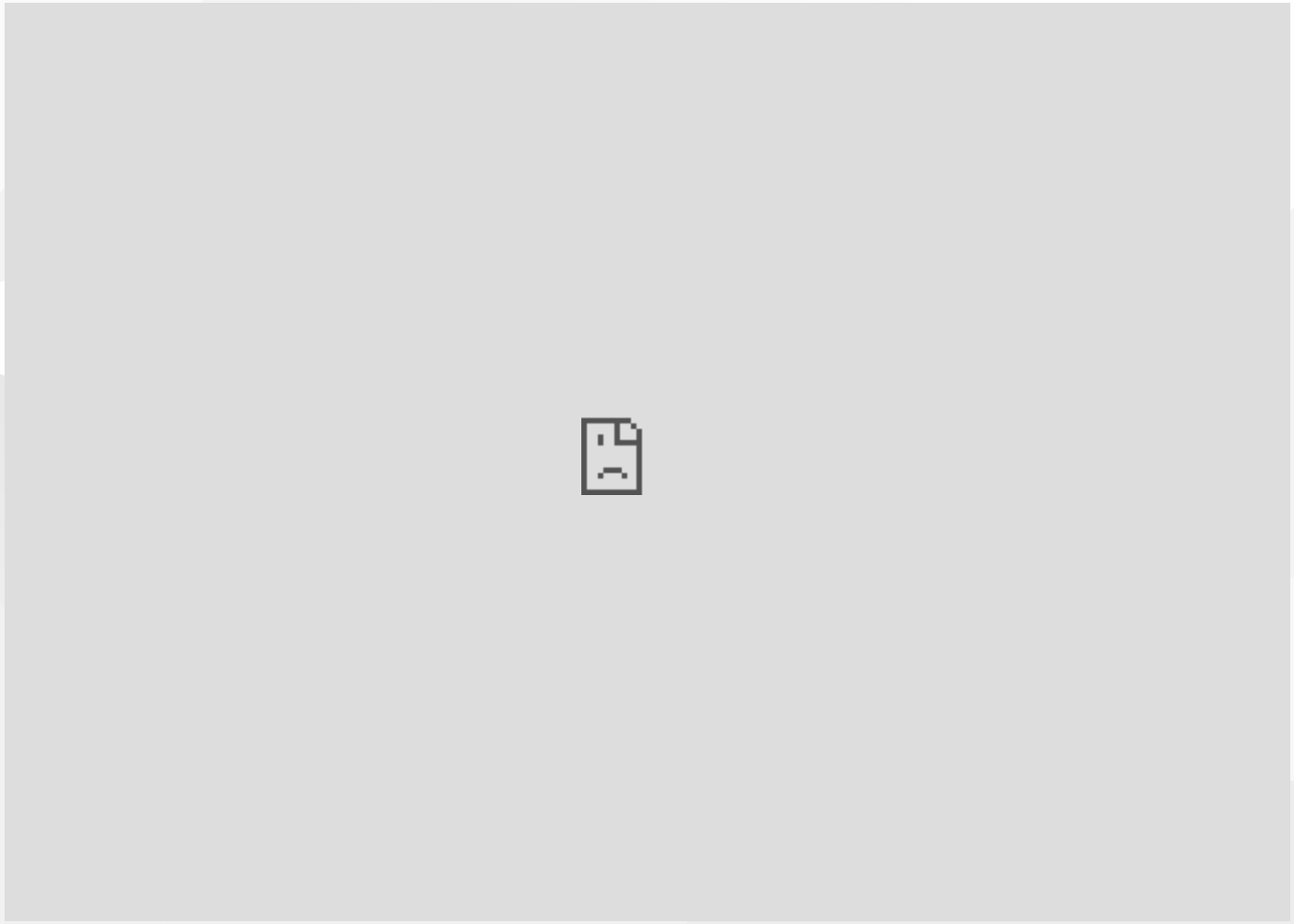
CEN310 Parallel Programming

Week-3

OpenMP Programming

Download

- PDF
- DOC
- SLIDE
- PPTX



Outline (1/4)

