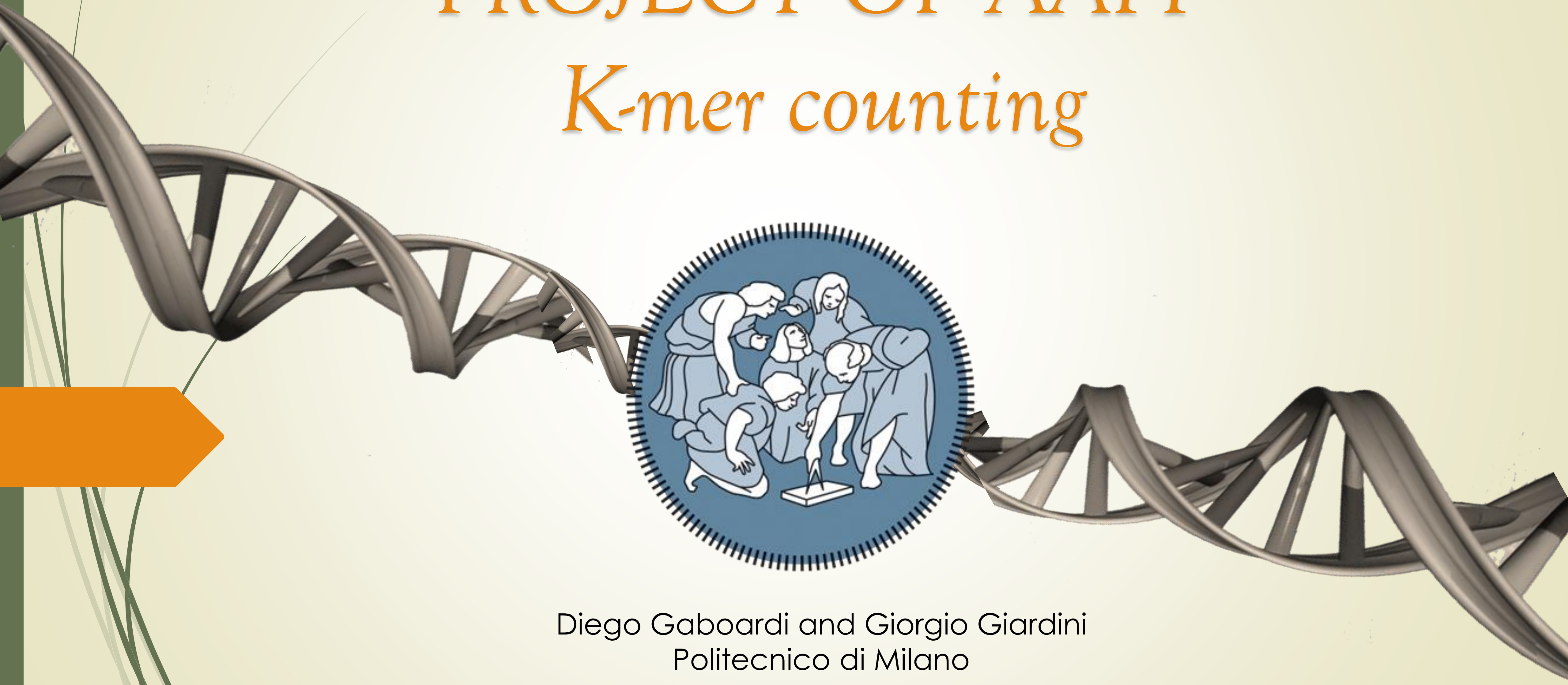
PROJECT OF AAPP
K-mer counting



Diego Gaboardi and Giorgio Giardini
Politecnico di Milano

PROBLEM

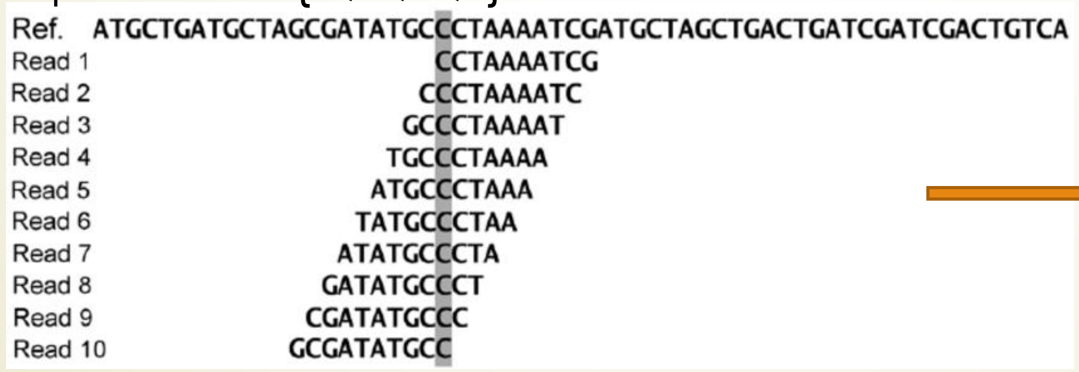
Counting the number of occurrences of every k-mer in a long string

K-mer → substring of length *k*

USAGE

genome assembly

alphabet = {A,C,G,T}



```
Ref.    ATGCTGATGCTAGCGATATGCCCTAAAATCGATGCTAGCTGACTGATCGATCGACTGTCA
Read 1                       CCTAAAATCG
Read 2                      CCCTAAAATC
Read 3                     GCCCTAAAAT
Read 4                    TGCCCTAAAA
Read 5                   ATGCCCTAAA
Read 6                  TATGCCCTAA
Read 7                 ATATGCCCTA
Read 8                GATATGCCCT
Read 9               CGATATGCCC
Read 10             GCGATATGCC
```

statistics

knowledge

STANDARD IMPLEMENTATION

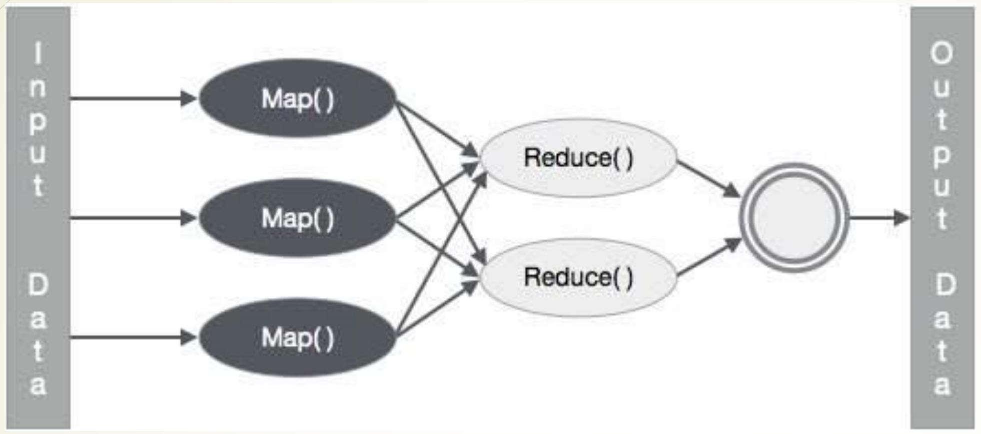
Hash table
Key: substring (k-mers)
Value: counter

Key	CCTA	ACGT	TTGC	ATCC	AATG	GAAC	TCCG	AGTC	CCAG	TGAA
Value	567	234	456	567	234	567	123	673	578	198

SPATIAL DIMENSION (in bit)
$A^k * log_2$(b)
Where A is the alphabet size
k the substrings lenght
b the maximum value of each counter

TEMPORAL DIMENTION
|S|* table_access_time
Where S is the input string

ON MAP REDUCE



Every node process
a substring

Straightforward implementation -> map operation emits a series of couples < k-mer , 1 > for each node -> large overhead

Efficient implementation -> map operation includes a counter and emits a series of couples < k-mer , count >

$\downarrow$ How to obtain efficiently?

