FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Autotuning Parallel Application in Heterogeneous Systems

João Alberto Trigo de Bordalo Morais

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Jorge Manuel Gomes Barbosa

February 10, 2017

Autotuning Parallel Application in Heterogeneous Systems

João Alberto Trigo de Bordalo Morais

Mestrado Integrado em Engenharia Informática e Computação

February 10, 2017

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

Abbreviations

CPU	Central Processing Unit
GPU	Graphic Processing Unit
FPGA	Field-Programmable Gate Array
OpenCL	Open Computing Language
CHC	Cooperative Heterogeneous Computing
OpenMP	Open Multi-Processing
WWW	*World Wide Web*

Chapter 1

Methodology

1.1 Introduction

According to the state of the art presented in chapter two, there are many means to, in some kind of automatic way, improve an applications performance. During my research, my focus was to find ways to automatically enhance the execution time in applications and programs. For this propose, Kremlin had a crucial impact in other to understand the viability of automatically parallelise code.

To study the utility and impact of automatic tools, the matrix multiplication algorithm will be used as a reference to make the performance comparison between original algorithm, an expert manually parallelising the original algorithm and using the Kremlin's indication to parallelise the original algorithm.

To increase the credibility of this experiment, two similar algorithms for the matrix multiplication were used. As mentioned and explained in the chapter two, there is the traditional way of multiplying square matrices, naming as a quick reference *Mult* algorithm, see in the appendix's list 3.1 this algorithm implementation, written in C++ programming language; and the optimized algorithm that multiplies each element from the first matrix with the correspondent line of this matrix element but for the second matrix, naming this algorithm as *MultLine*, see in the appendix's list 3.2 this algorithm implementation, written in C++ programming language. These algorithms differs from one another in the variables preparation and the order of the loops, which differs in the memory access. The *MultLine* algorithm is an optimized version for matrix multiplication because it takes advantages of what is preloaded in cache and starts pre-calculating the intermediate values that will lead to the final and correct result of the multiplication, which means that won't be needed to load unnecessary values to cache memory and/or will need afterwards.

Several experiments were conducted to understand the influence of Kremlin's indications versus code being manually parallelized by an expert. The data's length, in this case, the matrix size; the number of threads used and if the code was parallelized were the used metrics to evaluate the results, based on a comparison of the execution time, changing these variables.

In this chapter it is explained the methodology and the steps followed that guided to report in the Results and Discussion chapter the results and conclusions obtained from the obtained outcomes coming from de experiences. This chapter also includes detailed information of the acquired data from the conducted experiences, as in, how it is obtained and its meaning; also includes the methods that were used to analyse the obtained data and the reason behind those methods; and, in the end, how the data was validated in order to verify its correctness, accuracy and reliability.

1.2 Research Method

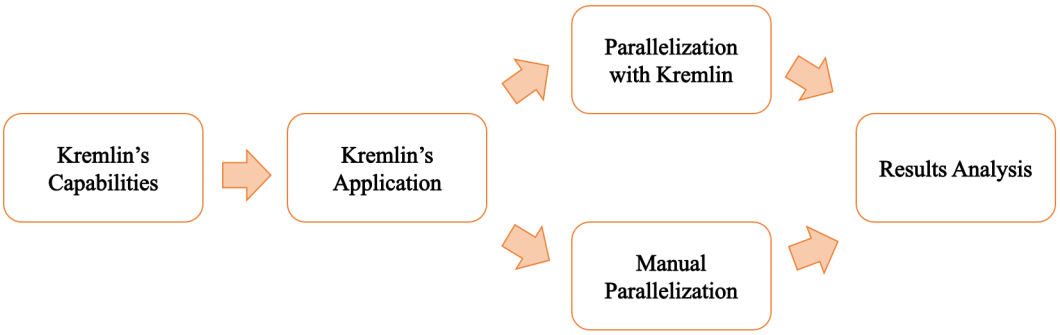

Figure 1.1: Followed up methodology

In the figure 1.1 is outlined the steps that were followed to study the impact of the code being automatic parallelised. This methodology has five states. Firstly and using simple applications, an evaluation was made to Kremlin in order to understand how to use this software tool and evaluate the results that Kremlin can achieve, for instance, if it has similar results comparing with an expert parallelizing manually the same code. After this, Kremlin will be applied to a set of codes with specific characteristics.

Before applying Kremlin, I manually parallelised the same sample of code in order to evaluate the results and, afterwards, compare with the Kremlin output. Since these states (the experiences with Kremlin and the manual code parallelization) required several attempts there were transitions between manual and Kremlin states.

Finally, in the last state, after several attempts and tuning exercises applied to both code cases (manual and kremlin), data was collected from this experience to evaluate and validate its correctness in order to conclude how helpfull can automatic parallelization can be.

To sum up, this methodology as three main stages: learn and evaluate Kremlin's uses and results; finding the tunning parameter through several attempts using Kremlin's outputs and manually parallelize the application's code; and, in the end, compare and analyze the results in every attempt to take conclusions;

1.2.1 Deep learn on Kremlin's usage

Firstly, and according to all the presented tools/frameworks mentioned in the second chapter, *Achieving the Highest Processing Power*, in the *Using Code Parallelization* section **??**, Kremlin was chosen because it presented the best results, easy usage and accessibility comparing to the other presented ones regarding the way the tool/framework could automatically parallelise code **??**.

Kremlin is a tool that indicates, for a sequential program, which block can be parallelised and teorical calculated values, such as, overall speedup; self parallelism for each block; the ideal time reduced for each block, in percentage; the actual time reduced for each block, in percentage; and the block coverage considering the whole program, in percentage. The way this tool was used is as it follows: first, an object file, **.o* extension, is required from the compilation of a sequential code. Afterwards, it is time to use the Kremlin's compiler with the generated object file so that it can profile the application. In order to do so, Kremlin's compiler runs the program as it is supposed to work. Now that the profiling is done, Kremlin generates the indications that should be followed to parallelise de provided sequential code. It also includes the blocks that can be parallelised and the impact of this theoretical parallelization with the calculations done during the profiling. Since this parallelization report is done, the program has interpret it, confront with the code an apply it.

The Kremlin's usage seems easy, linear and fast forward, however it has some limitations that I experienced during the learning of Kremlin's capabilities: Kremlin's requires a specific environment mentioned in the Kremlin's repository [**?**]. It requires several software, libraries, compilers installations and a modern Unix operative system as its bases, such as MAC OS, RHEL 7 or other Linux distribution compatible with the software specification required. Additionally, when installing the Kremlin's tool, some minor fixed are required in order to successfully install.

From the experiences that I have been through, Kremlin has another limitation: it can not compile and profile all kind of programs: it can only profile programs that use C/C++ as its programming language; programs that take advantage of data structures from the *Standard Library*, such as, stack, list, priority queue, queue, list, hash table, map, multimap, heap, etc., since it doesn't recognize these structures; another Kremlin's limitations is its capability of compiling programs that have a deep function call level greater that seven. By deep function call level I mean the depth a function has starting from the *main* function until it is called, like a tree function call tree. For instance: in a program there is the *main()* function, a first level, that calls a *foo1()* function, and this function calls a *foo2()*, that this calls a *foo3()* function, and so on. In this case, the deph of *foo3()* function is four. Another small issue that kremlin's tool has is the definition of the iterator variable used in the *for*'s loops must be defined outside of the loop, as it is in C programming language.

1.2.2 Kremlin's application in specific code samples

After all the experiences made in the previous state and as mentioned in the introduction of this chapter, the matrix multiplication algorithm was used to see the potentialities of Kremlin's compiler to profile and indicate the regions that can be parallelised. So, Kremlin was used in two similar, relatively in the code structure, matrix multiplication codes. The reason behind the choice was because these two versions of the algorithm are really close to one another, which means that the testing environment is similar to one another, consequently, the results should be similar.

1.2.3 Code parallelization with Kremlin's data

Kremlin's tool just points the regions/blocks where the program can be parallelised. In both code samples there are various numbers of inner *for* loops, for the *Mult* code there are three inner *for* loops, which one of them has a degree of three and the rest a degree of two; and for the *MultLine* code there are four inner *for* loops, which one of them has a degree of three and the rest a degree of two aswell. At this time, after reading the report provided by Kremlin's tool, the developer must locate the loops, apply which loop should be parallelised, if it should be, and in case of inner *for* loops, what loop should be parallelised using the OpenMP *pragma* directives.

In my case, I followed all the instructions provided by Kremlin, located all the *for* loops blocks indicated by kremlin's tool and applied the OpenMP *pragma* directives.

Following the two reports, 3.8 3.9, and looking at the code's structure for both codes, it can be divided in 2 bigger parts: the *for* loops used for matrices initialization and the *for* loop for the matrix multiplication. With this information, code understanding and using an expert knowledge, the code parallelization was done.

1.2.4 Manually Code parallelization

In order to not be manipulated by the Kremlin's indications, the both codes were previously manually parallelised, this way it was guaranteed that the expert parallelization wasn't bias nor influenced.

For this parallelization, as mentioned before, it requires knowledge in, firstly, matrix multiplication algorithm; code understanding; best practice in what can and can not be parallelised, taking into account the overhead that could occur; and understand the thread behaviour in order to make it do the proper job without jeopardizing the programs outputs and/or possible caused overhead.

