SwiftMPI API Reference
Function Reference with Usage Examples

SwiftMPI Documentation

2025

Contents

1 Initialization and Finalization

1.1 SwiftMPI.initialize()

Initialize MPI environment without command line arguments.

Usage:

```swift
import SwiftMPI

do {
    try SwiftMPI.initialize()
    // MPI is now initialized
} catch {
    print("MPI␣initialization␣failed:␣\(error)")
}
```

1.2 SwiftMPI.initialize(argc:argv:)

Initialize MPI environment with command line arguments.

Usage:

```swift
import SwiftMPI

let argc = CommandLine.argc
let argv = CommandLine.unsafeArgv

do {
    try SwiftMPI.initialize(argc: argc, argv: argv)
    // MPI is now initialized
} catch {
    print("MPI␣initialization␣failed:␣\(error)")
}
```

1.3 SwiftMPI.finalize()

Finalize MPI environment and clean up resources.

Usage:

```swift
import SwiftMPI

do {
    try SwiftMPI.finalize()
    // MPI resources cleaned up
} catch {
    print("MPI␣finalization␣failed:␣\(error)")
}
```

1.4 SwiftMPI.wtime()

Get wall clock time in seconds since arbitrary time.

Usage:

```swift
import SwiftMPI

let startTime = SwiftMPI.wtime()
// ... perform computation ...
let endTime = SwiftMPI.wtime()
let elapsed = endTime - startTime
print("Computation␣took␣\(elapsed)␣seconds")
```

1.5 SwiftMPI.wtick()

Get resolution of MPI_Wtime in seconds.

Usage:

```
import SwiftMPI

let resolution = SwiftMPI.wtick()
print("Time␣resolution:␣\(resolution)␣seconds")
```

1.6 SwiftMPI.world

Get world communicator containing all processes.

Usage:

```
import SwiftMPI

let comm = SwiftMPI.world
let rank = comm.rank()
let size = comm.size()
print("Process␣\(rank)␣of␣\(size)")
```

1.7 SwiftMPI.abort(comm:errorCode:)

Abort all MPI processes with specified error code.

Usage:

```
import SwiftMPI

if someErrorCondition {
    SwiftMPI.abort(comm: SwiftMPI.world, errorCode: 1)
    // Process exits with error code 1
}
```

2 Communicator Operations

2.1 Communicator.rank()

Get rank of current process in this communicator.

Usage:

```
import SwiftMPI

let comm = SwiftMPI.world
let myRank = comm.rank()
print("My␣rank␣is␣\(myRank)")
```

2.2 Communicator.size()

Get number of processes in this communicator.

Usage:

```
import SwiftMPI

let comm = SwiftMPI.world
let totalProcesses = comm.size()
print("Total␣processes:␣\(totalProcesses)")
```

2.3 Communicator.duplicate()

Duplicate this communicator creating new independent communicator.

Usage:

```
import SwiftMPI

let comm = SwiftMPI.world
do {
    let newComm = try comm.duplicate()
    // Use newComm independently
} catch {
    print("Failed␣to␣duplicate␣communicator:␣\(error)")
}
```

2.4 Communicator.free()

Free this communicator and release associated resources.

Usage:

```
import SwiftMPI

let comm = SwiftMPI.world
do {
    try comm.free()
    // Communicator resources released
} catch {
    print("Failed␣to␣free␣communicator:␣\(error)")
}
```

3 Point-to-Point Communication

3.1 Communicator.send(_:count:datatype:dest:tag:)

Blocking send operation sending data to destination process.

Usage:

```
import SwiftMPI

let comm = SwiftMPI.world
let rank = comm.rank()
let data: [Int32] = [1, 2, 3, 4, 5]

if rank == 0 {
    do {
        try data.withUnsafeBufferPointer { buffer in
            try comm.send(buffer, count: data.count,
                         datatype: .int, dest: 1, tag: 0)
        }
        print("Sent␣data␣to␣process␣1")
    } catch {
        print("Send␣failed:␣\(error)")
    }
}
```

3.2 Communicator.receive(_:count:datatype:source:tag:)

Blocking receive operation receiving data from source process.

