Parallel and Distributed Computing

First Project

Bruno Oliveira - up202208700
Mansur Mustafin - up202102355
Rodrigo Silva - up202205188

Bachelor's in Informatics and Computing Engineering
Parallel and Distributed Computing

Professor: João Resende

March 2025

Contents

1 Introduction

This report presents the algorithms, performance results and respective analysis of the first project in the Parallel and Distributed Computing course. The objective of this project was to evaluate and compare the performance of different memory access patterns and parallelization techniques.

Matrix multiplication was chosen as the central problem due to its relative simplicity and its suitability for testing various performance-related aspects. In particular, we examine how operating systems and hardware handle memory access. Specifically, most CPUs employ techniques such as caching, where data is temporarily stored closer to the CPU for faster access, and prefetching, where data is retrieved in contiguous blocks (cache lines). Consequently, when one element is accessed, the processor often fetches adjacent elements as well and keeps them cached. To observe this effect, we implemented three single-core algorithms. To explore how different parallelization strategies can improve performance, we also implemented two multi-core algorithms (see Algorithms Explanation section).

By developing the code for these tasks, we became more familiar with modern and high performance C++ and Lua, since we chose it as our alternative implementation language. Moreover, we also had to research about matrix multiplication algorithms and in order to implement multi-core solutions, we explored and experimented with the OpenMP standard, allowing us to learn more about CPU multithreading.

2 Algorithms Explanation

To explore how cache locality and parallel computing affect matrix multiplication, we implemented five different algorithms. Although cache-related behavior is largely independent of the programming language, we additionally developed Lua implementations to demonstrate that performance differences arise from algorithmic choices rather than language specifics. In spite of these algorithms performing the same operations and producing the same result, the order of operations can significantly affect memory access patterns and thus performance.

2.1 Naive Matrix Multiplication

The *naive* matrix multiplication algorithm is closely related to its hand calculation. This method for multiplication of two matrices: $A \times B = C$, where $A(m, n)$, $B(n, p)$ and $C(m, p)$, obtains the sum of a cell (i, j) in the solution, by linear combination of row i of A and column j of B.

Implementing this algorithm is fairly straighforward, as we only need to iterate through every row and column of the solution matrix, and, for each cell, perform the linear combination of the values as shown in Figure 1.

As we can expect from the 3 nested loops, the overall time complexity for the matrix multiplication algorithm is $\Theta(n^3)$. This time complexity is maintained among

```
for (i = 0; i < m; i++)
    for (j = 0; j < p; j++)
        for (k = 0; k < n; k++)
            C[i][j] += A[i][k] * B[k][j];
```

Figure 1: Overview of Naive Matrix Multiplication Algorithm

all the implemented algorithms, therefore performance impacts are mainly caused by cache accesses and parallelism.

2.2 Line Matrix Multiplication

The line matrix multiplication algorithm is very similar to the previous algorithm. The difference is that instead of calculating the result matrix cell by cell, we instead focus on calculating and storing some terms of the linear combination line by line. Since matrices are stored in row-major format, accessing data consecutively along a row can improve cache locality, and, consequently reduce the time execution.

The only difference is that the second and third loops switch places as shown in figure 2.

```
for (i = 0; i < m; i++)
    for (k = 0; k < n; k++)
        for (j = 0; j < p; j++)
            C[i][j] += A[i][k] * B[k][j];
```

Figure 2: Overview of Line Matrix Multiplication Algorithm

2.3 Block Matrix Multiplication

The last single-core algorithm implemented is the block matrix multiplication algorithm. This approach expands on top of the previous one and the same logic applied to calculate subresults line by line is now used to calculate the solution block by block. The main idea is that now, we have divided the matrices into smaller blocks and the operations are performed block by block. This approach can further enhance cache locality by keeping calculations within a more localized area, potentially improving overall efficiency.

For this, we need 6 nested loops as shown in Figure 3.

```
for (I = 0; I < m; I += block_size)
    for (K = 0; K < n; K += block_size)
        for (J = 0; J < p; J += block_size)
            for (i = I; i < min(I + block_size, m); i++)
                for (k = K; k < min(K + block_size, n); k++)
                    for (j = J; j < min(J + block_size, p); j++)
                        C[i][j] += A[i][k] * B[k][j];
```

Figure 3: Overview of Block Matrix Multiplication Algorithm

2.4 Parallel Matrix Multiplication

Besides the previous three single-core algorithms, we also implemented two variations of the line algorithm using OpenMP API for parallel programming in C++. The next sections show an explanation and overview of the implementations.

2.4.1 First Version

The first approach aims to parallelize the outermost loop `i` of the line algorithm. There will be a division of `i` values between the different threads so that each one works on separate rows at the same time (they also get their own `k` and `j` variables). For this approach, the threads all run independently and then synchronize once when the loop finishes.

