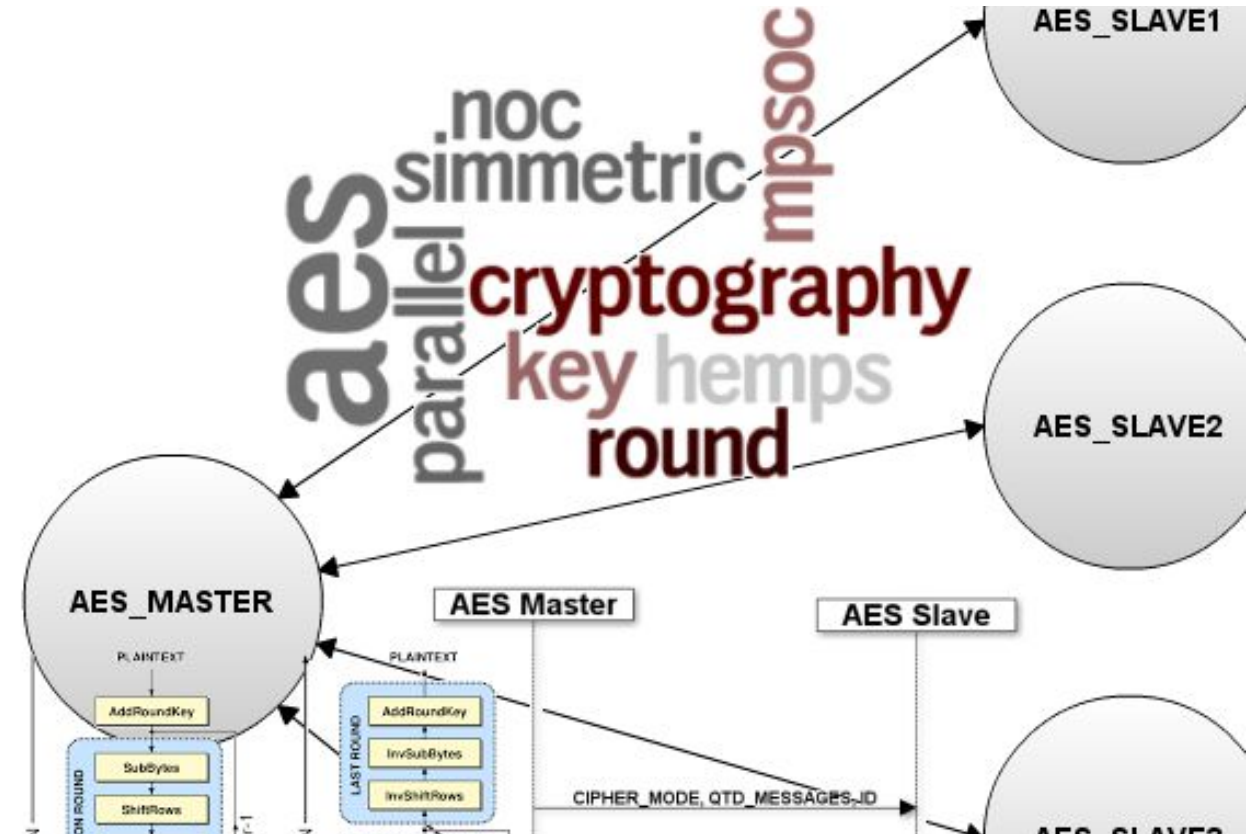

Desenvolvimento da Aplicação AES para HeMPS

PUCRS - FACIN - PPGCC - GAPH - 06.10.2016

Leonardo Rezende e Luciano L. Caimi
Disciplina: Sistemas Integrados em Chip - 2016/01
Professor: Fernando G. Moraes

1 - Introdução

Este relatório apresenta as atividades realizadas no desenvolvimento da aplicação de criptografia usando AES para a plataforma HeMPS (Hermes Multi-Processor System), proposta na disciplina Sistemas Integrados em Chip do PPGCC da PUCRS em 2016/01.

Na implementação desenvolvida foi utilizado o modelo mestre/escravo. O mestre é responsável por quebrar a mensagem em blocos e envia-los para os escravos fazerem a criptografia. Após obter toda a mensagem criptografada, o mestre quebra esta mensagem e envia cada bloco para os escravos fazerem a descriptografia dos mesmos de forma a obter a mensagem original novamente.

Nos experimentos utilizou-se uma mensagem de tamanho 2048 *bytes*, e variou-se a quantidade de escravos entre 1 e 20. Os resultados obtidos mostraram tempos de execução entre 76,8 ms e 10,09 ms (1 e 20 escravos trabalhando, respectivamente), *speedup* máximo 8,32 e eficiência acima de 50% para até 15 escravos.

O relatório está organizado da seguinte maneira: inicialmente será apresentado como executar a aplicação desenvolvida na plataforma; a seguir, na Seção 3, será apresentada uma visão geral da plataforma HeMPS e os modelos de comunicação e de execução de aplicações utilizados na mesma; na Seção 4 é apresentado o sistema criptográfico AES, seu funcionamento e suas características. Na sequência, na Seção 5, a descrição da implementação da aplicação na plataforma é apresentada; finalmente, na Seção 6, são apresentados os resultados da execução da aplicação na HeMPS.

2 - Como executar

Esta aplicação permite parametrizar o tamanho da mensagem, a quantidade de escravos alocados, a quantidade de escravos que a aplicação irá utilizar efetivamente na criptografia, e se o modo de verificação (*debug*) está ativado ou desativado.

Para isso, foi desenvolvido um *shell script* que utiliza como base arquivos **“.c”** e **“.h”** referentes ao código da tarefa mestre e das tarefas escravos. Estes arquivos são copiados e alterados através do comando **sed**. O número de cópias do escravo é dado pelo parâmetro de quantidade de escravos alocados, citado anteriormente.