Usage:

```
import SwiftMPI

let comm = SwiftMPI.world
let rank = comm.rank()

if rank == 1 {
    var buffer = [Int32](repeating: 0, count: 5)
    do {
        let status = try buffer.withUnsafeMutableBufferPointer { buf in
            try comm.receive(buf, count: 5, datatype: .int,
                           source: 0, tag: 0)
        }
        print("Received␣\(status.elementCount)␣elements␣from␣process␣
           \(status.source)")
        print("Data:␣\(buffer)")
    } catch {
        print("Receive␣failed:␣\(error)")
    }
}
```

3.3 Communicator.send(_:to:tag:) - Convenience for Int32

Send array of integers to destination process.

Usage:

```
import SwiftMPI

let comm = SwiftMPI.world
let data: [Int32] = [10, 20, 30]

do {
    try comm.send(data, to: 1, tag: 0)
    print("Sent␣integer␣array")
} catch {
    print("Send␣failed:␣\(error)")
}
```

3.4 Communicator.receive(count:from:tag:) - Convenience for Int32

Receive array of integers from source process.

Usage:

```swift
import SwiftMPI

let comm = SwiftMPI.world

do {
    let received = try comm.receive(count: 3, from: 0, tag: 0)
    print("Received:␣\(received)")
} catch {
    print("Receive␣failed:␣\(error)")
}
```

3.5 Communicator.send(_:to:tag:) - Convenience for Double

Send array of doubles to destination process.

Usage:

```swift
import SwiftMPI

let comm = SwiftMPI.world
let data: [Double] = [3.14, 2.71, 1.41]

do {
    try comm.send(data, to: 1, tag: 1)
    print("Sent␣double␣array")
} catch {
    print("Send␣failed:␣\(error)")
}
```

3.6 Communicator.receiveDoubles(count:from:tag:)

Receive array of doubles from source process.

Usage:

```swift
import SwiftMPI

let comm = SwiftMPI.world

do {
    let received = try comm.receiveDoubles(count: 3, from: 0, tag: 1)
    print("Received␣doubles:␣\(received)")
} catch {
    print("Receive␣failed:␣\(error)")
}
```

4 Non-blocking Communication

4.1 Communicator.isend(_:count:datatype:dest:tag:)

Non-blocking send operation initiating asynchronous send.

Usage:

```swift
import SwiftMPI

let comm = SwiftMPI.world
let data: [Int32] = [1, 2, 3, 4, 5]
```

```

do {
    let request = try data.withUnsafeBufferPointer { buffer in
        try comm.isend(buffer, count: data.count,
                       datatype: .int, dest: 1, tag: 0)
    }
    // Continue with other work...
    let status = try request.wait()
    print("Send␣completed")
} catch {
    print("Isend␣failed:␣\(error)")
}
```

4.2 Communicator.ireceive(_:count:datatype:source:tag:)

Non-blocking receive operation initiating asynchronous receive.

Usage:

```
import SwiftMPI

let comm = SwiftMPI.world
var buffer = [Int32](repeating: 0, count: 5)

do {
    let request = try buffer.withUnsafeMutableBufferPointer { buf in
        try comm.ireceive(buf, count: 5, datatype: .int,
                          source: 0, tag: 0)
    }
    // Continue with other work...
    let status = try request.wait()
    print("Receive␣completed:␣\(buffer)")
} catch {
    print("Ireceive␣failed:␣\(error)")
}
```

4.3 Request.wait()

Wait for this request to complete and return status.

Usage:

```
import SwiftMPI

let comm = SwiftMPI.world
let data: [Int32] = [1, 2, 3]

do {
    let request = try comm.send(data, to: 1, tag: 0)
    // Do other work...
    let status = try request.wait()
    print("Request␣completed")
} catch {
    print("Wait␣failed:␣\(error)")
}
```

4.4 Request.test()

Test if this request has completed without blocking.