Jellyfish parallel algorithm on each node

JELLYFISH

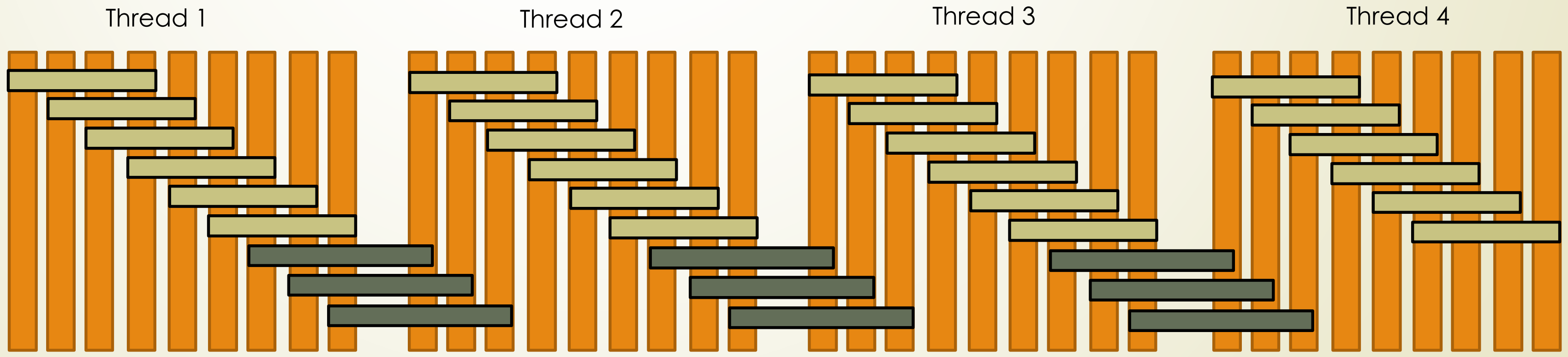
- **GOALS**:
  - Fast and multithreaded
  - Memory efficient

- **SOLUTION:**
  - Lock free Hash-table with CAS assembly instruction
  - Merging intermediate hash tables
  - Reduced memory usage of hash entry
  - Space-efficient encoding of keys

PARALLELISM: SIMD

- Single Instruction
  - Same code with a loop cycle
  - Each iteration of the loop is executed by one of the threads in the team
- Multiple Data
  - Partitioning of the string between the threads
  - Each thread process a contiguous sequence of characters

⚠ **Border issue:** overlapping at border



OUR IMPLEMENTATION

Language: C++ & open mp

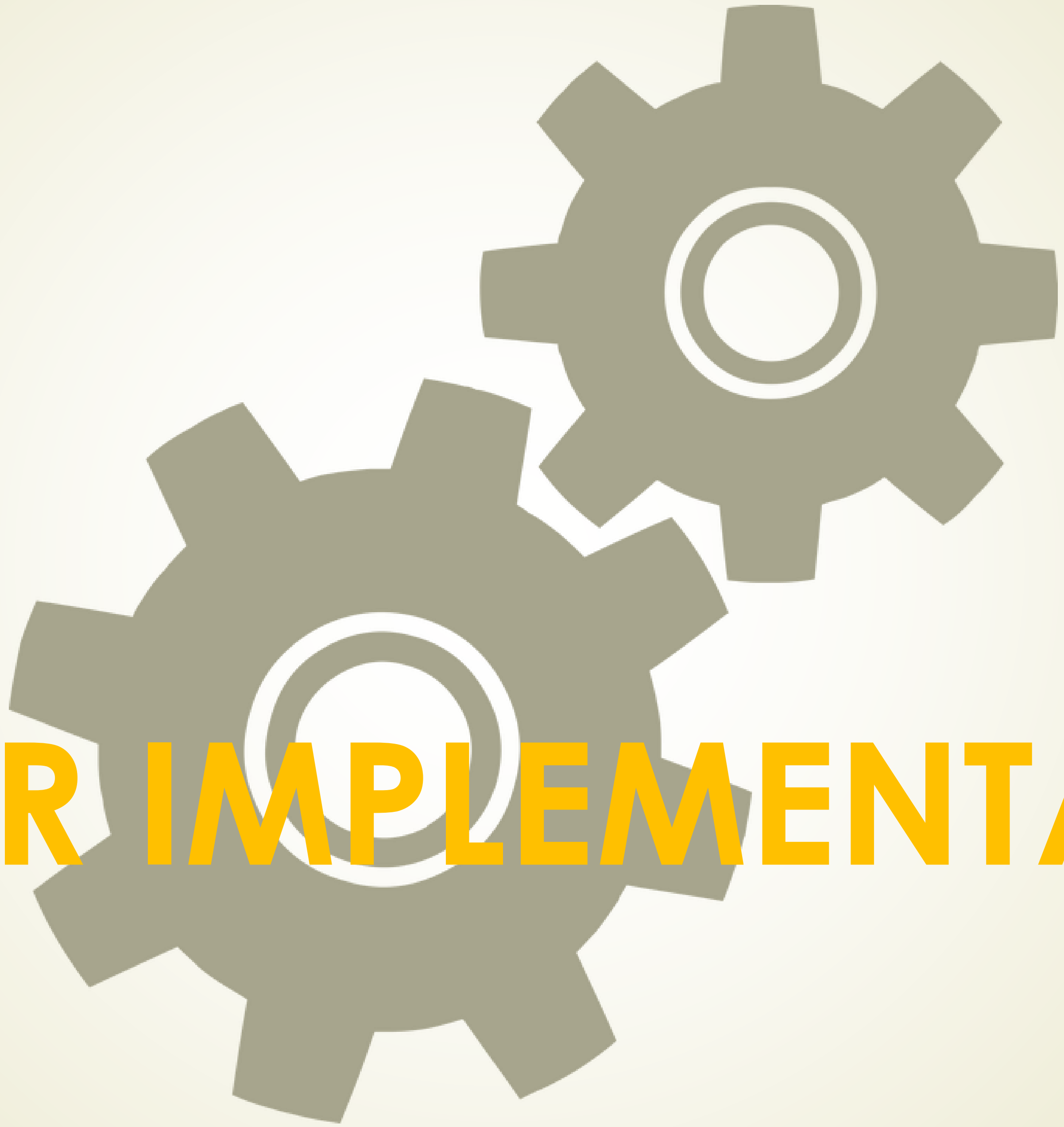
- Support shared memory multiprocessing
- Easy to use simply adding some commands to standard C++ code

GIT and GIT HUB

- For working in group

WORK PLAN

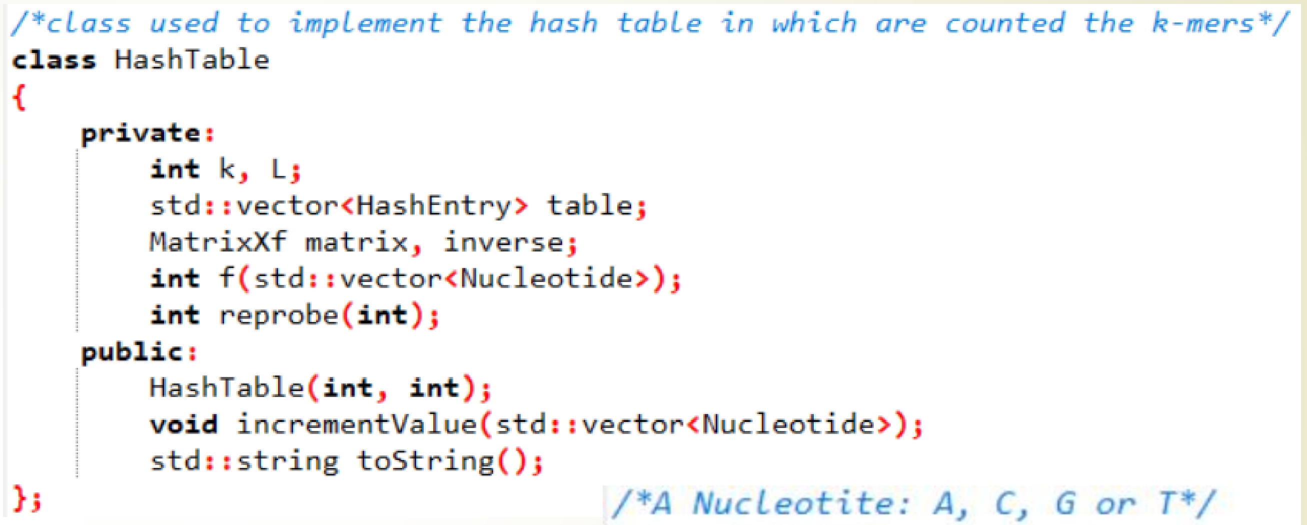
- Parser of the DNA sequence
- Serial version of the hash table
- Multithreaded hash table with locks
- Multithreaded hash table without locks
- Make the program memory efficient
- Evaluation of the algorithm