Para executar o comando, utilizar a seguinte sintaxe:

./aes_generator <MSG_LENGHT> <MAX_SLAVES> <NUMBER_SLAVES> <DEBUG>

Onde:

- **MSG_LENGHT** é o tamanho da mensagem;

- **MAX_SLAVES** é o número máximo de escravos alocados;
- **NUMBER_SLAVES** é o número de escravos que participarão do processo de criptografia e descriptografia na aplicação;
- **DEBUG** ativa ou desativa as mensagem de verificação. Pode assumir valores debug_0/debug_1/debug_2/debug_3, sendo:
  - debug_0: modo debug desativo;
  - debug_1: debug da comunicação entre mestre/escravo ativa;
  - debug_2: debug do algoritmo AES ativo
  - debug_3: debug da comunicação entre mestre/escravo e do algoritmo AES ativo

O exemplo a seguir mostra a geração de uma aplicação com uma mensagem de tamanho 256 *bytes*, com 8 escravos alocados, 6 escravos usados efetivamente e com as mensagens de verificação desativadas:

./generate_aes.sh 256 8 6 debug_0

Com esse comando, serão criados oito arquivos **"aes_slave#.c"**, um **"aes_master.c",** um **"aes_master.h"** e um **"aes.h"**. Entre as diferentes simulações realizadas o usuário ainda pode editar o arquivo **"aes_master.c"** para alterar a quantidade de escravos utilizados, sendo o mínimo um e o máximo, para este exemplo, oito.

Após criar os arquivos da aplicação através do script, deve-se criar um arquivo com extensão .hmp com as configurações da plataforma HeMPS e da aplicação, e executar a simulação nos moldes do procedimento de execução de aplicações da plataforma.

3 - Visão Geral da plataforma HeMPS

A HeMPS (*Hermes Multi-Processor System*) é um MPSOC (*Multiprocessor System on Chip*) baseado em NoC (*Network on Chip*) desenvolvido pelo Grupo de Apoio ao Projeto de Hardware (GAPH). A HeMPS possui processamento homogêneo, memória distribuída e utiliza comunicação por troca de mensagens através de diretivas de *Send* e *Receive*. A Figura 1 apresenta uma visão geral da HeMPS.

O MPSoC contém três tipos de Elementos de Processamento (PEs): o Mestre Global (GM), Mestre Local (CM) e Escravo (SP). O CM é responsável pelo controle do cluster, executando funções como o mapeamento de tarefas, migração de tarefas, monitoramento, verificação de deadlines e comunicação entre outros CMs e o GM. O GM contém todas as funções do CM, e as funções relacionadas com a gerência global do sistema. Um exemplo dessas funções inclui a escolha de qual cluster uma dada aplicação irá ser mapeada, o controle de recursos disponíveis em cada cluster, o recebimento de mensagens de controle e de debug dos CMs, acesso ao repositório de aplicações (memória externa contendo os códigos objetos de todas as tarefas) e recebimento de

requisições de novas aplicações de uma interface externa. Os SPs são responsáveis pela execução das tarefas.

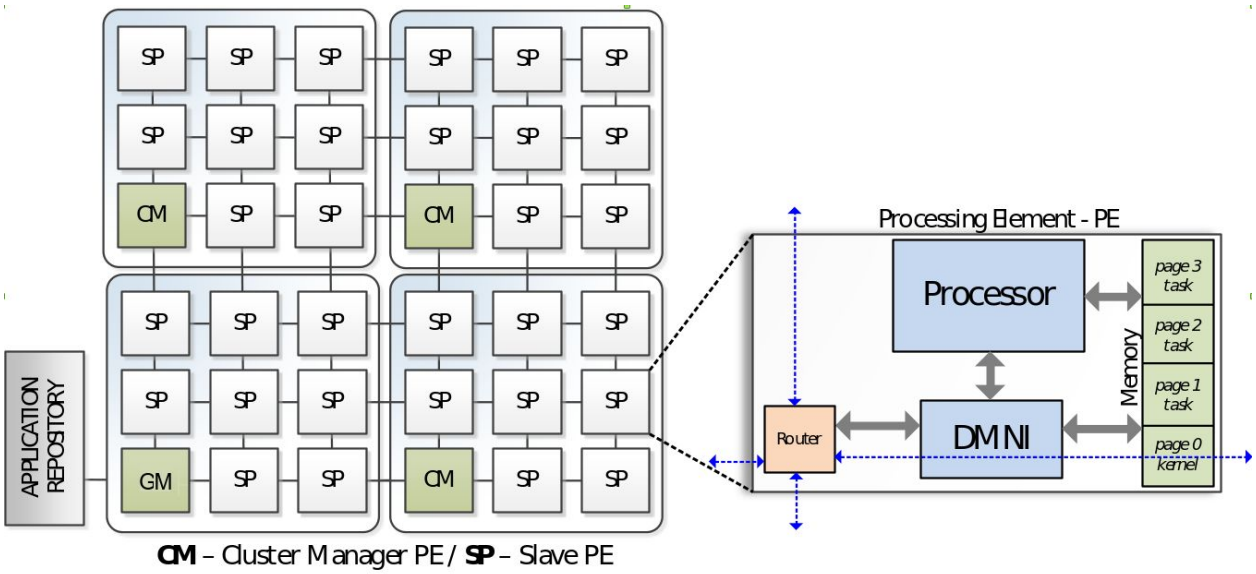

Figura 1: Visão geral da plataforma HeMPS

O repositório de aplicações (*application repository*) é uma memória externa ao MPSoC que contém o código-objeto de todas as tarefas que executarão no sistema. As primeiras posições do repositório de tarefas contêm os descritores de cada tarefa, com informações tais como identificador único, posição inicial no repositório de tarefas, tamanho da tarefa, entre outras informações.

O MPSoC HeMPS assume que todas as aplicações são modeladas através de um grafo de tarefas, onde os vértices representam tarefas e as arestas representam a comunicações entre as tarefas da aplicação. As tarefas sem dependências de recepção de dados são denominadas iniciais. No momento de inicialização de uma dada aplicação, a heurística de mapeamento carrega no MPSoC somente as tarefas iniciais. As demais tarefas são inseridas dinamicamente no sistema em função das requisições de comunicação e dos recursos disponíveis.

Como colocado, a comunicação entre tarefas no sistema é realizada através de trocas de mensagens utilizando um modelo *MPI-like*. Na API de programação da plataforma são disponibilizadas duas diretivas de comunicação para o desenvolvedor - Send() e Receive(). As diretivas estão implementadas no microkernel do MPSoC HeMPS, de forma que o Send() é assíncrono e o Receive() síncrono.

Desta forma, uma dada mensagem só é injetada na NoC se esta foi requisitada pelo receptor, reduzindo o congestionamento da rede, pois não há a possibilidade de pacotes ficarem bloqueando a rede para serem posteriormente consumidos. Para

implementar um Send() assíncrono, um espaço dedicado de memória no microkernel, chamado pipe, armazena cada mensagem que foi escrita pelas tarefas.

