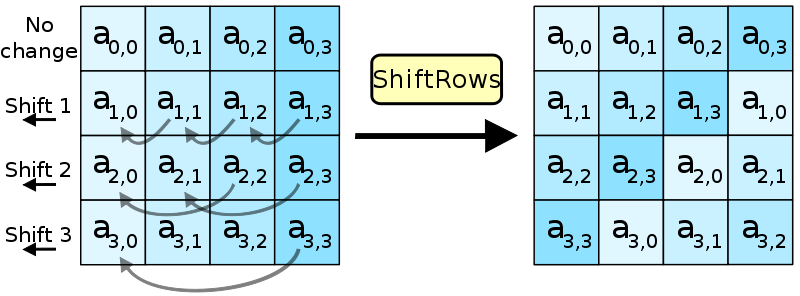
1. Формулировка задачи

Выполнение операции AES ShiftRows над набором из N-матриц. Данные передаются по протоколу TCP.

2. Алгоритм выполнения бизнес-логики

В процедуре ShiftRows байты в каждой строке матрицы циклически сдвигаются влево. Размер смещения байтов каждой строки зависит от её номера. Алгоритм выполнения операции представлен на рисунке 1.

*Рисунок 1 – Алгоритм выполнения операции ShiftRows*

Данный алгоритм применяется для каждой из N матриц.

3. Последовательный вариант исполнения

Исходный код программы-вычислителя представлен в листинге 1.

#include <netdb.h>

#include <netinet/in.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <sys/socket.h>

#include <sys/types.h>

#include <time.h>

#define PORT 8080

#define SA struct sockaddr

int create() {

int sockfd, connfd, len;

struct sockaddr\_in servaddr, cli;

// socket create and verification

sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

if (sockfd == -1) {

printf("socket creation failed...\n");

exit(0);

}

else

printf("Socket successfully created..\n");

if (setsockopt(sockfd, SOL\_SOCKET, SO\_REUSEADDR, &(int){1}, sizeof(int)) < 0)

printf("setsockport(SO\_REUSEADDR) failed \n");

bzero(&servaddr, sizeof(servaddr));

// assign IP, PORT

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

servaddr.sin\_port = htons(PORT);

// Binding newly created socket to given IP and verification

if ((bind(sockfd, (SA\*)&servaddr, sizeof(servaddr))) != 0) {

printf("socket bind failed...\n");

exit(0);

}

else

printf("Socket successfully binded..\n");

// Now server is ready to listen and verification

if ((listen(sockfd, 5)) != 0) {

printf("Listen failed...\n");

exit(0);

}

else

printf("Server listening..\n");

len = sizeof(cli);

// Accept the data packet from client and verification

connfd = accept(sockfd, (SA\*)&cli, &len);

if (connfd < 0) {

printf("server acccept failed...\n");

exit(0);

}

else

printf("server acccept the client...\n");

return connfd;

}

void func(int sockfd)

{

int n = 0;

read(sockfd, &n, sizeof(n));

int width = 0;

read(sockfd, &width, sizeof(width));

int height = 0;

read(sockfd, &height, sizeof(height));

n = ntohl(n);

width = ntohl(width);

height = ntohl(height);

printf("Received header:\n Number of matrices = %d\n Width of each = %d\n Height of each = %d\n", n, width, height);

u\_char \*in = malloc(sizeof(u\_char) \* n \* width \* height);

u\_char \*out = malloc(sizeof(u\_char) \* n \* width \* height);

u\_char buff[height \* width];

for (size\_t i = 0; i < n; i++) {

read(sockfd, in + i \* height \* width, sizeof(buff));

}

struct timeval start, end;

gettimeofday(&start, NULL);

process(n, height, width, in, out);

gettimeofday(&end, NULL);

double delta = ((end.tv\_sec - start.tv\_sec) \* 1000000u +

end.tv\_usec - start.tv\_usec) / 1.e6;

printf("\nElapsed: %lf ms\n", delta);

printf("Result send.\n");

printf("Sending result...\n");

for (size\_t i = 0; i < n; i++) {

write(sockfd, out + i \* height \* width, sizeof(buff));

}

write(sockfd, &delta, sizeof(delta));

printf("Result send.\n");

free(in);

free(out);

}

void process(int n, int height, int width, u\_char \*in, u\_char \*out) {

printf("Start calculations\n");

int i, j, k;

for(i = 0; i < n; i++) {

for(j = 0; j < height; j++) {

// printf("i = %d, j= %d, threadId = %d \n", i, j, omp\_get\_thread\_num());

for(k = 0; k < width; k++) {

int current = i \* width \* height + j \* width + k;

if (k + j < width) {

out[current] = in[current + j];

} else {

out[current] = in[current + j - width];

}

}

}

}

printf("Calculations complete\n");

}

// Driver function

int main()

{

int sockfd;

sockfd = create();

func(sockfd);

close(sockfd);

}

Исходный код программы-генератора преведён в листинге 2.

