INF560 - ALGORITHMIQUE PARALLÈLE ET DISTRIBUÉE

Approximate Pattern Matching

11 mars 2023

Chinonso Stanislaus Ngwu & Changjie Wang

1 INTRODUCTION

This project uses the concept of high performance computing to implement a parallel version of approximate pattern matching algorithm (including string matching, sequence matching...) which is an important class of algorithms in Big Data and HPDA (High-Performance Data Analytics)[1]. The sequential algorithm provided by [1] was implemented using three different parallel computing protocols ; MPI (Message Passing Interface), OpenMP and Cuda. MPI is a communication protocol that enables multiple processors or nodes in a distributed computing environment to communicate and coordinate with each other to solve a single problem. It is widely used in scientific and engineering applications that require high-performance computing[2]. OpenMP (Open Multi-Processing) is an API (Application Programming Interface) that supports shared-memory multiprocessing programming in C, C++, and Fortran. It allows developers to write parallel code that can be executed on a single machine with multiple CPUs or cores[3]. CUDA (Compute Unified Device Architecture) : CUDA is a parallel computing platform and programming model developed by NVIDIA for their graphics processing units (GPUs). It allows developers to harness the power of GPUs to accelerate computations in a wide range of applications, from machine learning to scientific simulations[4].

The approximate pattern matching allows a sequence to be found in a large database with some differences including insertion and deletion. Thus, the notion of distance is used to check if a match occurs or not between the searched pattern and the current processed text. The code is based on the computation of the Levenshtein Distance which is a method for calculating the difference between two sequences, often used in information theory, linguistics, and computer science. It measures the minimum number of edits (insertions, deletions, or substitutions of single characters) needed to change one sequence into the other. It is named after the mathematician Vladimir Levenshtein, who introduced the concept in 1965 [5]. The sequential C implementation used as the basis for this project can be found in [6].

2 REPORT ORGANIZATION

The rest of the report is organized as follows, section 3 gives the general overview of all the algorithms. 3.1 gives an explanation of the different MPI implementations, while 3.2 discusses that of OpenMP and section 3.3 gives a general overview of the CUDA implementation. Section 4 discusses in detail the different experiments carried out so as to build the decision tree for our project while section 5 discusses the decision three. Finally, section 6 summarizes our results and performance improvement over the sequential version.

3
PARALLEL ALGORITHMS

3.1 MPI (Message Parsing Interface)

Four different algorithms were implemented for MPI taking into account three factors which includes ;

— The number of MPI processes
— The file size and
— The number of patterns.

The four algorithms will be discussed below.

— **Dynamic Distribution Over Patterns** : This MPI implementation uses process 0 to allocate patterns to other MPI processes dynamically. The performance and conditions favorable to this algorithm will be discussed in section 4.
— **Static Decomposition** : This algorithm divides the file evenly to the different MPI processes and each process searches for all the patterns in the allocated chunk (portion of file received).
— **Dynamic Distribution** : In this algorithm, the input file is divided into chunks of 10000 bytes. Processes zero then dynamically allocates each chunk to the other processes until the chunks are exhausted. This implementation has two variations :
  — **Patterns One by One** :In this approach, all the chunks are run through one pattern at a time. We take one pattern, divide the input file and search for matches and repeat the process for all the patterns.
  — **Patterns Together** : In this approach, for each chunk, we search for all the patterns at the same time.

3.2 OpenMP

Two different algorithms were implemented for OpenMP taking into account the number of threads available. The two implementations are discussed below :

— **OpenMP over Patterns** : This implementation assigns each pattern to an OpenMP thread.
— **OpenMP over chunks** : In this implementation, the chunks received by each MPI process are executed in parallel using OpenMP threads.

3.3 CUDA

The CUDA implementation is very similar to that of OpenMP except that it is run on the device. If CUDA is available, part of the work is allocated to CUDA while the remaining will be left for OpenMP. As will be discussed in section 4, we found that a ratio of 90% allocated to CUDA gave the best performance, all other factors remaining the same.