For this, we used OpenMP's `parallel for` and `private` variable pragmas, as shown in Figure 4.

```
#pragma omp parallel for private(i, k, j)
for (i = 0; i < m; i++)
    for (k = 0; k < n; k++)
        for (j = 0; j < p; j++)
            C[i][j] += A[i][k] * B[k][j];
```

Figure 4: Overview of First Parallel Matrix Multiplication Algorithm

2.4.2 Second Version

The second approach aims to parallelize the innemost loop `j` of the line algorithm. The thread pool will work on the same row at the same time but on different columns. In this case, the threads need to be synchronized every time the inner loop finishes, which happens $i \times k$ times, leading to some overhead.

For this, we used OpenMP's `parallel` region, `for` and `private` variable pragmas, as it can be seen in Figure 5.

```
#pragma omp parallel private(i, k)
for (i = 0; i < m; i++)
    for (k = 0; k < n; k++)
        #pragma omp for private(j)
        for (j = 0; j < p; j++)
            C[i][j] += A[i][k] * B[k][j];
```

Figure 5: Overview of Second Parallel Matrix Multiplication Algorithm

3 Performance Metrics

The performance of the algorithms described in the previous sections was evaluated using the following metrics:

- **Time:** The time required to execute the algorithm. This is a crucial metric for comparing the speed and efficiency of multicore implementations.
- **Speedup:** Defined by the ratio of the sequential execution time T_{seq} to the parallel execution time T_{par}:

  $$\text{Speedup} = \frac{t_{\text{seq}}}{t_{\text{par}}}.$$

  This indicates how much faster the parallel algorithm runs compared to the sequential version. A higher speedup indicates more effective utilization of parallel resources.
- **Efficiency:** Defined by the ratio of the speedup to the number of threads p:

  $$\text{Efficiency} = \frac{\text{Speedup}}{p}.$$

  This measures how effectively the algorithm scales with an increasing number of threads. An efficiency of 1 (or 100%) indicates perfect scaling.
- **FLOPS:** The number of floating-point operations per second, given by the formula

  $$\text{FLOPS} = \frac{2 \times n^3}{t},$$

  where $2 \times n^3$ represents the total number of arithmetic operations (one multiplication and one addition per iteration) in the matrix multiplication, and t is the total execution time. Higher FLOPS values indicate more numerical computations for the same amount of time.
- **Cache Misses:** To evaluate memory access patterns and their performance impact, we use number of cache miss, such as `L1_DCM` (Level 1 Data Cache Misses), `L2_DCM` (Level 2 Data Cache Misses), and `L3_TCM` (Level 3 Total Cache Misses). Each cache miss forces the processor to fetch data from a slower cache level or main memory, incurring extra clock cycles. Reducing cache misses can improve time execution by minimizing these wasted cycles.

4 Results and Analysis

To benchmark the implementations previously discussed, we executed the test set shown below 5 times, storing the results for analysis. All algoritms were conducted on university computers with Intel Core i7-9700 with 64KB of L1 cache, 256KB L2 cache and 12MB shared among cores L3 cache, operating under Ubuntu and C++ version compiled using g++ with -O2 flag. The graphs produced are available in the Appendix section.

- Naive, Line and Parallel implementations (across both C++ and Lua) with input matrices from 600×600 to 3000×3000 with size increments of 400.
- Line, Block and Parallel implementations (in C++) with input matrices from 4096×4096 to 10240×10240 with size increments of 2048, with block sizes 128, 256 and 512.

4.1 Performance Evaluation of a Single Core Implementation

The first comparison in the single core implementations is between the naive and the line approach, which was evaluated both in C++ and Lua. The results show that, independently of the language of implementation, the execution times are smaller for the latter. It can also be observed that execution time is faster in C++ when compared to Lua. Evidence for this can be seen in graphs 6 and 7.

The main reason why the line implementation is more performant than the naive approach is due to sequential data processing, which reduces cache misses, as shown in graph 8. Due to the prefetching mechanism, when one memory address is accessed, adjacent data is also loaded into memory. This ensures that subsequent memory accesses hit the cache instead of requiring slower memory fetches.

Having established the advantages of the line implementation, we can try to further optimize the code to reduce cache misses during memory accesses. The best approach for this is to subdivide the matrix calculation and perform it in small blocks, limiting the memory region being processed. We decided to test this with different matrix and block sizes, and the results can be consulted in figures 9, 10, 11, 12 and 13.

Overall, the block algorithms are proportionally better than the line algorithms in relation to the L1 and L2 cache misses, as shown in figures 9, 11 and 12.

However, this is not the case for matrix size of 8192×8192, where the block algorithms with block sizes of 256 and 512 suffer a spike in execution time (and consequent decrease in Mflops), shown in charts 9 and 10. These results were caused by the number of cache misses that occur on the L3 cache, suggesting that these block sizes don't fit well in this cache.

