Faculdade de Engenharia da Universidade do Porto

CPD Project 1

Licenciatura em Engenharia Informática e Computação

Turma 6 - Grupo 16

João Sousa (up202106996)

João Teixeira (up202108738)

Index

1. Problem Description

This project aimed to assess the impact of memory hierarchy on processor performance during extensive data access, utilizing matrix multiplication as a case study due to its computational intensity and relevance across various fields. We measured program performance using the Performance API (PAPI) across different programming languages and algorithm implementations. The tasks included:

- Comparing processing times of matrix multiplication in C/C++ and another chosen language (in our case, Java), with matrix sizes ranging from 600x600 to 3000x3000.
- Analyzing a line-by-line multiplication approach and a block-oriented strategy in the selected languages, including performance measurement for large matrices up to 10240x10240 in size.
- Exploring parallel processing techniques to enhance computational efficiency and assess their impact on performance.

2. Algorithms

2.1 Simple Matrix Multiplication

We were given a basic C/C++ algorithm that multiplies two matrices, where one row from the first matrix is multiplied by each column of the second matrix. We opted to implement

this algorithm in Java as our alternative programming language choice. Here is the pseudo code:

```
for(i=0; i<m_ar; i++)
        {       for( j=0; j<m_br; j++)
                {       temp = 0;
                        for( k=0; k<m_ar; k++)
                        {
                                temp += pha[i*m_ar+k] * phb[k*m_br+j];
                        }
                        phc[i*m_ar+j]=temp;
                }
        }
```

2.2 Line Matrix Multiplication

The line multiplication is a variation of the basic multiplication, where the entries in the ith row of A are multiplied by the corresponding entries in the jth row of B and thus accumulated in the respective position of the resulting matrix. Here is the pseudocode

```
for(i=0; i < m_ar; i++)
      for(k = 0; k < m_ar; k++)
            for(j = 0; j < m_br; j++)
                        phc[i * m_ar + j] += pha[i * m_ar + k] * phb[k *
                  m_br + j];
```

2.3 Block Matrix Multiplication

The block matrix multiplication algorithm enhances computational efficiency by dividing the input matrices into smaller blocks, allowing for improved memory cache utilization. It iterates over the matrices in increments defined by bkSize, focusing on submatrices for each multiplication step. This approach ensures that boundary conditions are handled, avoiding out-of-bounds errors for matrices not evenly divisible by bkSize. By processing these blocks, the algorithm optimizes memory access patterns, reducing cache misses and accelerating computation, particularly beneficial for large matrices. Here is the pseudocode:

```
for (i = 0; i < m_ar; i += bkSize)
```

```
        for (k = 0; k < m_ar; k += bkSize)
                for (j = 0; j < m_br; j += bkSize)
                        for (l = i; l < min(i + bkSize, m_ar); l++)
                                for (n = k; n < min(k + bkSize, m_ar); n++)
                                        for (m = j; m < min(j + bkSize,
m_br); m++)
                                                phc[l * m_ar + m] += pha[l *
m_ar + n] * phb[n * m_br + m];
```

2.4 Parallel Matrix Multiplication

For part 2, we used a parallel processing model called, OpenMP, that offers a straightforward way to parallelize code for multi-core processors. This section evaluates two OpenMP directives for parallelizing a line-by-line implementation of matrix multiplication, focusing on performance metrics like MFlops (million floating-point operations per second), speedup, and efficiency.

The first method leverages OpenMP to parallelize the outer loop (l) of the matrix multiplication algorithm. By placing the #pragma omp parallel for directive before the outer loop, OpenMP automatically distributes the iterations of this loop across the available processor cores. Each thread performs multiplication and addition for a subset of rows from the first matrix against the entire second matrix, storing the results in the corresponding rows of the output matrix. Here is the pseudocode:

```
#pragma omp parallel for
for(int l=0; l < m_ar; l++)
     for(int k = 0; k < m_ar; k++)
          for(int z = 0; z < m_br; z++)
               phc[l * m_ar + z] += pha[l * m_ar + k] * phb[k * m_br + z];
```

In the second method, the parallel region is initiated with #pragma omp parallel outside all loops, and the #pragma omp for directive is applied specifically to the innermost loop (z). This strategy distributes the computation of the elements within each row of the output matrix across the available threads. Each thread calculates portions of the output matrix's columns for each row in parallel. Here is the pseudocode:

```
#pragma omp parallel
for(int l=0; l < m_ar; l++)
    for(int k = 0; k < m_ar; k++)
        #pragma omp for
```

```
for(int z = 0; z < m_br; z++)
        phc[l * m_ar + z] += pha[l * m_ar + k] * phb[k * m_br + z];
```

3. Performance Metrics

In evaluating the performance of the algorithms implemented in C/C++, we leveraged the Performance API (PAPI), which offers detailed insights into various CPU metrics, including CPU Cache memory usage at different levels. This method enabled us to accurately measure the execution time of the algorithms, the total number of Floating Point Operations (FLOPs), and cache misses for both L1 and L2 cache levels. Tracking cache misses is essential for assessing the efficiency of an implementation since they significantly increase processing time.

