Cloth Simulation Report

Yunzhong Zhang

Contents

1 Introduction

In the report we would introduce our work on improvement of cloth simulation problem. Firstly, we modify on the velocity when the collision happened to ensure the correct result. Then, We optimize the codes using the **Newton's Third Law Of Motion** and some other basic skills. Secondly, we use AVX and OpenMP to vectorize the optimized code respectively. Thirdly, based upon the AVX version code, we first modify the code structure of it to **avoid dependency** which is introduced in previous step, where it is noticed that the modification takes no extra computation load but brings some **cache unfriendly problem**, then OpenMP is applied do multi thread calculation and we do some analysis on it including roofline analysis. Lastly we implement the block-wise OMP version to **solve the cache unfriendly problem** and make relavent analysis. In each part we would present the performance comparison to see the influence of each modification and in the end of report we would show the performance of all methods. To help the readers better understand the report, we list the version of code here:

- Main version: the version that correctly solve the problem without any optimization, which is the benchmark of this report.
- OPT version: the version based upon Main version, applied some optimized skills like Newton Third law of Motion ,except parallel or vectorization.
- AVX version: the version based upon OPT version, is applied AVX vectorization.
- Open-SIMD version: the version based upon OPT version, is applied SIMD(vectorization) provided by Open MP.
- p-thread-OMP/Round-robin OMP: Based upon AVX version, do some modification on the structure(to avoid dependency) , which is called 1-thread-OMP. and then apply multiple threads on it. Notice 1-thread-OMP is a special case of p-thread-OMP when only master thread is used during all stages.
- Block-Wise-OMP: Based upon AVX version, do some midification on the structure(to avoid dependency) and then apply multiple threads on it.

2 Solve the collision problem and optimize the initial code

In the section, we would show how we change the velocity when collision happened and how we optimize the initial given code. As can be shown in 1, to get the velocity that tangent to the ball, according to vector operations, we can use the formula

$$v_t = v - v_n \tag{1}$$

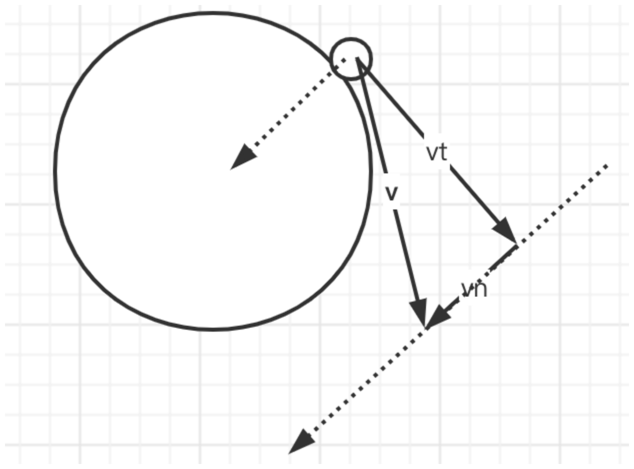


Figure 1: velocity change when collision

By using the position of two coordinate, we can obtain the direction of the node to the central of ball v_{pos} , and apply projection of v on this direction is v_t. Then the formula to compute the new velocity v_t is

$$v_t = v - \frac{v * v_{pos}}{||v_{pos}||} * \frac{v_{pos}}{||v_{pos}||} \tag{2}$$

where damping factor is not shown.

As for optimizing the initial code to achieve a better performance with using one kernel and without any manual vectorization, we do the following modification

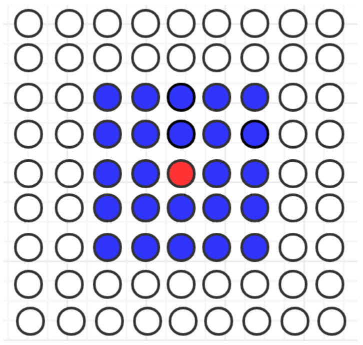
- During update force and potential energy, instead of using the method of given code , we apply Newton's Third Law Of Motion.
- Turns the nested loop into combined single loop
- Combined the loops together to avoid unnecessary iterating change of index and makes cache read and write more friendly.
- Reduce some duplicated computation like some shared-used constant.
- Eliminate the if-statement branch in the eval_fun(...).
- Add restrict qualifier for parameters to hint compiler

As the most computational intensive part, the computation of force and potential energy occupy most of the execution time and the modification on it contribute mostly. Applying Newton's Third Law Of Motion theoratically decrease half of the computation. More exactly, given a node, its neighboring nodes would interact with it and produce a force to it. In the meantime, the given node would produce an equal-magnitude and anti-direction force to the adjacent node, which makes possible for us to update the interaction force between two nodes for the two nodes in the same time. As can be shown from Figure2, given the node(red), we need to calculate the interaction force with the adjacent nodes(blue nodes in Figure 2(a)) in the original given code, while after applying Newton's Third Law of Motion, we only have to compute the force and potential energy on the right hand side and bottom of the given node Figure 2(b), where it is intuitive to only half of the computation required now.

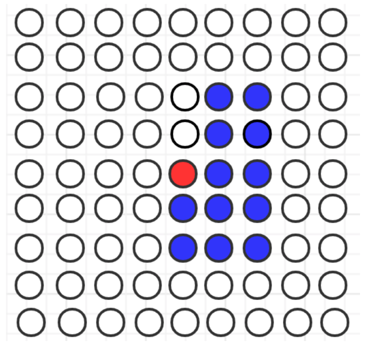
The improvement come along with disadvantage. The method also introduce the dependency, which makes it tricky to use OpenMP to parallelize the programme efficiently. However, we have overcame this by eliminating the dependency with the cost of cache-unfriendly, which we would show in section 4.1. And the cache-unfriendly problem introduced by the proposed method would be solved by the method proposed in 4.2. We do experimental comparison on the original version and optimized version. As for the parameter setting, it is obvious that the problem scale is decided by the number of nodes(n), the node interaction level(δ) and the iteration. We would fixed the iteration to 10 and any other parameters that no related to problem scale remain the same as lecturer and tutor's setting. As for the n and δ, we would fix one of them and vary another, to see what difference would be produced for each implement method. As can be seen from Figure 3, the optimized version has both advantage on the execution time and the number of float points than the original version, and this gap become larger as the number of nodes increase. In the same time, we notice that both of the execution time and the number of float points of the optimized version are almost half of the original version's, which is consistent with our prediction. As for different δ shown in Figure4, it behaves similar and more distinguished.

