Project INF560
Approximate Pattern Matching

Lino Moises Mediavilla Ponce, Paolo Calcagni

17 March 2022

1 Parallelization Strategies

1.1 Patterns Over Ranks

The overall strategy is to distribute the patterns over the available MPI ranks and within each of them, search the given pattern by leveraging Open MP parallelism.

1.1.1 MPI Ranks

Rank 0. It is in charge of reading the database (in FASTA format) from the file system and distributing it to all other ranks via an MPI Broadcast operation. Afterwards, it distributes the patterns among the other ranks, in a round-robin fashion if there are fewer ranks than patterns to search. Then it simply waits in a loop receiving the results of each 'worker' rank, updating a local array to report the results when all the patterns have been processed.

Other Ranks. Each of them will work on 1 pattern at a time; searching it against the entire database (stored in memory). To speed up this computation, each Rank independently uses a combination of CUDA and OpenMP, and at the end, returns the computed number of matches corresponding to the assigned pattern. It then waits until receiving a new pattern or a stop value from the master rank.

1.1.2 Open MP Threads

Inside each MPI Rank (apart from the master rank) there is an Open MP parallel region to split the work of searching 1 pattern in the complete database.

The number of OpenMP threads can be 1; in this case, the code behaves sequentially within each rank. But if there are more threads available, each one takes a chunk of the input database and searches occurrences of the pattern assigned to their parent MPI rank with the Levenshtein function[1].

If there is a GPU available, the Open MP threads will work on only a fraction of the input database, and the rest will be the duty of the GPU.

1.1.3 GPU

If there is a GPU available, each MPI Rank divides the search of its assigned pattern between the available OpenMP threads and the GPU (via a CUDA kernel). Both of these computations run concurrently.

In case there is not such a device, the code falls back to parallelization with Open MP only.

[1]No parallelization took place inside the Levenshtein function itself

1.2 Database Over Ranks

The main idea is to split the database into different pieces and assign each of them to different MPI Ranks: every MPI Rank searches for all patterns in its own piece of the database. We also improved parallelization through openMP threads and GPU.

1.2.1 MPI Ranks

After doing some basic stuff (initializing MPI, reading command lines etc.) the workflow is split depending on the number of the MPI Rank.

Rank 0

- Reads the database and the patterns.
- Divides the database into n pieces where n is *Processes - 1*. It's subtracting 1 from the total number of processes since the Rank 0 itself doesn't participate actively in the pattern search.
- Communicates with all the other ranks[2], in order to assign them their own pieces of the database. More in detail, it sends two integers that represent the first and the last byte of the piece of the database that should be read. If the division *sizeDatabase/numberPiecesDatabase* has a remainder, the left characters will be assigned to the last rank.
- Initializes the number of matches to 0.
- Iterates over the patterns: for each pattern, Rank 0 waits to receive the answers from all the other ranks (number of matches found for that pattern for that piece of database).
- Prints the results.

Other Ranks

- Reads the database and the patterns.
- Receives from the Rank 0 two integers representing the first and the last byte of the piece assigned.

[2]Through MPISend

- Iterates over the patterns: reads the piece of database and calculates the number of matches for each pattern. If the rank is not the last one, it needs to include some extra characters from the next piece in order to not lose matches that are placed between one piece and another one. To be more precise, it has to take into account *sizePattern - 1* characters.

- Sends to Rank 0 the number of the matches found for each pattern.

1.2.2 OpenMP Threads

If there are more threads available, they are exploited in this way: every thread searches for a different pattern in the piece of the database assigned to the MPIRank.

1.2.3 GPU

If a GPU exists and the number of patterns is greater than one, the GPU will be used to further parallelize the searching of different patterns. Patterns will be split: GPU takes care of the first half while openMP threads take care of the last ones.

The workflow of the program is the following:

- Before entering the parallel region, the only thread available sets up the GPU and calls the kernel function. The calls of CUDA kernel, so the control comes immediately back to the CPU, that doesn't have to wait for the kernel to finish its operations.

- The parallel region is entered and threads search for patterns in parallel.

- After the parallel region is exited, the result of the GPU are transferred to the host.