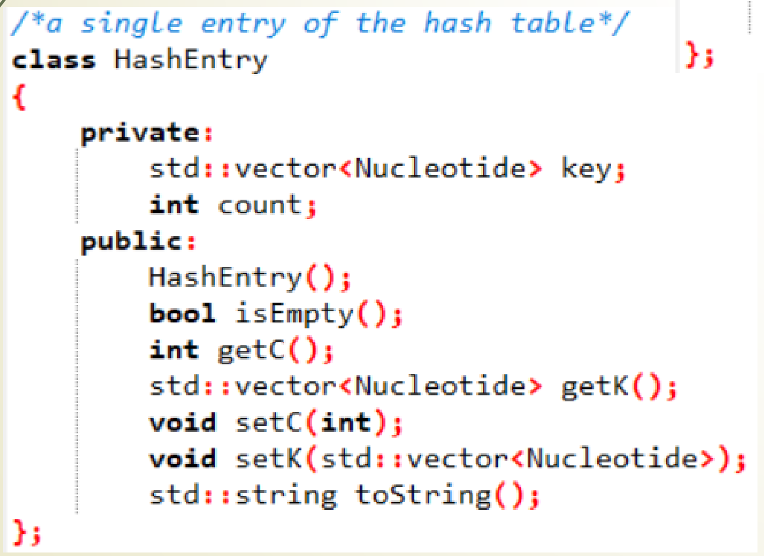
OUR IMPLEMENTATION

SERIAL IMPLEMENTATION

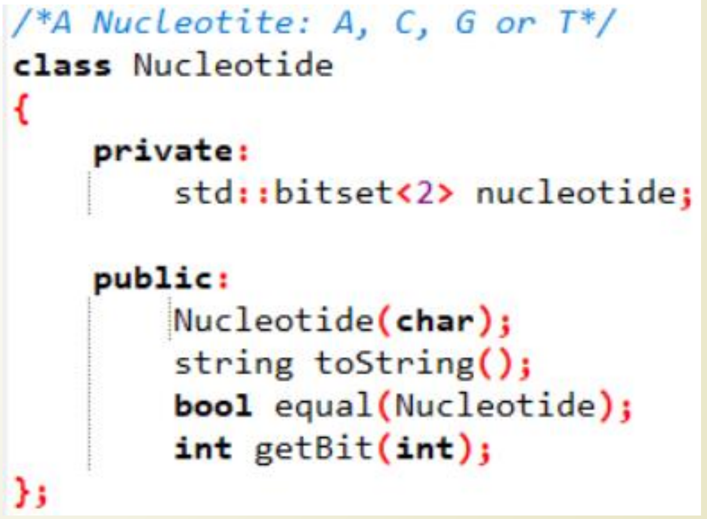
- Class model
  - HashTable
  - HashEntry
  - Structures



```
/*class used to implement the hash table in which are counted the k-mers*/
class HashTable
{
    private:
        int k, L;
        std::vector<HashEntry> table;
        MatrixXf matrix, inverse;
        int f(std::vector<Nucleotide>);
        int reprobe(int);
    public:
        HashTable(int, int);
        void incrementValue(std::vector<Nucleotide>);
        std::string toString();
};
```



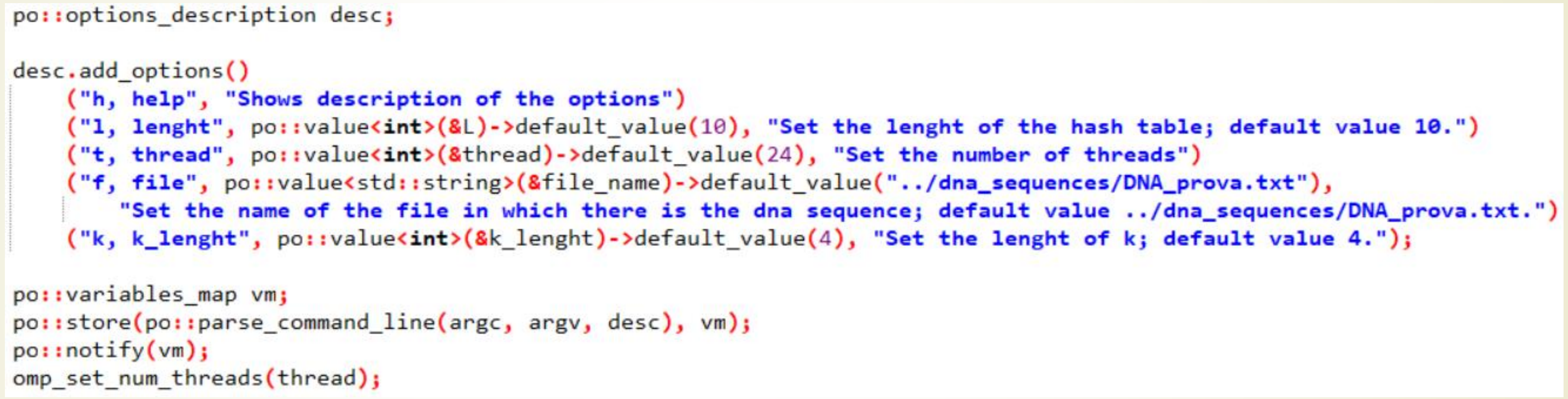
```
/*a single entry of the hash table*/
class HashEntry
{
    private:
        std::vector<Nucleotide> key;
        int count;
    public:
        HashEntry();
        bool isEmpty();
        int getC();
        std::vector<Nucleotide> getK();
        void setC(int);
        void setK(std::vector<Nucleotide>);
        std::string toString();
};
```



```
/*A Nucleotite: A, C, G or T*/
class Nucleotide
{
    private:
        std::bitset<2> nucleotide;

    public:
        Nucleotide(char);
        string toString();
        bool equal(Nucleotide);
        int getBit(int);
};
```

SERIAL IMPLEMENTATION

- Program option

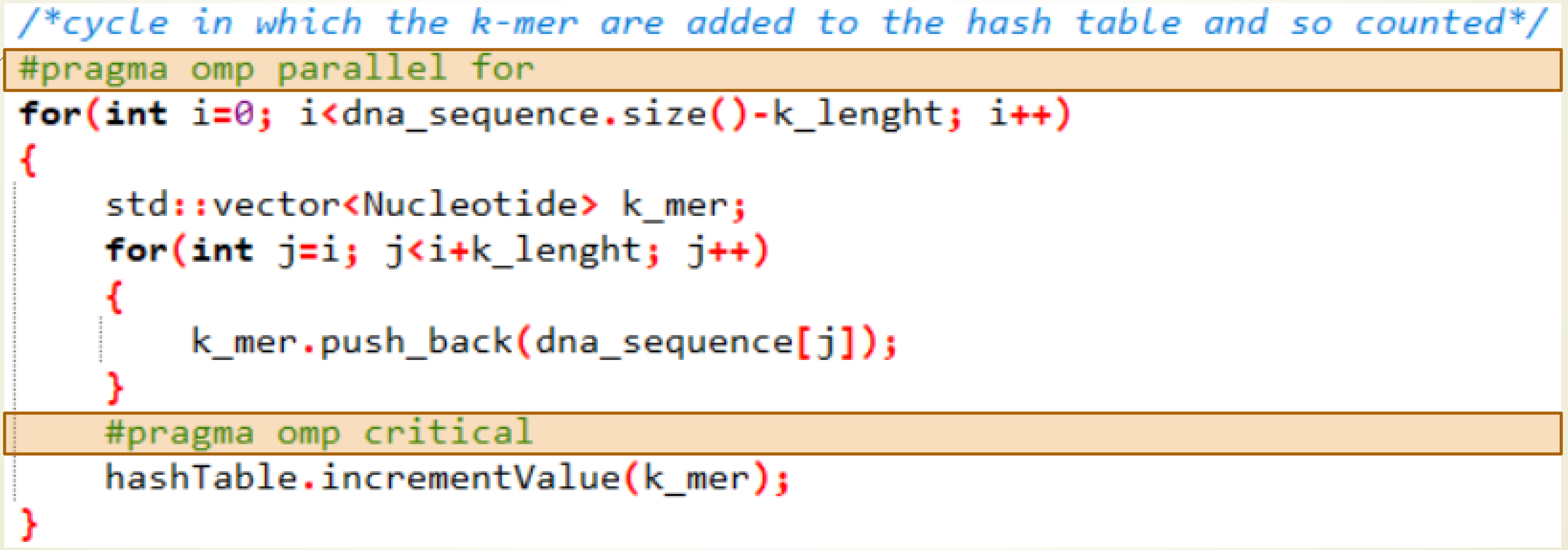


```
po::options_description desc;

desc.add_options()
    ("h, help", "Shows description of the options")
    ("l, lenght", po::value<int>(&L)->default_value(10), "Set the lenght of the hash table; default value 10.")
    ("t, thread", po::value<int>(&thread)->default_value(24), "Set the number of threads")
    ("f, file", po::value<std::string>(&file_name)->default_value("../dna_sequences/DNA_prova.txt"),
        "Set the name of the file in which there is the dna sequence; default value ../dna_sequences/DNA_prova.txt.")
    ("k, k_lenght", po::value<int>(&k_lenght)->default_value(4), "Set the lenght of k; default value 4.");

po::variables_map vm;
po::store(po::parse_command_line(argc, argv, desc), vm);
po::notify(vm);
omp_set_num_threads(thread);
```

Program options let user set the following parameters at execution time:
l– hash table lenght
t – number of thread
f – source file
k – k-mer lenght

LOCK MULTITHREAD IMPLEMENTATION

OUR CHOICE: use of a critical call of increment function, in an openmp parallel cycle



```
/*cycle in which the k-mer are added to the hash table and so counted*/
#pragma omp parallel for
for(int i=0; i<dna_sequence.size()-k_lenght; i++)
{
    std::vector<Nucleotide> k_mer;
    for(int j=i; j<i+k_lenght; j++)
    {
        k_mer.push_back(dna_sequence[j]);
    }
    #pragma omp critical
    hashTable.incrementValue(k_mer);
}
```

OPTIMIZATIONS

LOCK-FREE HASH TABLE

CAS instruction:

- Reads a memory location
- Compare the read value with the second parameter of the instruction
- If they are equal write the memory with the 3rd parameter
- Return the previously held value