4 EXPERIMENTS

4.1 Choice of MPI implementation

4.1.1 • Dynamic Over Patterns Vs Dynamic over input file

The first experiment we conducted was to determine if it is best to divide the patterns over the MPI process (Each MPI process searches pattern matching using one pattern) as explained in section 3.1or divide the files into smaller chunks (10000 bytes found suitable in our tests) and have each ready MPI process dynamically assigned available chunk by rank 0 until all the chunks are exhausted. Then for each chunk, all the patterns are matched locally with the final global match aggregated at the end of all the processes by process 0. Below are the results using an input file size of 182KB, 10 processes and number of patterns ranging from 1 to 20.

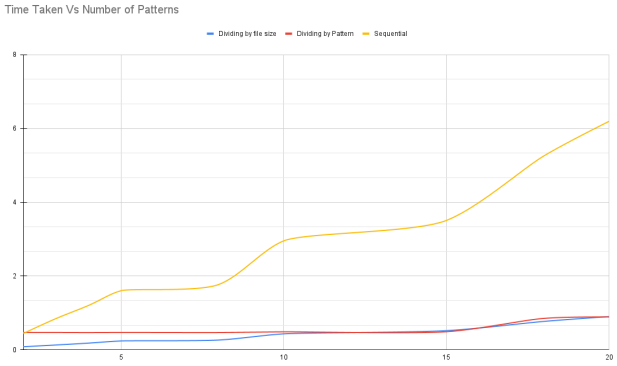


Figure 1 – Time taken Vs number of patterns

From the results obtained as can be observed from the figure above, we observed the strong correlation between the number of patterns and the number of available MPI processes. Mathematically and practically, when the number of patterns modulo MPI Process is equals to 0, we

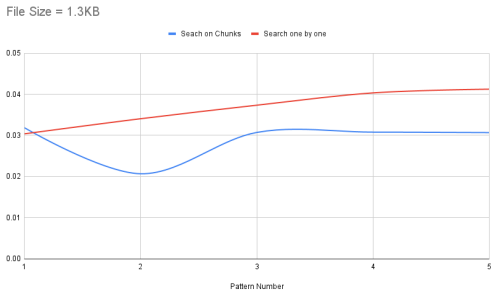
observed that dividing on patterns produces better results, otherwise, dividing by file (chunks) gives better outcome.

4.1.2 • Patterns One by One or together

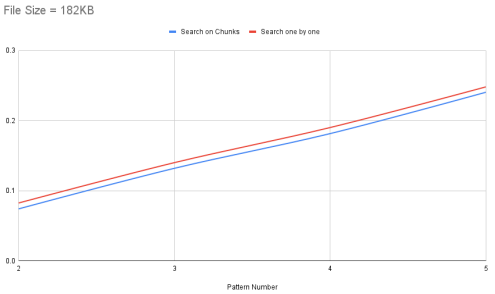
The second experiment tries to ascertain the best performance for two different implementations.

- When the MPI process 0, for each pattern splits the input file into chunks, matches the pattern with the entire file and repeats for all the input patterns. We called this implementation Dynamic Distribution One by One as explained in section 3.1
- The second scenario is when processes zero divides the file into chunks and for each chunk, all the input patterns are matched, this implementation is called Dynamic Distribution Patterns Together also explained in section 3.1

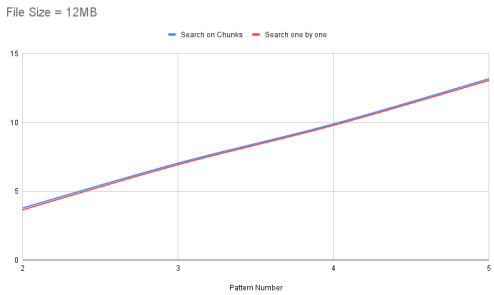
We conducted our experiment using four different file sizes ; 1.3K, 182K, 12MB and 50MB. The number of MPI processes was kept constant at 10 and the test was carried out for input patterns ranging from 1 to 5. The results are shown in the figures below ;