To ensure the accuracy and consistency of our data, all tests were carried out on the same system, a FEUP computer running Ubuntu 22.04, equipped with an i7 9th generation processor. By conducting the measurements three times and averaging the results, we minimized the impact of any minor inconsistencies, thereby ensuring the data's reliability. The code was compiled with the -O2 optimization flag, optimizing compilation time and code performance.

For serial execution timing, we employed the system time clock due to its straightforward integration with both single-threaded applications and the operating system, providing reliable method for measuring execution time. In contrast, for the parallel segments of our C/C++ implementations, we utilized omp_get_wtime() from the OpenMP API to measure execution time. This function was chosen for its precision in capturing the wall clock time of parallel executions, offering an accurate measure of performance improvements gained through parallelization.

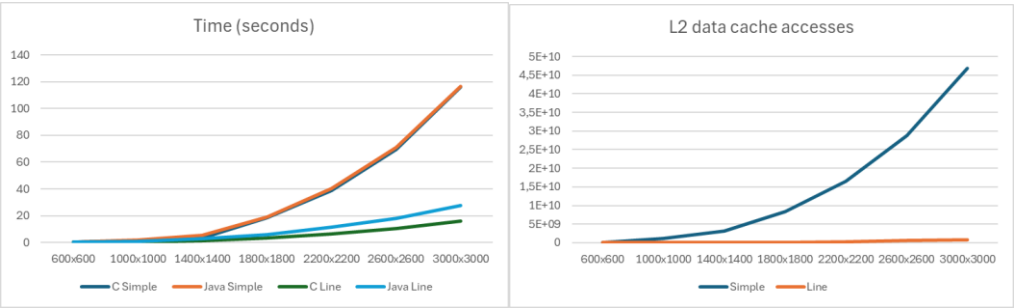
In addition to comparing the execution times of the C/C++ and Java versions, we also calculated the FLOPs using the formula 2*n^3/parallelTime, where 'n' represents the size of the matrix. In parallel computing, "speedup" is defined as the ratio of the time required to complete a task on a single core to the time it takes on multiple cores, so we used the formula serialTime/parallelTime. To evaluate computational efficiency per core, we divided the speedup by the number of cores, in this case, 8.

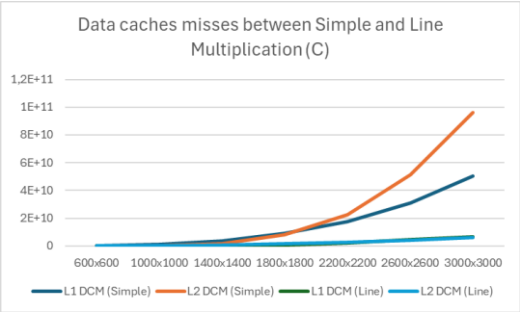
Furthermore, it's important to explain the concept of L2 DCA (Level 2 Data Cache Access). L2 cache is a secondary cache level that stores data not found in the L1 cache. Access to the L2 cache is slower than to the L1 cache but faster than accessing the main memory. When the CPU needs data, it first checks the L1 cache, then the L2 cache before reaching out to the slower main memory. Efficient use of L2 cache can significantly reduce memory access times and improve overall performance. In our evaluation, monitoring L2 cache accesses provided additional insights into the algorithms' efficiency in managing memory hierarchy.

4. Results and analysis

The results are the average of three consecutive tests, and they are presented on the vertical axis of each graph. On the horizontal axis is the matrix row/column dimension.