Possibility to detect simultaneous access to shared resources

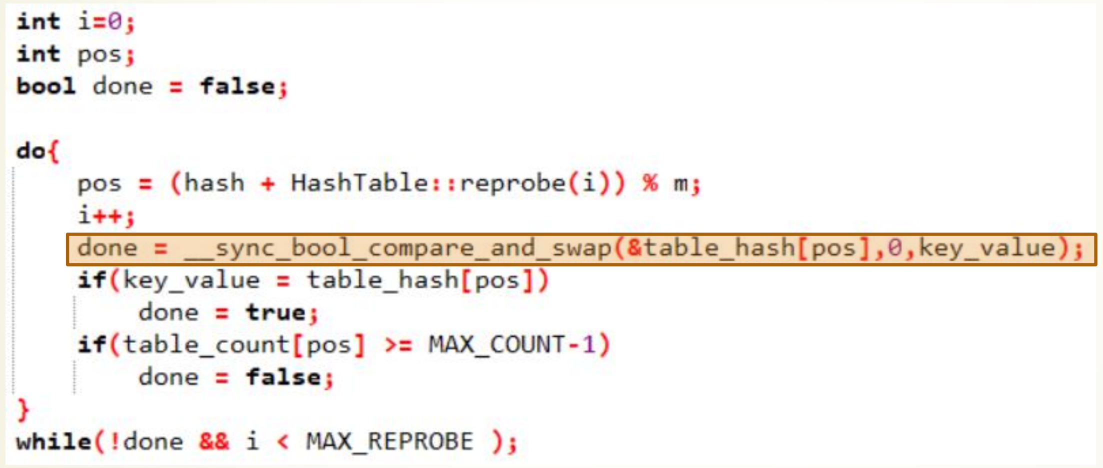
USED TO:

- Finds the location in the hash table
- Increments the value associated with the key

✓ IMPLEMENTATION

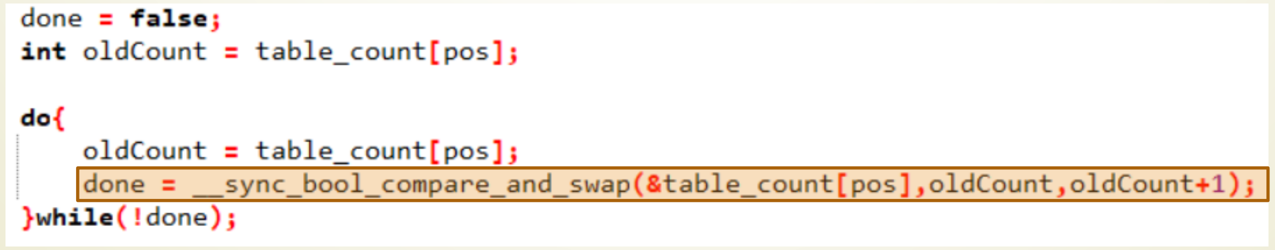
OUR CHOICE: use the GCC function __sync_bool_compare_and_swap()

- Find the location in the hash table



```
int i=0;
int pos;
bool done = false;

do{
    pos = (hash + HashTable::reprobe(i)) % m;
    i++;
    done = __sync_bool_compare_and_swap(&table_hash[pos],0,key_value);
    if(key_value = table_hash[pos])
        done = true;
    if(table_count[pos] >= MAX_COUNT-1)
        done = false;
}
while(!done && i < MAX_REPROBE );
```

- Increments the value associated with the key



```
done = false;
int oldCount = table_count[pos];

do{
    oldCount = table_count[pos];
    done = __sync_bool_compare_and_swap(&table_count[pos],oldCount,oldCount+1);
}while(!done);
```

MERGING INTERMEDIATE HASH TABLE

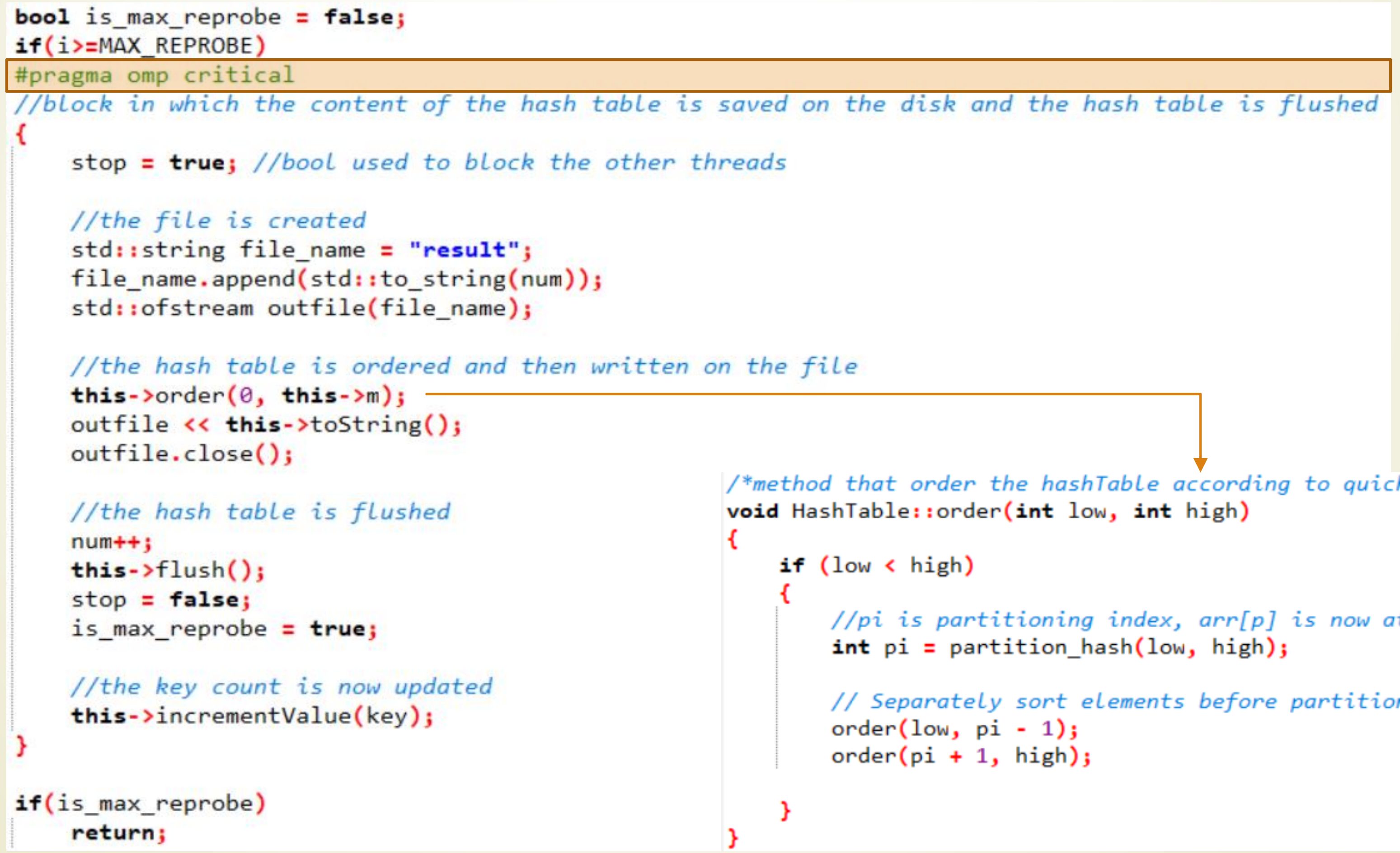
- Once computed the hash table is written to disk in order

- In this way it is possible to:
  - Query quickly using binary search
  - Merge two hash table

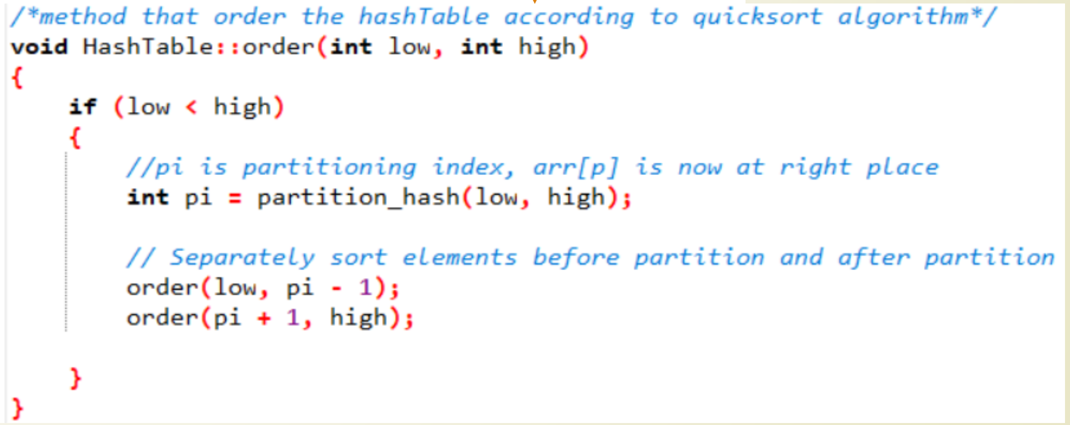
- If there isn't enouth memory for the entire computation:
  - Intermediate results are saved to disk
  - The hash table is cleared and we begin counting afresh
  - At the end intermediate results are merged

