Maximum Common Substructure between Molecules Parallel and Sequential approach

Dipartimento di Elettronica Informazione e Bioingegneria, Politecnico di Milano
Students : Davide Villani (10844273) Francesco Virgulti (10802921)
Tutor : Gianluca Palermo, Davide Gadioli, Gianmarco Accordi

Abstract

The project focuses on implementing the McSplit algorithm to efficiently search for the Maximum Common Substructure (MCS) between a pair of molecules.
We will explore different approaches to the problem, starting with CPU implementations of both a *recursive* and *iterative* algorithm in C++.
Their performance will be compared with the State of Art MCSplit recursive algorithm implemented in Python.
We will then introduce the GPU algorithm, and some possible but less efficient variants, implemented in CUDA Nvidia, showing the benefits and drawbacks of a parallel approach compared to the sequential CPU approaches.

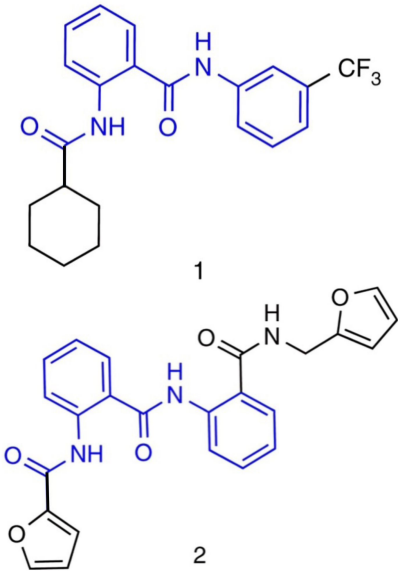

Figure 1: Example of a Max Common Substructure between two molecules, the MCS is highlighted in blue

1 Introduction

The *Max Common Substructure* between molecules plays an important role in drug-discovery simulations, providing information about compounds affinity within molecules and simplifying the computation of the *RBFE, relative binding free energy.*
This is an *NP-Complete* problem since it can be modeled as a Graph, where both nodes and edges are labeled to store information about the atoms type and the bond type, transforming the problem into a Max Common SubGraph between labeled graphs.

There are already many different approaches to solve the subgraph problem, but in this case we are going to focus on the MCSplit approach, a *Branch and Bound* recursive approach whose strength comes from the smart labelling of atoms groups, based on their type and their bond, this method is derived by RDKit, a well known library for molecule manipulation . These Labels are really useful because they give us a powerful bounding, based on the numbers of atoms included in the selected label, that efficiently close some branches in the searching tree.

2 MCSplit implementation and changes

The very first Python algorithm implements a slightly modified version of MCSplit that works for connected subgraphs and labeled nodes and edges, introducing also a new constraint in the search of the optimal subgraph, since a substructure in a molecule needs to consider the presence of rings. When we match two atoms together we also need to check if they belongs to a ring, and if they do, in order for them to be part of the optimal solution, all the rings they belong to needs to be equal in both molecules (figure 2).

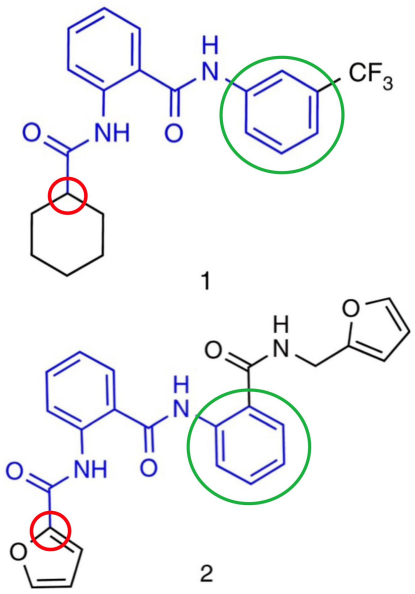

Figure 2: In red a Carbon atom that cannot be matched since the rings aren't matching, in green two matching rings