Usage:

```swift
import SwiftMPI

let comm = SwiftMPI.world
let data: [Int32] = [1, 2, 3]

do {
    let request = try comm.send(data, to: 1, tag: 0)

    while true {
        let (completed, status) = try request.test()
        if completed {
            print("Request␣completed")
            break
        }
        // Do other work while waiting...
        usleep(1000)
    }
} catch {
    print("Test␣failed:␣\(error)")
}
```

4.5 waitAll(_:)

Wait for multiple requests to complete.

Usage:

```swift
import SwiftMPI

let comm = SwiftMPI.world
let rank = comm.rank()

do {
    var requests: [Request] = []

    // Initiate multiple sends
    for dest in 0..<comm.size() {
        if dest != rank {
            let data: [Int32] = [Int32(rank), Int32(dest)]
            let req = try comm.send(data, to: dest, tag: 0)
            requests.append(req)
        }
    }

    // Wait for all to complete
    let statuses = try waitAll(requests)
    print("All␣sends␣completed")
} catch {
    print("WaitAll␣failed:␣\(error)")
}
```

4.6 testAll(_:)

Test if all requests in array have completed without blocking.

Usage:

```swift
import SwiftMPI

let comm = SwiftMPI.world

do {
    var requests: [Request] = []
    // ... create requests ...

    while true {
        let (allCompleted, statuses) = try testAll(requests)
        if allCompleted {
            print("All requests completed")
            break
        }
        // Continue working...
        usleep(1000)
    }
} catch {
    print("TestAll failed: \(error)")
}
```

4.7 waitAny(_:)

Wait for any one request in array to complete.

Usage:

```swift
import SwiftMPI

let comm = SwiftMPI.world

do {
    var requests: [Request] = []
    // ... create multiple requests ...

    let (index, status) = try waitAny(requests)
    print("Request \(index) completed first")
} catch {
    print("WaitAny failed: \(error)")
}
```

5 Collective Operations

5.1 Communicator.barrier()

Barrier synchronization - all processes wait until all arrive.

Usage:

```swift
import SwiftMPI

let comm = SwiftMPI.world
let rank = comm.rank()

print("Process \(rank) before barrier")
do {
    try comm.barrier()
    print("Process␣\(rank)␣after␣barrier")
} catch {
    print("Barrier␣failed:␣\(error)")
}
```

5.2 Communicator.broadcast(_:count:datatype:root:)

Broadcast operation - root sends data to all processes.

Usage:

```
import SwiftMPI

let comm = SwiftMPI.world
let rank = comm.rank()
var data: [Int32] = [0, 0, 0]

if rank == 0 {
    data = [10, 20, 30]
}

do {
    try data.withUnsafeMutableBufferPointer { buffer in
        try comm.broadcast(buffer, count: 3,
                           datatype: .int, root: 0)
    }
    print("Process␣\(rank)␣received:␣\(data)")
} catch {
    print("Broadcast␣failed:␣\(error)")
}
```

5.3 Communicator.reduce(sendBuffer:recvBuffer:count:datatype:op:root:)

Reduce operation - combine values from all processes to root.

Usage:

```
import SwiftMPI

let comm = SwiftMPI.world
let rank = comm.rank()
let sendValue: [Int32] = [Int32(rank + 1)]
var recvValue: [Int32] = [0]

do {
    try sendValue.withUnsafeBufferPointer { sendBuf in
        try recvValue.withUnsafeMutableBufferPointer { recvBuf in
            try comm.reduce(sendBuffer: sendBuf,
                            recvBuffer: recvBuf,
                            count: 1,
                            datatype: .int,
                            op: .sum,
                            root: 0)
        }
    }

    if rank == 0 {
        print("Sum␣of␣all␣ranks:␣\(recvValue[0])")
    }
} catch {
    print("Reduce␣failed:␣\(error)")
}
```

5.4 Communicator.allReduce(sendBuffer:recvBuffer:count:datatype:op:)

Allreduce operation - reduce and broadcast result to all processes.