3 AVX and OpenMP SIMD Vectorization

Following we would show our work on vectorizing the optimized version code, using two different ways, AVX and SIMD provided by OpenMP. In AVX version, we choose to use $\_\_m256d$, which implies the compiler could operates 4 doubles simultaneously with the same instruction. Although there is a larger register version ($\_\_m512d$ version) we can choose, we still use $\_\_m256d$ by the reason that in the eval_fun(..) we convert the neighboring nodes into avx data type, in the way that it may left some nodes alone and be operated sequentially. As we could assume $1 < \delta < 10$, if we use $\_\_m512d$ version, there would be little situation where δ should be large enough that fewer data are left. As shown in Figure5,

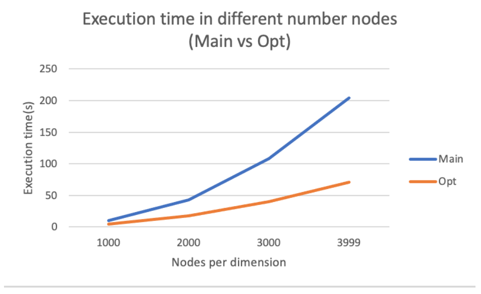


(a)

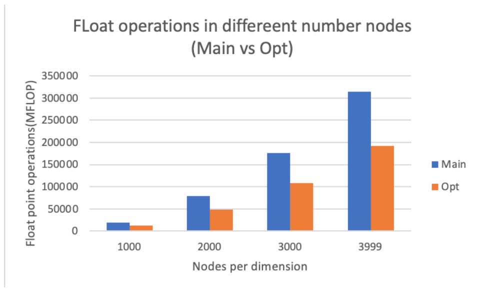


(b)

Figure 2: Difference of computing force and potential energy. 2(a) Before applying Newton' s Third Law of Motion; 2(b) After applying Newton' s Third Law of Motion

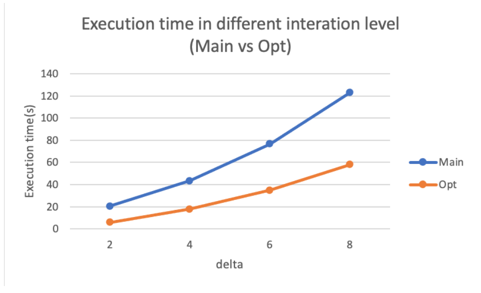


(a)

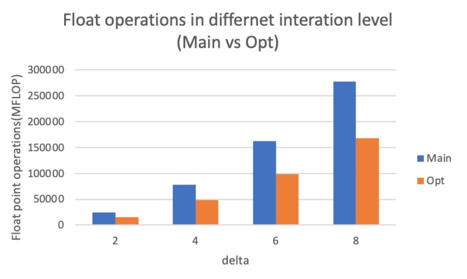


(b)

Figure 3: Execution Time & Float point operations with different number of nodes(Main vs Opt). 3(a) Execution time with different number of nodes; 3(b) Float point operations with different number of nodes



(a)



(b)

Figure 4: Execution Time & Float point operations with different interaction level(Main vs Opt). 3(a) Execution time with different interaction level; 3(b) Float point operations with different interaction level

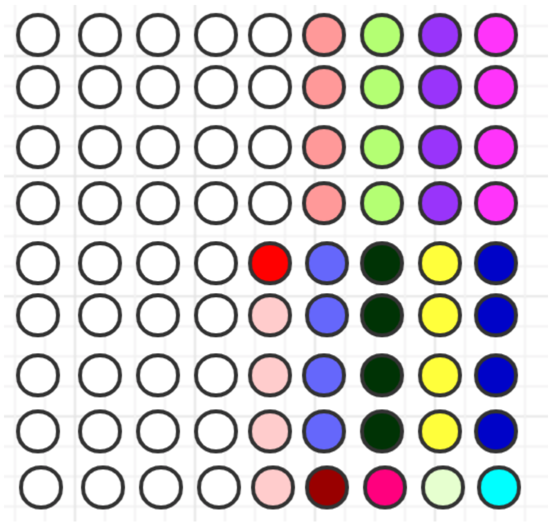
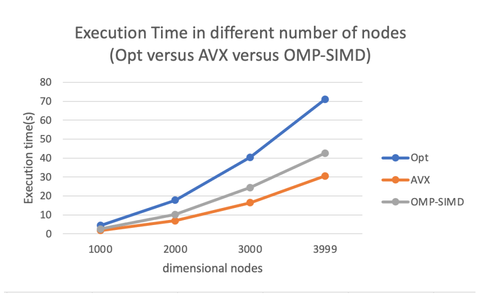
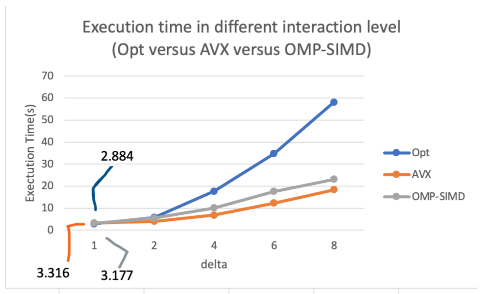


Figure 5: avx-256d-datatype reading neighboring nodes



(a)

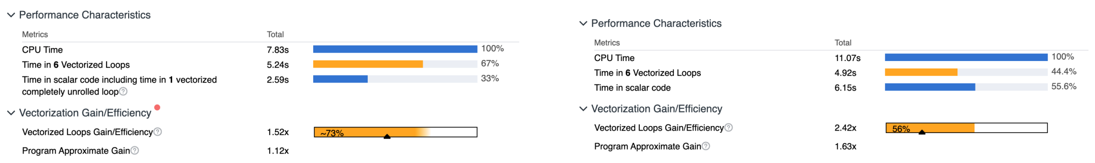


(b)

