Message Passing Interface (MPI)
Study-Ready Notes

Compiled by Andrew Photinakis

Contents

1 Introduction to MPI

1.1 What is MPI?

- Message Passing Interface (MPI) - a specification, not a library
- Message-passing parallel programming model: data moved from address space of one process to another through cooperative operations
- Well-known implementations: Open MPI and MVAPICH2
- All parallelism is explicit: programmer responsible for identifying parallelism and implementing parallel algorithms

[Summary: MPI is a standardized message-passing specification for parallel programming where processes communicate by explicitly sending and receiving messages between their separate address spaces.]

1.2 Parallel Computing Systems

- Distributed Memory: Computers in a network
- Distributed Shared Memory: Computers in a cluster
- Multiprocessor systems
- Multicore systems
- In-cloud and Edge computing

1.3 MPI Program Structure

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

int main(int argc, char *argv[]) {
    int numtasks, rank, dest, source, rc, count, tag=1;
    char inmsg, outmsg='x';
    MPI_Status Stat;

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

    if (rank == 0) {
        dest = 1; source = 1;
        rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
        rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
    }
    else if (rank == 1) {
        dest = 0; source = 0;
        rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
        rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    }

    MPI_Get_count(&Stat, MPI_CHAR, &count);
    printf("Task %d: Received %d char(s) from task %d with tag %d\n",
           rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);

    MPI_Finalize();
}
```

[Summary: Basic MPI programs initialize with MPI_Init, get process count and rank, perform communication operations, and finalize with MPI_Finalize. Processes are identified by unique ranks within communicators.]

2 MPI Communication Fundamentals

2.1 Basic Send and Receive Operations

- **MPI_Send(void *sendbuf, int nelems, int dest, int tag, MPI_Comm comm)**
- **MPI_Recv(void *recvbuf, int nelems, int source, int tag, MPI_Comm comm, MPI_Status *status)**
- Parameters:
  - sendbuf/recvbuf: pointers to data buffers
  - nelems: number of data elements
  - dest/source: process identifiers
  - tag: message identifier (non-negative integer)
  - comm: communicator (usually MPI_COMM_WORLD)

2.2 Communicators and Groups

- **Communicators**: Define which processes can communicate
- **MPI_COMM_WORLD**: Predefined communicator including all processes
- **Rank**: Unique integer identifier for each process within a communicator
- Ranks are contiguous starting from 0

- Used to specify source/destination of messages and control program flow

[Summary: MPI uses communicators to define communication groups, with ranks identifying individual processes for message routing and conditional execution.]

3 Point-to-Point Communication

3.1 Communication Types

- Synchronous send
- Blocking send/blocking receive
- Non-blocking send/non-blocking receive
- Buffered send
- Combined send/receive
- "Ready" send
- Any send type can pair with any receive type

3.2 System Buffers

- Buffer space reserved for data in transit
- Managed entirely by MPI library (opaque to programmer)
- Finite resource that can be exhausted
- Can exist on sending side, receiving side, or both
- Allows send-receive operations to be asynchronous

[Concept Map: Point-to-Point Communication → Blocking/Non-blocking → Synchronous/Asynchronou
→ Buffered/Non-buffered → System Buffer Management]

3.3 Blocking vs Non-blocking Operations

3.3.1 Blocking Operations

- **Blocking Send**: Returns only when safe to modify application buffer
- **Blocking Receive**: Returns only when data has arrived and is ready
- Can be synchronous (handshaking) or asynchronous (using system buffers)

3.3.2 Non-blocking Operations

- Return immediately without waiting for communication events
- Simply ”request” MPI to perform operation when able
- Unsafe to modify application buffer until operation completes
- Use ”wait” routines (MPI_Wait, MPI_Test) to check completion
- Used to overlap computation with communication

[Mnemonic: Blocking = Wait for completion, Non-blocking = Fire and (maybe) forget until checked]

4 Collective Communication

4.1 Types of Collective Operations

- **Synchronization**: Processes wait until all reach synchronization point (MPI_Barrier, MPI_Waitall)
- **Data Movement**: Broadcast, scatter/gather, all to all
- **Collective Computation (Reductions)**: One member collects data and performs operations (min, max, add, multiply)

4.2 Important Collective Routines

4.2.1 Broadcast (MPI_Bcast)

- Broadcasts message from one task to all others in communicator
- `MPI_Bcast(&buffer, count, datatype, root, comm)`

4.2.2 Scatter (MPI_Scatter)

- Sends chunks of data from one task to all tasks
- `MPI_Scatter (sendbuf, sendcnt, sendtype, recvbuf, recvcnt, recvtype, root, comm)`

4.2.3 Gather (MPI_Gather)

- Gathers data from all tasks to a single task
- Inverse of scatter operation

4.2.4 Reduce (MPI_Reduce)

- Performs reduction (sum, max, min, etc.) across all tasks
- Result stored in one task
- `MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm)`

4.2.5 Allreduce (MPI_Allreduce)

- Performs reduction and stores result across all tasks

[Summary: Collective operations involve all processes in a communicator for synchronization, data movement, or collective computation, with routines like broadcast, scatter/gather, and reductions.]

5 Derived Data Types

5.1 Continuous Derived Data Type

- Represents contiguous blocks of data
- `MPI_Type_contiguous(count, oldtype, &newtype)`
- Example: Representing a row of a 2D array

5.2 Vector Derived Data Type

- Represents regularly spaced data blocks
- `MPI_Type_vector(count, blocklength, stride, oldtype, &newtype)`
- Example: Representing a column of a 2D array

5.3 Indexed Derived Data Type

- Represents irregular data patterns
- `MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements, oldtype, &newtype)`

5.4 Struct Derived Data Type

- Represents heterogeneous data structures
- `MPI_Type_struct(count, array_of_blocklengths, array_of_offsets, array_of_types, &newtype)`
- Example: Representing a particle with position, velocity, and type information

[Mnemonic: Continuous = Contiguous blocks, Vector = Regular spacing, Indexed = Irregular patterns, Struct = Mixed data types]

6 Groups and Communicators

6.1 Definitions

- **Group**: Ordered set of processes with unique integer ranks (0 to N-1)
- **Communicator**: Encompasses a group of processes that can communicate
- **MPI_COMM_WORLD**: Default communicator containing all processes
- Groups and communicators are dynamic - can be created/destroyed during execution
- Processes can be in multiple groups/communicators with different ranks

6.2 Group Management Operations

1. Extract global group: `MPI_Comm_group(MPI_COMM_WORLD, &group)`
2. Create new group: `MPI_Group_incl(group, n, ranks, &newgroup)`
3. Create new communicator: `MPI_Comm_create(comm, group, &newcomm)`
4. Get new rank: `MPI_Comm_rank(newcomm, &rank)`
5. Free resources: `MPI_Comm_free(), MPI_Group_free()`

6.3 Communicator Splitting

- `MPI_Comm_split(comm, color, key, &newcomm)`
- Groups processes by color, sorts by key within each color group
- Useful for organizing processes by function or data partitioning

[Summary: Groups define process sets, communicators enable communication within those sets, and MPI provides operations to dynamically create and manage these entities for flexible parallel programming.]

7 Virtual Topologies

7.1 Concept and Purpose

- Virtual topology: Mapping/ordering of MPI processes into geometric shapes
- Types: Cartesian (grid) and Graph topologies

- Virtual - no relation to physical machine structure required
- Must be programmed by application developer

7.2 Benefits

- **Convenience**: Match communication patterns to application needs
- **Efficiency**: Potential optimization for specific hardware architectures
- Example: Cartesian topology for grid-based computations with nearest-neighbor communication

7.3 Cartesian Topology Example

- Processes arranged in 2D grid with row-major ordering
- Each process identified by (row, column) coordinates
- Supports operations like `MPI_Cart_sub()` for creating sub-grids

[Concept Map: Virtual Topologies → Cartesian/Graph Types → Process Mapping → Communication Patterns → Potential Hardware Optimization]

8 MPI Programming Examples

8.1 Monte Carlo Pi Calculation

8.1.1 Mathematical Basis

- Circle area: πr^2, Square area: $4r^2$
- Ratio: $\frac{\pi r^2}{4r^2} = \frac{\pi}{4}$
- Random points: $M \approx \frac{N\pi}{4}$
- Approximation: $\pi \approx \frac{4M}{N}$

8.1.2 Parallel Implementation Strategy

- Divide total points among available tasks
- Each task calculates points in circle for its subset
- Master task collects results using send/receive operations
- No data dependencies between tasks

8.2 Nearest Neighbor Exchange