Before getting into the next section and explain how the iterative approach works, we need to talk about *LabelClasses*, how we build them and how we use them for the Branch and Bound.
A *LabelClass* is defined by five elements, a vector of integers containing the indexes of the atoms positions (of the same type) in the first molecule, similarly a vector for the atoms of the second molecule. A matrix of integers where each column represent a ring and each row the indexes of atoms contained in that ring. A string with the name of the label (the type of atom).The last attribute of the class is a boolean value called adjacent, since a *LabelClass* (other than the very first) is created starting from a pair of atoms that matched together, one from the first molecule and the other from the second, these atoms will create one new *LabelClass* for the adjacent atoms and one for the non adjacent, the boolean tells us which group these atoms are part of. These classes are useful because we use them to match atoms efficiently, by knowing their type (since atoms with the same label name are in the same LabelClass we don't need to go through all the molecule every time) and the rings that they are part of, but they are also the base for the bounding function of the searching tree:
Since the matching starts from atoms contained in the first two vectors of the *LabelClass*, every time we match a pair we are basically creating a new Branch of the searching tree and these atoms will create other *LabelClasses* until we reach empty *LabelClasses*. We can use the information contained in the length of vectors to calculate an *UpperBound* for the searching tree.
Since we can match at most as many atoms as the sum of the length of vectors of atoms with minimum length in all the *LabelClasses* generated before, we can sum to this length the length of the partial solution found until now in this branch, getting the *UpperBound*, following the Brand and Bound logic if this is lower than the current best solution found in the tree, we can close the Branch.

M is the vector containing the partial solution
G and H are the atoms vectors of each *LabelClass*

$$UpperBound = |M| + \sum \min(G, H) \qquad (1)$$

3 CPU Sequential Iterative approach

We are not going to get too deep into the *recursive* C++ implementation since it's just a translation of the already existing python algorithm, but in the next Section we will talk about it in terms of efficiency.
The *iterative* approach follows the same logic as the McSplit, but with the big difference of the introduction of the Stack, and a better usage of the heuristics to efficiently select our vertexes and labels. Firstly we need to introduce the role of the *Stack*, the McSplit implements a Branch and Bound and best way to go through the searching tree is by using a *DFS* (depth first search), since by getting deep into one Branch allows us to find a big partial solution that can help us cut most of the branches in the first levels.

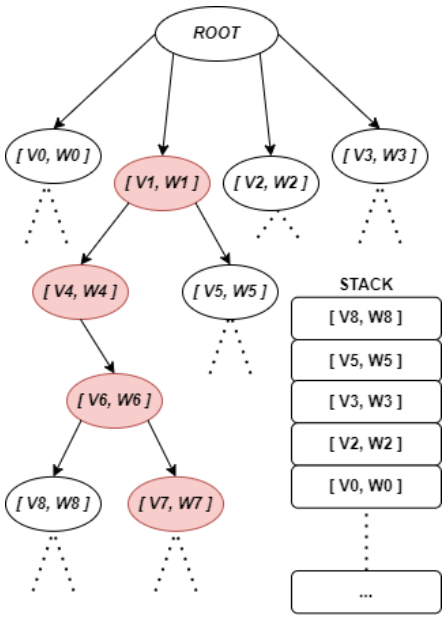

Figure 3: Example of the searching tree, in red the main branch created by the first DFS, the white nodes haven't been visited yet and stays in the stack

