Guy Kabiri
312252224

Parallel Sequence Alignment

This project is part of `Parallel Computation` course.

Intro

> In bioinformatics, a sequence alignment is a way of arranging the sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences. [1]

Sequence Alignment Evaluation

Each pair of characters generates a special character that indicates the degree of similarity between them.
The special characters are `*` (asterisk), `:` (colon), `.` (dot), and `_` (space).
The following definitions apply:

- Equal characters will produce a `*`.
- If two characters are not equal, but present in the same conservative group, they will produce a `*` sign.
- If characters of a pair are not in the same conservative group but are in a semi-conservative group, then they will produce a `.`.
- If none of the above is true, the characters will result in a `_` sign.

Equation

Since each sign is weighted, the following equation will result from comparing two sequences:

$$S = N_1 \times W_1 - N_2 \times W_2 - N_3 \times W_3 - N_4 \times W_4$$

> N_i represents the amount, and W_i represents the weight, respectively, of `*`, `:`, `.`, and `_`.

Groups

Conservative Groups			Semi-Conservative Groups		
NDEQ	NEQK	STA	SAG	ATV	CSA
MILV	QHRK	NHQK	MSGNDILV	STPA	STNK
FYW	HY	MILF	NEQHRK	NDEQHK	SNDEQK
			HFY	FVLIM	

An example of a pair-wise evaluation

```
PSEKHLQCLLQRHKGK
HSKSHLQHLLQRHKSQ
_*:.***_******.:
```

The following can be seen above:

- The 2nd pair consists of the characters `S` and `S`, they are equal, and hence result in the `*` sign.
- The 3rd pair, `E` and `K`, are not equal, but present in the conservative group `NEQK`, so the result is a `:`.
- The 4th pair, `K` and `S`, don't belong to the same conservative group, but rather the same semi-conservative group `STNK`. Therefore, they result in a `.` sign.
- The 1st pair consists of `P` and `H` without applying any of the rules defined above, so they result in the `_` sign.

The similarity of two sequences `Seq1` and `Seq2` defined as followed:

- `Seq2` is places under the Sequence Seq1 with offset `n` from the start of `Seq1`. Where `Seq2` do not allowed to pass behind the end of `Seq1`.
- The letters from `Seq1` that do not have a corresponding letter from `Seq2` are ignored.
- The Alignment Score is calculated according the pair-wise procedure described above.

Examples:

1.
```
LQRHKRTHTGEKPYEPSHLQYHERTHTGEKPYECHQCHQAFKKCSLLQRHKRTH
                     HERTHTGEKPYECHQCRTAFKKCSLLQRHK
                     ****************: ************
```

> Weights: 1.5 2.6 0.3 0.2
> Offset: 21
> Score: 39.2

1.
```
ELMVRTNMYTONEWVFNVJERVMKLWEMVKL
MSKDVMSDLKWEV
: .:: :  :* .
```

> Weights: 5 4 3 2
> Offset: 3
> Score: -31

Mutation

For a given Sequence `S` we define a Mutant Sequence $MS(n)$ which is received by substitution of one or more characters by other character defined by Substitution Rules:

- The original character is allowed to be substituted by another character if there is no conservative group that contains both characters.
  For example:
  - `N` is not allowed to be substituted by `H` because both characterss present in conservative group `NHQK`.
  - `N` may be substituted by `W` because there is now conservative group that contains both `N` and `W`.
- It is not mandatory to substitute all instances of some characters by same substitution character, for example the sequence `PSHLSPSQ` has Mutant Sequence `PFHLSPLQ`.

Project Definition

In the given assignment, two sequences `Seq1`, `Seq2`, and a set of weights is provided.
A mutation of the sequences `Seq2` and it's offset is need to be found, which produce the `MAX` or `MIN` score (will be given as an input as well).

Solution

Initially, a basic iterative solution was implemented. By iterating over the offsets and then for each pair of letters in the offset, the problem can be solved sequentially.
Comparing each pair of letters to determine whether they are equal or fall into one of the conservative or semi-conservative groups, then finding their best substitutions (if possible).
Hence, save any better substitution found for a pair than the previous one.
The main objective of this project is to parallelize the CPU and GPU simultaneously, taking advantage of their maximum abilities.
While the CPU tasks will be parallelized with `OpenMP`, the GPU tasks will be parallelized with `CUDA`.
It is first necessary to examine what can be parallelized in this problem, that is, what tasks are being performed independently of one another.
The program will run on two machines at the same time, which will require using `MPI` to send data between them.

