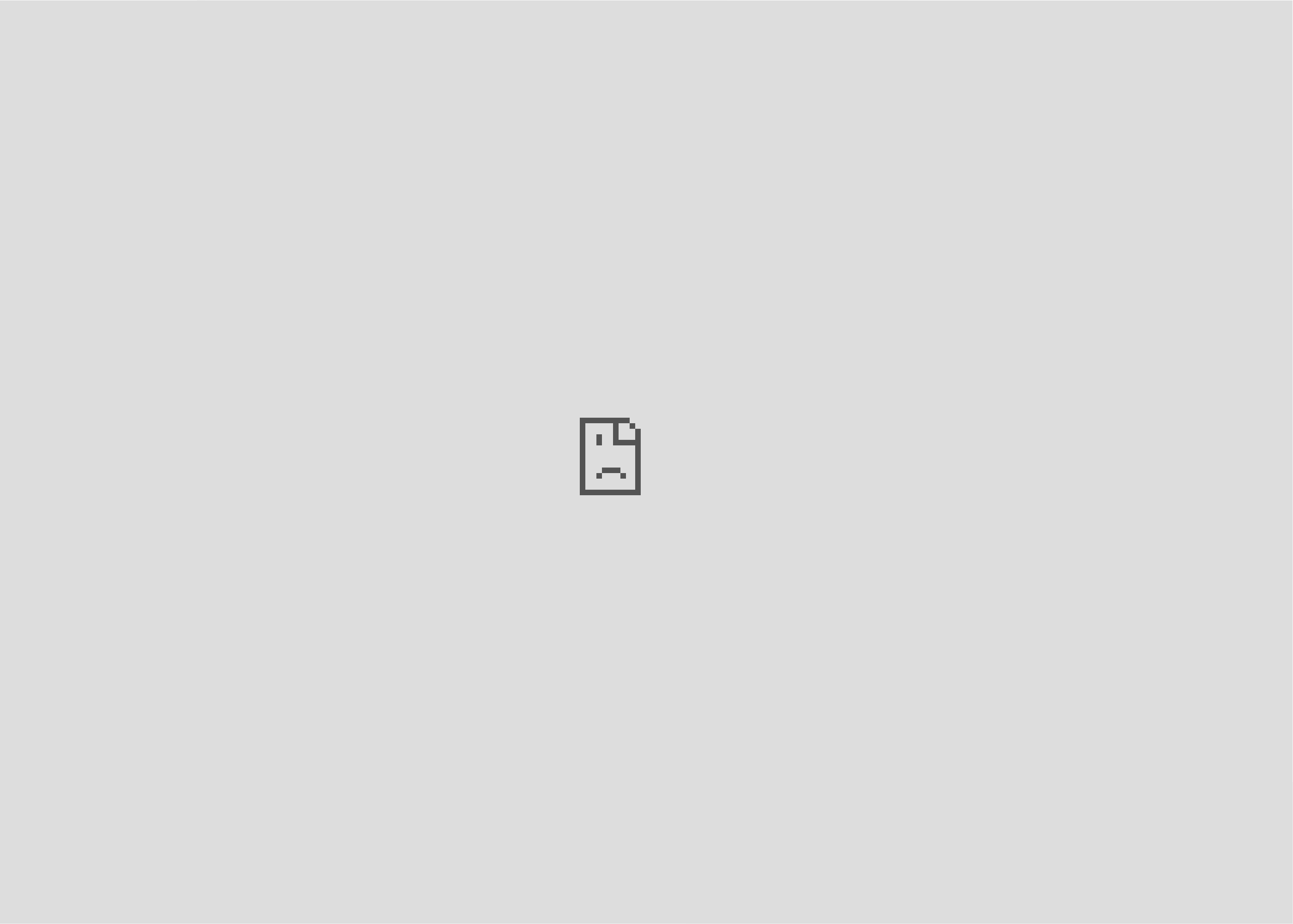
CEN310 Parallel Programming

Week-6

Performance Optimization

Download

- PDF
- DOC
- SLIDE
- PPTX



Outline (1/4)