Um consequência desta implementação é que o Receive() é bloqueante e o Send() é não bloqueante, ou seja, no programa do usuário, ao chamar o Send(), a mensagem é escrita no pipe pelo microKernel e o programa continua sua execução. No caso da diretiva Receive(), quando o usuário chama o Receive() o programa fica aguardando até a mensagem esperada chegar.

4 - *Advanced Encryption System* - AES

O *Advanced Encryption Standard* (AES) é um esquema criptográfico de chave secreta (ou chave simétrica), publicado pelo *National Institute of Standards and Technology* (NIST) em novembro de 2001. O algoritmo foi vencedor de uma chamada pública realizada pelo NIST para definição de um novo padrão de esquema criptográfico de chave secreta. Foi proposto por Joan Daemen e Vincent Rijmen, e é padronizado pela norma FIPS PUB 197.

O AES constitui-se uma rede de substituição e permutação que realiza a criptografia sobre blocos de tamanho 128 bits. Assim, quando o tamanho da mensagem a ser criptografada não é múltiplo 128 é necessário o *preenchimento* do bloco. Tal procedimento é chamado de *padding* e há esquemas e padrões específicos a serem observados.

Para desenvolvimento da seção foi utilizada a norma NIST fips-197 (disponível em http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf) e a literatura clássica da área de criptografia tais como Schneier, Menezes, Katz e Stinson. A seguir é apresentada uma descrição sucinta do AES, para o entendimento preciso e discussões detalhadas recomenda-se a leitura do material referenciado.

O bloco é representado como uma matriz de 4 x 4 onde cada elemento é um byte:

$$\begin{matrix} a_{[0,0]} & a_{[0,1]} & a_{[0,2]} & a_{[0,3]} \\ a_{[1,0]} & a_{[1,1]} & a_{[1,2]} & a_{[1,3]} \\ a_{[2,0]} & a_{[2,1]} & a_{[2,2]} & a_{[2,3]} \\ a_{[3,0]} & a_{[3,1]} & a_{[3,2]} & a_{[3,3]} \end{matrix}$$

A Figura 2 apresenta uma visão de alto nível do processo de cifragem e do processo de decifragem do AES, onde cada etapa é um conjunto de operações realizadas sobre o bloco.

O número de rodadas realizada pelo algoritmo (valor N) depende do tamanho da chave utilizada: 10 rodadas para chave de 128 bits; 12 rodadas para chave de 192 bits e 14 rodadas para chave de 256 bits.

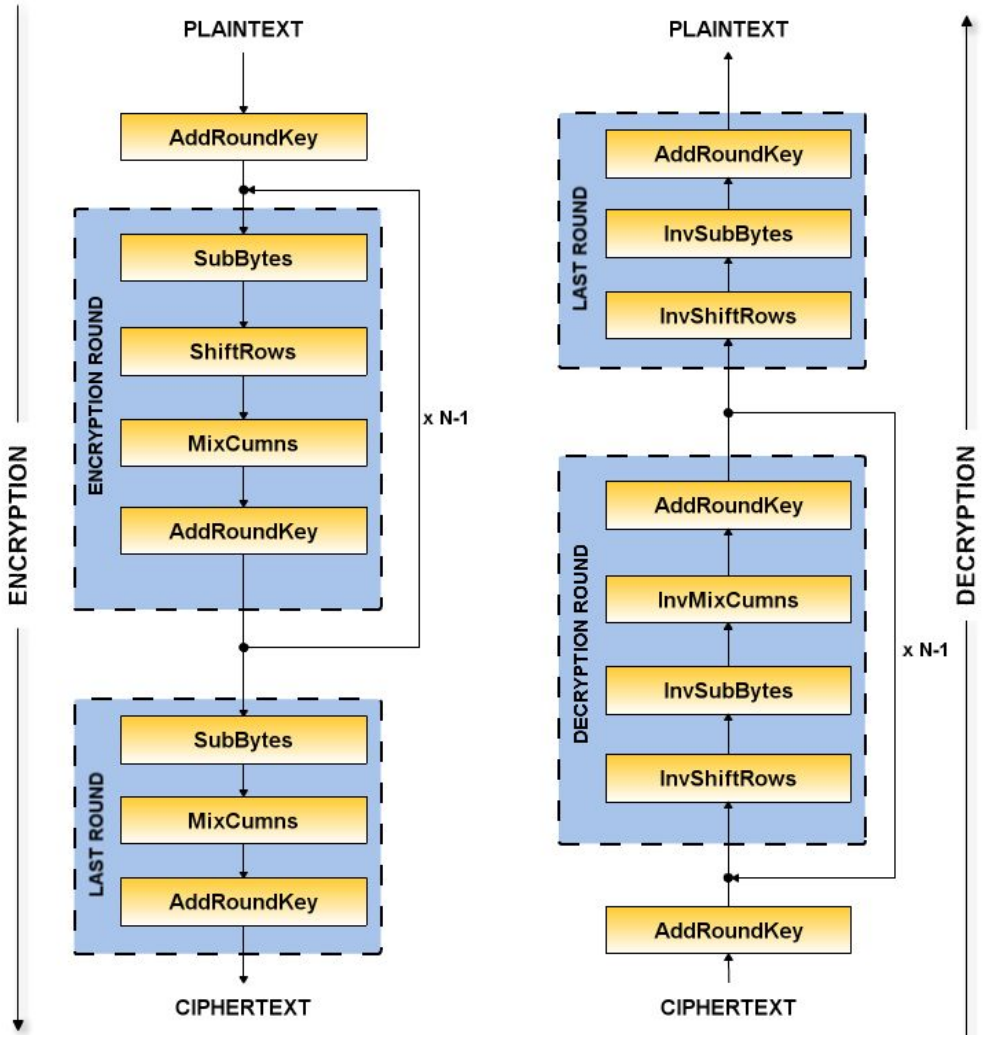

Figura 2: AES

Abaixo é descrito de forma resumida as operações realizadas sobre a matriz em cada etapa:

- **SubBytes:** este passo realiza a substituição de cada um dos bytes de entrada usando blocos de substituição (*S-boxes*). As *S-boxes* constituem funções não-lineares aplicadas em cada byte do bloco. Ao final desta etapa nenhum bloco permanece inalterado;
- **ShiftRows:** este passo realiza a transposição de bytes. Cada uma das linhas da matriz é deslocada de forma circular para a esquerda por 0, 1, 2 e 3 posições, respectivamente;
- **MixColumns:** este passo realiza uma transformação linear em cada coluna da matriz. Esta transformação utiliza operações de deslocamento para esquerda e a operação OU exclusivo apenas;

- **AddRoundKey:** neste passo a sub-chave da rodada é combinada com o estado atual. Para cada rodada, uma sub-chave é derivada da chave principal usando o escalonamento de chaves do Rijndael; cada sub-chave é do mesmo tamanho que o estado (128, 192 ou 256 bits). A sub-chave é somada combinando cada *byte* do estado com o *byte* correspondente da sub-chave, utilizando a operação OU exclusivo;

Como visto, enquanto cifra de bloco, o AES trabalha com bloco de tamanho fixo (16 bytes), desta forma quando a mensagem possui tamanho maior que o tamanho do bloco a mesma deve ser quebrada em blocos de tamanho 16 bytes.