We didn't use the other approach which consisted in splitting the workload on more MPI Ranks on the same node (one using the GPU and the other ones exploiting threads). In this approach, the MPI rank that deals with GPU doesn't do anything other than setting up the GPU and waiting for the result. Moreover, if you have the same number of nodes and processes[3] no GPUs are exploited.

This approach can still be improved: the program should understand when using a GPU creates too much overhead and it's not worth it. In other words, if the use of the GPU implies an increase of the execution time and not a reduction of it, the GPU shouldn't be used even if available[4].

[3]For example salloc -N 5 -n 5

[4]It has not been possible to implement this reasoning in the code for one reason explained later on.

2 Cost Model

2.1 1 Pattern

Both of the approaches previously presented have their own advantages and disadvantages. Let's begin to reflect when it's better to use one than the other one considering very simple cases: assume now that there is just one pattern to search for.

Patterns over Ranks. Only one MPI rank is exploited; however, openMP threads always work since they split the database in pieces.
Needless to say, the best-case scenario is when the hardware provided is 1 MPI rank and multiple threads: the program could use the rank and all the threads, making the most of the hardware.
The opposite case is the worst-case scenario: multiple ranks wouldn't be used and only one thread would be used. Basically, it behaves like a sequential program.

Database over Ranks. Every MPI rank searches for the pattern in its piece of the database. OpenMP threads are not active because there is just 1 pattern to search for.
With 1 rank and multiple threads, this code behaves like the sequential program: the database is not split into pieces and threads are not used since there is only 1 pattern.
Instead, with multiple ranks and 1 thread, the code uses all the MPI ranks to split the database and uses the only thread available to search for the pattern: this is the best-case scenario with all the hardware being used.

It's really intuitive to understand how many threads are lost in the respective approaches:

$$LostThreadsPatternsoverRanks = (ActiveMPIRanks - 1) * OMPThreads$$

$$LostThreadsDatabaseoverRanks = ActiveMPIRanks * (OMPThreads - 1)$$

If we would want to choose the best approach in the case we have only 1 pattern, we could simply choose the approach that loses loss threads.

For example, with 5 MPI Ranks and 2 threads, Patterns Over Ranks would lose 8 threads, while Database Over Ranks 5. Database Over Ranks should be used. With 10 Threads, Patterns over Ranks loses 40 threads and Database over Ranks 45 threads. Patterns over Ranks should be used.

2.2 Multiple Patterns

If we have multiple patterns the cost model is more tricky.

Patterns over Ranks

- If the number of patterns is equal to the actual MPI ranks, there is no round-robin. All the hardware is being used. Every MPI rank has a pattern. All the ranks will finish at the same time, after one iteration.
- If the number of patterns is greater than the actual MPI ranks round robin takes place. If the execution time of an iteration of round-robin is a divider of the time slice, all the hardware is being used. Otherwise, the use of the hardware is not optimized.
- If the number of patterns is less than the actual MPI ranks, some MPI ranks are not used. The use of the hardware is not optimized.

Database over Ranks

- If the number of patterns is equal to the number of threads, there is no round-robin. All the hardware is being used.
- If the number of patterns is greater than the number of threads, there is round-robin. If the execution time of an iteration is a divider of the time slice, all the hardware is being used. Otherwise, the use of hardware is not optimized.
- If the number of patterns is smaller than the threads, some threads are not used. The use of the hardware is not optimized.

2.2.1 Cost Model

To calculate the best approach to use, in other words the one that loses less hardware, we can do the following:

$dimensionOfIterationPatternsOverRanks = ActiveMPIRanks/patterns$

$dimensionOfIterationDatabaseOverRanks = ThreadsPerRank/patterns$

We can calculate HardwareOptimizationPatternsOverDatabase and HardwareOptimizationDatabaseOverPatterns in this way. We assume that x is dimensionOfIterationPatternsOverRanks or dimensionOfIterationDatabaseOverRanks.

```
while(x < 1){
x = x * 2
}
ratioHardwareOptimizationApproachChosen = x - 1;
```

From now on we abbreviate HardwareOptimizationPatternsOverDatabase with ratioPatterns and HardwareOptimizationDatabaseOverPatterns with ratioDatabase.

```
if(ratioPatterns == 0 && ratioDatabase == 0){
   Both of the approaches use the hardware at its maximum capacity.
   We could choose the approach randomly or in a predetermined way
      (as we did in the code).
}
else{
   Otherwise, we calculate the minimum between ratioPattern and
      ratioDatabase and we choose the approach which optimizes
      better the use of the hardware.
}
```

Example 1. Let's assume we have 2 MPIRanks, 4 threads, 4 patterns.

$$dimensionOfIteratioPatterns = ActiveMPIRanks/patterns = 2/4 = 0.5$$

$$dimensionOfIterationDatabase = ThreadsPerRank/patterns = 4/4 = 1$$

$$ratioPatterns = 0$$

$$ratioDatabase = 0$$

Both of the approaches maximize the usage of hardware: let's see why.
Let's assume that the database contains 100 characters.