1. Performance Analysis Tools
    - Profiling Tools Overview
    - Hardware Performance Counters
    - Performance Metrics
    - Bottleneck Detection
    - Benchmarking Techniques
2. Memory Optimization
    - Cache Optimization
    - Memory Access Patterns
    - Data Layout Strategies
    - False Sharing Prevention
    - Memory Bandwidth Utilization

Outline (2/4)

3. Algorithm Optimization
    - Load Balancing Techniques
    - Work Distribution Strategies
    - Communication Pattern Optimization
    - Synchronization Overhead Reduction
    - Scalability Analysis Methods
4. Parallel Pattern Optimization
    - Map-Reduce Optimization
    - Pipeline Pattern Tuning
    - Task Parallelism Efficiency
    - Data Parallelism Strategies
    - Hybrid Approaches

Outline (3/4)

5. Advanced Optimization Techniques
   - Vectorization Strategies
   - Loop Optimization Methods
   - Thread Affinity Management
   - Compiler Optimization Flags
   - Hardware-Specific Tuning
6. Tools and Frameworks
   - Intel VTune Profiler
   - Perf Tools
   - TAU Performance System
   - PAPI Interface
   - Custom Profiling Tools

Outline (4/4)

7. Best Practices
   - Performance Measurement
   - Optimization Workflow
   - Documentation Methods
   - Testing Strategies
   - Maintenance Considerations
8. Case Studies
   - Scientific Computing
   - Data Processing
   - Real-time Systems
   - High-Performance Computing

1. Performance Analysis Tools

Profiling Tools (1/4)

```
// Example using Intel VTune instrumentation
#include <ittnotify.h>

void profile_region_example() {
    // Create a domain for the measurement
    __itt_domain* domain = __itt_domain_create("MyDomain");

    // Create named tasks
    __itt_string_handle* task1 = __itt_string_handle_create("Task1");
    __itt_string_handle* task2 = __itt_string_handle_create("Task2");

    // Begin task measurement
    __itt_task_begin(domain, __itt_null, __itt_null, task1);

    // Measured code section 1
    #pragma omp parallel for
    for(int i = 0; i < N; i++) {
        heavy_computation1(i);
    }

    __itt_task_end(domain);

    // Begin another task
    __itt_task_begin(domain, __itt_null, __itt_null, task2);

    // Measured code section 2
    #pragma omp parallel for
    for(int i = 0; i < N; i++) {
        heavy_computation2(i);
    }

    __itt_task_end(domain);
}
```

Profiling Tools (2/4)

Hardware Counters

```c
#include <papi.h>

void hardware_counter_example() {
    int events[3] = {PAPI_TOT_CYC,    // Total cycles
                     PAPI_L1_DCM,      // L1 cache misses
                     PAPI_L2_DCM};     // L2 cache misses
    long long values[3];

    // Initialize PAPI
    if(PAPI_library_init(PAPI_VER_CURRENT) != PAPI_VER_CURRENT) {
        fprintf(stderr, "PAPI initialization failed\n");
        exit(1);
    }

    // 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);

    // Code to measure
    #pragma omp parallel for
    for(int i = 0; i < N; i++) {
        compute_intensive_task(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]);
}
```

Profiling Tools (3/4)

Custom Performance Metrics

```
class PerformanceMetrics {
private:
    struct Measurement {
        std::string name;
        double start_time;
        double total_time;
        int calls;
    };

    std::map<std::string, Measurement> measurements;

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

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

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

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

Profiling Tools (4/4)

Using Performance Metrics

```cpp
void demonstrate_metrics() {
    PerformanceMetrics metrics;
    const int N = 1000000;