IMPLEMENTATION

OUR CHOICE: use an openmp critical section in order to avoid writing conflicts
order the table efficently using quicksort algorithm



```
bool is_max_reprobe = false;
if(i>=MAX_REPROBE)
#pragma omp critical
//block in which the content of the hash table is saved on the disk and the hash table is flushed
{
    stop = true; //bool used to block the other threads

    //the file is created
    std::string file_name = "result";
    file_name.append(std::to_string(num));
    std::ofstream outfile(file_name);

    //the hash table is ordered and then written on the file
    this->order(0, this->m);
    outfile << this->toString();
    outfile.close();

    //the hash table is flushed
    num++;
    this->flush();
    stop = false;
    is_max_reprobe = true;

    //the key count is now updated
    this->incrementValue(key);
}

if(is_max_reprobe)
    return;
```



```
/*method that order the hashTable according to quicksort algorithm*/
void HashTable::order(int low, int high)
{
    if (low < high)
    {
        //pi is partitioning index, arr[p] is now at right place
        int pi = partition_hash(low, high);

        // Separately sort elements before partition and after partition
        order(low, pi - 1);
        order(pi + 1, high);

    }
}
```

REDUCED MEMORY USAGE

- Using a value field large enough for the most repeated k-mer: **INEFFICIENT**

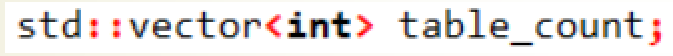
- Most of the k-mers appear:
  - **1** $\longrightarrow$ sequencing error
  - **C** $\longrightarrow$ sequencing coverage

- **IDEA:**
  - Small value field
  - Allow key to have more than one entry

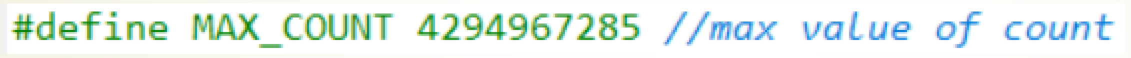
IMPLEMENTATION

- Small value field

Definition of count attribute as an int (32bit)

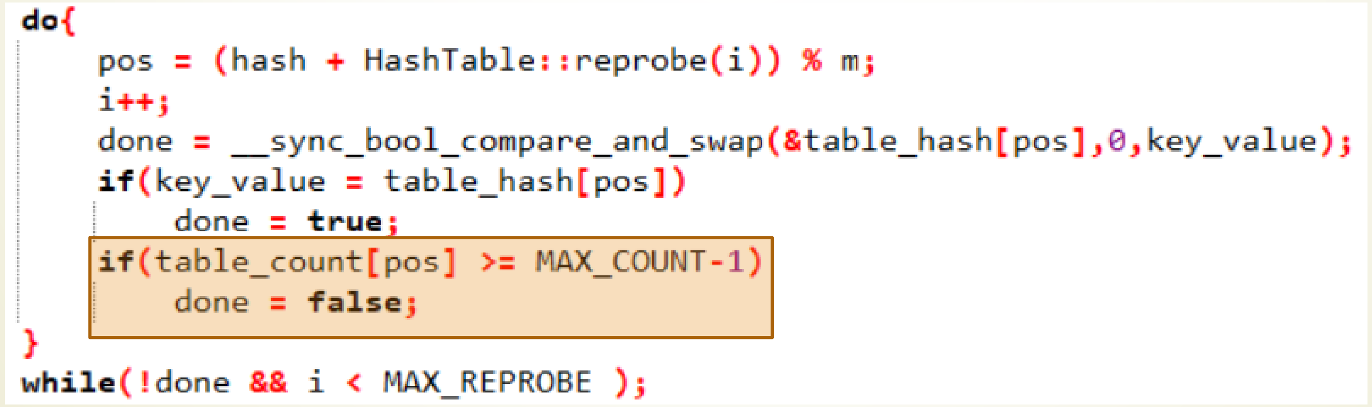


```
std::vector<int> table_count;
```



```
#define MAX_COUNT 4294967285 //max value of count
```

- Allow keys to have more than one entry



```
do{
    pos = (hash + HashTable::reprobe(i)) % m;
    i++;
    done = __sync_bool_compare_and_swap(&table_hash[pos],0,key_value);
    if(key_value = table_hash[pos])
        done = true;
    if(table_count[pos] >= MAX_COUNT-1)
        done = false;
}
while(!done && i < MAX_REPROBE );
```

KEY ENCODING

$M = 2l \longrightarrow$ lenght of hash-table

$pos(m, i) = (hash(m) + reprobe(i)) \bmod M$

The position of the key give as information about the l-lower bits

IDEA: store only 2k-l higher bits and the reprobe count

In many application we can obtain a space per key that is independent of the lenght of the l-mers and of the input string

IMPLEMENTATION

OUR CHOICE: do not implement, not fully compatible with other optimizations

Why?
The optimization is based on the fact that the key is encoded in the smallest possible field.
In order to take advantage from this optimization we must use a key value field that is the minimum large to encode a k-mer.
The dimension of this field is variable and depends on k.
The CAS function is an assembly primitive. It does not work with non-primitive data type.
Primitive data types have all fixed lenght, minimum 16bit for short int.
The only way to implement this optimization is to use CAS with short int, but this do not improve so much the performance of the algorithm so we do not include this optimization in our final implementation

TIME EVALUATION

- 3 Version of the algorithm:
  - Serial version
  - Multithreaded hash-table with locks
  - Multithreaded hash-table without locks
- We want to:
  - Show the speed up with respect to the number of threads
  - How much time it takes with respect to the lenght of the sequence
- We did each execution of the programs three times and then calcolate the average

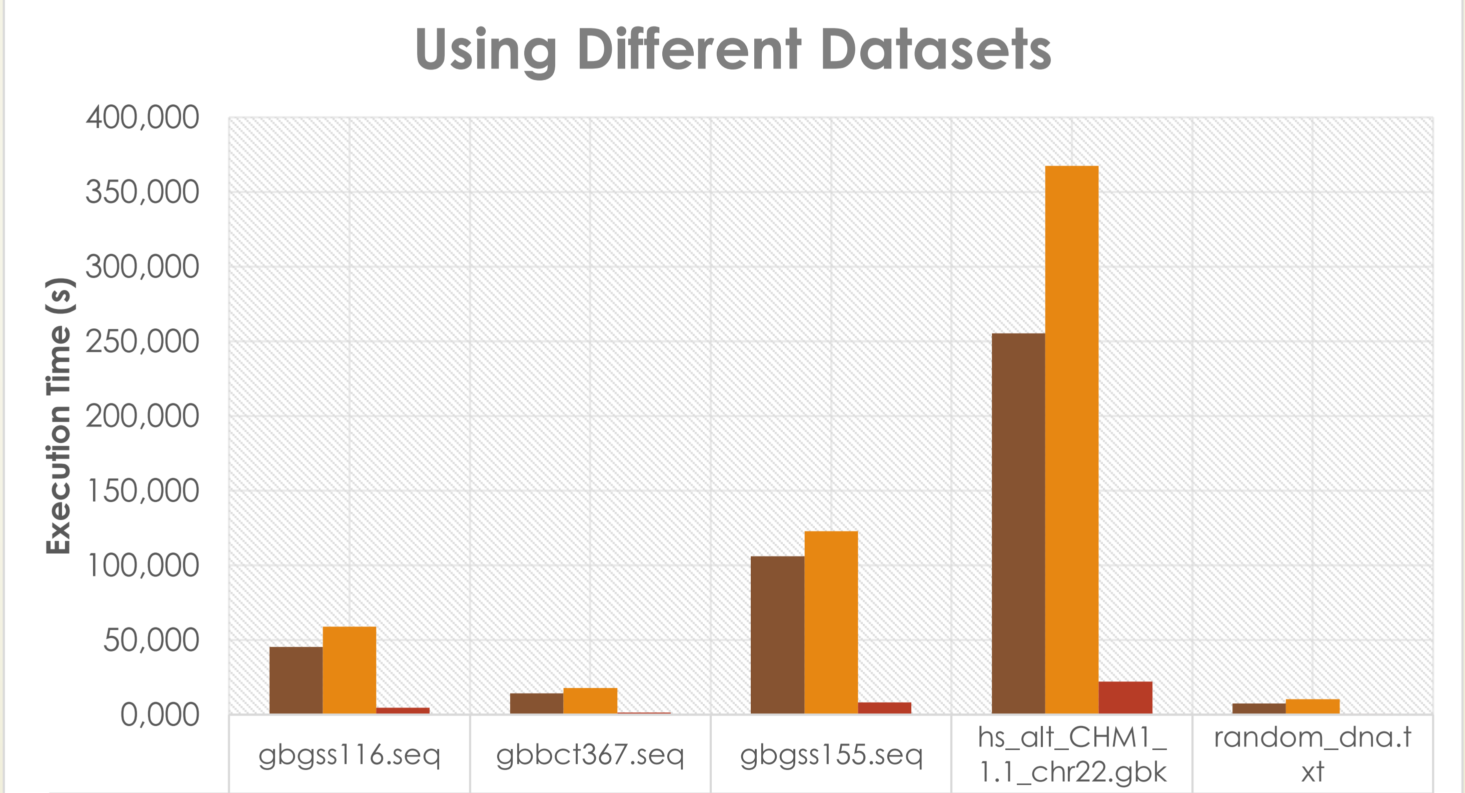
DATASET

- **DNA randomly generated**
- **gbbct155.seq:** complete genome of Escherichia Coli
- **gbbct367.seq**: complete genome of Lactococcus lactis
- **gbgss116.seq**: Chlorocebus aethiops genomic clone CH252-491O17, genomic surveey sequence
- **hs_alt_CHM1_1.1_chr22.gbk**: Homo sapiens chromosome 22 genomic scaffold