Using a Stack is the best option because we can save into it all the new generated list of *LabelClasses* that will be used by the next iteration to go through the branches, this data structure will also include the local solution to be able to calculate the bound.
The algorithm, as shown in the pseudocode, is divided in two functions, the MCS() function generate the initial LabelClasses and going through them choose the best vertex (line 3-4) from both molecules, in this context it's worth explaining how we used the heuristics to select vertexes. Given a LabelClass this function returns a node from both molecules that has the adjacent nodes with the highest degree(in our case has the atoms that it bonds with), this function is really powerful since it helps us to create much bigger LabelClasses that contains more nodes for both molecules, lowering the amount of branches that we create. If the two selected atoms don't match we select only the first one and find a match by going through all the combinations of the second molecule atoms (line 9 to 12).
After choosing the nodes that can be matched, we put them in the first index of the local solution (a vector containing pairs of nodes) and we call the function STACKINSERT() (lines 7 and 12) that just creates new LabelClasses based on the selected nodes and push the new labels with the local solution in the stack.
At this point we have our stack that is full of lists of LabelClasses and we call the second function SOLVE(), which is called iteratively until the stack is empty.
In this function we first get the first element from the stack(line 1), and we compute another really important heuristic, SELECTLABEL() function(line 2), that gets a list of labelClasses and returns the one with the highest amount ot atoms that were adjacent to the pair of atoms who created the list (that we will call parents) in the beginning, by doing so we can efficiently detect in the next step a large amount of atoms that are adjacent and match with the *parent* pair. After choosing the labelClass we finally compute the *Bounding* operation (line 3 to 5 and equation(1)) and all the possible matches of pairs (line 7 to 18), if we find a match, like in function MCS(), we do the same operations and we add line 13 and 14 to check if the local maximum is bigger than the best solution found so far.

Algorithm 1 CPU Iterative Algorithm

```
stack ← ∅
bestSolution ← ∅

MCS() :
initiaLabelClasses ← GENLABELS()
lc ← SELECTLABEL(initiaLabelClasses)
lcg ← lc.g
while lcg != ∅ do
    v ← SELECTVERTEX(lcg)
    localSol ← ∅
    lch ← lc.h
    while lch != ∅ do
        w ← SELECTVERTEX(lch)
        if MATCHABLE(v, w) then
            localSol ← lc.solution ∪ [v, w]
            STACKINSERT(v, w, stack)
            if |localSol| >= |bestSolution| then
                bestSolution ← localSol
            end if
        end if
        lch ← lch − w
    end while
end while
while stack ≠ ∅ do
    SOLVE()
end while

SOLVE() :
labelClasses ← POPBACK(stack)
lc ← SELECTLABEL(labelClasses)
bound ← |lc.solution| + CALCBOUND(lc)
if bound <= bestSolution then
    return
end if
for v ∈ lc.g do
    localSol ← ∅
    for w ∈ lc.h do
        if MATCHABLE(v, wertex) then
            localSol ← lc.solution ∪ [v, w]
            STACKINSERT(v, w, stack)
            if |localSol| >= |bestSolution| then
                bestSolution ← localSol
            end if
        end if
    end for
end for
```

4 GPU Parallel approach

In this section we are going to talk about the GPU implementation and some possible parallel approaches, their pro and their downsides.
Firstly we need to keep in mind that we are dealing with a Branch and Bound and as said before, the power of this algorithm comes from the fact that by operating a DFS we can easily compute a big Bound that will cut most of the branches that we create during our way down in the searching tree, this can create some problems in a parallel approach. Before talking about the final approach we are will go through a couple of approaches we discarded:
*1.)*If we try to parallelize from the beginning by choosing the first level created by the root, we can easily end up computing some paths that aren't really necessary and that will lead us to use too much memory without actually improving the efficiency, but lowering it since we just compute useless branches that could be cut and with more space that needs to be allocated.
*2.)*Another possible approach could be a total change of perspective and instead of applying a DFS we instead go for a Breadth-first search (BFS), the algorithm is basically the same as the one presented in Section 2 but by using a Queue we don't pop the last inserted element but the first, by doing so we can go through the searching tree level by level, but this approach is really inefficient for two main reasons: the first is that we are not really using the full power of the Branch and Bound because the computation of the bound is almost useless, since every branch is always on the same level the bound is not going to be really different from one node to the other, and we can't efficiently cut the useless branches; the second reason is about memory, this is a huge downside in a parallel approach, since all the memory needs to be pre-allocated before the computation starts, and using BFS means that the size of the Queue is not going to be at most the length of the shortest molecule (thing that happen in the DFS, since the lowest level is as deep as the length of the solution), but the size is much larger depending on how many labelClasses each pair creates (by using some chemical limitations each pair can at most create 4 useful list of