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

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

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

    // Print report
    metrics.report();
}
```

2. Memory Optimization

Cache Optimization (1/4)

```
// Bad: Cache-unfriendly access
void bad_matrix_access(double** matrix, int N) {
    for(int j = 0; j < N; j++) {
        for(int i = 0; i < N; i++) {
            matrix[i][j] = compute(i, j);  // Column-major access
        }
    }
}

// Good: Cache-friendly access
void good_matrix_access(double** matrix, int N) {
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++) {
            matrix[i][j] = compute(i, j);  // Row-major access
        }
    }
}

// Better: Block-based access
void block_matrix_access(double** matrix, int N) {
    const int BLOCK_SIZE = 32;  // Tune for your cache size

    for(int i = 0; i < N; i += BLOCK_SIZE) {
        for(int j = 0; j < N; j += BLOCK_SIZE) {
            // Process block
            for(int bi = i; bi < std::min(i + BLOCK_SIZE, N); bi++) {
                for(int bj = j; bj < std::min(j + BLOCK_SIZE, N); bj++) {
                    matrix[bi][bj] = compute(bi, bj);
                }
            }
        }
    }
}
```

Cache Optimization (2/4)

Data Layout Strategies

```cpp
// Structure of Arrays (SoA)
struct ParticlesSoA {
    std::vector<double> x, y, z;
    std::vector<double> vx, vy, vz;

    void update(int i) {
        #pragma omp parallel for
        for(int i = 0; i < x.size(); i++) {
            x[i] += vx[i];
            y[i] += vy[i];
            z[i] += vz[i];
        }
    }
};

// Array of Structures (AoS)
struct ParticleAoS {
    struct Particle {
        double x, y, z;
        double vx, vy, vz;
    };

    std::vector<Particle> particles;

    void update(int i) {
        #pragma omp parallel for
        for(int i = 0; i < particles.size(); i++) {
            particles[i].x += particles[i].vx;
            particles[i].y += particles[i].vy;
            particles[i].z += particles[i].vz;
        }
    }
};
```

Cache Optimization (3/4)

False Sharing Prevention

```cpp
// Bad: False sharing prone
struct BadCounter {
    int count;  // Adjacent counters share cache line
};

// Good: Padded to prevent false sharing
struct GoodCounter {
    int count;
    char padding[60];  // Pad to cache line size
};

void parallel_counting() {
    const int NUM_THREADS = omp_get_max_threads();

    // Bad example
    std::vector<BadCounter> bad_counters(NUM_THREADS);

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        for(int i = 0; i < 1000000; i++) {
            bad_counters[tid].count++;  // False sharing!
        }
    }

    // Good example
    std::vector<GoodCounter> good_counters(NUM_THREADS);

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        for(int i = 0; i < 1000000; i++) {
            good_counters[tid].count++;  // No false sharing
        }
    }
}
```

Cache Optimization (4/4)

Memory Bandwidth Optimization

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

    // Bad: Multiple passes over data
    #pragma omp parallel for
    for(int i = 0; i < N; i++) {
        data[i] = std::sin(data[i]);
    }

    #pragma omp parallel for
    for(int i = 0; i < N; i++) {
        data[i] = std::sqrt(data[i]);
    }

    // Good: Single pass over data
    #pragma omp parallel for
    for(int i = 0; i < N; i++) {
        data[i] = std::sqrt(std::sin(data[i]));
    }

    // Better: Vectorized single pass
    #pragma omp parallel for simd
    for(int i = 0; i < N; i++) {
        data[i] = std::sqrt(std::sin(data[i]));
    }
}
```

