High Performance Computing: OpenMP and MPI

Prepared for Exam Revision

May 2025

A comprehensive guide to OpenMP and MPI for shared-memory and distributed-memory parallel programming.

Contents

1 Introduction

This document provides a detailed course on **OpenMP** and **MPI**, two key technologies for parallel computing. OpenMP is designed for shared-memory systems using threads, while MPI is suited for distributed-memory systems using processes. The course includes concepts, syntax, examples, and comparisons to aid in preparing for your High Performance Computing exam.

Note

This document uses blue for OpenMP-related content and green for MPI-related content to enhance clarity.

2 OpenMP

OpenMP (Open Multi-Processing) is an API for shared-memory parallel programming in C, C++, and Fortran. It uses threads to execute code concurrently on a single machine with shared memory.

2.1 Overview

- **Purpose**: Parallelize code on multi-core processors using threads.
- **Key Features**:
  - Shared memory model: All threads access the same memory space.
  - Directives to parallelize loops, sections, or tasks.
  - Synchronization mechanisms to manage thread interactions.
- **Advantages**: Easy to use, portable across platforms.
- **Limitations**: Limited to single-node systems, thread overhead.

2.2 Compiling and Running

To compile, use the `-fopenmp` flag with `gcc`:

```
gcc -fopenmp source.c -o output
./output
```

Example: A simple "Hello World" program.

```c
#include <stdio.h>
#include <omp.h>
int main() {
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        printf("Hello from thread %d\n", thread_id);
    }
    return 0;
}
\newpage
```

Expected Output (varies by number of threads, e.g., for 4 threads):

```
Hello from thread 0
Hello from thread 1
Hello from thread 2
Hello from thread 3
```

Note: The order of output lines may vary due to thread scheduling.

2.3 Setting the Number of Threads

Threads can be set in three ways:

- **Programmatically**: `omp_set_num_threads(NUM_THREADS);`
- **Directive**: `#pragma omp parallel num_threads(NUM_THREADS)`
- **Environment**: `export OMP_NUM_THREADS=NUM_THREADS`

Comparison:

- `omp_set_num_threads`: Global setting for all parallel regions.
- `num_threads`: Specific to a parallel region.
- `OMP_NUM_THREADS`: Default for the entire program.

Example: Using all three methods.

```c
#include <stdio.h>
#include <omp.h>
int main() {
    omp_set_num_threads(4); // Programmatic
    #pragma omp parallel
    {
        printf("Thread %d (programmatic)\n", omp_get_thread_num());
    }
    #pragma omp parallel num_threads(4) // Directive
    {
        printf("Thread %d (directive)\n", omp_get_thread_num());
    }
    #pragma omp parallel // Environment (export OMP_NUM_THREADS=4)
    {
        printf("Thread %d (environment)\n", omp_get_thread_num());
    }
    return 0;
}
```

Expected Output (assuming `OMP_NUM_THREADS=4`):

```
Thread 0 (programmatic)
Thread 1 (programmatic)
Thread 2 (programmatic)
Thread 3 (programmatic)
Thread 0 (directive)
Thread 1 (directive)
Thread 2 (directive)
Thread 3 (directive)
Thread 0 (environment)
Thread 1 (environment)
Thread 2 (environment)
Thread 3 (environment)
```

Note: The order of threads within each section may vary.

2.4 Private and Shared Variables

Variables are **shared** by default. Use `private` to give each thread its own copy.

- `private(var)`: Each thread has an independent copy.
- `shared(var)`: All threads access the same variable.

Example: Private vs. shared variables.

```c
#include <stdio.h>
#include <unistd.h>
#include <omp.h>
int main() {
    int prv = 0, pub = 0;
    printf("Thread ID, Private, Shared\n");
    #pragma omp parallel num_threads(4) private(prv) shared(pub)
    {
        int thread_id = omp_get_thread_num();
        prv = 0;
        prv += thread_id;
        sleep(1);
        pub += thread_id;
        printf("%d, %d, %d\n", thread_id, prv, pub);
    }
    return 0;
}
```

Expected Output:

```
Thread ID, Private, Shared
0, 0, 0
1, 1, 1
2, 2, 3
3, 3, 6
```

Note: The shared variable pub accumulates thread IDs, and the output order may vary.