#include <netdb.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/socket.h>

#include <stdbool.h>

#include <time.h>

#include <errno.h>

#include <string.h>

#define SA struct sockaddr

u\_char randomByte()

{

return (u\_char) randInRange( 0, 255 );

}

int randInRange( int min, int max )

{

double scale = 1.0 / (RAND\_MAX + 1);

double range = max - min + 1;

return min + (int) ( rand() \* scale \* range );

}

int create(char\* addr, char\* port) {

int sockfd;

struct sockaddr\_in servaddr, cli;

// socket create and varification

sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

if (sockfd == -1) {

printf("socket creation failed...\n");

exit(0);

}

else

printf("Socket successfully created..\n");

bzero(&servaddr, sizeof(servaddr));

// assign IP, PORT

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = inet\_addr(addr);

servaddr.sin\_port = htons(port);

// connect the client socket to server socket

if (connect(sockfd, (SA\*)&servaddr, sizeof(servaddr)) != 0) {

printf("connection with the server failed...\n");

exit(0);

}

else

printf("connected to the server..\n");

return sockfd;

}

void send\_data(int sockfd, int count, int width, int height, int silent)

{

printf("Sending data...");

int n = htonl(count);

write(sockfd, &n, sizeof(n));

int w = htonl(width);

write(sockfd, &w, sizeof(w));

int h = htonl(height);

write(sockfd, &h, sizeof(h));

int sz = width \* height;

u\_char buff[sz];

for(size\_t i = 0; i < count; i++)

{

bzero(buff, sizeof(buff));

for(size\_t j = 0; j < sz; j++)

{

buff[j] = randomByte();

if (!silent) {

if (j % width == 0) {

printf("\n");

}

if (j % sz == 0) {

printf("\n");

}

printf("\t%x", buff[j]);

}

}

write(sockfd, buff, sizeof(buff));

}

printf("\nData have been send.");

}

double receive\_data(int sockfd, int count, int width, int height, int silent) {

int size = width \* height;

u\_char buff[size];

bzero(buff, sizeof(buff));

printf("\nReceiving result...");

u\_char \*in = malloc(sizeof(u\_char) \* count \* width \* height);

for(size\_t i = 0; i < count; i++)

{

read(sockfd, in + i \* size, sizeof(buff));

if (!silent) {

for(size\_t j = 0; j < size; j++) {

if (j % width == 0) {

printf("\n");

}

if (j % size == 0) {

printf("\n");

}

printf("\t%x", in[i \* size + j]);

}

}

}

double time = 0;

read(sockfd, &time, sizeof(time));

printf("\nResult received.");

free(in);

return time;

}

void save\_report(char\* file\_name, long int size\_bytes, double time\_ms) {

FILE \*file;

file = fopen(file\_name, "ab");

if (!file) {

printf("something went wrong: %s", strerror(errno));

exit(1);

}

fprintf(file, "%ld:%lf\n", (long int) size\_bytes, (double) time\_ms);

fclose(file);

printf("Report saved.\n");

}

int main(int argc, char \*\*argv)

{

int sockfd;

char\* host = argv[1];

int port = atoi(argv[2]);

int count = atoi(argv[3]);

int height = atoi(argv[4]);

int width = atoi(argv[4]);

char\* file\_name = argv[5];

bool silent = false;

if (argv[6])

silent = true;

sockfd = create(host, port);

clock\_t begin = clock();

send\_data(sockfd, count, width, height, silent);

double time = receive\_data(sockfd, count, width, height, silent);

clock\_t end = clock();

printf("\nElapsed: %ld ms", (long int)(end - begin));

close(sockfd);

printf("\nSocket successfully closed.\n");

save\_report(file\_name, count \* height \* width, time);

}

График зависимости времени выполнения от размера данных представлен на рисунке 2. Операция проводилась над N матрицами размером 1024x1024, таким образом N – количество данных в Мб. Использовался процессор Intel(R) Core(TM) i5-8500 CPU @ 3.00GHz. Исходные данные для постороения графиков были взяты из сгенерированного файла results.txt. Содержимое файла results.txt приведено в таблице 1.