- **PatternsOverRanks** spreads the first 2 patterns over the 2 MPI ranks and when they finish, it spreads the 2 left patterns over the ranks again. Each MPIRank split the database in the number of threads, 4 in this case: each of the 4 threads has to analyze 25 characters. Suppose that each thread needs 1 second for each character: it means that each thread needs 25 seconds. After 25 seconds the 2 threads finish. A second-round has to be done: the other 25 seconds will pass. The total execution time will be then 50 seconds.
- **DatabaseOverRanks** spreads the database over ranks, 2 in this case. Every rank has 50 characters in the database. Each MPIRank has 4 threads, in this case, each thread will search for a different pattern in all the database: 50 characters. Each thread will finish in 50 seconds. After both of the threads will finish after 50 seconds, there is no need for a second round.

As we can see, the execution time is the same.

Example 2. Let's assume we have 4 MPIranks, 5 threads and 6 patterns.

$dimensionOfIteratioPatterns = ActiveMPIRanks/patterns = 4/6 = 0.66$

$dimensionOfIterationDatabase = ThreadsPerRank/patterns = 5/6 = 0.83$

$$ratioPatterns = 0.32$$

$$ratioDatabase = 0.66$$

Let's verify that Patterns Over Ranks is the best approach.
Let's assume that the database contains 100 characters.

- **PatternsOverRanks** spreads the first 4 patterns over the 4 MPI ranks and when they finish, it will spread the 2 left patterns over the ranks again. Each MPIRank splits the database in the number of threads, 5 in this case. Each of the 5 threads has to analyze 20 characters.

Suppose that the thread needs 1 second for each character: it means that each thread needs 20 seconds. After 20 seconds the 5 threads finish; a second round has to be done. 2 left patterns are spread over the ranks. The total execution time will be then 40 seconds.

- **DatabaseOverRanks** spreads the database over ranks, 4 in this case: every rank has 25 characters of the database. Each MPIRank 5 threads in this case: each thread will search in its piece of the database: 25 characters. Each thread will finish in 25 seconds. One pattern is still left. So the program will finish in 50 seconds.

As we can see, both of them are not using all the hardware; however, Patterns Over Ranks finishes before, as expected.

3 Experimental Results

Here are some experimental results: all of them can be found in the file *experiments.xlsx*.

Example 1. Let's assume we have 4 MPIranks, 3 threads, 6 patterns and a database with 100 characters.

$$dimensionOfIterationPatterns = ActiveMPIRanks/patterns = 4/6 = 0.6$$

$$dimensionOfIterationDatabase = ThreadsPerRank/patterns = 3/6 = 0.5$$

$$ratioPatterns = 0.2$$

$$ratioDatabase = 0$$

Execution time of Patterns over Ranks: 54 seconds
Execution time of Database over Ranks: 41 seconds

Example 2. Let's assume we have 4 MPIranks, 5 threads, 6 patterns and a database with 100 characters.

$$dimensionOfIteratioPatterns = ActiveMPIRanks/patterns = 4/6 = 0.6$$

$$dimensionOfIterationDatabase = ThreadsPerRank/patterns = 5/6 = 0.8$$

$$ratioPatterns = 0.2$$

$$ratioDatabase = 0.6$$

Execution time of Patterns over Ranks: 30.42 seconds
Execution time of Database over Ranks: 43.24 seconds

Further Experiments. Several such examples were carried out, in which the number of MPI Ranks and OpenMP threads were modified systematically to understand the behaviour of both approaches and verify the hypotheses of our formulated Cost Model.