2.5 Parallelizing For Loops

`#pragma omp for` distributes loop iterations across threads. **Example**: 8 iterations across 4 threads.

```c
#include <stdio.h>
#include <omp.h>
#define NUM_THREADS 4
#define NUM_ITERS 8
int main() {
    #pragma omp parallel num_threads(NUM_THREADS)
    {
        #pragma omp for
        for (int i = 0; i < NUM_ITERS; i++) {
            int id = omp_get_thread_num();
            printf("Thread %d: Loop iteration %d\n", id, i);
        }
    }
    return 0;
}
```

Expected Output (sample for static scheduling):

```
Thread 0: Loop iteration 0
Thread 0: Loop iteration 1
Thread 1: Loop iteration 2
Thread 1: Loop iteration 3
Thread 2: Loop iteration 4
Thread 2: Loop iteration 5
Thread 3: Loop iteration 6
Thread 3: Loop iteration 7
```

Note: The assignment of iterations to threads depends on the scheduling type (default is static). Output order may vary.

2.6 Scheduling

Scheduling controls how iterations are assigned:

- **Static**:Divides the iterations of the loop into equal-sized chunks, with each chunk assigned to a thread statically at compile time. The optional chunk parameter specifies the size of each chunk.
  (Equal chunks at compile time.)
  - Syntax: `#pragma omp for schedule(static[,chunk])`
  - Pros: Low overhead.
  - Cons: Poor load balancing.
- **Dynamic**: Divides the iterations of the loop into chunks of size chunk, and assigns these chunks dynamically to threads as threads become available to do work.
  (Assigns chunks dynamically as threads become available.)
  - Syntax: `#pragma omp for schedule(dynamic[,chunk])`
  - Pros: Better load balancing.
  - Cons: Higher overhead.

- **Guided**:Similar to dynamic scheduling, but the size of the chunks decreases dynamically over time. Initially, larger chunks are assigned, which helps reduce scheduling overhead, but as the loop progresses, the chunk size decreases to balance the workload among threads.
  (Assigns larger chunks initially, decreasing over time.)
  - Syntax: `#pragma omp for schedule(guided[,chunk])`
  - Pros: Balances overhead and load.
  - Cons: Less predictable.
- **Runtime**: Allows the scheduling policy to be determined at runtime using the OMP SCHEDULE environment variable or the omp set schedule function. This provides flexibility in choosing the scheduling policy without recompiling the code.
  Set via `OMP_SCHEDULE`.
  - Syntax: `#pragma omp for schedule(runtime)`
  - Pros: Flexible.
  - Cons: Requires external setup.

Example: Comparing scheduling types.

```c
#include <stdio.h>
#include <omp.h>
#define NUM_THREADS 4
#define NUM_ITERS 16
int main() {
    printf("Static Scheduling:\n");
    #pragma omp parallel num_threads(NUM_THREADS)
    {
        #pragma omp for schedule(static, 2)
        for (int i = 0; i < NUM_ITERS; i++) {
            printf("Thread %d: Iteration %d\n", omp_get_thread_num(), i);
        }
    }
    printf("\nDynamic Scheduling:\n");
    #pragma omp parallel num_threads(NUM_THREADS)
    {
        #pragma omp for schedule(dynamic, 2)
        for (int i = 0; i < NUM_ITERS; i++) {
            printf("Thread %d: Iteration %d\n", omp_get_thread_num(), i);
        }
    }
    printf("\nGuided Scheduling:\n");
    #pragma omp parallel num_threads(NUM_THREADS)
    {
        #pragma omp for schedule(guided, 2)
        for (int i = 0; i < NUM_ITERS; i++) {
            printf("Thread %d: Iteration %d\n", omp_get_thread_num(), i);
        }
    }
    printf("\nRuntime Scheduling:\n");
    #pragma omp parallel num_threads(NUM_THREADS)
    {
        #pragma omp for schedule(runtime)
        for (int i = 0; i < NUM_ITERS; i++) {
            printf("Thread %d: Iteration %d\n", omp_get_thread_num(), i);
```

```
        }
    }
    return 0;
}
```