According to the memory hierarchy, the cost cache misses on L3 is much higher than L1 and L2, because it requires data to be fetched from an even lower level, therefore, the total execution is higher than the expected. To verify this, we ran some tests while measuring the `L3_TCM` and the results are shown in 13.

4.2 Performance Evaluation of a Multi-Core Implementation

Besides the single core algorithms, we also implemented two multi-core approaches, based on the line algorithm, using OpenMP. As expected, both multi-core algorithms have better overall performance when compared to the single core line approach.

By distributing the operation load across different processing units, these approaches are able to perform a larger number of floating point operations per second and, consequently, have smaller execution times, as we can see in graphs 14 and 15.

Nonetheless, the multi-core implementations follow different ideas, as discussed in the Algorithms section and have substantially different results. While still being faster than the sequential code, the first implementation is able to execute 3 to 4 times more Mflops than the second one, as seen in 15.

While both take advantage of parallelism, the second approach has an associated synchronization overhead after each iteration of the inner parallelized loop, which occurs $size^2$ times, leading to sizeable overheads increasing quadratically with matrix size. Nonetheless, it will still be faster than the single-core algorithm because the number of operations performed by each thread also grows.

As a result of this, we observe faster execution times for the first parallel approach which uses parallelism on the outermost loop, only requiring synchronization in the end. The comparison between these two algorithms is clear when analyzing their speedup and efficiency.

While the second approach has a speedup of slightly over $1.0\times$ for matrices over around 1000×1000 and efficiency below 0.2, the first implementation is at least $4\times$ faster (usually even around $5\times$) and has an efficiency over 0.5. These conclusions can be seen in the figures 16 and 17.

5 Conclusion

In conclusion, this project allowed us to deepen our understanding of fundamental programming concepts, such as data accesses, with emphasis on the impacts of cache misses, prefetching mechanisms and parallel computing. By implementing multiple valid approaches to the same problem and comparing their performance metrics, we were able to verify the concepts taught in the theoretical classes.

Developing the code across two languages, in our case C++ and Lua, also provided us with better insights in those languages, particularly C++, where we explored the Performance API (PAPI) for benchmarking and OpenMP for the multi-core programming.

Overall, the project successfully met its objectives, giving us hands-on experience with the intricacies of parallel computing and demonstrating the performance capabilities of programs developed with memory layout and multi-core mechanisms in mind.

6 Appendix

6.1 Comparison Charts

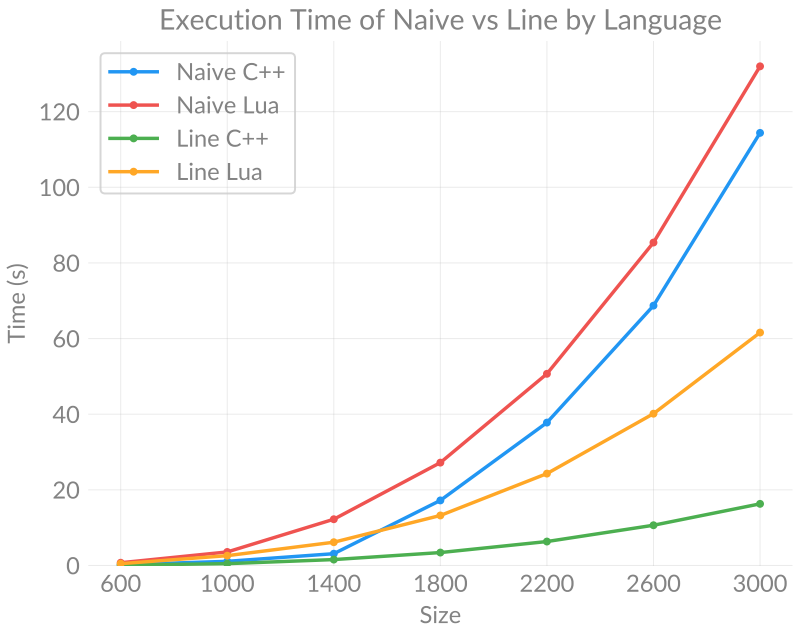

Figure 6: Time comparison between naive and line multiplication, in both C++ and Lua