Figure 6: Execution Time with different number of nodes(Main vs Opt) or interaction level. 6(a) Execution time with different number of nodes; 6(b) Execution time with different interaction level

the nodes in the same color are the nodes that processed in the same batch. We notice when δ if we choose 4-double/batch, only 4 nodes are left. It is no doubt that we could adjust the version according to the δ or the pieces of code. However, to keep the codes simple, consistency and readability, we only use $\_\_m256d$ version. Due to the reason above, It is expected that there is a big influence of δ value on the performance and we would show the experimental later. It is noticed that in the AVX version code, we use the combination of $\_mm256\_cmp\_pd$ and $_m m256\_cmp\_pd$ to eliminate the if-statement lied in loopcode(..), while the branch in the evalperm(..) has already eliminated by our layout of optimized code. As shown in Figure 6(a), both the AVX version and OMP-SIMD gives a speed up based up on optimized version. It is noticed that here we set $\delta = 4$, which is SIMD-friendly for our code. In the meantime, the discussion on the performance with different δ is rather interesting. It can be seen from 6(b) that in some interaction level($\delta = 2, 4, 6, 8$), AVX demonstrates more outstanding performance compared to the other two version. However, in some interaction level($\delta = 1$), we notice that AVX produce a poor performance than both Opt and OMP-SIMD does, if zooming in. We demonstrate this phenomenon by the reason that if the left coloum is not enough to package 4 doubles into a batch($\_\_m256d$), then they would be computed sequentially. It can be imaged that when $\delta = 1$, there is no data could be packaged together(About this fact you can refer to Figure2(b) and the code cloth_codesse.c++). That is the reason why it cannot win its regeneration version-Opt version at $\delta = 1$. Also we make a reasonable guess that OpenMP had done a job that detect how much data should be packaged together on its SIMD function so that it can adjust any value of δ. But generality come along with particularity and that is why it is not performed as good as self-implemented AVX version.

Also we present the report generated by intel-advisor here. The parameters setting here is $\delta = 4$ and



Performance Characteristics

Metrics	Total	
CPU Time	7.83s	100%
Time in **6** Vectorized Loops	5.24s	67%
Time in scalar code including time in **1** vectorized completely unrolled loop	2.59s	33%

Vectorization Gain/Efficiency

Vectorized Loops Gain/Efficiency	1.52x	~73%
Program Approximate Gain	1.12x	

(a)

Performance Characteristics

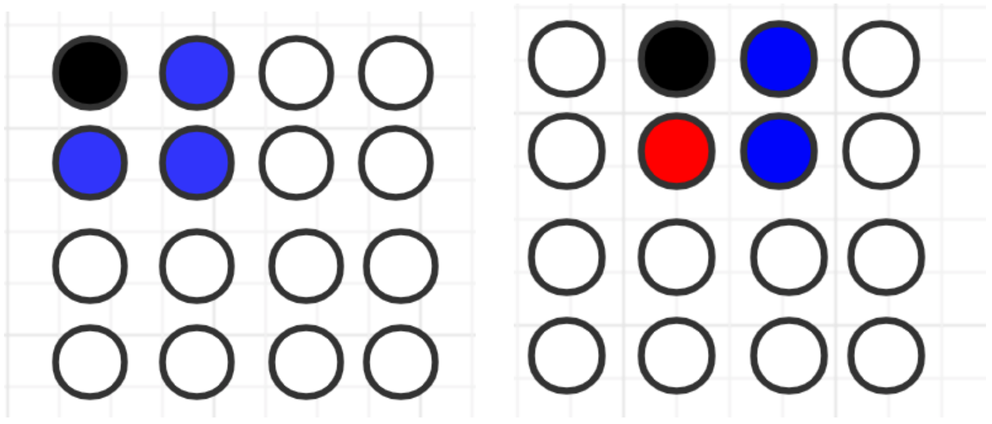
Metrics	Total	
CPU Time	11.07s	100%
Time in **6** Vectorized Loops	4.92s	44.4%
Time in scalar code	6.15s	55.6%

Vectorization Gain/Efficiency

Vectorized Loops Gain/Efficiency	2.42x	56%
Program Approximate Gain	1.63x	

(b)

Figure 7: Vectorization analysis 7(a) AVX; 7(b) OMP-SIMD



(a) (b)

Figure 8: The dependency between two steps: : the black node is given node and blue nodes are its interaction nodes, while the red node is the node in dependency. 9(a) the node and its interaction nodes; 9(b) the nodes and its interaction nodes and the dependent nodes.

$n = 2000$. As can be seen from Figure 7, both AVX version and OMP-SIMD version gain a efficiency compared to the scale versions after applying vectorization. Also, we notice that under such parameter setting, we find that the averaged speed-up of vectorization version of AVX is more distinguished than OMP-SIMD.

4 OpenMP Parallelization: Round-robin and Block-wise(6464 part)

4.1 Round-robin OMP

As we have mentioned previously, while using Newton's third law of motion would decrease half of the computation, it always brings the denpendency shown in 8 problem, which makes it tricky to use OpenMP to parallelize the programme. Instead of using some clauses or directives like critical or atomic which needs execute the instructions sequentially, we use a way to eliminate the dependency the codes. More specifically, shown in Figure9 instead of moving one node by one node, we skip $2 * \delta$ nodes that cause dependency and in the following loop we kick off from the skipped nodes where during this loop part of the skipped nodes would be computed. Again and again, all the nodes would be covered to compute. It is noticed that this method almostly increase no computational amount compared to the original one, while at the cost of memory assess which will be discussed later.

4.1.1 Performance of OMP versus AVX

It is shown in Figure 10 that 24-thread-OMP version produce distinguished performance compared to its 1-thread-OMP version. In the same time, it is quite dramatic that the 1-kernel-OMP version who effectively eliminate the dependency without introducing any additional computation amount, is slower than its father version-AVX. We demonstrate this phenomenon by the reason that cache unfriendly.

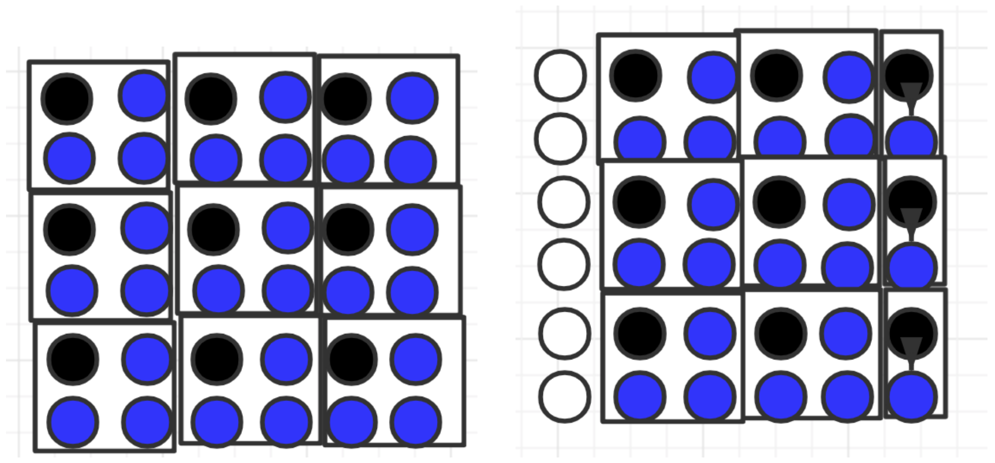


Figure 9: A method to eliminate the dependency 9(a) First loop ; 9(b) Second loop

