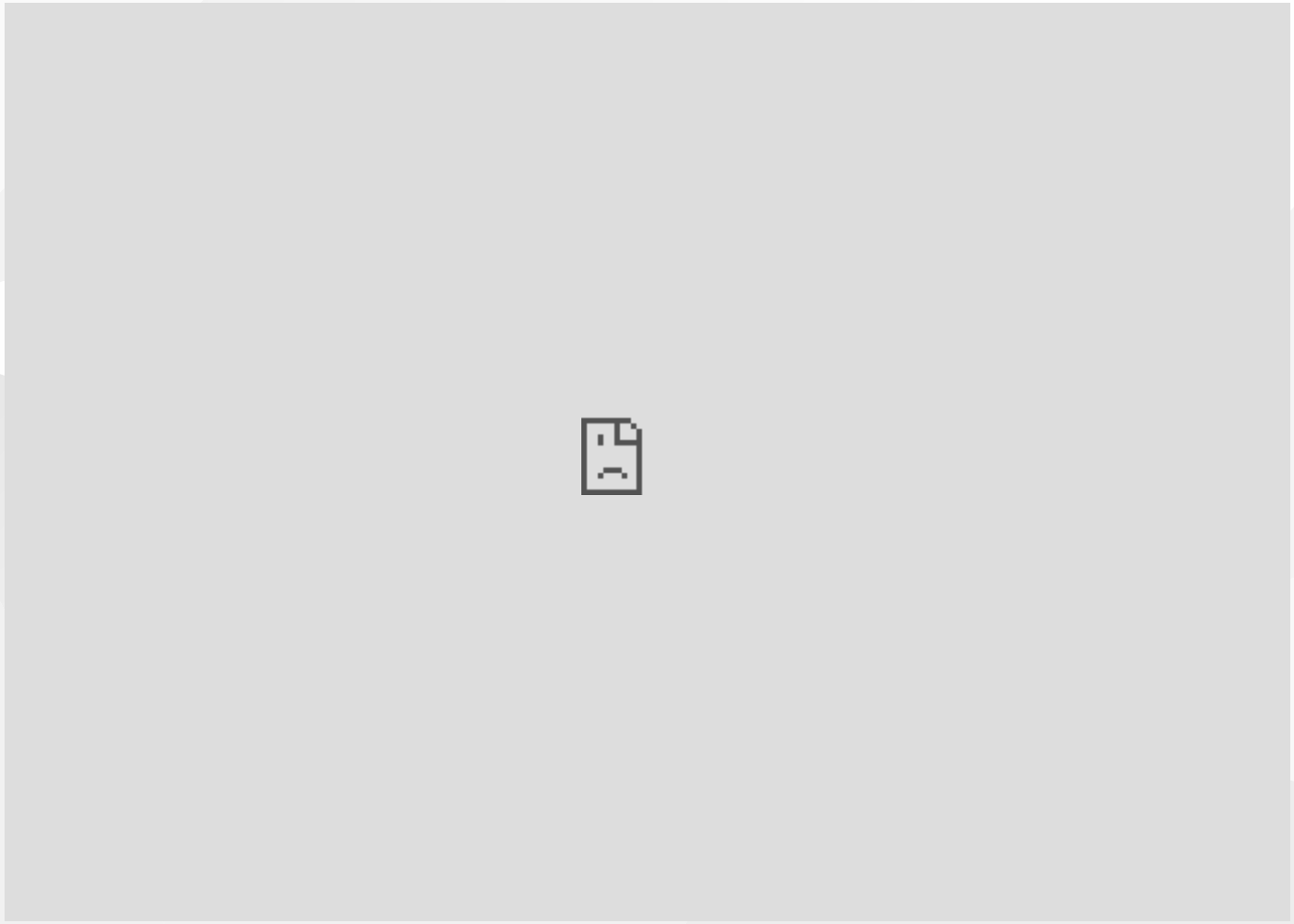
CEN310 Parallel Programming

Week-2

Parallel Computing Fundamentals

Download

- PDF
- DOC
- SLIDE
- PPTX



1. Parallel Computing Architectures
    - Flynn's Taxonomy
    - Memory Architectures
    - Interconnection Networks
    - Modern Processor Architectures
    - Cache Coherence
    - Memory Consistency Models
2. Performance Metrics
    - Speedup Types
    - Efficiency Analysis
    - Amdahl's Law
    - Gustafson's Law
    - Scalability Measures

Outline (2/4)

3. Parallel Algorithm Design
    - Decomposition Strategies
        - Data Decomposition
        - Task Decomposition
        - Pipeline Decomposition
    - Load Balancing
        - Static vs Dynamic
        - Work Stealing
    - Communication Patterns
        - Point-to-Point
        - Collective Operations
    - Synchronization Methods
        - Barriers
        - Locks

Outline (3/4)