1. OpenMP Fundamentals
    - Introduction to OpenMP
    - Compilation and Execution
    - Runtime Library Functions
    - Environment Variables
    - Parallel Regions
2. Work-Sharing Constructs
    - Parallel For Loops
    - Sections
    - Single Execution
    - Task Constructs
    - Workshare Directives

Outline (2/4)

3. Data Management
    - Shared vs Private Variables
    - Data Scope Attributes
    - Reduction Operations
    - Array Sections
    - Memory Model
4. Synchronization
    - Critical Sections
    - Atomic Operations
    - Barriers
    - Ordered Sections
    - Locks and Mutexes

Outline (3/4)

5. Advanced Features
    - Nested Parallelism
    - Task Dependencies
    - SIMD Directives
    - Device Offloading
    - Thread Affinity
6. Performance Optimization
    - Scheduling Strategies
    - Load Balancing
    - Cache Optimization
    - False Sharing
    - Performance Analysis

Outline (4/4)

7. Best Practices
    - Code Organization
    - Error Handling
    - Debugging Techniques
    - Portability
    - Common Pitfalls
8. Real-World Applications
    - Scientific Computing
    - Data Processing
    - Financial Modeling
    - Image Processing

1. OpenMP Fundamentals

Introduction to OpenMP (1/4)

OpenMP (Open Multi-Processing) is:

- An API for shared-memory parallel programming
- Supports C, C++, and Fortran
- Based on compiler directives
- Portable and scalable

Key Components:

1. Compiler Directives
2. Runtime Library Functions
3. Environment Variables

Introduction to OpenMP (2/4)

Basic Structure

```c
#include <omp.h>

int main() {
    // Serial code

    #pragma omp parallel
    {
        // Parallel region
        int thread_id = omp_get_thread_num();
        int total_threads = omp_get_num_threads();

        printf("Thread %d of %d\n",
               thread_id, total_threads);
    }

    // Serial code
    return 0;
}
```

Introduction to OpenMP (3/4)

Compilation and Execution

```
# GCC Compilation
g++ -fopenmp program.cpp -o program

# Intel Compilation
icpc -qopenmp program.cpp -o program

# Microsoft Visual C++
cl /openmp program.cpp
```

Environment Variables:

```
# Set number of threads
export OMP_NUM_THREADS=4

# Set thread affinity
export OMP_PROC_BIND=true

# Set scheduling policy
export OMP_SCHEDULE="dynamic,1000"
```

Introduction to OpenMP (4/4)

Runtime Library Functions

```
void runtime_control_example() {
    // Get maximum threads available
    int max_threads = omp_get_max_threads();

    // Set number of threads
    omp_set_num_threads(4);

    // Get current thread number
    int thread_id = omp_get_thread_num();

    // Check if in parallel region
    bool in_parallel = omp_in_parallel();

    // Get processor time
    double start = omp_get_wtime();
    // ... computation ...
    double end = omp_get_wtime();

    printf("Time taken: %f seconds\n", end - start);
}
```

2. Work-Sharing Constructs

Parallel For Loops (1/4)

```
void parallel_for_example() {
    const int N = 1000000;
    std::vector<double> data(N);

    // Basic parallel for
    #pragma omp parallel for
    for(int i = 0; i < N; i++) {
        data[i] = heavy_computation(i);
    }

    // With scheduling clause
    #pragma omp parallel for schedule(dynamic, 1000)
    for(int i = 0; i < N; i++) {
        data[i] = variable_work(i);
    }

    // Nested loops
    #pragma omp parallel for collapse(2)
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < M; j++) {
            matrix[i][j] = compute(i, j);
        }
    }
}
```

Parallel For Loops (2/4)

Schedule Types

```
void demonstrate_scheduling() {
    const int N = 1000000;

    // Static scheduling
    #pragma omp parallel for schedule(static)
    for(int i = 0; i < N; i++)
        work_static(i);

    // Dynamic scheduling
    #pragma omp parallel for schedule(dynamic, 1000)
    for(int i = 0; i < N; i++)
        work_dynamic(i);

    // Guided scheduling
    #pragma omp parallel for schedule(guided)
    for(int i = 0; i < N; i++)
        work_guided(i);

    // Auto scheduling
    #pragma omp parallel for schedule(auto)
    for(int i = 0; i < N; i++)
        work_auto(i);
}
```

