

AES Algorithm: A Performance Survey

Marco Rando S4249928
Riccardo Bianchini S4231932

February 10, 2020

Contents

1	Introduction	2
1.1	Hardware & Tools	2
2	AES: Advanced Encryption Standard	3
2.1	The Algorithm	3
2.1.1	Add Round Key	5
2.1.2	Byte Substitution: SubByte	5
2.1.3	Shiftrows	6
2.1.4	Mix Columns	6
2.2	Implementation	7
2.2.1	CTR Mode	10
2.3	Sequential Version	11
2.4	OpenMP Version	12
2.5	MPI Version	17
2.6	MPI with OpenMP Version	21
2.7	CUDA	23
3	Conclusions	30

Chapter 1

Introduction

In this report, we will discuss about our HPC project which consists in implement the AES encryption algorithm. Specifically, we will start by providing a high-level explanation of the algorithm in order to illustrate how this works and then we will provide implementations.

For the implementation part, we will initially see an implementation of the only AES algorithm (both encryption and decryption algorithms) and then we will show the CTR mode (mode suitable for efficiency) to extend the AES algorithm in order to encrypt and decrypt texts greater than the block size.

We will start from a sequential implementation, arriving to a distributed implementation (with MPI) and a hybrid implementation (MPI and OpenMP). Finally we will use the GPU through an implementation in CUDA.

At the end, we will do a brief sum-up and we will compare the performances (expressed in term of time) of the different implementations trying also to explain the results. For distributed implementations, we will perform a scaling analysis considering strong and weak scaling[5]. The implementations are based on AES-128 (by the way the other version of AES doesn't differ from this one in term of algorithms, we'll explain this later).

1.1 Hardware & Tools

To compile, execute and analyze the implementations we used the cluster provided by the university. This consists of 11 nodes all having an Intel Xeon Phi 7210 processor. The programs are compiled using Intel Parallel Studio 2017. The implementations in CUDA are instead performed on machines provided by Google Cloud Platform. Plots are realized using gnuplot.

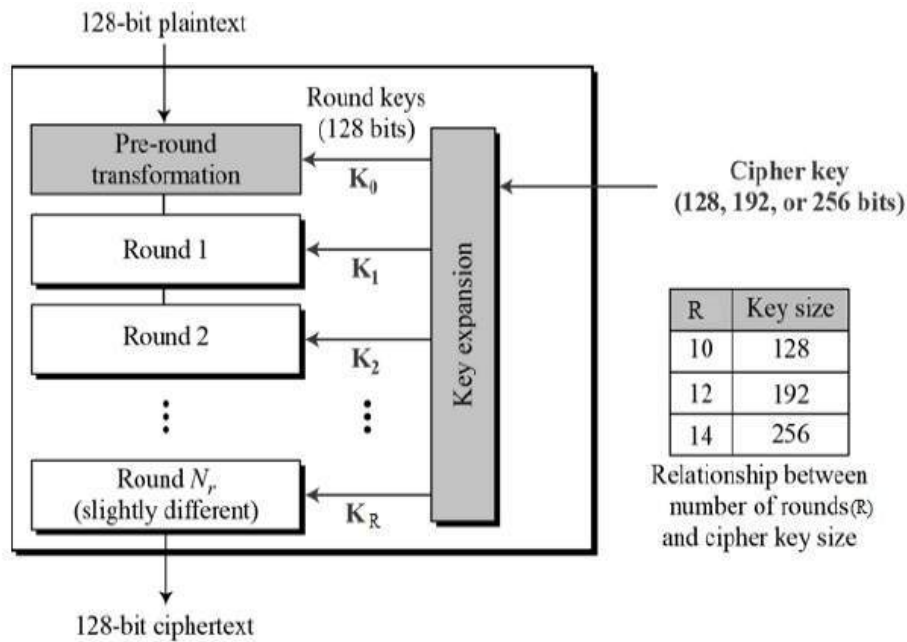
Chapter 2

AES: Advanced Encryption Standard

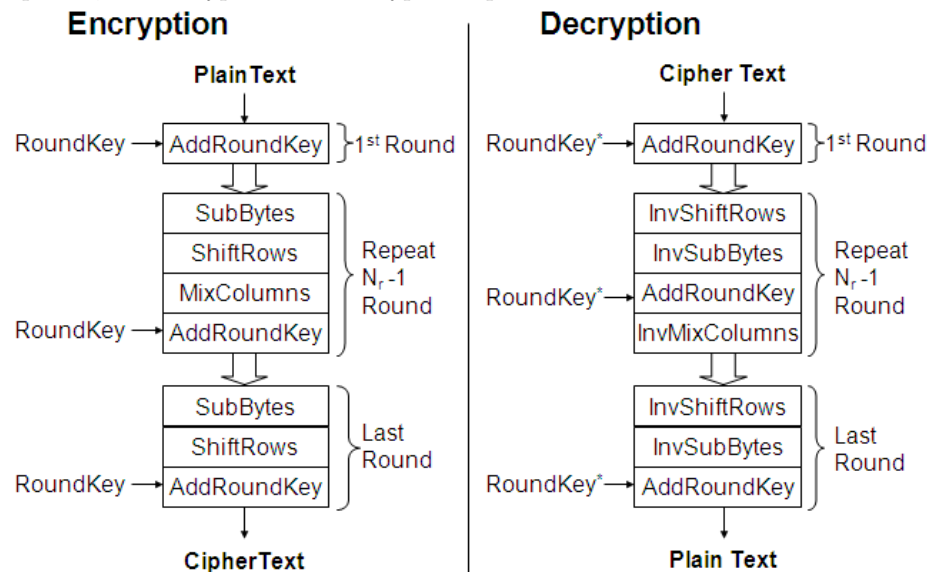
AES (Advanced Encryption Standard) is a specification for the encryption of data. The algorithm described by this specification is a symmetric key algorithm[7] and in particular it is a subset of the Rijndael block cipher[2] (so size of the key must be equal to the size of the block). In nowadays, three version of AES exist: 128, 192 and 256 (this number indicates the size of the block).

2.1 The Algorithm

The algorithm is based on a particular design principle called *substitution-permutation network*[6], in particular, it comprises of a series of linked operations, some of which involve replacing inputs by specific outputs (substitutions) and others involve shuffling bits around (permutations). The general explanation of the algorithm can be shown with this image:



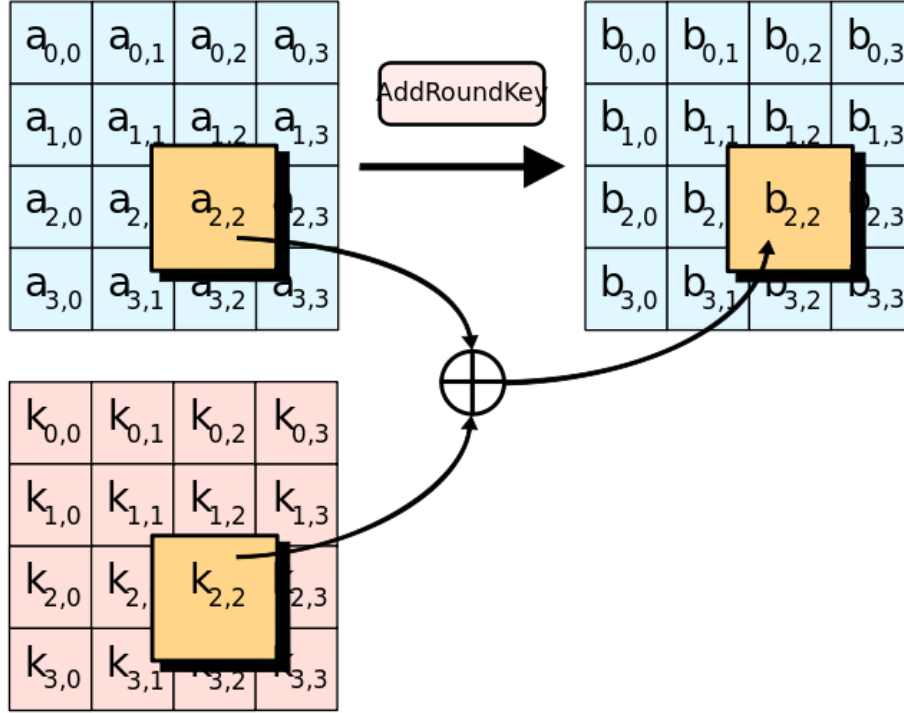
The idea consists in considering the block to encrypt as a matrix which is modified through different operations repeated in a fixed number of rounds. In specific, the encryption and decryption operations are defined as follow:



Let's see the operations in detail.

2.1.1 Add Round Key

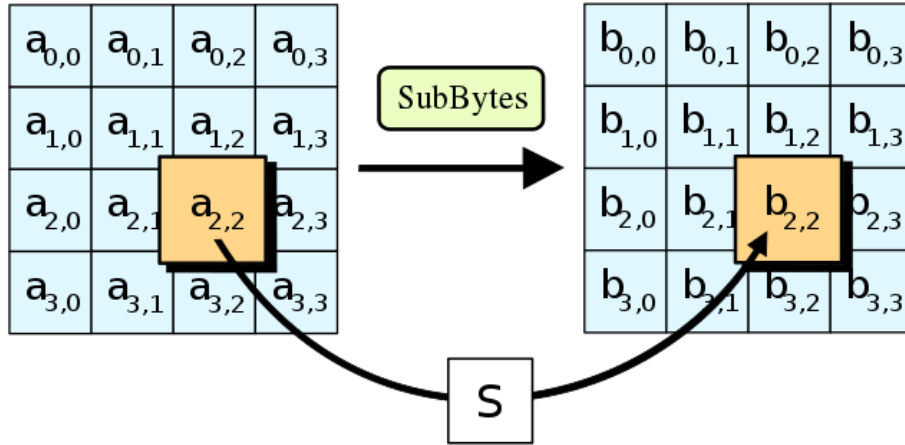
The 16 bytes of the matrix are now considered as 128 bits and are XORed to the 128 bits of the round key. If this is the last round then the output is the ciphertext. Otherwise, the resulting 128 bits are interpreted as 16 bytes and we begin another similar round.