Os modos de operação permitem que a criptografia em blocos forneça confidencialidade para mensagens de tamanho arbitrário pois mesmo que uma mensagem seja criptografada com a mesma chave mais de uma vez produzirá diferentes mensagens cifradas.

Os diferentes modos de operação apresentam diferentes maneiras de combinar os blocos da mensagem a ser cifrada, a chave criptográfica e um vetor de inicialização (ou contador). Os modos de operação clássicos são: (a) ECB - Electronic CodeBook; (b) CBC - Cipher-Block Chaining; (c) CFB - Cipher Feedback; (d) OFB - Output Feedback; (e) CTR - Counter.

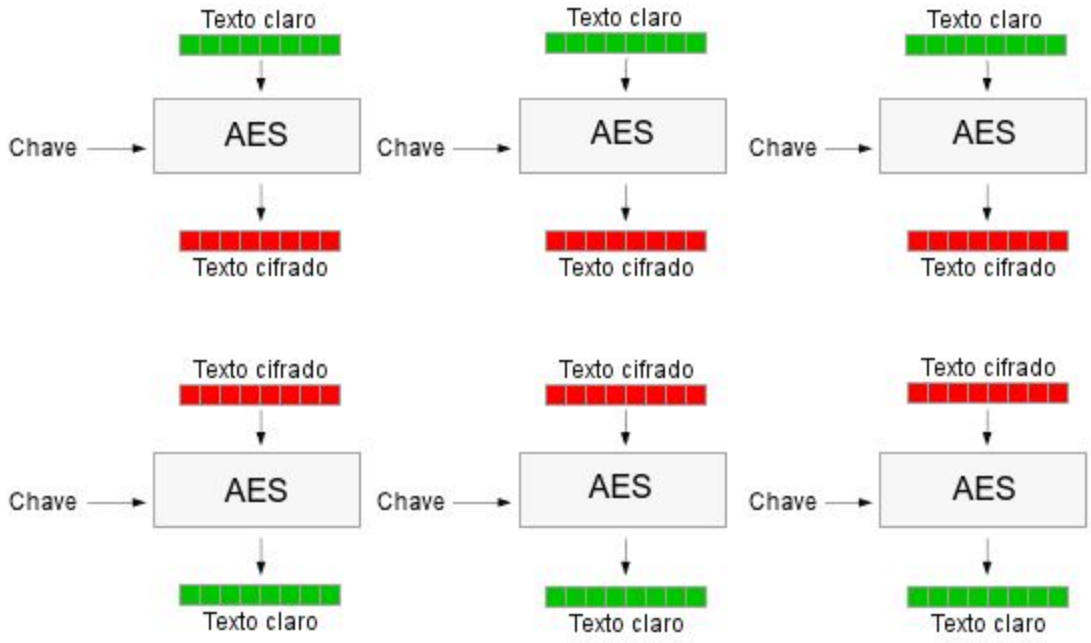

Figura 3: Modo de operação ECB

Apesar de amplamente utilizados, os modos de operação tais como CBC, OFB e CFB preveem somente confidencialidade e não asseguram a integridade da mensagem. Outros modos de operação foram desenvolvidos para assegurar tanto a

confidencialidade quanto integridade da mensagem, tais como os modos de operação IAPM, CCM, EAX, GCM e OCB.

O modo de operação mais simples é o ECB que consiste unicamente em dividir a mensagem em blocos e criptografá-los individualmente, sem a utilização de um vetor de inicialização ou um contador. Por apenas dividir a mensagem, este modo não garante confidencialidade nem autenticidade não sendo usado em aplicações práticas. A Figura 3 ilustra o funcionamento do modo de operação ECB.

Dado sua simplicidade, foi utilizado o modo de operação ECB na implementação da aplicação.

5 - Implementação

O modelo de paralelismo utilizado para modelar está aplicação foi o mestre-escravo. Este modelo consiste em utilizar uma tarefa para coordenar a distribuição da carga de trabalho a ser realizado (mestre) e **N** tarefas responsáveis por fazer o processamento das informações recebidas (escravos). A Figura 4 ilustra o comportamento descrito acima em forma de grafo.

O **AES_MASTER** é reponsável por gerar a mensagem de acordo com o tamanho especificado pelo usuário, distribui-la entre os escravos, aguardar a resposta dos escravos e juntar os blocos da mensagem processada (código mostrado no Anexo 1). Para facilitar a geração e a verificação do resultado, a mensagem gerada é uma sequência de letras que se repete 16 vezes. O exemplo a seguir mostra uma mensagem gerada pela aplicação de 48 bytes:

AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCC

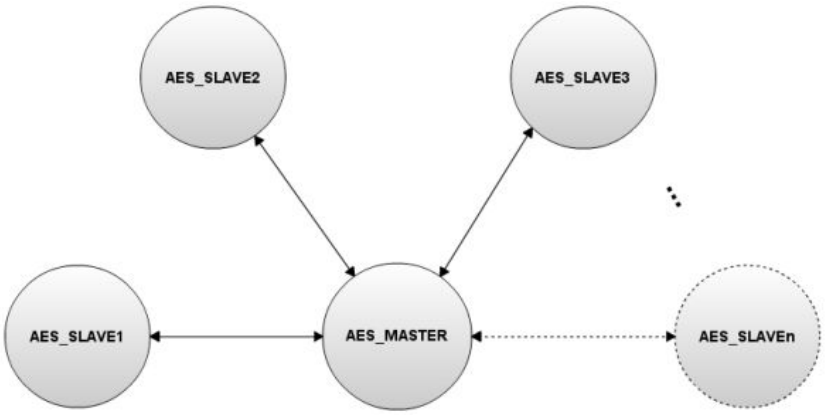

Figura 4: Grafo da aplicação AES

Como o AES trabalha com blocos de 16 bytes, a mensagem é dividida em blocos com este tamanho e os blocos são enviados para os escravos. Dado o formato da mensagem gerada, cada escravo recebe uma letra repetida 16 vezes.