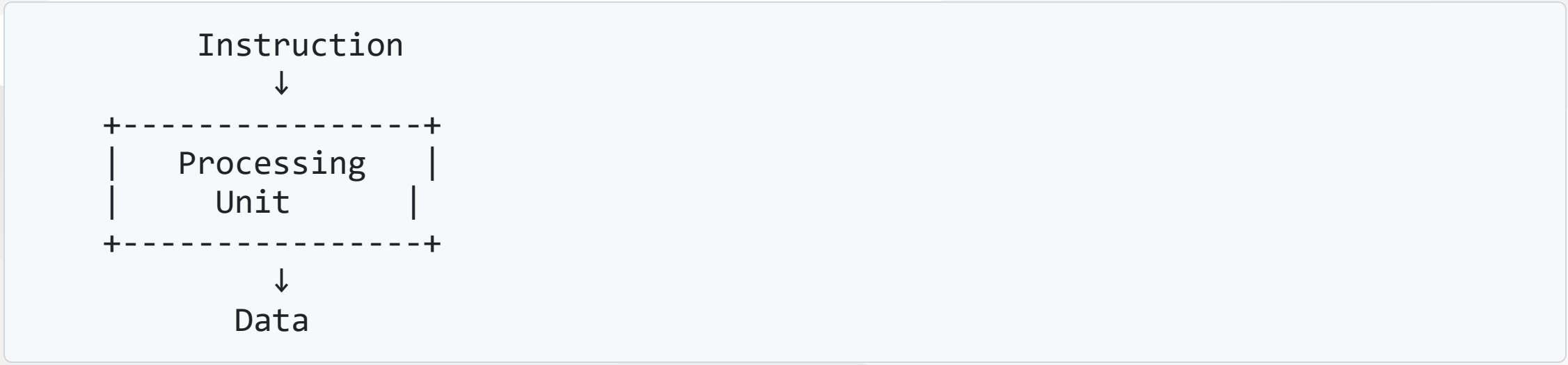
4. Programming Models
   - Shared Memory
     - OpenMP Basics
     - Pthreads Overview
   - Message Passing
     - MPI Concepts
     - Communication Types
   - Data Parallel
     - SIMD Instructions
     - Vector Operations
   - Hybrid Models
     - OpenMP + MPI
     - CPU + GPU

Outline (4/4)

5. Performance Analysis & Optimization
    - Profiling Tools
    - Bottleneck Analysis
    - Memory Access Patterns
    - Cache Optimization
    - Communication Overhead
    - Load Imbalance Detection
6. Real-World Applications
    - Scientific Computing
    - Data Processing
    - Image Processing
    - Financial Modeling

1. Parallel Computing Architectures

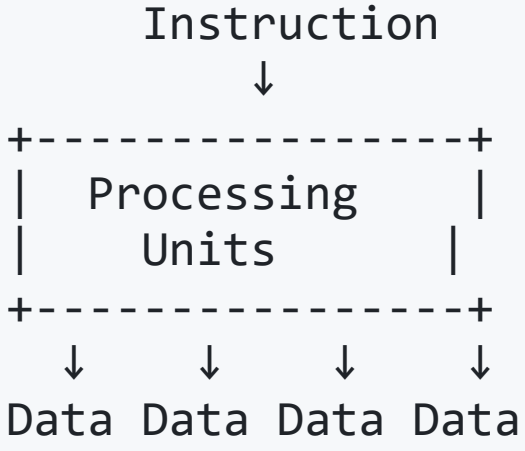
Flynn's Taxonomy (1/4)



- Traditional sequential computing (SISD)
- One instruction stream, one data stream

Flynn's Taxonomy (2/4)

SIMD Architecture



Flynn's Taxonomy (3/4)

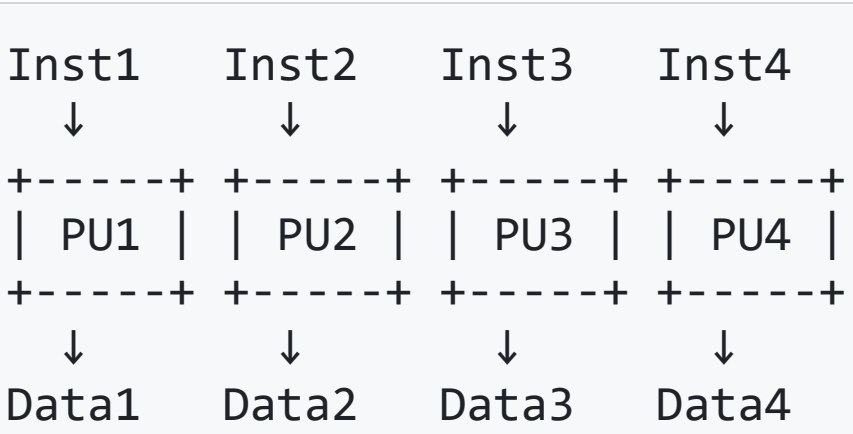
SIMD Example Code (Part 1)

```
#include <immintrin.h>

void vectorAdd(float* a, float* b, float* c, int n) {
    // Process 8 elements at once using AVX
    for (int i = 0; i < n; i += 8) {
        __m256 va = _mm256_load_ps(&a[i]);
        __m256 vb = _mm256_load_ps(&b[i]);
        __m256 vc = _mm256_add_ps(va, vb);
        _mm256_store_ps(&c[i], vc);
    }
}
```

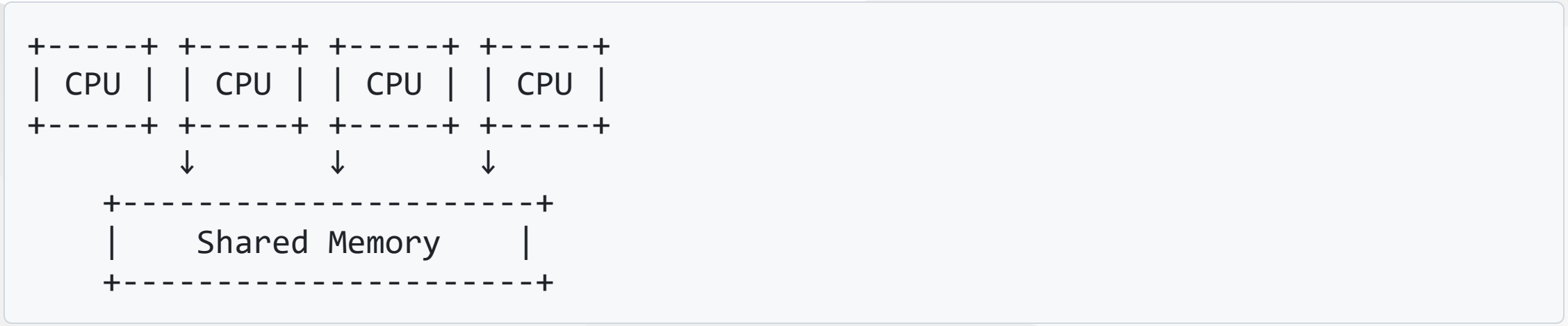
Flynn's Taxonomy (4/4)