For each round, a subkey is derived from the main key using a key schedule algorithm[1]. Each subkey is the same size as the state. The subkey is added by combining each byte of the state with the corresponding byte of the subkey using bitwise XOR. In the decryption phase, subkeys are used in reverse (i.e. the last subkey is used for the iteration 0 and so on).

2.1.2 Byte Substitution: SubByte

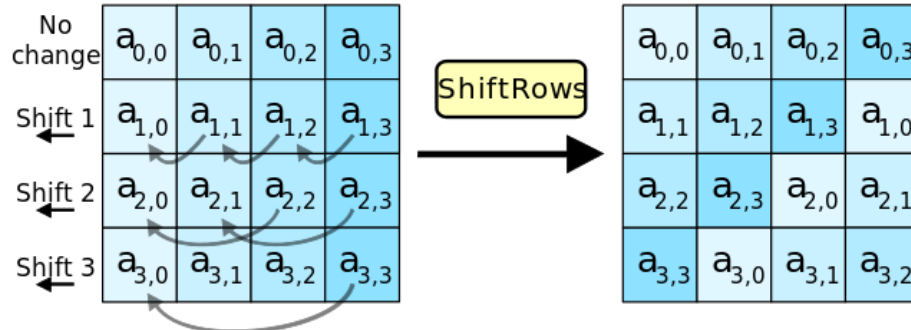
The 16 input bytes are substituted by looking up a fixed table (S-box[4]) given in design. The result is in a matrix of four rows and four columns.



In the inverse operation (invSubByte) the table used is the inverse of the S-box (RS-box).

2.1.3 Shiftrows

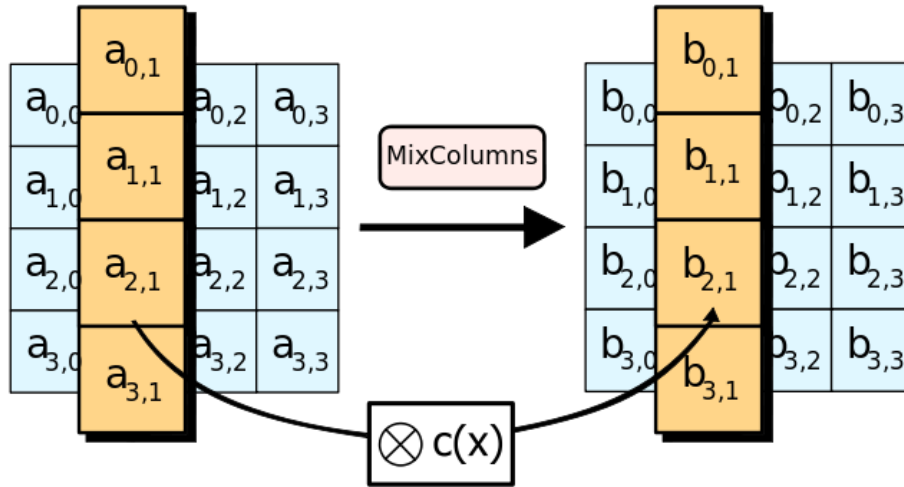
Each of the four rows of the matrix is shifted to the left. Any entries that 'fall off' are re-inserted on the right side of row. Each row is shifted of n positions with n number of line (so the line 0 will be shifted by 0 positions, the line 1 will be shifted by 1 position and so on).



The inverse operation (invShiftRow) consists in shifting the matrix to the right.

2.1.4 Mix Columns

Each column of four bytes is now transformed using a special mathematical function[3]. This function takes as input the four bytes of one column and outputs four completely new bytes, which replace the original column. The result is another new matrix consisting of 16 new bytes.



It should be noted that this step is not performed in the last round. In the inverse operation (`invMixColumns`) the function used is different.

2.2 Implementation

In this section we will see the main parts of the AES-128 algorithm (AES-192 and AES-256 are analogous), the full code can be seen in the GitHub repository. Before starting, we'll make some considerations:

- The sub-keys used in the encrypting and decrypting phase are equal.
- Pre-computing them, we can avoid to generate the sub-keys at each round.

The encryption phase can be coded in this way:

```
void aes_encrypt(unsigned char* text, unsigned char* key, unsigned char* sub_keys,
                char* encrypted, int rounds, int rowsize){
    int tot_size = rowsize * rowsize;
    unsigned char text_mat[rowsize][rowsize];
    build_matrix(&text[0], text_mat, rowsize);
    xor_key(&text_mat[0][0], sub_keys, rowsize);
    for(int r = 0; r < rounds-1; r++){
        sub_bytes(&text_mat[0][0], tot_size);
        left_shift_rows(&text_mat[0][0], rowsize, tot_size);
        mix_column128(&text_mat[0][0], tot_size);
        xor_key(&text_mat[0][0], &sub_keys[r*(rounds+1)+1], rowsize);
    }
    sub_bytes(&text_mat[0][0], tot_size);
    left_shift_rows(&text_mat[0][0], rowsize, tot_size);
    xor_key(&text_mat[0][0], &sub_keys[rounds*(rounds+1)], rowsize);
    for(int i = 0; i < rowsize; i++)
```



```

    for(int j = 0; j < rowsize; j++)
        encrypted[i*rowsize+j] = text_mat[i][j];
}

```

Where the function *build_matrix* simply transform the plain text in a matrix (by rows). The add round key (*xor_key*) it's just a xor of every byte:

```

void xor_key(char* text, char* key, int rowsize){
    for(int i = 0; i < rowsize; i++)
        for(int j = 0; j < rowsize; j++)
            text[i*rowsize+j] = text[i*rowsize+j] ^ key[i*rowsize+j];
}

```

The sub byte function is defined as we saw in the theoretical explanation:

```

void sub_bytes(unsigned char* mat, int totsize){
    for(int i = 0; i < totsize; i++)
        mat[i] = get_sbox_value(mat[i]);
}

```

The shift row operation is a standard row shifting algorithm:

```

void left_shift_rows(unsigned char* mat, int rowsize){
    int k,tmp;
    for(int i = 1; i < rowsize; i++){
        for(int j = 0; j < i; j++){
            tmp = mat[i*rowsize+j];
            for(k = 0; k < rowsize-i; k++)
                mat[i*rowsize+j+k] = mat[i*rowsize+j+k+1];
            mat[i*rowsize+j+k] = tmp;
        }
    }
}

```

And the mix column operation:

```

void mix_column128(unsigned char* text_mat, int totsize){
    unsigned char b0, b1, b2, b3;
    unsigned char result[totsize];
    int i;
    for (i = 0; i < 4; i ++){
        b0 = text_mat[i];
        b1 = text_mat[i + 4];
        b2 = text_mat[i + 8];
        b3 = text_mat[i + 12];
        result[i] = mul[b0][0] ^ (mul[b1][0]^b1) ^ b2 ^ b3;
        result[i + 4] = b0 ^ mul[b1][0] ^ (mul[b2][0]^b2) ^ b3;
        result[i + 8] = b0 ^ b1 ^ mul[b2][0] ^ (mul[b3][0]^b3);
        result[i + 12] = (mul[b0][0]^b0) ^ b1 ^ b2 ^ mul[b3][0];
    }
}

```

```

    }
    for(i = 0; i < tosize; i++)
        text_mat[i]=result[i];
}

```

Where *mul* is a matrix in which is stored every possible product by two and three for every possible byte (attention: the product by three is defined as the product by two with a xor in order to avoid the overflow). The decryption algorithm is the inverse of these operations:

```

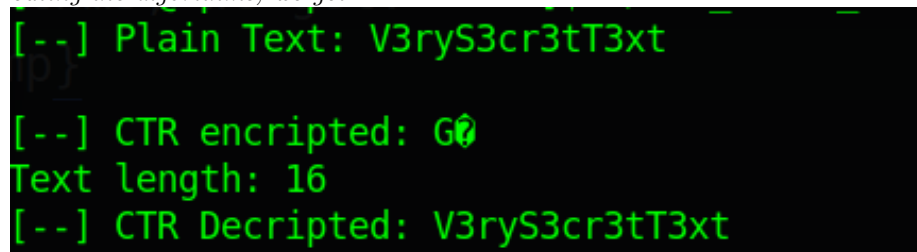
void aes_decrypt(char* text, char* key,unsigned char* sub_keys,
    unsigned char* decrypted,int rounds,int rowsize){
    int tot_size = rowsize * rowsize;
    unsigned char text_mat[rowsize][rowsize];
    build_matrix(text,text_mat,rowsize);
    xor_key(&text_mat[0][0],&sub_keys[rounds*(rounds+1)],rowsize);
    right_shift_rows(&text_mat[0][0],rowsize);
    inv_sub_bytes(&text_mat[0][0],tot_size);

    for(int r = rounds-1; r > 0; --r){
        xor_key(&text_mat[0][0],&sub_keys[r*(rounds+1)],rowsize);
        inv_mix_column128(&text_mat[0][0]);
        right_shift_rows(&text_mat[0][0],rowsize);
        inv_sub_bytes(&text_mat[0][0],tot_size);
    }
    xor_key(&text_mat[0][0],sub_keys,rowsize);
    for(int i = 0; i < rowsize; i++)
        for(int j = 0; j < rowsize; j++)
            decrypted[i*rowsize+j] = text_mat[i][j];
}

