Project 1

Computação Paralela e Distribuída

Turma 5 - Grupo 15

- David Amorim Cordeiro - up202108820
- Diogo Silveira Viana - up202108803
- Tiago Tavares Simões - up202108857

Introduction

This project aimed to study the memory hierarchy's impact on processor performance during large data access. It analysed and compared different matrix multiplication algorithm implementations with varying square matrix sizes. Part 1 tested these implementations using a single core, while Part 2 compared two parallel versions of the same algorithm. Performance metrics included execution time and L1 and L2 Data Cache Misses for Part 1, and execution time, mflops, speedup, and efficiency for Part 2. Parallel versions were implemented using OpenMP.

Part 1

Line x Column

The first implementation involves multiplying two square matrices by performing element-wise multiplication on each row and column, then adding the results to create a new matrix. In C++, the function "OnMult" declares matrices a, b, and c by allocating memory blocks of a specified size in the heap, initialising a with 1 and b with values from 1 to size. The multiplication itself is performed by these 3 nested for loops, in the following code block:

```
C/C++
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;
```

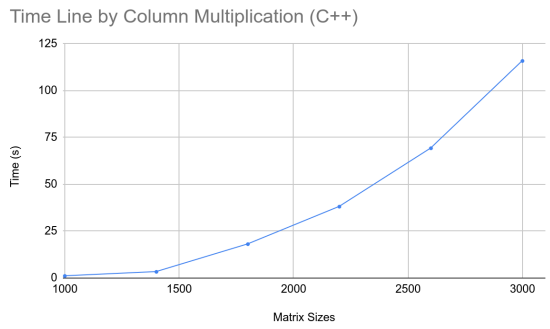
We used C's clock to determine the execution time of the algorithm and print the first 10 elements of the matrix to verify its correctness.

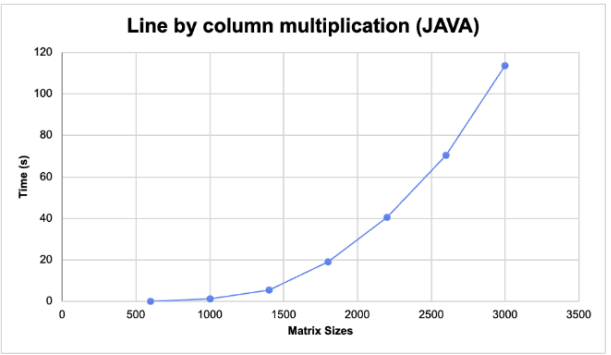
```
C/C++
Time1 = clock();
...
Time2 = clock();
sprintf(st, "Time: %3.3f seconds\n", (double)(Time2 - Time1) / CLOCKS_PER_SEC);
cout << st;
// display 10 elements of the result matrix to verify correctness
cout << "Result matrix: " << endl;
for(i=0; i<1; i++)
        for(j=0; j<min(10,m_br); j++)
                cout << phc[j] << " ";
cout << endl;
...
```

We compared Java with C++, trying to make it as close to the C++ function as possible, especially in representing a matrix as a single memory block, avoiding the overhead of nested Array objects.

We compiled our C++ code with the -O2 optimization flag and tested square matrices from 600 to 3000 in increments of 400. We recorded times for both C++ and Java. In C++, we used papi to track L1 and L2 data cache misses, but we're omitting those results until the next section.

The results are displayed in the subsequent graphs:





Conclusions

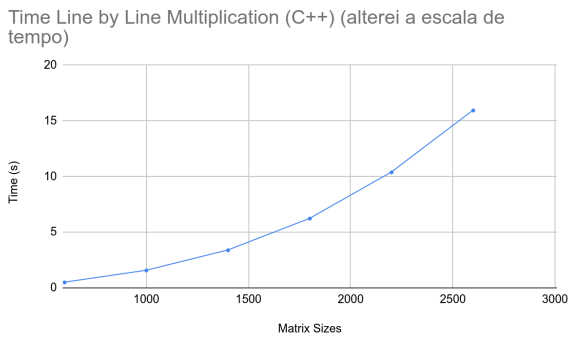
Execution time is similar in both languages tested. Java's automatic garbage collection, which could have been a differentiating factor, introduced minimal overhead. As matrix size increases, time increases exponentially, rendering this implementation inefficient for large matrices. This is due to an increase in the number of reads for elements in the same column (more and larger columns), leading to a significant increase in data cache misses and poorer performance.