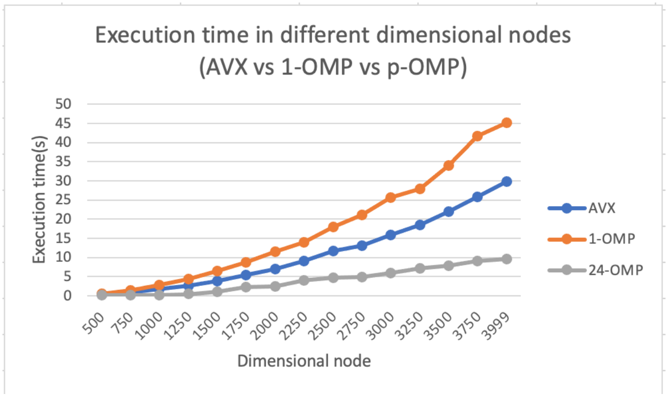
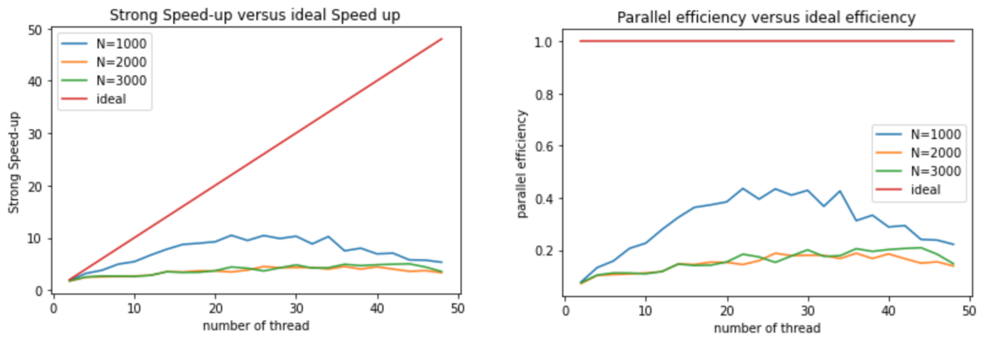


Figure 10: Execution time under different dimensional nodes (SSE vs single-thread-OMP vs p-thread-OMP)

Indeed, if we have a view on the Opt/AVX version code, we would discover that it demonstrates high locality, both in space and time. Temporarily skipping δ nodes makes the it difficult for fully utilizing the ability of cache. More specifically, we assume that there is 2500 nodes in one dimensional, and the interaction level is 4. Noticed that in the CPU of Gadi server, there are 512 double cache line(L1 double cache size / L1 cache line size = 32*1024/64), which implies ideally if CPU read/write a given node(red node in Figure 5)'s arrays, then CPU drag the cache lines that node loacated and also needs to drag the cache lines of its interaction nodes(other colorful nodes). In the general situation, it gives almost 5 cache miss, much worse than than OPT/AVX version which both intensively read/write that may lead to almost no cache miss. Also, after we skip the *delta* nodes once by once, and after one round when we finally back the nodes near to it, the cache line contains it may highly probably be replaced. (Since we have only 512 cache line and above discussion even just talk one array).

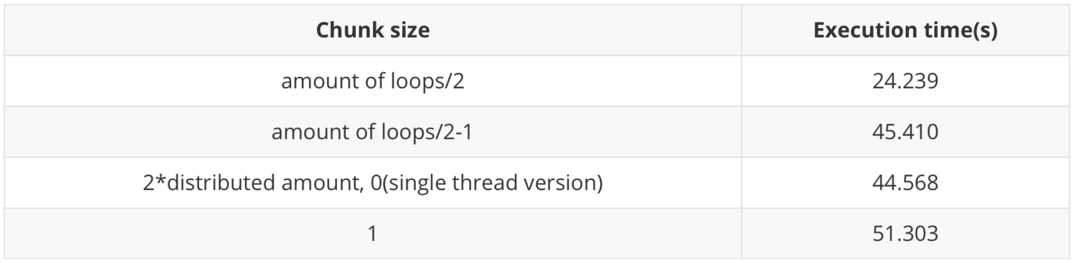
4.1.2 Speed-up and efficiency

Also we present the speed-up and efficiency in 11, where we can see that under the low amount of nodes, OMP's speed up and efficiency is more satisfactory, that is, more near to ideal speed up and efficiency. Combined with the analysis above and the results shown in Figure 10 who has little gap under low amount of nodes, it is reasonable to make guess that memory access makes more dimensional node leads to more obvious gap between AVX and its eliminating-dependency version: p-OMP version. Later we would discuss our ideas to relieve this shortcoming.



(a) (b)

Figure 11: Speed-up and efficiency of OMP in different number of nodes 11(a) Speed-up of OMP in different number of nodes; 11(b) Efficiency of OMP in different number of nodes



Chunk size	Execution time(s)
amount of loops/2	24.239
amount of loops/2-1	45.410
2*distributed amount, 0(single thread version)	44.568
1	51.303

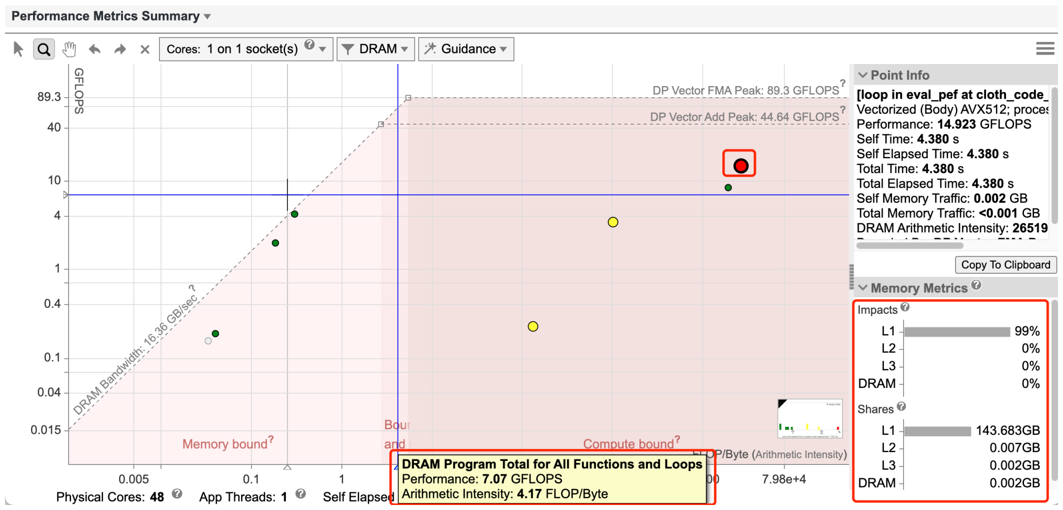
Figure 12: The execution time of with setting different for chunk size under two threads (n=3999, d=4, p=2(last row except), i=10)