RESULT

- Execution time using different datasets

K = 4 L = 8 THREADS = 24

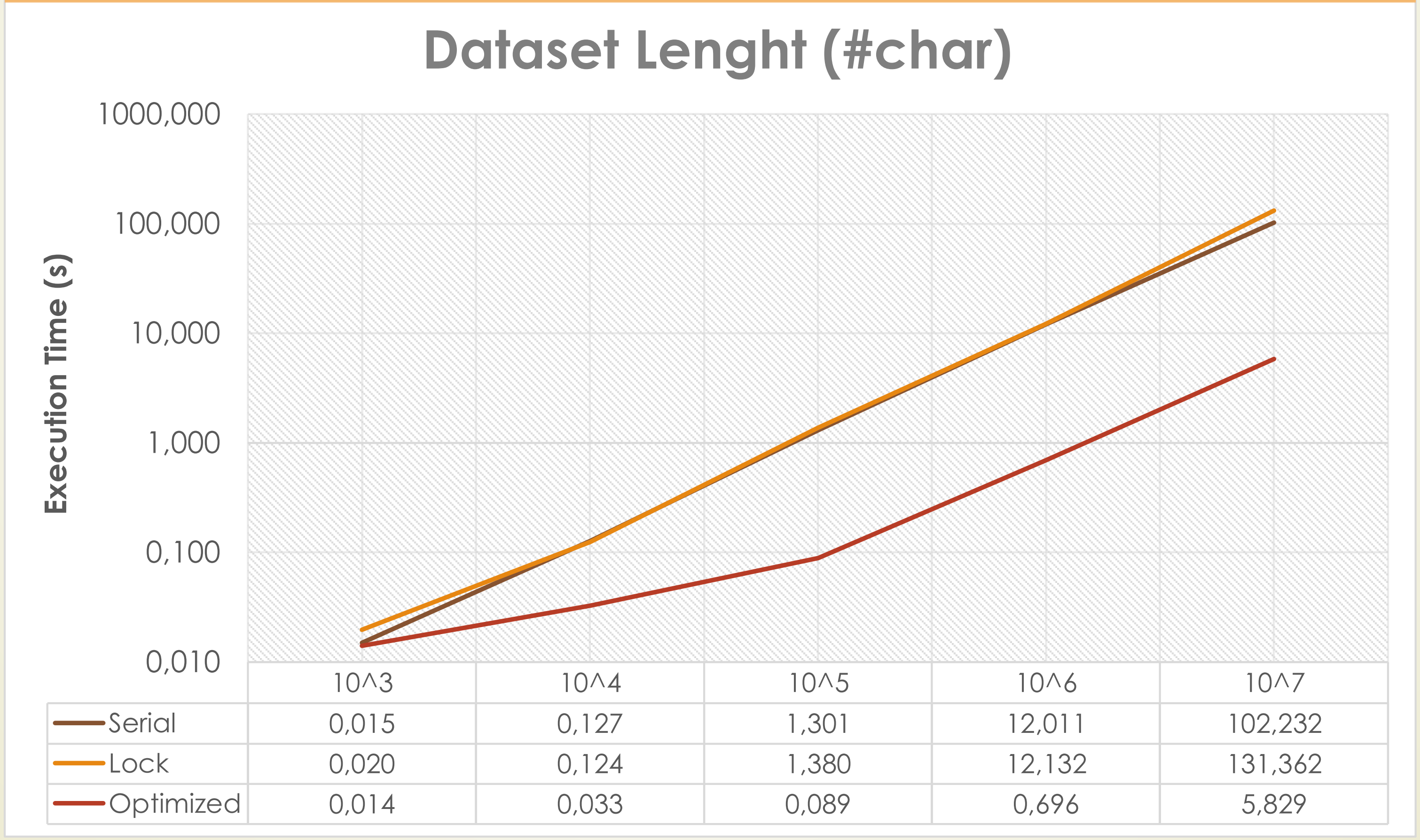


	gbgss116.seq	gbbct367.seq	gbgss155.seq	hs_alt_CHM1_ 1.1_chr22.gbk	random_dna.t xt
■ Serial	0.046	0.014	0.106	0.255	0.008
■ Lock	0.059	0.018	0.123	0.367	0.010
■ Optimized	0.005	0.001	0.008	0.022	0.001

RESULT

- Execution time compered to dataset length (random dna sequence)

K = 4 L = 8 THREADS = 24

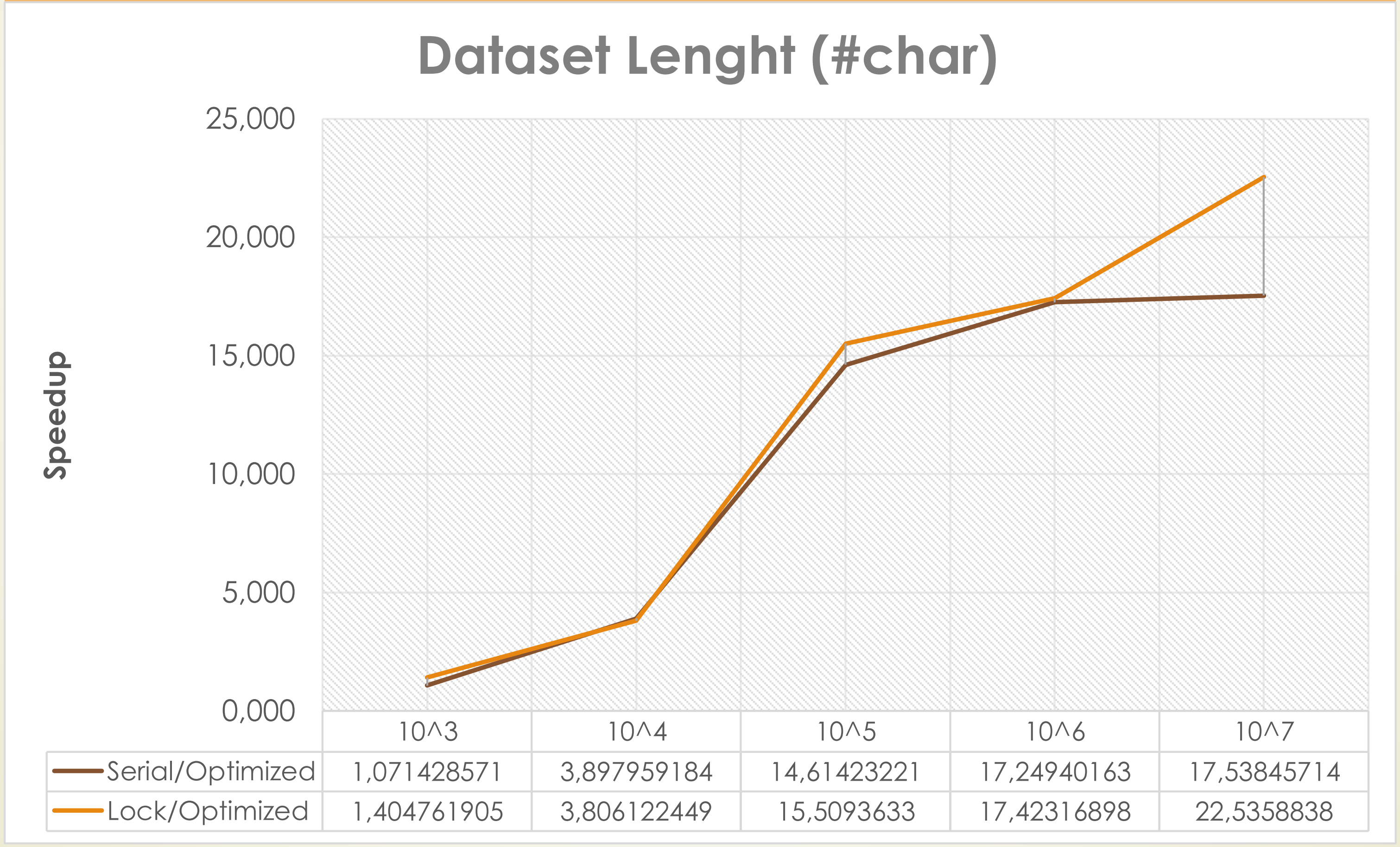


	10^3	10^4	10^5	10^6	10^7
Serial	0,015	0,127	1,301	12,011	102,232
Lock	0,020	0,124	1,380	12,132	131,362
Optimized	0,014	0,033	0,089	0,696	5,829