Analysing the code, only the *for* loop for the matrix multiplication was parallelised and applied the OpenMP *pragma* directives applied to the innermost *for* loop. In this case, each code as a slight difference because for the *Mult* code each value of the result matrix but be calculated individually, so each thread is responsible for it and must treat that value as a private variable that isn't shared by the other threads. In the opposite, and since the *MultLine* code calculates the values by adding the multiplication to the respective matrix's cell, each thread don't need to have their own private variable.

The bigger part of the code responsible for the matrices initialization wasn't parallelised, unlike in kremlin's case, because the gain would be noticeable on a large matrix size or it even could cause thread trampling, which could lead an overhead increase.

1.2.5 Results analysis

To obtain the final execution time of each implementation (Original Matrix Multiplication 3.1, Original Matrix Multiplication by line 3.2, Manual Matrix Multiplication by an expert 3.3, Manual Matrix Multiplication by line by an expert 3.4, Kremlin Matrix Multiplication **??** and Kremlin Matrix Multiplication by line 3.6), these six implementations suffered many modifications and tweaks since this process is a try-error until it is found the believed best parallelization. It is hardly possible to parallelise a whole program at the first try.

After compiling all these implementations and registering all the execution time for different matrix sizes and number of threads, in this case not applied to the original codes, this data was organized so it could be used to compare results and conclude about the performed experiences.

1.3 Data collection from executed experiences

From all the developed work, the obtained data can be divided in to moments: Kremlin's indications reports and the execution times for the six code variations.

1.3.1 Kremlin's indications reports

For the *Mult* 3.8 and *MultLine* 3.9 codes were generated a report done by Kremlin. This report displays for each parallelizable block:

Time reduced percentage of time reduced if parallelization is implemented;

Ideal Time reduced percentage of ideal time reduced if parallelization is implemented;

Coverage percentage of sequential execution time in a block;

Self Parallelism amount of parallelism in a block;

Parallelism type classification of the parallelizable block;

Loop location block lines range of the parallelizable block;

Function location function name of the parallelizable block, mentioning the line where the function is defined;

File location file name of the parallelizable block, mentioning the line where the function is called;

The guidelines given by the report must be followed by the order it is suggested because the first detected block has the biggest impact in programs performance and should be parallelised first. The Coverage and Self Parallelism are metrics that indicates the speedup of the block, which, consequently, interferes with the time reduced. This block speedup must be equal or less then the overall program speedup, based on Amdahl's law, and can be calculated as it follows [**?**]:

$$speedup \leq \frac{1}{(1 - Coverage) - \frac{Coverage}{SelfParalelism}} \tag{1.1}$$

1.3.2 Execution times for the code variations

After running the six implementations, the results can be divided in three major groups and each group is has the respective implementations of the *Mult* and *MultLine* algorithms. Each group has its own experimental environment, with their own variables and their own meaning according to the given context.

1.3.2.1 Original code

The Original code only variable is the matrix size. This group is a reference group to compare the results of the others group and to quantify the impact of the others groups. The results of this group are the execution times running both matrix multiplications algorithm versions.

1.3.2.2 Manual code parallelization

The Manual code parallelization by an expert has as variables the matrix size and number of threads used for each run. This groups's results are the execution times for both matrix multiplication algorithm versions. These results were used to confront with the following group in order to evaluate the improvement that a guided parallelizations, using Kremlin, can for an application.

1.3.2.3 Kremlin's code parallelization

Like the previous group, Kremlin's code parallelization indications use the same variables and provide the same type of the results. However, this group is responsible to define if it is advantageous to use software tools to help in automatically parallelise code.

1.4 Data analysis method

Analysing data is a very important stage because the it is necessary to have good and correct conclusions. From the experiments made a lot of data as been generated and without a proper organization it is hard to understand the meaning, therefore, hard to take good conclusions from its analysis. There are three groups of code and in each group two different algorithms implementation. In order to make a good analysis from the execution times for each situation and comparing with the others situations, the collected results were stored in tables along with the

experimental related variables. Additionally, calculations were required, such as,the difference between executed times between groups and algorithm implementation; the ratio between these executed times; the percentage of the increase/decrease for these executed times; and the impact in the execution time that the others two groups have comparing with the Original code group. After these data manipulation, the best way to analyse all this generated and calculated data is by a dispersion plot. Using a dispersion plot transforms data into information visually understandable and easier to conclude because these plots display the variation of results according to the experimental environment variables.

1.5 Data validation

A considerable amount of data was generated and, more importantly, it is important that this data is scientific correct, or at least there is an explanation. In order to keep its fidelity, it is crucial to validate each and every piece of data. The first measure is to have a critical position every time by questioning if the obtained value makes sense when compared with th theoretical, or expected or referenced value. In this current case, for instance, the Original group implementation is the reference, which means if the other groups have a higher execution time, or the data was some defect caused by some hardware component, or the implementation isn't good enough, or any other reason that can justify the data invalidation.

Criticism can not be the only measure because it could be luck and the gathered data happened to be correct. It is important consistency. To do so, the tests must be performed several times under the same circumstances and with a plausible and considerable amount of values to find patterns. For this particular case it is used a matrix size large enough, [1000,2000,3000,4000], and a wide range for the number of threads, [1,2,3,4,5,6,7,8]. For instance, if the matrix size was small, such as one hundred, the execution time would be so low and with so much error accumulated since the CPU executed fast enough that it couldn't count the time with precision.

Finally, to make reasonable comparisons and analogies between results, the experience environments must have some connection in its variables or environment. Without a connection the data as no meaning, therefore, it turns impossible to take conclusions. In the performed experiences, it was used the same algorithm, Matrix Multiplication, with small modifications in the implementations but with the same structure. Additionally, the environment variables were the same: number of threads and matrix size.

Chapter 2

Results and Discussion

2.1 Introduction

Generally, the first objective a developer has when building software is making it work. After some experience and good practice the development becomes faster, elegant and concerns about its performance. When dealing if performance issues, there is a lot of measures to pay attention from the lowest hardware level to the highest software level. Now-a-days performance is as much important as the creating software, because it makes the work faster, less costs, and, in the end, more revenue. However it is really hard for a single person masters performance as a whole because there are to many variables, conditions, aspects, and realities making humanly impossible mastering everything.

The approach to achieve high performance level is to have handful of expertises in each concrete area. Even so, mastering specific fields in the performance level, it is hard, takes time, lots of effort and most of the times impossible to achieve the perfect performance. Since achieving high level of performance is so important and requires a lot of effort to try to achieve it, then, first of all, is it possible to achieve high performance level in applications in an automatic way? If so, how can it be achievable? Can it totally replace an expert?

Focusing these question to the field of code parallelization, more will rise, not necessarily related to this specific filed: since exists tools, like Kremlin, which help to, automatically, parallelise code, how acceptable are their results? How this specific tool can help in getting one step closer to automatic code parallelization? In which way can Kremlin be better than an expert?

In order to answer all previous questions, this chapter is divided in two main sections: the first section is related to the Kremlin's activity and how is it helpful. The second section is the confrontation of all the gathered data to verify what is better and how can it contribute to the future of automatic code parallelization.

2.2 Kremlin's reports

When Kremlin's compiles and profiles a sequential code, it provides a report with locations of the blocks that can be parallelised. Additional it gives some values that indicates the theoretical gain if the parallelization is implemented.

For the *Mult* and *MultLine* algorithm, Kremlin gave these reports 3.8 3.9, respectively. Taking into account that the manual code parallelization was done in the first place, the risk of being bias is null and, additionally, helps to understand if Kremlin is reporting things correctly.

In this case, Kremlin detected the block code with the most impact on application performance for both implementations (*Mult* and *MultLine*). Additionally, Kremlin's report pointed the locations of more block to be parallelised, however, the impact of these blocks being parallelised might have a low impact or jeopardizing the applications' performance, also for both implementations. The impact of the parallization made in the others block is analysed in the next section because a verdict can be made after comparing the execution times of the Manual's group against Kremlin's group.

The justification behind these analysis is based on the *time reduced*, *ideal time reduced*, *coverage* and *self parallelism* values and the block location, provided by the report, comparing with the expected result and manual code parallelization by an expert, in both implementation.

Getting a close look in these reports, at the left side is *Mult* report values, 3.8, and on the right side is *MultLine* report values, 3.9:

Table 2.1: Kremlin's report values for the matrix multiplication block

Time reduced	66.38%
Ideal time reduced	70.96%
Coverage	88.51%
Self parallelism	5.05

Time reduced	63.01%
Ideal time reduced	63.20%
Coverage	84.02%
Self parallelism	4.03

In both reports, the high percentage of the reduced time and reduced time means that parallelizing these blocks the execution time of this block is, theoretically, reduced in between those two values.

Taking a close look in the others blocks, their locations refers to the matrices initialization and the values of timed reduced and ideal timed reduced are really low, around 3% in both implementations, which means that the improved performance is insignificant and might cause delay in during de applications executions. However, this situations is confirmed in the next section.