CPU Implementation

Having written the sequential solution, I realized it would be time-wasting to check whether each pair of letters belongs to a conservative or semi-conservative group several times during the run.
Despite the fact that iterations over the groups are non-linear ($O(1)$) (since the number of groups and letters in each group is constant), the groups are given ahead of time, so each evaluation of two letters can be done before the program is run, saving significant time.

Consequently, I created a hashtable of 26 letters and one `-` character (27 X 27). Although each pair is still evaluated in $O(1)$, this method is much faster than the previous one.
Additionally, `OpenMP` can be used for filling this hashtable, since each cell in the table is calculated independently.
The hashtable (spaces were used instead of `_`) is as follows:

```
   A B C D E F G H I J K L M N O P Q R S T U V W X Y Z -
   _____________________________________________________
A |*   .       .                 .     : :   .
B |  *
C |.   *                               .
D |      * :   . .     .     :     :   .
E |      : *     .     :     :     : . .
F |          *   . :     : :                 . :   :
G |.     .     *             .         .
H |      . . .   *     :     :     : :             :
I |          :     *     : :                 :
J |                  *
K |      . :     :     *     :     : : . .
L |          :     :     * :                 :
M |          :     :     : *                 :
N |      : :   . :     :     *     : . . .
O |                            *
P |.                             *     . .
Q |      : :     :     :     :     * : .
R |        .     :     :     .     : *
S |:   . . .   .       .     .   . .   * :
T |:                   .     .   .     : *   .
U |                                        *
V |.         .     :     : :             .   *
W |          :                                 *   :
X |                                              *
Y |          :   :                             :   *
Z |                                                  *
- |                                                    *
```

It is now necessary to implement a parallel solution. As the project will run simultaneously on two machines, each should handle half of the tasks.
A single machine should be able to download the input data and write the output data to the file, as specified in the project. The data should be sent between machines using MPI before beginning the search algorithm.
Since each offset and pair of letters within each offset are independent, parallelizing with the CPU can be accomplished in two ways: either parallelize offsets between the two sequences and iterate sequentially over the pairs at offsets, or parallelize offsets and parallel the pairs at offsets.
By taking the second method and parallelizing the pairs in each offset it will be necessary later to somehow sum the pairs score, or to use mechanism such as `critical blocks` to find the total offset's score. Therefore, the first method appears to be more efficient. Using MPI, one can easily determine the number of processes, so after passing data between processes, one can figure out how many offsets in total will be accomplished. As each process has its own ID, it can determine which specific offsets it will handle (taking into account when dividing the number of offsets unevenly among the number of machines).
Additionally, the machine provided has quad-core CPUs. Creating more threads than there are cores will not improve performance, and may even result in slower run times because the CPU will have to switch between them.