MIMD Architecture and Example



Memory Architectures (1/5)

Shared Memory Overview



Memory Architectures (2/5)

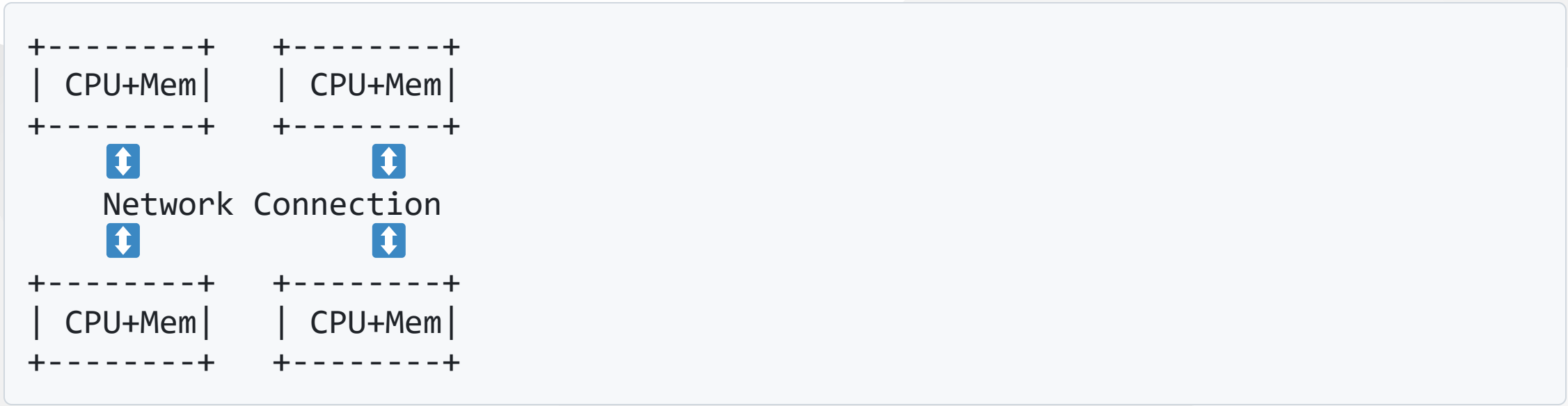
Shared Memory Example

```
// Basic OpenMP shared memory example
int shared_array[1000];

#pragma omp parallel for
for(int i = 0; i < 1000; i++) {
    shared_array[i] = heavy_computation(i);
}
```

Memory Architectures (3/5)

Distributed Memory Overview



Memory Architectures (4/5)

Distributed Memory Example (Part 1)

```
// MPI distributed memory example
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

// Local computation
int local_result = compute_local_part(rank, size);
```

Memory Architectures (5/5)

Distributed Memory Example (Part 2)

```
// Gather results from all processes
int* all_results = NULL;
if(rank == 0) {
    all_results = new int[size];
}

MPI_Gather(&local_result, 1, MPI_INT,
           all_results, 1, MPI_INT,
           0, MPI_COMM_WORLD);
```

2. Performance Metrics

Speedup Analysis (1/4)

Theoretical Speedup

$$S(n) = \frac{T_1}{T_n}$$

where:

- T_1$$ is sequential execution time

Speedup Analysis (2/4)

Measurement Code

```
Timer t;

// Sequential version
t.start();
sequential_algorithm();
double t1 = t.stop();

// Parallel version
t.start();
parallel_algorithm();
double tn = t.stop();

double speedup = t1/tn;
printf("Speedup: %.2f\n", speedup);
```

Speedup Analysis (3/4)

Amdahl's Law

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

where:

- p$$ is the parallel portion

Speedup Analysis (4/4)

Amdahl's Law Implementation

```
double amdahl_speedup(double p, int n) {
    return 1.0 / ((1-p) + p/n);
}

// Calculate theoretical speedups
for(int n = 1; n <= 16; n *= 2) {
    printf("Processors: %d, Max Speedup: %.2f\n",
           n, amdahl_speedup(0.95, n));
}
```

3. Parallel Algorithm Design

Decomposition Strategies (1/4)

Data Decomposition

```
// Matrix multiplication with data decomposition
void parallel_matrix_multiply(const Matrix& A,
                              const Matrix& B,
                              Matrix& C) {
    #pragma omp parallel for collapse(2)
    for(int i = 0; i < A.rows; i++) {
        for(int j = 0; j < B.cols; j++) {
            double sum = 0;
            for(int k = 0; k < A.cols; k++) {
                sum += A(i,k) * B(k,j);
            }
            C(i,j) = sum;
        }
    }
}
```

Decomposition Strategies (2/4)

Task Decomposition

```
// Pipeline processing example
class Pipeline {
    std::queue<Task> stage1_queue, stage2_queue;

public:
    void run() {
        #pragma omp parallel sections
        {
            #pragma omp section
            stage1_worker();

            #pragma omp section
            stage2_worker();

            #pragma omp section
            stage3_worker();
        }
    }

private:
    void stage1_worker() {
        while(has_input()) {
            Task t = read_input();
            stage1_queue.push(t);
        }
    }

    // Similar implementations for other stages...
};
```

Decomposition Strategies (3/4)

