INF560 Project: Distributed Approximate Matching

Fabien Roger and Raphaël Habsieger

March 2023

Abstract

In this project, we developed a program to quickly search through a large corpus for approximates matches using the Levenshtein distance. We focus on parallelizing the search by splitting the data and the workload between different nodes using MPI, and between different CPU and GPU threads using OpenMP and Cuda. We run experiments to compare different strategies and tweak hyperparameters of our program. Our final program scales well with the number of nodes: it takes only 25% more time to run it when multiplying the number of nodes and patterns by a factor of 8.

1 Introduction

In this report, we take a closer look at the problem of find approximate pattern matching in a given string. More precisely, given a long string of DNA, we attempt to find the number pattern matches within a certain distance. Here, the distance used is the Levenshtein distance.

We will attempt to speed up a classical sequential version by parallelizing it using tools such as MPI, OpenMP and CUDA. We will have at our disposal the entirety of Ecole Polytechnique's student cluster which contains 172 machines, total 1472 physicals cores and 172 GPUs.

Note: this is a parallelization class and therefore no attempt at improving the core Levenshtein search was made.

2 General Methodology

2.1 Methodology Used for the Experiments

For the sake of simplicity, we restrict most of our experiment to 4 use cases with similar workloads (the complexity of the task is $O(N \sum_i m_i^2)$, where N is the length of the base sequences and m_i are the lengths of the patterns) but very different problem shapes:

- A search over large files (4 files of 10000 characters each) of a single 100 character pattern,
- A search over large files (4 files of 10000 characters each) of 100 patterns of 10-character long patterns,
- A search over smaller files (4 files of 1000 characters each) of 10 patterns of 100-character long patterns,
- A search over smaller files (4 files of 1000 characters each) of 1000 patterns of 10-character long patterns,

We work with 4 nodes, unless for scaling experiments. All experiments are done 5 times on randomly generated patterns (with a fixed seed). Both the mean and the 1 sigma standard deviation of runtime are reported.

2.2 Controlling the Program's Behavior

The behavior of our program can be controlled by environment variable, which allows us to experiment with different settings. When they are not provided, we use the defaults, which make the program perform at its best performance across a wide range of settings. Here is their definition and their default values:

1. `DISTRIBUTE_PATTERNS` (default : 1) : If true, each node will be assigned only a certain subset of the patterns. More details in section 3.2.
2. `PERCENTAGE_GPU` (default : 90) : Integer between 0 and 100 representing the percentage of data the GPU should process. More details in section 5.2.
3. `ONLY_RANK_0` (default : 0) : If true, only the node of rank 0 on each machine will process the data.
4. `FORCE_GPU` (default : 0) : If true, will abort if we can not run computation on the CPU.
5. `THREAD_PER_BLOCK` (default 128) : Integer specifying the number of thread per block to use when using the GPU.
6. `BLOCK_PER_GRID` (default : 65535) : Integer specifying the number of blocks per grid to use when using the GPU. More detail about these last two parameters in section 5.

3 Parallelization through MPI

3.1 General Approach

To simulate the case where processes have access to an advanced file sharing system, the user provides a folder name in the distributed file system of Ecole Polytechnique, and then each process tries to open a file name `i.tkt` where `i` is the rank of the MPI process. The final output is given in the stdout of the rank 0 process.

To deal with this parallel computing problem, we divided the problem in three steps:

1. Share the data between nodes with asynchronous send and receive
2. Process the data in each MPI process using OpenMP and the GPU
3. Reducing the number of time each pattern has been seen using reduce

Empirically, the second step takes the longest (it's in $O(m^2)$ where m is pattern length), therefore the overhead due to the slightly inefficient implementation of the first and last steps doesn't have a noticeable impact on performance.

To simplify the task of OpenMP and the GPU, each MPI process fakes having an array as long as the combined array of all sequence bits, by allocating only the required memory on the heap and the shifting the allocated pointer back.

3.2 Should We Distribute Patterns? Two Different Approaches

To split the work load and the data between MPI processes, we looked at two following two options:

1. Splitting only the base sequences. This minimizes the amount of data being shared between nodes and maximizes cache utilization for the base sequence.
2. Splitting the patterns, then splitting the sequence between nodes dealing with the same pattern. This makes it easier to share the work load between threads in a individual MPI process, and maximizes cache utilization for the patterns. It also minimizes the amount of unnecessary communications at each boundary.