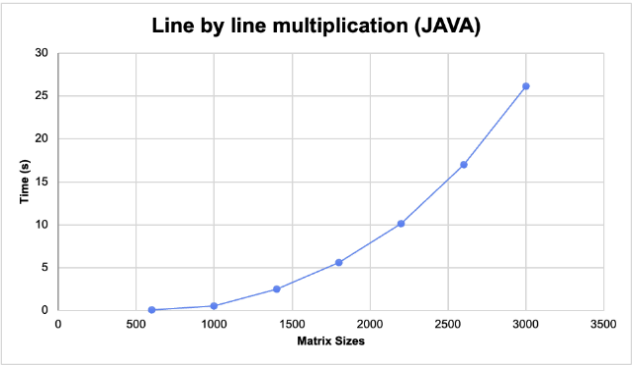
Line x Line

The second algorithm implementation, “OnMultLine”, alters the matrix product computation. Each kth element of the ith row in the first matrix is multiplied with every jth element of the kth row in the second matrix, calculating the value of each element of the i^{th} row in the result matrix. The for loops are reshaped to partially calculate all row elements in the result matrix simultaneously, rather than fully calculating each one at a time. This is reflected in code by a minor change, making the second innermost loop vary on k instead of j. All other aspects remain unchanged.

```
C/C++
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];
```

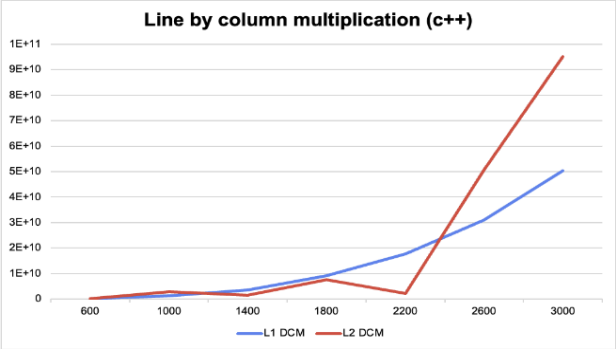
This was the graph obtained for the times registered, in the same range of sizes as in the previous sections:

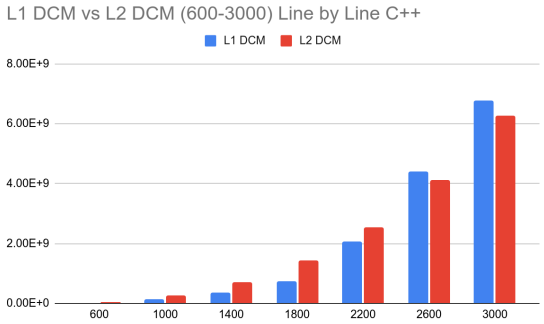




We can immediately see that for the same range of sizes, the execution time increases much more linearly, in this second implementation (both in C++ and in Java). Which, in the case of a matrix size of 3000x3000, can result in times around 7 times smaller.

To understand the reason for this speedup, we need only to compare the graphs for the L1 and L2 DCM, between this and the previous implementation:





There are, on average, around 10 times fewer L1 data cache misses in the line x line implementation than in the line x column, with the number of L2 DCMs varying from around the same (matrix size of 2200), to 15 times smaller (matrix size of 3000).

Conclusions

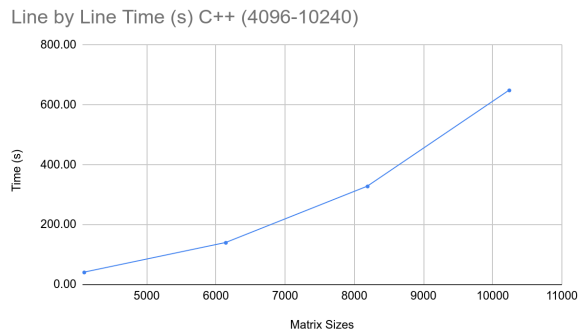
Cache memory, based on spatial and temporal data locality, reads an entire cache line and its neighbours when an element is read, aiming to enhance efficiency as cache reading is more efficient than memory reading. The first algorithm implementation requested a different column element in the second matrix in each iteration, triggering a data cache miss and bringing a cache line with the value's neighbours into memory, only to be discarded in the next iteration. This worked against data locality and became more noticeable as the matrix size increased. The revised implementation addresses this by performing calculations while reading elements from the second matrix in a row-wise manner, reducing cache misses.