Parallel For Loops (3/4)

Loop Dependencies

```
// Incorrect - Loop carried dependency
for(int i = 1; i < N; i++) {
    data[i] = data[i-1] + 1;  // Dependency!
}

// Correct - No dependencies
#pragma omp parallel for
for(int i = 0; i < N; i++) {
    data[i] = initial[i] + 1;  // Independent
}

// Using ordered clause
#pragma omp parallel for ordered
for(int i = 0; i < N; i++) {
    // Parallel work
    result = heavy_computation(i);

    #pragma omp ordered
    {
        // Sequential part
        output[i] = result;
    }
}
```

Parallel For Loops (4/4)

Performance Considerations

```
void optimize_parallel_for() {
    const int N = 1000000;
    std::vector<double> data(N);

    // Bad - Too fine-grained
    #pragma omp parallel for
    for(int i = 0; i < N; i++) {
        data[i] = sin(i);  // Too little work per iteration
    }

    // Better - Chunked processing
    const int CHUNK = 1000;
    #pragma omp parallel for schedule(static, CHUNK)
    for(int i = 0; i < N; i++) {
        // More work per iteration
        data[i] = complex_computation(i);
    }

    // Best - Vectorization + Parallelization
    #pragma omp parallel for simd
    for(int i = 0; i < N; i++) {
        data[i] = compute_optimized(i);
    }
}
```

Sections (1/3)

```
void parallel_sections_example() {
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            // Task 1
            process_data_part1();
        }

        #pragma omp section
        {
            // Task 2
            process_data_part2();
        }

        #pragma omp section
        {
            // Task 3
            process_data_part3();
        }
    }
}
```

Sections (2/3)

Load Balancing with Sections

```
void load_balanced_sections() {
    std::vector<Task> tasks = get_tasks();
    int num_tasks = tasks.size();

    #pragma omp parallel
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                for(int i = 0; i < num_tasks; i += 3)
                    process_task(tasks[i]);
            }

            #pragma omp section
            {
                for(int i = 1; i < num_tasks; i += 3)
                    process_task(tasks[i]);
            }

            #pragma omp section
            {
                for(int i = 2; i < num_tasks; i += 3)
                    process_task(tasks[i]);
            }
        }
    }
}
```

Sections (3/3)

Nested Sections

```
void nested_sections() {
    #pragma omp parallel
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                // Outer section 1
                #pragma omp parallel sections
                {
                    #pragma omp section
                    { /* Inner task 1.1 */ }

                    #pragma omp section
                    { /* Inner task 1.2 */ }
                }
            }

            #pragma omp section
            {
                // Outer section 2
                #pragma omp parallel sections
                {
                    #pragma omp section
                    { /* Inner task 2.1 */ }

                    #pragma omp section
                    { /* Inner task 2.2 */ }
                }
            }
        }
    }
}
```

3. Data Management

Shared vs Private Variables (1/4)

```
void data_sharing_example() {
    int shared_var = 0;       // Shared by default
    int private_var = 0;      // Will be private
    int firstprivate_var = 5; // Initial value preserved
    int lastprivate_var;      // Final value preserved

    #pragma omp parallel private(private_var) \
                         firstprivate(firstprivate_var) \
                         lastprivate(lastprivate_var) \
                         shared(shared_var)
    {
        private_var = omp_get_thread_num(); // Each thread has own copy
        firstprivate_var += 1; // Each thread starts with 5
        lastprivate_var = compute(); // Last iteration value kept

        #pragma omp atomic
        shared_var += private_var; // Update shared variable
    }

    printf("Final shared_var: %d\n", shared_var);
    printf("Final lastprivate_var: %d\n", lastprivate_var);
}
```

Shared vs Private Variables (2/4)

