Walchand College of Engineering, Sangli
Department of Computer Science and Engineering

Class: *Final Year (Computer Science and Engineering)*

Year: *2024-25* **Semester:** *1*

Course: *High Performance Computing Lab*

Practical No. 3

Omkar Rajesh Auti

Exam Seat No: 21510042

Title of practical:

Study and Implementation of schedule, nowait, reduction, ordered and collapse clauses

Problem Statement 1:

Analyse and implement a Parallel code for below program using OpenMP.

// C Program to find the minimum scalar product of two vectors (dot product)

Screenshots:

Code:

```
#include <iostream>
#include <vector>
#include <algorithm> // for sort
#include <omp.h> // OpenMP header

int main() {
// Example vectors
std::vector<int> a = {1, 3, 5, 2, 4};
std::vector<int> b = {7, 9, 2, 6, 8};

// Sort vector a in ascending order
std::sort(a.begin(), a.end());
// Sort vector b in descending order
std::sort(b.begin(), b.end(), std::greater<int>());

// Initialize the minimum scalar product for sequential and parallel runs
int min_scalar_product_seq = 0;
int min_scalar_product_par = 0;
```

```
// Measure time for the sequential computation
double start_time_seq = omp_get_wtime();

for (int i = 0; i < a.size(); ++i) {
min_scalar_product_seq += a[i] * b[i];
}

double end_time_seq = omp_get_wtime();
double time_seq_ms = (end_time_seq - start_time_seq) * 1000; // Convert to milliseconds

std::cout << "Minimum Scalar Product (Sequential): " << min_scalar_product_seq << std::endl;
std::cout << "Time taken (Sequential): " << time_seq_ms << " milliseconds" << std::endl;

// Measure time for the parallel computation
double start_time_par = omp_get_wtime();

#pragma omp parallel for reduction(+:min_scalar_product_par)
for (int i = 0; i < a.size(); ++i) {
int thread_id = omp_get_thread_num(); // Get the ID of the current thread
min_scalar_product_par += a[i] * b[i];
std::cout << "Thread " << thread_id << " processed element " << i << ": a[" << i << "] * b[" << i <<
"] = "
<< a[i] << " * " << b[i] << " = " << a[i] * b[i] << std::endl;
}

double end_time_par = omp_get_wtime();
double time_par_ms = (end_time_par - start_time_par) * 1000; // Convert to milliseconds

std::cout << "Minimum Scalar Product (Parallel): " << min_scalar_product_par << std::endl;
std::cout << "Time taken (Parallel): " << time_par_ms << " milliseconds" << std::endl;

return 0;
}
```

Output:

```
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment03$ g++ -fopenmp -o c 01_c.cpp
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment03$ ./c
Minimum Scalar Product (Sequential): 80
Time taken (Sequential): 0.000293 milliseconds
Thread 1 processed element 3: a[3] * b[3] = 4 * 6 = 24
Thread 1 processed element 4: a[4] * b[4] = 5 * 2 = 10
Thread 0 processed element 0: a[0] * b[0] = 1 * 9 = 9
Thread 0 processed element 1: a[1] * b[1] = 2 * 8 = 16
Thread 0 processed element 2: a[2] * b[2] = 3 * 7 = 21
Minimum Scalar Product (Parallel): 80
Time taken (Parallel): 3.34857 milliseconds
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment03$
```

Information and analysis:

Sqeuential time < Parallel time (Small Size input)

Sequential Execution:
The sequential computation is extremely fast, taking only 0.000293 milliseconds.

Parallel Execution:
The parallel computation, however, takes 3.34857 milliseconds, which is significantly longer than the sequential execution time.

Problem Statement 2:

Write OpenMP code for two 2D Matrix addition, vary the size of your matrices from 250, 500, 750, 1000, and 2000 and measure the runtime with one thread (Use functions in C in calculate the execution time or use GPROF)
i. For each matrix size, change the number of threads from 2,4,8., and plot the speedup versus the number of threads.
ii. Explain whether or not the scaling behaviour is as expected.