4.1.3 Chunk Size Problem

Speaking of schedule chunk size, as for this problem, from general perspective, it seems that balanced distribution is a good choice; on the angle, the problem is spacity intensive, which guide us to hope one thread to finish the whole line of the nodes. Here we gives two interesting experiment, compared to the default one: equally, round robin distributed among the threads. It is recall that after we do the modification in order to prevent dependency, our step has became $2*\delta$. This implies the parallelization sections totally contains $\frac{n^2}{(2*threads+1)^2}$ and further each thread are assigned $\frac{n^2}{(2*threads+1)^2*p}$ under the equally distribution. Let us consider an polar and representative situation, if we have 2 threads, and set the chunk size to be $\frac{n^2}{(2*threads+1)^2*p} - 1$, that is, if there is 1000 iterations, one thread is assigned 999 while the other only get one, which cause imbalanced load and the thread get little loop would finish very early and then idle. As can be seen from the Table 12, in first row, we notice that the chunk size that distribute in balance gives the best performance among these. The second row sets the chunk size to be total amount of loops-1, which implies one thread would get one and one thread would get all the remaining loops. And the results in this situation is rather poor, which is about the same with using the single thread version(third row). In the same time, we also set the chunk size to be 1, which implies the each thread get 1 loops in turn, and it even demonstrates poor performance than the single thread version. We demonstrate this phenomenon by the reason that it may loss some locality and cache is not as friendly as before.

4.1.4 Roofline analysis

Here we would first compare the difference of roofline between AVX and 1-thread-OMP to see the operation to avoid dependency gives how much change on the computation and memory. And we put the rooflines of the p-thread-OMP and block-wise OMP together in section 4.2 to do further analysis. Notice the key parameter we use is $n = 2500, d = 4, i = 10$ Focusing on the computational intensity part in Figure 13 and 14, denoted by red dot, we notice AVX version is located under the computational bound area, while most of the snippet of the 1-thread-OMP version is lied on the DRAM roof, which


Performance Metrics Summary
Cores: 1 on 1 socket(s)
DRAM
Guidance
GFLOPS
89.3
40
10
4
1
0.4
0.1
0.04
0.015
DP Vector FMA Peak: 89.3 GFLOPS
DP Vector Add Peak: 44.64 GFLOPS
DRAM Bandwidth: 16.36 GB/sec
Memory bound
Compute bound
FLOP/Byte (Arithmetic Intensity)
0.005
0.1
1
7.98e+4
Copy To Clipboard

Point Info
[loop in eval_pef at cloth_code_
Vectorized (Body) AVX512; proces
Performance: **14.923** GFLOPS
Self Time: **4.380** s
Self Elapsed Time: **4.380** s
Total Time: **4.380** s
Total Elapsed Time: **4.380** s
Self Memory Traffic: **0.002** GB
Total Memory Traffic: **<0.001** GB
DRAM Arithmetic Intensity: **26519**

Memory Metrics

Impacts

L1	99%
L2	0%
L3	0%
DRAM	0%

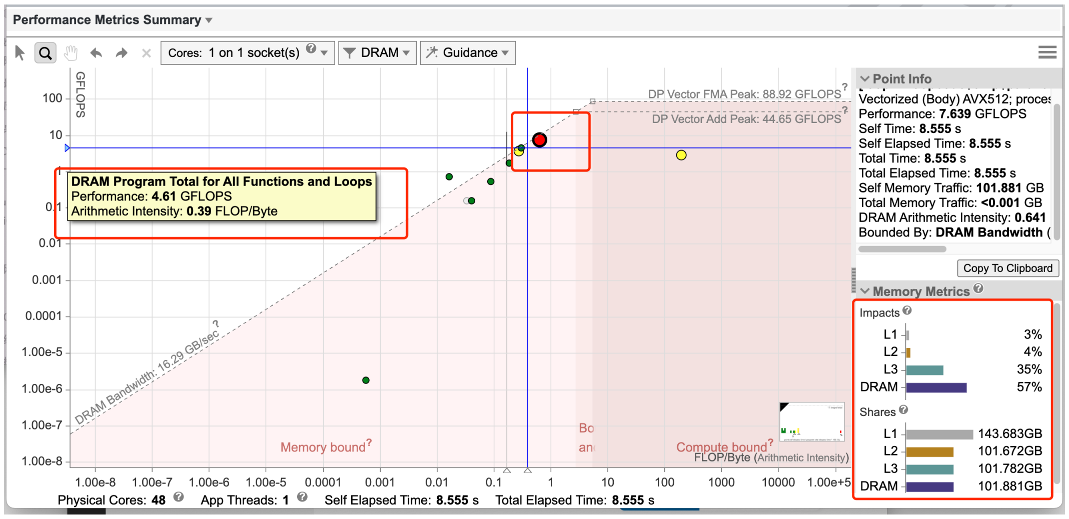
Shares

L1	143.683GB
L2	0.007GB
L3	0.002GB
DRAM	0.002GB

DRAM Program Total for All Functions and Loops
Performance: **7.07** GFLOPS
Arithmetic Intensity: **4.17** FLOP/Byte

Physical Cores: **48**   App Threads: **1**   Self Elapsed

Figure 13: Roofline of AVX version



Point Info
Vectorized (Body) AVX512; proces
Performance: **7.639** GFLOPS
Self Time: **8.555** s
Self Elapsed Time: **8.555** s
Total Time: **8.555** s
Total Elapsed Time: **8.555** s
Self Memory Traffic: **101.881** GB
Total Memory Traffic: **<0.001** GB
DRAM Arithmetic Intensity: **0.641**
Bounded By: **DRAM Bandwidth** (

Memory Metrics

Impacts

L1	3%
L2	4%
L3	35%
DRAM	57%

Shares

L1	143.683GB
L2	101.672GB
L3	101.782GB
DRAM	101.881GB

DRAM Program Total for All Functions and Loops
Performance: **4.61** GFLOPS
Arithmetic Intensity: **0.39** FLOP/Byte