Default Clause

```
void default_sharing() {
    int var1 = 1, var2 = 2, var3 = 3;

    // All variables shared by default
    #pragma omp parallel default(shared)
    {
        // Must explicitly declare private variables
        int private_var = omp_get_thread_num();
        var1 += private_var; // Potential race condition
    }

    // No implicit sharing
    #pragma omp parallel default(none) \
                         shared(var1) private(var2)
    {
        var1 += var2; // Must specify all variables
        // var3 will cause compilation error
    }
}
```

Shared vs Private Variables (3/4)

Array Handling

```
void array_sharing() {
    const int N = 1000;
    std::vector<double> shared_array(N);

    #pragma omp parallel
    {
        // Thread-local array
        double private_array[N];

        // Initialize private array
        for(int i = 0; i < N; i++)
            private_array[i] = omp_get_thread_num();

        // Combine results into shared array
        #pragma omp critical
        for(int i = 0; i < N; i++)
            shared_array[i] += private_array[i];
    }
}
```

Shared vs Private Variables (4/4)

Complex Data Structures

```
class ThreadSafeCounter {
    std::atomic<int> count;

public:
    void increment() {
        count++;
    }

    int get() const {
        return count.load();
    }
};

void complex_data_sharing() {
    ThreadSafeCounter counter;
    std::vector<std::string> results;
    std::mutex results_mutex;

    #pragma omp parallel shared(counter, results)
    {
        // Thread-local string
        std::string local_result =
            process_data(omp_get_thread_num());

        // Update shared counter
        counter.increment();

        // Add to shared vector safely
        {
            std::lock_guard<std::mutex> lock(results_mutex);
            results.push_back(local_result);
        }
    }
}
```

4. Synchronization

Critical Sections (1/4)

```
void critical_section_example() {
    std::vector<int> results;
    int sum = 0;

    #pragma omp parallel
    {
        int local_result = compute();

        // Basic critical section
        #pragma omp critical
        {
            results.push_back(local_result);
            sum += local_result;
        }

        // Named critical sections
        #pragma omp critical(update_results)
        results.push_back(local_result);

        #pragma omp critical(update_sum)
        sum += local_result;
    }
}
```

Critical Sections (2/4)

Atomic Operations

```
void atomic_operations() {
    int counter = 0;
    double x = 0.0;

    #pragma omp parallel
    {
        // Atomic increment
        #pragma omp atomic
        counter++;

        // Atomic update
        #pragma omp atomic update
        x += 1.0;

        // Atomic read
        #pragma omp atomic read
        int current = counter;

        // Atomic write
        #pragma omp atomic write
        x = 1.0;

        // Atomic capture
        #pragma omp atomic capture
        {
            current = counter;
            counter++;
        }
    }
}
```

Critical Sections (3/4)

Locks and Mutexes

```
void lock_example() {
    omp_lock_t lock;
    omp_init_lock(&lock);

    #pragma omp parallel
    {
        // Simple lock
        omp_set_lock(&lock);
        // Critical section
        omp_unset_lock(&lock);

        // Nested locks
        omp_nest_lock_t nest_lock;
        omp_init_nest_lock(&nest_lock);

        omp_set_nest_lock(&nest_lock);
        // Outer critical section
        {
            omp_set_nest_lock(&nest_lock);
            // Inner critical section
            omp_unset_nest_lock(&nest_lock);
        }
        omp_unset_nest_lock(&nest_lock);

        omp_destroy_nest_lock(&nest_lock);
    }

    omp_destroy_lock(&lock);
}
```

Critical Sections (4/4)

Performance Considerations

```
void optimize_critical_sections() {
    const int N = 1000000;
    std::vector<int> results;
    results.reserve(N); // Prevent reallocation

    // Bad - Too many critical sections
    #pragma omp parallel for
    for(int i = 0; i < N; i++) {
        int result = compute(i);
        #pragma omp critical
        results.push_back(result);
    }

    // Better - Local collection then merge
    #pragma omp parallel
    {
        std::vector<int> local_results;
        local_results.reserve(N/omp_get_num_threads());

        #pragma omp for nowait
        for(int i = 0; i < N; i++) {
            local_results.push_back(compute(i));
        }

        #pragma omp critical
        results.insert(results.end(),
                       local_results.begin(),
                       local_results.end());
    }
}
```