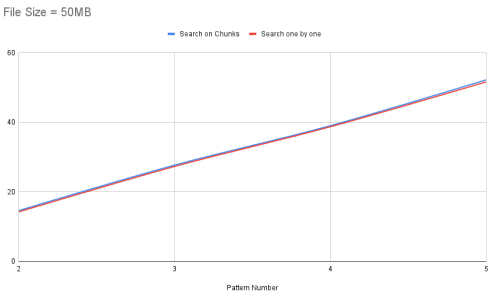
(a) File Size = 1.3KB



(b) File Size = 182KB



(c) File Size = 12MB



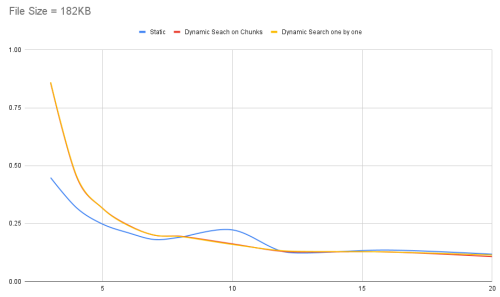
(d) File Size = 50MB

Figure 2 – Time taken Vs Number of Patterns

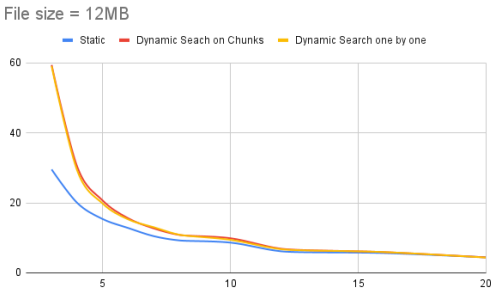
From the figures shown, we observed that searching one by one performed better when the file size is greater than 12MB. We have decided to use 10MB as the base since 12 MB performed better, we assume that the performance threshold will be around 10MB. This will be reflected later in our tree diagram.

4.1.3 • Static or Dynamic

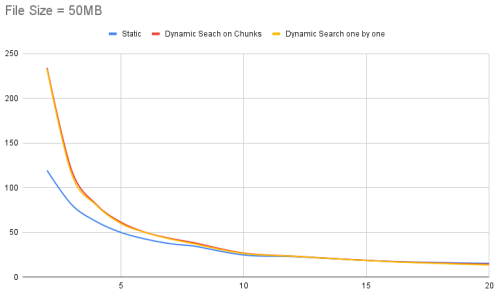
Here we compare dividing the input file equally between all the MPI processes or dividing it into fixed chunks and using process 0 to assign the tasks to ready MPI processes dynamically. We collected data for different numbers of MPI processes and also for different file sizes, using the two dynamic distribution implementations and the static decomposition. The results are shown below.



(a) File Size = 182KB



(b) File Size = 12MB



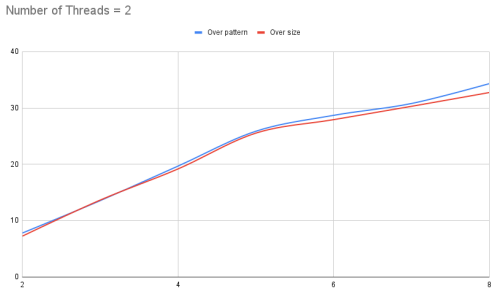
(c) File Size = 50MB

Figure 3 – Time taken Vs Number of process

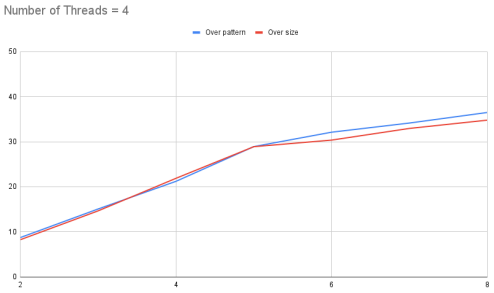
From the charts, we observed that the static implementation outperforms the dynamic implementations when the number of MPI processes is less than 16, after which the dynamic implementation outperforms the static.