Note: the reason why there is overhead at each boundary is that to compute if a pattern matches the base sequence at a given position, one need to know the next characters in the base sequence after that for as many characters as the pattern is long. Therefore, if a sequence of length L is split between N different nodes, the total communication and cost will be $N(L/N+m) = L+Nm$ if the maximum pattern length is m. This overhead can get large if there are many MPI processes and if the patterns are long.

3.3 Empirical Investigation

Using MPI + OMP, we get the following results when we vary the size of the dataset while maintaining the total load constant by varying the number of sequences:

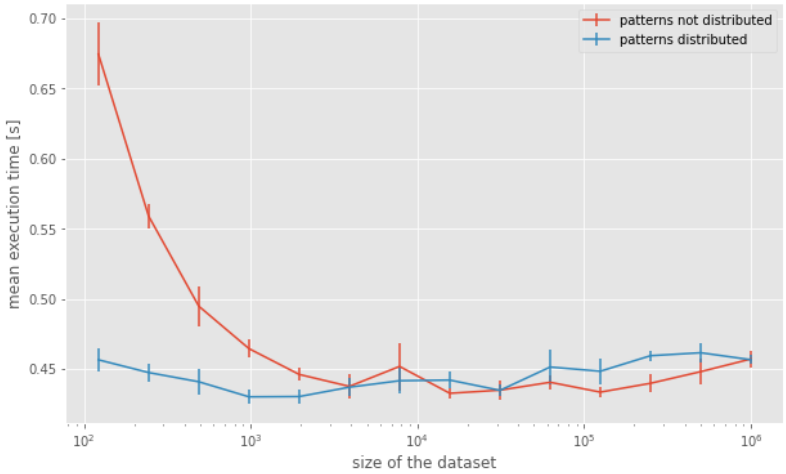

Figure 1: Impact of size and pattern distribution on execution time when using MPI and OpenMP given a fixed total workload

Using MPI + GPU we get the following:

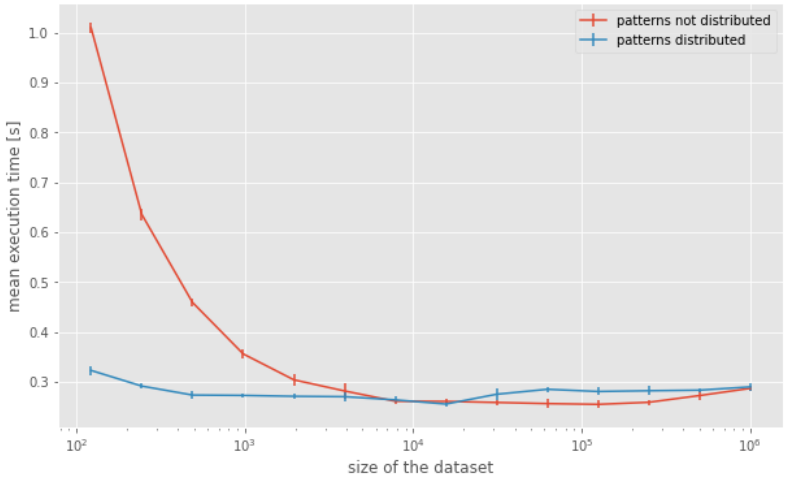

Figure 2: Impact of size and pattern distribution on execution time when using MPI and Cuda given a fixed total workload

Therefore, we decide to always distribute the pattern, as this is more robust, even though in some situations it can be slightly better to not distribute patterns.

4 Parallelization through OpemMP

Each OpenMP worker in the parallel section of our program goes through the following phases:

- Allocate the column computation buffer once per thread, using the size needed for the pattern of maximum length, and allocate a local matches counters;
- Go through the core loop and compute the number of matches for each pattern at positions OpenMP decided this worker had to deal with;
- Aggregates counts stored in the local matches counters using atomic operations (and free the buffer).

For the core loops, static scheduling as the cost of each subtask is the same. We also experimented with moving the OMP pragma instruction one line up, and adding `collapse(2)` to distribute both the positions and the patterns, but we found that it didn't improve performance, while making the program worse when patterns are of different sizes.

```
1 /* Loop over patterns */
2 for (i = starti; i < endi; i++)
3 #pragma omp for schedule(static)
4 /* Loop over positions */
5 for (j = start_openmp; j < end_openmp; j++)
```

This approach is suboptimal if there were billions of small patterns, but given that patterns are given as arguments and not in a file, the general way of the program would not support such a large number of patterns anyway. The other option mentioned above, coupled with a dynamic for loop attribution, would solve the problem, but introduce new complications about the block size used in dynamic attribution.

5 Parallelization through the GPU

5.1 How the GPU works

We decided to make each thread take care of checking the correspondence between a pattern and the base sequence at different positions. Each block is composed of threads working on different positions.