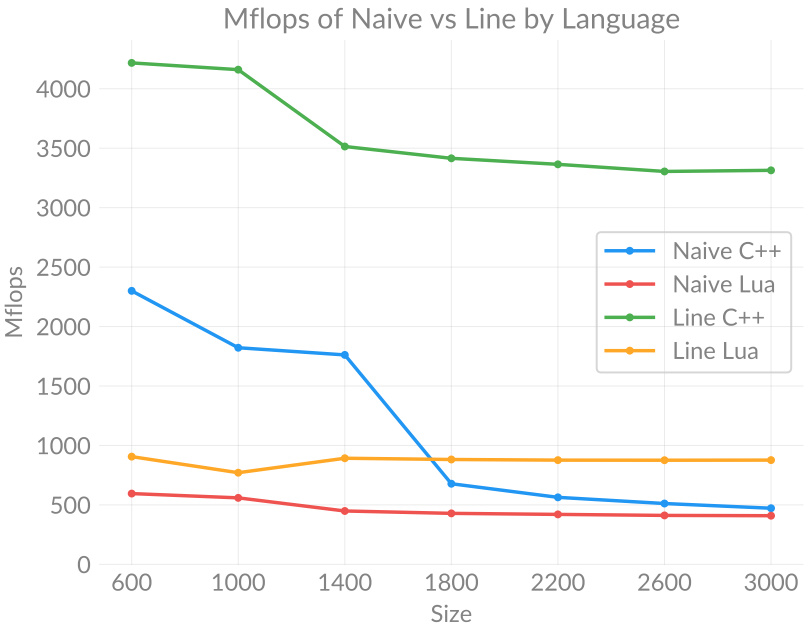

Figure 7: Mflops comparison between naive and line multiplication, in both C++ and Lua

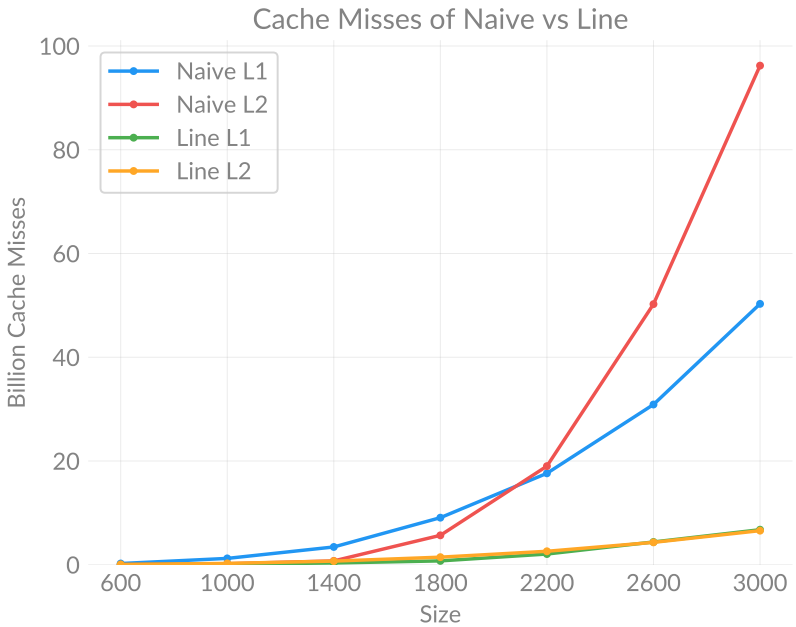

Figure 8: Cache misses comparison between naive and line multiplication, in C++

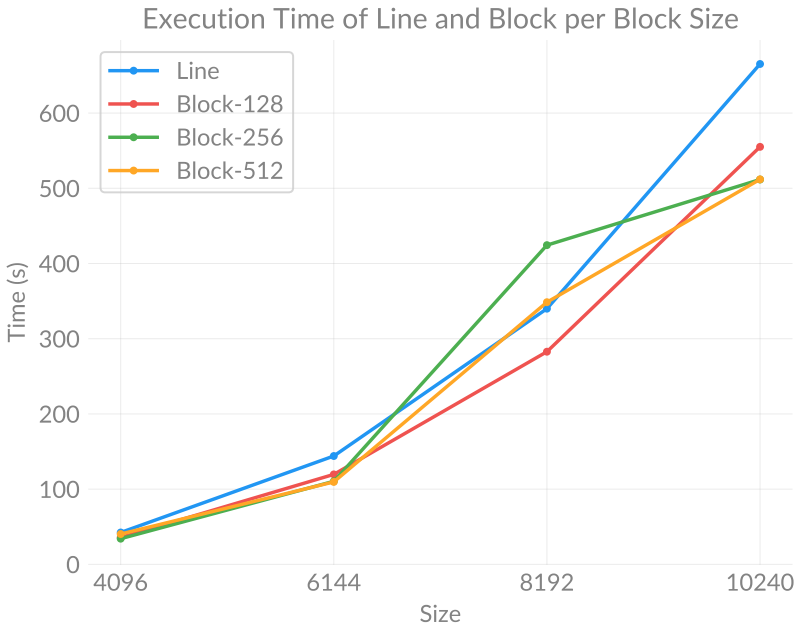

Figure 9: Time comparison between line and block multiplication, with different block sizes