4.2 Choice of OPENMP implementation : over pattern or over chunk

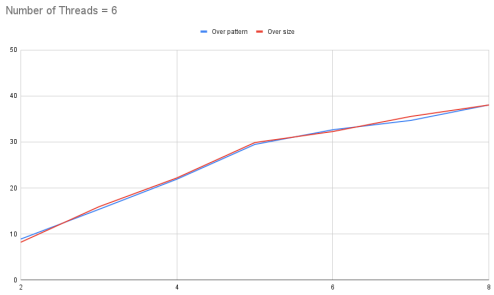
As explained in section 3.2, we have two OpenMP implementations, that is running the threads over the file chunks or over patterns. From the results obtained as shown in the figures below ;



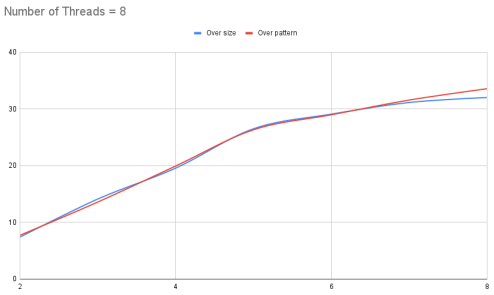
(a) Number of Threads = 2



(b) Number of Threads = 4



(c) Number of Threads = 6



(d) Number of Threads = 8

Figure 4 – Time taken Vs Number of Patterns

We observed that the run time of the OpenMP over file chunks and running it over patterns are almost same, where we cannot tell absolute difference. And we notice that when threads number is 2 or 4, running the OpenMP over file chunks tends to have a better outcome to some degree. As such, going forward, we will only feature OpenMP threads over the file chunks.

4.3 Choice of CUDA implementation

4.3.1 • GPU over dynamic pattern distribution or static decomposition

The fourth experiments we did was to test whether it is meaningful to implement CUDA on a dynamic size distribution. As a file is divided into large quantities of subfiles, it increases

the number of tasks, which should send/receive data from/to the GPU. This will definitely increase CPU-GPU interaction, possibly leading to poor performance. Table below shows the result. The test is done with number of processes = 8, number of threads = 8 and number of patterns = 6.

Table 1 – **CUDA Over dynamic or Static MPI**

File size	Methods	Time
50MB	Static	0.644516
	Dynamic	2.747882
12MB	Static	0.18714
	Dynamic	0.658648
4.5MB	Static	0.098804
	Dynamic	0.250363

From the data obtained, running a dynamic distribution over size with CUDA produced very poor performance. So if distribution over size is chosen and a CUDA is allowed, it is always better to choose Static decomposition.

4.3.2 • Balance between OpenMP and GPU

Since our CUDA implementation processes part of the chunks received by the MPI process in the device, we experimented to find the split ratio that gives the best performance between GPU and the OpenMP. We used 4 MPI processes, 2 nodes , 2 threads and 2 patterns for this test. Our results are shown in the figure below ;

From the chart shown below, we observed best performance for the ratio of 90% for GPU and 10% for OpenMP. (However, as the configuration of the machine varies, the percentage will not be fixed).