```
// Non-blocking ring communication
MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);
MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

// Do work while communications progress
MPI_Waitall(4, reqs, stats);
```

[Summary: MPI enables various parallel algorithms through point-to-point and collective communication, with examples including Monte Carlo simulations and nearest-neighbor exchanges in ring topologies.]

9 MPI Advantages and Characteristics

9.1 Key Features

- **Practical**: Easy to implement on any system
- **Portable**: Works on networks, clusters, multiprocessors
- **Efficient**: Programmer has more control over communication
- **Flexible**: Easy to scale problem and processor size
- **Explicit Parallelism**: Burden on programmer to identify and manage parallelism

10 Study Aids

Key Concepts to Master

- Difference between blocking and non-blocking operations
- Purpose and usage of communicators and groups
- Collective communication operations and when to use them
- Derived data types and their applications
- Virtual topologies and their benefits
- MPI program structure and initialization/finalization

Common MPI Functions Reference

- Environment: MPI_Init, MPI_Finalize, MPI_Comm_size, MPI_Comm_rank
- Point-to-point: MPI_Send, MPI_Recv, MPI_Isend, MPI_Irecv, MPI_Wait
- Collective: MPI_Bcast, MPI_Scatter, MPI_Gather, MPI_Reduce, MPI_Barrier
- Groups/Communicators: MPI_Comm_split, MPI_Comm_create, MPI_Group_incl

[Mnemonic: MPI Basics - Init, Size, Rank, Send, Recv, Finalize. Collective Ops - Bcast, Scatter, Gather, Reduce. Advanced - Groups, Topologies, Derived Types]