5. Advanced Features

Nested Parallelism (1/4)

```
void nested_parallel_example() {
    // Enable nested parallelism
    omp_set_nested(1);

    #pragma omp parallel num_threads(2)
    {
        int outer_id = omp_get_thread_num();

        #pragma omp parallel num_threads(2)
        {
            int inner_id = omp_get_thread_num();
            printf("Outer thread %d, Inner thread %d\n",
                   outer_id, inner_id);

            // Nested parallel for
            #pragma omp parallel for
            for(int i = 0; i < 100; i++) {
                heavy_computation(i, outer_id, inner_id);
            }
        }
    }
}
```

Nested Parallelism (2/4)

Task Dependencies

```
void task_dependencies() {
    int x = 0, y = 0, z = 0;

    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp task depend(out: x)
        x = compute_x();

        #pragma omp task depend(out: y)
        y = compute_y();

        #pragma omp task depend(in: x, y) depend(out: z)
        z = combine(x, y);

        #pragma omp task depend(in: z)
        process_result(z);
    }
}
```

Nested Parallelism (3/4)

SIMD Directives

```
void simd_operations() {
    const int N = 1000000;
    float a[N], b[N], c[N];

    // Basic SIMD
    #pragma omp simd
    for(int i = 0; i < N; i++)
        c[i] = a[i] + b[i];

    // SIMD with reduction
    float sum = 0.0f;
    #pragma omp simd reduction(+:sum)
    for(int i = 0; i < N; i++)
        sum += a[i] * b[i];

    // Combined parallel for SIMD
    #pragma omp parallel for simd
    for(int i = 0; i < N; i++)
        c[i] = std::sqrt(a[i] * a[i] + b[i] * b[i]);
}
```

Nested Parallelism (4/4)

Device Offloading

```
void device_offload_example() {
    const int N = 1000000;
    std::vector<float> a(N), b(N), c(N);

    // Initialize data
    for(int i = 0; i < N; i++) {
        a[i] = i;
        b[i] = i * 2;
    }

    // Offload computation to device
    #pragma omp target teams distribute parallel for \
            map(to: a[0:N], b[0:N]) map(from: c[0:N])
    for(int i = 0; i < N; i++)
        c[i] = a[i] + b[i];

    // Check result
    for(int i = 0; i < N; i++)
        assert(c[i] == a[i] + b[i]);
}
```

6. Performance Optimization

Scheduling Strategies (1/4)

```
void demonstrate_scheduling_impact() {
    const int N = 1000000;
    std::vector<int> workload(N);

    // Generate varying workload
    for(int i = 0; i < N; i++)
        workload[i] = (i * 17) % 1000;

    Timer t;
    double times[4];

    // Static scheduling
    t.start();
    #pragma omp parallel for schedule(static)
    for(int i = 0; i < N; i++)
        heavy_work(workload[i]);
    times[0] = t.stop();

    // Dynamic scheduling
    t.start();
    #pragma omp parallel for schedule(dynamic, 1000)
    for(int i = 0; i < N; i++)
        heavy_work(workload[i]);
    times[1] = t.stop();

    // Guided scheduling
    t.start();
    #pragma omp parallel for schedule(guided)
    for(int i = 0; i < N; i++)
        heavy_work(workload[i]);
    times[2] = t.stop();

    // Auto scheduling
    t.start();
    #pragma omp parallel for schedule(auto)
    for(int i = 0; i < N; i++)
        heavy_work(workload[i]);
    times[3] = t.stop();

    // Compare results
    printf("Static:  %.3f s\n", times[0]);
    printf("Dynamic: %.3f s\n", times[1]);
    printf("Guided:  %.3f s\n", times[2]);
    printf("Auto:    %.3f s\n", times[3]);
}
```

Scheduling Strategies (2/4)