Physical Cores: **48**   App Threads: **1**   Self Elapsed Time: **8.555** s   Total Elapsed Time: **8.555** s

Figure 14: Roofline of 1-thread-OMP version

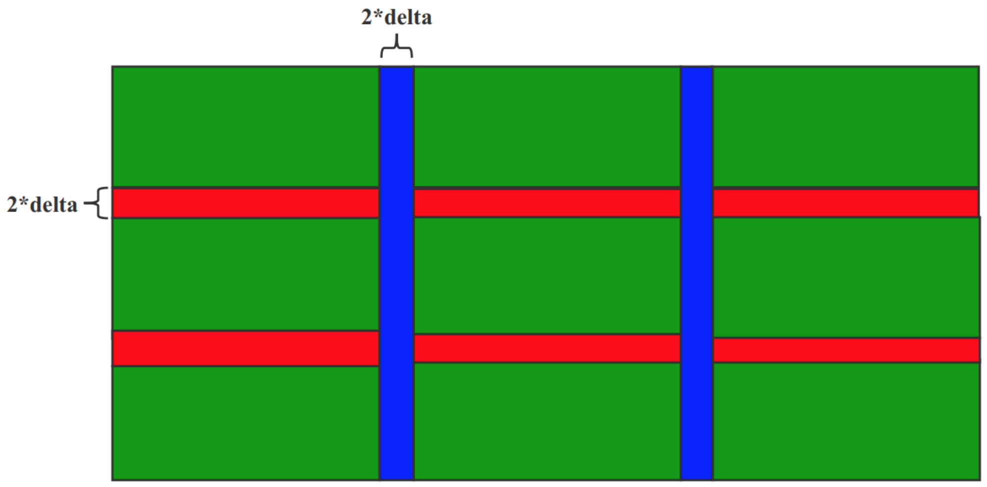


Figure 15: Block-wise structure:

implies the latter version is limited by the data transportation between cpu and memory. Moreover, we notice the right down of both figures, which is the memory metrics of most computational intensive spippet(red dot) of each version, behavior quite different. As for AVX version Figure 13, almost all data is serverd by L1 cache, while the data of 1-thread-OMP is mostly served by DRAM and L3, which are much more slower than L1. These phenomenon tell us the performance of OMP version is dominated by the memory access and also prove the guess and analysis we provided in **??**. Also, we notice that the AI and float point operation performance are highly decrease after we modify the AVX into 1-thread-OMP version. Next we would discuss how to solve these problems, by using block-wise per thread parallel.

4.2 Block-wise-OMP

We notice if directly apply the block splitting, according to the layout of our OMP code, there would be dependency. The structure of designed block-wise OMP prevent in 15 show the layout of our block-wise-OMP code. There are 3 types(colors) of blocks, when computing, we separately compute different color blocks. More speccifically, we use three large loops to calculate each type of blocks and parallel them inside each of them so that we could eliminate the dependency between each two blocks. It is noticed that inside each loop, we applied the computational method introduced in **??** to eliminate the dependency inside the block. This layout would prevent dependency, and in the mean time, if we select a small block size for the green blocks(which are computational intensive part), it is possible to get a a good locality. Due to limted time, we do not have time do adjust the block dimensional size, while we set it to 8, inspired by the cache line size.

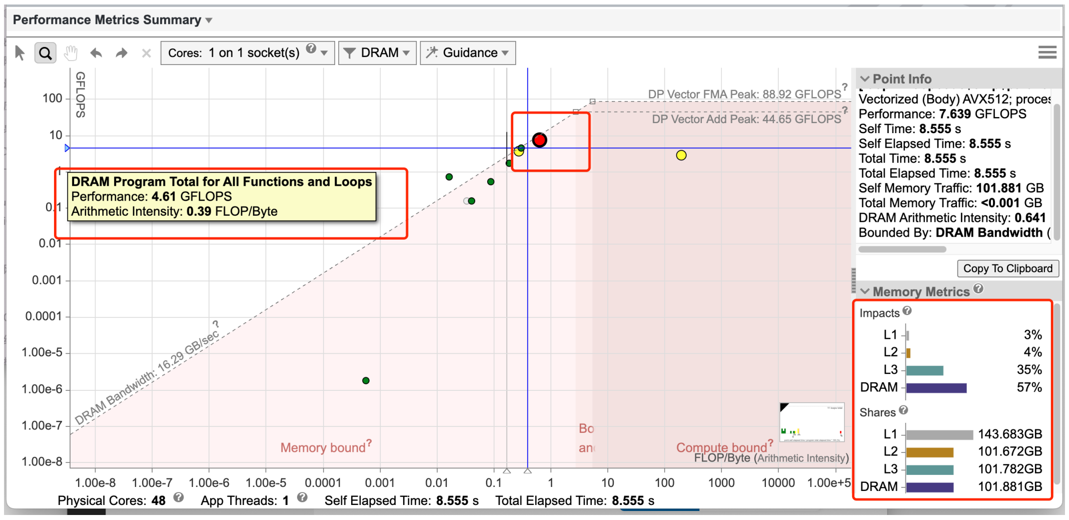
4.2.1 Roofline analysis of block-wise-OMP

Now we present the rooflines of p-thread-OMP and block-wise-OMP.

The key parameters here are $n = 2500; p = 36; i = 10$. Campared with p-thread-OMP version 16 whose data are mostly served by DRAM and L3, nearly half of data in block-wise-omp version data are served by L1 and L2, which implies cpu could get data faster and also prove we have enhance the locality. In the meantime, we change the base value type for Point Weight into "Self memory traffic" and we notice that in the p-thread-OMP there are two snippets are in memory traffic(red dot), while in block-wise-OpenMP version we notice all dot are green, which implies the memory traffic is relieved. Also, we notice that there is round 8 times improvement for the block-wise-OpenMP on both AI(arithmetic intensity) and float point performance.