A mensagem é distribuida entre os escravos de forma circular, ou seja, para uma apllicação com nove blocos e oito escravos, os escravos **AES_SLAVE2** até **AES_SLAVE8** receberão um bloco para processar, enquanto o escravo **AES_SLAVE1** receberá dois blocos. Se o tamanho da mensagem gerada for de dez blocos (160 *bytes*) os escravos **AES_SLAVE3** até **AES_SLAVE8** receberão um bloco para processar, e os escravos **AES_SLAVE1** e **AES_SLAVE2** receberão dois blocos. O código fonte dos escravos é mostrado no Anexo 2.

Na aplicação desenvolvida a quantidade de escravos instanciados pode ser diferente da quantidade de escravos que realmente receberão uma carga de trabalho. Assim, os escravos alocados que não são utilizados recebem uma mensagem do mestre informando que eles não receberão carga de trabalho. Ao receber esta mensagem o escravo encerra o seu processamento e a tarefa é finalizada.

A Figura 5 ilustra a comunicação entre a tarefa mestre e uma tarefa escravo na forma de um diagrama de sequência. Primeiramente, o mestre envia uma mensagem para o escravo contendo o modo de operação, a quantidade de mensagens, e o ID do escravo. O modo de operação indica em que modo o algoritmo do AES irá operar, podendo ser entre criptografia, descriptografia ou encerramento da tarefa. A quantidade de mensagens informa quantas vezes a tarefa escravo irá realizar a operação (a quantidade de blocos que ele irá receber). O ID é o identificador do escravo.

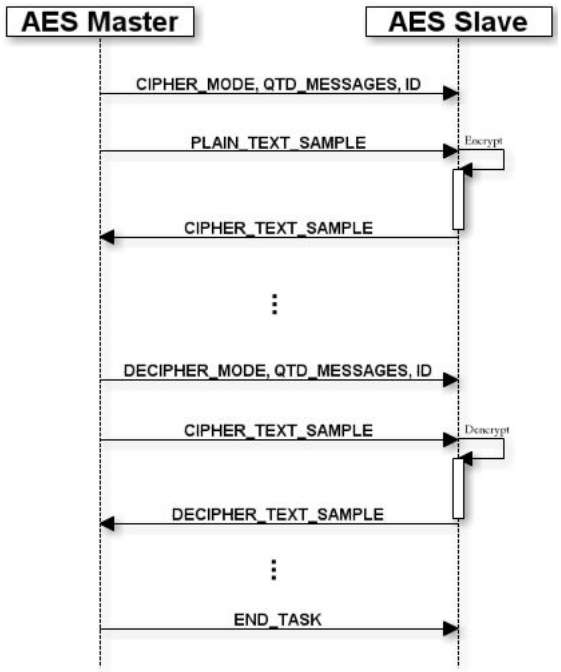

Figura 5: Diagrama de sequência da aplicação AES

Após receber a mensagem informando o modo de operação e a quantidade de blocos a ser processado o escravo executará repetidamente a sequencia: a) recebe o bloco; b) processa o bloco conforme o modo de operação; c) envia o resultado. Está sequencia se repete de acordo com o número de blocos a ser processado pelo escravo informado na primeira mensagem. Após isto o escravo aguarda uma nova mensagem, com o mesmo formato da primeira (modo de operação, quantidade de blocos, ID) para realizar um novo processamento ou encerrar (caso o modo de operação indicar).

Algoritmo 1: AES_Slave

```
1.  do{
2.    receive(operation, number_of_blocks, ID);
3.    for(processed_blocks < number_of_blocks){
4.        receive(block);
5.        if(operation == encrypt)
6.            result = encrypt(block);
7.        if(operation == decrypt)
8.            result = decrypt(block);
9.        send(result);
10.       processed_blocks++;
11.    }
12.   }while (operation <> END)
```

Na aplicação desenvolvida foi utilizado uma chave fixa, definida no **AES_SLAVE** de tamanho 256 bits.

O algoritmo AES utilizado como base para o desenvolvimento da aplicação paralela está disponível em https://github.com/B-Con/crypto-algorithms, sendo de domínio público e sem qualquer tipo de restrição ou licença. Consiste em uma implementação basica do AES, projetado com fins didáticos. Sua implementação não visa performance ou otimização de tamanho, e foi validado utilizando vetores de teste padrões.

Como visto na Seção 2, também foi desenvolvido um *shell script* para automatização da aplicação, que permite configurar o tamanho da mensagem, quantidade de escravos alocados e a quantidade de escravos efetiva que farão a operação de criptografia ou decriptografia. O código deste *script* encontrasse no Anexo 3.

6 - Resultados

Em termos de tamanho do código executável gerado a tarefa **AES_MASTER** possui tamanho 5,5KB, e as tarefas **AES_SLAVE** possuem tamanho 16,9KB (sendo 2KB de dados estáticos). Como comparação, a aplicação MPEG possui as tarefas IDCT, IQUANT, IVLC e START com tamanhos respectivamente de 2,25KB, 0,8KB, 1,8KB e 0,9KB. Tal resultado demonstra que a aplicação AES ocupa grande área de código, requerendo bastante esforço algoritmico no seu processamento.

Para avaliar a robustez e a performance da aplicação desenvolvida na plataforma HeMPS foi criado um conjunto de experimentos utilizando a seguinte configuração:

- Dimensões da HeMPS: 10x10
- Tamanho do Cluster: 5x5
- Tamanho da mensagem: 2048 *bytes*
- Número de escravos alocados: 20
- Tamanho da chave AES: 256 bits

A avaliação variou a quantidade de escravos efetivamente realizando processamento de criptografia e descriptografia. A tabela 1 mostra os tempos de execução da aplicação, o speedup e a eficiência da aplicação em função do número de escravos.