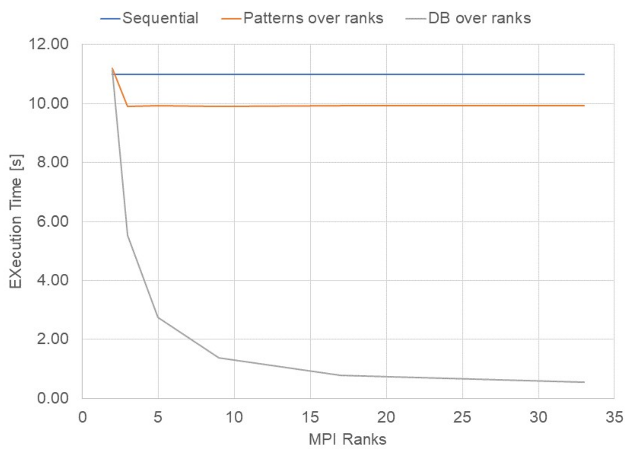

Figure 1: Execution time as a function of the number of MPI Ranks on 1 pattern and a medium-sized database

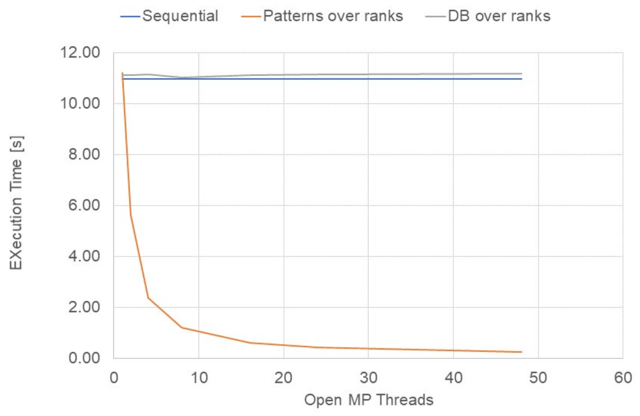

Figure 2: Execution time as a function of the number of Open MP Threads on 1 pattern and a medium-sized database

In Figures 1 and 2 we can observe the complementary nature of our parallelization strategies. For instance, the *Patterns Over Ranks* approach scales with the number of threads but is indifferent to the number of MPI Ranks as soon as these begin to exceed the number of patterns to search. Whereas, the *Database Over Ranks* approach, scales with the number of MPI Ranks but does not benefit from an increase in the number of Open MP threads

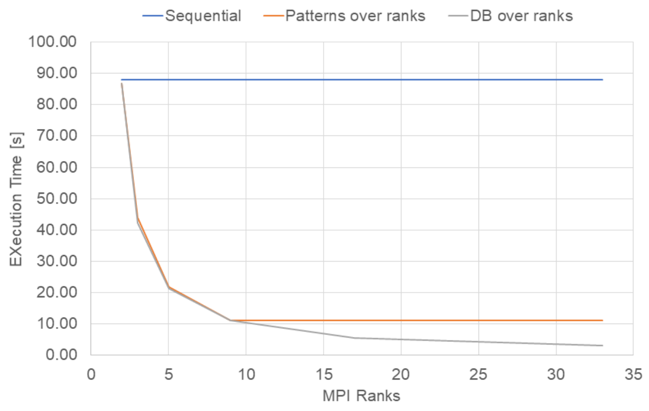

Figure 3: Execution time as a function of the number of MPI Ranks on 8 patterns and a medium-sized database

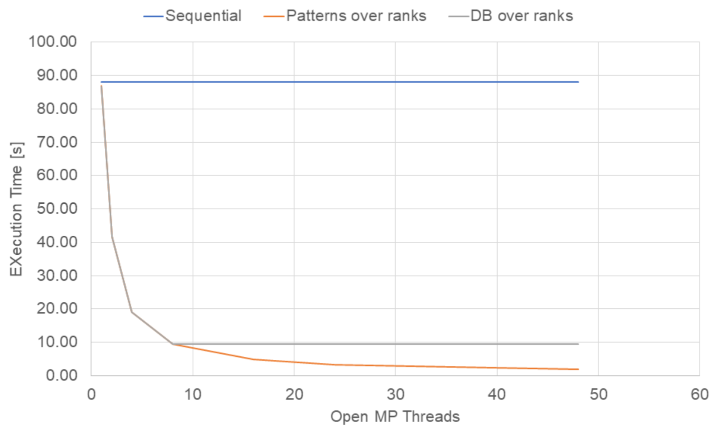

Figure 4: Execution time as a function of the number of Open MP Threads on 8 patterns and a medium-sized database

when they begin to exceed the number of patterns to search.

