Звіт до лабораторної роботи з паралельного програмування

*“Алгоритм шифрування Blowfish”*

Пядика Любомира, МІ-3, 2018 р.

# Постановка задачі

Потрібно було реалізувати алгоритм Blowfish - криптографічний алгоритм, який реалізує блочне симетричне шифрування.

Розроблений Брюсом Шнайєром в 1993 році. Являє собою шифр на основі мережі Фейстеля. Виконано на простих і швидких операціях: XOR, підстановка, додавання. Не запатентований і вільно поширюваний.

Цей алгоритм розпаралелювався за допомогою MPI та OpenMP і порівнювалися результати.

# Опис алгоритму

**Параметри**

Секретний ключ K (від 32 до 448 біт)

32-бітові ключі шифрування P1-P18

32-бітові таблиці замін S1-S4:

S1 [0] S1 [1] .. S1 [255]

S2 [0] S2 [1] .. S2 [255]

S3 [0] S3 [1] .. S3 [255]

S4 [0] S4 [1] .. S4 [255]

**Функція F (x)**

32-бітний блок ділиться на чотири 8-бітних блоки (X1, X2, X3, X4), кожен з яких є індексом масиву таблиці замін S1-S4

Значення S1[X1] і S2[X2] складаються по модулю 2^32, після "XOR"яться з S3[X3] і, нарешті, додаються з S4[X4] по модулю 2^32.

Результат цих операцій — значення F (x).

**Етапи алгоритму**

Розділений на 2 етапи:

Підготовчий — формування ключів шифрування по секретному ключу.

Ініціалізація масивів P і S за допомогою секретного ключа K

Ініціалізація P1-P18 фіксованим рядком, що складається з шістнадцяткових цифр мантиси числа пі.

Проводиться операція XOR над P1 з першими 32 бітами ключа K, над P2 з другими 32-бітами і так далі.

Якщо ключ K коротше, то він накладається циклічно.

Шифрування ключів і таблиць замін

Алгоритм шифрування 64-бітного блоку, використовуючи початкові ключі P1-P18 і таблицю замін S1-S4, шифрує 64 бітну нульовий (0x0000000000000000) рядок. Результат записується в P1, P2.

P1 і P2 шифруються зміненими значеннями ключів і таблиць замін. Результат записується в P3 і P4.

Шифрування триває до зміни всіх ключів P1-P18 і таблиць замін S1-S4.

Шифрування тексту отриманими ключами і F(x), з попереднім розбиттям на блоки по 64 біти. Якщо неможливо розбити початковий текст точно на блоки по 64 біти, використовуються різні режими шифрування для побудови повідомлення, що складається з цілого числа блоків. Сумарні необхідна пам'ять 4168 байт: P1-P18: 18 змінних по 32 біта; S1-S4: 4x256 змінних по 32 бита.

Розшифрування відбувається аналогічно, тільки P1-P18 застосовуються у зворотному порядку.

**Вибір початкового значення P-масиву і таблиці замін**

Немає нічого особливого в цифрах числа пі. Цей вибір полягає в ініціалізації послідовності, не пов'язаної з алгоритмом, яка могла б бути збережена як частина алгоритму або отримана при необхідності (Пі (число)). Як вказує Брюс Шнайєр: «Підійде будь-який рядок з випадкових бітів цифр числа e, RAND-таблиці, або випадкові згенеровані цифри.»

# Отримані результати

iterations=1000000

### Debug ###

input-1024.txt

MPI OpenMP

1: 33.4s

2: 16.9s 17.3s

4: 8.7s 8.6s

8: 6.6s 6.7s

input-2048.txt

MPI OpenMP

1: 65.8s

2: 33.7s 33s

4: 17.7s 17.2s

8: 13.1s 12.4s

### Release ###

input-2048.txt

MPI OpenMP

1: 35s

2: 15.7s 19s

4: 7.8s 9.8s

8: 4.5s 5.5s

# Послідовна програмна реалізація

#define S(x, i) (SBoxes[i][x.w.byte##i])

#define bf\_F(x) (((S(x, 0) + S(x, 1)) ^ S(x, 2)) + S(x, 3))

#define ROUND(a, b, n) (a.dword ^= bf\_F(b) ^ PArray[n])

CBlowFish::CBlowFish()

{

PArray = new DWORD[18];

SBoxes = new DWORD[4][256];

}

CBlowFish::~CBlowFish()

{

delete PArray;

delete[] SBoxes;

}

void CBlowFish::Blowfish\_encipher(DWORD\* xl, DWORD\* xr)

{

union aword Xl, Xr;

Xl.dword = \*xl;

Xr.dword = \*xr;

Xl.dword ^= PArray[0];

ROUND(Xr, Xl, 1); ROUND(Xl, Xr, 2);

ROUND(Xr, Xl, 3); ROUND(Xl, Xr, 4);