2.3 Comparison between Original, Manual and Kremlin

The previous section has an important role because the reports credibility and correctness influences the the results in this chapter, consequently, lead to misguided and wrong conclusions. Since the report gave correct feedback and it is well justified, the following values are valid.

To get a satisfying answer for the initial questions, it is necessary to, in first place, understand context of each experience and respective results; following the evolution and the comparison is made between data. So, in a first instance, each group (Original, Manual, Kremlin) are going to be analysed individually to stablish the context and basis knowledge. Then, the second subsection will focus in measuring the impact of Manual and Kremlin group have to the Original group to prove that these measures, in practical terms, have and huge impact improving applications performance. After this knowledge also as been established, the results will prove if the kremlin's guidelines make the code with better performance comparing to the expert's results parallelizing the code manually, and respond to the question if automatic is a reliable and good practice to parallelise code.

2.3.1 The Three groups individually analysed

As mentioned in the previous chapter, each group has its purpose based on the variables used and results obtained. To understand the overall impact of these implementations, it is important to firstly understand the experiences that were made in each group separately and analyse their results, step by step.

2.3.1.1 Original

This group has the sequential code version of the *Mult* and *MultLine* algorithms. As mentioned earlier, this group stablish the base reference for the execution time. From now on, all experiences should have better performance, unless there is an explanation for the Original Group has better in some particular cases better results.

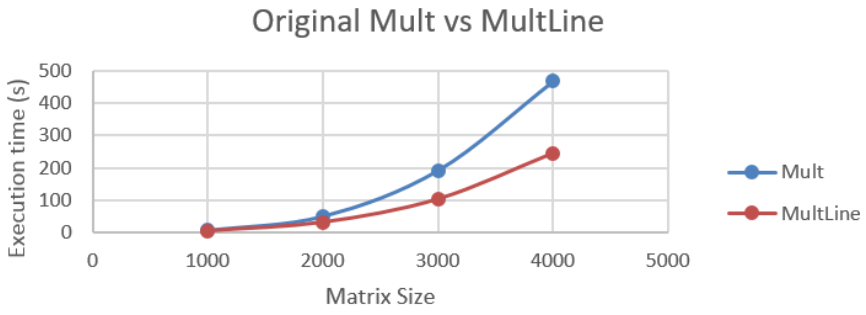

Figure 2.1: Dispersion plot that represents the evolution of execution time with the matrix size increase in both *Mult* and *MultLine* sequential implementations

According to Figure 2.1, the *MultLine* algorithm has better results the bigger the matrix size is, as expected, since this algorithm takes advantage of the values preloaded in memory cache.

Another aspect noteworthy is the increase of the function inclination variation, in both implementations, as the matrix size increases. Specially for the *Mult* implementation. This is related

to the memory cache size. The smaller the size the higher will be de variation of the function inclination.

2.3.1.2 Manual

Manually parallelizing a code requires a lot of effort, time and know-how since the way it is done requires de expert to understand the code, have practice in detecting potential parallelized blocks of code, identify the best way to parallelise those blocks and test the work until it gives a reasonable result. So, this group corresponds to this situation and, in a theoretical speaking, has the bests results.

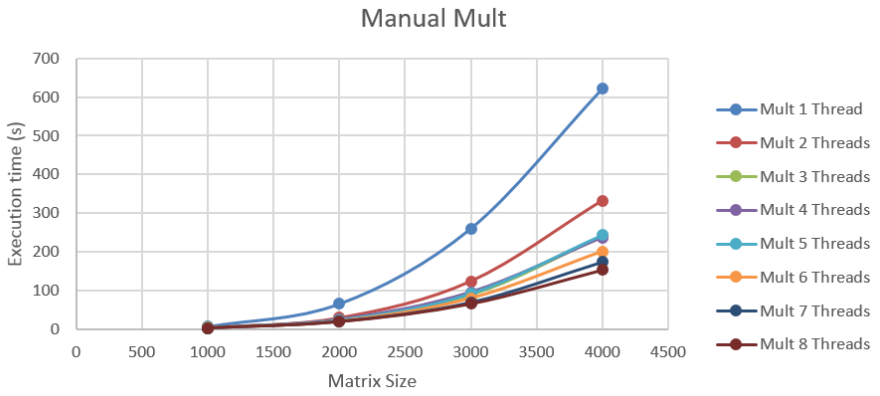

Figure 2.2: Dispersion plot that represents the evolution of execution time with the matrix size increase in function of the number of threads. Version of the manual code parallelization for *Mult* algorithm

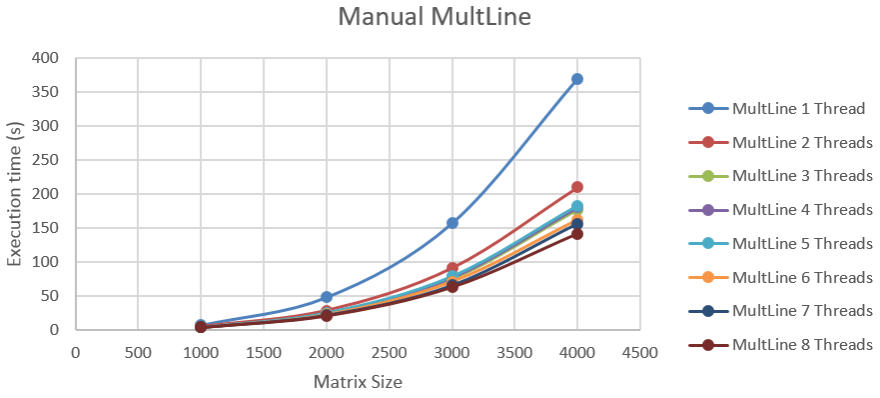

Figure 2.3: Dispersion plot that represents the evolution of execution time with the matrix size increase in function of the number of threads. Version of the manual code parallelization for *MultLine* algorithm

Figure 2.2 and Figure 2.3 have similar behaviours including the fact that using eight threads makes the application with the bets performance because the executed time is in inferior as long as the matrix size increases.

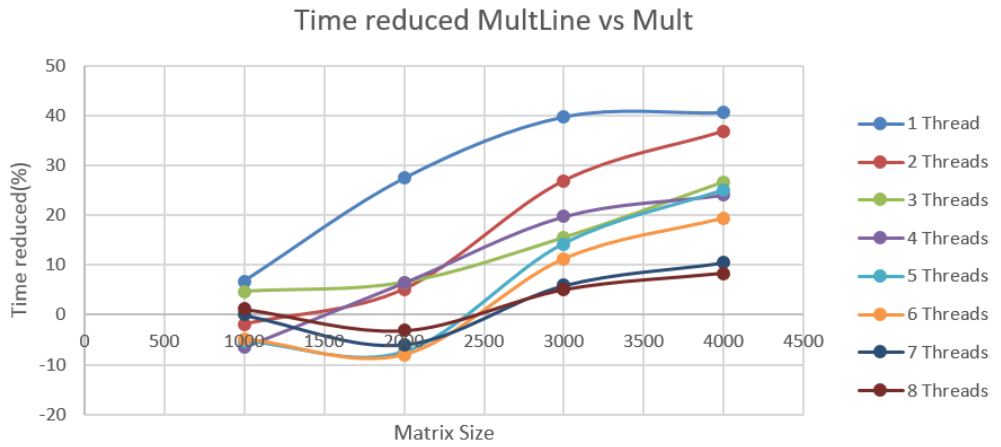

Figure 2.4: Dispersion plot that represents the evolution of time reduced with the matrix size in function of the number of threads. Versions of *Mult* and *MultLine* manual implementations.

However, there is a difference between these two plots is the value of the execution time. This difference can be analysed in Figure 2.4. Time reduced is a percentage of how much the *MultLine* implementation reduces comparing to the *Mult* implementation.

It is a fact, observed in Figure 2.4, that increasing the matrix size, the time reduced tends to a certain value which is independent of the number of threads. This value is the ceiling where *MultLine* algorithm can not be better than *Mult* algorithm for the same hardware components; meaning the existence of a hardware limitation (processor power, memory ram and cache size) since the increase of the matrix size will, proportionally, increase the number of loads, writes and cache missed for both algorithms.

Another fact is that the higher the thread number, lesser will be de timed reduced, and lesser will be the executed time, also related to the maximum capacity of threads a multi-core processor can have working at the same time.

Another noteworthy fact is , for low values of matrix size, 1000 and 2000, and the higher the number of threads being used, the time reduced is negative, meaning executed time for *Mult* implementation is lower than *MultLine* implementations, concluding that *Mult* implementation is better suited for low size data in case a hight number of threads are being used. This is due to many threads are being used simultaneously and they are trampling each other in order to complete their task, which increases the overall overhead, and so the reason behind ne time reduced negative value.

2.3.1.3 Kremlin

This group is constituted by the result of the implementations indicated by Kremlin's report. The experiences conducted in this group and the respective obtained results will demonstrate if Kremlin can actually bring acceptable results.