RESULT

- Speedup with respect to dataset length (random dna sequence)

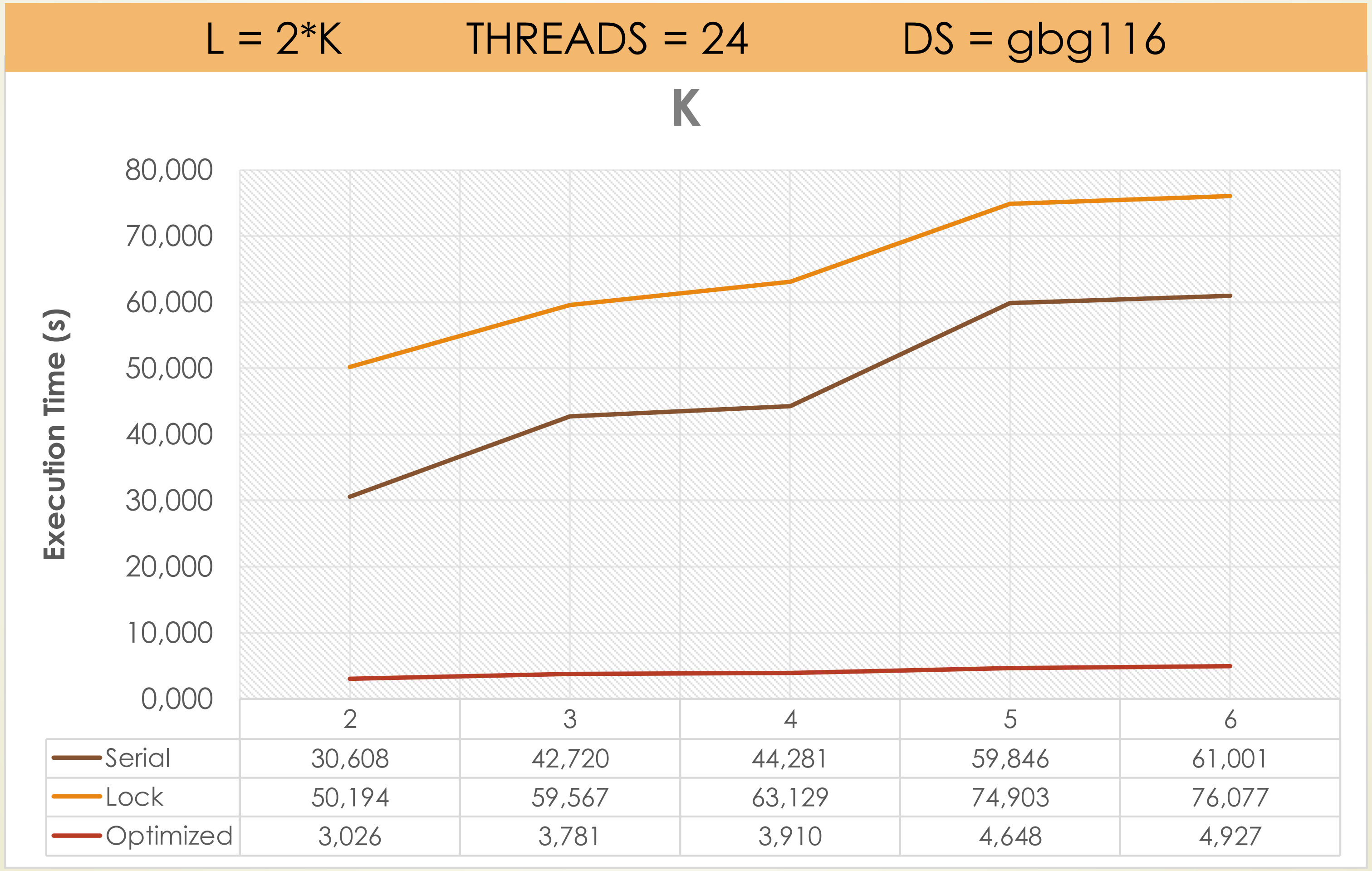
K = 4 L = 8 THREADS = 24



	10^3	10^4	10^5	10^6	10^7
Serial/Optimized	1,071428571	3,897959184	14,61423221	17,24940163	17,53845714
Lock/Optimized	1,404761905	3,806122449	15,5093633	17,42316898	22,5358838

RESULT

- Execution time compered to K

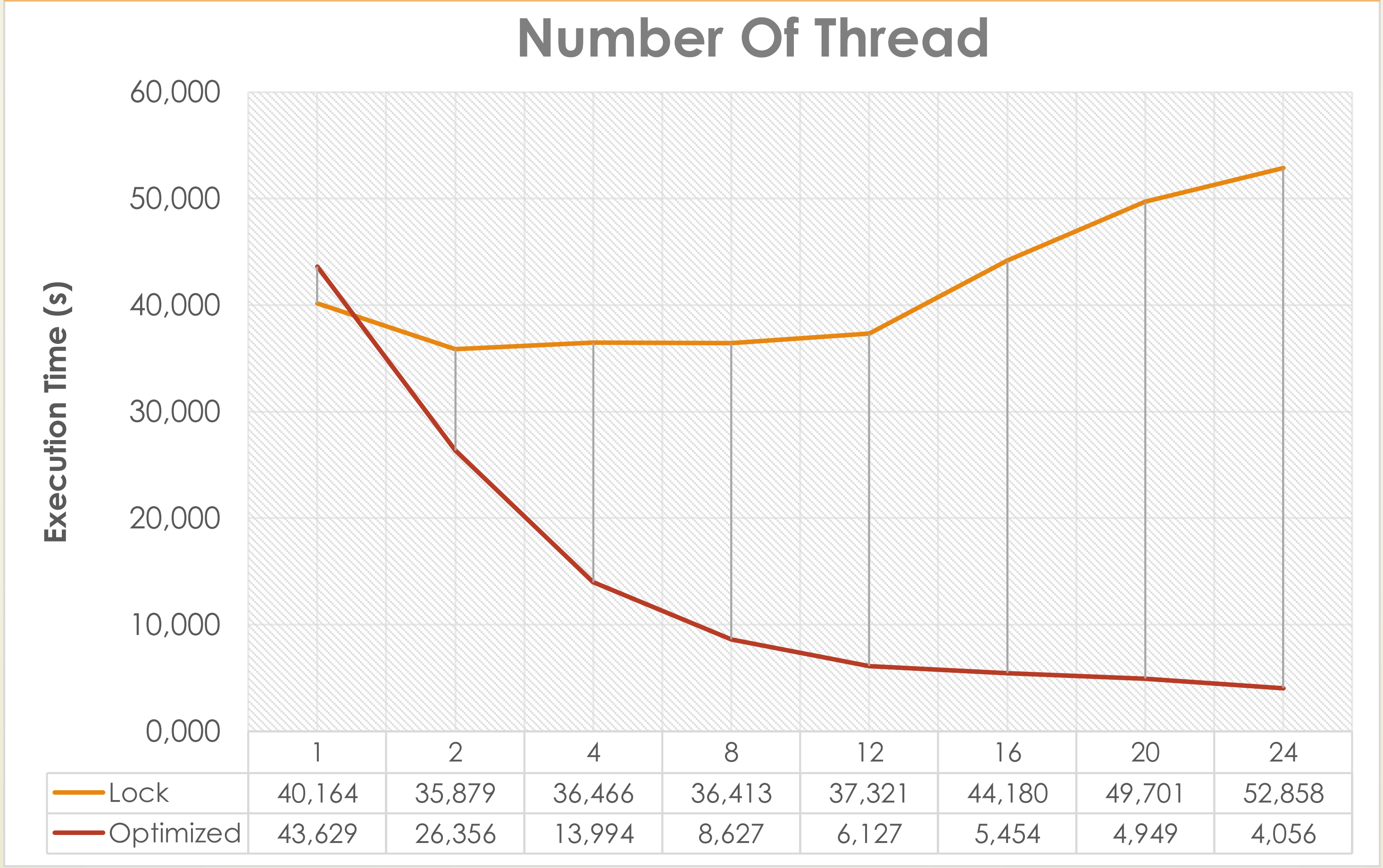


	2	3	4	5	6
Serial	30,608	42,720	44,281	59,846	61,001
Lock	50,194	59,567	63,129	74,903	76,077
Optimized	3,026	3,781	3,910	4,648	4,927