Load Balancing

```
// Dynamic load balancing example
void dynamic_load_balance(std::vector<Task>& tasks) {
    #pragma omp parallel
    {
        #pragma omp for schedule(dynamic)
        for(size_t i = 0; i < tasks.size(); i++) {
            process_task(tasks[i]);
        }
    }
}

// Work stealing implementation
class WorkStealingQueue {
    std::deque<Task> tasks;
    std::mutex mtx;

public:
    void push(Task t) {
        std::lock_guard<std::mutex> lock(mtx);
        tasks.push_back(std::move(t));
    }

    bool steal(Task& t) {
        std::lock_guard<std::mutex> lock(mtx);
        if(tasks.empty()) return false;
        t = std::move(tasks.front());
        tasks.pop_front();
        return true;
    }
};
```

Decomposition Strategies (4/4)

Communication Patterns

```
// Collective communication example (MPI)
void parallel_sum(std::vector<int>& local_data) {
    int local_sum = std::accumulate(local_data.begin(),
                                    local_data.end(), 0);
    int global_sum;

    MPI_Allreduce(&local_sum, &global_sum, 1, MPI_INT,
                  MPI_SUM, MPI_COMM_WORLD);

    printf("Local sum: %d, Global sum: %d\n",
           local_sum, global_sum);
}
```

Shared Memory Programming (1/3)

OpenMP Basics

```
// Basic parallel regions
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    printf("Hello from thread %d\n", tid);
}

// Work sharing
#pragma omp parallel for
for(int i = 0; i < N; i++) {
    heavy_computation(i);
}

// Synchronization
#pragma omp parallel
{
    #pragma omp critical
    {
        // Critical section
    }

    #pragma omp barrier
    // All threads synchronize here
```

Shared Memory Programming (2/3)

Advanced OpenMP Features

```
// Task parallelism
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        long_running_task1();

        #pragma omp task
        long_running_task2();
    }
}

// Nested parallelism
void nested_parallel() {
    #pragma omp parallel num_threads(2)
    {
        #pragma omp parallel num_threads(2)
        {
            int tid = omp_get_thread_num();
            int team = omp_get_team_num();
            printf("Team %d, Thread %d\n", team, tid);
        }
    }
}
```

Shared Memory Programming (3/3)

Performance Considerations

```
// False sharing example
struct PaddedCounter {
    int value;
    char padding[60]; // Prevent false sharing
};

void increment_counters() {
    PaddedCounter counters[NUM_THREADS];

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

5. Performance Analysis & Optimization

Profiling Tools (1/3)

Using Intel VTune

```
// Code instrumentation example
#include <ittnotify.h>

void analyze_performance() {
    __itt_domain* domain = __itt_domain_create("MyDomain");
    __itt_string_handle* task = __itt_string_handle_create("MyTask");

    __itt_task_begin(domain, __itt_null, __itt_null, task);
    heavy_computation();
    __itt_task_end(domain);
}
```

Profiling Tools (2/3)

Custom Performance Metrics

```
class PerformanceMetrics {
    std::chrono::high_resolution_clock::time_point start;
    std::string name;

public:
    PerformanceMetrics(const std::string& n)
        : name(n), start(std::chrono::high_resolution_clock::now()) {}

    ~PerformanceMetrics() {
        auto end = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::microseconds>
                        (end - start).count();
        printf("%s took %ld microseconds\n", name.c_str(), duration);
    }
};
```

Profiling Tools (3/3)

Memory Access Analysis

```
// Cache-friendly vs cache-unfriendly access
void analyze_memory_access() {
    const int SIZE = 1024 * 1024;
    int* arr = new int[SIZE];

    // Sequential access
    PerformanceMetrics m1("Sequential");
    for(int i = 0; i < SIZE; i++) {
        arr[i] = i;
    }

    // Random access
    PerformanceMetrics m2("Random");
    for(int i = 0; i < SIZE; i++) {
        arr[(i * 16) % SIZE] = i;
    }

    delete[] arr;
}
```

Scientific Computing (1/3)

N-Body Simulation

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

void simulate_n_body(std::vector<Particle>& particles) {
    #pragma omp parallel for
    for(size_t i = 0; i < particles.size(); i++) {
        for(size_t j = 0; j < particles.size(); j++) {
            if(i != j) {
                update_velocity(particles[i], particles[j]);
            }
        }
    }

    #pragma omp parallel for
    for(auto& p : particles) {
        update_position(p);
    }
```

Scientific Computing (2/3)

Matrix Operations

```
// Parallel matrix multiplication with blocking
void block_matrix_multiply(const Matrix& A,
                           const Matrix& B,
                           Matrix& C,
                           int block_size) {
    #pragma omp parallel for collapse(2)
    for(int i = 0; i < A.rows; i += block_size) {
        for(int j = 0; j < B.cols; j += block_size) {
            for(int k = 0; k < A.cols; k += block_size) {
                multiply_block(A, B, C, i, j, k, block_size);
            }
        }
    }
}
```

Scientific Computing (3/3)

Performance Analysis

```
void analyze_block_sizes() {
    Matrix A(1024, 1024), B(1024, 1024), C(1024, 1024);

    std::vector<int> block_sizes = {8, 16, 32, 64, 128};
    for(int block_size : block_sizes) {
        Timer t;
        block_matrix_multiply(A, B, C, block_size);
        printf("Block size %d: %.2f seconds\n",
               block_size, t.elapsed());
    }
}
```

7. Parallel Programming Paradigms

SPMD Pattern (1/3)

```
// Single Program Multiple Data Example
void spmd_example() {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Same program, different data portions
    std::vector<double> local_data = get_local_data(rank, size);
    double local_sum = std::accumulate(local_data.begin(),
                                       local_data.end(), 0.0);

    // Combine results
    double global_sum;
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_DOUBLE,
               MPI_SUM, 0, MPI_COMM_WORLD);
}
```