Screenshots:

```
#include <iostream>
#include <vector>
#include <omp.h> // OpenMP header
#include <chrono> // For measuring time

// Function to add two matrices
void add_matrices(const std::vector<std::vector<int>>& A, const std::vector<std::vector<int>>&
B, std::vector<std::vector<int>>& C, int num_threads) {
int rows = A.size();
int cols = A[0].size();

#pragma omp parallel for num_threads(num_threads) collapse(2)
for (int i = 0; i < rows; ++i) {
for (int j = 0; j < cols; ++j) {
C[i][j] = A[i][j] + B[i][j];
}
}
}

// Function to initialize a matrix with random values
void initialize_matrix(std::vector<std::vector<int>>& matrix) {
int rows = matrix.size();
int cols = matrix[0].size();
for (int i = 0; i < rows; ++i) {
for (int j = 0; j < cols; ++j) {
```

```
matrix[i][j] = rand() % 100; // Random values between 0 and 99
}
}
}

// Function to measure time and perform matrix addition
void perform_test(int size, int num_threads) {
std::vector<std::vector<int>> A(size, std::vector<int>(size));
std::vector<std::vector<int>> B(size, std::vector<int>(size));
std::vector<std::vector<int>> C(size, std::vector<int>(size, 0));

initialize_matrix(A);
initialize_matrix(B);

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

add_matrices(A, B, C, num_threads);

auto end_time = std::chrono::high_resolution_clock::now();
std::chrono::duration<double, std::milli> time_ms = end_time - start_time;

std::cout << "Matrix size: " << size << "x" << size << ", Threads: " << num_threads << ", Time: " <<
time_ms.count() << " milliseconds" << std::endl;
}

int main() {
// Different matrix sizes to test
std::vector<int> matrix_sizes = {250, 500};
// Different numbers of threads to test
std::vector<int> thread_counts = {1, 2, 4};

for (int size : matrix_sizes) {
for (int num_threads : thread_counts) {
perform_test(size, num_threads);
}
}

return 0;
}
```

Output:

```
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment03$ g++ -fopenmp -o a 02_a.cpp
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment03$ ./a
Matrix size: 250x250, Threads: 1, Time: 0.778661 milliseconds
Matrix size: 250x250, Threads: 2, Time: 2.79087 milliseconds
Matrix size: 250x250, Threads: 4, Time: 12.3526 milliseconds
Matrix size: 500x500, Threads: 1, Time: 6.99278 milliseconds
Matrix size: 500x500, Threads: 2, Time: 3.01508 milliseconds
Matrix size: 500x500, Threads: 4, Time: 14.3634 milliseconds
```

Information and analysis:

For the 250x250 matrix, the execution time increases when more threads are used. This is contrary to the expected behavior where more threads should decrease the computation time or at least keep it similar due to parallelization benefits. Similar behavior is observed with the 500x500 matrix; execution time decreases when moving from 1 thread to 2 threads but increases significantly when using 4 threads.

Problem Statement 3:

For 1D Vector (size=200) and scalar addition, Write a OpenMP code with the following: i. Use STATIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyze the speedup. ii. Use DYNAMIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyze the speedup. iii. Demonstrate the use of nowait clause.

Screenshots:

```
#include <iostream>
#include <vector>
#include <omp.h> // OpenMP header
#include <chrono> // For measuring time

// Function to perform scalar addition of a 1D vector
void scalar_addition(std::vector<int>& vec, int scalar, int num_threads, const std::string&
schedule_type, int chunk_size) {
int size = vec.size();

// Parallel region with specified schedule type and chunk size
if (schedule_type == "static") {
#pragma omp parallel for num_threads(num_threads) schedule(static, chunk_size)
for (int i = 0; i < size; ++i) {
vec[i] += scalar;
}
} else if (schedule_type == "dynamic") {
#pragma omp parallel for num_threads(num_threads) schedule(dynamic, chunk_size)
for (int i = 0; i < size; ++i) {
vec[i] += scalar;
}
}
}

// Function to measure time and perform scalar addition
void perform_test(int size, int num_threads, const std::string& schedule_type, int chunk_size) {
std::vector<int> vec(size, 0);
int scalar = 5;

// Measure the start time
```

```
auto start_time = std::chrono::high_resolution_clock::now();

// Perform scalar addition
scalar_addition(vec, scalar, num_threads, schedule_type, chunk_size);

// Measure the end time
auto end_time = std::chrono::high_resolution_clock::now();
std::chrono::duration<double, std::milli> time_ms = end_time - start_time;

// Print the results
std::cout << "Vector size: " << size << ", Threads: " << num_threads
<< ", Schedule: " << schedule_type << ", Chunk size: " << chunk_size
<< ", Time: " << time_ms.count() << " milliseconds" << std::endl;
}

int main() {
// Vector size
int size = 200;
// Different numbers of threads to test
std::vector<int> thread_counts = {1, 2, 4, 8};
// Chunk sizes for testing
std::vector<int> chunk_sizes = {1, 10, 50};

// Perform tests with static scheduling
for (int num_threads : thread_counts) {
for (int chunk_size : chunk_sizes) {
perform_test(size, num_threads, "static", chunk_size);
}
}

// Perform tests with dynamic scheduling
for (int num_threads : thread_counts) {
for (int chunk_size : chunk_sizes) {
perform_test(size, num_threads, "dynamic", chunk_size);
}
}

return 0;
}
```