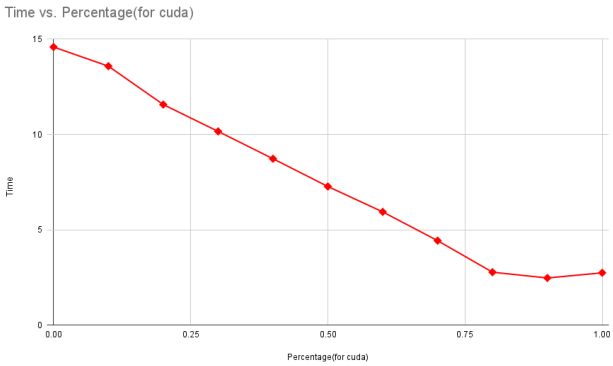


Figure 5 – Time taken Vs Percentage(for cuda)

5
DECISION TREE AND PROJECT IMPLEMENTATION

From the experiments conducted, our decision tree is shown below featuring 12 different possible scenarios, as shown in Figure 6, where decision 1 - 12 are different implementations featuring different combinations of MPI, OpenMP and CUDA as defined below :

- Dynamic Pattern Distribution
  - Decision 1 : pure MPI
  - Decision 2 : MPI + OpenMP
  - Decision 3 : MPI + CUDA
  - Decision 4 : MPI + hybrid (CUDA + OpenMP with balance)
- Dynamic Size Distribution
  - Decision 5 : pure MPI with pattern together
  - Decision 6 : MPI + OpenMP with pattern together
  - Decision 7 : pure MPI with pattern one by one
  - Decision 8 : MPI + OpenMP with pattern one by one
- Static decomposition
  - Decision 9 : pure MPI
  - Decision 10 : MPI + OpenMP
  - Decision 11 : MPI + CUDA
  - Decision 12 : MPI + hybrid (CUDA + OpenMP with balance)

How to go through the tree ?

1) Whether the number of patterns modulo N (number of process) is equal to 0 ?
   If so, choose dynamic pattern distribution and choose between Decision 1-4 based on one's own machine configuration, typically the number of threads per core and the GPU number.
2) If not, choose size distribution methods. Whether we have a GPU or not ?
   If so, choose the static decomposition with GPU, with Decision 11 and 12 varying based on OpenMP support.
3) If not, check whether we have enough processes larger than 16 ?
   If so, still choose the static decomposition but without GPU, with Decision 9 and 10 varying based on OpenMP support.
4) If not, differentiate the final methods by file size.
   If file size is higher than 10MB, we choose dynamic pattern distribution one by one, with Decision 7 and 8 varying based on OpenMP. If not, we choose the dynamic pattern distribution together, with Decision 5 and 6 as usual.