ROUND(Xr, Xl, 5); ROUND(Xl, Xr, 6);

ROUND(Xr, Xl, 7); ROUND(Xl, Xr, 8);

ROUND(Xr, Xl, 9); ROUND(Xl, Xr, 10);

ROUND(Xr, Xl, 11); ROUND(Xl, Xr, 12);

ROUND(Xr, Xl, 13); ROUND(Xl, Xr, 14);

ROUND(Xr, Xl, 15); ROUND(Xl, Xr, 16);

Xr.dword ^= PArray[17];

\*xr = Xl.dword;

\*xl = Xr.dword;

}

void CBlowFish::Blowfish\_decipher(DWORD\* xl, DWORD\* xr)

{

union aword Xl;

union aword Xr;

Xl.dword = \*xl;

Xr.dword = \*xr;

Xl.dword ^= PArray[17];

ROUND(Xr, Xl, 16); ROUND(Xl, Xr, 15);

ROUND(Xr, Xl, 14); ROUND(Xl, Xr, 13);

ROUND(Xr, Xl, 12); ROUND(Xl, Xr, 11);

ROUND(Xr, Xl, 10); ROUND(Xl, Xr, 9);

ROUND(Xr, Xl, 8); ROUND(Xl, Xr, 7);

ROUND(Xr, Xl, 6); ROUND(Xl, Xr, 5);

ROUND(Xr, Xl, 4); ROUND(Xl, Xr, 3);

ROUND(Xr, Xl, 2); ROUND(Xl, Xr, 1);

Xr.dword ^= PArray[0];

\*xl = Xr.dword;

\*xr = Xl.dword;

}

void CBlowFish::Initialize(BYTE key[], int keybytes)

{

int i, j;

DWORD data, datal, datar;

union aword temp;

for (i = 0; i < 18; i++)

PArray[i] = bf\_P[i];

for (i = 0; i < 4; i++)

{

for (j = 0; j < 256; j++)

SBoxes[i][j] = bf\_S[i][j];

}

j = 0;

for (i = 0; i < NPASS + 2; ++i)

{

temp.dword = 0;

temp.w.byte0 = key[j];

temp.w.byte1 = key[(j + 1) % keybytes];

temp.w.byte2 = key[(j + 2) % keybytes];

temp.w.byte3 = key[(j + 3) % keybytes];

data = temp.dword;

PArray[i] ^= data;

j = (j + 4) % keybytes;

}

datal = 0;

datar = 0;

for (i = 0; i < NPASS + 2; i += 2)

{

Blowfish\_encipher(&datal, &datar);

PArray[i] = datal;

PArray[i + 1] = datar;

}

for (i = 0; i < 4; ++i)

{

for (j = 0; j < 256; j += 2)

{

Blowfish\_encipher(&datal, &datar);

SBoxes[i][j] = datal;

SBoxes[i][j + 1] = datar;

}

}

}

DWORD CBlowFish::GetOutputLength(DWORD lInputLong)

{

DWORD lVal;

lVal = lInputLong % 8;

if (lVal != 0)

return lInputLong + 8 - lVal;

else

return lInputLong;

}

DWORD CBlowFish::Encode(BYTE\* pInput, BYTE\* pOutput, DWORD lSize)

{

DWORD lOutSize = GetOutputLength(lSize);

for (DWORD lCount = 0; lCount < lOutSize; lCount += 8)

{

if (lCount < lSize - 7)

{

BYTE\* pi = pInput + lCount;

BYTE\* po = pOutput + lCount;

for (int i = 0; i < 8; i++)

\*po++ = \*pi++;

Blowfish\_encipher((DWORD\*)(pOutput + lCount), (DWORD\*)(pOutput + lCount + 4));

}

else

{

DWORD lGoodBytes = lSize - lCount;

BYTE\* pi = pInput + lCount;

BYTE\* po = pOutput + lCount;

int i;

for (i = 0; i < (int)lGoodBytes; i++)

\*po++ = \*pi++;

for (int j = i; j < 8; j++)

\*po++ = 0;

Blowfish\_encipher((DWORD\*)(pOutput + lCount), (DWORD\*)(pOutput + lCount + 4));

}

}

return lOutSize;

}

void CBlowFish::Decode(BYTE\* pInput, BYTE\* pOutput, DWORD lSize)

{

for (DWORD lCount = 0; lCount < lSize; lCount += 8)

{

BYTE\* pi = pInput + lCount;

BYTE\* po = pOutput + lCount;

for (int i = 0; i < 8; i++)

\*po++ = \*pi++;

Blowfish\_decipher((DWORD\*)(pOutput + lCount), (DWORD\*)(pOutput + lCount + 4));

}

}

# Паралельна реалізація за допомогою MPI

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

#include <string>

#include <cstdio>

#include <mpi.h>

#include <chrono>

#include <cstdlib>

#include <cassert>

#include "blowfish.h"