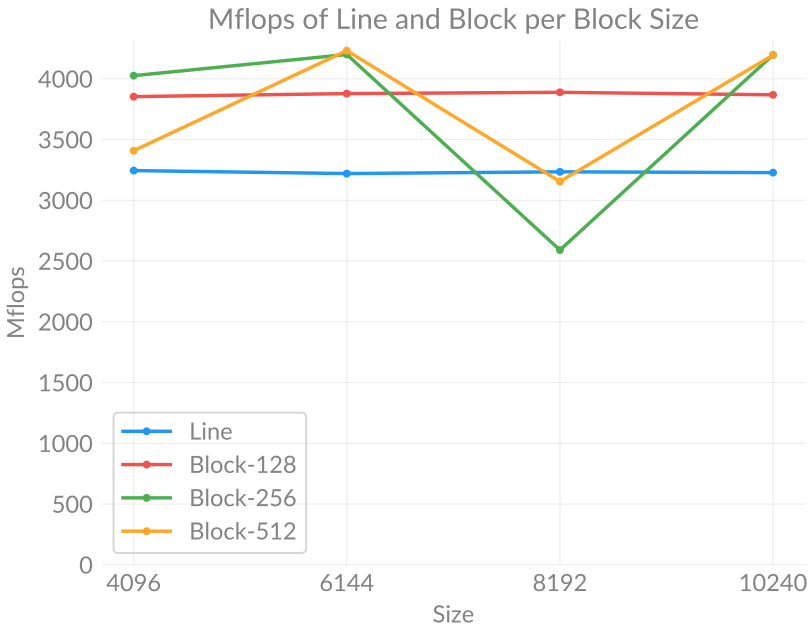

Figure 10: Mflops comparison between line and block multiplication, with different block sizes

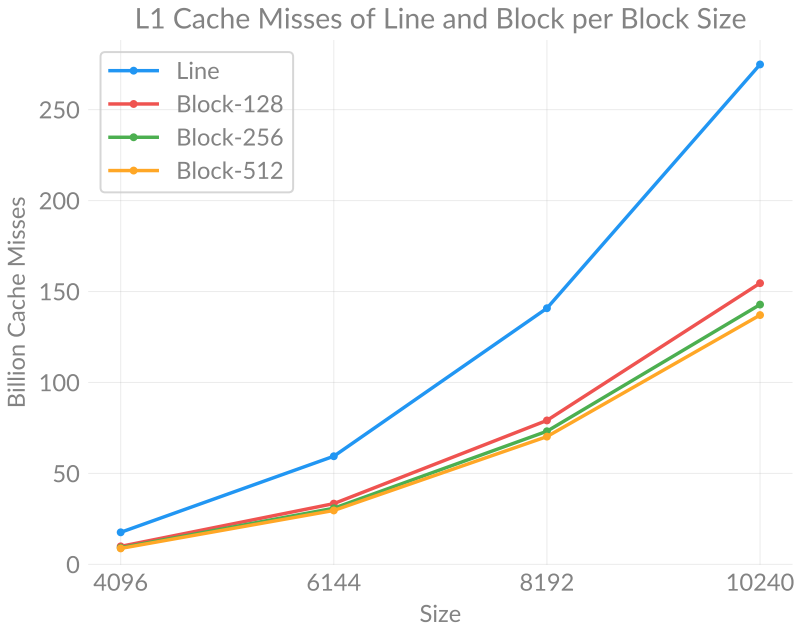

Figure 11: L1 cache miss comparison between line and block multiplication, with different block sizes

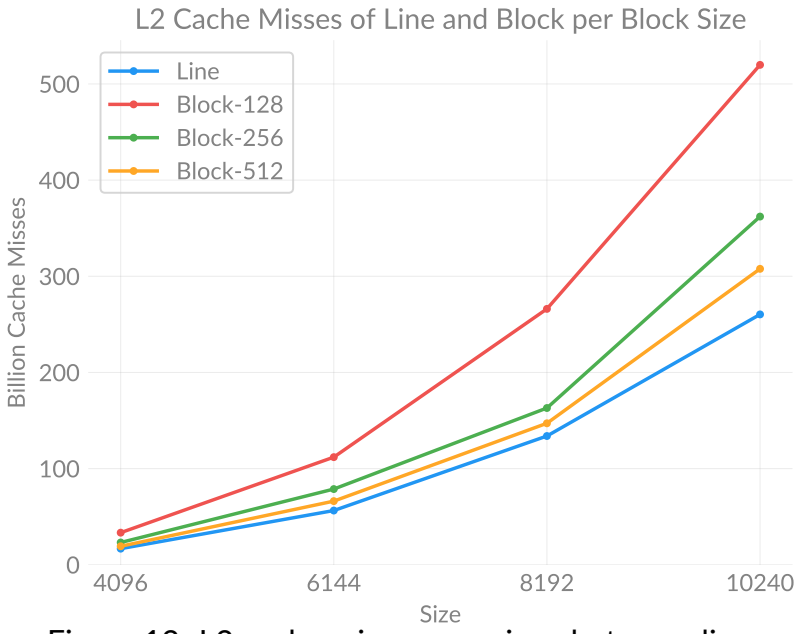

Figure 12: L2 cache miss comparison between line and block multiplication, with different block sizes