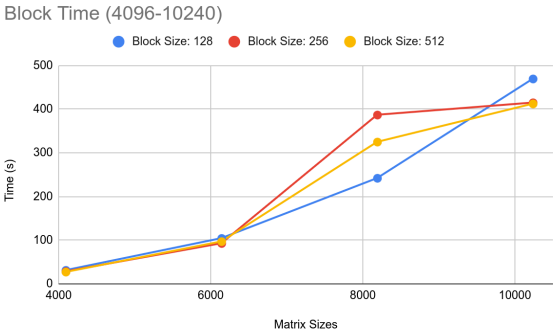
Block

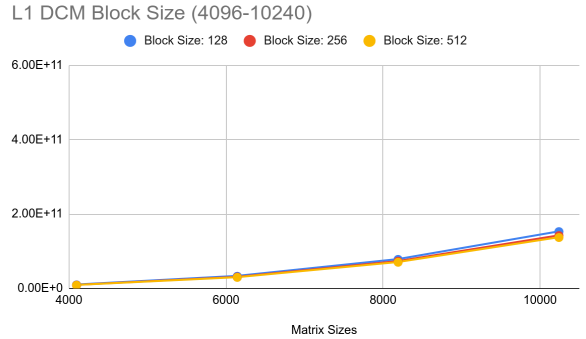
Finally, this third implementation of the matrix multiplication algorithm, uses another 3 outermost loops, over the previous version, to iterate over the larger matrix, localised in blocks of smaller sizes (in our case 128, 256 and 512). The idea is again to work in favour of data locality and the values in the cache by working with elements that are closer together, as well as using the concept of "divide and conquer", in the algorithm.

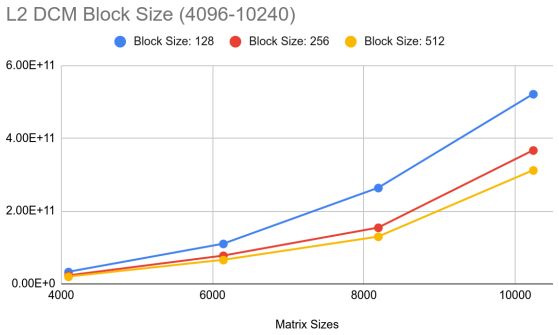
```C/C++
for (i=0; i<m_ar; i+=bkSize)
    for (k=0; k<m_ar; k+=bkSize)
      for (j=0; j<m_br; j+=bkSize)
        for (int ii=i; ii<i+bkSize; ii++)
          for (int kk=k; kk<k+bkSize; kk++)
            for (int jj=j; jj<j+bkSize; jj++)
              phc[ii*m_ar + jj] += pha[ii*m_ar + kk] * phb[kk*m_br + jj];
```

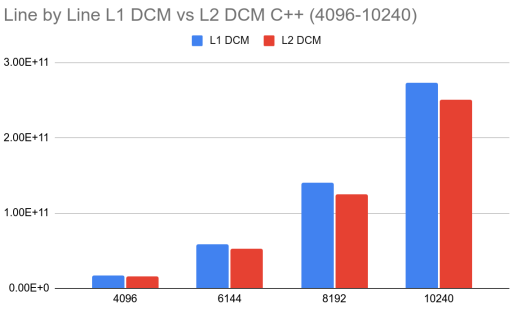
Here are the results obtained, for matrix sizes of 4096 to 10240, with an increment of 2048, and block sizes of 128, 256 and 512, for each:











Conclusions

Better results can be achieved with this approach compared to the line-by-line method for certain matrix sizes. However, results vary significantly depending on the matrix size. For sizes 4096 and 10240, performance improves with each block size increase, but not for a size of 8192, where a block size of 256 performs worse than a block size of 128. Cache misses decrease as block size increases, but there's a trade-off between block size and the type of calculations performed. Increasing block size reduces the number of blocks the matrix can be divided into, leading to fewer multiplications of smaller matrices, but also means larger sub-matrices are multiplied, which takes longer. This trade-off can be beneficial in some cases, but not in others.