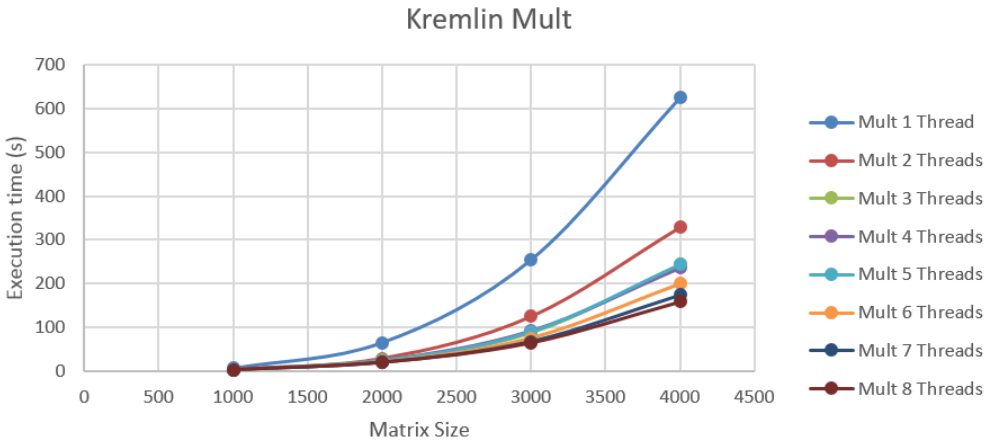

Figure 2.5: Dispersion plot that represents the evolution of execution time with the matrix size increase in function of the number of threads. Version of the Kremlin code parallelization for *Mult* algorithm

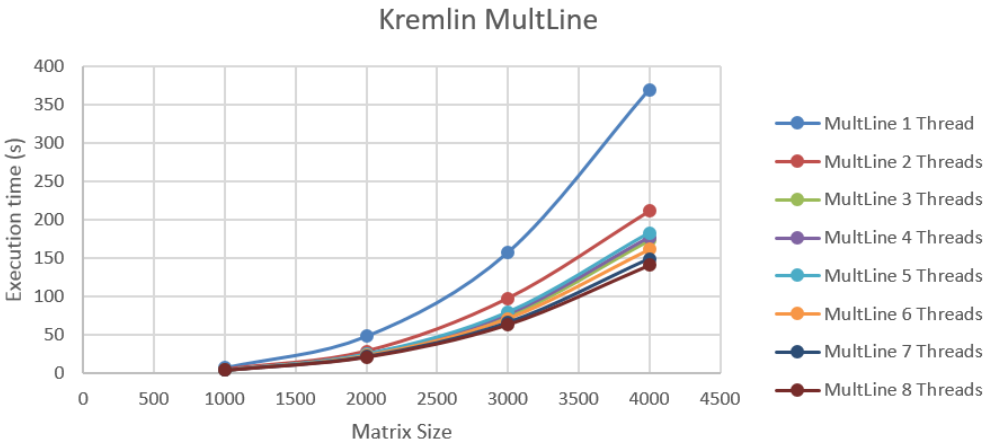

Figure 2.6: Dispersion plot that represents the evolution of execution time with the matrix size increase in function of the number of threads. Version of the Kremlin parallelization code for *Multline* algorithm

Figure 2.5 and Figure 2.6, as expected, have similar behaviour compared to manual group, since the code samples from both groups have the same structure and the experimental environment is the same: same experimental variables(matrix size, number of threads, parallelised code) and same result type (execution time).

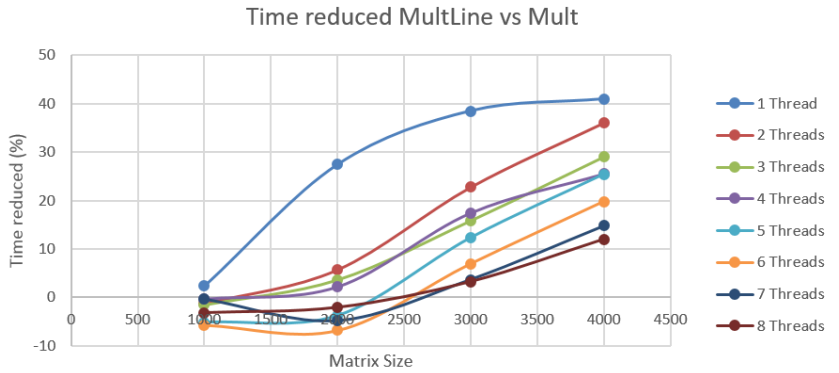

Figure 2.7: Dispersion plot that represents the evolution of time reduced with the matrix size in function of the number of threads. Versions of *Mult* and *MultLine* Kremlin's implementations.

Additionally, and for the same reason, Figure 2.7 also has the same behaviour has the manual group and, therefore, the analysis is the same.

2.3.1.4 Overall review

Through the analysis of these three tests groups, separately, and since these groups were tested under the same circumstances, it is possible to conclude that using these versions of the matrix multiplication algorithm do not jeopardize the results, since they have similar behaviours,moreover, they increase assurance and credibility for the following up analysis and conclusions. It is because of the similarity of behaviours that this comparison is valid and correct, even so the performance is different, which was expected, as explained before.

2.3.2 Measuring the performance's impact using code parallelization

The conducted analysis presented in previous section explains the characteristics of each group and the plausible connection with each other. To quantify how beneficial can code parallelization be, the next sub sub sections will prove its impact.

The first sub sub section compares how much better was the improvement for Manual group, using the Original group as base. The second sub sub sections compares the same ways as the previous sub sub sections but, instead using the Manual group, it will be the Kremlin group. Finally, overall conclusions will be presented about both sub sub sections.

For this analysis, either Manual or Kremlin groups, the number of threads that will be used is eight since, and for both groups, using eight threads gave the best performance results. This does not mean that the for the other number of threads the results were worst than the Original group, far from that. The goal is achieving the highest performance, so the best results were picked.

2.3.2.1 Comparison between Manual and Original groups

In Figure 2.8 is presented a dispersion plot demonstrating that Manual group has an overwhelming better performance and that performance increases with the matrix size to a certain point, explained in 2.3.1.2. This includes both implementations.

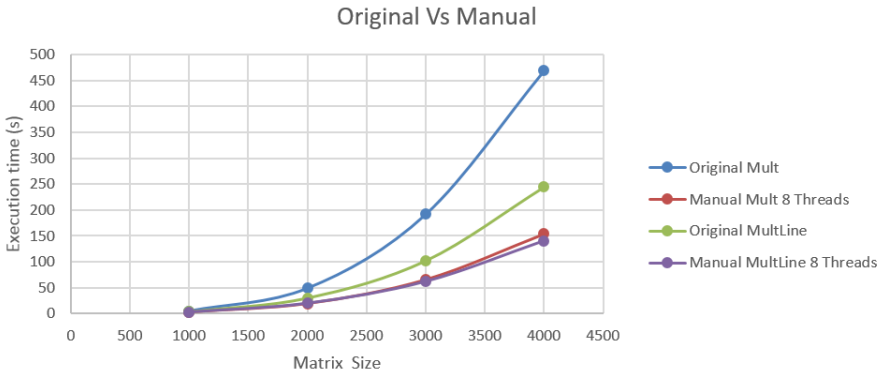

Figure 2.8: Dispersion plot that represents the evolution of execution time with the matrix size in function of Manual and Original.

With this plot it is proved that using parallelization techniques in programs and applications, these can achieve higher performances. In this particular case, it can be more than twice better for the *Mult* case and almost twice for *MultLine*. These values are presented in 2.3.2.3.

2.3.2.2 Comparison between Kremlin and Original groups

Regarding the Kremlin experimental results, in Figure 2.9 is shown a dispersion plot revealing, as expected, and like in the previous sub sub section, that Kremlin's group surpassed far greater the Original group in terms of execution time, in both implementations.

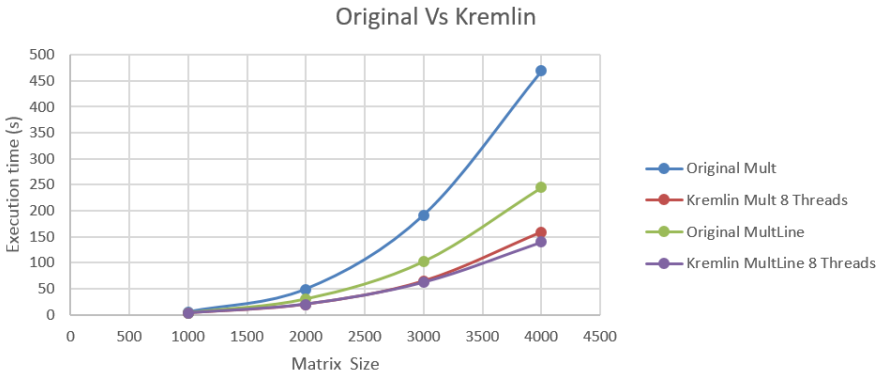

Figure 2.9: Dispersion plot that represents the evolution of execution time with the matrix size in function of Kremlin and Original groups implementations.

Like in previous case, this plot proves that using Kremlin as an automatic tool to help identifying parallelizable code blocks can enhance applications performance and is a good asset because it can save time by indicating where it is possible to make a block parallel, instead of the developer looking for them.