LabelClasses) and this is a downside for both computing the pre-allocation and the time all the allocations take.
After going through a couple of possible solutions (that we are not going to consider in the Test Result section since their performances were too slow) we can get into our final GPU CUDA implementation.
As we said before, the optimal approach is to use a DFS, in this case we start our first thread that works in the same way of the CPU Iterative, but in this case instead of going through all of them, when the number of LabelClasses instances is more than $B_N T_N$ (Thread Blocks Number and Thread Number per Block), we call the KERNEL() function (line 21 22).
In this approach we are going to use two data structures to save the instances of the new problems, one is the stack, as we saw in the previous algorithm, but we now add a ThreadPool array were we put every instance that we will parallelize, we are going to fill both data structures in a way that in the stack we will only have the instances to proceed in the DFS, but we will also add instances of problems that don't reach the requirement to be parallelize, but we are going to discuss about this in the next lines.
The initial part of the Algorithm follows the same logic as before to create the first level, but in this case in the STACKINSERT() we add another condition, if the level is deep enough (compared to the possible solution length) it means that the Thread would just do little to no work, so it's actually useless to put them in the ThreadPool and parallelize them, so we introduce a *max depth* after which we stop putting the new problem instances in the ThreadPool (figure 4).

Before getting to the kernel we need to go through some stack filters that will help us avoid allocating useless space:
first of all the function SOLVE() works in the same way as before, with the little modification said before, then with the FILTERSTACK() (line 19) function we basically select all the instances that respect the Bounding criteria, this way we bound before calling the solve function in the CUDA threads, and avoiding useless computing by Threads that would just compute an *if* operation and return.

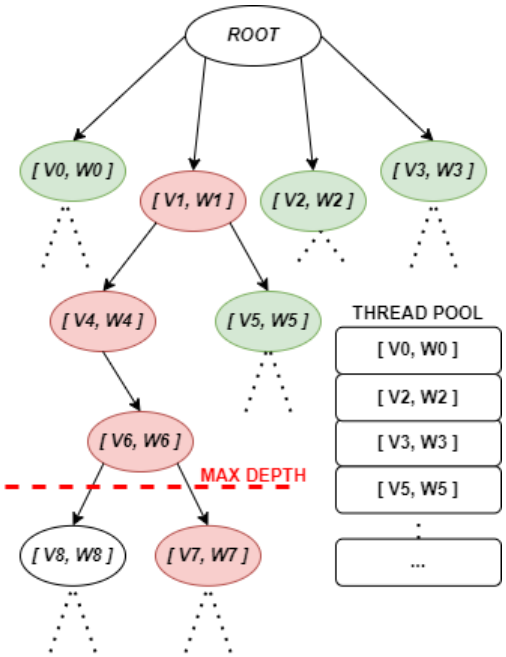

Figure 4: Example on how the searching tree looks like during the execution, in red the first branch, in green all the nodes we put in the ThreadPool, and the red line represents the max depth

Before talking about the function in line 20, we need to better understand what the list of labelclasses contains:
As we said in the previous sections, each LabelClass is created based on the last pair of atoms we put in the local solution, and we create each label based on the atoms of each type that are adjacent or not to the atoms in the pair. But we have a special type of LabelClass that is used to understand if an atom is in a ring (i.e. let's say we have a Carbon atom that is in a ring, we will create an atom of type CR and not C and so on for every other atom in at least one ring) and these LabelClasses are usually the longest, since for the other atoms that are not in a ring, we can easily bound the length of the vectors in the LabelClass based on their chemical properties, an atom of Carbon (C) can at most be adjacent to other 4 atoms, so using these chemical limitations we can actually im-