4.1 Simple And Line Matrix Multiplication

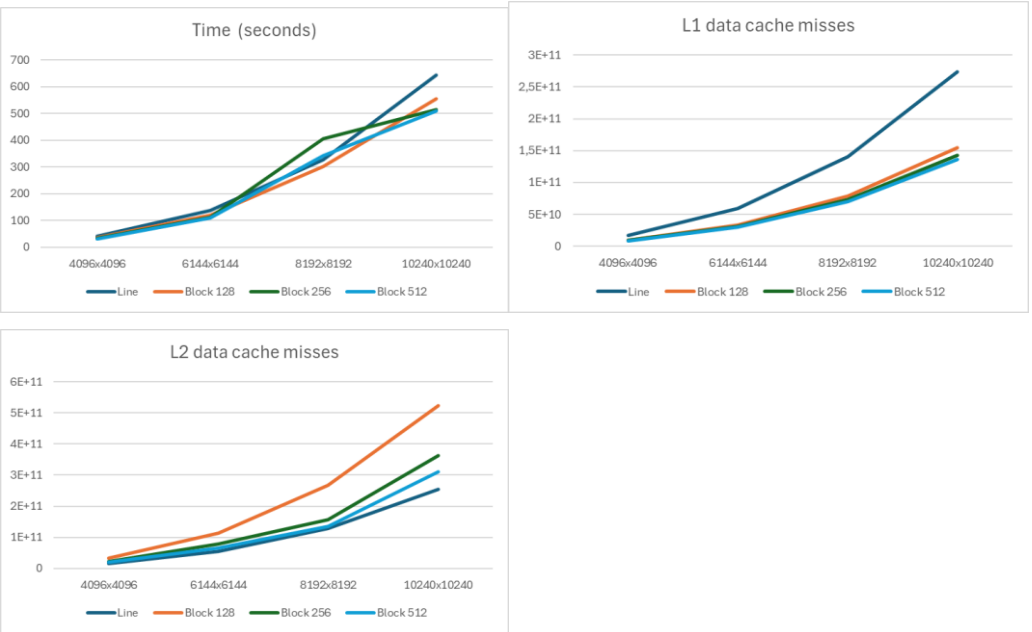




In both of these algorithms, the processing speeds were nearly identical because they're compiled and adhere to a similar programming approach. The line multiplication algorithm outperforms the simple multiplication in terms of efficiency, with a reduction in cache misses, cache accesses and overall execution time. This improvement is achieved by employing the result matrix as accumulative storage, alongside an intermediary loop that accesses the matrix row by row. This strategy ensures data is more readily available in lower-level memory, effectively reducing cache access and maximizing cache hits (when data is found in the cache), therefore, minimizing cache misses.

The performance evaluation of the two algorithms, when implemented in C and Java, shows no significant differences. Java compiles code to bytecode, which is then run on the Java Virtual Machine (JVM), and C is a compiled language that translates code directly into machine language. While Java is predominantly object-oriented, it also supports procedural programming paradigms. C, fundamentally a procedural language, excels in system-level and performance-critical applications. Despite these distinctions, their performance in executing the given algorithms is comparably efficient, underscoring their robustness and versatility in handling complex computational tasks.

4.2 Block Matrix Multiplication

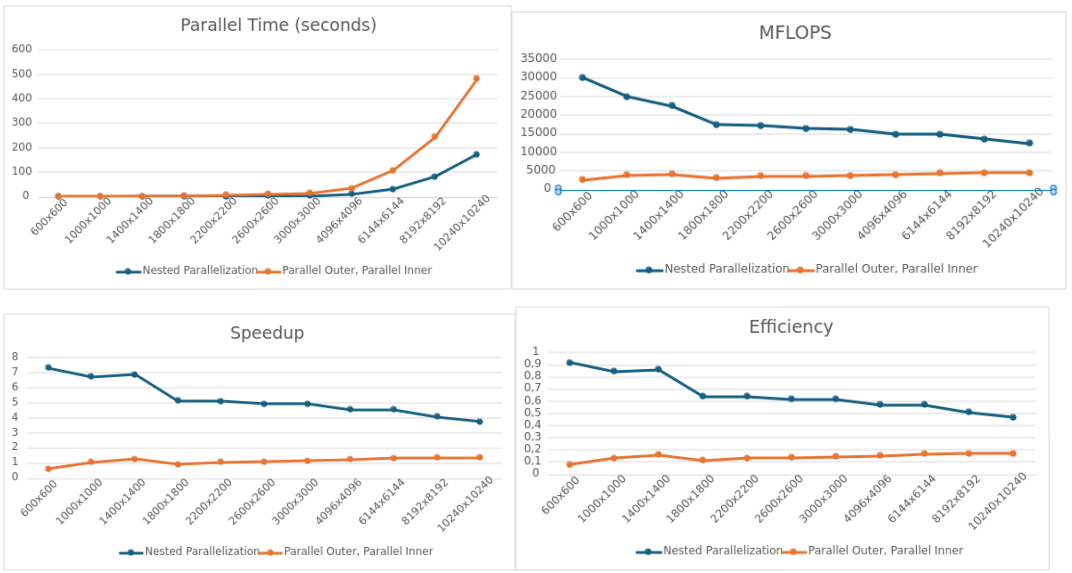


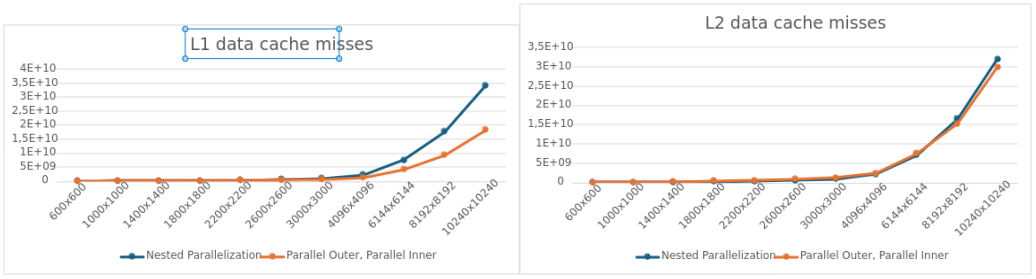
The Block Matrix Multiplication Algorithm demonstrates improved efficiency at the L1 cache level by reducing cache miss rates compared to previous algorithms.