Usage:

```
import SwiftMPI

let comm = SwiftMPI.world
let rank = comm.rank()
let sendValue: [Double] = [Double(rank) * 1.5]
var recvValue: [Double] = [0.0]

do {
    try sendValue.withUnsafeBufferPointer { sendBuf in
        try recvValue.withUnsafeMutableBufferPointer { recvBuf in
            try comm.allReduce(sendBuffer: sendBuf,
                             recvBuffer: recvBuf,
                             count: 1,
                             datatype: .double,
                             op: .sum)
        }
    }
    print("Process␣\(rank):␣sum␣=␣\(recvValue[0])")
} catch {
    print("AllReduce␣failed:␣\(error)")
}
```

5.5 Communicator.gather(sendBuffer:sendCount:sendType:recvBuffer:recvCount:recvTy

Gather operation - collect data from all processes to root.

Usage:

```
import SwiftMPI

let comm = SwiftMPI.world
let rank = comm.rank()
let sendData: [Int32] = [Int32(rank), Int32(rank * 2)]
var recvData = [Int32](repeating: 0, count: comm.size() * 2)

do {
    try sendData.withUnsafeBufferPointer { sendBuf in
        try recvData.withUnsafeMutableBufferPointer { recvBuf in
            try comm.gather(sendBuffer: sendBuf,
                          sendCount: 2,
                          sendType: .int,
                          recvBuffer: recvBuf,
                          recvCount: 2,
                          recvType: .int,
                          root: 0)
        }
    }

    if rank == 0 {
        print("Gathered␣data:␣\(recvData)")
    }
} catch {
    print("Gather␣failed:␣\(error)")
}
```

5.6 Communicator.scatter(sendBuffer:sendCount:sendType:recvBuffer:recvCount:recvT

Scatter operation - distribute data from root to all processes.

Usage:

```
import SwiftMPI

let comm = SwiftMPI.world
let rank = comm.rank()
var sendData: [Int32] = []
var recvData = [Int32](repeating: 0, count: 2)

if rank == 0 {
    sendData = [0, 0, 1, 2, 2, 4, 3, 6] // Data for 4 processes
}

do {
    try sendData.withUnsafeBufferPointer { sendBuf in
        try recvData.withUnsafeMutableBufferPointer { recvBuf in
            try comm.scatter(sendBuffer: sendBuf,
                           sendCount: 2,
                           sendType: .int,
                           recvBuffer: recvBuf,
                           recvCount: 2,
                           recvType: .int,
                           root: 0)
        }
    }
    print("Process␣\(rank)␣received:␣\(recvData)")
} catch {
    print("Scatter␣failed:␣\(error)")
}
```

5.7 Communicator.allGather(sendBuffer:sendCount:sendType:recvBuffer:recvCount:rec

Allgather operation - gather data from all processes to all processes.

Usage:

```
import SwiftMPI

let comm = SwiftMPI.world
let rank = comm.rank()
let sendData: [Int32] = [Int32(rank)]
var recvData = [Int32](repeating: 0, count: comm.size())

do {
    try sendData.withUnsafeBufferPointer { sendBuf in
        try recvData.withUnsafeMutableBufferPointer { recvBuf in
            try comm.allGather(sendBuffer: sendBuf,
                             sendCount: 1,
                             sendType: .int,
                             recvBuffer: recvBuf,
                             recvCount: 1,
                             recvType: .int)
        }
    }
    print("Process␣\(rank)␣has␣all␣data:␣\(recvData)")
} catch {
    print("AllGather␣failed:␣\(error)")
}
```

5.8 Communicator.allToAll(sendBuffer:sendCount:sendType:recvBuffer:recvCount:recv

Alltoall operation - each process sends distinct data to each process.

Usage:

```
import SwiftMPI

let comm = SwiftMPI.world
let rank = comm.rank()
let size = comm.size()
let sendData = (0..<size).map { Int32(rank * size + $0) }
var recvData = [Int32](repeating: 0, count: size)

do {
    try sendData.withUnsafeBufferPointer { sendBuf in
        try recvData.withUnsafeMutableBufferPointer { recvBuf in
            try comm.allToAll(sendBuffer: sendBuf,
                            sendCount: 1,
                            sendType: .int,
                            recvBuffer: recvBuf,
                            recvCount: 1,
                            recvType: .int)
        }
    }
    print("Process␣\(rank)␣received:␣\(recvData)")
} catch {
    print("AllToAll␣failed:␣\(error)")
}
```

5.9 Communicator.scan(sendBuffer:recvBuffer:count:datatype:op:)

Scan operation - inclusive prefix reduction across all processes.