We need to dynamically allocate a buffer of the same size as the pattern length. Because the memory per block is limited, we cap the number of threads per block by asking the GPU how much memory per block is available and comparing that to the processed block.

But if patterns are small, we still want to split the workload between different blocks, which is why we allow ourselves to limit the maximum number of threads per block below what's required by the GPU.

And if patterns are small, we might also want each thread to deal with a lot of different positions so that the execution time of each thread is long enough, which is why we allow ourselves to limit the number of blocks per grid.

Here are the performance for each combination of those two parameters:

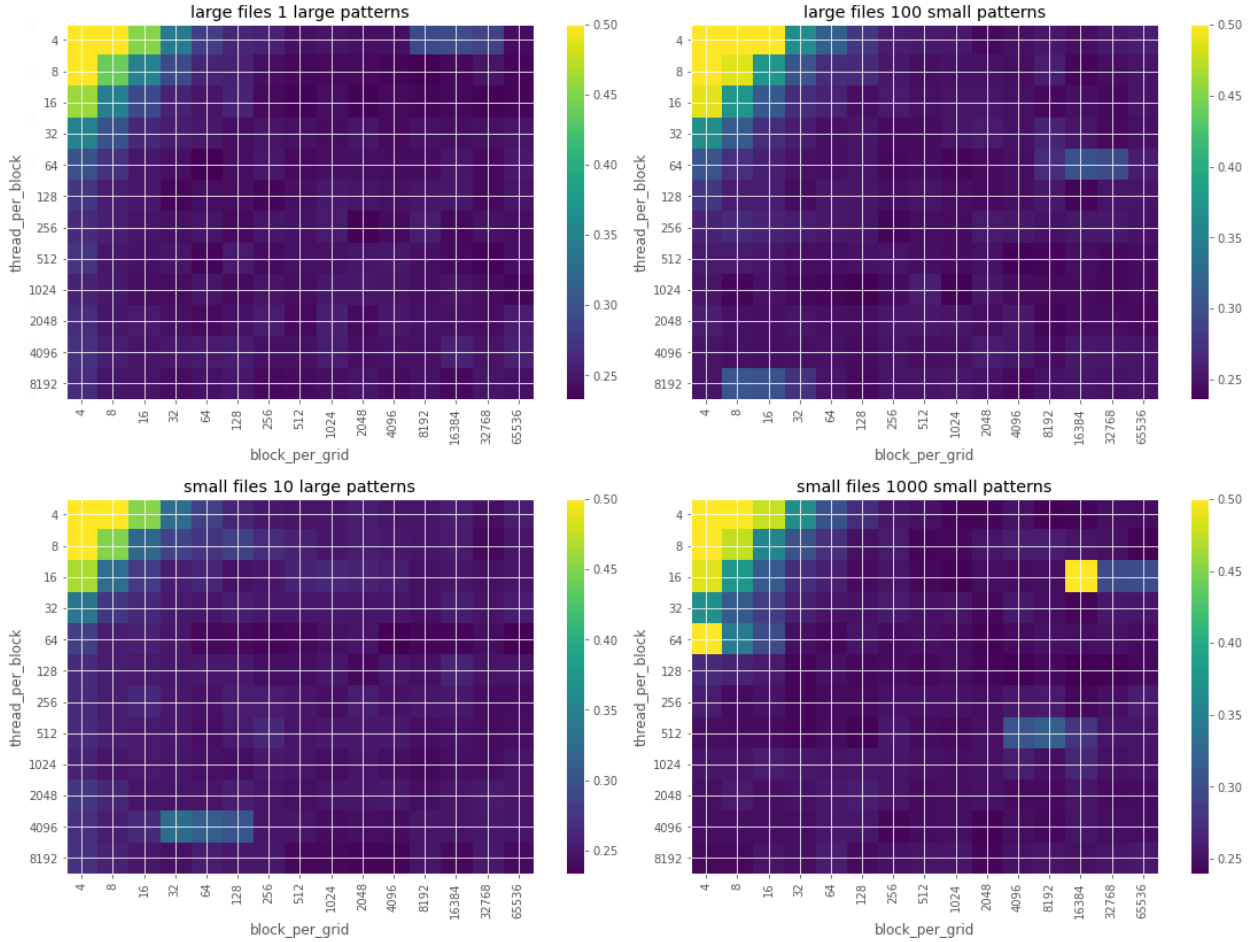

Figure 3: Run times for various combinations of the `THREAD_PER_BLOCK` and `BLOCK_PER_GRID` parameters for each scenario.