Таблица 1. Результаты работы программы с последовательным выполнением



*Рисунок 2 – График зависимости для варианта программы с последовательным исполнением*

4. OpenMP

В качестве начального варианта распараллеливания программы было выбрано поматричное распараллеливание.

Исходный код программы-вычислителя представлен в листинге 3.

#include <netdb.h>

#include <netinet/in.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <sys/socket.h>

#include <sys/types.h>

#include <omp.h>

#include <time.h>

#define PORT 8080

#define SA struct sockaddr

int create() {

int sockfd, connfd, len;

struct sockaddr\_in servaddr, cli;

// socket create and verification

sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

if (sockfd == -1) {

printf("socket creation failed...\n");

exit(0);

}

else

printf("Socket successfully created..\n");

if (setsockopt(sockfd, SOL\_SOCKET, SO\_REUSEADDR, &(int){1}, sizeof(int)) < 0)

printf("setsockport(SO\_REUSEADDR) failed \n");

bzero(&servaddr, sizeof(servaddr));

// assign IP, PORT

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

servaddr.sin\_port = htons(PORT);

// Binding newly created socket to given IP and verification

if ((bind(sockfd, (SA\*)&servaddr, sizeof(servaddr))) != 0) {

printf("socket bind failed...\n");

exit(0);

}

else

printf("Socket successfully binded..\n");

// Now server is ready to listen and verification

if ((listen(sockfd, 5)) != 0) {

printf("Listen failed...\n");

exit(0);

}

else

printf("Server listening..\n");

len = sizeof(cli);

// Accept the data packet from client and verification

connfd = accept(sockfd, (SA\*)&cli, &len);

if (connfd < 0) {

printf("server acccept failed...\n");

exit(0);

}

else

printf("server acccept the client...\n");

return connfd;

}

void func(int sockfd, int n\_threads)

{

int n = 0;

read(sockfd, &n, sizeof(n));

int width = 0;

read(sockfd, &width, sizeof(width));

int height = 0;

read(sockfd, &height, sizeof(height));

n = ntohl(n);

width = ntohl(width);

height = ntohl(height);

printf("Received header:\n Number of matrices = %d\n Width of each = %d\n Height of each = %d\n", n, width, height);

u\_char \*in = malloc(sizeof(u\_char) \* n \* width \* height);

u\_char \*out = malloc(sizeof(u\_char) \* n \* width \* height);

u\_char buff[height \* width];

for (size\_t i = 0; i < n; i++) {

read(sockfd, in + i \* height \* width, sizeof(buff));

}

struct timeval start, end;

gettimeofday(&start, NULL);

process(n, height, width, in, out, n\_threads);

gettimeofday(&end, NULL);

double delta = ((end.tv\_sec - start.tv\_sec) \* 1000000u +

end.tv\_usec - start.tv\_usec) / 1.e6;

printf("\nElapsed: %lf ms\n", delta);

printf("Result send.\n");

printf("Sending result...\n");

for (size\_t i = 0; i < n; i++) {

write(sockfd, out + i \* height \* width, sizeof(buff));

}

write(sockfd, &delta, sizeof(delta));

printf("Result send.\n");

free(in);

free(out);

}

void process(int n, int height, int width, u\_char \*in, u\_char \*out, int n\_threads) {

printf("Start calculations\n");

omp\_set\_dynamic(0);

omp\_set\_num\_threads(n\_threads);

int i, j, k;

#pragma omp parallel shared(in, out) private(i,j,k)

{

#pragma omp for // collapse(2)

for(i = 0; i < n; i++) {

for(j = 0; j < height; j++) {

// printf("i = %d, j= %d, threadId = %d \n", i, j, omp\_get\_thread\_num());

for(k = 0; k < width; k++) {

int current = i \* width \* height + j \* width + k;

if (k + j < width) {

out[current] = in[current + j];

} else {

out[current] = in[current + j - width];

}

}

}

}

}

printf("Calculations complete\n");