Load Balancing

```
void load_balancing_example() {
    std::vector<Task> tasks = get_tasks();
    std::atomic<int> completed = 0;

    // Poor load balancing
    #pragma omp parallel for schedule(static)
    for(size_t i = 0; i < tasks.size(); i++) {
        process_task(tasks[i]);
        completed++;
    }

    // Better load balancing
    #pragma omp parallel
    {
        std::vector<Task> local_tasks;

        #pragma omp for schedule(dynamic, 10)
        for(size_t i = 0; i < tasks.size(); i++) {
            local_tasks.push_back(tasks[i]);
            if(local_tasks.size() >= 10) {
                process_task_batch(local_tasks);
                local_tasks.clear();
            }
        }

        // Process remaining tasks
        if(!local_tasks.empty())
            process_task_batch(local_tasks);
    }
}
```

Scheduling Strategies (3/4)

Cache Optimization

```
void cache_friendly_processing() {
    const int N = 1024;
    Matrix matrix(N, N);

    // Cache-unfriendly access
    #pragma omp parallel for
    for(int j = 0; j < N; j++)
        for(int i = 0; i < N; i++)
            matrix(i,j) = compute(i,j);

    // Cache-friendly access
    #pragma omp parallel for collapse(2)
    for(int i = 0; i < N; i++)
        for(int j = 0; j < N; j++)
            matrix(i,j) = compute(i,j);

    // Block processing for better cache usage
    const int BLOCK_SIZE = 32;
    #pragma omp parallel for collapse(2)
    for(int i = 0; i < N; i += BLOCK_SIZE)
        for(int j = 0; j < N; j += BLOCK_SIZE)
            process_block(matrix, i, j, BLOCK_SIZE);
}
```

Scheduling Strategies (4/4)

False Sharing Prevention

```
void prevent_false_sharing() {
    const int N = omp_get_max_threads();

    // Bad - False sharing
    int counters[N];

    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        for(int i = 0; i < 1000000; i++)
            counters[id]++;
    }

    // Good - Padded structure
    struct PaddedInt {
        int value;
        char padding[60]; // Separate cache lines
    };

    PaddedInt padded_counters[N];

    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        for(int i = 0; i < 1000000; i++)
            padded_counters[id].value++;
    }
}
```

7. Best Practices

Code Organization (1/4)

```
// Good practice - Modular parallel functions
class ParallelProcessor {
private:
    std::vector<double> data;
    const int N;

    // Helper function for parallel region
    void process_chunk(int start, int end) {
        for(int i = start; i < end; i++)
            data[i] = heavy_computation(i);
    }

public:
    ParallelProcessor(int size) : N(size), data(size) {}

    void process() {
        #pragma omp parallel
        {
            int num_threads = omp_get_num_threads();
            int thread_id = omp_get_thread_num();

            int chunk_size = N / num_threads;
            int start = thread_id * chunk_size;
            int end = (thread_id == num_threads - 1) ?
                      N : start + chunk_size;

            process_chunk(start, end);
        }
    }
};
```

Code Organization (2/4)

Error Handling

```
class ParallelError : public std::runtime_error {
public:
    ParallelError(const std::string& msg)
        : std::runtime_error(msg) {}
};

void safe_parallel_execution() {
    try {
        #pragma omp parallel
        {
            #pragma omp single
            {
                if(omp_get_num_threads() < 2)
                    throw ParallelError(
                        "Insufficient threads available");
            }

            try {
                // Parallel work
                #pragma omp for
                for(int i = 0; i < N; i++) {
                    if(!is_valid(i))
                        throw ParallelError(
                            "Invalid data at index " +
                            std::to_string(i));
                    process(i);
                }
            }
            catch(...) {
                #pragma omp critical
                {
                    // Handle thread-specific error
                }
            }
        }
    }
    catch(const ParallelError& e) {
        std::cerr << "Parallel execution failed: "
                  << e.what() << std::endl;
    }
}
```

Code Organization (3/4)