```

Let's consider the following example:

Example 2.2.1 Encryption and Decryption We want to encrypt and decrypt the text "V3ryS3cr3tT3xt" using AES128 with key "K1ng_G30rg3_rul3". Executing the algorithms, we get:



```

[--] Plain Text: V3ryS3cr3tT3xt
[--] CTR encrypted: G0
Text length: 16
[--] CTR Decrypted: V3ryS3cr3tT3xt

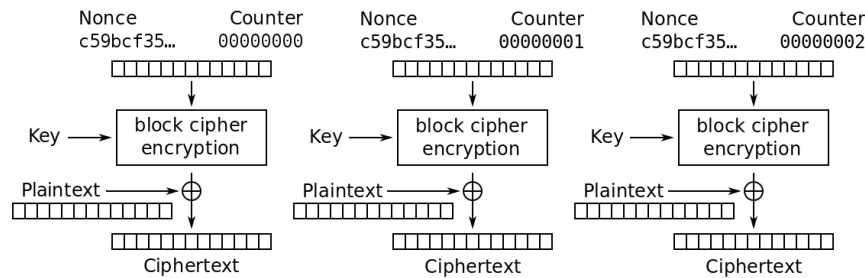
```

Naturally, in this way we can only encrypt and decrypt elements of size 16. If the size:

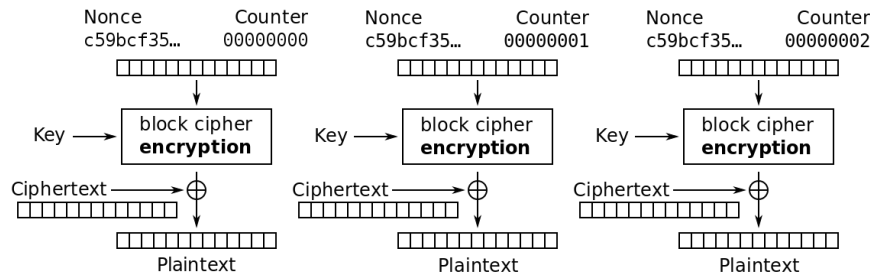
- is lower than 16: we can fill the block using some padding in order to reach the size of 16 (we used 0-padding).
- is higher than 16: we need to define a modality in order to consider bigger messages.

2.2.1 CTR Mode

As we saw before, we need a modality (a way to consider more blocks together) in order to encrypt and decrypt messages having size greater than 16 bytes. The modality we've chosen is the CTR Mode (Counter):



Counter (CTR) mode encryption



Counter (CTR) mode decryption

We split the messages in blocks of 16 bytes, then we compute the AES between the secret key and a initialization vector (called nonce) plus a counter (such that $iv + counter$ is unique per block) at the end we compute the xor between the plain text block and the result of the AES. In this way we get different advantages:

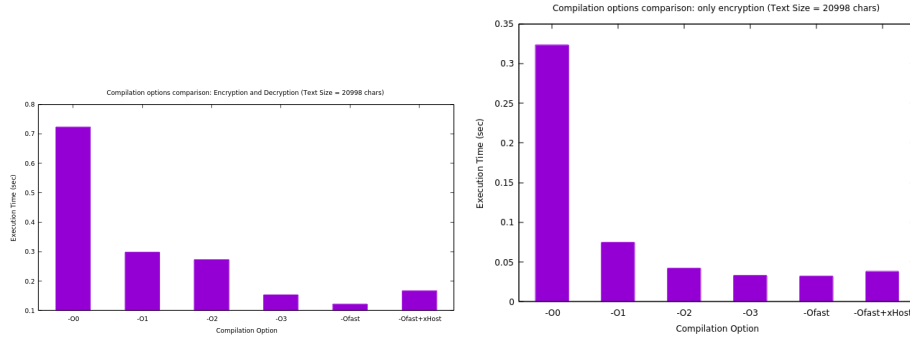
- We don't need to add padding (the initialization vector is composed by 16 bytes as well as the key).
- We can avoid to implement the decryption operation "explicitly" (we can decrypt using the xor properties).

- Each encrypted block is dependent only by the plain text block and the initialization vector, so if we split the message in blocks and compute every initialization vector (plus counter) at the start, we can encrypt and decrypt a message in parallel.

After we implemented it (the code is on the GitHub repository), we can start to do some analysis.

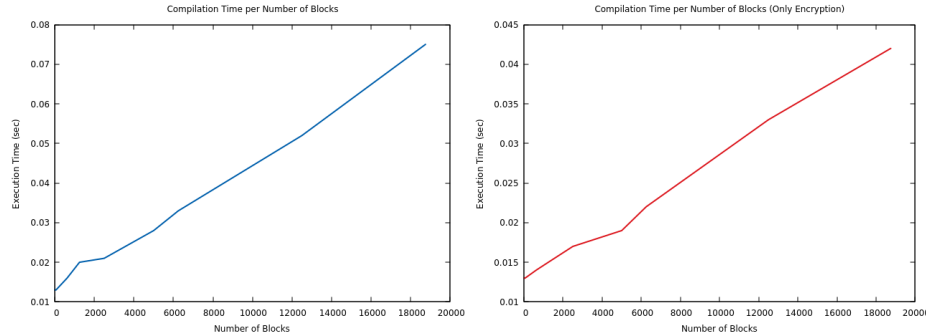
2.3 Sequential Version

The easiest way to implement the algorithm is surely by providing a sequential implementation. Once done, we search for the best optimization option for the compiler:

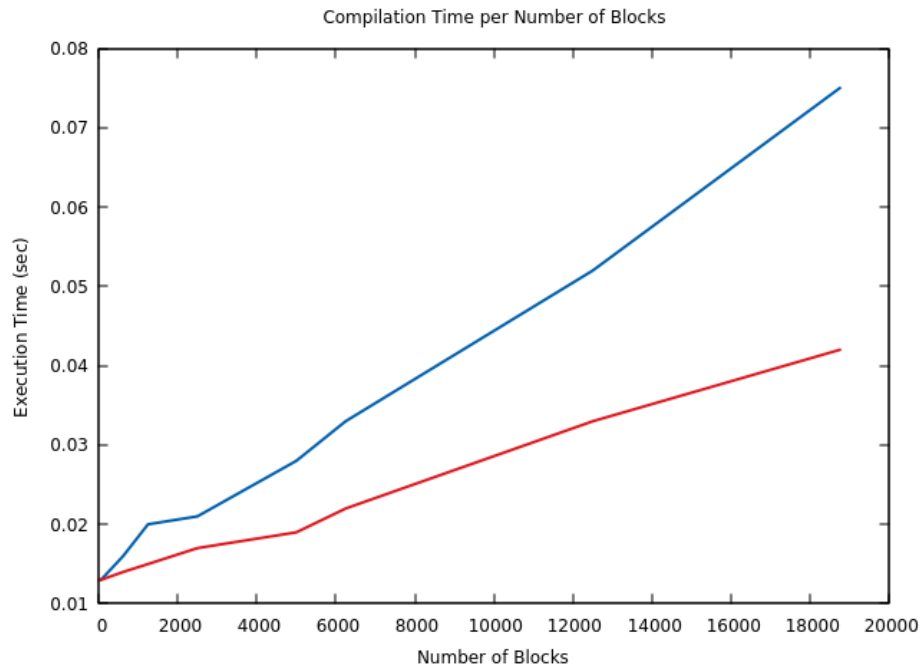


As we can see, the best compiler option (for our case) is the *fast* option and the xHost option worsens performance as there are no expensive vectorizable operations. We don't consider to do an analysis considering only the decryption time since decryption and encryption are the same operation (as we can see in the image above) for the properties of the xor operator.

Another analysis we can do consists in looking how the size of the plain text influence the time of encryption (and decryption). We will use the best compilation option to compute this:



From these plots we can notice that the time grows very slowly (the algorithm is very fast). Moreover, if we compare the two curves:



The only encryption curve is similar to the half of the time for encryption and decryption only for a certain number of blocks: this happen because of:

- the execution time is extremely low (so if we have only 1 or 2 blocks it's obvious that the time is irrelevant for a small number of blocks).
- the execution time is relevant for a big number of blocks.