RESULT

- Execution time compered to number of thread

L = 2*K K = 4 DS = gbg116

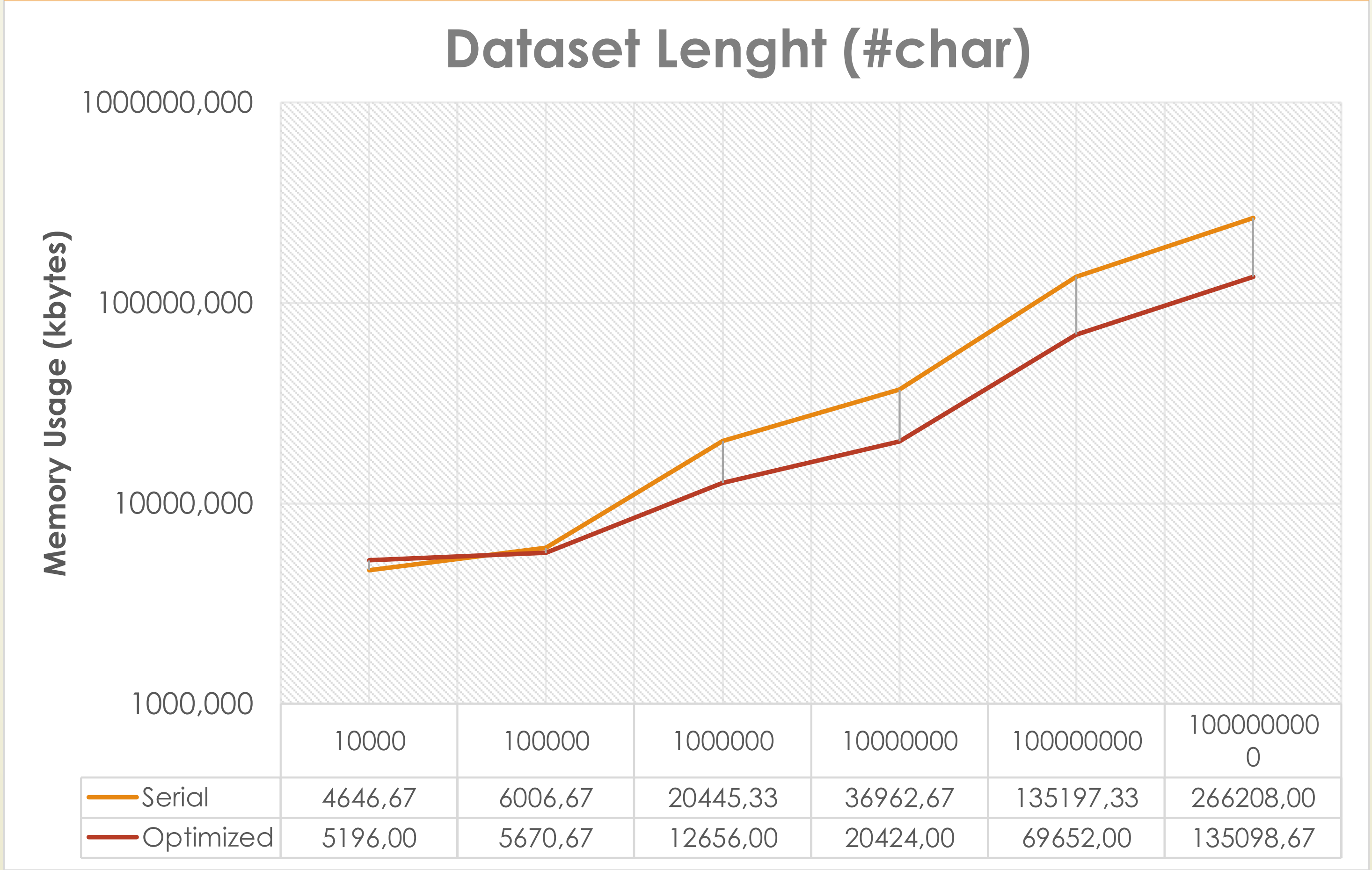


	1	2	4	8	12	16	20	24
Lock	40,164	35,879	36,466	36,413	37,321	44,180	49,701	52,858
Optimized	43,629	26,356	13,994	8,627	6,127	5,454	4,949	4,056

RESULT

- Memory usage compered to dataset length (random dna sequence)

K = 4 L = 8 THREADS = 24

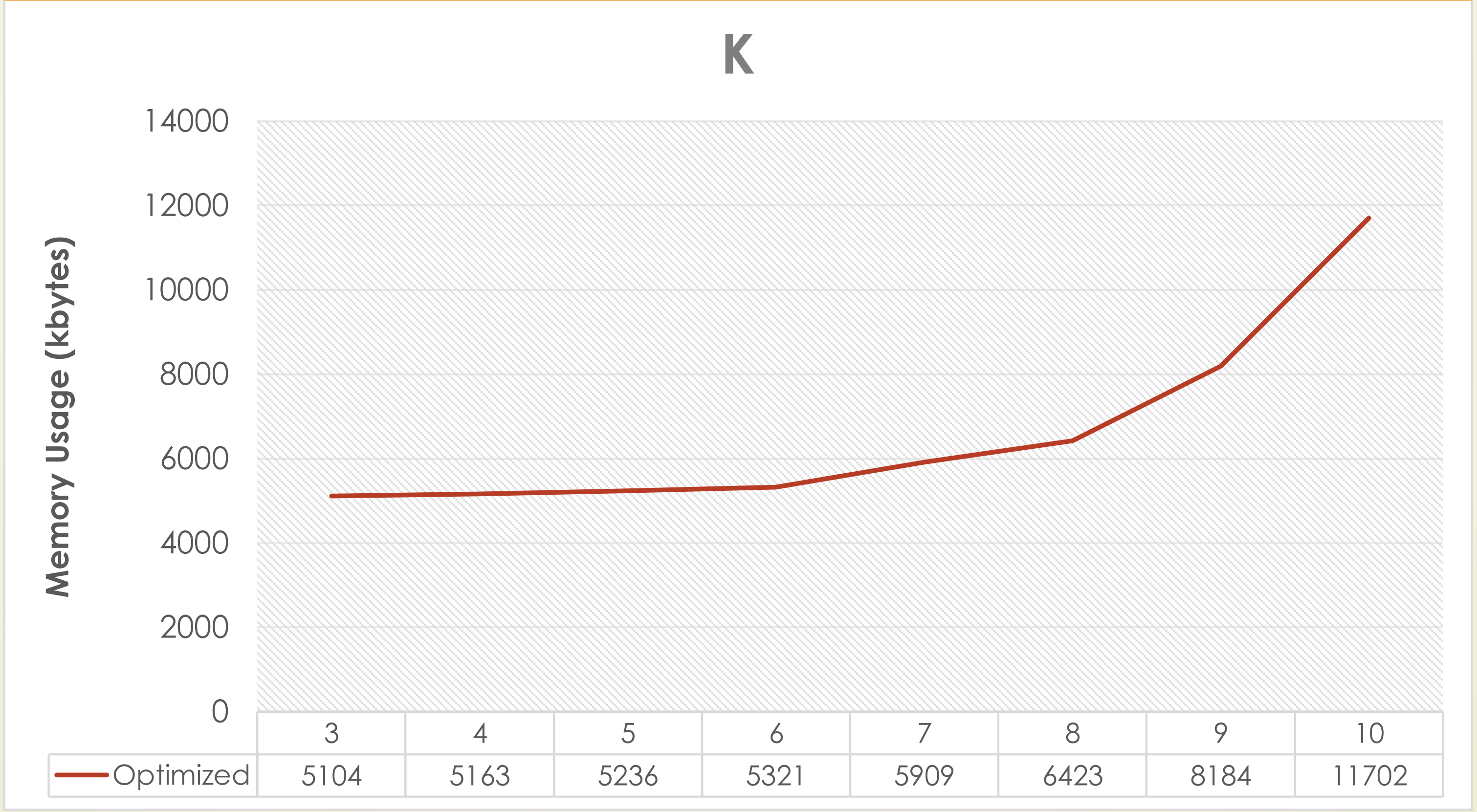


	10000	100000	1000000	10000000	100000000	1000000000
Serial	4646,67	6006,67	20445,33	36962,67	135197,33	266208,00
Optimized	5196,00	5670,67	12656,00	20424,00	69652,00	135098,67

RESULT

- Memory usage compered to K (all version have a similar behaviour with respect to k)

L = 2*K   THREADS = 24   DS = gbg116



	3	4	5	6	7	8	9	10
Optimized	5104	5163	5236	5321	5909	6423	8184	11702

CONSIDERATIONS

- Lock-parallel-version has always very bad performances
- Optimised version is always better than the other algorithms

- Lock-parallel-version scales bad with the number of threads
- Optimised version scales well with the number of threads

- Execution time grows linearly with the input length
- Speedup grows with the input length
- Execution time grows linearly with k
- Memory usage grows linearly with the input length
- Memory usage grows exponential with k

REFERENCES

[National Center for Biotechnology Information https://www.ncbi.nlm.nih.gov/genbank/release/145/]

[Random DNA Generator http://www.faculty.ucr.edu/mmaduro/random.htm]

[A fast, lock-free approach for efficient parallel counting of occurrences of k-mers Guillaume Marçais1, and Carl Kingsford]

[GitHub repository https://github.com/DiegoGabo/ProgettoAAPP/tree/master/dna_sequences]

[Compare and swap documentation http://www.cplusplus.com/reference/atomic/atomic/compare_exchange_strong/]