We can see that the upper left corner should be avoided, while all other values are about as good as each others, no matter the scenario. we chose to use the constant values of 256 maximum threads per block and at most 65536 blocks per grid (the maximum on most GPUs).

A strange finding is that some combinations make performance drop. We're unsure why.

5.2 How to split the data between GPU and OpemMP

First and foremost, before using we perform multiple checks:

- Indeed a GPU is available on the machine using `cudaGetDeviceCount`
- Afterwards, we check that there is enough shared memory per block in the GPU to store an int array of the size of the biggest pattern : such an array is used when computing the levenshtein distance. We perform those check with `cudaGetDeviceProperties` and by accessing the `sharedMemPerBlock` field of the properties.

If any of those checks fail, we decide not to use the GPU at all even if smaller adjustments may have been possible.

If we decided to use the GPU, we split the data according to the parameter `PERCENTAGE_GPU` so that `PERCENTAGE_GPU`% of the data will be processed by the GPU and (100 - `PERCENTAGE_GPU`)% of the data will be processed regularly using the CPU and MPI. We copy slightly more data to the GPU than `PERCENTAGE_GPU`% for the sake of the computation. More precisely we copy the data

+ the length of the longest pattern + 1 bytes to the GPU. It can be summarized in the following scheme :

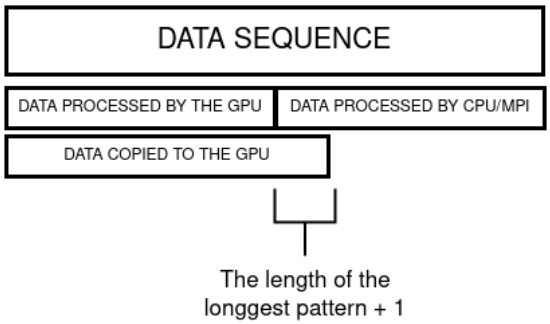

Figure 4: Data copied and processed by the GPU when `PERCENTAGE_GPU` = 50%

After integrating GPU usage, we decided to implement a hybrid approach where we split the data between the GPU and CPU. By dividing the workload between the two devices, we can handle larger amounts of data and more complex computations, resulting in faster and more efficient processing.

To determine the optimal value for `PERCENTAGE_GPU`, we ran on each scenario the measurement with different value for `PERCENTAGE_GPU`. The results can be seen in the figure below.

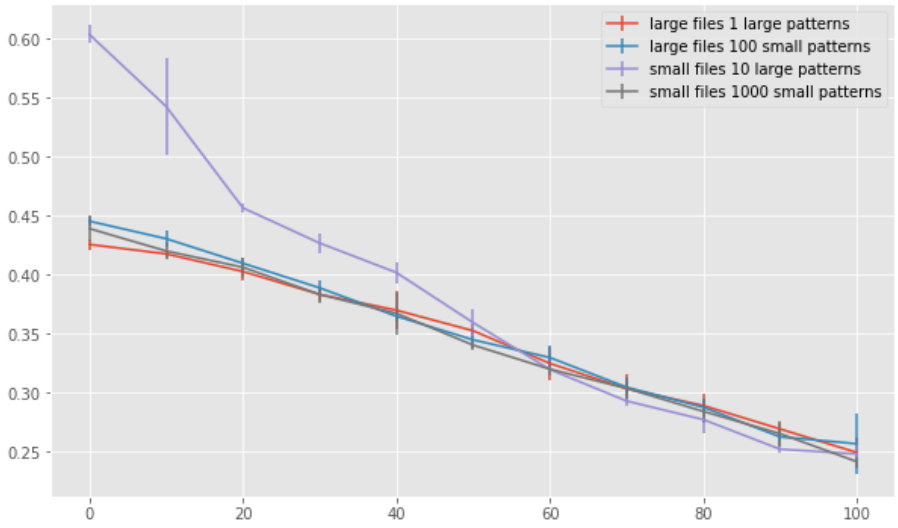

Figure 5: 4 scenarios, effect of `PERCENTAGE_GPU` on the run time

We see that higher values of `PERCENTAGE_GPU` lead to better result up to a threshold around 90 percent where the runtime remain constant. This can be explained by the fact that the GPU is fully used and using the cpu starts to become worth it.

6 Results

6.1 Comparison of the performance on the scenarios

We compared in different scenarios the speed of using the different techniques. The detail of each scenario can be found in section 2.1. First without parallelization, then adding only MPI, then adding OpenMP and finally also using the GPU. We notice that every time we add a new parallelization layer, it provides a speed-up over the previous version.

