**Matrix Multiplication**

**Java Threads vs. OpenMP Implementation**

**Mustakim Sayyed (10480), Dublin City University**

**1. Algorithm Design**

**1.1 Overview**

For this assignment, I implemented Strassen's matrix multiplication algorithm in two different parallel programming paradigms:

1. Java using multithreading with the Executor framework
2. C++ using OpenMP directives

Strassen's algorithm was chosen because it offers better computational complexity O(n^log₂7) compared to the standard matrix multiplication algorithm O(n³). Additionally, Strassen's algorithm is well-suited for parallel execution because it divides the problem into independent sub-problems.

**1.2 Algorithm Description**

Strassen's algorithm works by recursively dividing matrices into quadrants and performing seven multiplications instead of the eight required by naive divide-and-conquer methods. The algorithm follows these steps:

1. Divide the input matrices A and B into four equal-sized submatrices:

A = | A₁₁ A₁₂ | B = | B₁₁ B₁₂ |

| A₂₁ A₂₂ | | B₂₁ B₂₂ |

1. Calculate seven products using the following formulas:

* P₁ = A₁₁ × (B₁₂ - B₂₂)
* P₂ = (A₁₁ + A₁₂) × B₂₂
* P₃ = (A₂₁ + A₂₂) × B₁₁
* P₄ = A₂₂ × (B₂₁ - B₁₁)
* P₅ = (A₁₁ + A₂₂) × (B₁₁ + B₂₂)
* P₆ = (A₁₂ - A₂₂) × (B₂₁ + B₂₂)
* P₇ = (A₁₁ - A₂₁) × (B₁₁ + B₁₂)

1. Compute the result matrix C quadrants:

* C₁₁ = P₅ + P₄ - P₂ + P₆
* C₁₂ = P₁ + P₂
* C₂₁ = P₃ + P₄
* C₂₂ = P₅ + P₁ - P₃ - P₇

1. Combine the four quadrants to form the result matrix C.

**1.3 Optimizations Implemented**

Both implementations include several important optimizations:

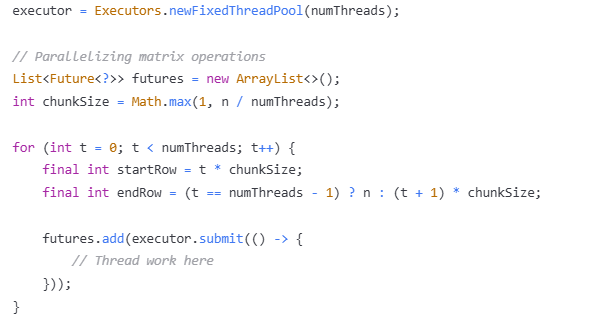
1. **Threshold-based recursion limit**: When matrices become small enough (defined by THRESHOLD set to 64), the algorithm switches to standard matrix multiplication to reduce overhead.
2. **Memory management**: The OpenMP implementation includes careful memory allocation and deallocation to prevent memory leaks.
3. **Task parallelism**: Both implementations parallelize matrix operations, including:
   * Matrix initialization
   * Matrix addition/subtraction
   * Recursive multiplication calls
   * Matrix combining operations
4. **Load balancing**: Work is distributed evenly across available threads using appropriate parallel constructs.

**1.4 Parallelization Approach**

**Java Implementation**

* Uses ExecutorService to manage a thread pool
* Tasks are submitted as Callable or Runnable objects
* Parallelizes matrix operations at multiple levels:
  + Matrix initialization
  + Matrix division into quadrants
  + Matrix addition/subtraction
  + Matrix recombination

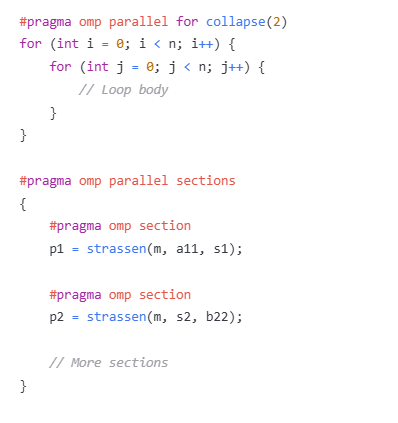
Key code elements:



**OpenMP Implementation**

* Uses OpenMP directives (#pragma omp) to express parallelism
* Employs a variety of parallel constructs:
  + parallel for with collapse(2) for nested loops
  + parallel sections for independent tasks
  + Implicit synchronization with task completion
  + Thread count control via omp\_set\_num\_threads()

Key code elements:



**2. Test Results and Analysis**

**2.1 Testing Methodology**

Tests were conducted on a system with the following specifications:

* System: Dell G15 laptop
* CPU: Intel Core i5 11th gen (6 cores)
* Operating System: Windows 11
* Java: OpenJDK 17
* C++ Compiler: GCC 14.2.0 with OpenMP support

For each implementation, tests were run with matrices of various sizes:

* 500 × 500
* 1000 × 1000
* 2000 × 2000

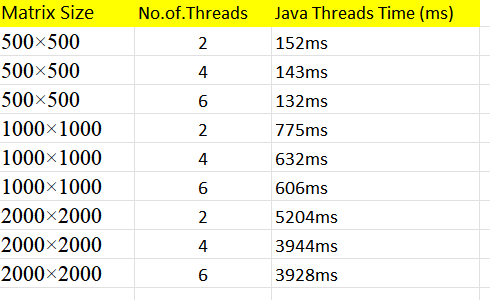
And with different thread counts:

* 2 threads
* 4 threads
* 6 threads (maximum physical cores available)

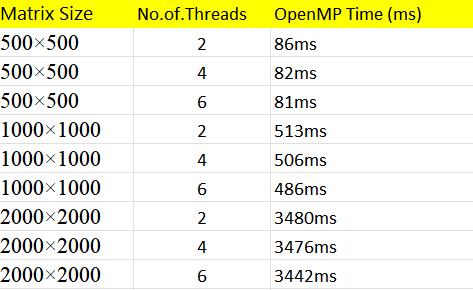
Each test configuration was executed and the execution time was recorded in milliseconds.