Usage:

```
import SwiftMPI

let comm = SwiftMPI.world
let rank = comm.rank()
let sendValue: [Int32] = [Int32(rank + 1)]
var recvValue: [Int32] = [0]

do {
    try sendValue.withUnsafeBufferPointer { sendBuf in
        try recvValue.withUnsafeMutableBufferPointer { recvBuf in
            try comm.scan(sendBuffer: sendBuf,
                         recvBuffer: recvBuf,
                         count: 1,
                          datatype: .int,
                          op: .sum)
        }
    }
    print("Process␣\(rank):␣prefix␣sum␣=␣\(recvValue[0])")
} catch {
    print("Scan␣failed:␣\(error)")
}
```

5.10 Communicator.exScan(sendBuffer:recvBuffer:count:datatype:op:)

Exscan operation - exclusive prefix reduction across all processes.

Usage:

```
import SwiftMPI

let comm = SwiftMPI.world
let rank = comm.rank()
let sendValue: [Int32] = [Int32(rank + 1)]
var recvValue: [Int32] = [0]

do {
    try sendValue.withUnsafeBufferPointer { sendBuf in
        try recvValue.withUnsafeMutableBufferPointer { recvBuf in
            try comm.exScan(sendBuffer: sendBuf,
                            recvBuffer: recvBuf,
                            count: 1,
                            datatype: .int,
                            op: .sum)
        }
    }
    print("Process␣\(rank):␣exclusive␣prefix␣sum␣=␣\(recvValue[0])")
} catch {
    print("ExScan␣failed:␣\(error)")
}
```

5.11 Communicator.gatherV(sendBuffer:sendCount:sendType:recvBuffer:recvCounts:dis

Gatherv operation - gather variable amounts of data to root process.

Usage:

```
import SwiftMPI

let comm = SwiftMPI.world
let rank = comm.rank()
let sendCount = Int32(rank + 1)
let sendData = (0..<Int(sendCount)).map { Int32($0) }
var recvData = [Int32](repeating: 0, count: 10)
let recvCounts: [Int32] = [1, 2, 3, 4] // For 4 processes
let displacements: [Int32] = [0, 1, 3, 6]

do {
    try sendData.withUnsafeBufferPointer { sendBuf in
        try recvData.withUnsafeMutableBufferPointer { recvBuf in
            try comm.gatherV(sendBuffer: sendBuf,
                            sendCount: sendCount,
                            sendType: .int,
                           recvBuffer: recvBuf,
                           recvCounts: recvCounts,
                           displacements: displacements,
                           recvType: .int,
                           root: 0)
        }
    }

    if rank == 0 {
        print("Gathered␣variable␣data:␣\(recvData)")
    }
} catch {
    print("GatherV␣failed:␣\(error)")
}
```

5.12 Communicator.scatterV(sendBuffer:sendCounts:displacements:sendType:recvBuffe

Scatterv operation - scatter variable amounts of data from root process.

Usage:

```
import SwiftMPI

let comm = SwiftMPI.world
let rank = comm.rank()
var sendData: [Int32] = []
var recvData = [Int32](repeating: 0, count: rank + 1)
let sendCounts: [Int32] = [1, 2, 3, 4]
let displacements: [Int32] = [0, 1, 3, 6]

if rank == 0 {
    sendData = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
}

do {
    try sendData.withUnsafeBufferPointer { sendBuf in
        try recvData.withUnsafeMutableBufferPointer { recvBuf in
            try comm.scatterV(sendBuffer: sendBuf,
                            sendCounts: sendCounts,
                            displacements: displacements,
                            sendType: .int,
                            recvBuffer: recvBuf,
                            recvCount: Int32(rank + 1),
                            recvType: .int,
                            root: 0)
        }
    }
    print("Process␣\(rank)␣received:␣\(recvData)")
} catch {
    print("ScatterV␣failed:␣\(error)")
}
```

5.13 Communicator.allGatherV(sendBuffer:sendCount:sendType:recvBuffer:recvCounts

Allgatherv operation - gather variable amounts of data to all processes.