2.4 OpenMP Version

Since we want to increase performances, we added parallelism using openMP. After reconsidering the structure of the AES algorithm:

```
void aes_encrypt(unsigned char* text, unsigned char* key, unsigned char* sub_keys,
                char* encrypted, int rounds, int rowsize){
    int tot_size = rowsize * rowsize;
    unsigned char text_mat[rowsize][rowsize];
    //this can be parallelized: each thread initializes a part of the matrix
    build_matrix(&text[0], text_mat, rowsize);
    //this can be parallelized
    xor_key(&text_mat[0][0], sub_keys, rowsize);
    //this cannot be parallelized in effcient way:
    // each round depends from the previous
    // moreover if we decide to use a static schedule we have to consider
    //that "rounds" will be alway 10 (a small number) so the cost of generating
```

```

//threads will be bigger than an sequential execution
for(int r = 0; r < rounds-1; r++){
    sub_bytes(&text_mat[0][0],tot_size);
    left_shift_rows(&text_mat[0][0],rowsize,tot_size);
    mix_column128(&text_mat[0][0],tot_size);
    xor_key(&text_mat[0][0],&sub_keys[r*(rounds+1)+1],rowsize);
}
//here a barrier is required
sub_bytes(&text_mat[0][0],tot_size);
left_shift_rows(&text_mat[0][0],rowsize,tot_size);
xor_key(&text_mat[0][0],&sub_keys[rounds*(rounds+1)],rowsize);
for(int i = 0; i < rowsize; i++)
    for(int j = 0; j < rowsize; j++)
        encrypted[i*rowsize+j] = text_mat[j][i];
}

```

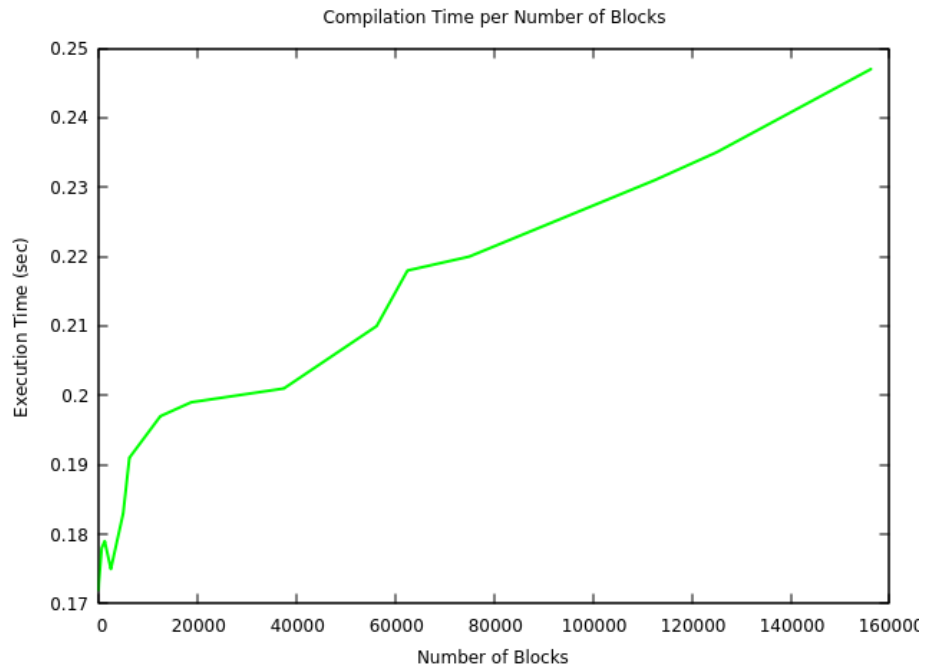
To get a more efficient parallelization we should parallelize the CTR mode: each thread will compute the AES and the xor for a certain number of blocks.

```

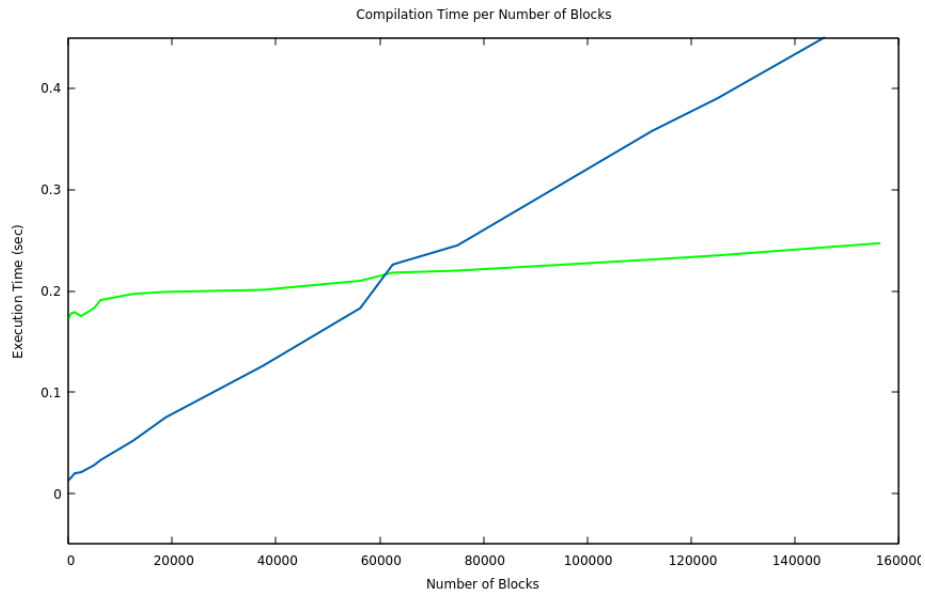
int ctr_enc(unsigned char* plain, unsigned char* key,unsigned char iv[16],
    unsigned char* sub_keys,int rounds,uint8_t* encrypted,int text_length){
    int num_blocks = (text_length / 16)+((text_length % 16)!=0);
    unsigned char counters[num_blocks][16];
    unsigned char blocks[num_blocks][16];
    uint8_t block[16];
    if(sub_keys==NULL)
        build_subkeys(key,sub_keys,16,rounds+1);
    #pragma omp parallel private(block) shared(blocks,counters)
    {
        #pragma omp single
        {
            build_counters(&iv[0],num_blocks,&counters[0]);
        }
        build_blocks(plain,num_blocks,blocks,text_length);
        #pragma omp for
        for(int i = 0; i < num_blocks; i++){
            aes128_encrypt(&counters[i][0],key,sub_keys,block);
            xor_string(block,blocks[i],&encrypted[i*16]);
        }
    }
    encrypted[(num_blocks)*16] = 0x0;
    return num_blocks;
}

```

The decryption operation is analogous. Computing the execution time for different document size, we got:



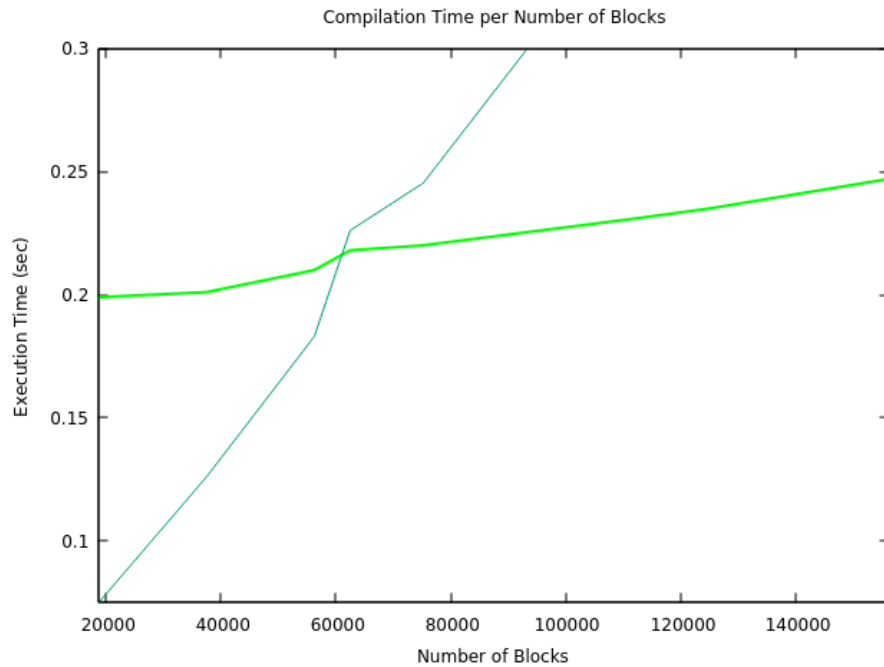
Apparently it seems more inefficient with respect to the performances of the sequential, but this happen because before we considered too small plain texts. If we consider bigger plain texts we can notice that at certain point the openMP version is faster:



This can be justified in a very simple way: initially the openMP is more ex-

pensive than the sequential version since we have to "pay" the cost of thread creation and because of this the sequential version is faster.

After a certain number of blocks (~ 62500) the sequential version is more expensive this because the parallel version curve grows slower than the sequential curve (threads work on more blocks at the same time reducing the speed of the curve). In the plot it seems constant because of the sequential curve by the way we know that it isn't. We can see better this result looking the point in which the sequential curve "exceed" the parallel curve:



Clearly the parallelism helps the performances "slowing down" the time curve. To further increase performances, we can study the different scheduling strategies: we considered just the static, dynamic and guided (these strategies are performed modifying the code in `ctr_mode.c`). Clearly for the `build_blocks` function we will use the **static** schedule since the work quantity at each iteration doesn't change:

```
void build_blocks(unsigned char* plain, int num_blocks,
    unsigned char (*blocks)[16], int size){
    #pragma omp for schedule(static)
    for(int i = 0; i < num_blocks; i++){
        for(int j = 0; j < 16 ; j++){
            blocks[i][j]=plain[i*16+j];
        }
    }
}
```