prove our spacial efficiency. Since LabelClasses are generated in a random order we would need to allocate for each of them the same amount of space to avoid *segmentation fault* and that's where the SORTSLABELS() function comes in handy:
The SORTLABELS() function (line 20) is a fundamental step to improve the spacial complexity, the function works on the stack to sort from the biggest to the shortest each list of LabelClasses, this way, before putting each instance of the problem in the ThreadPool, we sort them so that the rings LabelClasses goes always to the top of the list, in this way we can allocate more space only for the first two rows, and allocate the remaining places based on chemical bonds restrictions as said previously.
At this point we optimized all the space and we are ready to check if the ThreadPool size is as big as all the threads we are going to run, if so, we proceed to call a KERNEL() function that just parallelize the call on the SOLVE() function, and we synchronize them.
This approach doesn't need any atomic operations between threads, each one of them just use as a bounding the upper bound found before the kernel call, this way we might not be able to perfectly apply the cut of the branches through different threads but we get much more efficiency by avoiding atomics operation that would need to be synchronized, after each thread is done we just consider the localSolution that are bigger than the previous bestSolution and we compare them to find the new best.
It's worth mentioning that a parallel approach needs a much bigger initialization work to set up the GPU shared memory, and the optimal solution, thanks to the heuristics, for small molecules, is going to be found almost always in the first branch, for this reasons the parallel approach is not recommended for small amount of computation, in our case small pair of molecules, a CPU sequential approach is always preferable in this cases, but we are going to get deeper into it in the next section.

Algorithm 2 GPU CUDA Algorithm

$stack \leftarrow \emptyset$
$threadPool \leftarrow \emptyset$
$bestSolution \leftarrow \emptyset$

MCS() :

1: $initiaLabelClasses \leftarrow GENLABELS()$
2: $lc \leftarrow SELECTLABEL(initiaLabelClasses)$
3: $lcg \leftarrow lc.g$
4: **while** lcg != $\emptyset$ **do**
5: $v \leftarrow SELECTVERTEX(lcg)$
6: $localSol \leftarrow \emptyset$
7: $lch \leftarrow lc.h$
8: **while** lch != $\emptyset$ **do**
9: $w \leftarrow SELECTVERTEX(lch)$
10: **if** $MATCHABLE(v, w)$ **then**
11: $localSol \leftarrow lc.solution \cup [v, w]$
12: STACKINSERT($v, w, stack$)
13: **if** $|localSol| >= |bestSolution|$ **then**
14: $bestSolution \leftarrow localSol$
15: **end if**
16: **end if**
17: $lch \leftarrow lch - w$
18: **end while**
19: **end while**
20: **while** $stack \neq \emptyset$ **do**
21: SOLVE()
22: FILTERSTACK($stack$)
23: SORTLABELS($stack$)
24: **if** $|threadPool| >= B_N T_N$ **then**
25: KERNEL()
26: **end if**
27: **end while**
28: **if** $|threadPool| > 0$ **then**
29: KERNEL()
30: **end if**

KERNEL() :

1: mem copy from CPU to GPU
2: SOLVE<<< B_N, T_N >>>()
3: cudaDeviceSynchronized()

global $SOLVE()$:

1: the same function as Algorithm 1 SOLVE()

5 Test Result and Comparison

In this section we will talk about the results of the tests, comparing *five* different algorithms:
The *Python* implementation of the MCSplit algorithm given by a previous student, our three different approaches, the *CPU Recursive MCSplit* which is just a C++ implementation of the Python code, the *CPU Iterative C++* and the *CUDA* approach, which we discussed in the last sections, the last algorithm is the one implemented by *RDKit* .

