COSC 3P93 PROJECT STEP 3

AES Encryption

Delivered by -

- Fahad Ansar
- Paramvir Singh

INTRODUCTION

AES algorithm is an acronym for Advanced Encryption Standard algorithm. Its original name is Rijndael. AES is a subset of the Rijndael block which was developed by two Belgian cryptographers Vincent and Joan. It is based on a design principle known as substitution permutation network which works on both software and hardware.

AES has a fixed size of 128 bits with a key sizes of 128 bits, 192 bits and 256 bits. It operates on a 4x4 matrix which is treated as an array of bytes.
AES has a fixed number of transformation rounds where steps are repeated using the last state of the matrix as an input. There are fixed number of rounds depending on size of the key,

- For 128 bit key its 10 rounds
- For 192 bit key its 12 rounds
- For 256 bit key its 14 rounds

ENCRYPTION PROCESS

AES-128 uses a 128 bit key for encryption.

Key and plain text message are stored as hexadecimal values ranging from 0x00 to 0xff.

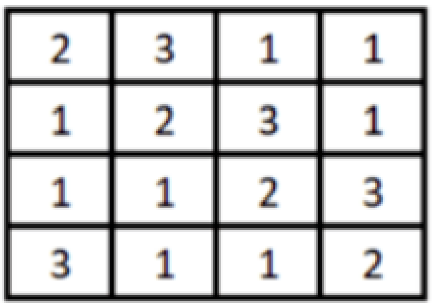
Both key and plain text message are stored as a 4*4 matrix. Plain text message matrix is referred to as a state-matrix while it is in the process of encryption.

We perform a xor operation on the initial key (provided by the user) and the plain text message, before the 1st round. The result of this operation known as the state matrix is passed on.

Each round has 4 operations (is carried out 10 times for AES-128)

- Sub bytes
    - Substitutes the values of the state-matrix from the s-box table.
    - The first 4 bytes of hexadecimal numbers are used to select the row number from the sub-byte table.
    - Another 4 bytes of the hexadecimal numbers used to select the column number from the sub-byte table.
    - For example, 0xa6 means row number a and column number 6.
    - The resultant of this function is passed on to the next function.
- Shift rows
    - All the rows of the state-matrix are shifted towards left
    - The first row is shifted zero times. The second row is shifted twice and so on.
    - The resultant is passed onto the next function.
- Mix columns (is not performed for the 10th round)

    - Each column is multiplied with Rijndael-Mix-Column-Table.



2	3	1	1
1	2	3	1
1	1	2	3
3	1	1	2

Rijndael Mix Column Table

        - The multiplication step consists of addition [XOR] and multiplication [modulo of the polynomial]. Each byte is considered as a polynomial of order x^7.
        - There is a chance of an overflow which occurs in the case when the result of multiplication is larger than OxFF. To tackle this overflow, we perform XOR with Ox1B on that byte

    - Resultant of this step is passed onto the next function.
- Add round key
    - Xor operation is carried out between the state-matrix and the first key from the key-expansion list.

The encrypted text is obtained.

PSEUDO CODE

```
/**
* Used to encrypt a block of text of 128 bits. For message bigger
* than 128 bits we can send multiple blocks of plain text of 128
* bits.
*/

key[4][4] = char_to_hex(user_key)
msg[4][4] = char_to_hex(user_message)

round_keys = key_expansion(key)

state = add_round_key(key, msg)

for i 0 to 9
      state = sub_bytes(state)
```

```
    state = shift_rows(state)
    state = mix_columns(state)
    state = add_round_key(state, round_keys[i])

state = sub_bytes(state)
state = shift_rows(state)
cipher = add_round_key(state)
```

OpenMP

Blocks of plain texts are passed to the thread that is ready to execute and the results are saved in-place. The encrypted text message is achieved once all the blocks of text are encrypted.

Data-Parallelism was utilized because the same AES encryption algorithm was used for all the blocks of text.

There was no coordination amongst threads during the encryption process which decreased the overhead which could have occurred due to coordination amongst threads.

Problem was faced when deciding how to parallelize the program and making sure the results were saved in place.
The solution to the problem was found by dividing the load equally amongst each thread.

There was output dependency between the sub-steps of encryption, which offered no opportunity to parallelize.

There was an idea to parallelize sub-steps, but the size of the sub-steps was very small. Overhead of managing the threads overweight the performance gains, so the idea was dropped.

Default scheduling provided the best results.

OpenMPI

Blocks of plain texts and a key is read from corresponding files and saved in a two dimensional array. Then converting it to one dimensional array to reduce the overhead. And make MPI messages passing better. Then is MPI initialized using *MPI_Init*. Size and rank variables were tied with the channel.

To send the array made a *MPI_Datatype* and also made a buffer to store the messages before sending and after receiving. The array with the whole text is splitted into small chunks and then transferred.

Followed the basic code flow using If and else for separately controlling the master and worker threads. If it's a worker thread, it will pack a message with its share of text i.e. some chunks of text from a buffer which has data stored depending on how many worker threads are used. After packing it will send. As every worker thread sends, the master would receive from every thread individually and stores it in a buffer. After collecting the text from worker threads, master threads saves it in an array and then the program writes the message on a file.

Problems faced were sending text and distribution of text in threads. Later we found out the solution as in using *MPI_Datatype* and a formula i.e. (Number of text block)/(Number of threads).

The only opportunity was to break the text and spread it among the threads. We thought to break the internal small steps of encryption in MPI also but there was an overhead in MPI also.

RUNNING THE OpenMP PROGRAM

Program uses C++ 14 Standard and CMake 3.10.

If you wish to run the code from the console, just go to the directory ***build***, then enter ***make*** to build the project and enter ***./AES <number of blocks> <number of threads>*** to run the application.

RUNNING THE OpenMPI PROGRAM

Program uses C++ 14 Standard.

If you wish to run the code from the console, just go to the directory ***MPI***, then enter ***mpic++ *.cpp****, to compile the code and* ***mpirun --oversubscribe -n <number of processes+1> a.out <number of blocks> <number of processes>***.

For example, mpirun --oversubscribe -n 5 a.out 400000 4. Will run the code with 4 processors.