#define MAX\_TEXT\_LEN (10000 + 1)

#define ITERATIONS\_CNT 1000000

using namespace std;

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

{

CBlowFish blowfish;

MPI\_Init(&argc, &argv);

int world\_rank;

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

int world\_size;

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

int elements\_per\_proc = 0;

char text\_ptr[MAX\_TEXT\_LEN];

memset(text\_ptr, 0, MAX\_TEXT\_LEN);

int text\_len = 0;

blowfish.Initialize((BYTE\*)"23452345", 8);

if (world\_rank == 0)

{

printf("Text: ");

flush(cout);

gets\_s(text\_ptr);

//scanf("%s", &text\_ptr);

text\_len = strlen(text\_ptr);

elements\_per\_proc = (text\_len + world\_size - 1) / world\_size;

}

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

MPI\_Bcast(&elements\_per\_proc, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

MPI\_Barrier(MPI\_COMM\_WORLD);

MPI\_Bcast(&text\_ptr, text\_len, MPI\_BYTE, 0, MPI\_COMM\_WORLD);

MPI\_Barrier(MPI\_COMM\_WORLD);

double start = MPI\_Wtime();

//printf("proc %d: %d %d %s\n", world\_rank, text\_len, elements\_per\_proc, text\_ptr);

//flush(cout);

char\* sub\_text = new char[elements\_per\_proc + 1];

assert(sub\_text != NULL);

memset(sub\_text, 0, elements\_per\_proc + 1);

MPI\_Scatter(text\_ptr, elements\_per\_proc, MPI\_BYTE, sub\_text, elements\_per\_proc, MPI\_BYTE, 0, MPI\_COMM\_WORLD);

//printf("rec %d: %d %s\n", world\_rank, strlen(sub\_text), sub\_text);

//flush(cout);

char enc\_sub\_text[MAX\_TEXT\_LEN];

memset(enc\_sub\_text, 0, MAX\_TEXT\_LEN);

char dec\_sub\_text[MAX\_TEXT\_LEN];

memset(dec\_sub\_text, 0, MAX\_TEXT\_LEN);

for (int i = 0; i < ITERATIONS\_CNT; ++i)

{

blowfish.Encode(sub\_text, enc\_sub\_text, strlen(sub\_text));

blowfish.Decode(enc\_sub\_text, dec\_sub\_text, strlen(enc\_sub\_text));

}

char\* result = nullptr;

if (world\_rank == 0)

{

result = new char[MAX\_TEXT\_LEN];

memset(result, 0, sizeof(char) \* (MAX\_TEXT\_LEN));

assert(result != NULL);

}

//printf("gather %d: %d %s\n", world\_rank, strlen(dec\_sub\_text), dec\_sub\_text);

//flush(cout);

MPI\_Gather(dec\_sub\_text, elements\_per\_proc, MPI\_BYTE, result, elements\_per\_proc, MPI\_BYTE, 0, MPI\_COMM\_WORLD);

if (world\_rank == 0)

{

double end = MPI\_Wtime();

printf("Elapsed time: %lf s\n", end - start);

printf("Decoded text: %d %s\n", strlen(result), result);

flush(cout);

}

if (world\_rank == 0)

{

delete[] result;

}

delete[] sub\_text;

MPI\_Finalize();

return 0;

}

# Паралельна реалізація за допомогою OpenMP

#include <iostream>

#include <string>

#include <chrono>

#include <omp.h>

#include "blowfish.h"

#define ITERATIONS\_CNT 10000000

using namespace std;

int main()

{

omp\_set\_dynamic(0);

omp\_set\_num\_threads(8);

CBlowFish blowfish;

blowfish.Initialize((BYTE\*)"23452345", 8);

string text;

cout << "Text: ";

getline(cin, text);

size\_t text\_len = text.size();

size\_t cipher\_len = blowfish.GetOutputLength(text\_len);

string cipher;

cipher.resize(cipher\_len);

string decoded\_text;

decoded\_text.resize(text\_len);

auto start = chrono::system\_clock::now();

#pragma omp parallel for

for (int i = 0; i < ITERATIONS\_CNT; ++i)

{

blowfish.Encode((BYTE\*)text.data(), (BYTE\*)cipher.data(), text\_len);

blowfish.Decode((BYTE\*)cipher.data(), (BYTE\*)decoded\_text.data(), cipher\_len);

}

auto end = chrono::system\_clock::now();

cout << "Elapsed time: " << chrono::duration\_cast<std::chrono::milliseconds>(end - start).count() / 1000. << " s" << endl;

cout << "Cipher: \"" << cipher << "\"" << endl;

cout << "Decoded text: \"" << decoded\_text << "\"" << endl;

system("pause");

return 0;

}