#Escravos	Tempo de execução (ms)	Speedup	Eficiência
1	75.87	1.00	1.00
2	40.26	1.88	0.94
3	28.56	2.66	0.89
4	22.48	3.37	0.84
5	19.20	3.95	0.79
6	17.01	4.46	0.74
7	15.43	4.92	0.70
8	13.69	5.54	0.69
9	13.23	5.73	0.64
10	12.14	6.25	0.63
11	11.65	6.51	0.59
12	11.13	6.82	0.57
13	10.66	7.12	0.55
14	10.70	7.09	0.51
15	10.10	7.51	0.50
16	9.52	7.97	0.50

17	9.59	7.91	0.47
18	9.39	8.08	0.45
19	9.12	8.32	0.44
20	9.11	8.32	0.42

Tabela 1: Resultados do experimento para mensagem de 2Kbytes

O cálculo de *speedup* foi realizado utilizando a seguinte fórmula:

$$Speedup = \frac{tempo\ de\ execução\ sequencial_{\ 1}}{tempo\ de\ execução\ paralela_{\ n}}$$

Onde o tempo de execução sequencial corresponde ao tempo de execução da aplicação utilizando um único escravo e o tempo de execução paralela corresponde ao tempo de execução da aplicação utilizando *n* escravos.

O cálculo de eficiência foi realizado utilizando a seguinte fórmula:

$$eficiência = \frac{tempo\ de\ execução\ paralela_{\ n}}{numero\ de\ escravos_{\ n}}$$

Pode se observar que conforme aumentamos a quantidade de escravos operando, o tempo de processamento em cada um deles diminuí. Os valores entre 18 escravos e 20 escravos são muito próximos, o que indica que em algum determinado momento, aumentar o número de escravos não melhoraria a eficiência.

A figura 6 apresenta um gráfico que mostra a relação tempo de execução da aplicação X número de escravos efetivamente processamento.

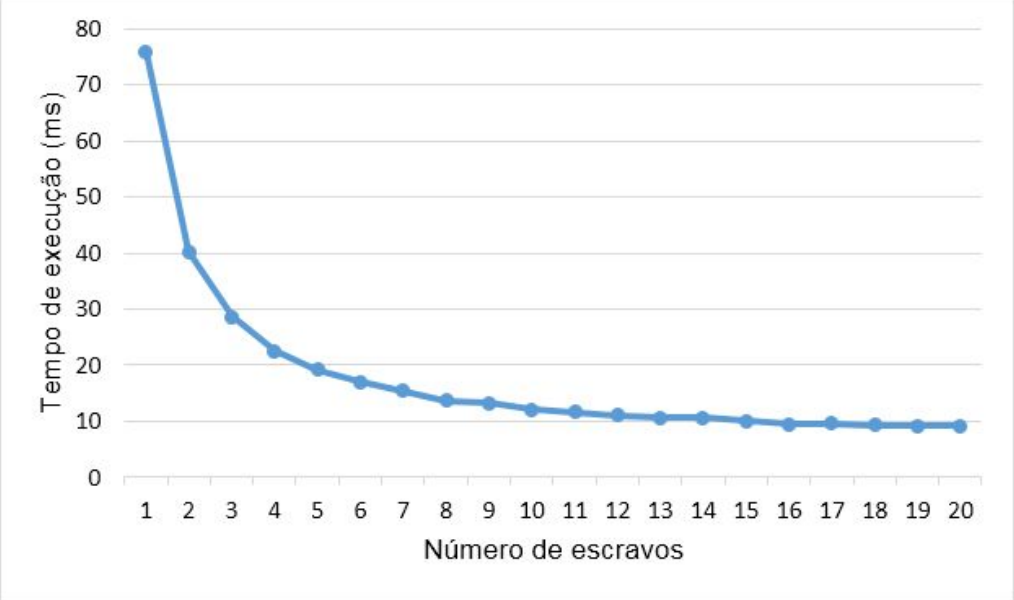

Figura 6:

Como pode-se perceber com até 8 escravos o tempo de execução da aplicação decresce rapidamente (de 75,87 ms para 13,69 ms). Após isto o aumento da quantidade de escravos trabalhando não se refletem em uma diminuição proporcional no tempo de execução.

A figura 7 apresentao *speedup* da aplicação. Percebe-se que conforme aumenta-se o número de escravos, o *speedup* diminui a taxa de crescimento.

Figura 7: *Speedup* da aplicação AES

Essas estabilizações ocorrem devido ao custo de comunicação, já que conforme aumentamos a quantidade de escravos, aumentamos a quantidade de mensagens trocadas entre PEs, o que faz com que não seja possível melhorar mais o tempo de execução, estagnando o valor de *speedup* e diminuindo a eficiência.

7 - Bibliografia

Schneier, B. "Applied cryptography: protocols, algorithms, and source code in C". Wiley Ed., New York, 1996.

Menezes, A. J.; Oorschot, P. C.; Vanstone, S.A.; Rivest R.L. "Handbook of Applied Cryptography". CRC Press, 1997.

AES: Advandec Encryption Standard}. Disponível http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, 2001;

Katz, J.; Lindell, Y. "Introduction to Modern Cryptography", Chapman & Hall/CRC, Boca Raton, 2014.

Stinson, D. R. "Cryptography : theory and practice". Chapman & Hall/CRC, Boca Raton, 2006.

Dworkin, M. NIST - Recommendation for Block Cipher Modes of Operation. Disponível em http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf, 2001.