5.1 Data Set

The data set that we used to run the tests contains around 3000 pairs of medium-small sized molecules, from 30 to 60 atoms (H atoms are not counted), that are paired together to make sure they have non trivial MCS (such as one-atom-only MCS or empty MCS).
It's also important to note that the data set contains molecules in the SMILE form, so before applying the algorithm we need to translate them into a set of vectors and adjency matrixes. The translation part is done using RDKit built-in functions and won't be considered in the next section in the execution time of the MCS search, since it's a constant time that every algorithm needs to go through, but the creation of label classes will be take in account.
For the CPU tests we run on *AMD EPYC 7302 (20) @ 2.999GHz* while for the CUDA algorithm we run everything on *NVIDIA A100 SXM4 40GB*

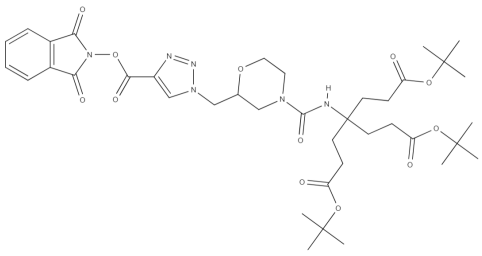

Figure 5: Visual Example of a molecule contained in the data-set

5.2 Plot Results

We started by running our data-set on the CPU algorithms first, in order to compare their MCS effectiveness with the *Python implementation* that was already proved to be correct.
On the y-axis of the plot we have execution time in *seconds* and on the x-axis the *number of molecules pairs* that were tested (fig.6).
We can clearly see how after less than 500 instances the python implementation slows down and its execution time rise exponentially compared to the C++ approaches, taking almost 1200 seconds to run all the pairs, which compared to the *recursive* approach (the python algorithm implemented in C++) is almost **50x** slower and more than **170x** slower than the *iterative*.

Looking into our C++ implementations we can

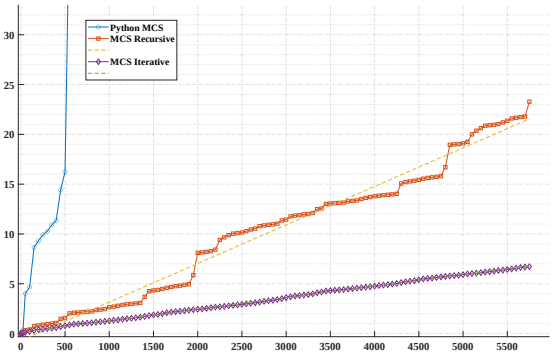

Figure 6: Comparison between CPU approaches

clearly see how our new *iterative* and heuristic approach is almost **4x** faster than the *recursive* one, this is mostly due to a better memory management since in the *iterative* approach we can understand how deep we can go into the search tree and we can avoid the stack to grow uncontrollably on it, differently from the *recursive* approach.
This difference is also shown in the general evolution of the time function, in the *recursive* approach we can see some "jumps" in time, while the *iterative* one is perfectly linear following the linear regres-

sion line shown in fig.6, once again this more stable time is achieved thanks to a better memory management avoiding the recursion to get too deep in useless branches of the tree.
The iterative approach was the key to build the parallel algorithm, since the memory management is fundamental for a successful CUDA approach we choose this approach as the skeleton of the new algorithm whose performances are considered in fig.7.