Master/Worker Pattern (2/3)

```
// Master-Worker Implementation
class TaskPool {
    std::queue<Task> tasks;
    std::mutex mtx;
    std::condition_variable cv;
    bool done = false;

public:
    void master_function() {
        while(has_more_tasks()) {
            Task t = generate_task();
            {
                std::lock_guard<std::mutex> lock(mtx);
                tasks.push(t);
            }
            cv.notify_one();
        }
        done = true;
        cv.notify_all();
    }

    void worker_function(int id) {
        while(true) {
            Task t;
            {
                std::unique_lock<std::mutex> lock(mtx);
                cv.wait(lock, [this]() {
                    return !tasks.empty() || done;
                });
                if(tasks.empty() && done) break;
                t = tasks.front();
                tasks.pop();
            }
            process_task(t, id);
        }
    }
};
```

Pipeline Pattern (3/3)

```
// Pipeline Pattern with OpenMP
template<typename T>
class Pipeline {
    std::queue<T> queue1, queue2;
    std::mutex mtx1, mtx2;
    std::condition_variable cv1, cv2;
    bool done = false;

public:
    void run_pipeline() {
        #pragma omp parallel sections
        {
            #pragma omp section
            stage1_producer();

            #pragma omp section
            stage2_processor();

            #pragma omp section
            stage3_consumer();
        }
    }

private:
    void stage1_producer() {
        while(has_input()) {
            T data = read_input();
            {
                std::lock_guard<std::mutex> lock(mtx1);
                queue1.push(data);
            }
            cv1.notify_one();
        }
    }

    // Similar implementations for other stages...
};
```

8. Modern CPU Architectures

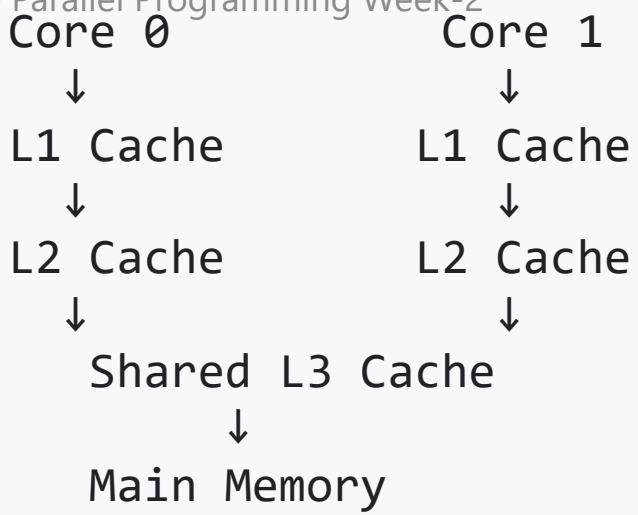
NUMA Architecture (1/3)

```
   CPU0     CPU1        CPU2     CPU3
    ↓        ↓           ↓        ↓
  Memory0  Memory1    Memory2  Memory3
    ↓        ↓           ↓        ↓
    ←--- Interconnect Network --->
```

```
// NUMA-aware allocation
void numa_allocation() {
    #pragma omp parallel
    {
        int numa_node = get_numa_node();
        void* local_mem = numa_alloc_onnode(size, numa_node);

        // Process data locally
        process_local_data(local_mem);

        numa_free(local_mem, size);
    }
}
```




```cpp
// Cache-conscious programming
void cache_optimization() {
    const int CACHE_LINE = 64;
    struct alignas(CACHE_LINE) CacheAligned {
        double data[8]; // 64 bytes
    };

    std::vector<CacheAligned> array(1000);

    #pragma omp parallel for
    for(int i = 0; i < array.size(); i++) {
        process_aligned_data(array[i]);
    }
}
```

Instruction Level Parallelism (3/3)

```
// Loop unrolling example
void optimize_loop() {
    const int N = 1000000;
    float a[N], b[N], c[N];

    // Original loop
    for(int i = 0; i < N; i++) {
        c[i] = a[i] + b[i];
    }

    // Unrolled loop
    for(int i = 0; i < N; i += 4) {
        c[i] = a[i] + b[i];
        c[i+1] = a[i+1] + b[i+1];
        c[i+2] = a[i+2] + b[i+2];
        c[i+3] = a[i+3] + b[i+3];
    }
}
```

9. Advanced Algorithm Analysis

Work-Time Analysis (1/3)

```
T_∞ = \text{Critical Path Length}
T_1 = \text{Total Work}
P = \text{Number of Processors}
T_P ≥ \max(T_∞, T_1/P)
```

Example Analysis:

```
// Parallel reduction analysis
void analyze_reduction() {
    const int N = 1024;
    int depth = log2(N);  // T_∞
    int total_ops = N-1;  // T_1

    printf("Critical path length: %d\n", depth);
    printf("Total work: %d\n", total_ops);
    printf("Theoretical min time with P processors: %f\n",
           max(depth, total_ops/P));
}
```

Isoefficiency Analysis (2/3)

```
// Isoefficiency measurement
void measure_isoefficiency() {
    std::vector<int> problem_sizes = {1000, 2000, 4000, 8000};
    std::vector<int> processor_counts = {1, 2, 4, 8, 16};

    for(int N : problem_sizes) {
        for(int P : processor_counts) {
            Timer t;
            parallel_algorithm(N, P);
            double Tp = t.elapsed();

            double efficiency = T1/(P * Tp);
            printf("N=%d, P=%d, Efficiency=%.2f\n",
                   N, P, efficiency);
        }
    }
}
```

Critical Path Analysis (3/3)

```
// Task graph analysis
class TaskGraph {
    struct Task {
        int id;
        std::vector<int> dependencies;
        int cost;
    };

    std::vector<Task> tasks;

public:
    int calculate_critical_path() {
        std::vector<int> earliest_start(tasks.size(), 0);

        for(const Task& t : tasks) {
            int max_dep_time = 0;
            for(int dep : t.dependencies) {
                max_dep_time = std::max(max_dep_time,
                    earliest_start[dep] + tasks[dep].cost);
            }
            earliest_start[t.id] = max_dep_time;
        }

        return *std::max_element(earliest_start.begin(),
                                 earliest_start.end());
    }
};
```