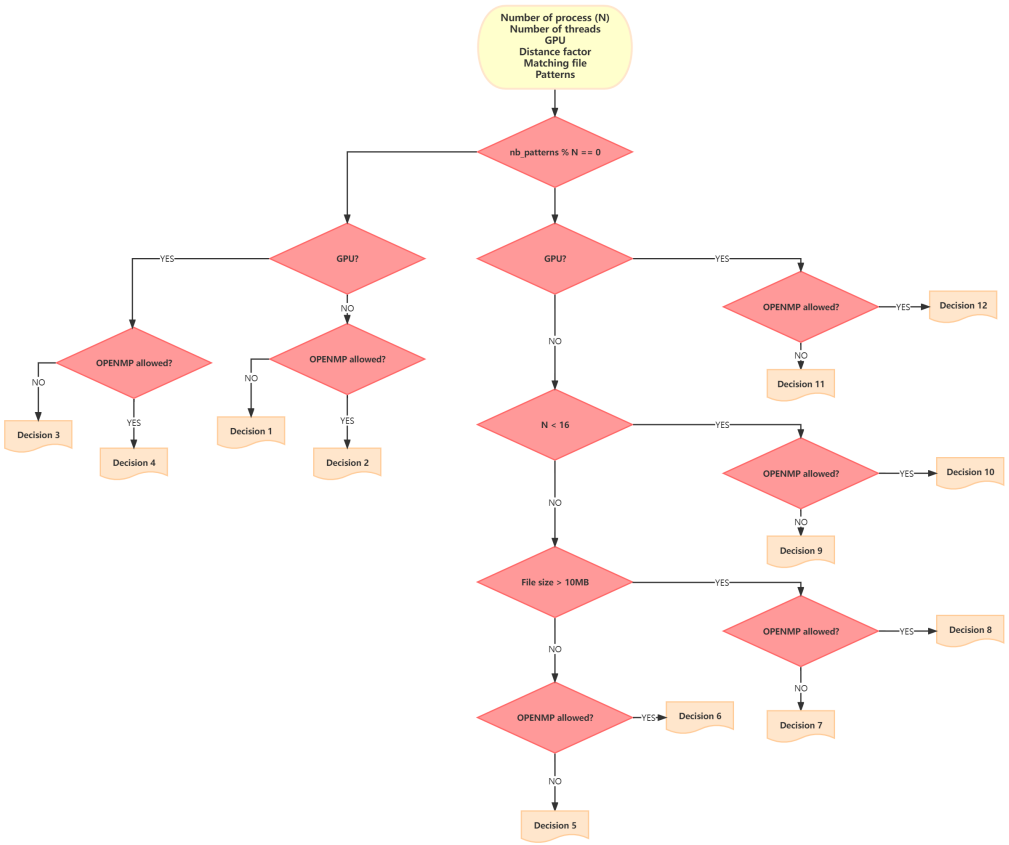


Figure 6 – Decision tree

6 RESULTS & CONCLUSION

From the implementation, we were able to obtain a performance increase of 101.53 using a combination of MPI + OpenMP + CUDA. The specific test parameters that gave these results are 8 MPI Processes, 8 OpenMP Threads, and a block dimension of 4 X 256. Better performance can be achieved by varying these parameters. The API we developed is able to provide the best parallel solution to pattern matching, according to the input features and machine configuration.

RÉFÉRENCES

[1] P. CARRIBAULT, "Inf560 - algorithmique parallèle et distribuée," 2023. [Online]. Available : https://www.enseignement.polytechnique.fr/profs/informatique/Patrick.Carribault/INF560/TD/projects/INF560-projects-0.html

[2] "Mpi : A message-passing interface standard version 3.1," 2015. [Online]. Available : https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[3] O. A. R. Board, "Openmp application program interface," 2013. [Online]. Available : https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf

[4] Nvidia, "Cuda c++ programming guide," 2023. [Online]. Available : https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[5] Wikipedia, "Levenshtein distance," 2023. [Online]. Available : https://en.wikipedia.org/wiki/Levenshtein_distance

[6] Wikipedia.org, "Algorithm implementation/strings/levenshtein distance," 2021. [Online]. Available : https://en.wikibooks.org/wiki/Algorithm_Implementation/Strings/Levenshtein_distance#C

A
DIVIDING BY PATTERN OR DIVIDING BY FILE SIZE (PROCESS = 10 , FILE SIZE = 182KB)

Pattern Number	Dividing by file size	Dividing by Pattern	Sequential
1	0.083856	0.471839	0.446333
2	0.131494	0.471042	0.84836
3	0.181654	0.467653	1.203276
4	0.240814	0.471821	1.603383
5	0.265404	0.468503	1.770617

B
DYNAMIC SIZE DISTRIBUTION ONE BY ONE OR TOGETHER (PROCESS = 10)

File size/nb$_patterns$		1	2	3	4	5
1.3K	Seach on Chunks	0.031867	0.020672	0.030716	0.030791	0.030674
1.3K	Search one by one	0.030382	0.034048	0.037333	0.040324	0.041237
182K	Search on Chunks	0.074003	0.131911	0.181469	0.240516	0.266647
182K	Search one by one	0.082393	0.140012	0.190119	0.248169	0.27026
50M	Search on Chunks	14.599001	27.680334	39.024858	52.216518	57.068166
50M	Search one by one	14.228341	27.298827	38.730045	51.63521	56.738859
4.5M	Search on Chunks	0.031388	0.020697	0.030939	0.030781	0.030766
4.5M	Search one by one	0.030354	0.033964	0.037071	0.040038	0.041587
12M	Search on Chunks	3.790627	7.051379	9.899495	13.188036	14.383363
12M	Search one by one	3.652503	6.942624	9.801624	13.069194	14.42314