3. Algorithm Optimization

Load Balancing (1/4)

```cpp
// Static load balancing
void static_distribution(const std::vector<Task>& tasks) {
    #pragma omp parallel for schedule(static)
    for(size_t i = 0; i < tasks.size(); i++) {
        process_task(tasks[i]);
    }
}

// Dynamic load balancing
void dynamic_distribution(const std::vector<Task>& tasks) {
    #pragma omp parallel for schedule(dynamic, 10)
    for(size_t i = 0; i < tasks.size(); i++) {
        process_task(tasks[i]);
    }
}

// Guided load balancing
void guided_distribution(const std::vector<Task>& tasks) {
    #pragma omp parallel for schedule(guided)
    for(size_t i = 0; i < tasks.size(); i++) {
        process_task(tasks[i]);
    }
}

// Custom load balancing
void custom_distribution(const std::vector<Task>& tasks) {
    const int NUM_THREADS = omp_get_max_threads();
    std::vector<std::vector<Task>> thread_tasks(NUM_THREADS);

    // Distribute tasks based on estimated cost
    for(size_t i = 0; i < tasks.size(); i++) {
        int cost = estimate_task_cost(tasks[i]);
        int target_thread = assign_to_thread(cost, NUM_THREADS);
        thread_tasks[target_thread].push_back(tasks[i]);
    }

    // Process distributed tasks
    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        for(const auto& task : thread_tasks[tid]) {
            process_task(task);
        }
    }
}
```

Load Balancing (2/4)

Work Stealing Implementation

```cpp
class WorkQueue {
private:
    std::deque<Task> tasks;
    std::mutex mtx;

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

    bool try_steal(Task& task) {
        std::lock_guard<std::mutex> lock(mtx);
        if(tasks.empty()) return false;

        task = std::move(tasks.front());
        tasks.pop_front();
        return true;
    }
};

void work_stealing_example() {
    const int NUM_THREADS = omp_get_max_threads();
    std::vector<WorkQueue> queues(NUM_THREADS);

    // Initialize work queues
    distribute_initial_tasks(queues);

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        Task task;

        while(work_remains()) {
            // Try to get work from own queue
            if(queues[tid].try_steal(task)) {
                process_task(task);
                continue;
            }

            // Try to steal work from other queues
            for(int i = 0; i < NUM_THREADS; i++) {
                if(i == tid) continue;

                if(queues[i].try_steal(task)) {
                    process_task(task);
                    break;
                }
            }
        }
    }
}
```

Load Balancing (3/4)

Task Pool Pattern

```
class TaskPool {
private:
    std::queue<Task> tasks;
    std::mutex mtx;
    std::condition_variable cv;
    bool done = false;

public:
    void add_task(const Task& task) {
        std::lock_guard<std::mutex> lock(mtx);
        tasks.push(task);
        cv.notify_one();
    }

    bool get_task(Task& task) {
        std::unique_lock<std::mutex> lock(mtx);
        cv.wait(lock, [this]() {
            return !tasks.empty() || done;
        });

        if(tasks.empty() && done) return false;

        task = std::move(tasks.front());
        tasks.pop();
        return true;
    }

    void finish() {
        std::lock_guard<std::mutex> lock(mtx);
        done = true;
        cv.notify_all();
    }
};

void task_pool_example() {
    TaskPool pool;

    // Producer thread
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            for(int i = 0; i < N; i++) {
                Task task = create_task(i);
                pool.add_task(task);
            }
            pool.finish();
        }

        // Consumer threads
        #pragma omp section
        {
            Task task;
            while(pool.get_task(task)) {
                process_task(task);
            }
        }
    }
}
```

Load Balancing (4/4)