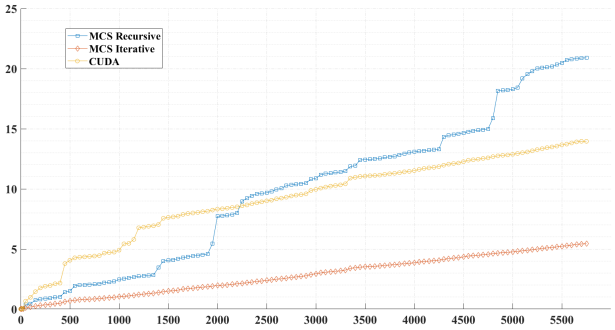

Figure 7: Comparison between C++ CPU approaches and CUDA parallel approach

Not surprisingly the *Parallel* approach isn't the fastest, that's because of the overhead weight, since we need to initialize all threads and synchronize them this makes us lose most of the time.
In the first instances the *parallel* approach seems to be much slower due to all the mallocs and the memory operations, but we can also notice how it fully recover and after 2000 instances take over the recursive approach, but still slower than the iterative one.
One interesting fact that we can see from fig.7 plot is that while the sequential algorithms follow almost a perfect linear grow, in the parallel one we have a slow start that fully recovers and stabilize in a much flatter grow that gets closer to the iterative time. To make it more clear and see all of the 4 algorithms together we plotted everything on a semilog scale to understand how many order of magnitude of difference we have between *python* and the cuda/c++ approaches (fig.8). Something important to note is that since the molecules that we considered are medium-small sized, we couldn't really go over *1Block* containing

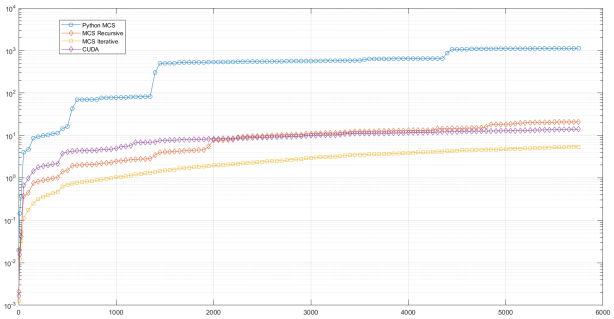

Figure 8: Semilog scale of the first 4 algorithms

32Threads, because for higher numbers of threads the algorithm wouldn't even call the kernel function but everything would just be solved iteratively.
We then decided to compare our fastest approach with RDKit fastest MCS algorithm (fig.9) but we didn't find any substantial differences, our algorithm appears to get a little faster around 3500 instances altought for almost all the time they seem to be really close to each other without no deep difference between in the execution times, the RDKit ended up being just 1 second slower then our algorithm on a 6000 instances execution.

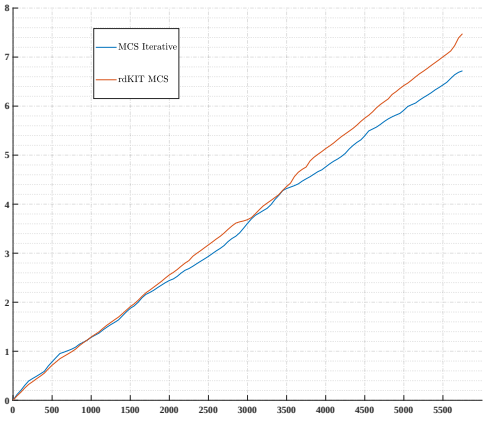

Figure 9: C++ Iterative execution time compared to RDKit algorithm

6 Conclusion

In this work we first started from the recursive MCSplit implementation, modifying it in a way that we could efficiently find all the right ring matches between molecules, differently from the standard MCSplit algorithm. We then proceeded to study a new solution that could give us a better power over managing the memory, so we created a new iterative approach, that we optimized using some heuristics to make it faster and more stable than every other approach.
Once we had this stable approach we used it to build our actual *Parallel* approach that only modifies the management of the stack to optimize it, lowering all the overhead time caused by cudaMallocs.
Unfortunately we could only test the parallel approach on small molecules, this way we couldn't really test the full potential of this CUDA approach that is restricted by the small size of the searching tree, but further data sets could lead to better tests and results, pushing the parallelism more and more and eventually overtaking the Iterative approach. Even though it shows to be slower than the CPU approach, we still managed to get better performances compared to the Recursive one that represent the MCSplit algorithm.
Testing the CUDA approach with bigger molecules will be our next goal in further research works.

References

[1] Ciaran McCreesh, Patrick Prosser, James Trimble
A Partitioning Algorithm for Maximum Common Subgraph Problems. In the Twenty-Sixth International Joint Conference on Artificial Intelligence

[2] Stefano Quer
A Parallel Many-core CUDA-based Graph Labeling Computation. In the International Conference on Software and Data Technologies 2020