2.3.2.3 Overall comparison

If code parallelization would not bring any beneficial impact in programs and applications, obviously, it would make no sense using it. Additional, and for this concrete case, if Kremlin's performance was worst, this tool would be useless to help in making paralleled code automatically.

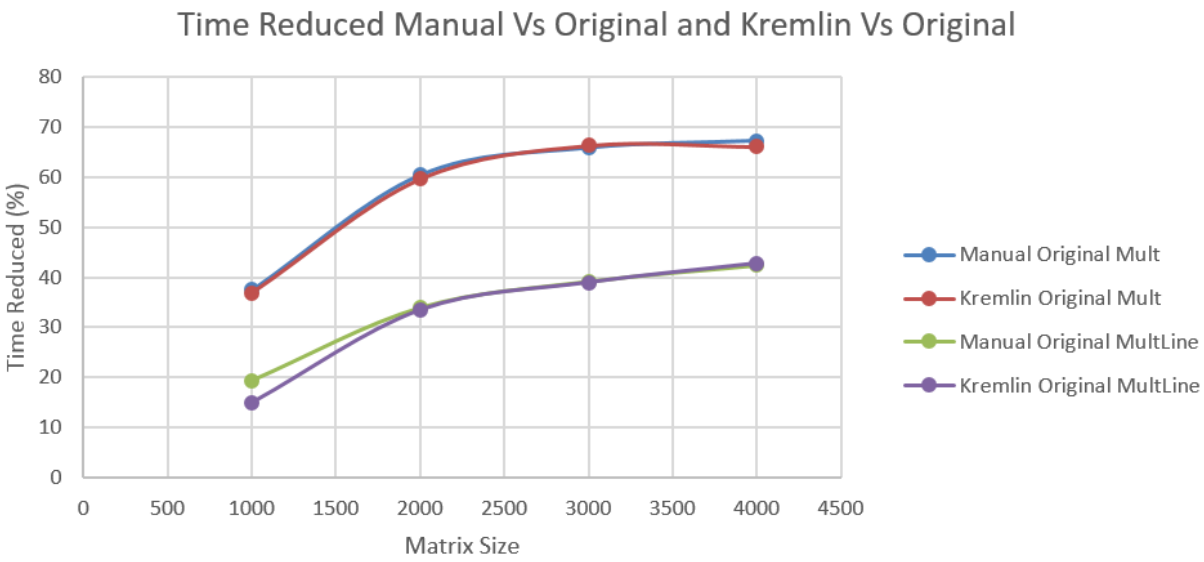

Figure 2.10: Dispersion plot that represents the evolution of time reduced with the matrix size in function of Manual and Kremlin groups implementations.

In Figure 2.10 is compared the time reduced for Manual and Kremlin groups comparatively to Original Group. From this plot it is drawn that *Mult* algorithm takes greater advantages of code parallelization, however *MultLine* has better execution times.

Quantifying such increase, for the *Mult* algorithm the time reduced can be around 67%, meaning that the *Mult* implementation, for both groups, can be 67% faster, more than twice, compared to the Original group, for the same algorithm. Regarding the *MultLine* algorithm, the time reduced can be around 42%. The 25% difference of both implementations represents the impact of the difference on matrix multiplication algorithm versions. Meaning that *MultLine* algorithm makes a huge difference on performance levels.

2.3.3 Comparison between Manual and Kremlin groups

Before starting the comparison between results obtained from running the parallelized code written by an expert and running the parallelized code with Kremlin's indication, the difference between them is that in the Kremlin group, more specifically, in the matrix initializations blocks, they are parallelised as well. These blocks were parallelised because Kremlin detected them, however the theoretical impact for these parallelised bocks could be small, positively or negatively, or even with no effect. Additional, in the expert perspective, they were not taken in consideration.

Now that the three groups were characterized and the comparison between Manual and Kremlin with Original group to stablish viability in both solutions, taking a close look at all this information and analysis and looking at Figure 2.10, the lines in the plot overlap or are really closed to one another in both algorithms. This observation confirms what is expected, which is manually parallelizing a code by an expert or following Kremlin's instructions gives the same results, approximately. However, they are not exactly the same because a small modifications was done to Kremlin's group code.

In order to evaluate if there is really a difference in result, the following plots present the the execution time ratio between Manual and Kremlin's group, for both implementations.

To correctly analyse Figures 2.11 and 2.12, it is important to visualize the range of the values:

Table 2.2: Interval of values for *Mult* (left table) and *MultLine* (right table)

Lower bound	0,966
Upper bound	1,049
Interval size	0,083

Lower bound	0,936
Upper bound	1,073
Interval size	0,137

Looking at the tabulated values, there are cases where Kremlin group had better performance then the Manual group, since there are values less then 1. However, looking at the interval size, for both cases, they are really small, meaning that the execution time for both groups is similar.

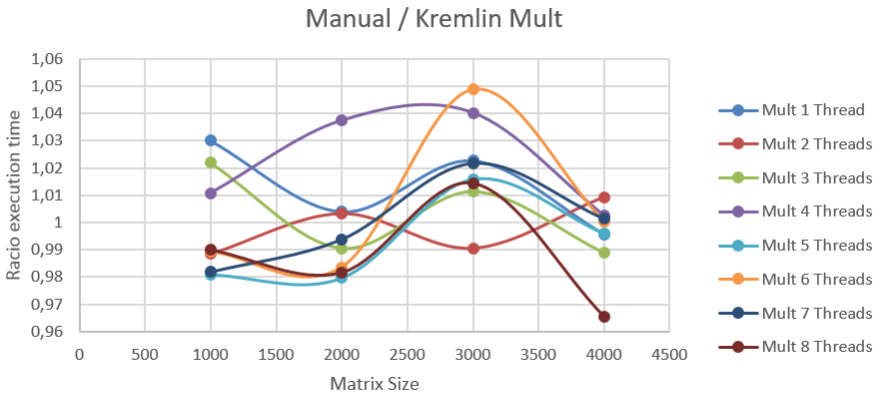

Figure 2.11: Dispersion plot that represents the evolution of execution time ratio with the matrix size in function of the number of threads, for the *Mult* algorithm .

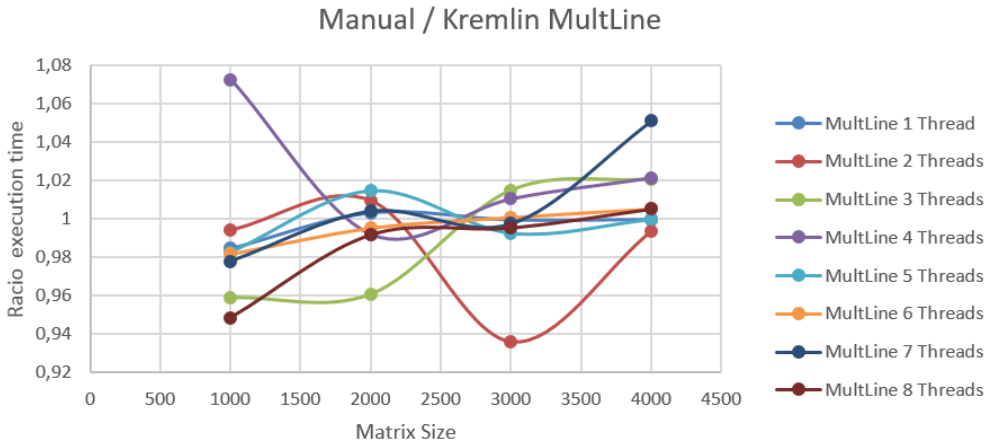

Figure 2.12: Dispersion plot that represents the evolution of execution time ratio with the matrix size in function of the number of threads, for the *MultLine* algorithm.

Confronting Figures 2.11 and 2.12, they seemed confusing, messed up, random, and that is partial true because the exceution time values from both groups are that close form one another that a slight alteration makes, apparently, an huge impact. It is also partial false because there are patterns in these plots: for each matrix size there is a concentration of lines, meaning that it is not so random and the explanation is that both parallelizations have similar performances. However, and again, the interval value is small.

Trying to se the data in a different angle and perspective to understand if there is any correlation, pattern or relation, these Figures, 2.13 and 2.14, are an unsuccessful attempt of it. The point of these two figures is to understand the variation of execution time ratio with number of threads in function of matrix size, however there is no relation for both cases. So, the number of threads has no direct relation with the execution time ratio in function of the matrix size.

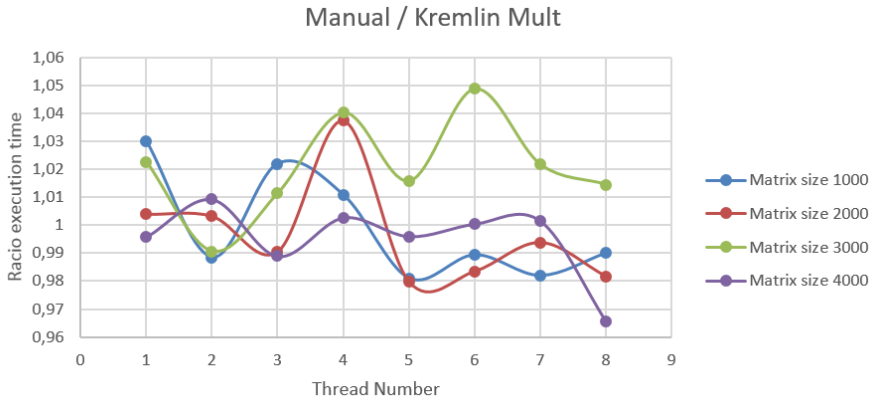

Figure 2.13: Dispersion plot that represents the evolution of execution time ratio with the number of threads in function of matrix size, for the *Mult* algorithm.

