
Budapest University of Technology and Economics

High Performance Parallel Computing

BMEVIIIMA06-EN

Teacher: Dr. Szeberényi Imre

# Homework Report:

# Parallel Rock-Paper-Scissors Game Simulation Using OpenMP

Written by: Ismayilzada Ismayil

Neptun#: DGY785

11/11/2023

# Report: Parallel Rock-Paper-Scissors Game Simulation Using OpenMP

## Introduction

This report details a C++ program designed to simulate multiple games of Rock-Paper-Scissors in parallel using OpenMP, a popular parallel programming model. The program not only simulates the games but also benchmarks the performance for different problem sizes.

## Program Components

### Enum Definition (Move)

The Move enum represents the possible moves in the game: Rock, Paper, and Scissors. This enumeration simplifies the representation of moves.

### Function: moveToString

Converts a Move enum value to its corresponding string representation.

Facilitates human-readable output of the game's results.

### Function: randomMove

Generates a random move (Rock, Paper, or Scissors) using the std::mt19937 random number generator.

thread_local ensures each thread has its own random number generator, avoiding conflicts in parallel sections.

omp_get_thread_num() is used as a seed, providing unique random sequences in each thread.

### Function: gameResult

Determines the score of a single game based on the moves of three players.

Implements the game's rules: Rock crushes Scissors, Scissors cut Paper, and Paper covers Rock.

Returns the total score of the game, which is the sum of points scored by all players.

### Main Function

Initiates an output file stream to write benchmark results.

Defines an array of different problem sizes to simulate different numbers of games.

For each problem size:

Measures the execution time using std::chrono::high_resolution_clock.

Uses OpenMP's #pragma omp parallel for directive to simulate games in parallel.

The reduction clause aggregates the total score from all parallel executions.

Each game's move and score are output to the console within an omp critical section to prevent data races.

Writes the problem size and execution time to both the console and a file.

## Parallelization with OpenMP

The parallelization is achieved using the #pragma omp parallel for directive, which distributes the execution of the loop iterations (games) across multiple threads. The reduction clause is used to safely accumulate the totalScore across all threads.

## Benchmarking

The program performs empirical benchmarks, measuring the execution time for different numbers of games (problem sizes). This data is valuable for analyzing the scalability and efficiency of the parallel implementation.

## File I/O

Benchmark results are written to a file named "benchmark_results.txt", allowing for easy storage and later analysis of the performance data.

## An empirical benchmark using different problem size:

Problem Size: 1, Execution Time: 1.2069 milliseconds

Problem Size: 10, Execution Time: 2.7545 milliseconds

Problem Size: 100, Execution Time: 22.498 milliseconds

Problem Size: 1000, Execution Time: 226.647 milliseconds

Problem Size: 5000, Execution Time: 1151.58 milliseconds

Problem Size: 10000, Execution Time: 2337.94 milliseconds

Problem Size: 50000, Execution Time: 11542.7 milliseconds

Problem Size: 100000, Execution Time: 23165.9 milliseconds

## Problem Size vs. Execution Time:

As the problem size (number of games) increases, the execution time also increases. This is expected because more games require more computation.

## Linear Scaling:

The scaling appears to be roughly linear, which is indicated by the fact that as you double the problem size, the execution time also roughly doubles. For example, when the problem size

increases from 5,000 to 10,000, the execution time approximately doubles (1151.58 ms to 2337.94 ms).

### Overhead of Parallelization:

The relatively high execution time for very small problem sizes (like 1 game taking 1.2069 milliseconds) might be due to the overhead of setting up parallel threads in OpenMP. This overhead becomes negligible as the problem size increases.

### Efficiency of Parallelization:

For very large problem sizes (e.g., 100,000 games), the execution time doesn't increase exponentially, which suggests that parallelization is providing a benefit. In a purely sequential execution, you might expect the time to grow more rapidly with the increase in problem size.

### System and Environment Factors:

These times are also influenced by the specific hardware and software environment in which the tests are run. Factors include the number of CPU cores, CPU speed, system load, compiler optimizations, and more.

### Considerations for Small vs. Large Problem Sizes:

For small problem sizes, the overhead of parallelism might not be worth it, as the execution time is dominated by the setup and teardown of parallel threads.

For larger problem sizes, parallelism shows its advantage, making the execution more efficient than a sequential approach.

## Conclusion

This program effectively demonstrates the use of OpenMP for parallelizing a simple game simulation and includes a methodology for empirical performance benchmarking. It showcases key concepts in parallel programming such as thread-local storage, parallel loops, critical sections, and performance measurement.