For a 128x128 block size, where each element is a double-precision floating-point number (8 bytes), the total memory footprint is 128KB, exactly matching the L1 cache capacity but also significantly occupying the L2 cache. This scenario leads to optimal use of the L1 cache but begins to push the boundaries of what the L2 cache can comfortably handle without frequent misses.

As block sizes increase to 256x256 and 512x512, requiring 512KB and 2MB of memory respectively, the impact on the L2 cache becomes more pronounced. These larger block sizes exceed the L1 cache's capacity and rely more heavily on the L2 cache. Interestingly, despite the increased demand these larger blocks place on the L2 cache, improvements in L2 cache miss rates are observed. This improvement can be attributed to the better utilization of the L2 cache's larger storage capacity compared to the L1 cache.

4.3 Parallel Matrix Multiplication





For the parallel matrix multiplication, the Nested Parallelization and Parallel Outer, Parallel Inner methods offer insights into the trade-offs between maximizing parallel hardware use and optimizing cache memory access. For instance, the Nested Parallelization method, applied to a 600x600 matrix, demonstrates an impressive speedup of 7.32547 with a parallel time average of 0.0143336 seconds and MFLOPS of 30139.1. This is indicative of high computational efficiency for smaller matrices but also reveals the challenge of scaling as matrix sizes increase. At larger sizes, like 10240x10240, we see a reduction in efficiency (MFLOPS: 12393.6) and speedup (3.75475), accompanied by significant increases in cache misses (L1 DCM: ~34 billion, L2 DCM: ~32 billion).

On the other hand, the Parallel Outer, Parallel Inner method showcases its capabilities with a 600x600 matrix, achieving a modest speedup of 0.615154 and an efficiency of 0.0768942, alongside higher L1 DCM (around 8.5 million) and L2 DCM (around 34 million) compared to Nested Parallelization at the same matrix size. This method's performance difference becomes more pronounced at larger sizes, such as 10240x10240, where it attains a speedup of 1.35169 and efficiency of 0.168961, albeit with significantly lower MFLOPS (4461.62) compared to Nested Parallelization.

These examples underscore the nuanced balance required between leveraging parallelism and managing cache performance. Nested Parallelization, while achieving higher computational efficiency and speedup for smaller matrices, faces scalability issues due to increased cache misses and overhead in synchronization as matrix sizes grow. The Parallel Outer, Parallel Inner method, although it starts with lower efficiency and speedup, exhibits a more gradual decline in performance, suggesting a slightly better management of cache efficiency but at the cost of overall computational throughput.

This comparison reveals that while both methods aim to exploit the capabilities of parallel hardware, their effectiveness is deeply influenced by the matrix size and the inherent trade-offs between parallel execution efficiency and cache memory optimization. It highlights the importance of choosing the appropriate parallelization strategy based on the specific requirements and constraints of the computational task at hand. Adding to this, it's crucial to note that despite the trade-offs, both parallelization strategies significantly outperform serial computation strategies in terms of cache misses and time. The parallel methods effectively utilize the cache hierarchy and computing resources, providing substantial performance improvements over traditional serial approaches, especially as the size of the matrix grows. This advantage becomes increasingly important for large-scale computational tasks, where the efficient use of hardware resources and minimizing cache misses are critical for achieving optimal performance.

5. Conclusions

In this project, we embarked on an in-depth exploration to understand the impact of memory hierarchy on processor performance, focusing on matrix multiplication as a pivotal case study. Through comparative analysis across various algorithms and programming languages, specifically C/C++ and Java, our investigation unveiled significant insights into how different computational strategies interact with the memory hierarchy to influence performance. The use of the Performance API (PAPI) allowed for a nuanced assessment of performance metrics, revealing that the implementation strategy is crucial in optimizing computational efficiency. Particularly, block matrix multiplication stood out for its ability to enhance cache utilization, demonstrating that careful algorithm selection can substantially mitigate the limitations imposed by memory access patterns.

Parallel processing techniques further accentuated the study's findings, illustrating the potent benefits of leveraging multicore processors to accelerate computations. The Nested Parallelization and Parallel Outer Parallel Inner strategies highlighted the importance of balancing parallel hardware utilization with cache memory optimization. This balance is critical in maximizing computational throughput while minimizing cache misses, especially for large-scale computations.