Usage:

```
import SwiftMPI
```

```

let comm = SwiftMPI.world
let rank = comm.rank()
let sendCount = Int32(rank + 1)
let sendData = (0..<Int(sendCount)).map { Int32($0) }
var recvData = [Int32](repeating: 0, count: 10)
let recvCounts: [Int32] = [1, 2, 3, 4]
let displacements: [Int32] = [0, 1, 3, 6]

do {
    try sendData.withUnsafeBufferPointer { sendBuf in
        try recvData.withUnsafeMutableBufferPointer { recvBuf in
            try comm.allGatherV(sendBuffer: sendBuf,
                              sendCount: sendCount,
                              sendType: .int,
                              recvBuffer: recvBuf,
                              recvCounts: recvCounts,
                              displacements: displacements,
                              recvType: .int)
        }
    }
    print("Process␣\(rank)␣has␣all␣variable␣data:␣\(recvData)")
} catch {
    print("AllGatherV␣failed:␣\(error)")
}
```

5.14 Communicator.allToAllV(sendBuffer:sendCounts:sendDisplacements:sendType:recv

Alltoallv operation - all-to-all with variable amounts of data.

Usage:

```
import SwiftMPI

let comm = SwiftMPI.world
let rank = comm.rank()
let size = comm.size()

// Prepare variable send data
let sendCounts: [Int32] = [1, 2, 1, 2] // For 4 processes
let sendDisplacements: [Int32] = [0, 1, 3, 4]
var sendData: [Int32] = [Int32(rank), Int32(rank), Int32(rank),
   Int32(rank), Int32(rank), Int32(rank)]

let recvCounts: [Int32] = [1, 2, 1, 2]
let recvDisplacements: [Int32] = [0, 1, 3, 4]
var recvData = [Int32](repeating: 0, count: 6)

do {
    try sendData.withUnsafeBufferPointer { sendBuf in
        try recvData.withUnsafeMutableBufferPointer { recvBuf in
            try comm.allToAllV(sendBuffer: sendBuf,
                             sendCounts: sendCounts,
                             sendDisplacements: sendDisplacements,
                             sendType: .int,
                             recvBuffer: recvBuf,
                             recvCounts: recvCounts,
                             recvDisplacements: recvDisplacements,
                             recvType: .int)
        }
    }
    print("Process␣\(rank)␣received:␣\(recvData)")
} catch {
    print("AllToAllV␣failed:␣\(error)")
}
```

5.15 Communicator.probe(source:tag:)

Probe operation - check for incoming message without receiving it.

Usage:

```
import SwiftMPI

let comm = SwiftMPI.world

do {
    let status = try comm.probe(source: 0, tag: 0)
    print("Message␣available␣from␣process␣\(status.source)␣with␣tag␣
        \(status.tag)")
} catch {
    print("Probe␣failed:␣\(error)")
}
```

5.16 Communicator.iprobe(source:tag:)

Iprobe operation - non-blocking probe for incoming message.

Usage:

```
import SwiftMPI

let comm = SwiftMPI.world

do {
    let (found, status) = try comm.iprobe(source: 0, tag: 0)
    if found, let stat = status {
        print("Message␣available␣from␣process␣\(stat.source)")
    } else {
        print("No␣message␣available␣yet")
    }
} catch {
    print("Iprobe␣failed:␣\(error)")
}
```

6 Datatypes

SwiftMPI provides predefined datatypes for common data types:

- `Datatype.char` - 8-bit signed character
- `Datatype.short` - 16-bit signed integer
- `Datatype.int` - 32-bit signed integer
- `Datatype.long` - 64-bit signed integer