Adaptive Load Balancing

```
class AdaptiveScheduler {
private:
    struct ThreadStats {
        double avg_task_time;
        int tasks_completed;
        std::mutex mtx;
    };

    std::vector<ThreadStats> stats;

public:
    AdaptiveScheduler(int num_threads)
        : stats(num_threads) {}

    void update_stats(int thread_id, double task_time) {
        auto& thread_stat = stats[thread_id];
        std::lock_guard<std::mutex> lock(thread_stat.mtx);

        thread_stat.avg_task_time =
            (thread_stat.avg_task_time * thread_stat.tasks_completed +
             task_time) / (thread_stat.tasks_completed + 1);
        thread_stat.tasks_completed++;
    }

    int get_chunk_size(int thread_id) {
        double thread_speed = 1.0 / stats[thread_id].avg_task_time;
        double total_speed = 0;

        for(const auto& stat : stats) {
            total_speed += 1.0 / stat.avg_task_time;
        }

        return static_cast<int>(
            BASE_CHUNK_SIZE * (thread_speed / total_speed));
    }
};

void adaptive_parallel_for(const std::vector<Task>& tasks) {
    AdaptiveScheduler scheduler(omp_get_max_threads());

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        int chunk_size = scheduler.get_chunk_size(tid);

        #pragma omp for schedule(dynamic, chunk_size)
        for(size_t i = 0; i < tasks.size(); i++) {
            double start = omp_get_wtime();
            process_task(tasks[i]);
            double time = omp_get_wtime() - start;

            scheduler.update_stats(tid, time);
        }
    }
```

4. Advanced Optimization Techniques

Vectorization (1/3)

```
// Enable vectorization with OpenMP SIMD
void vector_operation(float* a, float* b, float* c, int n) {
    #pragma omp parallel for simd
    for(int i = 0; i < n; i++) {
        c[i] = a[i] * b[i];
    }
}

// Manual vectorization with intrinsics
#include <immintrin.h>

void manual_vector_operation(float* a, float* b, float* c, int n) {
    // Process 8 elements at a time 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_mul_ps(va, vb);
        _mm256_store_ps(&c[i], vc);
    }

    // Handle remaining elements
    for(int i = (n/8)*8; i < n; i++) {
        c[i] = a[i] * b[i];
    }
}
```

Vectorization (2/3)

Vectorized Reduction

```
float vector_reduction(float* data, int n) {
    float sum = 0.0f;

    // Vectorized reduction
    #pragma omp parallel for simd reduction(+:sum)
    for(int i = 0; i < n; i++) {
        sum += data[i];
    }

    return sum;
}

// Manual vectorized reduction with AVX
float manual_vector_reduction(float* data, int n) {
    __m256 vsum = _mm256_setzero_ps();

    // Process 8 elements at a time
    for(int i = 0; i < n; i += 8) {
        __m256 v = _mm256_load_ps(&data[i]);
        vsum = _mm256_add_ps(vsum, v);
    }

    // Horizontal sum of vector
    float sum[8];
    _mm256_store_ps(sum, vsum);

    return sum[0] + sum[1] + sum[2] + sum[3] +
           sum[4] + sum[5] + sum[6] + sum[7];
}
```

Vectorization (3/3)

Vectorization Barriers

```
// Non-vectorizable due to dependencies
void dependency_example(float* a, int n) {
    for(int i = 1; i < n; i++) {
        a[i] = a[i-1] * 2.0f;  // Loop carried dependency
    }
}

// Vectorizable version
void vectorizable_example(float* a, float* b, int n) {
    #pragma omp parallel for simd
    for(int i = 0; i < n; i++) {
        b[i] = a[i] * 2.0f;  // No dependencies
    }
}

// Conditional vectorization
void conditional_vectorization(float* a, float* b, int n) {
    #pragma omp parallel for simd
    for(int i = 0; i < n; i++) {
        if(a[i] > 0.0f) {
            b[i] = a[i] * 2.0f;
        } else {
            b[i] = -a[i];
        }
    }
}
```