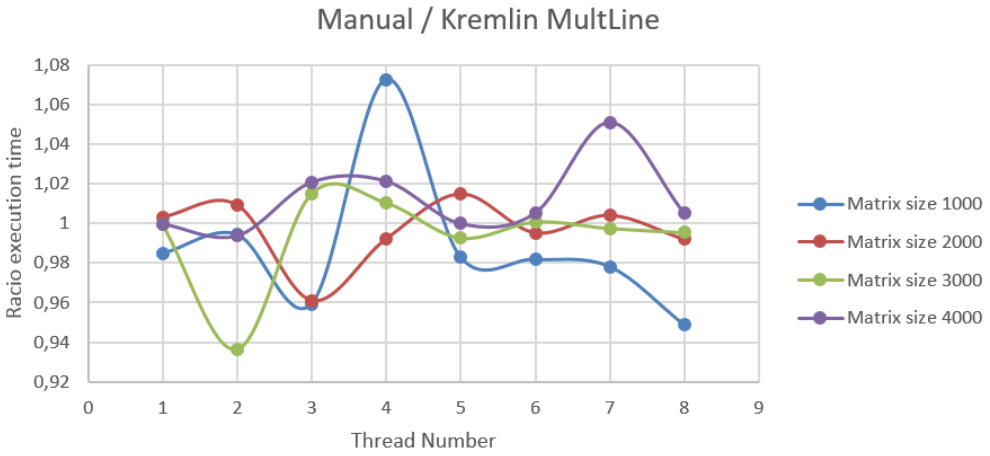

Figure 2.14: Dispersion plot that represents the evolution of execution time ratio with the number of threads in function of matrix size, for the *MultLine* algorithm.

So far, in this sub sub section, the analysis done is a general plane. this Figure 2.15 is about execution time ratio in function of matrix size using eight threads. This specific case was chosen because for both groups, using 8 threads had the best results, as previously mentioned and verified. In the green line is established as a reference: the values under this line mean Kremlin performed better than Manual, and the values above this line mean the opposite. Looking at those values, there is more values under the line than above, meaning that Kremlin group performed better, for both implementations.

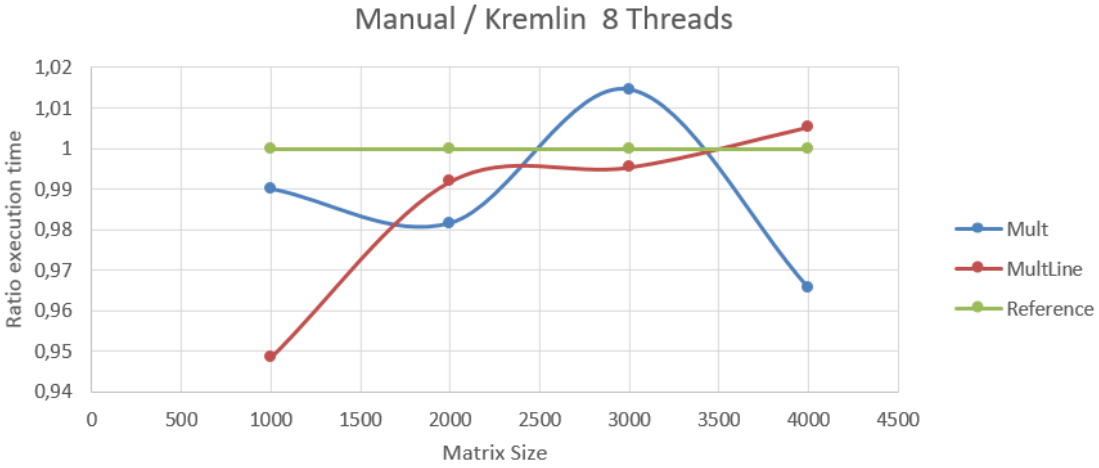

Figure 2.15: Dispersion plot that represents the evolution of execution time ratio with the matrix size in for 8 threads being used, using *Mult* and *MultLine* algorithms.

Chapter 3

Appendices

3.1 Developed code

3.1.1 Original Matrix Multiplication (Mult)

Listing 3.1: Matrix Multiplication original algorithm, written in C++

```
double OnMult(int m_ar, int m_br)
{
    double Time1, Time2;
    double temp;
    int i, j, k;
    double *pha, *phb, *phc;

    //Matrixes Memory allocation
    pha = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phb = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phc = (double *)malloc((m_ar * m_ar) * sizeof(double));

    //Starting counting time
    Time1 = omp_get_wtime();

    //Loading matrix values
    for(i=0; i<m_ar; i++)
        for(j=0; j<m_ar; j++)
            pha[i*m_ar + j] = (double)1.0;

    for(i=0; i<m_br; i++)
        for(j=0; j<m_br; j++)
            phb[i*m_br + j] = (double)(i+1);


    //Matrix Multiplication
    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;
        }
    }

    //Stoping time
    Time2 = omp_get_wtime();

    //Freeing memory used for matrixes
    free(pha);
    free(phb);
    free(phc);

    return Time2 - Time1;
}
```

3.1.2 Original Matrix Multiplication By line (MutLine)

Listing 3.2: Matrix Multiplication by line original algorithm, written in C++

```
double OnMultLine(int m_ar, int m_br)
{
    double Time1, Time2;
    double temp;
    int i, j, k;
    double *pha, *phb, *phc;

    //Matrixes Memory allocation
    pha = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phb = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phc = (double *)malloc((m_ar * m_ar) * sizeof(double));

    //Starting counting time
    Time1 = omp_get_wtime();

    //Loading matrix values
    for(i=0; i<m_ar; i++)
        for(j=0; j<m_ar; j++)
            pha[i*m_ar + j] = (double)1.0;

    for(i=0; i<m_br; i++)
        for(j=0; j<m_br; j++)
            phb[i*m_br + j] = (double)(i+1);

    for(i=0; i<m_ar; i++)
        for(j=0; j<m_ar; j++)
            phc[i*m_ar + j] = (double)0.0;


    //Matrix Multiplication
    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];
            }

        }
    }

    //Stoping time
    Time2 = omp_get_wtime();

    //Freeing memory used for matrixes
    free(pha);
    free(phb);
    free(phc);

    return Time2 - Time1;;
}
```

3.1.3 Manual Matrix Multiplication (Mult)

Listing 3.3: Matrix Multiplication manually parallelised using OpenMP library, written in C++

```
double OnMultThreading(int m_ar, int m_br,int x)
{
    double Time1, Time2;
    double temp;
    int i, j, k;
    double *pha, *phb, *phc;

    //Matrixes Memory allocation
    pha = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phb = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phc = (double *)malloc((m_ar * m_ar) * sizeof(double));

    //Starting counting time
    Time1 = omp_get_wtime();

    //Loading matrix values
    for(i=0; i<m_ar; i++)
        for(j=0; j<m_ar; j++)
            pha[i*m_ar + j] = (double)1.0;

    for(i=0; i<m_br; i++)
        for(j=0; j<m_br; j++)
            phb[i*m_br + j] = (double)(i+1);


    //Matrix Multiplication
    for(i=0; i<m_ar; i++)
    {   for( j=0; j<m_br; j++)
        {   temp = 0;
            #pragma omp parallel for reduction(+:temp) num_threads (x)
            for( k=0; k<m_ar; k++)
            {
```

```
                temp += pha[i*m_ar+k] * phb[k*m_br+j];
            }
            phc[i*m_ar+j]=temp;
        }
    }

    //Stoping time
    Time2 = omp_get_wtime();

    //Freeing memory used for matrixes
    free(pha);
    free(phb);
    free(phc);

    return Time2 - Time1;
}
```

3.1.4 Manual Matrix Multiplication By line (MultLine)

Listing 3.4: Matrix Multiplication by line manually parallelised using OpenMP library, written in C++

```
double OnMultLineThreading(int m_ar, int m_br,int x)
{
    double Time1, Time2;
    int i, j, k;
    double *pha, *phb, *phc;

    //Matrixes Memory allocation
    pha = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phb = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phc = (double *)malloc((m_ar * m_ar) * sizeof(double));

    //Starting counting time
    Time1 = omp_get_wtime();

    //Loading matrix values
    for(i=0; i<m_ar; i++)
        for(j=0; j<m_ar; j++)
            pha[i*m_ar + j] = (double)1,0;

    for(i=0; i<m_br; i++)
        for(j=0; j<m_br; j++)
            phb[i*m_br + j] = (double)(i+1);

    for(i=0; i<m_ar; i++)
        for(j=0; j<m_ar; j++)
            phc[i*m_ar + j] = (double)0,0;


    //Matrix Multiplication
    for(i=0; i<m_ar; i++)
    {   for( k=0; k<m_ar; k++)
```

```
        {
            #pragma omp parallel for num_threads (x)
            for( j=0; j<m_br; j++)
            {
                phc[i*m_ar+j] += pha[i*m_ar+k] * phb[k*m_br+j];
            }

        }
    }

    //Stoping time
    Time2 = omp_get_wtime();

    //Freeing memory used for matrixes
    free(pha);
    free(phb);
    free(phc);

    return Time2 - Time1;
}
```