Expected Output (sample):

```
Static Scheduling:
Thread 0: Iteration 0
Thread 0: Iteration 1
Thread 0: Iteration 2
Thread 0: Iteration 3
Thread 1: Iteration 4
Thread 1: Iteration 5
Thread 1: Iteration 6
Thread 1: Iteration 7
Thread 2: Iteration 8
Thread 2: Iteration 9
Thread 2: Iteration 10
Thread 2: Iteration 11
Thread 3: Iteration 12
Thread 3: Iteration 13
Thread 3: Iteration 14
Thread 3: Iteration 15

Dynamic Scheduling:
Thread 0: Iteration 0
Thread 0: Iteration 1
Thread 1: Iteration 2
Thread 1: Iteration 3
Thread 2: Iteration 4
Thread 2: Iteration 5
Thread 3: Iteration 6
Thread 3: Iteration 7
Thread 0: Iteration 8
Thread 0: Iteration 9
Thread 1: Iteration 10
Thread 1: Iteration 11
Thread 2: Iteration 12
Thread 2: Iteration 13
Thread 3: Iteration 14
Thread 3: Iteration 15

Guided Scheduling:
Thread 0: Iteration 0
Thread 0: Iteration 1
Thread 0: Iteration 2
Thread 0: Iteration 3
Thread 1: Iteration 4
Thread 1: Iteration 5
Thread 1: Iteration 6
Thread 2: Iteration 7
Thread 2: Iteration 8
```

```
Thread 2: Iteration 9
Thread 3: Iteration 10
Thread 3: Iteration 11
Thread 0: Iteration 12
Thread 1: Iteration 13
Thread 2: Iteration 14
Thread 3: Iteration 15

Runtime Scheduling:
Thread 0: Iteration 0
Thread 0: Iteration 1
Thread 0: Iteration 2
Thread 0: Iteration 3
Thread 1: Iteration 4
Thread 1: Iteration 5
Thread 1: Iteration 6
Thread 1: Iteration 7
Thread 2: Iteration 8
Thread 2: Iteration 9
Thread 2: Iteration 10
Thread 2: Iteration 11
Thread 3: Iteration 12
Thread 3: Iteration 13
Thread 3: Iteration 14
Thread 3: Iteration 15
```

Note: Static scheduling assigns fixed chunks (e.g., 4 iterations per thread). Dynamic and guided scheduling assign chunks as threads become available, with guided reducing chunk sizes over time. Runtime scheduling depends on `OMP_SCHEDULE`, so the output may vary.

2.7 Reduction Clauses

`reduction(op:var)` combines thread results using operators like `+`, `-`, `*`, `&`, `|`, `^`, `&&`, `||`, `max`, `min`. **Example**: Summing an array.

```c
#include <stdio.h>
#include <omp.h>
#define NUM_THREADS 4
#define ARRAY_SIZE 8
int main() {
    int arr[ARRAY_SIZE];
    int total = 0;
    for (int i = 0; i < ARRAY_SIZE; i++) {
        arr[i] = i + 1;
    }
    #pragma omp parallel num_threads(NUM_THREADS)
    {
        #pragma omp for reduction(+:total)
        for (int i = 0; i < ARRAY_SIZE; i++) {
            total += arr[i];
        }
    }
    printf("Total = %d\n", total); // 1+2+3+4+5+6+7+8 = 36
    return 0;
```

```
}
```

Expected Output:

```
Total = 36
```

2.8 Synchronization

- **Critical**:Ensures that a specific block of code is executed by only one thread at a time. all threads will eventually execute the code in this section but not in the same time. (One thread at a time)
- **Single**:Specifies that a block of code should be executed by only one thread, but it doesn't specify which thread. (One thread, others wait.)
- **Master**:master: like single but the block is ensured to be executed by the master (main program,thread with ID 0). (Master thread (ID 0) only.)
- **Barrier**:Synchronizes all threads in a parallel region. Each thread waits at the barrier until all threads have reached it, then they all proceed. (All threads wait.)
- **Atomic**:Specifies that a specific operation is atomic,it is used for indivisible operations like an increment. (Indivisible operation.)