Debugging Techniques

```
void debug_parallel_execution() {
    // Debug information
    #ifdef _OPENMP
        printf("OpenMP version: %d\n", _OPENMP);
    #else
        printf("OpenMP not enabled\n");
    #endif

    // Thread information
    #pragma omp parallel
    {
        #pragma omp critical
        {
            printf("Thread %d/%d on CPU %d\n",
                   omp_get_thread_num(),
                   omp_get_num_threads(),
                   sched_getcpu());
        }
    }

    // Performance debugging
    double start = omp_get_wtime();

    #pragma omp parallel for schedule(dynamic)
    for(int i = 0; i < N; i++) {
        #pragma omp critical
        {
            printf("Thread %d processing i=%d\n",
                   omp_get_thread_num(), i);
        }
        process(i);
    }

    double end = omp_get_wtime();
    printf("Time: %.3f seconds\n", end - start);
}
```

Code Organization (4/4)

Portability Considerations

```
// Ensure portability across compilers
#ifdef _OPENMP
    #include <omp.h>
#else
    // OpenMP stub functions
    inline int omp_get_thread_num() { return 0; }
    inline int omp_get_num_threads() { return 1; }
    inline double omp_get_wtime() {
        return std::chrono::duration<double>(
            std::chrono::high_resolution_clock::now()
            .time_since_epoch()).count();
    }
#endif

// Portable parallel
```

Scientific Computing (1/3)

Matrix Multiplication

```
void parallel_matrix_multiply(const Matrix& A,
                              const Matrix& B,
                              Matrix& C) {
    const int N = A.rows();
    const int M = A.cols();
    const int P = B.cols();

    // Basic parallel implementation
    #pragma omp parallel for collapse(2)
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < P; j++) {
            double sum = 0.0;
            for(int k = 0; k < M; k++) {
                sum += A(i,k) * B(k,j);
            }
            C(i,j) = sum;
        }
    }
}

// Cache-optimized version
void block_matrix_multiply(const Matrix& A,
                           const Matrix& B,
                           Matrix& C) {
    const int N = A.rows();
    const int BLOCK = 32;  // Tune for your cache size

    #pragma omp parallel for collapse(3)
    for(int i = 0; i < N; i += BLOCK) {
        for(int j = 0; j < N; j += BLOCK) {
            for(int k = 0; k < N; k += BLOCK) {
                multiply_block(A, B, C, i, j, k, BLOCK);
            }
        }
    }
```

Scientific Computing (2/3)

N-Body Simulation

```
struct Particle {
    double x, y, z;
    double vx, vy, vz;
    double mass;
};

void simulate_n_body(std::vector<Particle>& particles,
                     double dt) {
    const int N = particles.size();
    std::vector<double> fx(N), fy(N), fz(N);

    // Compute forces
    #pragma omp parallel for schedule(dynamic)
    for(int i = 0; i < N; i++) {
        double local_fx = 0, local_fy = 0, local_fz = 0;

        for(int j = 0; j < N; j++) {
            if(i != j) {
                compute_force(particles[i], particles[j],
                              local_fx, local_fy, local_fz);
            }
        }

        fx[i] = local_fx;
        fy[i] = local_fy;
        fz[i] = local_fz;
    }

    // Update positions and velocities
    #pragma omp parallel for
    for(int i = 0; i < N; i++) {
        update_particle(particles[i],
                        fx[i], fy[i], fz[i], dt);
    }
}
```

Scientific Computing (3/3)

Monte Carlo Integration

```
double parallel_monte_carlo_pi(long long samples) {
    long long inside_circle = 0;

    #pragma omp parallel reduction(+:inside_circle)
    {
        unsigned int seed = omp_get_thread_num();

        #pragma omp for
        for(long long i = 0; i < samples; i++) {
            double x = (double)rand_r(&seed) / RAND_MAX;
            double y = (double)rand_r(&seed) / RAND_MAX;

            if(x*x + y*y <= 1.0)
                inside_circle++;
        }
    }

    return 4.0 * inside_circle / samples;
}
```