10. Performance Modeling

Roofline Model (1/3)

```
// Roofline model analysis
struct RooflineParams {
    double peak_performance;    // FLOPS
    double memory_bandwidth;    // bytes/s
    double operational_intensity; // FLOPS/byte
};

double predict_performance(const RooflineParams& params) {
    return std::min(params.peak_performance,
                    params.memory_bandwidth *
                    params.operational_intensity);
}
```

LogP Model (2/3)

```
// LogP model parameters
struct LogPParams {
    double L;  // Latency
    double o;  // Overhead
    double g;  // Gap
    int P;     // Processors
};

double estimate_communication_time(const LogPParams& params,
                                   int message_size) {
    return params.L + 2 * params.o +
           (message_size - 1) * params.g;
}
```

BSP Model (3/3)

```
// Bulk Synchronous Parallel model
class BSPComputation {
    struct SuperStep {
        double computation_time;
        double communication_volume;
        int synchronization_cost;
    };

    std::vector<SuperStep> steps;

public:
    double estimate_total_time(int P, double g, double L) {
        double total = 0;
        for(const auto& step : steps) {
            total += step.computation_time +
                     g * step.communication_volume +
                     L;  // barrier synchronization
        }
        return total;
    }
};
```

11. Advanced Optimization Techniques

Vectorization (1/4)

```
// Auto-vectorization example
void vectorized_operation(float* a, float* b,
                          float* c, int n) {
    // Hint for vectorization
    #pragma omp simd
    for(int i = 0; i < n; i++) {
        c[i] = std::sqrt(a[i] * a[i] + b[i] * b[i]);
    }
}

// Explicit vectorization
void explicit_vector() {
    __m256 a = _mm256_set1_ps(1.0f);
    __m256 b = _mm256_set1_ps(2.0f);
    __m256 c = _mm256_add_ps(a, b);
}
```

Loop Transformations (2/4)

```
// Loop interchange
void matrix_transform() {
    int A[1000][1000];

    // Original (cache-unfriendly)
    for(int j = 0; j < 1000; j++)
        for(int i = 0; i < 1000; i++)
            A[i][j] = compute(i,j);

    // Transformed (cache-friendly)
    for(int i = 0; i < 1000; i++)
        for(int j = 0; j < 1000; j++)
            A[i][j] = compute(i,j);
}
```

Memory Coalescing (3/4)

```
// Memory coalescing for GPU
struct SOA {
    float* x;
    float* y;
    float* z;
};

struct AOS {
    struct Point {
        float x, y, z;
    };
    Point* points;
};

void coalesced_access() {
    SOA soa;
    // Coalesced access
    #pragma omp target teams distribute parallel for
    for(int i = 0; i < N; i++) {
        soa.x[i] = compute_x(i);
        soa.y[i] = compute_y(i);
        soa.z[i] = compute_z(i);
    }
}
```

Prefetching (4/4)

```
// Software prefetching
void prefetch_example(int* data, int size) {
    const int PREFETCH_DISTANCE = 16;

    for(int i = 0; i < size; i++) {
        // Prefetch future data
        __builtin_prefetch(&data[i + PREFETCH_DISTANCE]);
        process(data[i]);
    }
}
```

Project 1: Advanced Matrix Operations

1. Implement parallel matrix operations with:
    - Cache blocking
    - SIMD optimization
    - NUMA awareness
2. Performance Analysis:
    - Roofline model analysis
    - Cache miss rates
    - Memory bandwidth utilization
3. Documentation:
    - Implementation details
    - Performance analysis
    - Optimization strategies

Next Week Preview

We will cover:

- OpenMP in detail
- Parallel regions and constructs
- Data sharing and synchronization
- Advanced OpenMP features

12. Parallel Design Patterns

Map-Reduce Pattern (1/3)

```
// Map-Reduce implementation
template<typename T, typename MapFn, typename ReduceFn>
T parallel_map_reduce(const std::vector<T>& data,
                      MapFn map_fn,
                      ReduceFn reduce_fn,
                      T initial_value) {
    std::vector<T> mapped_data(data.size());

    // Map phase
    #pragma omp parallel for
    for(size_t i = 0; i < data.size(); i++) {
        mapped_data[i] = map_fn(data[i]);
    }

    // Reduce phase
    T result = initial_value;
    #pragma omp parallel for reduction(reduce_fn:result)
    for(size_t i = 0; i < mapped_data.size(); i++) {
        result = reduce_fn(result, mapped_data[i]);
    }

    return result;
}

// Usage example
void map_reduce_example() {
    std::vector<int> data = {1, 2, 3, 4, 5, 6, 7, 8};

    auto square = [](int x) { return x * x; };
    auto sum = [](int a, int b) { return a + b; };

    int result = parallel_map_reduce(data, square, sum, 0);
    printf("Sum of squares: %d\n", result);
}
```

Fork-Join Pattern (2/3)

```
// Fork-Join pattern with recursive task decomposition
template<typename T>
T parallel_divide_conquer(T* data, int start, int end,
                          int threshold) {
    int length = end - start;
    if(length <= threshold) {
        return sequential_solve(data, start, end);
    }

    T left_result, right_result;
    int mid = start + length/2;

    #pragma omp task shared(left_result)
    left_result = parallel_divide_conquer(data, start, mid,
                                          threshold);

    #pragma omp task shared(right_result)
    right_result = parallel_divide_conquer(data, mid, end,
                                           threshold);

    #pragma omp taskwait
    return combine_results(left_result, right_result);
}
```