GPU Implementation

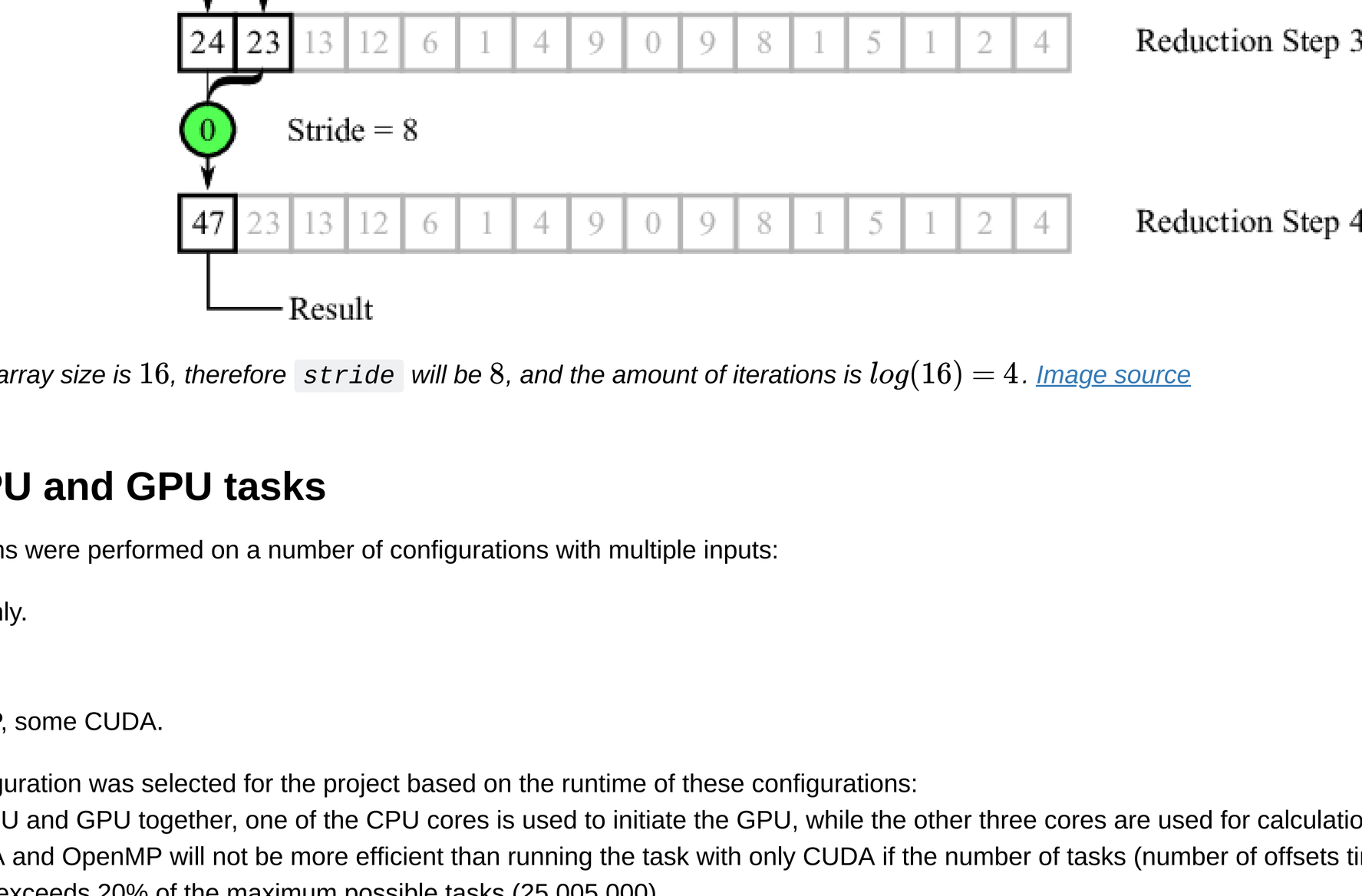
In the beginning, an implementation similar to that on the CPU was performed. The number of threads created was equal to how many offsets the GPU has to handle.
On second thought, that could lead to a failure to utilize all of the GPU's resources, when, for example, there are 3 offsets with each 1,000 characters.
The GPU will only allocate three threads, although a higher number could have been allocated.
CUDA provides a maximum of 1,024 threads per block, and 65,535 blocks (in each dimension of the grid), which results in a maximum of 67,107,840 threads per block (in one dimension block case).
The project limitation is 10,000 letters for `Seq1` and 5,000 letters for `Seq2`, which adds up to 25,000,000 pairs of letters.
The pairs of letters and offsets are independent of each other as discussed above.
Allocating a thread for each offset and letter would be a much better idea.
In CUDA, threads are structured into blocks, with each thread having a unique `block-id` and `thread-id` that can be used to determine which offset and pair in that offset it should handle. Now, each thread will handle a specific pair of letters at a specific offset.
Once the threads have completed evaluating the letters, the program has an array of mutations for each pair of letters and the original score of the original letters.
In order to sum up the array and determine which mutation is optimal, a reduction is required.
A reduction of pairs in each offset is necessary, in order to sum the offset's score and the optimal offset's mutation.
After that, a second reduction is needed to determine which offset has the best mutation. Instead of linear iteration over the array, the reduction could be implemented in parallel.
While investigating the parallel reduce algorithm, I realized that the mutations for a given offset will often end up in different thread blocks when the given input has a letter sequence that exceeds 1,024 letters.
Because CUDA does not support over-grid thread-synchronization, but only per block, it will be very difficult to implement the reduction algorithm.
Several ways of handling this situation are suggested over the internet, such as using `counter lock`, which acts like a barrier, or CUDA's `cooperative-groups`, which allows threads to synchronize over the whole group.
A different solution had to be found due to time constraints. Finally, it was decided to generate the number of blocks as the number of offsets, so that if there are more than 1,024 pairs of letters in each offset, some threads will have to calculate a mutation up to 5 times (since the maximum number of letters can be up to 5,000).

Parallel Reduction

Parallel reduction refers to algorithms that combine an array of elements to produce a single value.
Among the problems that can be solved by this algorithm are those involving operators that are associative and commutative.
The following are some examples:

- Sum of an array.
- Minimum/Maximum of an array.

If one has an array of n values and n threads, the reduction algorithm provides a solution of $log(n)$.
Reduce an array with n elements requires the algorithm to calculate the ceiling number of n, which is a power of 2 ($m = 2^{\lceil log(n) \rceil}$).
At the beginning of the algorithm, a $m/_2$ `stride` constant is defined.
For each iteration of the algorithm, every cell performs the reduced operation between itself (i) and $i + stide$. After each iteration, divide `stride` by 2.