- `Datatype.longLong` - 64-bit signed integer
- `Datatype.unsignedChar` - 8-bit unsigned character
- `Datatype.unsignedShort` - 16-bit unsigned integer
- `Datatype.unsigned` - 32-bit unsigned integer
- `Datatype.unsignedLong` - 64-bit unsigned integer
- `Datatype.unsignedLongLong` - 64-bit unsigned integer
- `Datatype.float` - 32-bit floating point
- `Datatype.double` - 64-bit floating point
- `Datatype.longDouble` - Extended precision float
- `Datatype.byte` - Raw byte data
- `Datatype.packed` - Packed data type
- `Datatype.cBool` - C boolean type
- `Datatype.cFloatComplex` - Complex float
- `Datatype.cDoubleComplex` - Complex double
- `Datatype.cLongDoubleComplex` - Complex long double

Usage:

```swift
import SwiftMPI

let comm = SwiftMPI.world
let data: [Double] = [3.14, 2.71]

do {
    try data.withUnsafeBufferPointer { buffer in
        try comm.send(buffer, count: data.count,
                      datatype: .double, dest: 1, tag: 0)
    }
} catch {
    print("Send␣failed:␣\(error)")
}
```

7 Reduction Operations

SwiftMPI provides predefined reduction operations:

- `Operation.max` - Maximum value operation
- `Operation.min` - Minimum value operation
- `Operation.sum` - Sum operation
- `Operation.product` - Product operation
- `Operation.logicalAnd` - Logical AND operation

- `Operation.bitwiseAnd` - Bitwise AND operation
- `Operation.logicalOr` - Logical OR operation
- `Operation.bitwiseOr` - Bitwise OR operation
- `Operation.logicalXor` - Logical XOR operation
- `Operation.bitwiseXor` - Bitwise XOR operation
- `Operation.minLoc` - Minimum with location
- `Operation.maxLoc` - Maximum with location

Usage:

```swift
import SwiftMPI

let comm = SwiftMPI.world
let rank = comm.rank()
let values: [Int32] = [Int32(rank * 10)]
var result: [Int32] = [0]

do {
    try values.withUnsafeBufferPointer { sendBuf in
        try result.withUnsafeMutableBufferPointer { recvBuf in
            // Find maximum value across all processes
            try comm.reduce(sendBuffer: sendBuf,
                            recvBuffer: recvBuf,
                            count: 1,
                            datatype: .int,
                            op: .max,
                            root: 0)
        }
    }

    if rank == 0 {
        print("Maximum␣value:␣\(result[0])")
    }
} catch {
    print("Reduce␣failed:␣\(error)")
}
```

8 Status

8.1 Status.source

Get source rank of received message.

Usage:

```swift
import SwiftMPI

let comm = SwiftMPI.world
var buffer = [Int32](repeating: 0, count: 5)

do {
    let status = try buffer.withUnsafeMutableBufferPointer { buf in
        try comm.receive(buf, count: 5, datatype: .int,
                        source: -1, tag: -1) // Receive from any source
    }
    print("Received␣message␣from␣process␣\(status.source)")
    print("Message␣tag:␣\(status.tag)")
    print("Element␣count:␣\(status.elementCount)")
} catch {
    print("Receive␣failed:␣\(error)")
}
```

8.2 Status.tag

Get tag of received message.

Usage:

```
import SwiftMPI

let comm = SwiftMPI.world
var buffer = [Int32](repeating: 0, count: 10)

do {
    let status = try buffer.withUnsafeMutableBufferPointer { buf in
        try comm.receive(buf, count: 10, datatype: .int,
                        source: -1, tag: -1) // Receive with any tag
    }
    print("Received␣message␣with␣tag␣\(status.tag)")
} catch {
    print("Receive␣failed:␣\(error)")
}
```

8.3 Status.count(datatype:)

Get count of received elements for given datatype.