}

// Driver function

int main(int argc, char \*\*argv)

{

int sockfd;

int n\_threads = atoi(argv[1]);

sockfd = create();

func(sockfd, n\_threads);

close(sockfd);

}

Таблица 2. Результаты работы программы с использованием OpenMP



*Рисунок 3 – График зависимости для варианта программы с использованием OpenMP*

В результате анализа результатов выполнения программы было установлено, что оптимальное количество потоков равно 8. Для этого количества потоков были проведены эксперименты с различными уровнями распараллеливания: поматричное, построчное, построчное с использование collapse. Результаты в таблице 3 и на рисунке 4. Код в листингах 4 и 5.

Листинг 4. Построчное распараллеливание

void process(int n, int height, int width, u\_char \*in, u\_char \*out, int n\_threads) {

printf("Start calculations\n");

omp\_set\_dynamic(0);

omp\_set\_num\_threads(n\_threads);

int i, j, k;

for(i = 0; i < n; i++) {

#pragma omp parallel shared(in, out, i) private(j,k)

{

#pragma omp for

for(j = 0; j < height; j++) {

// printf("i = %d, j= %d, threadId = %d \n", i, j, omp\_get\_thread\_num());

for(k = 0; k < width; k++) {

int current = i \* width \* height + j \* width + k;

if (k + j < width) {

out[current] = in[current + j];

} else {

out[current] = in[current + j - width];

}

}

}

}

}

printf("Calculations complete\n");

}

Листинг 5. Построчное распараллеливание с использование collapse

void process(int n, int height, int width, u\_char \*in, u\_char \*out, int n\_threads) {

printf("Start calculations\n");

omp\_set\_dynamic(0);

omp\_set\_num\_threads(n\_threads);

int i, j, k;

#pragma omp parallel shared(in, out) private(i,j,k)

{

#pragma omp for collapse(2)

for(i = 0; i < n; i++) {

for(j = 0; j < height; j++) {

// printf("i = %d, j= %d, threadId = %d \n", i, j, omp\_get\_thread\_num());

for(k = 0; k < width; k++) {

int current = i \* width \* height + j \* width + k;

if (k + j < width) {

out[current] = in[current + j];

} else {

out[current] = in[current + j - width];

}

}

}

}

}

printf("Calculations complete\n");

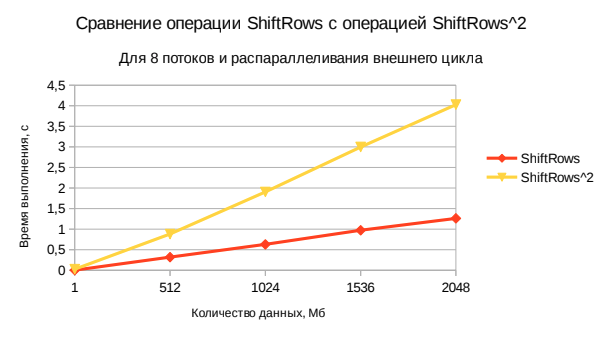
}

Таблица 3. Результаты работы программы с использованием OpenMP для 8 потоков с различными уровнями распараллеливания



*Рисунок 4 – График зависимости для варианта программы с использованием OpenMP для 8 потоков с различными уровнями распараллеливания*

В результате анализа результатов выполнения программы было установлено, что наиболее эффективными видами распараллеливания является поматричное и построчное с использованием collapse, который заранее (до выполнения внешнего цикла) распределяет строки между потоками, в отличии от обычного построчного распараллеливания, которое на каждой итерации внешнего цикла производит распределение строк между потоками.



5. MPI.

Исходный код программы-вычислителя представлен в листинге 6.

#include <netdb.h>

#include <netinet/in.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <sys/socket.h>

#include <sys/types.h>

#include <time.h>

#include <mpi.h>

#define PORT 8080

#define SA struct sockaddr

int create() {

int sockfd, connfd, len;

struct sockaddr\_in servaddr, cli;

// socket create and verification

sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

if (sockfd == -1) {

printf("socket creation failed...\n");

exit(0);

}

else

printf("Socket successfully created..\n");

if (setsockopt(sockfd, SOL\_SOCKET, SO\_REUSEADDR, &(int){1}, sizeof(int)) < 0)

printf("setsockport(SO\_REUSEADDR) failed \n");

bzero(&servaddr, sizeof(servaddr));