Example: Synchronization directives.

```c
#include <stdio.h>
#include <omp.h>
int main() {
    int counter = 0;
    #pragma omp parallel num_threads(4)
    {
        int id = omp_get_thread_num();
        #pragma omp critical
        {
            counter++;
            printf("Thread %d incremented counter to %d\n", id, counter);
        }
        #pragma omp single
        {
            printf("Thread %d in single section\n", id);
        }
        #pragma omp master
        {
            printf("Master thread %d\n", id);
        }
        #pragma omp barrier
        printf("Thread %d passed barrier\n", id);
        #pragma omp atomic
        counter++;
    }
    printf("Final counter: %d\n", counter);
    return 0;
}
```

Expected Output (sample for 4 threads):

```
Thread 0 incremented counter to 1
Thread 1 incremented counter to 2
Thread 2 incremented counter to 3
Thread 3 incremented counter to 4
Thread 2 in single section
Master thread 0
Thread 0 passed barrier
Thread 1 passed barrier
Thread 2 passed barrier
Thread 3 passed barrier
Final counter: 8
```

Note: The order of critical section outputs may vary. The single section is executed by one thread (ID varies). The final counter is 8 (4 from critical + 4 from atomic).

2.9 Sections

`#pragma omp sections` assigns independent tasks to threads. **Example**: Parallelizing two functions.

```c
#include <stdio.h>
#include <omp.h>
int f1(int b, int c) { return b + c; }
int f2(int d, int e) { return d * e; }
int main() {
    int a = 0, b = 1, c = 2, d = 3, e = 4;
    int a1, a2;
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            a1 = f1(b, c);
            printf("Section 1: a1 = %d\n", a1);
        }
        #pragma omp section
        {
            a2 = f2(d, e);
            printf("Section 2: a2 = %d\n", a2);
        }
    }
    a = a1 + a2;
    printf("a = %d\n", a);
    return 0;
}
```

Expected Output:

```
Section 1: a1 = 3
Section 2: a2 = 12
a = 15
```

Note: The order of section outputs may vary due to thread assignment.

3 MPI

MPI (Message Passing Interface) is a standard for distributed-memory parallel computing using processes.

3.1 Overview

- **Purpose**: Parallelize code across multiple nodes/processes.
- **Key Features**:
  - Distributed memory: Each process has its own memory.
  - Message passing for communication.
  - Scalable to thousands of nodes.
- **Advantages**: Scalable, works on clusters.
- **Limitations**: Complex, higher overhead for small tasks.

3.2 Compiling and Running

Compile with `mpicc`, run with `mpirun`:

```
mpicc source.c -o output
mpirun -np <NUM_PROCESSES> ./output
```

Example: MPI "Hello World".

```c
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Hello from process %d of %d\n", rank, size);
    MPI_Finalize();
    return 0;
}
```

Expected Output (for 4 processes):

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

Note: The order of process outputs may vary.

3.3 General Structure

- `MPI_Init(&argc, &argv);`
- Code and communications.
- `MPI_Finalize();`

3.4 Number of Processes and Rank

- `MPI_Comm_size(MPI_COMM_WORLD, &size);`
- `MPI_Comm_rank(MPI_COMM_WORLD, &rank);`

3.5 Message Passing

`MPI_Send` and `MPI_Recv` for point-to-point communication. **Example**: Sending a number.

```c
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int number;
    if (rank == 0) {
        number = 42;
        MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent %d to process 1\n", number);
    } else if (rank == 1) {
        MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
        printf("Process 1 received %d from process 0\n", number);
    }
    MPI_Finalize();
    return 0;
}
```

Expected Output (for 2 processes):

```
Process 0 sent 42 to process 1
Process 1 received 42 from process 0
```

Note: The order of outputs may vary.

3.6 Broadcast

`MPI_Bcast` sends data from a root process to all others. **Example**: Broadcasting a number.

```c
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int number = 0;
    if (rank == 0) {
        number = 100;
    }
    MPI_Bcast(&number, 1, MPI_INT, 0, MPI_COMM_WORLD);
    printf("Process %d has number %d\n", rank, number);
    MPI_Finalize();
    return 0;
}
```

Expected Output (for 4 processes):

```
Process 0 has number 100
Process 1 has number 100
Process 2 has number 100
Process 3 has number 100
```

Note: The order of process outputs may vary.