As can see above, array size is 16, therefore `stride` will be 8, and the amount of iterations is $log(16) = 4$. Image source

Dividing CPU and GPU tasks

Run-time evaluations were performed on a number of configurations with multiple inputs:

- Sequentially only.
- OpenMP only.
- CUDA only.
- Some OpenMP, some CUDA.

The following configuration was selected for the project based on the runtime of these configurations:
Parallelizing the CPU and GPU together, one of the CPU cores is used to initiate the GPU, while the other three cores are used for calculations. The separation of CUDA and OpenMP will not be more efficient than running the task with only CUDA if the number of tasks (number of offsets times number of letters to evaluate) exceeds 20% of the maximum possible tasks (25,005,000).
Furthermore, if the amount of tasks is small, it would be wasteful to allocate and copy memory over the GPU.
If the tasks are smaller than 20%, only OpenMP will handle them; otherwise, just the GPU.
A further explanation follows the complexity section.

Complexity

The complexity of this solution depends on the length of both sequences.
Using $len(seq1) = n$, and $len(seq2) = m$, the amount of offset will be $n - m + 1$.
Each offset evolves the evaluation of m pairs of letters. Calculation of CPU and GPU will be done separately for simplicity.

CPU Complexity

By parallelizing the offsets, each thread will handle $\frac{n-m+1}{4}$ offsets, which has a complexity of $O(n-m)$.
In each offset, a sequential iteration over the letters is performed, which takes $O(m)$.
Having found the best mutation for each thread, all threads will be compared. There are as many threads as there are cores in the CPU, so the evaluation is linear.
Thus, the complexity of the CPU is $O((n-m) \times m) = O(nm - m^2)$.

GPU Complexity

The GPU represents each offset as a block of threads, each thread, as discussed earlier, will handle a maximum of five pairs of letters, which means that all possible mutations are evaluated in $O(1)$.
A reduction algorithm is run twice after evaluating the mutations. Initially, each block of threads will reduce its own mutations, since each offset has m pairs, it takes $O(log(m))$.
Having $n - m + 1$ offsets, the complexity of the second reduction is $O(log(n-m))$.
All these operations are performed separately, combining all of them will produce complexity $O(1) + O(log(m)) + O(log(n-m))$, resulting in $O(log(m)) + O(log(n-m))$.

Complexity and Configuration Summery

Because it is determined at runtime, and based on input, whether to allocate tasks to the CPU or GPU, the total complexity is $O(nm - m^2) + O(log(m)) + O(log(n-m))$. This project does not run both CPU and GPU, even though it is implemented to do so:
At most, we will have $offsets \times letters = (n - m + 1) \times m$.
Considering the maximum values of $n = 10,000$ and $m = 5,000$, CUDA can handle input with a complexity of $O(log(m)) + O(log(n-m))$.
Thus, the maximum number of CUDA iterations in this project is limited to $O(log(5000)) + O(log(5000)) = 25$ per thread.
With OpenMP, the complexity would be $O((n-m) \times m)$; since each thread handles a quarter of the offsets, this would result in $\frac{10,000-5,000}{4} \times 5,000 = 2,500 \times 5,000 = 6,250,000$ iterations per thread.
So, even when allocating and copying data into the GPU memory, it can handle big inputs very quickly.
While for a smaller input, allocating and copying data to the GPU would take more time than directing the fourth thread of the CPU to perform the calculations.

How To Run

The project was developed using `MPI`, `OpenMP`, and `CUDA`. Therefore, all of those library had to be installed for the project to run.
An input file with a name of `input.txt` or `input.dat`, and with the following structure has to be present in the root directory:

- The first line will contain 4 weights (decimal or non decimal) in the exact order of `W1`, `W2`, `W3`, and `W4`.
- The seconde line will contains the first sequence `Seq1` (up to 10,000 characters).
- The third line will contains the second sequence `Seq2` (up to 5,000characters).
- The last line will contain the string `maximum` or `minimum` to define the algorithm which defines the goal of the search.

The output file will results with the mutant of `Seq2` in the first line, and it's offset and score in the second line.

A machinefile (`mf`) with subnetwork IP addresses is required for this project to run on two machines at the same time.
Once the executable program is present on both machines and the file have been created on the main machine, run the following:

```
mpiexec -np 2 -machinefile mf -map-by node ./{EXECUTABLE}
```

where `{EXECUTABLE}` is the name of the executable file.

The following can be run on a single machine:

```
mpiexec -np {NUM} ./{EXECUTABLE}
```

In this case, `{EXECUTABLE}` is the name of the executable file, and `{NUM}` is the number of processes to be initiated.