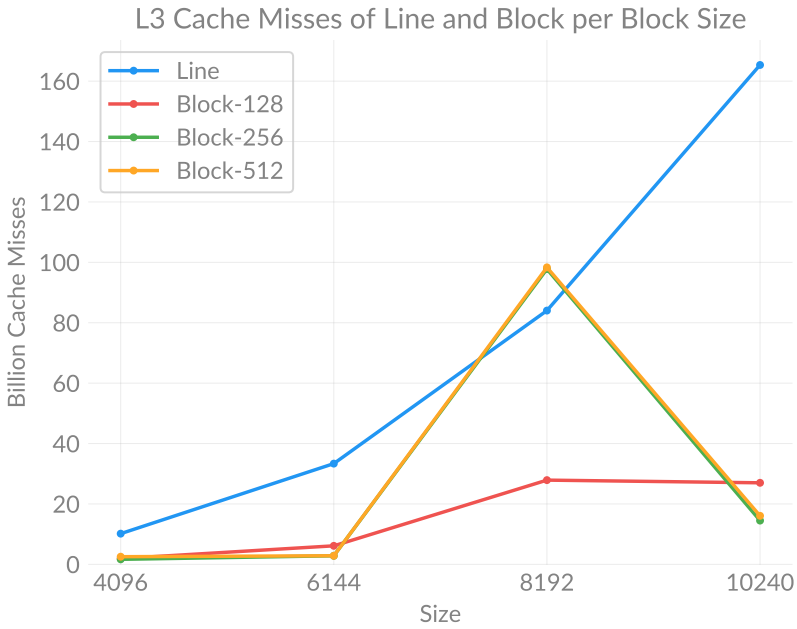

Figure 13: L3 cache miss comparison between line and block multiplication, with different block sizes