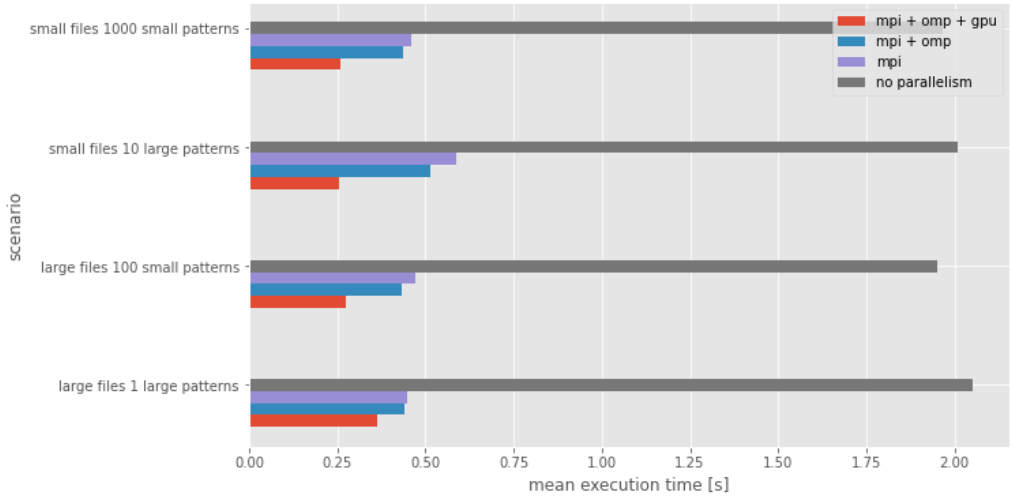

Figure 6: Speed up of different parallelization layer on different scenarios

6.2 Scaling

We measure the mean run time as we scale the number of patterns and nodes, by modifying the scenarios and scaling the number of patterns proportionally to the default case (4 nodes).

By Amdahl's Law, we expect the execution time not to be linear with respect to the number of nodes. This is caused by the sequential part of our program as well as some part of the overhead of the different parallelization techniques, which sometimes scales with the number of nodes.

More precisely, we find that there is a significant slow-down between 1 and 2 nodes, which is probably due to a high communication cost between nodes, then the run time stays relatively stable until 32 nodes, before going a bit up when we scale up to 64 nodes. Overall, this shows that our program scales relatively well when the amount of data and compute increases.

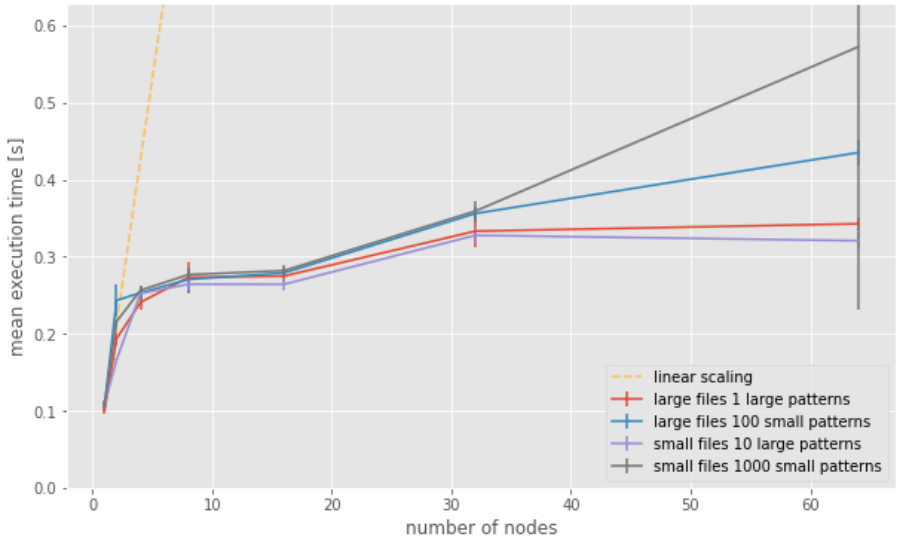

Figure 7: Weak scaling law of our program: mean run time as we scale the number of patterns and nodes.

7 Conclusion

Our program achieves efficient data parallelism by sharing patterns then the base sequence with MPI and sharing work loads within nodes with OpenMP and Cuda, enabling the user to search thousands of 50-character long patterns over megabytes of text using the costly Levenshtein distance, in less than a second.

8 Acknowledgments

We express our appreciation to Patrick Carribault, our teacher, for his valuable guidance and support throughout the course.