Wavefront Pattern (3/3)

```
// Wavefront pattern implementation
void wavefront_computation(Matrix& matrix, int N) {
    for(int wave = 0; wave < 2*N-1; wave++) {
        #pragma omp parallel for
        for(int i = max(0, wave-N+1);
            i <= min(wave, N-1); i++) {
            int j = wave - i;
            if(j < N) {
                compute_cell(matrix, i, j);
            }
        }
    }
}

// Example usage for matrix computation
void compute_cell(Matrix& matrix, int i, int j) {
    matrix(i,j) = matrix(i-1,j) +
                  matrix(i,j-1) +
                  matrix(i-1,j-1);
}
```

13. Parallel Data Structures

Lock-free Queue (1/4)

```
template<typename T>
class LockFreeQueue {
    struct Node {
        T data;
        std::atomic<Node*> next;
    };

    std::atomic<Node*> head;
    std::atomic<Node*> tail;

public:
    void enqueue(T value) {
        Node* new_node = new Node{value, nullptr};

        while(true) {
            Node* last = tail.load();
            Node* next = last->next.load();

            if(last == tail.load()) {
                if(next == nullptr) {
                    if(last->next.compare_exchange_weak(next, new_node)) {
                        tail.compare_exchange_weak(last, new_node);
                        return;
                    }
                } else {
                    tail.compare_exchange_weak(last, next);
                }
            }
        }
    }

    bool dequeue(T& result) {
        while(true) {
            Node* first = head.load();
            Node* last = tail.load();
            Node* next = first->next.load();

            if(first == head.load()) {
                if(first == last) {
                    if(next == nullptr) {
                        return false;
                    }
                    tail.compare_exchange_weak(last, next);
                } else {
                    result = next->data;
                    if(head.compare_exchange_weak(first, next)) {
                        delete first;
                        return true;
                    }
                }
            }
        }
    }
};
```

Concurrent Hash Map (2/4)

```
template<typename K, typename V>
class ConcurrentHashMap {
    struct Bucket {
        std::mutex mtx;
        std::unordered_map<K,V> data;
    };

    std::vector<Bucket> buckets;
    size_t num_buckets;

public:
    ConcurrentHashMap(size_t n) : num_buckets(n) {
        buckets.resize(n);
    }

    void insert(const K& key, const V& value) {
        size_t bucket_idx = std::hash<K>{}(key) % num_buckets;
        std::lock_guard<std::mutex> lock(buckets[bucket_idx].mtx);
        buckets[bucket_idx].data[key] = value;
    }

    bool find(const K& key, V& value) {
        size_t bucket_idx = std::hash<K>{}(key) % num_buckets;
        std::lock_guard<std::mutex> lock(buckets[bucket_idx].mtx);
        auto it = buckets[bucket_idx].data.find(key);
        if(it != buckets[bucket_idx].data.end()) {
            value = it->second;
            return true;
        }
        return false;
    }
};
```

Thread-safe Vector (3/4)

```
template<typename T>
class ThreadSafeVector {
    std::vector<T> data;
    mutable std::shared_mutex mutex;

public:
    void push_back(const T& value) {
        std::unique_lock lock(mutex);
        data.push_back(value);
    }

    T at(size_t index) const {
        std::shared_lock lock(mutex);
        return data.at(index);
    }

    void update(size_t index, const T& value) {
        std::unique_lock lock(mutex);
        data[index] = value;
    }

    size_t size() const {
        std::shared_lock lock(mutex);
        return data.size();
    }

    // Atomic operation example
    void atomic_update(size_t index,
                       std::function<void(T&)> update_fn) {
        std::unique_lock lock(mutex);
        update_fn(data[index]);
    }
};
```

Lock-free Stack (4/4)

```
template<typename T>
class LockFreeStack {
    struct Node {
        T data;
        std::atomic<Node*> next;
    };

    std::atomic<Node*> head;

public:
    void push(T value) {
        Node* new_node = new Node{value};

        do {
            new_node->next = head.load();
        } while(!head.compare_exchange_weak(new_node->next,
                                            new_node));
    }

    bool pop(T& result) {
        Node* old_head;

        do {
            old_head = head.load();
            if(old_head == nullptr) {
                return false;
            }
        } while(!head.compare_exchange_weak(old_head,
                                            old_head->next));

        result = old_head->data;
        delete old_head;
        return true;
    }
};
```

14. Parallel Debugging Techniques

Race Condition Detection (1/3)

```
// Thread Sanitizer usage example
void race_condition_example() {
    int shared_var = 0;

    #pragma omp parallel for
    for(int i = 0; i < 100; i++) {
        // Race condition here
        shared_var++;
    }

    // Fixed version
    #pragma omp parallel for reduction(+:shared_var)
    for(int i = 0; i < 100; i++) {
        shared_var++;
    }
}

// Custom race detector
class RaceDetector {
    std::atomic<int> access_count{0};
    std::atomic<std::thread::id> last_writer;

public:
    void on_read(void* addr) {
        access_count++;
        // Log read access
    }

    void on_write(void* addr) {
        access_count++;
        last_writer = std::this_thread::get_id();
        // Log write access
    }

    void check_race() {
        if(access_count > 1) {
            printf("Potential race detected!\n");
        }
    }
};
```

Deadlock Detection (2/3)

```
// Deadlock detection implementation
class DeadlockDetector {
    struct LockInfo {
        std::thread::id thread_id;
        void* lock_addr;
        std::chrono::system_clock::time_point acquire_time;
    };

    std::map<void*, std::vector<LockInfo>> lock_graph;
    std::mutex graph_mutex;

public:
    void on_lock_attempt(void* lock_addr) {
        std::lock_guard<std::mutex> guard(graph_mutex);

        auto& info = lock_graph[lock_addr];
        info.push_back({
            std::this_thread::get_id(),
            lock_addr,
            std::chrono::system_clock::now()
        });

        detect_cycle();
    }

    void on_lock_acquire(void* lock_addr) {
        // Update lock status
    }

    void on_lock_release(void* lock_addr) {
        std::lock_guard<std::mutex> guard(graph_mutex);
        lock_graph[lock_addr].clear();
    }

private:
    bool detect_cycle() {
        // Implement cycle detection in lock graph
        return false;
    }
};
```