9. Performance Analysis

Profiling Tools (1/3)

```
class PerformanceProfiler {
    struct ProfilePoint {
        std::string name;
        double start_time;
        double total_time;
        int calls;
    };

    std::map<std::string, ProfilePoint> points;

public:
    void start(const std::string& name) {
        auto& point = points[name];
        point.name = name;
        point.start_time = omp_get_wtime();
        point.calls++;
    }

    void stop(const std::string& name) {
        auto& point = points[name];
        point.total_time += omp_get_wtime() - point.start_time;
    }

    void report() {
        printf("\nPerformance Report:\n");
        printf("%-20s %10s %10s %10s\n",
               "Name", "Calls", "Total(s)", "Avg(ms)");

        for(const auto& [name, point] : points) {
            printf("%-20s %10d %10.3f %10.3f\n",
                   name.c_str(),
                   point.calls,
                   point.total_time,
                   (point.total_time * 1000) / point.calls);
        }
    }
};
```

Profiling Tools (2/3)

Using the Profiler

```
void demonstrate_profiling() {
    PerformanceProfiler profiler;
    const int N = 1000000;

    profiler.start("initialization");
    std::vector<double> data(N);
    std::iota(data.begin(), data.end(), 0);
    profiler.stop("initialization");

    profiler.start("computation");
    #pragma omp parallel for
    for(int i = 0; i < N; i++) {
        data[i] = heavy_computation(data[i]);
    }
    profiler.stop("computation");

    profiler.start("reduction");
    double sum = 0;
    #pragma omp parallel for reduction(+:sum)
    for(int i = 0; i < N; i++) {
        sum += data[i];
    }
    profiler.stop("reduction");

    profiler.report();
}
```

Profiling Tools (3/3)

Hardware Performance Counters

```c
#include <papi.h>

void hardware_counters_example() {
    int events[3] = {PAPI_TOT_CYC, PAPI_L1_DCM, PAPI_L2_DCM};
    long long values[3];

    // Initialize PAPI
    PAPI_library_init(PAPI_VER_CURRENT);

    // Create event set
    int event_set = PAPI_NULL;
    PAPI_create_eventset(&event_set);
    PAPI_add_events(event_set, events, 3);

    // Start counting
    PAPI_start(event_set);

    // Your parallel code here
    #pragma omp parallel for
    for(int i = 0; i < N; i++) {
        process_data(i);
    }

    // Stop counting
    PAPI_stop(event_set, values);

    printf("Total cycles: %lld\n", values[0]);
    printf("L1 cache misses: %lld\n", values[1]);
    printf("L2 cache misses: %lld\n", values[2]);
}
```

10. Advanced Topics

Task-Based Parallelism (1/3)

```
void recursive_task_example(Node* root) {
    if(!root) return;

    #pragma omp parallel
    {
        #pragma omp single
        {
            #pragma omp task
            {
                process_node(root);

                #pragma omp task
                recursive_task_example(root->left);

                #pragma omp task
                recursive_task_example(root->right);
            }
        }
    }
}
```

Task-Based Parallelism (2/3)

Task Dependencies

```
void task_dependency_example() {
    double *a, *b, *c, *d;
    // ... allocate arrays ...

    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp task depend(out: a[0:N])
        initialize_array(a, N);

        #pragma omp task depend(out: b[0:N])
        initialize_array(b, N);

        #pragma omp task depend(in: a[0:N], b[0:N]) \
                         depend(out: c[0:N])
        vector_add(a, b, c, N);

        #pragma omp task depend(in: c[0:N]) \
                         depend(out: d[0:N])
        vector_multiply(c, 2.0, d, N);

        #pragma omp taskwait
    }
}
```

Task-Based Parallelism (3/3)

Task Priorities

```
void priority_task_example() {
    #pragma omp parallel
    #pragma omp single
    {
        for(int i = 0; i < 100; i++) {
            int priority = compute_priority(i);

            #pragma omp task priority(priority)
            {
                process_with_priority(i, priority);
            }
        }
    }
}
```

2. Performance Analysis:
    - Compare different scheduling strategies
    - Measure speedup and efficiency
    - Analyze cache performance
3. Documentation Requirements:
    - Implementation details
    - Performance measurements
    - Analysis and conclusions
    - Optimization strategies

Example starter code:

```
class Image {
    std::vector<unsigned char> data;
    int width, height, channels;
public:
```

Next Week Preview

We will cover:

- MPI Programming
- Distributed Memory Parallelism
- Point-to-Point Communication
- Collective Operations

$$End - Of - Week - 3$$