Part 2

In this part, we tested and compared the 2 OpenMP parallelized versions provided to us in the project's instructions.

```C/C++
#pragma omp parallel for
for (int i=0; i<n; i++)
        for (int k=0; k<n; k++)
                for (int j=0; j<n; j++)
```

The `#pragma omp parallel for` directive tells the compiler to execute the following loop in parallel. OpenMP will create a team of threads, and each thread will execute a portion of the iterations of the loop independently and concurrently with the other threads, which means that multiple rows of the resulting matrix can be computed simultaneously.

```C/C++
#pragma omp parallel
for (int i=0; i<n; i++)
        for (int k=0; k<n; k++)
                #pragma omp for
                for (int j=0; j<n; j++)
```

The second version of the code differs from the first in the way it applies parallelism. In this version, the `#pragma omp parallel` directive creates a team of threads at the beginning, and then the `#pragma omp for` directive distributes the iterations of the innermost loop (j) among these threads, as opposed to the iterations of the outermost loop, in the previous version.

We ran several tests, and computed the following performance measures:

$$MFLOPS = \frac{2 \times n^3}{T}$$

$MFLOPS$: Million Floating Point Operations Per Second
n: matrix size (number of lines/columns)
T: execution time

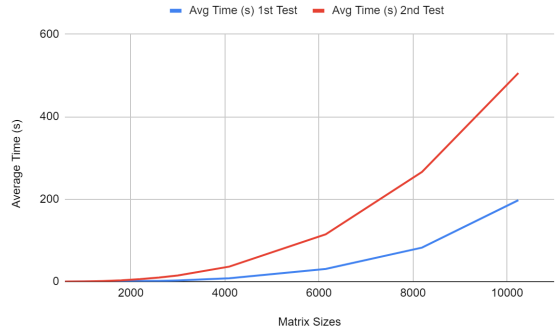
$$S = \frac{T_{sequential}}{T_{parallel}}$$

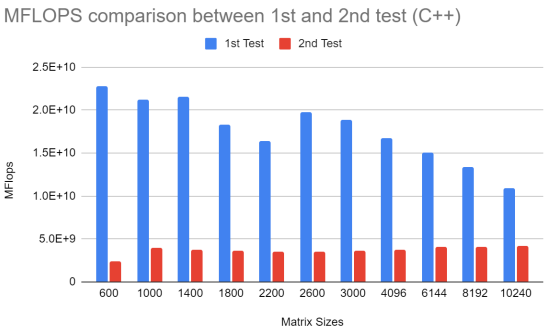
S: speedup
$T_{sequential}$: sequential execution time
$T_{parallel}$: parallel execution time

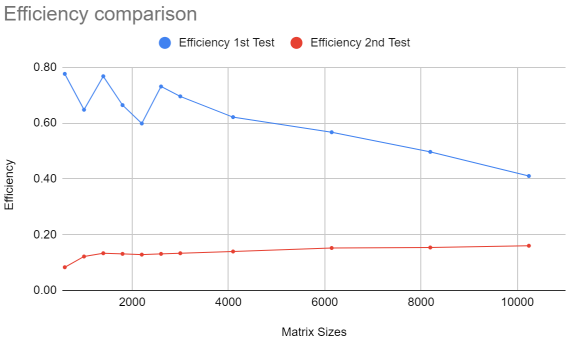
$$E = \frac{S}{p}$$

E: efficiency
S: speedup
p: nº of cores

These were the results obtained:







Conclusions:

Observing the graphs we can see that with the first version, we gain a significant speedup over the sequential version (more than 5x in some cases). It does decrease, as the matrix size increases, due to also increasing synchronisation, but for all our tests it performs better than the regular version. The second one, however, has little to no apparent gain over the sequential version. This is because, by parallelizing the iterations of the innermost for loop, we end up gaining a lot of overhead due to thread synchronisation at each iteration.

Tables & Graphs:
https://docs.google.com/spreadsheets/d/1p4eCc69VyLs1qHcsjbrdsOqKPra23KBXP-SUhKTsJpE/edit?usp=sharing