Anexo 1 - aes_master.c

```
/*********************************************************************
* Filename:   aes_main.c
* Author:       Leonardo Rezende Juracy and Luciano L. Caimi
* Copyleft:
* Disclaimer: This code is presented "as is" without any guarantees.
* Details:
*********************************************************************/

/*************************** HEADER FILES ***************************/
#include <stdlib.h>
#include <task.h>
#include "aes_master.h"
/****************************** DEFINES ******************************/
// total message length
#define MSG_LENGHT
// number of efectived used slaves
#define NUMBER_OF_SLAVES
// number of total slaves allocated
#define MAX_SLAVES

/**************************** VARIABLES *****************************/

//index of slaves (slave names)
int Slave[MAX_SLAVES] = {};
Message msg;

/*************************** MAIN PROGRAM ***************************/

int main()
{
   volatile int x, y, i,j;
   int plain_msg[MSG_LENGHT];
   int cipher_msg[MSG_LENGHT], decipher_msg[MSG_LENGHT];
   int msg_length, blocks, qtd_messages[MAX_SLAVES];
   int pad_value, aux_msg[3];
   int aux1_blocks_PE;
   int aux2_blocks_PE;

   // fill each block with values 'A', 'B', ...
   for(x = 0; x < MSG_LENGHT; x++){
            plain_msg[x] = ((x/16)%26)+0x41;
   }

           Echo("task AES started.");
           Echo(itoa(GetTick()));

   // calculate number of block and pad value (PCKS5) of last block
   msg_length = MSG_LENGHT;
```

```
   blocks = (MSG_LENGHT%AES_BLOCK_SIZE)==0 ? (MSG_LENGHT/AES_BLOCK_SIZE) :
(MSG_LENGHT/AES_BLOCK_SIZE)+1;
   pad_value = (AES_BLOCK_SIZE - (msg_length%AES_BLOCK_SIZE))%AES_BLOCK_SIZE;

   Echo(" ");
   Echo("Blocks:");
   Echo(itoa(blocks));

#ifdef debug_comunication_on
            Echo(" ");
            Echo("plain msg");
            for(x=0; x<MSG_LENGHT-1;x++){
             Echo(itoh(plain_msg[x]));
   }
#endif

   //    Calculate number of blocks/messages to sent
   //   to each Slave_PE
   aux1_blocks_PE = blocks / NUMBER_OF_SLAVES;
   aux2_blocks_PE = blocks % NUMBER_OF_SLAVES;

   ////////////////////////////////////////////////
   //                              Start Encrypt                              //
   ////////////////////////////////////////////////
   for(x = 0; x < MAX_SLAVES; x++){
            qtd_messages[x] = aux1_blocks_PE;
            if(x < aux2_blocks_PE)
                     qtd_messages[x] += 1;
   }

   // Send number of block and operation mode and ID
   // to each Slave_PE
   for(x=0; x < MAX_SLAVES; x++){
            msg.length = sizeof(aux_msg);
            aux_msg[0] = CIPHER_MODE;
            aux_msg[1] = qtd_messages[x];
            aux_msg[2] = x+1;
            if(x >= NUMBER_OF_SLAVES) // zero messages to Slave not used
                     aux_msg[0] = END_TASK;
            memcpy(&msg.msg, &aux_msg, 4*msg.length);
            Send(&msg, Slave[x]);
   }

   // Send blocks to Cipher and
   // Receive the correspondent block Encrypted
   for(x = 0; x < blocks+1; x += NUMBER_OF_SLAVES){
            // send a block to Slave_PE encrypt
            for(y = 0; y < NUMBER_OF_SLAVES; y++){
                     if(qtd_messages[(x+y) % NUMBER_OF_SLAVES] != 0){
                              msg.length = 4*AES_BLOCK_SIZE;
                              memcpy(msg.msg, &plain_msg[(x+y)*AES_BLOCK_SIZE], 4*AES_BLOCK_SIZE);
                              Send(&msg, Slave[(x+y) % NUMBER_OF_SLAVES]);
                     }
            }
```

```
            // Receive Encrypted block from Slave_PE
            for(y = 0; y < NUMBER_OF_SLAVES; y++){
                        if(qtd_messages[(x+y) % NUMBER_OF_SLAVES] != 0){
                                    Receive(&msg, Slave[(x+y) % NUMBER_OF_SLAVES]);
                                    j = 0;
                                    for (i=(x+y)*AES_BLOCK_SIZE;i < ((x+y)*AES_BLOCK_SIZE) + AES_BLOCK_SIZE;
i++)
                                    {
                                                cipher_msg[i] = msg.msg[j];
                                                j++;
                                    }
                                    j = 0;
                                    qtd_messages[(x+y) % NUMBER_OF_SLAVES]--;
                        }
            }
   }
#ifdef debug_comunication_on
   Echo(" ");
   Echo("cipher msg");
   for(i=0; i<MSG_LENGHT;i++){
            Echo(itoh(cipher_msg[i]));
   }
   Echo(" ");
#endif

   ////////////////////////////////////////////////
   //                        Start Decrypt                        //
   ////////////////////////////////////////////////
   for(x = 0; x < NUMBER_OF_SLAVES; x++){
            qtd_messages[x] = aux1_blocks_PE;
            if(x <= aux2_blocks_PE)
                        qtd_messages[x] += 1;
   }

   // Send number of block and operation mode
   // to each Slave_PE
   for(x=0; x < NUMBER_OF_SLAVES; x++){
            msg.length = sizeof(aux_msg);
            aux_msg[0] = DECIPHER_MODE;
            aux_msg[1] = qtd_messages[x];
            memcpy(&msg.msg, &aux_msg, 4*msg.length);
            Send(&msg, Slave[x]);
   }

   // Send blocks to Cipher and
   // Receive the correspondent block Encrypted
   for(x = 0; x < blocks+1; x += NUMBER_OF_SLAVES){
            // send each block to a Slave_PE
            for(y = 0; y < NUMBER_OF_SLAVES; y++){
                        if(qtd_messages[(x+y) % NUMBER_OF_SLAVES] != 0){
                                    msg.length = 4*AES_BLOCK_SIZE;
                                    memcpy(msg.msg, &cipher_msg[(x+y)*AES_BLOCK_SIZE], 4*AES_BLOCK_SIZE);
                                    Send(&msg, Slave[(x+y) % NUMBER_OF_SLAVES]);
```

```
                }
            }
            // Receive Encrypted block from Slave_PE
            for(y = 0; y < NUMBER_OF_SLAVES; y++){
                if(qtd_messages[(x+y) % NUMBER_OF_SLAVES] != 0){
                    Receive(&msg, Slave[(x+y) % NUMBER_OF_SLAVES]);
                    j = 0;
                    for (i=(x+y)*AES_BLOCK_SIZE;i < ((x+y)*AES_BLOCK_SIZE) + AES_BLOCK_SIZE;
i++)
                    {
                        decipher_msg[i] = msg.msg[j];
                        j++;
                    }
                    j = 0;
                    qtd_messages[(x+y) % NUMBER_OF_SLAVES]--;
                }
            }
    }
#ifdef debug_comunication_on
    Echo("decipher msg");
            for(x=0; x<MSG_LENGHT-1;x++){
             Echo(itoh(decipher_msg[x]));
    }
#endif
    //  End tasks still running
    //  End Applicattion
    for(x=0; x < NUMBER_OF_SLAVES; x++){
             msg.length = sizeof(aux_msg);
             aux_msg[0] = END_TASK;
             aux_msg[1] = 0;
             memcpy(&msg.msg, &aux_msg, 4*msg.length);
             Send(&msg, Slave[x]);
    }
            Echo("task AES finished.");
            Echo(itoa(GetTick()));

//#ifdef debug_comunication_on
    Echo(" ");
    Echo("Final Result");
    unsigned int int_aux2 = 0;
            for(x=0; x<MSG_LENGHT;x+=4){
             int_aux2 = decipher_msg[0+x] << 24;
             int_aux2 = int_aux2 | decipher_msg[1+x] << 16;
             int_aux2 = int_aux2 | decipher_msg[2+x] << 8;
             int_aux2 = int_aux2 | decipher_msg[3+x];
             Echo( &int_aux2 );
             int_aux2 = 0;
    }
//#endif

    exit();
}
```