Output:

```
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment03$ g++ -fopenmp -o a 03_a.cpp
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment03$ ./a
Vector size: 200, Threads: 1, Schedule: static, Chunk size: 1, Time: 0.081291 milliseconds
Vector size: 200, Threads: 1, Schedule: static, Chunk size: 10, Time: 0.00782 milliseconds
Vector size: 200, Threads: 1, Schedule: static, Chunk size: 50, Time: 0.00144 milliseconds
Vector size: 200, Threads: 2, Schedule: static, Chunk size: 1, Time: 4.29528 milliseconds
Vector size: 200, Threads: 2, Schedule: static, Chunk size: 10, Time: 4.19927 milliseconds
Vector size: 200, Threads: 2, Schedule: static, Chunk size: 50, Time: 1.67433 milliseconds
Vector size: 200, Threads: 4, Schedule: static, Chunk size: 1, Time: 1.31636 milliseconds
Vector size: 200, Threads: 4, Schedule: static, Chunk size: 10, Time: 0.828253 milliseconds
Vector size: 200, Threads: 4, Schedule: static, Chunk size: 50, Time: 5.90126 milliseconds
Vector size: 200, Threads: 8, Schedule: static, Chunk size: 1, Time: 2.29597 milliseconds
Vector size: 200, Threads: 8, Schedule: static, Chunk size: 10, Time: 11.8372 milliseconds
Vector size: 200, Threads: 8, Schedule: static, Chunk size: 50, Time: 0.795211 milliseconds
Vector size: 200, Threads: 1, Schedule: dynamic, Chunk size: 1, Time: 0.007864 milliseconds
Vector size: 200, Threads: 1, Schedule: dynamic, Chunk size: 10, Time: 0.026806 milliseconds
Vector size: 200, Threads: 1, Schedule: dynamic, Chunk size: 50, Time: 0.002154 milliseconds
Vector size: 200, Threads: 2, Schedule: dynamic, Chunk size: 1, Time: 3.65455 milliseconds
Vector size: 200, Threads: 2, Schedule: dynamic, Chunk size: 10, Time: 3.66974 milliseconds
Vector size: 200, Threads: 2, Schedule: dynamic, Chunk size: 50, Time: 3.08112 milliseconds
Vector size: 200, Threads: 4, Schedule: dynamic, Chunk size: 1, Time: 3.93512 milliseconds
Vector size: 200, Threads: 4, Schedule: dynamic, Chunk size: 10, Time: 0.680519 milliseconds
Vector size: 200, Threads: 4, Schedule: dynamic, Chunk size: 50, Time: 0.950323 milliseconds
Vector size: 200, Threads: 8, Schedule: dynamic, Chunk size: 1, Time: 2.89 milliseconds
Vector size: 200, Threads: 8, Schedule: dynamic, Chunk size: 10, Time: 4.60928 milliseconds
Vector size: 200, Threads: 8, Schedule: dynamic, Chunk size: 50, Time: 2.07595 milliseconds
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment03$
```

Information and analysis:

1. Static Scheduling:

Smaller Chunk Sizes: *Higher overhead due to frequent scheduling. This results in longer execution times, especially with more threads.*

Larger Chunk Sizes: *Reduced overhead and better performance for single-threaded cases. With multiple threads, performance improvements are mixed, with some degradation due to increased thread management overhead.*

2. Dynamic Scheduling:

Smaller Chunk Sizes: *Most efficient, providing low execution times due to effective load balancing and reduced scheduling overhead.*

Larger Chunk Sizes: *Performance remains competitive, but the benefits are less pronounced as chunk sizes increase. Efficient load balancing remains a key factor in performance.*

3. Thread Count:

Increasing the number of threads generally reduces execution time up to a point but can lead to overhead and inefficiencies with too many threads. Optimal performance is achieved with a balanced number of threads and appropriate scheduling strategy.

Github Link:

https://github.com/omkarauti11/HPC_LAB