// assign IP, PORT

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

servaddr.sin\_port = htons(PORT);

// Binding newly created socket to given IP and verification

if ((bind(sockfd, (SA\*)&servaddr, sizeof(servaddr))) != 0) {

printf("socket bind failed...\n");

exit(0);

}

else

printf("Socket successfully binded..\n");

// Now server is ready to listen and verification

if ((listen(sockfd, 5)) != 0) {

printf("Listen failed...\n");

exit(0);

}

else

printf("Server listening..\n");

len = sizeof(cli);

// Accept the data packet from client and verification

connfd = accept(sockfd, (SA\*)&cli, &len);

if (connfd < 0) {

printf("server acccept failed...\n");

exit(0);

}

else

printf("server acccept the client...\n");

return connfd;

}

void process(int n, int height, int width, u\_char \*in/\*, u\_char \*out\*/) {

u\_char \*out = (u\_char \*) malloc(height \* width \* sizeof(u\_char) \* n);

printf("Start calculations\n width = %d height = %d", height, width);

int i, j, k;

for(i = 0; i < n; i++) {

for(j = 0; j < height; j++) {

for(k = 0; k < width; k++) {

int current = i \* width \* height + j \* width + k;

if (k + j < width) {

out[current] = in[current + j];

} else {

out[current] = in[current + j - width];

}

}

}

}

printf("Calculations complete\n");

//Тестовый вывод

for(size\_t i = 0; i < n; i++)

{

for(size\_t j = 0; j < height \* width; j++) {

if (j % width == 0) {

printf("\n");

}

if (j % height \* width == 0) {

printf("\n");

}

printf("\t%x", out[i \* height \* width + j]);

}

}

}

// Driver function

int main(int argc, char \*\*argv)

{

int sockfd;

int rank, size, rc;

int n, width, height;

u\_char \*in = NULL;

u\_char \*out = NULL;

u\_char buff[height \* width];

if ((rc = MPI\_Init(&argc, &argv)) != MPI\_SUCCESS) {

fprintf(stderr, "Error starting MPI program. Terminating.\n");

MPI\_Abort(MPI\_COMM\_WORLD, rc);

}

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

if (rank == 0) {

sockfd = create();

read(sockfd, &n, sizeof(n));

read(sockfd, &width, sizeof(width));

read(sockfd, &height, sizeof(height));

n = ntohl(n);

width = ntohl(width);

height = ntohl(height);

printf("Received header:\n Number of matrices = %d\n Width of each = %d\n Height of each = %d\n", n, width, height);

in = malloc(sizeof(u\_char) \* n \* width \* height);

out = malloc(sizeof(u\_char) \* n \* width \* height);

for (size\_t i = 0; i < n; i++) {

read(sockfd, in + i \* height \* width, sizeof(buff));

}

}

MPI\_Bcast(&n, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

MPI\_Bcast(&width, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

MPI\_Bcast(&height, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

int p = n / size;

printf("process %d of %d всего матриц %ld количество матриц на поток %ld\n", rank, size, n, p);

printf("%d \n", height \* width \* p);

//получить данные от рута

u\_char \*recvbuf = (u\_char \*) malloc(height \* width \* sizeof(u\_char) \* p);

MPI\_Scatter(in, height \* width \* p, MPI\_UNSIGNED\_CHAR, recvbuf, height \* width \* p,

MPI\_UNSIGNED\_CHAR, 0,

MPI\_COMM\_WORLD);

//что то сделать с данными

process(p, height, width, recvbuf);

MPI\_Gather(recvbuf, height \* width \* p, MPI\_UNSIGNED\_CHAR, out, height \* width \* p,

MPI\_UNSIGNED\_CHAR, 0,

MPI\_COMM\_WORLD);

printf("\nGather complete!\n");

if (rank == 0) {

printf("Sending result...\n");

for (size\_t i = 0; i < n; i++) {

write(sockfd, out + i \* height \* width, sizeof(buff));

}

printf("Result send.\n");

free(in);

free(out);

close(sockfd);

}

MPI\_Finalize();

}

Результаты выполнения в таблице 4 и на рисунке 5.

Таблица 3. Результаты работы программы с использованием MPI для 8 потоков

*Рисунок 4 – График зависимости для варианта программы с использованием MPI для 8 потоков*