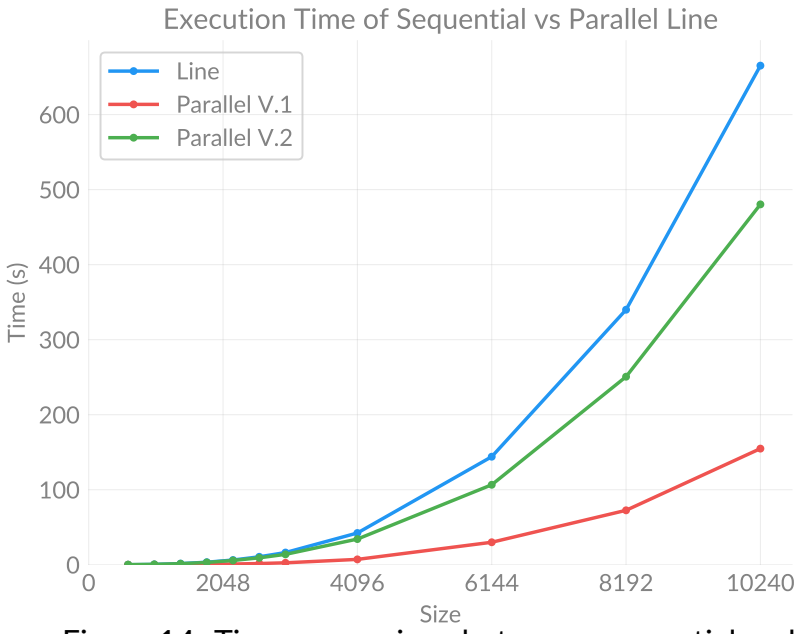

Figure 14: Time comparison between sequential and parallel line multiplication

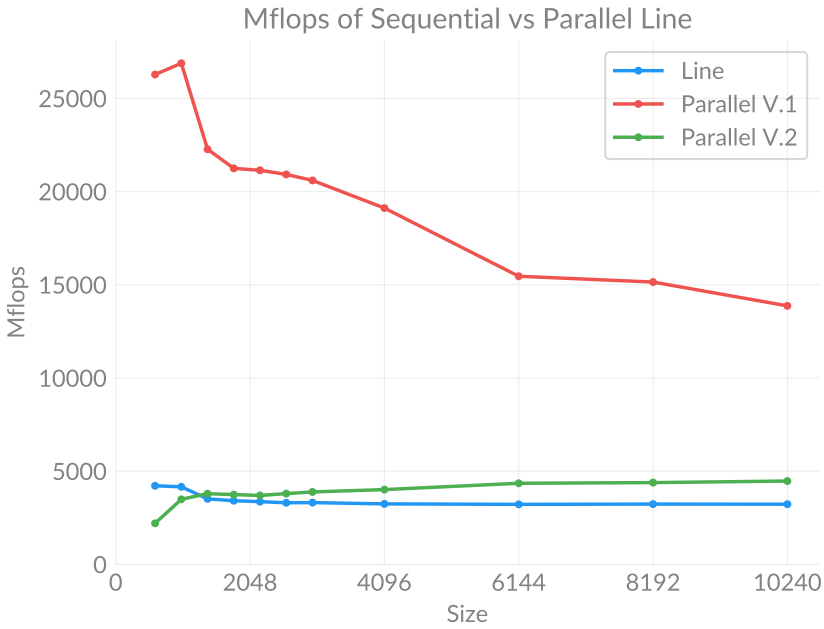

Figure 15: Mflops comparison between sequential and parallel line multiplication

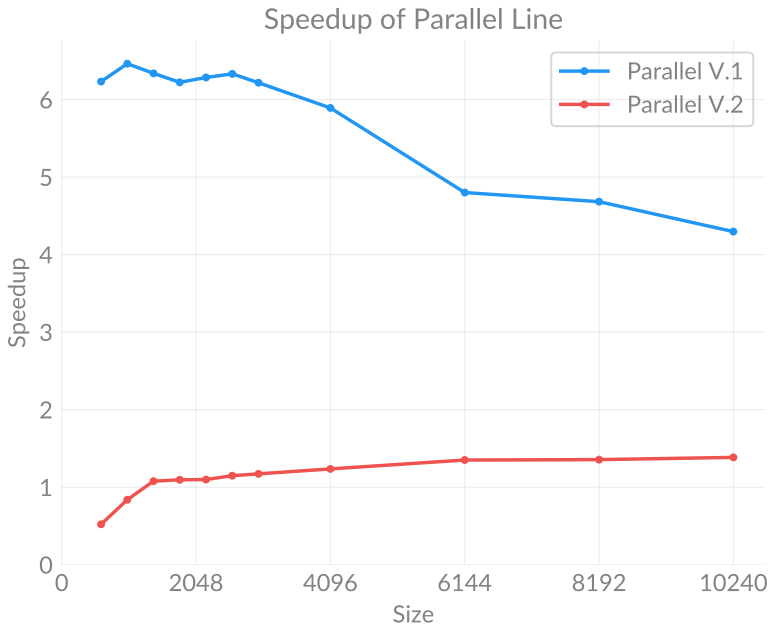

Figure 16: Speedup comparison between versions of parallel line multiplication, relative to the sequential version

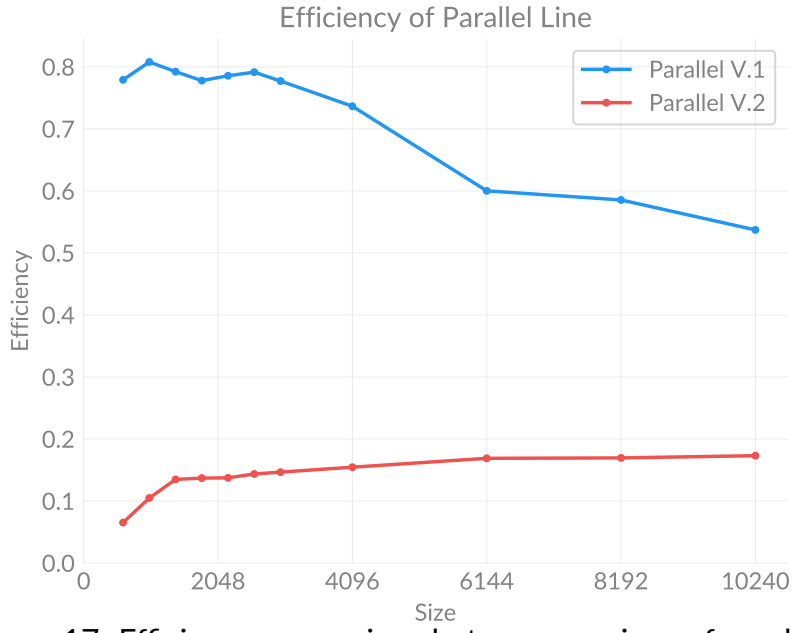

Figure 17: Efficiency comparison between versions of parallel line multiplication, relative to the sequential version