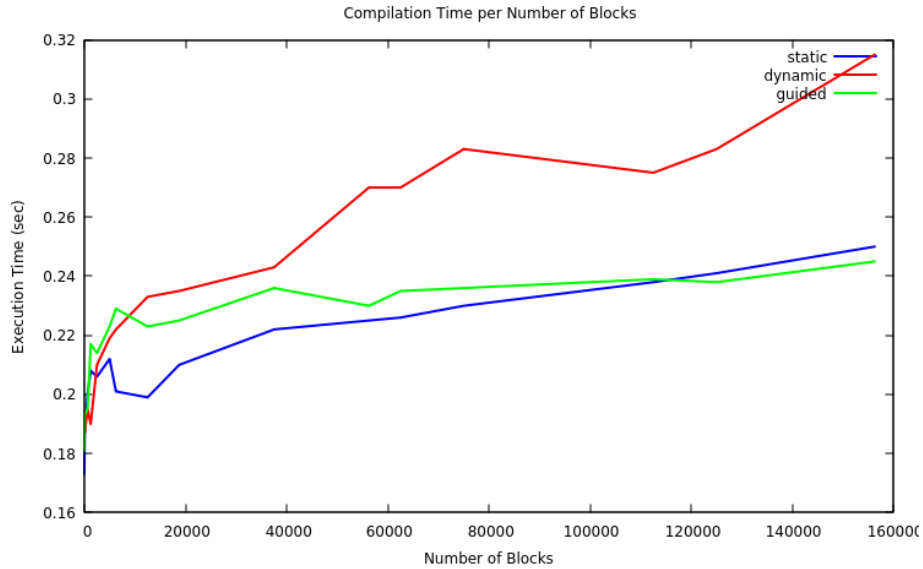
The `build_counters` function will be executed by only one thread:


```

void build_counters(unsigned char iv[16], int num_blocks, unsigned char (*counters)[16]){
    int i, j;
    for(i = 0; i < num_blocks; i++){
        for(j = 0; j < 16; j++){
            counters[i][j] = iv[j];
        }
        for(j = 15; j > 0; j--){
            if(iv[j] == 255){
                iv[j]=0;
                continue;
            }
            iv[j]++;
            break;
        }
    }
}

```

This because, in case of parallelism, we should manage the write on iv and the read of the “if” (it is cheaper to do it with a single thread, the total cost will be $\frac{text_length}{16} * 32$ which is linear). While for the encryption and decryption functions we can consider the different strategies:



As we can expect, the static and guided schedule strategies are faster than the dynamic scheduling. We could expect this since:

- The static scheduler would divide a loop over N elements into M subsets, and each subset would then contain strictly N/M elements.
- The load of each iteration is equal.
- Computing the load on the fly will be expensive

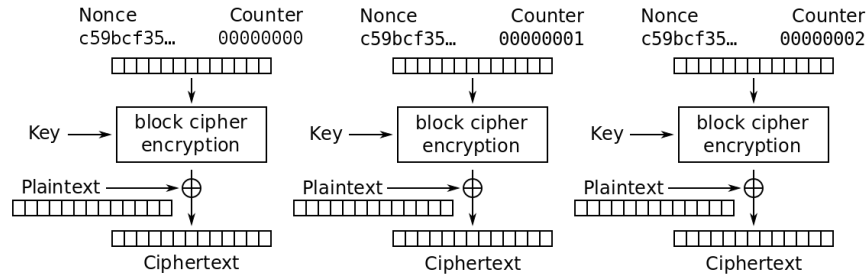
- Static scheduling is appropriate, where each iteration requires the same amount of work.
- Dynamic scheduling is appropriate for the case of a for build with iterations that require different or even unpredictable amounts of work (but this is not our case).
- Guided scheduling is appropriate for the case where threads can arrive at different times to build each iteration by requiring the same amount of work (and this is also our case and it is the reason why static and guided scheduling gives similar result in our case).

With a huge number of blocks, we suggest to use the guided scheduling since it's more probable that threads can arrive at different times in the loop (because of the build_blocks function).

2.5 MPI Version

We provided an MPI version in which we distribute the data in the cluster. Also in this case let's try to understand what we should (and can) distribute.

Surely we have to consider as minimal unit the block (16 byte) this because of the AES. At the same time we can only distribute the initialization vectors with counters.



Counter (CTR) mode encryption

Attention: we cannot distribute the plain text this because otherwise the whole algorithm loose its meaning (distributing the plain text means sending potential sensitive information which can be intercepted by a potential attacker). In order to try to get the best performances we will assume that each node knows the private key (this assumption is reasonable since it is the standard assumption of a symmetric cipher), so, every node can compute the set of subkeys a priori. We can imagine a master which sends the initialization vector with the counter (these informations are public so they can be sent) and receive the block encrypted which will be used to compute the xor operations. The code we produced is the following:

```
int ctr_enc(int rank, int nprocs, unsigned char* plain,
            unsigned char* key, unsigned char iv[16], unsigned char* sub_keys,
```

```

        int rounds,uint8_t* encrypted,int text_length){
int num_blocks = (text_length / 16)+((text_length % 16)!=0);
unsigned char counters[num_blocks][16];
unsigned char blocks[num_blocks][16];

int thread_nblock = num_blocks / nprocs;
unsigned char process_counters[thread_nblock][16];
unsigned char partial_blocks[thread_nblock][16];
unsigned char recv_blocks[num_blocks][16];

if(sub_keys==NULL) build_subkeys(key,sub_keys,16,rounds+1);

if(rank == 0){

    // master node init counters and blocks
    build_counters(&iv[0],num_blocks,&counters[0]);
    build_blocks(plain,num_blocks,blocks,text_length);

}
MPI_Scatter(&counters[0][0],thread_nblock*16,MPI_CHAR,
    &process_counters[0][0],thread_nblock*16,MPI_CHAR,0,MPI_COMM_WORLD);

for(int i = 0; i < thread_nblock; i++){
    aes128_encrypt(&process_counters[i][0],key,sub_keys,partial_blocks[i]);
}
if(rank !=0){
    MPI_Gather(&partial_blocks[0],thread_nblock*16,MPI_CHAR,NULL,
        thread_nblock*16,MPI_CHAR,0,MPI_COMM_WORLD);
}else{
    MPI_Gather(&partial_blocks[0],thread_nblock*16,MPI_CHAR,&recv_blocks[0],
        thread_nblock*16,MPI_CHAR,0,MPI_COMM_WORLD);
    for(int i = 0; i < num_blocks; i++){
        xor_string(recv_blocks[i],blocks[i],&encrypted[i*16]);
        encrypted[i*16+16]=0x0;
    }
    encrypted[(num_blocks)*16] = 0x0;
}
return num_blocks;
}

```

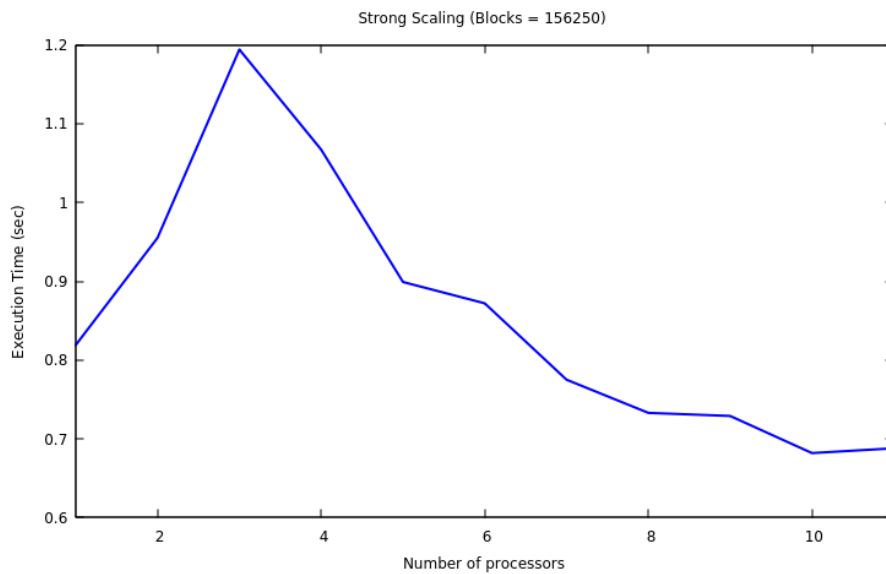
The MPI.Init and the MPI.Finalize are used in the main program this because otherwise we cannot use the encryption and decryption operations in the same program (MPI allows only one call of MPI.Init and MPI.Finalize). We used the scatter to send each initialization vector to each node and the gather to get the encoded blocks. At the end, the master node will compose the result.

For this version, we'll analyze the scaling (strong and weak) considering the time, speedup and efficiency.

First let's present some definitions:

- Strong scaling: how the solution time varies with the number of processors for a fixed total problem size[5].
- Weak scaling: how the solution time varies with the number of processors for a fixed problem size per processor[5].

Let's first analyze the strong scaling (we'll consider a text of 2500000 characters):

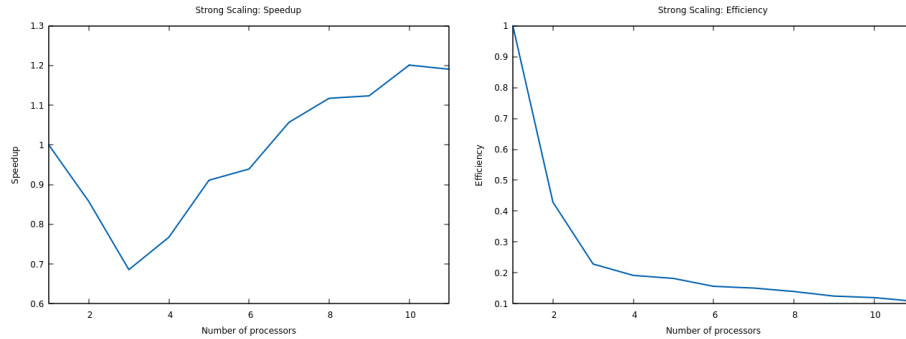


As we can see with a low number of nodes the performances decreases. This can be justified from the fact that the improvement obtained from the splitting of the workload is lower than the worsening generated from the communication between nodes. In brief:

- With only one node, we have to compute whole computations in that node, but there is "no network" (i.e. no communication costs).
- With 2/3 nodes, the communication cost is very relevant and each node will have to compute the AES128 for 78125/52083 blocks (an important workload). For this reason, performances decrease.