3.1.5 Kremlin Matrix Multiplication (Mult)

Listing 3.5: Matrix Multiplication with Kremlin's indications for parallelization, written in C++

```
double OnMultKremlin(int m_ar, int m_br,int x)
{
    double Time1, Time2;
    double temp;
    int i, j, k;
    double *pha, *phb, *phc;

    //Matrixes Memory allocation
    pha = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phb = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phc = (double *)malloc((m_ar * m_ar) * sizeof(double));

    //Starting counting time
    Time1 = omp_get_wtime();

    //Loading matrix values
    for(i=0; i<m_ar; i++)
        #pragma omp parallel for num_threads (x)
        for(j=0; j<m_ar; j++)
            pha[i*m_ar + j] = (double)1.0;

    for(i=0; i<m_br; i++)
        #pragma omp parallel for num_threads (x)
        for(j=0; j<m_br; j++)
            phb[i*m_br + j] = (double)(i+1);


    //Matrix Multiplication
    for(i=0; i<m_ar; i++)
```

```
    {   for( j=0; j<m_br; j++)
        {   temp = 0;
            #pragma omp parallel for reduction(+:temp) num_threads (x)
            for( k=0; k<m_ar; k++)
            {
                temp += pha[i*m_ar+k] * phb[k*m_br+j];
            }
            phc[i*m_ar+j]=temp;
        }
    }

    //Stoping time
    Time2 = omp_get_wtime();

    //Freeing memory used for matrixes
    free(pha);
    free(phb);
    free(phc);

    return Time2 - Time1;
}
```

3.1.6 Kremlin Matrix Multiplication By line (MultLine)

Listing 3.6: Matrix Multiplication by line with Kremlin's indications for parallelization, written in C++

```
double OnMultLineKremlin(int m_ar, int m_br,int x)
{
    double Time1, Time2;
    int i, j, k;
    double *pha, *phb, *phc;

    //Matrixes Memory allocation
    pha = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phb = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phc = (double *)malloc((m_ar * m_ar) * sizeof(double));

    //Starting counting time
    Time1 = omp_get_wtime();

    //Loading matrix values
    for(i=0; i<m_ar; i++)
        #pragma omp parallel for num_threads (x)
        for(j=0; j<m_ar; j++)
            pha[i*m_ar + j] = (double)1.0;

    for(i=0; i<m_br; i++)
        #pragma omp parallel for num_threads (x)
        for(j=0; j<m_br; j++)
            phb[i*m_br + j] = (double)(i+1);

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

```
        #pragma omp parallel for num_threads (x)
        for(j=0; j<m_ar; j++)
            phc[i*m_ar + j] = (double)0.0;

    //Matrix Multiplication
    for(i=0; i<m_ar; i++)
    {   for( k=0; k<m_ar; k++)
        {
            #pragma omp parallel for num_threads (x)
            for( j=0; j<m_br; j++)
            {
                phc[i*m_ar+j] += pha[i*m_ar+k] * phb[k*m_br+j];
            }

        }
    }

    //Stoping time
    Time2 = omp_get_wtime();

    //Freeing memory used for matrixes
    free(pha);
    free(phb);
    free(phc);

    return Time2 - Time1;
}
```

3.1.7 Sequential program compiled and profiled by Kremlin

Listing 3.7: Program compiled and profiled by Kremlin with both Mult and MultLine algorithms, written in C++

```
//#include <omp.h>
#include <stdio.h>
#include <iostream>
#include <iomanip>
#include <time.h>
#include <cstdlib>
//#include <papi.h>
#include <fstream>
#include <chrono>

using namespace std;

#define SYSTEMTIME clock_t
/*
double OnMultLineThreading(int m_ar, int m_br,int x)
{
    double Time1, Time2;

    char st[100];
    double temp;
```

```
    int i, j, k;

    double *pha, *phb, *phc;



    pha = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phb = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phc = (double *)malloc((m_ar * m_ar) * sizeof(double));

    for(i=0; i<m_ar; i++)
        for(j=0; j<m_ar; j++)
            pha[i*m_ar + j] = (double)1.0;



    for(i=0; i<m_br; i++)
        for(j=0; j<m_br; j++)
            phb[i*m_br + j] = (double)(i+1);

    for(i=0; i<m_ar; i++)
        for(j=0; j<m_ar; j++)
            phc[i*m_ar + j] = (double)0.0;



    Time1 = omp_get_wtime();

    for(i=0; i<m_ar; i++)
    {   for( k=0; k<m_ar; k++)
        {
            #pragma omp parallel for num_threads (x)
            for( j=0; j<m_br; j++)
            {
                phc[i*m_ar+j] += pha[i*m_ar+k] * phb[k*m_br+j];
            }

        }
    }


    Time2 = omp_get_wtime();
    //sprintf(st, "Time: %3.3f seconds\n", (double)(Time2 - Time1));
    //cout << st;

    /*cout << "Result matrix: " << endl;
    for(i=0; i<min(10,m_ar); i++)
    {   for(j=0; j<min(10,m_br); j++)
            cout << phc[j] << " ";
    }
    cout << endl;*/
/*
    free(pha);
    free(phb);
    free(phc);
    return (double)(Time2 - Time1);

```

```
}*/
/*
double OnMultThreading(int m_ar, int m_br,int x)
{

    double Time1, Time2;

    char st[100];
    double temp;
    int i, j, k;

    double *pha, *phb, *phc;



    pha = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phb = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phc = (double *)malloc((m_ar * m_ar) * sizeof(double));

    for(i=0; i<m_ar; i++)
        for(j=0; j<m_ar; j++)
            pha[i*m_ar + j] = (double)1.0;



    for(i=0; i<m_br; i++)
        for(j=0; j<m_br; j++)
            phb[i*m_br + j] = (double)(i+1);



    Time1 = omp_get_wtime();

    for(i=0; i<m_ar; i++)
    {   for( j=0; j<m_br; j++)
        {   temp = 0;
            #pragma omp parallel for reduction(+:temp) num_threads (x)
            for( k=0; k<m_ar; k++)
            {
                temp += pha[i*m_ar+k] * phb[k*m_br+j];
            }
            phc[i*m_ar+j]=temp;
        }
    }


    Time2 = omp_get_wtime();
    //sprintf(st, "Time: %3.3f seconds\n", (double)(Time2 - Time1));
    //cout << st;

    /*cout << "Result matrix: " << endl;
    for(i=0; i<1; i++)
    {   for(j=0; j<min(10,m_br); j++)
            cout << phc[j] << " ";
    }
    cout << endl;*/
/*
```

```
    free(pha);
    free(phb);
    free(phc);
    return (double)(Time2 - Time1);

}*/

double OnMult(int m_ar, int m_br)
{

    //double Time1, Time2;

    char st[100];
    double temp;
    int i, j, k;

    double *pha, *phb, *phc;



    pha = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phb = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phc = (double *)malloc((m_ar * m_ar) * sizeof(double));

    for(i=0; i<m_ar; i++)
        for(j=0; j<m_ar; j++)
            pha[i*m_ar + j] = (double)1.0;



    for(i=0; i<m_br; i++)
        for(j=0; j<m_br; j++)
            phb[i*m_br + j] = (double)(i+1);



    //Time1 = omp_get_wtime();
    auto Time1 = std::chrono::high_resolution_clock::now();

    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;
        }
    }


    //Time2 = omp_get_wtime();
    auto Time2 = std::chrono::high_resolution_clock::now();
    //sprintf(st, "Time: %3.3f seconds\n", (double)(Time2 - Time1));
    //cout << st;

    /*cout << "Result matrix: " << endl;
```

```
    for(i=0; i<1; i++)
    {   for(j=0; j<min(10,m_br); j++)
            cout << phc[j] << " ";
    }
    cout << endl;*/

    free(pha);
    free(phb);
    free(phc);
    auto time = Time2-Time1;
    return (double) std::chrono::duration_cast<std::chrono::milliseconds>(time).count();

}


double OnMultLine(int m_ar, int m_br)
{
    //double Time1, Time2;

    char st[100];
    double temp;
    int i, j, k;

    double *pha, *phb, *phc;



    pha = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phb = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phc = (double *)malloc((m_ar * m_ar) * sizeof(double));

    for(i=0; i<m_ar; i++)
        for(j=0; j<m_ar; j++)
            pha[i*m_ar + j] = (double)1.0;



    for(i=0; i<m_br; i++)
        for(j=0; j<m_br; j++)
            phb[i*m_br + j] = (double)(i+1);

    for(i=0; i<m_ar; i++)
        for(j=0; j<m_ar; j++)
            phc[i*m_ar + j] = (double)0.0;



    //Time1 = omp_get_wtime();
    auto Time1 = std::chrono::high_resolution_clock::now();