In Figures 3 and 4 however, we observe that the approaches both scale at a similar rate, but then begin to diverge and adopt a similar behaviour to what was observed in the case for 1 pattern, i.e. scaling stops completely once the number of ranks or threads exceeds the number of patterns to search, depending on the approach.

4 Notes

4.1 Performance of reading database

PatternsOverRanks: the node master is the only rank that is reading the buffer and patterns from the command line. After that, it broadcasts data over the other patterns.

DatabaseOverRanks All of the MPI Ranks are reading the buffer and the patterns from the command line, using the shared file system (since there was no constraint about this).

Of course DatabaseOverRanks is faster since all the ranks are reading at the same time: the ranks don't have to wait for the master to send the data. However, this improvement in performance is not visible in the experiments: this is because we started the timer after all the ranks received patterns and buffer. In this way, we could compare the two approaches.

4.2 Database Over Ranks - Bug openMP Threads

There is an annoying unexpected behaviour in the implementation of openMP of DatabaseOverRanks: even if everything is implemented well, we don't see the results expected. With double of the threads we would expect the execution time to halve but this doesn't happen. We spent a lot of time trying to find the problem[5] but we didn't manage to do it. We found that the problem is inside the Levenshtein function. We guess that there should be some problems with the compiler or at execution time, nothing correlated with our code.

In order to be able to compare the hybrid approach MPI + openMP with other approaches, we added a special flag *TESTPERFORMANCENOLEVEN-SHTEIN*. If it's 1 the code inside the Levenshtein function is not executed; instead, a sleep of 1 microsecond is performed. In this way, we obtain the results expected[6].

[5]Including a videocall with professor on Wednesday 9th March

[6]After this implementation we were also reasonably sure that it was not our code's fault since with the sleep everything parallelizes well).

4.2.1 Sleep in CUDA Code

Our idea was to use the same approach in CUDA, putting a sleep instead of the lines of the Leveinstein function. The function *__nanosleep(ns)* seemed to be perfect: however, we discovered that it requires *Compute Capability = 7.0* and on the lab machines we have 6.0. After some research on the internet, we realized that there is no good alternative. People suggested using the function *clock()*: actually, it's possible to let the GPU sleep for a certain number of clocks but this doesn't solve our problem. We don't know how many clocks correspond to 1 microsecond. We could try to understand this through measurements but this is not so reliable: different GPUs can have different speeds (maybe someone in the cluster are running with a higher clock speed).

Since we couldn't compare the use of GPU among the different approaches, we weren't able to consider GPU in the cost model.

4.3 Bug in Database Over Ranks

When the number of ranks is greater than 1, sometimes happens that the approach DatabaseOverRanks see more matches than the real ones. This happens because of a bug present in the sequential program. If you search for a pattern that is at the end of the file without the last character(s), the program would count it as a match anyway. It's clear why this bug is amplified if DatabaseOverRanks approach is used: the database is split and this situation could happen more often.

4.4 Assumptions and Possible Improvements

Currently, in both of the implemented parallelization strategies, the Master Rank is only used for distributing work to the other ranks. This means the minimum ranks we require to launch the program is 2, with one of them not being fully exploited. The reasons for this choice were ease of development and debugging.

With regards to GPUs, our code queries the device count at runtime to determine if it is possible to make use of CUDA kernels later on, and it includes fallback mechanisms in case there were none.

However, our implementation does not take into account the possibility of a node containing multiple GPUs. This could lead to a performance loss or sub-optimal resource usage if, for example, several MPI ranks in the same node have to take turns using a single GPU, meanwhile there are other devices available but unused.

5 Project Directory Structure

The project source code is distributed as a .zip file, inside which there is:

- README
- Makefile
- DNA files
- Source code, Headers
- Scripts directory, with short .batch files for tests, and subdirectories organized by database size, number of patterns and several ranks/threads parameter combinations that we used for our experiments.

We also provide a simple test script (compiles and runs the program with inputs). It can be run with: `bash scripts/basic_test.batch`, at the root of the directory.