5. Performance Measurement

Benchmarking Framework (1/3)

```
class Benchmark {
private:
    std::string name;
    std::vector<double> timings;

public:
    Benchmark(const std::string& n) : name(n) {}

    template<typename Func>
    void run(Func&& func, int iterations = 10) {
        // Warmup
        func();

        // Actual measurements
        for(int i = 0; i < iterations; i++) {
            double start = omp_get_wtime();
            func();
            double end = omp_get_wtime();

            timings.push_back(end - start);
        }
    }

    void report() {
        // Calculate statistics
        double sum = 0.0;
        double min_time = timings[0];
        double max_time = timings[0];

        for(double t : timings) {
            sum += t;
            min_time = std::min(min_time, t);
            max_time = std::max(max_time, t);
        }

        double avg = sum / timings.size();

        // Calculate standard deviation
        double variance = 0.0;
        for(double t : timings) {
            variance += (t - avg) * (t - avg);
        }
        double stddev = std::sqrt(variance / timings.size());

        // Print report
        printf("\nBenchmark: %s\n", name.c_str());
        printf("Iterations: %zu\n", timings.size());
        printf("Average time: %.6f seconds\n", avg);
        printf("Min time: %.6f seconds\n", min_time);
        printf("Max time: %.6f seconds\n", max_time);
        printf("Std dev: %.6f seconds\n", stddev);
    }
};
```

Benchmarking Framework (2/3)

Using the Framework

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

    // Initialize data
    std::iota(data.begin(), data.end(), 0);

    // Benchmark different implementations
    Benchmark b1("Sequential Sum");
    b1.run([&]() {
        double sum = 0.0;
        for(int i = 0; i < N; i++) {
            sum += data[i];
        }
        return sum;
    });
    b1.report();

    Benchmark b2("Parallel Sum");
    b2.run([&]() {
        double sum = 0.0;
        #pragma omp parallel for reduction(+:sum)
        for(int i = 0; i < N; i++) {
            sum += data[i];
        }
        return sum;
    });
    b2.report();

    Benchmark b3("Vectorized Sum");
    b3.run([&]() {
        double sum = 0.0;
        #pragma omp parallel for simd reduction(+:sum)
        for(int i = 0; i < N; i++) {
            sum += data[i];
        }
        return sum;
    });
    b3.report();
}
```

Benchmarking Framework (3/3)

Performance Comparison

```
class PerformanceComparison {
private:
    struct Result {
        std::string name;
        double time;
        double speedup;
        double efficiency;
    };

    std::vector<Result> results;
    double baseline_time;

public:
    template<typename Func>
    void add_benchmark(const std::string& name,
                      Func&& func,
                      bool is_baseline = false) {
        Benchmark b(name);
        b.run(func);

        double time = b.get_average_time();
        if(is_baseline) {
            baseline_time = time;
        }

        results.push_back({
            name,
            time,
            baseline_time / time,
            (baseline_time / time) / omp_get_max_threads()
        });
    }

    void report() {
        printf("\nPerformance Comparison:\n");
        printf("%-20s %10s %10s %10s\n",
               "Implementation", "Time(s)", "Speedup", "Efficiency");

        for(const auto& r : results) {
            printf("%-20s %10.6f %10.2f %10.2f\n",
                   r.name.c_str(), r.time, r.speedup, r.efficiency);
        }
    }
};
```

6. Case Studies

Matrix Multiplication Optimization (1/3)

```
// Basic implementation
void matrix_multiply_basic(const Matrix& A,
                           const Matrix& B,
                           Matrix& C) {
    int N = A.rows();

    for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++) {
            double sum = 0.0;
            for(int k = 0; k < N; k++) {
                sum += A(i,k) * B(k,j);
            }
            C(i,j) = sum;
        }
    }
}

// Parallel implementation
void matrix_multiply_parallel(const Matrix& A,
                              const Matrix& B,
                              Matrix& C) {
    int N = A.rows();

    #pragma omp parallel for collapse(2)
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++) {
            double sum = 0.0;
            #pragma omp simd reduction(+:sum)
            for(int k = 0; k < N; k++) {
                sum += A(i,k) * B(k,j);
            }
            C(i,j) = sum;
        }
    }
}
```

Matrix Multiplication Optimization (2/3)

Cache-Optimized Version