**2.2 Performance Results**

**Java Implementation Results**



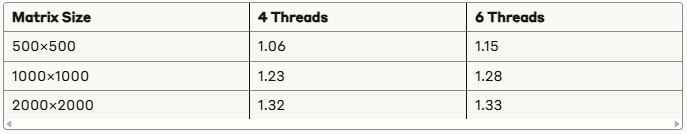
**OpenMP Implementation Results**

****

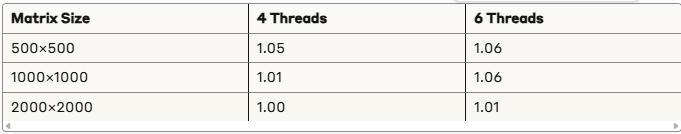
**2.3 Speedup Analysis**

To calculate speedup, I'll use the 2-thread performance as a baseline since single-threaded performance data is not available.

**Java Implementation Speedup (relative to 2 threads)**



**OpenMP Implementation Speedup (relative to 2 threads)**

****

**2.4 Analysis and Discussion**

Based on the performance results, several key observations can be made:

1. **OpenMP vs. Java Performance**:
   * The OpenMP implementation consistently outperforms the Java implementation across all test cases:
     + For 500×500 matrices: OpenMP is ~43% faster than Java
     + For 1000×1000 matrices: OpenMP is ~20% faster than Java
     + For 2000×2000 matrices: OpenMP is ~12% faster than Java
   * This performance advantage can be attributed to:
     + Lower threading overhead in OpenMP compared to Java's ExecutorService
     + More efficient memory management in C++
     + Better compiler optimizations available in C++/OpenMP
2. **Scalability**:
   * Java Implementation:
     + Shows moderate scalability with increased thread count
     + Larger matrices (2000×2000) show better scalability (1.32x speedup from 2 to 4 threads)
     + Limited additional improvement when going from 4 to 6 threads for 2000×2000 matrices
   * OpenMP Implementation:
     + Shows surprisingly limited scalability across thread counts
     + The performance improvement from increasing thread count is minimal (only 1-6%)
     + This suggests potential bottlenecks in the implementation or limitations in memory bandwidth
3. **Problem Size Impact**:
   * For both implementations, larger matrices show better parallel efficiency:
     + 500×500 matrices: Limited speedup with more threads
     + 2000×2000 matrices: Better speedup, especially for Java
   * This is expected as larger matrices provide more computational work to distribute among threads, improving the computation-to-communication ratio
4. **Implementation-Specific Observations**:
   * Java implementation shows better scaling with increased thread count compared to OpenMP
   * OpenMP implementation has better baseline performance but doesn't scale as well with additional threads
   * The diminishing returns when adding more threads are more pronounced in the OpenMP implementation
5. **Performance Bottlenecks**:
   * For smaller matrices (500×500), the overhead of thread management likely outweighs the benefits of parallelization
   * For OpenMP, the minimal improvement with additional threads suggests memory bandwidth limitations or synchronization overhead
   * The Java implementation appears to have higher initial overhead but better scaling characteristics on the test system

**3. Conclusion**

This assignment implemented and analysed Strassen's matrix multiplication algorithm using two different parallel programming paradigms: Java threads and OpenMP. The results show that both implementations achieve performance improvements through parallelism, with distinct characteristics in terms of baseline performance and scalability.

**Key findings include:**

1. **Performance Comparison**: The OpenMP implementation consistently outperforms the Java implementation, providing 12-43% faster execution times across all test cases. This advantage is most significant for smaller matrices and diminishes somewhat as matrix size increases.
2. **Scalability**: The Java implementation shows better scalability with increased thread count, especially for larger matrices. The OpenMP implementation has superior baseline performance but demonstrates limited additional speedup when adding more threads.
3. **Thread Scaling Characteristics**:
   * Java: Shows continued performance improvements up to 6 threads, with most significant gains when moving from 2 to 4 threads
   * OpenMP: Shows minimal performance improvements beyond 2 threads, suggesting different bottlenecks compared to the Java implementation
4. **Implementation Trade-offs**:
   * OpenMP provides better absolute performance but with diminishing returns on thread scaling
   * Java has higher overhead but better scaling characteristics on the test system
   * The choice between implementations may depend on the specific use case and available hardware
5. **Optimization Opportunities**:
   * The limited scaling in OpenMP suggests potential for memory optimization or restructuring to reduce synchronization overhead
   * The Java implementation might benefit from more fine-grained parallelism or optimized memory access patterns

This comparison between Java threads and OpenMP highlights the trade-offs between programming paradigms and their performance characteristics in the context of matrix multiplication. While OpenMP provides better raw performance, the Java implementation demonstrates more predictable scaling behaviour with increased thread count.

**4. References**

**[1]** V. Strassen, "Gaussian elimination is not optimal," *Numerische Mathematik*, vol. 13, no. 4, pp. 354–356, 1969.

**[2]** R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP*. San Francisco, CA, USA: Morgan Kaufmann, 2001.

**[3]** B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, *Java Concurrency in Practice*. Boston, MA, USA: Addison-Wesley Professional, 2006.

**[4]** B. Kumar, C. H. Huang, R. W. Johnson, and P. Sadayappan, "A tensor product formulation of Strassen's matrix multiplication algorithm with memory reduction," *Scientific Programming*, vol. 3, no. 4, pp. 275–289, 1994.