4.2.2 Performance analysis of block-wise-OMP

Here we present the speed-up and efficiency of this parallel version. Compared to the speed-up and efficiency of p-thread-OMP in Figure 11, the block-wise-OMP produce a better performance in terms


GFLOPS
100
10
1
0.1
0.01
0.001
0.0001
1.00e-5
1.00e-6
1.00e-7
1.00e-8
DP Vector FMA Peak: 88.92 GFLOPS
DP Vector Add Peak: 44.65 GFLOPS
DRAM Bandwidth: 16.29 GB/sec
Memory bound
Compute bound
Bo
an
1.00e-8 1.00e-7 1.00e-6 1.00e-5 0.0001 0.001 0.01 0.1 1 10 100 1000 10000 1.00e+5
FLOP/Byte (Arithmetic Intensity)
Copy To Clipboard

Performance Metrics Summary

Cores: 1 on 1 socket(s) | DRAM | Guidance

DRAM Program Total for All Functions and Loops
Performance: **4.61** GFLOPS
Arithmetic Intensity: **0.39** FLOP/Byte

Point Info
Vectorized (Body) AVX512; proces
Performance: **7.639** GFLOPS
Self Time: **8.555** s
Self Elapsed Time: **8.555** s
Total Time: **8.555** s
Total Elapsed Time: **8.555** s
Self Memory Traffic: **101.881** GB
Total Memory Traffic: **<0.001** GB
DRAM Arithmetic Intensity: **0.641**
Bounded By: **DRAM Bandwidth** (

Memory Metrics

Impacts

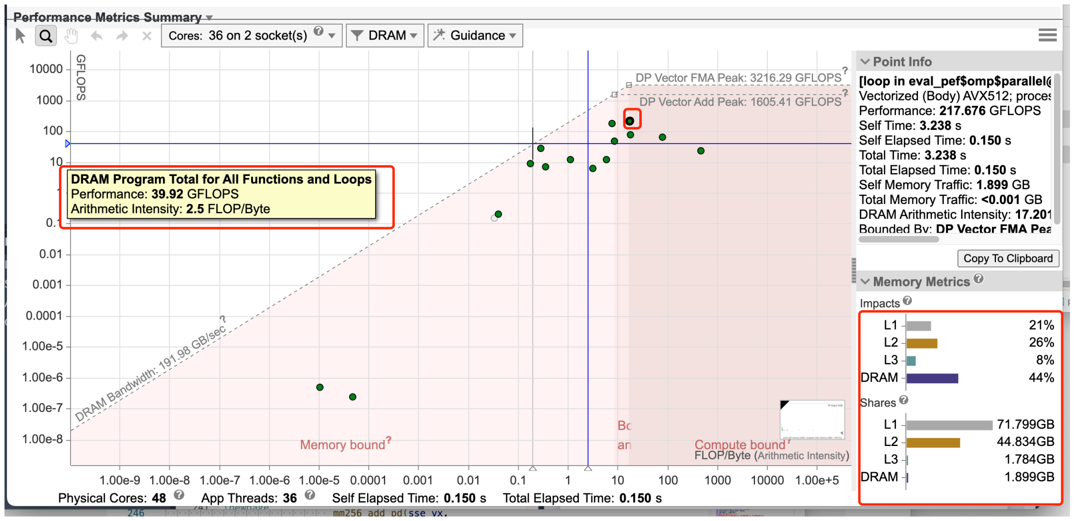
L1	3%
L2	4%
L3	35%
DRAM	57%

Shares

L1	143.683GB
L2	101.672GB
L3	101.782GB
DRAM	101.881GB

Physical Cores: **48** App Threads: **1** Self Elapsed Time: **8.555** s Total Elapsed Time: **8.555** s

Figure 16: Roofline of p-thread-OMP version


GFLOPS
10000
1000
100
10
1
0.1
0.01
0.001
0.0001
1.00e-5
1.00e-6
1.00e-7
1.00e-8
DP Vector FMA Peak: 3216.29 GFLOPS
DP Vector Add Peak: 1605.41 GFLOPS
DRAM Bandwidth: 191.98 GB/sec
Memory bound
Compute bound
Bo
an
1.00e-9 1.00e-8 1.00e-7 1.00e-6 1.00e-5 0.0001 0.001 0.01 0.1 1 10 100 1000 10000 1.00e+5
FLOP/Byte (Arithmetic Intensity)
Copy To Clipboard

Performance Metrics Summary

Cores: 36 on 2 socket(s) | DRAM | Guidance

DRAM Program Total for All Functions and Loops
Performance: **39.92** GFLOPS
Arithmetic Intensity: **2.5** FLOP/Byte

Point Info
[loop in eval_pef$omp$parallel(
Vectorized (Body) AVX512; proces
Performance: **217.676** GFLOPS
Self Time: **3.238** s
Self Elapsed Time: **0.150** s
Total Time: **3.238** s
Total Elapsed Time: **0.150** s
Self Memory Traffic: **1.899** GB
Total Memory Traffic: **<0.001** GB
DRAM Arithmetic Intensity: **17.201**
Bounded By: **DP Vector FMA Pea**

Memory Metrics

Impacts

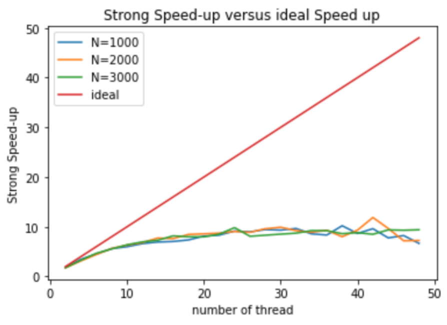
L1	21%
L2	26%
L3	8%
DRAM	44%

Shares

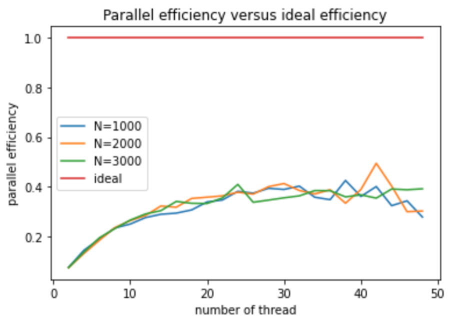
L1	71.799GB
L2	44.834GB
L3	1.784GB
DRAM	1.899GB

Physical Cores: **48** App Threads: **36** Self Elapsed Time: **0.150** s Total Elapsed Time: **0.150** s

Figure 17: Roofline of Block-Wise-OMP version



(a)



(b)

Figure 18: Speed-up and efficiency of block-wise OMP in different number of nodes 11(a) Speed-up of block-wise OMP in different number of nodes; 11(b) Efficiency of block-wise OMP in different number of nodes