Increasing the number of processors, we split in more parts the workload, so, each node will have a small work to do. The communication cost is absorbed by the reduction of the workload per node, so performances increase.

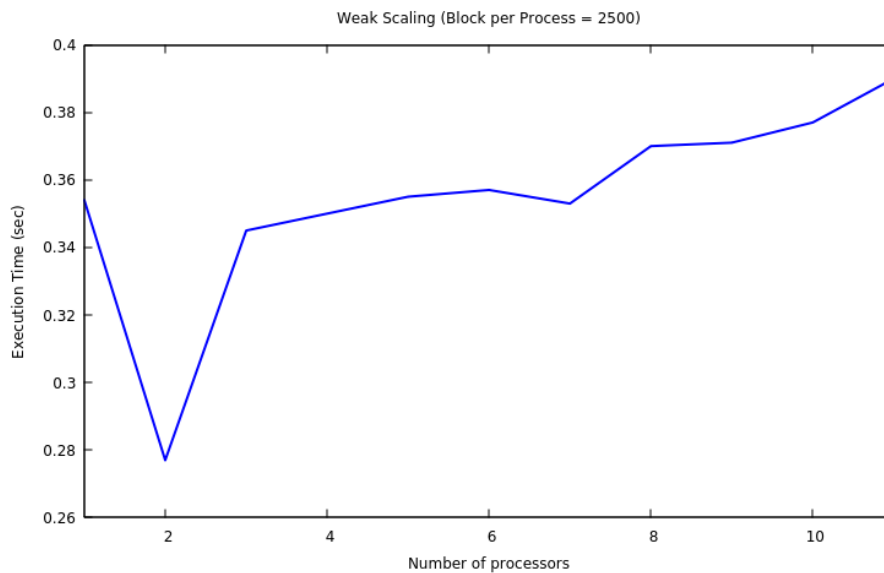
We can confirm this analysis looking the speedup and the efficiency plots:



As you can see, the speedup plot represents exactly the situation we saw in the execution time plot:

- For 2/3 processors performances decreases because of the communication cost and the size of the problem (workload).
- With a number of processors greater than 3, performances start to increase (for reason explained before).

By the way, despite the fact that we increase our performances with more processors (with 7 processors we get better performances than a sequential execution), the performances increasing is not enough as we can see from the efficiency plot. Infact we expect a bigger speedup with a big number of processors. Now let's look the weak scaling. For this, we'll consider 2500 blocks per processor:

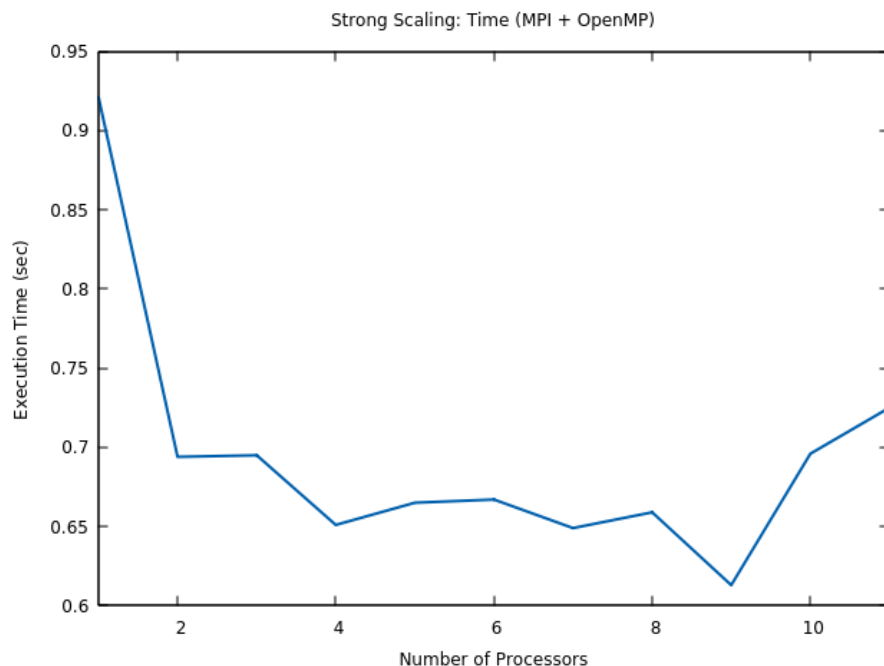


We can notice that increasing the number of processors maintaining fixed the workload per processor, the execution time "remain" constant using 3 to 7 processors and it start to increase with 8 processors.

The result obtained is not quite good in term of performances: distributing the computations we don't get a result better than the sequential or the parallel version. Let's try to add parallelism to this solution in order to see if results will be improved.

2.6 MPI with OpenMP Version

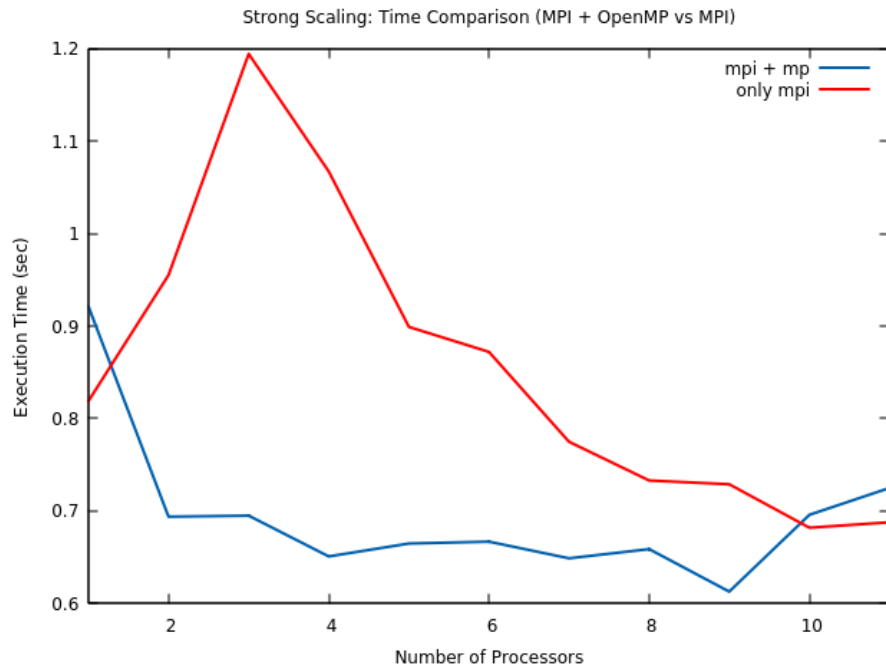
For the strong scaling (as we did before, we'll consider a text of 2500000 characters):



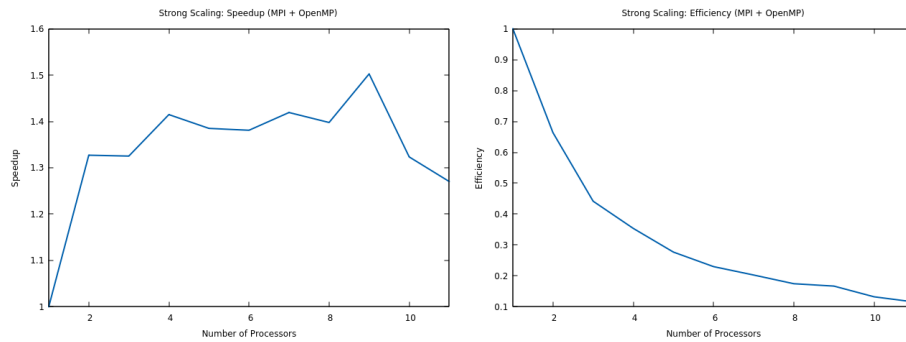
We can immediately notice some differences:

- With 1 node the cost is sigly higher: this because we have to pay the "price" of threads creation.
- With 2 (until 9) nodes the time is lower: the parallelism help in computations allowing to spend less time in computing AES blocks.
- With more than 9 nodes (with this workload) the time is sigly higher: this could depends from the communication cost plus the thread creation cost.