Anexo 2 - aes_slave.c

```
/*********************************************************************
* Filename:   aes_sl(n).c
* Author:       Leonardo Rezende Juracy and Luciano L. Caimi
* Copyleft:
* Disclaimer: This code is presented "as is" without any guarantees.
* Details:
*********************************************************************/

/*************************** HEADER FILES ***************************/
#include <stdlib.h>
#include <task.h>
#include "aes.h"
/*************************** VARIABLES *****************************/

Message msg;

/*************************** MAIN PROGRAM ***************************/

int main()
{
   unsigned int key_schedule[60];
   int qtd_messages, op_mode, x, flag=1, id = -1, i;
   unsigned int enc_buf[128];
   unsigned int input_text[16];
   unsigned int key[1][32] = {

{0x60,0x3d,0xeb,0x10,0x15,0xca,0x71,0xbe,0x2b,0x73,0xae,0xf0,0x85,0x7d,0x77,0x81,0x1f,0x35,0x2c,0
x07,0x3b,0x61,0x08,0xd7,0x2d,0x98,0x10,0xa3,0x09,0x14,0xdf,0xf4}
   };

   Echo(itoa(GetTick()));
          Echo("task AES SLAVE started - ID:");
   aes_key_setup(&key[0][0], key_schedule, 256);

          while(flag){
           Receive(&msg, aes_master);
           memcpy(input_text, msg.msg, 12);

#ifdef debug_comunication_on
   Echo(" ");
   Echo("Slave configuration");
   for(i=0; i<3;i++){
           Echo(itoh(input_text[i]));
   }
   Echo(" ");
#endif

           op_mode = input_text[0];
           qtd_messages = input_text[1];
```

```
		x = input_text[2];

		if(id == -1){
				id = x;
				Echo(itoa(id));
		}
		Echo("Operation:");
		Echo(itoa(op_mode));
		Echo("Blocks:");
		Echo(itoa(qtd_messages));

		if (op_mode == END_TASK){
			flag = 0;
			qtd_messages = 0;
		}
		for(x = 0; x < qtd_messages; x++){
			Receive(&msg, aes_master);
			memcpy(input_text, msg.msg, 4*AES_BLOCK_SIZE);

#ifdef debug_comunication_on
   Echo(" ");
   Echo("received msg");
   for(i=0; i<16;i++){
		Echo(itoh(input_text[i]));
   }
   Echo(" ");
#endif

			if(op_mode == CIPHER_MODE){
				Echo("encript");
				aes_encrypt(input_text, enc_buf, key_schedule, KEY_SIZE);
			}
			else{
				Echo("decript");
				aes_decrypt(input_text, enc_buf, key_schedule, KEY_SIZE);
			}
			msg.length = 4*AES_BLOCK_SIZE;
			memcpy( msg.msg, enc_buf,4*AES_BLOCK_SIZE);
			Send(&msg, aes_master);
		}
   }
		Echo("task AES SLAVE finished  - ID: ");
		Echo(itoa(id));
		Echo(itoa(GetTick()));

   exit();
}
```

Anexo 3 - aes_generator.sh

```
#!/bin/bash

MSG_LENGHT=$1
MAX_SLAVES=$2
NUMBER_OF_SLAVES=$3
DEBUG="debug_0"

echo "Creating a AES application..."
echo "--MSG_LENGHT: ${MSG_LENGHT}"
echo "--MAX_SLAVES: $2"
echo "--NUMBER_OF_SLAVES: $3"
echo "--DEBUG: ${4^^}"

if [ $# -gt 3 ]; then
   DEBUG=$4
fi

rm -rf ../aes_*.c

echo " "
echo "Creating main..."

string="int Slave[MAX_SLAVES] = {"
aux=""
for i in `seq 1 $MAX_SLAVES`
do
   if [ $i -eq $MAX_SLAVES ]; then
            aux="aes_slave${i}"
            string+=${aux}
           else
            aux="aes_slave${i},"
            string+=${aux}
           fi
done
aux="};"
string+=${aux}

cp aes_master.c ../aes_master.c
sed -i -e "s/int Slave\[MAX_SLAVES\] = {};/$string/" -e "s/#define MSG_LENGHT/#define MSG_LENGHT
$MSG_LENGHT/" -e "s/#define MAX_SLAVES/#define MAX_SLAVES $MAX_SLAVES/" -e "s/#define
NUMBER_OF_SLAVES/#define NUMBER_OF_SLAVES $NUMBER_OF_SLAVES/" ../aes_master.c

echo "aes_master.c created!"
echo " "

echo "Creating slaves..."
for i in `seq 1 $MAX_SLAVES`
do
            cp aes_sl.c "../aes_slave${i}.c"
```

```
        echo "aes_slave${i}.c created!"
done

echo "Creating files \".h\"..."
cp *.h ./..

if [ "${4^^}" = "DEBUG_1" ]; then
   sed -i -e "s/\/\/#define debug_comunication_on/#define debug_comunication_on/" ../aes_master.h
   sed -i -e "s/\/\/#define debug_comunication_on/#define debug_comunication_on/" ../aes.h
elif [ "${4^^}" = "DEBUG_2" ]; then
   sed -i -e "s/\/\/#define debug_aes_on/#define debug_aes_on/" ../aes.h
elif [ "${4^^}" = "DEBUG_3" ]; then
   sed -i -e "s/\/\/#define debug_aes_on/#define debug_aes_on/" ../aes.h
   sed -i -e "s/\/\/#define debug_comunication_on/#define debug_comunication_on/" ../aes.h
   sed -i -e "s/\/\/#define debug_comunication_on/#define debug_comunication_on/" ../aes_master.h
fi

cp aes.cfg ./..
```