Memory Leak Detection (3/3)

```
// Memory leak detector
class MemoryLeakDetector {
    struct Allocation {
        void* ptr;
        size_t size;
        std::string file;
        int line;
        std::thread::id thread_id;
    };

    std::map<void*, Allocation> allocations;
    std::mutex mtx;

public:
    void on_allocation(void* ptr, size_t size,
                       const char* file, int line) {
        std::lock_guard<std::mutex> guard(mtx);
        allocations[ptr] = {
            ptr,
            size,
            file,
            line,
            std::this_thread::get_id()
        };
    }

    void on_deallocation(void* ptr) {
        std::lock_guard<std::mutex> guard(mtx);
        allocations.erase(ptr);
    }

    void report_leaks() {
        std::lock_guard<std::mutex> guard(mtx);
        for(const auto& [ptr, alloc] : allocations) {
            printf("Leak: %p, size: %zu, file: %s, line: %d\n",
                   ptr, alloc.size, alloc.file.c_str(),
                   alloc.line);
        }
    }
};
```

15. Energy Efficiency in Parallel Computing

Power-Aware Computing (1/2)

```
// Power monitoring and management
class PowerMonitor {
    struct CoreStats {
        int frequency;
        double temperature;
        double power_consumption;
    };

    std::vector<CoreStats> core_stats;

public:
    void monitor_power_consumption() {
        #pragma omp parallel
        {
            int tid = omp_get_thread_num();

            while(true) {
                update_core_stats(tid);

                if(core_stats[tid].temperature > THRESHOLD) {
                    reduce_frequency(tid);
                }

                std::this_thread::sleep_for(
                    std::chrono::milliseconds(100));
            }
        }
    }

private:
    void update_core_stats(int core_id) {
        // Read hardware counters
        // Update statistics
    }

    void reduce_frequency(int core_id) {
        // Implement frequency scaling
    }
};
```

Energy Efficiency Metrics (2/2)

```cpp
// Energy efficiency calculation
struct EnergyMetrics {
    double energy_consumption;  // Joules
    double execution_time;      // Seconds
    double performance;         // FLOPS

    double calculate_efficiency() {
        return performance / energy_consumption;
    }
};

class EnergyProfiler {
    std::vector<EnergyMetrics> measurements;

public:
    EnergyMetrics profile_algorithm(
        std::function<void()> algorithm) {

        auto start_energy = measure_energy();
        auto start_time = std::chrono::high_resolution_clock::now();

        algorithm();

        auto end_time = std::chrono::high_resolution_clock::now();
        auto end_energy = measure_energy();

        return {
            end_energy - start_energy,
            std::chrono::duration<double>(
                end_time - start_time).count(),
            measure_performance()
        };
    }

private:
    double measure_energy() {
        // Read energy counters
        return 0.0;
    }

    double measure_performance() {
        // Calculate FLOPS
        return 0.0;
    }
};
```

1. Implement parallel algorithms with energy monitoring:
    - Matrix multiplication
    - Sorting algorithms
    - Graph algorithms
2. Energy Analysis:
    - Power consumption measurement
    - Performance per watt analysis
    - Temperature monitoring
3. Optimization Strategies:
    - Frequency scaling
    - Load balancing
    - Task scheduling
4. Documentation:

Next Week Preview

We will cover:

- OpenMP Advanced Features
- Task Parallelism
- Nested Parallelism
- SIMD Operations

16. Advanced Data Structures and Algorithms

Parallel Search Trees (1/4)

```
template<typename T>
class ParallelBST {
    struct Node {
        T data;
        std::atomic<Node*> left, right;
        std::atomic<bool> marked;  // For deletion
    };

    std::atomic<Node*> root;

public:
    bool insert(const T& value) {
        Node* new_node = new Node{value, nullptr, nullptr, false};

        while(true) {
            Node* current = root.load();
            if(!current) {
                if(root.compare_exchange_strong(current, new_node)) {
                    return true;
                }
                continue;
            }
        }
    }
};
```

Parallel Search Trees (2/4)

```
template<typename T>
class ParallelBST {
    struct Node {
        T data;
        std::atomic<Node*> left, right;
        std::atomic<bool> marked;  // For deletion
    };

    std::atomic<Node*> root;

public:
    bool insert(const T& value) {
        Node* new_node = new Node{value, nullptr, nullptr, false};

        while(true) {
            Node* current = root.load();
            if(!current) {
                if(root.compare_exchange_strong(current, new_node)) {
                    return true;
                }
                continue;
            }
        }
    }
};
```

Parallel Search Trees (3/4)

```
template<typename T>
class ParallelBST {
    struct Node {
        T data;
        std::atomic<Node*> left, right;
        std::atomic<bool> marked;  // For deletion
    };

    std::atomic<Node*> root;

public:
    bool insert(const T& value) {
        Node* new_node = new Node{value, nullptr, nullptr, false};

        while(true) {
            Node* current = root.load();
            if(!current) {
                if(root.compare_exchange_strong(current, new_node)) {
                    return true;
                }
                continue;
            }
        }
    }
};
```

Parallel Search Trees (4/4)

```
template<typename T>
class ParallelBST {
    struct Node {
        T data;
        std::atomic<Node*> left, right;
        std::atomic<bool> marked;  // For deletion
    };

    std::atomic<Node*> root;

public:
    bool insert(const T& value) {
        Node* new_node = new Node{value, nullptr, nullptr, false};

        while(true) {
            Node* current = root.load();
            if(!current) {
                if(root.compare_exchange_strong(current, new_node)) {
                    return true;
                }
                continue;
            }
        }
    }
};
```

$$End - Of - Week - 2$$