C
STATIC OR DYNAMIC

File size	NB of process	2	3	4	5	6	7
50M	Static	119.290645	81.000506	62.070878	49.937954	42.455378	37.302567
	Dy (together)	234.081337	119.534521	81.352959	61.420454	50.010703	43.300922
	Dy (1x1)	232.312304	115.157773	80.478969	59.940338	49.724668	42.603404
12M	Static	29.550039	19.99502	15.366316	12.806569	10.473688	9.263509
	Dy (together)	59.409114	30.339956	20.645996	15.577794	12.69661	10.831846
	Dy (1x1)	59.062	29.250962	19.775321	15.267422	13.032014	10.882884
182K	Static	0.448337	0.318008	0.247846	0.210062	0.181251	0.191386
	Dy (together)	0.856989	0.451822	0.316164	0.242905	0.199763	0.195638
	Dy (1x1)	0.86068	0.456001	0.315561	0.240872	0.200315	0.194179
File size	NB of process	8	10	12	16	20	
50M	Static	34.522428	24.634955	22.811929	17.432599	15.242223	
	Dy (together)	38.2682	26.758947	23.379859	17.322745	14.296442	
	Dy (1x1)	37.054641	26.306375	23.177278	17.095829	13.522306	
12M	Static	8.64163	6.148058	5.660406	4.425882	3.795535	
	Dy (together)	9.884097	6.877896	5.927106	4.417429	3.551383	
	Dy (1x1)	9.441547	6.749474	5.935196	4.364714	3.660339	
182K	Static	0.222943	0.128238	0.135803	0.118049	0.131009	
	Dy (together)	0.162723	0.129491	0.127514	0.107389	0.090463	
	Dy (1x1)	0.160142	0.132315	0.127697	0.114206	0.087236	

D
STATIC OPENMP OVER PATTERN OR OVER SIZE? FILE SIZE = 12MB, PROCESS = 4

Thread Number	NB of Pattern	1	2	3	4
2	Over pattern	7.787925	13.539926	19.676829	25.865487
	Over size	7.237489	13.639108	19.170797	25.532206
4	Over pattern	8.730862	15.079972	21.214063	28.923515
	Over size	8.251127	14.661209	21.884359	28.904862
6	Over pattern	8.903309	15.34437	21.904542	29.464965
	Over size	8.212474	15.914038	22.182327	29.870681
8	Over pattern	7.728897	13.585251	19.935278	26.329415
	Over size	7.400943	14.120212	19.563817	26.499888
Thread Number	NB of Pattern	5	6	7	8
2	Over pattern	28.693977	30.795033	34.318644	36.221448
	Over size	27.926565	30.294215	32.739121	35.112242
4	Over pattern	32.123113	34.156248	36.497642	40.200091
	Over size	30.367876	32.958923	34.795917	37.382618
6	Over pattern	32.653484	34.68086	38.044508	40.6921
	Over size	32.267879	35.543164	38.029238	40.461576
8	Over pattern	28.966144	31.561978	33.563261	36.213318
	Over size	29.080277	31.132042	32.012596	36.226874

E
BEST PERCENTAGE TO USE GPU AND OPENMP(4 PROCESS 2 THREADS 2 PATTERNS)

Percentage(for cuda)	0	0.1	0.2	0.3	0.4	0.5
Time	14.586843	13.581344	11.569686	10.162482	8.728846	7.274345
Percentage(for cuda)	0.6	0.7	0.8	0.9	1	
Time	5.94355	4.43497	2.782366	2.479046	2.750077	