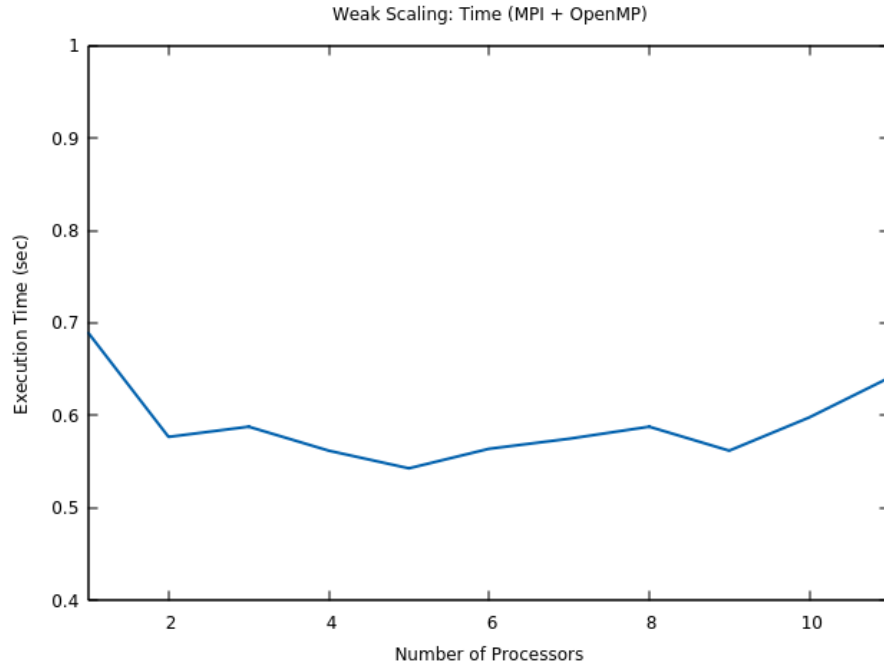
Comparing the two curves we can easily see that using parallelism we get advantages:



Observing the speedup and the efficiency, we can get a better idea of the advantages provided by this method:



As you can see, the speedup plot represents exactly the situation we saw in the execution time plot (in general speed continue to increase until we reach the 9 nodes). Now let's look the weak scaling. As we did before, we'll consider 2500 blocks per processor:



By the way, we get better performances using the sequential version or the parallel version. If you want to use a distributed version, we suggest to use a lot of node if the workload is large and to use the parallel version (MPI with OpenMP). To increase performances we can try to implement a GPU version of the algorithm.

2.7 CUDA

Since the AES algorithm works with matrices (we can see the algorithm as a series of transformations applied to the plain text matrix) we can increase performances by carrying out the algorithm on GPU. To do this, we need to modify also our implementation of the AES algorithm (aes.cu file) in order to execute it on GPU (using CUDA).

The produced code is structured as follows:

- *ctr_exec*: a function that realizes the CTR mode.
- *aes_encrypt*: kernel function that implements the encrypting phase according to the AES scheme.
- *sub_bytes*: device function that implements the bytes replacement phase using the Rijndael's sbox matrix.
- *shift_rows*: device function that implements the shifting phase of the rows of the matrix.

- *mix_columns*: device function that implements the mixing columns phase.
- *xor_key*: device function that implements the add round key phase (the xor between the text matrix and the new sub-key).
- *build_subkeys*: a function that produces the set of all sub-keys in advance for every round (realized for permonces reasons).

Below, we report the code created to implement these functions:

```
int ctr_exec(unsigned char* plain, unsigned char* result,
unsigned char* sub_keys, int rounds,int text_length){
    int num_blocks = (text_length / 16)+((text_length % 16)!=0);
    int tpb = 1024;

    if(sub_keys == NULL){
        printf("[xx] Error: you must pass the subkeys set!\n");
        exit(1);
    }

    // map message in gpu
    unsigned char* dev_plain;
    cudaMalloc((void**)&dev_plain,text_length*sizeof(unsigned char));
    cudaMemcpy(dev_plain,plain,text_length*sizeof(unsigned char),
        cudaMemcpyHostToDevice);

    // map result zone in gpu
    unsigned char *dev_result;
    cudaMalloc((void **)&dev_result, text_length * sizeof(unsigned char));
    cudaMemcpy(dev_result, plain, text_length * sizeof(unsigned char),
        cudaMemcpyHostToDevice);

    // map sbos on gpu
    unsigned char *dev_sbox;
    cudaMalloc((void **)&dev_sbox, 256 * sizeof(unsigned char));
    cudaMemcpy(dev_sbox, sbox, 256 * sizeof(unsigned char),
        cudaMemcpyHostToDevice);

    //map subkeys on gpu
    unsigned char *dev_keys;
    cudaMalloc((void **)&dev_keys, 10 * 16 * sizeof(unsigned char));
    cudaMemcpy(dev_keys, sub_keys, 10 * 16 * sizeof(unsigned char),
        cudaMemcpyHostToDevice);

    //execute aes
    int blk = ceil(num_blocks/tpb)==0?1:ceil(num_blocks/tpb);
    aes_encrypt<<<blk , tpb>>>(dev_plain, dev_result, dev_sbox, dev_keys
```

```

    , text_length);

    //map result to main memory
    cudaMemcpy(result, dev_result, text_length * sizeof(unsigned char),
               cudaMemcpyDeviceToHost);

    //free cuda
    cudaFree(dev_result);
    cudaFree(dev_plain);
    cudaFree(dev_keys);
    cudaFree(dev_sbox);

    return num_blocks;
}

__global__ void aes_encrypt(unsigned char *mat, unsigned char *result, unsigned char *sbox,
                           int id = (blockDim.x * blockIdx.x + threadIdx.x) * 16;

__shared__ unsigned char shared_sbox[256]; // the s-box matrix in shared memory
__shared__ unsigned char shared_keys[176]; // the sub-keys in shared memory
if (threadIdx.x == 0){
    for (int i = 0; i != 256; ++i){
        shared_sbox[i] = sbox[i];
        if (i < 176) shared_keys[i] = keys[i];
    }
}
__syncthreads(); // barrier
short idx, c = 0;
unsigned char iv_new[16];
unsigned char temp[16];

#pragma unroll
for (int i = 0; i < 16; ++i){
    temp[i] = mat[id + i];
    iv_new[i]=IV[i];
}

// build new initialization vector
for (idx = 15; idx >= 0; idx--){
    short shift = (16 - (idx + 1)) * 8;
    unsigned char op1 = IV[idx];
    unsigned char op2 = ((id & 0xff << shift) >> shift);
    iv_new[idx] = op1 + op2 + c;
    c = (iv_new[idx] > op1 && iv_new[idx] > op2) ? 0 : 1;
}
// AES algorithm

```

```

        xor_key(iv_new, shared_keys, 0);

#pragma unroll
for(int i = 1; i < 10; i++){
    byte_sub(iv_new, shared_sbox);
    shift_rows(iv_new);
    mix_columns(iv_new);
    xor_key(iv_new, shared_keys, i);
}
byte_sub(iv_new, shared_sbox);
shift_rows(iv_new);
xor_key(iv_new, shared_keys, 10);

//XOR with plain block
#pragma unroll
for (int i = 0; i < 16; ++i){
    unsigned char res = iv_new[i] ^ temp[i];
    iv_new[i] = res;
}
#pragma unroll
for (int i = 0; i < 16; ++i)
    result[id + i] = iv_new[i];
    result[width] = 0x0;
}

// Key Addition Kernel
__device__ void xor_key(unsigned char *mat, unsigned char *key,
    const unsigned int &round){
    #pragma unroll
    for (int i = 0; i < 16; ++i)
        mat[i] ^= key[(16 * round) + i];
}

// byte substitution (S-Boxes)
__device__ void byte_sub(unsigned char *mat, unsigned char* s_sbox){
    #pragma unroll
    for (int i = 0; i < 16; ++i)
        mat[i] = s_sbox[mat[i]];
}