    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];
            }
```

```
249
250             }                                                                                   2
251         }
252                                                                                                 4
253
254         //Time2 = omp_get_wtime();                                                              6
255         auto Time2 = std::chrono::high_resolution_clock::now();
256         //sprintf(st, "Time: %3.3f seconds\n", (double)(Time2 - Time1));                        8
257         //cout << st;
258                                                                                                 10
259         /*cout << "Result matrix: " << endl;
260         for(i=0; i<min(10,m_ar); i++)                                                           12
261         {   for(j=0; j<min(10,m_br); j++)
262                 cout << phc[j] << " ";                                                          14
263         }
264         cout << endl;*/                                                                         16
265
266         free(pha);                                                                              18
267         free(phb);
268         free(phc);                                                                              20
269         auto time = Time2-Time1;
270         return (double) std::chrono::duration_cast<std::chrono::milliseconds>(time).count();    22
271
272 }                                                                                               24
273
274 void OutputToFile(int lin, int col, int inc, int limit/*, char* filename*/)                     26
275 {
276     int i;                                                                                      28
277     double temp;
278     ofstream myfile;                                                                            30
279     myfile.open (/*filename*/"matrixMultResult.csv");
280     myfile << "i,Algoritmo a,Algoritmo b\n";                                                    32
281     for(i=lin;i <= limit; i=i+inc)
282     {                                                                                           34
283         temp=OnMult(i,i);
284         myfile << i << "," << temp << ",";                                                      36
285         temp=OnMultLine(i,i);
286         myfile << temp << "\n";                                                                 38
287     }
288     myfile.close();                                                                             40
289 }
290                                                                                                 42
291
292                                                                                                 44
293
294 float produtoInterno(float *v1, float *v2, int col)                                             46
295 {
296     int i;                                                                                      48
297     float soma=0.0;
298                                                                                                 50
299     for(i=0; i<col; i++)
300         soma += v1[i]*v2[i];                                                                    52
301
302     return(soma);                                                                               54
303
304 }                                                                                               56
305 /*)
```

```
306 void handle_error (int retval)
307 {
308   printf("PAPI error %d: %s\n", retval, PAPI_strerror(retval));
309   exit(1);
310 }
311
312 void init_papi() {
313   int retval = PAPI_library_init(PAPI_VER_CURRENT);
314   if (retval != PAPI_VER_CURRENT && retval < 0) {
315     printf("PAPI library version mismatch!\n");
316     exit(1);
317   }
318   if (retval < 0) handle_error(retval);
319
320   std::cout << "PAPI Version Number: MAJOR: " << PAPI_VERSION_MAJOR(retval)
321             << " MINOR: " << PAPI_VERSION_MINOR(retval)
322             << " REVISION: " << PAPI_VERSION_REVISION(retval) << "\n";
323 }*/
324
325
326 int main (int argc, char *argv[])
327 {
328
329     char c;
330     int lin, col, nt=1, inc, limit,x;
331     int op;
332     char* filename;
333     //int EventSet = PAPI_NULL;
334     long long values[2];
335     int ret;
336     /*
337     cout << "Numero de processadores: " << omp_get_num_procs() << endl;
338     ret = PAPI_library_init( PAPI_VER_CURRENT );
339     if ( ret != PAPI_VER_CURRENT )
340         std::cout << "FAIL" << endl;
341
342
343     ret = PAPI_create_eventset(&EventSet);
344         if (ret != PAPI_OK) cout << "ERRO: create eventset" << endl;
345
346
347     ret = PAPI_add_event(EventSet,PAPI_L1_DCM);
348     if (ret != PAPI_OK) cout << "ERRO: PAPI_L1_DCM" << endl;
349
350
351     ret = PAPI_add_event(EventSet,PAPI_L2_DCM);
352     if (ret != PAPI_OK) cout << "ERRO: PAPI_L2_DCM" << endl;
353
354     */
355
356
357     op=1;
358     do {
359         cout << endl;
360         cout << "1. Multiplication" << endl;
361         cout << "2. Line Multiplication" << endl;
362         cout << "3. outputToFile" << endl;
```

```
        cout << "4. multithreading on Multiplication" << endl;
        cout << "5. multithreading on LineMultiplication" << endl;
        cout << "Selection?: ";

        cin >>op;
        if (op == 0)
            break;

        printf("Dimensions: lins cols ? ");
        cin >> lin >> col;


        if(op == 3)
        {
            printf("Dimensional increment: inc ? ");
            cin >> inc;
            printf("Limit: limit ? ");
            cin >> limit;
            /*printf("outputFile: filename.csv (must be an existing file) ? ");
            cin >> filename;*/
        }
        /*if(op == 4 || op == 5)
        {
            printf("Number of threads: x");
            cin >> x;
        }*/


        // Start counting
        /*ret = PAPI_start(EventSet);
        if (ret != PAPI_OK) cout << "ERRO: Start PAPI" << endl;
*/
        switch (op){
            case 1:
                cout << OnMult(lin, col)<< endl;
                break;
            case 2:
                cout << OnMultLine(lin, col)<<endl;
                break;
            case 3:
                OutputToFile(lin, col, inc, limit/*, filename*/);
                break;
            /*case 4:
                cout << OnMultThreading(lin, col,x)<< endl;
                break;
            case 5:
                cout << OnMultLineThreading(lin, col,x)<< endl;
                break;*/
        }

        /*ret = PAPI_stop(EventSet, values);
        if (ret != PAPI_OK) cout << "ERRO: Stop PAPI" << endl;
        printf("L1 DCM: %lld \n",values[0]);
        printf("L2 DCM: %lld \n",values[1]);
        ret = PAPI_reset( EventSet );
        if ( ret != PAPI_OK )
            std::cout << "FAIL reset" << endl; */
```

```
420
421
422
423     }while (op != 0);
424 /*
425         ret = PAPI_remove_event( EventSet, PAPI_L1_DCM );
426         if ( ret != PAPI_OK )
427             std::cout << "FAIL remove event" << endl;
428
429         ret = PAPI_remove_event( EventSet, PAPI_L2_DCM );
430
431
432         ret = PAPI_destroy_eventset( &EventSet );
433         if ( ret != PAPI_OK )
434             std::cout << "FAIL destroy" << endl;
435         */
436
437 }
```

3.2 Kremlin's Reports

3.2.1 Kremlin report for Matrix Multiplication, Mult version

Listing 3.8: Kremlin's indication of the blocks that should be parallelised and theorical variables that where calculated for Mult algorithm version

```
1 [ 0] TimeRed(4)=66.38%, TimeRed(Ideal)=70.96%, Cov=88.51%, SelfP=5.05, DOALL
2   LOOP matrixmul.cpp [ 148 -  181]:      OnMult
3   FUNC matrixmul.cpp [ 142 -  142]:      OnMult called at file matrixmul.cpp, line  397
4
5 [ 1] TimeRed(4)=3.10%, TimeRed(Ideal)=3.37%, Cov=4.13%, SelfP=5.44, DOALL
6   LOOP matrixmul.cpp [ 149 -  167]:      OnMult
7   FUNC matrixmul.cpp [ 142 -  142]:      OnMult called at file matrixmul.cpp, line  397
8
9 [ 2] TimeRed(4)=3.10%, TimeRed(Ideal)=3.37%, Cov=4.13%, SelfP=5.44, DOALL
10  LOOP matrixmul.cpp [ 149 -  161]:      OnMult
11  FUNC matrixmul.cpp [ 142 -  142]:      OnMult called at file matrixmul.cpp, line  397
```

3.2.2 Kremlin report for Matrix Multiplication, MultLine version

Listing 3.9: Kremlin's indication of the blocks that should be parallelised and theorical variables that where calculated for MultLine algorithm version

```
1 ..................................
2 [ 0] TimeRed(4)=63.01%, TimeRed(Ideal)=63.20%, Cov=84.02%, SelfP=4.03, DOALL
3   LOOP matrixmul.cpp [ 213 -  247]:  OnMultLine
4   FUNC matrixmul.cpp [ 207 -  207]:  OnMultLine called at file matrixmul.cpp, line  400
5
6 [ 1] TimeRed(4)=2.86%, TimeRed(Ideal)=2.96%, Cov=3.81%, SelfP=4.45, DOALL
```

```
  LOOP matrixmul.cpp [ 213 -  235]:  OnMultLine
  FUNC matrixmul.cpp [ 207 -  207]:  OnMultLine called at file matrixmul.cpp, line  400

[ 2] TimeRed(4)=2.86%, TimeRed(Ideal)=2.96%, Cov=3.81%, SelfP=4.45, DOALL
  LOOP matrixmul.cpp [ 213 -  231]:  OnMultLine
  FUNC matrixmul.cpp [ 207 -  207]:  OnMultLine called at file matrixmul.cpp, line  400

[ 3] TimeRed(4)=2.86%, TimeRed(Ideal)=2.96%, Cov=3.81%, SelfP=4.45, DOALL
  LOOP matrixmul.cpp [ 213 -  225]:  OnMultLine
  FUNC matrixmul.cpp [ 207 -  207]:  OnMultLine called at file matrixmul.cpp, line  400
```