6.2 PC Specifications

```
Available PAPI preset and user defined events plus hardware information.
--------------------------------------------------------------------------------
PAPI version             : 7.1.0.0
Operating system         : Linux 6.5.0-15-generic
Vendor string and code   : GenuineIntel (1, 0x1)
Model string and code    : Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz (158, 0x9e)
CPU revision             : 13.000000
CPUID                    : Family/Model/Stepping 6/158/13, 0x06/0x9e/0x0d
CPU Max MHz              : 4700
CPU Min MHz              : 800
Total cores              : 8
SMT threads per core     : 1
Cores per socket         : 8
Sockets                  : 1
Cores per NUMA region    : 8
NUMA regions             : 1
Running in a VM          : no
Number Hardware Counters : 10
Max Multiplex Counters   : 384
Fast counter read (rdpmc): yes
--------------------------------------------------------------------------------

================================================================================
    PAPI Preset Events
================================================================================
    Name        Code    Deriv Description (Note)
PAPI_L1_DCM  0x80000000  No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  No   Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002  Yes  Level 2 data cache misses
PAPI_L2_ICM  0x80000003  No   Level 2 instruction cache misses
PAPI_L1_TCM  0x80000006  Yes  Level 1 cache misses
PAPI_L2_TCM  0x80000007  No   Level 2 cache misses
PAPI_L3_TCM  0x80000008  No   Level 3 cache misses
PAPI_CA_SNP  0x80000009  No   Requests for a snoop
PAPI_CA_SHR  0x8000000a  No   Requests for exclusive access to shared cache line
PAPI_CA_CLN  0x8000000b  No   Requests for exclusive access to clean cache line
PAPI_CA_ITV  0x8000000d  No   Requests for cache line intervention
PAPI_L3_LDM  0x8000000e  No   Level 3 load misses
PAPI_TLB_DM  0x80000014  Yes  Data translation lookaside buffer misses
PAPI_TLB_IM  0x80000015  No   Instruction translation lookaside buffer misses
PAPI_L1_LDM  0x80000017  No   Level 1 load misses
PAPI_L1_STM  0x80000018  No   Level 1 store misses
PAPI_L2_LDM  0x80000019  No   Level 2 load misses
PAPI_L2_STM  0x8000001a  No   Level 2 store misses
PAPI_PRF_DM  0x8000001c  No   Data prefetch cache misses
PAPI_MEM_WCY 0x80000024  No   Cycles Stalled Waiting for memory writes
PAPI_STL_ICY 0x80000025  No   Cycles with no instruction issue
PAPI_FUL_ICY 0x80000026  Yes  Cycles with maximum instruction issue
PAPI_STL_CCY 0x80000027  No   Cycles with no instructions completed
PAPI_FUL_CCY 0x80000028  No   Cycles with maximum instructions completed
PAPI_BR_UCN  0x8000002a  Yes  Unconditional branch instructions
PAPI_BR_CN   0x8000002b  No   Conditional branch instructions
PAPI_BR_TKN  0x8000002c  Yes  Conditional branch instructions taken
PAPI_BR_NTK  0x8000002d  No   Conditional branch instructions not taken
PAPI_BR_MSP  0x8000002e  No   Conditional branch instructions mispredicted
PAPI_BR_PRC  0x8000002f  Yes  Conditional branch instructions correctly predicted
PAPI_TOT_INS 0x80000032  No   Instructions completed
PAPI_LD_INS  0x80000035  No   Load instructions
PAPI_SR_INS  0x80000036  No   Store instructions
PAPI_BR_INS  0x80000037  No   Branch instructions
PAPI_RES_STL 0x80000039  No   Cycles stalled on any resource
PAPI_TOT_CYC 0x8000003b  No   Total cycles
PAPI_LST_INS 0x8000003c  Yes  Load/store instructions completed
PAPI_L2_DCA  0x80000041  No   Level 2 data cache accesses
PAPI_L3_DCA  0x80000042  Yes  Level 3 data cache accesses
PAPI_L2_DCR  0x80000044  No   Level 2 data cache reads
PAPI_L3_DCR  0x80000045  No   Level 3 data cache reads
PAPI_L2_DCW  0x80000047  Yes  Level 2 data cache writes
PAPI_L3_DCW  0x80000048  No   Level 3 data cache writes
PAPI_L2_ICH  0x8000004a  No   Level 2 instruction cache hits
```

```
PAPI_L2_ICA  0x8000004d  No   Level 2 instruction cache accesses
PAPI_L3_ICA  0x8000004e  No   Level 3 instruction cache accesses
PAPI_L2_ICR  0x80000050  No   Level 2 instruction cache reads
PAPI_L3_ICR  0x80000051  No   Level 3 instruction cache reads
PAPI_L2_TCA  0x80000059  Yes  Level 2 total cache accesses
PAPI_L3_TCA  0x8000005a  No   Level 3 total cache accesses
PAPI_L2_TCR  0x8000005c  Yes  Level 2 total cache reads
PAPI_L3_TCR  0x8000005d  Yes  Level 3 total cache reads
PAPI_L2_TCW  0x8000005f  Yes  Level 2 total cache writes
PAPI_L3_TCW  0x80000060  No   Level 3 total cache writes
PAPI_SP_OPS  0x80000067  Yes  Floating point operations; optimized to count scaled single
↪  precision vector operations
PAPI_DP_OPS  0x80000068  Yes  Floating point operations; optimized to count scaled double
↪  precision vector operations
PAPI_VEC_SP  0x80000069  Yes  Single precision vector/SIMD instructions
PAPI_VEC_DP  0x8000006a  Yes  Double precision vector/SIMD instructions
PAPI_REF_CYC 0x8000006b  No   Reference clock cycles
-----------------------------------------------------------------------------
Of 59 available events, 18 are derived.
```