```

```

// Shift rows
__device__ void shift_rows(unsigned char *mat){
    int k,tmp;
    #pragma unroll
    for(int i = 1; i < 4; i++){
        for(int j = 0; j < i; j++){
            tmp = mat[i*4];
            for(k = 0; k < 3; k++)
                mat[i*4+k] = mat[i*4+k+1];
            mat[i*4+k] = tmp;
        }
    }
}

// Mix column
__device__ void mix_columns(unsigned char* text_mat){
    unsigned char b0, b1, b2, b3;
    unsigned char result[16];
    int i;
    #pragma unroll
    for (i = 0; i < 4; i ++){
        b0 = text_mat[i];
        b1 = text_mat[i + 4];
        b2 = text_mat[i + 8];
        b3 = text_mat[i + 12];
        result[i] = dob[b0] ^ triple[b1] ^ b2 ^ b3;
        result[i + 4] = b0 ^ dob[b1] ^ triple[b2] ^ b3;
        result[i + 8] = b0 ^ b1 ^ dob[b2] ^ triple[b3];
        result[i + 12] = triple[b0] ^ b1 ^ b2 ^ dob[b3];
    }
    #pragma unroll
    for(i = 0; i < 16; i++)
        text_mat[i]=result[i];
}

void build_subkeys(unsigned char* key,unsigned char* sub_keys, int totdsize,
                  int rounds){
    for(int i = 0; i < rounds; i++){
        if(i == 0){
            for(int j = 0; j < totdsize; j++){
                sub_keys[j] = (key[j] == 0x0)?0x0:key[j];
            }
        }else{
            sub_keys[i*totdsize] = (sub_keys[(i-1)*totdsize]^
                (get_sbox_value(sub_keys[(i-1)*totdsize+13]) ^rc[i-1]));
        }
    }
}

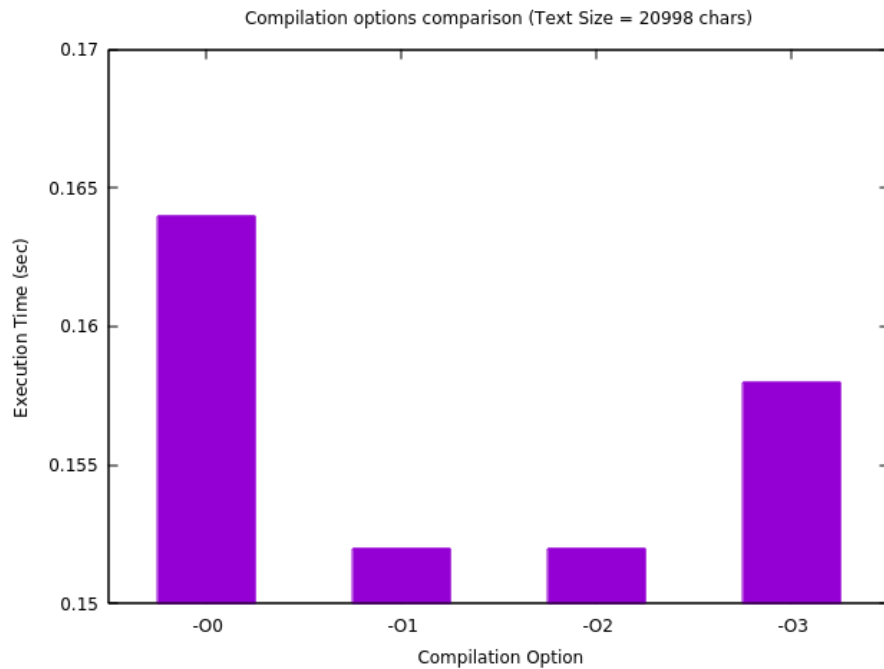
```

```

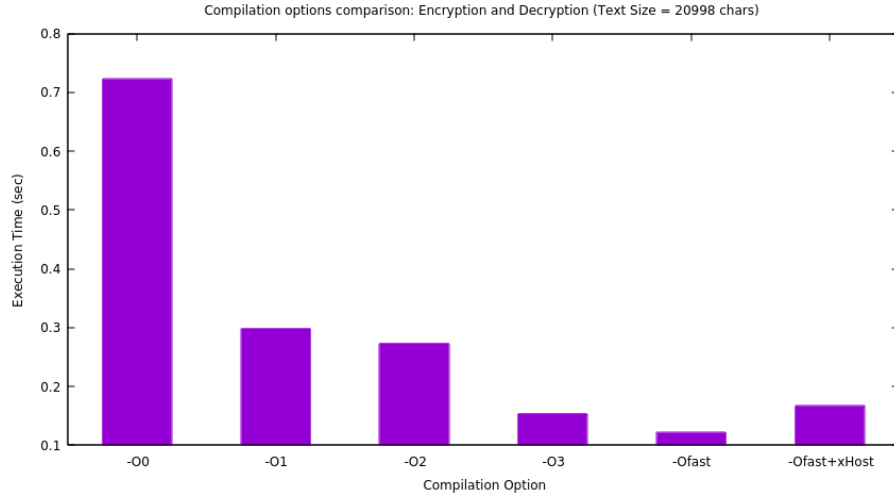
sub_keys[i*totsize+1] = (sub_keys[(i-1)*totsize+1]^
    get_sbox_value(sub_keys[(i-1)*totsize+14]));
sub_keys[i*totsize+2] = (sub_keys[(i-1)*totsize+2]^
    get_sbox_value(sub_keys[(i-1)*totsize+15]));
sub_keys[i*totsize+3] = (sub_keys[(i-1)*totsize+3]^
    get_sbox_value(sub_keys[(i-1)*totsize+12]));
for(int j = 4; j < totsize; j+=4){
    sub_keys[i*totsize+j] = (sub_keys[(i*totsize)+j-4]^
        sub_keys[((i-1)*totsize)+j]);
    sub_keys[i*totsize+(j+1)] = (sub_keys[(i*totsize)+j-3]^
        sub_keys[((i-1)*totsize)+j+1]);
    sub_keys[i*totsize+(j+2)] = (sub_keys[(i*totsize)+j-2]^
        sub_keys[((i-1)*totsize)+j+2]);
    sub_keys[i*totsize+(j+3)] = (sub_keys[(i*totsize)+j-1]^
        sub_keys[((i-1)*totsize)+j+3]);
}
}
}
}

```

First, let's find the best compilation option:

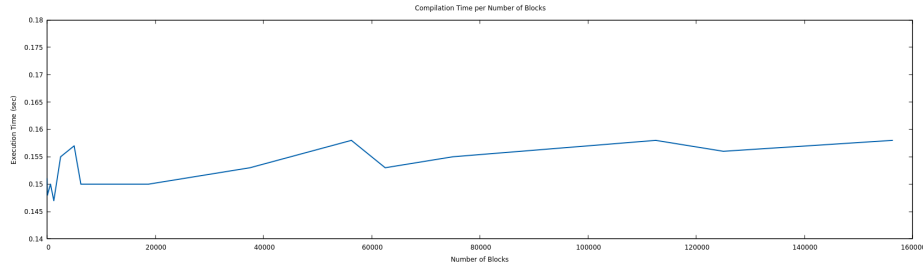


It can be noticed that performances for the difference compilation options are increased with respect to the sequential version:



We can see that with the O3 optimization option the performances are worse than the use of the O2 optimization option. This fact can be explained by the optimization aggressiveness provided by the O3 option: an intuition for why this is somewhat common is that O3 does a lot of optimizations that increase code size (e.g., more aggressive loop unrolling), which almost always increases performance in microbenchmarks, but can sometimes decrease performance in workloads with a larger code footprint.

Another analysis we can do consists in looking how the size of the plain text influence the time of encryption and decryption. We will use the best compilation option to compute this:

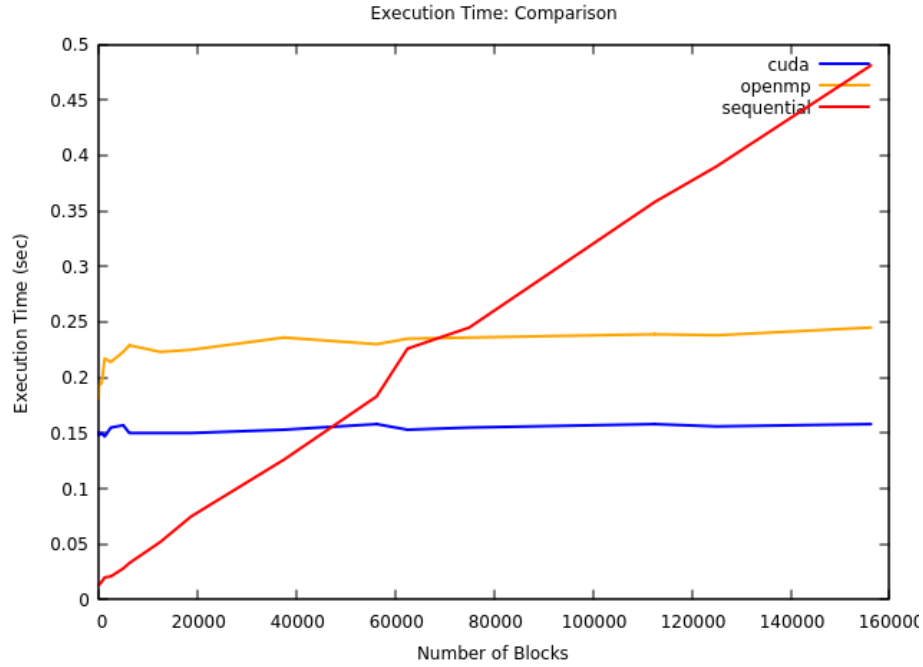


We can see that the execution time of the algorithm grows very slowly, making explicit the advantage of using the GPU. In fact, the difference in execution times between an encryption and decryption of a 1 block text and a 156250 block text is only 0.007 seconds.

Chapter 3

Conclusions

We can conclude showing a comparison between the sequential, openmp and cuda methods. We have excluded MPI since the performances are worse than the sequential version in the tested cases, however we are aware of the fact that with an input composed of a sufficiently large number of blocks the performances of MPI will surpass those of the sequential version:



We can see that for sufficiently small inputs ($n_{blocks} \lesssim 40000$) it is convenient to use the sequential version for the following reasons:

- there is no overhead (e.g. thread creation).

- for small input dimensions the problem is 'simple' enough to be carried out in a short time.

We can also notice that the version in CUDA has a trend similar to the trend of the openMP version, but it solves the same problem (same size of the input) in less time (~ 0.05 seconds less).

After looking at the results, we can suggest to use:

- sequential version: small texts and images (e.g. web contents).
- CUDA version (or openMP in case of incompatible hardware): large files (such as disk images, RAW files, etc).

Bibliography

- [1] Wikipedia contributors. *AES key schedule* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 5-February-2020]. 2019. URL: https://en.wikipedia.org/w/index.php?title=AES_key_schedule&oldid=921145964.
- [2] Wikipedia contributors. *Block cipher* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 5-February-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=Block_cipher&oldid=938211524.
- [3] Wikipedia contributors. *Rijndael MixColumns* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 5-February-2020]. 2019. URL: https://en.wikipedia.org/w/index.php?title=Rijndael_MixColumns&oldid=931079146.
- [4] Wikipedia contributors. *Rijndael S-box* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 5-February-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=Rijndael_S-box&oldid=938796905.
- [5] Wikipedia contributors. *Scalability* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 5-February-2020]. 2020. URL: <https://en.wikipedia.org/w/index.php?title=Scalability&oldid=938076363>.
- [6] Wikipedia contributors. *Substitution-permutation network* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 5-February-2020]. 2019. URL: https://en.wikipedia.org/w/index.php?title=Substitution%E2%80%9393permutation_network&oldid=925673615.
- [7] Wikipedia contributors. *Symmetric-key algorithm* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 5-February-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=Symmetric-key_algorithm&oldid=939109108.