors
    int degree;
    MPI_Graph_neighbors_count(graph_comm, rank, &degree);

    std::vector<int> neighbors(degree);
    MPI_Graph_neighbors(graph_comm, rank, degree,
                       neighbors.data());

    printf("Process %d has %d neighbors: ",
           rank, degree);
    for(int i = 0; i < degree; i++)
        printf("%d ", neighbors[i]);
    printf("\n");
}
```

Virtual Topologies (3/3)

Communication Patterns

```
void communication_pattern_example() {
    int rank, coords[2];
    MPI_Comm cart_comm;

    // Setup cartesian topology
    // ... (as shown before)

    // Implement stencil computation
    const int ITERATIONS = 100;
    std::vector<double> local_data(LOCAL_SIZE);
    std::vector<double> ghost_left(LOCAL_SIZE);
    std::vector<double> ghost_right(LOCAL_SIZE);

    for(int iter = 0; iter < ITERATIONS; iter++) {
        // Exchange ghost cells
        MPI_Sendrecv(local_data.data(), LOCAL_SIZE,
                     MPI_DOUBLE, left, 0,
                     ghost_left.data(), LOCAL_SIZE,
                     MPI_DOUBLE, right, 0,
                     cart_comm, MPI_STATUS_IGNORE);

        MPI_Sendrecv(local_data.data(), LOCAL_SIZE,
                     MPI_DOUBLE, right, 1,
                     ghost_right.data(), LOCAL_SIZE,
                     MPI_DOUBLE, left, 1,
                     cart_comm, MPI_STATUS_IGNORE);

        // Update local data using ghost cells
        update_stencil(local_data, ghost_left, ghost_right);
    }
}
```

6. Performance Optimization

Communication Optimization (1/3)

```cpp
void optimize_communication() {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    const int LARGE_SIZE = 1000000;
    std::vector<double> large_data(LARGE_SIZE);

    // Bad: Many small messages
    for(int i = 0; i < LARGE_SIZE; i++) {
        if(rank == 0)
            MPI_Send(&large_data[i], 1, MPI_DOUBLE,
                     1, 0, MPI_COMM_WORLD);
    }

    // Good: Single large message
    if(rank == 0) {
        MPI_Send(large_data.data(), LARGE_SIZE,
                MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
    }

    // Better: Non-blocking with computation overlap
    MPI_Request request;
    if(rank == 0) {
        MPI_Isend(large_data.data(), LARGE_SIZE,
                  MPI_DOUBLE, 1, 0, MPI_COMM_WORLD,
                  &request);

        // Do other work while communication progresses
        do_computation();

        MPI_Wait(&request, MPI_STATUS_IGNORE);
    }
}
```

Communication Optimization (2/3)

Derived Datatypes

```
void derived_datatype_example() {
    struct Particle {
        double x, y, z;
        double vx, vy, vz;
        int id;
    };

    // Create MPI datatype for Particle
    MPI_Datatype particle_type;
    int blocklengths[] = {3, 3, 1};
    MPI_Aint offsets[3];
    MPI_Datatype types[] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};

    // Calculate offsets
    MPI_Get_address(&particle.x, &offsets[0]);
    MPI_Get_address(&particle.vx, &offsets[1]);
    MPI_Get_address(&particle.id, &offsets[2]);

    // Make relative
    for(int i = 2; i >= 0; i--)
        offsets[i] -= offsets[0];

    // Create and commit type
    MPI_Type_create_struct(3, blocklengths, offsets,
                          types, &particle_type);
    MPI_Type_commit(&particle_type);

    // Use the new type
    std::vector<Particle> particles(100);
    MPI_Send(particles.data(), particles.size(),
             particle_type, dest, tag, MPI_COMM_WORLD);

    MPI_Type_free(&particle_type);
}
```

Communication Optimization (3/3)

Persistent Communication

```
void persistent_communication_example() {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Create persistent request
    MPI_Request request;

    if(rank == 0) {
        MPI_Send_init(buffer, count, MPI_DOUBLE,
                      1, tag, MPI_COMM_WORLD, &request);
    }
    else if(rank == 1) {
        MPI_Recv_init(buffer, count, MPI_DOUBLE,
                      0, tag, MPI_COMM_WORLD, &request);
    }

    // Use in iteration
    for(int iter = 0; iter < NUM_ITERATIONS; iter++) {
        // Start communication
        MPI_Start(&request);

        // Do other work
        do_computation();

        // Wait for completion
        MPI_Wait(&request, MPI_STATUS_IGNORE);

        // Process received data
        process_data();
    }

    // Free persistent request
    MPI_Request_free(&request);
}
```

7. Best Practices

Error Handling and Debugging (1/3)

```
void error_handling_best_practices() {
    // Initialize MPI with thread support
    int provided;
    int required = MPI_THREAD_MULTIPLE;

    int init_result =
        MPI_Init_thread(NULL, NULL, required, &provided);

    if(init_result != MPI_SUCCESS) {
        fprintf(stderr, "Failed to initialize MPI\n");
        exit(1);
    }

    if(provided < required) {
        fprintf(stderr,
                "Insufficient thread support level\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    // Set error handler
    MPI_Comm_set_errhandler(MPI_COMM_WORLD,
                           MPI_ERRORS_RETURN);

    // Check all MPI calls
    int result = MPI_Send(data, count, MPI_INT,
                         dest, tag, MPI_COMM_WORLD);
    if(result != MPI_SUCCESS) {
        char error_string[MPI_MAX_ERROR_STRING];
        int length;
        MPI_Error_string(result, error_string, &length);
        fprintf(stderr, "MPI error: %s\n", error_string);
        MPI_Abort(MPI_COMM_WORLD, result);
    }
}
```

Error Handling and Debugging (2/3)

Debugging Tools

```
void debugging_example() {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Add debug prints
    #ifdef DEBUG
        printf("[%d] Starting computation\n", rank);
    #endif

    // Validate input
    if(input_size <= 0) {
        if(rank == 0) {
            fprintf(stderr, "Invalid input size\n");
        }
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    // Check for buffer overflow
    size_t buffer_size = calculate_buffer_size();
    if(buffer_size > MAX_BUFFER_SIZE) {
        fprintf(stderr,
                "[%d] Buffer overflow detected\n", rank);
        MPI_Abort(MPI_COMM_WORLD, 2);
    }

    // Synchronization point for debugging
    MPI_Barrier(MPI_COMM_WORLD);

    #ifdef DEBUG
        printf("[%d] Passed validation\n", rank);
    #endif
}
```

Error Handling and Debugging (3/3)

Memory Management

```
class MPIBuffer {
private:
    void* buffer;
    int size;

public:
    MPIBuffer(int size) : size(size) {
        buffer = malloc(size);
        if(!buffer) {
            throw std::runtime_error("Memory allocation failed");
        }
    }

    ~MPIBuffer() {
        if(buffer) {
            free(buffer);
        }
    }

    void* get() { return buffer; }
    int get_size() { return size; }

    // Prevent copying
    MPIBuffer(const MPIBuffer&) = delete;
    MPIBuffer& operator=(const MPIBuffer&) = delete;
};

void safe_memory_usage() {
    try {
        MPIBuffer send_buffer(1024);
        MPIBuffer recv_buffer(1024);

        MPI_Send(send_buffer.get(), send_buffer.get_size(),
                 MPI_BYTE, dest, tag, MPI_COMM_WORLD);
    }
    catch(const std::exception& e) {
        fprintf(stderr, "Error: %s\n", e.what());
        MPI_Abort(MPI_
```