3.7 Scatter and Gather

- `MPI_Scatter`: Distributes data from root to all processes.
- `MPI_Gather`: Collects data from all processes to root.

Example: Scatter and gather.

```c
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int sendbuf[4] = {10, 20, 30, 40};
    int recvbuf;
    MPI_Scatter(sendbuf, 1, MPI_INT, &recvbuf, 1, MPI_INT, 0,
        MPI_COMM_WORLD);
    printf("Process %d received %d\n", rank, recvbuf);
    int gatherbuf[4];
    recvbuf *= 2;
    MPI_Gather(&recvbuf, 1, MPI_INT, gatherbuf, 1, MPI_INT, 0,
        MPI_COMM_WORLD);
    if (rank == 0) {
        printf("Gathered data: ");
        for (int i = 0; i < size; i++) {
            printf("%d ", gatherbuf[i]);
        }
```

```
        printf("\n");
    }
    MPI_Finalize();
    return 0;
}
```

Expected Output (for 4 processes):

```
Process 0 received 10
Process 1 received 20
Process 2 received 30
Process 3 received 40
Gathered data: 20 40 60 80
```

Note: The order of scatter outputs may vary, but the gathered data is deterministic.

3.8 Reduce

`MPI_Reduce` combines data using an operation. **Example**: Summing values.

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int value = rank + 1;
    int result;
    MPI_Reduce(&value, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0) {
        printf("Sum of values: %d\n", result);
    }
    MPI_Finalize();
    return 0;
}
```

Expected Output (for 4 processes):

```
Sum of values: 10
```

Note: The sum is $1 + 2 + 3 + 4 = 10$, printed only by rank 0.

3.9 Synchronization

`MPI_Barrier` synchronizes all processes. **Example**: Ordered "Hello World".

```c
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    int size, rank;
    char dummy;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank != 0) {
        int src = (rank + size - 1) % size;
        MPI_Recv(&dummy, 1, MPI_CHAR, src, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
    }
    printf("Process %d is saying Hello World.\n", rank);
    if (rank != size - 1) {
        MPI_Send(&dummy, 1, MPI_CHAR, (rank + 1) % size, 0,
            MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}
```

Expected Output (for 4 processes):

```
Process 0 is saying Hello World.
Process 1 is saying Hello World.
Process 2 is saying Hello World.
Process 3 is saying Hello World.
```

Note: The outputs are ordered due to the token-passing synchronization.

3.10 Cluster Setup

Master Node:

1. Update: `sudo apt update && sudo apt upgrade`
2. Install MPI: `sudo apt install build-essential mpi-default-dev`
3. SSH: `ssh-keygen -t rsa`, `ssh-copy-id user@slave_ip`
4. NFS: `sudo apt install nfs-server`, configure `/etc/exports`.
5. Machines file: List slave IPs.
6. Run: `mpirun -n 2 -machinefile machines ./program`

Slave Node:

1. Update system.
2. Install MPI and SSH server.
3. NFS client: `sudo apt install nfs-client`, mount master's share.

Table 1: Comparison of OpenMP and MPI

Feature	OpenMP	MPI
Memory Model	Shared memory (threads share memory)	Distributed memory (processes have own memory)
Parallel Unit	Threads	Processes
Ease of Use	Easier (directives, minimal changes)	Harder (explicit message passing)
Scalability	Limited to single node	Scales to clusters
Communication	Implicit (shared variables)	Explicit (send/receive)
Synchronization	Critical, single, master, barrier, atomic	Barrier, send/receive ordering
Use Case	Multi-core CPUs, simple tasks	Clusters, large-scale computing

4 OpenMP vs. MPI

5 Study Tips for Your Exam

- **Understand Concepts**: Differentiate shared vs. distributed memory.
- **Memorize**: Key directives (`#pragma omp parallel`) and MPI functions (`MPI_Send`).
- **Practice**: Compile and modify provided examples.
- **Compare**: Scheduling types, synchronization, OpenMP vs. MPI.
- **Debug**: Watch for race conditions (OpenMP) and deadlocks (MPI).