Usage:

```
import SwiftMPI

let comm = SwiftMPI.world
var buffer = [Int32](repeating: 0, count: 10)

do {
    let status = try buffer.withUnsafeMutableBufferPointer { buf in
        try comm.receive(buf, count: 10, datatype: .int,
                        source: 0, tag: 0)
    }
    let elementCount = status.count(datatype: .int)
    print("Received␣\(elementCount)␣elements")
} catch {
    print("Receive␣failed:␣\(error)")
}
```

9 Complete Example

Here is a complete example demonstrating multiple SwiftMPI functions:

```
import SwiftMPI

```

```
do {
    // Initialize MPI
    try SwiftMPI.initialize()

    let comm = SwiftMPI.world
    let rank = comm.rank()
    let size = comm.size()

    print("Process␣\(rank)␣of␣\(size)␣started")

    // Barrier synchronization
    try comm.barrier()

    // Point-to-point communication
    if rank == 0 {
        let data: [Int32] = [1, 2, 3, 4, 5]
        try comm.send(data, to: 1, tag: 0)
        print("Process␣0␣sent␣data␣to␣process␣1")
    } else if rank == 1 {
        let received = try comm.receive(count: 5, from: 0, tag: 0)
        print("Process␣1␣received:␣\(received)")
    }

    // Broadcast
    var broadcastData: [Int32] = [0, 0, 0]
    if rank == 0 {
        broadcastData = [10, 20, 30]
    }
    try broadcastData.withUnsafeMutableBufferPointer { buffer in
        try comm.broadcast(buffer, count: 3, datatype: .int, root: 0)
    }
    print("Process␣\(rank)␣received␣broadcast:␣\(broadcastData)")

    // Reduce operation
    let sendValue: [Int32] = [Int32(rank + 1)]
    var recvValue: [Int32] = [0]
    try sendValue.withUnsafeBufferPointer { sendBuf in
        try recvValue.withUnsafeMutableBufferPointer { recvBuf in
            try comm.reduce(sendBuffer: sendBuf,
                            recvBuffer: recvBuf,
                            count: 1,
                            datatype: .int,
                            op: .sum,
                            root: 0)
        }
    }
    if rank == 0 {
        print("Sum␣of␣all␣ranks:␣\(recvValue[0])")
    }

    // AllReduce
    let allSendValue: [Double] = [Double(rank) * 1.5]
    var allRecvValue: [Double] = [0.0]
    try allSendValue.withUnsafeBufferPointer { sendBuf in
        try allRecvValue.withUnsafeMutableBufferPointer { recvBuf in
            try comm.allReduce(sendBuffer: sendBuf,
                               recvBuffer: recvBuf,
                               count: 1,
                             datatype: .double,
                             op: .sum)
        }
    }
    print("Process \(rank): all-reduce sum = \(allRecvValue[0])")

    // Finalize MPI
    try SwiftMPI.finalize()

} catch {
    print("MPI error: \(error)")
    exit(1)
}
```

10 Error Handling

All SwiftMPI functions that can fail throw `MPIError` exceptions. Always wrap MPI calls in do-catch blocks:

```
import SwiftMPI

do {
    try SwiftMPI.initialize()
    // ... use MPI functions ...
    try SwiftMPI.finalize()
} catch MPIError.alreadyInitialized {
    print("MPI already initialized")
} catch MPIError.notInitialized {
    print("MPI not initialized")
} catch MPIError.communicationFailed {
    print("Communication failed")
} catch {
    print("Other MPI error: \(error)")
}
```

11 MPIError Types

The following error types are defined:

- `MPIError.alreadyInitialized` - MPI already initialized
- `MPIError.notInitialized` - MPI not initialized
- `MPIError.initializationFailed` - MPI initialization failed
- `MPIError.finalizationFailed` - MPI finalization failed
- `MPIError.invalidCommunicator` - Invalid communicator provided
- `MPIError.invalidRank` - Invalid rank specified
- `MPIError.invalidTag` - Invalid message tag
- `MPIError.invalidDatatype` - Invalid datatype specified
- `MPIError.communicationFailed` - Communication operation failed

- `MPIError.operationFailed(operation: String)` - Generic operation failure
- `MPIError.processSpawnFailed` - Failed to spawn processes
- `MPIError.connectionFailed` - Failed to establish connection