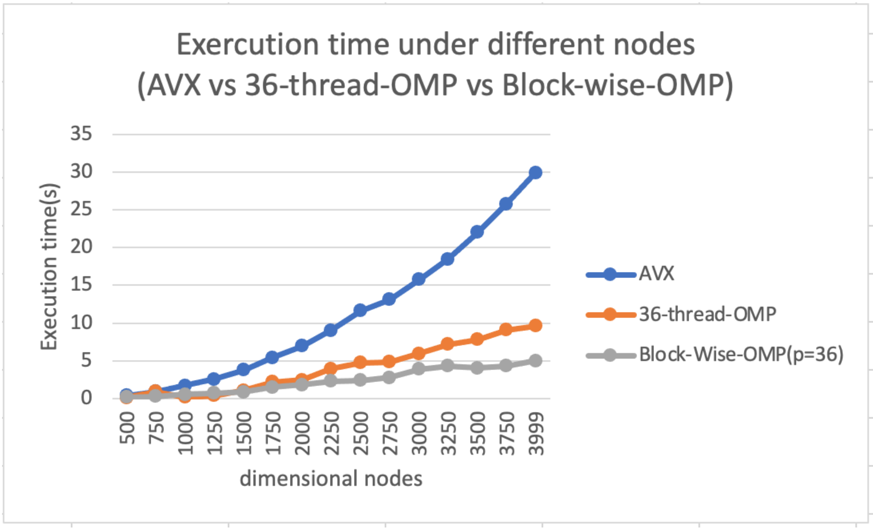


Figure 19: Execution time in different nodes(AVX vs p-thread-OMP vs Block-wise-OMP)

of both speed-up and efficiency in Figure 18. We also make a comparison with AVX version and p-thread-version. As can be seen from19(key parameters:$p = 36, i = 10, d = 4$), both inherit from AVX version, the p-thread-OMP and block-wise-omp both give acceleration and the block-wise OMP gives a more promising performance.

We will present the execution time of all methods required by this report in section 5

5 Final Performance Comparison

The key parameters here is $n = 36, d = 4, i = 10, p = 36(if available)$. As can be seen from the Table 21, for each step, the version inherit from the previous version all produce a remarkably better performance, especially when the problem size is larger. In detail, from Figure 20, we can see that the final version - OMP-block, has about **40 times improvement**, that is, 40 times faster than the original Main version. We would stop the report here and express my sincerely thanks for my lecturer and tutor who always pay their patience to answer my questions.

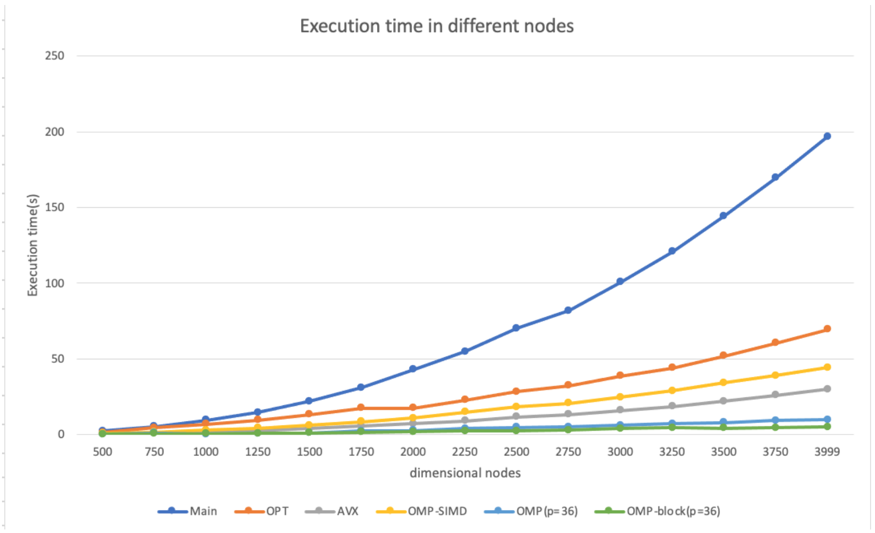
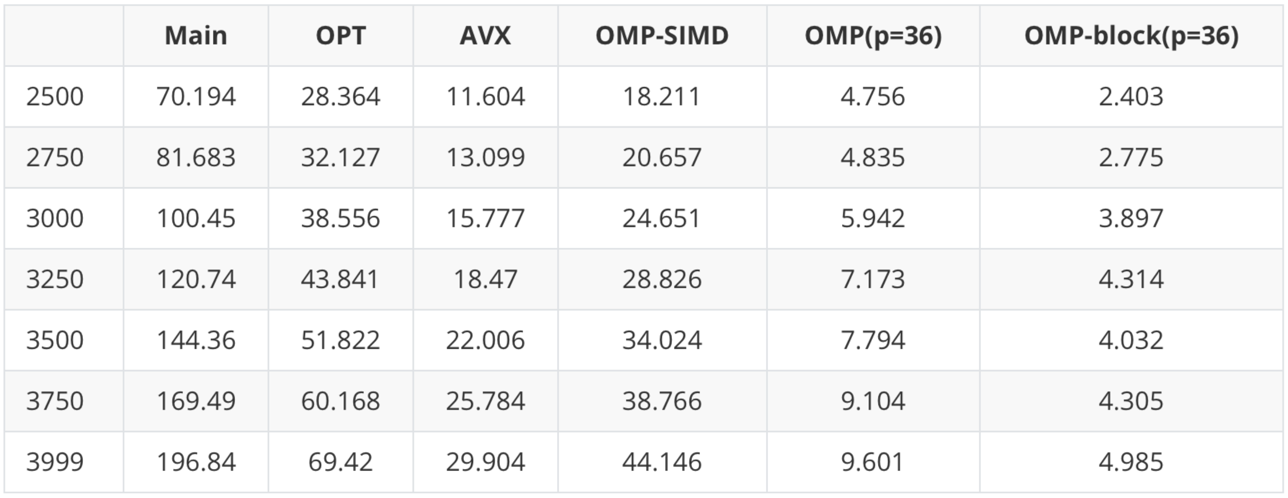


Figure 20: Execution time of all required method in different dimensional nodes)



	Main	OPT	AVX	OMP-SIMD	OMP(p=36)	OMP-block(p=36)
2500	70.194	28.364	11.604	18.211	4.756	2.403
2750	81.683	32.127	13.099	20.657	4.835	2.775
3000	100.45	38.556	15.777	24.651	5.942	3.897
3250	120.74	43.841	18.47	28.826	7.173	4.314
3500	144.36	51.822	22.006	34.024	7.794	4.032
3750	169.49	60.168	25.784	38.766	9.104	4.305
3999	196.84	69.42	29.904	44.146	9.601	4.985

Figure 21: Execution time of all required method in different dimensional nodes)