**КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА**

Факультет комп’ютерних наук та кібернетики

Кафедра теорії та технології програмування

**Звіт**

**до лабораторної роботу**

з дисципліни «Розподілене та паралельне програмування»

на тему

«Реалізація афінного шифру за допомогою послідовного й паралельного програмування»

|  |  |
| --- | --- |
| Виконав студент 3-го курсу  Групи ТТП-31  Єріс Євген Євгенович |  |

КИЇВ 2022

**Завдання**

Реалізувати кодування й декодування повідомлення за допомогою афінного шифру. Підготувати три реалізації: послідовну, паралельну з використанням MPI і паралельну з OpenMP. Зробити висновки щодо ефективності реалізацій.

**Алгоритм**

Шифрування й дешифрування відбувається таким чином:

Де a і b – деякі цілі взаємно прості числа, m – довжина алфавіту, a-1 – число, що задовольняє наступному:

Повний код програм знаходиться за посиланням: <https://github.com/YevhenYeris/ParallelAffineCipherLab>

Наведемо реалізації функцій для послідовного шифрування й дешифрування:

void encryption(char\* m, char\*& c, int len) {

for (int i = 0; i < len; i++) {

if (m[i] != ' ') {

c[i] = (char)((((a \* (m[i] - 'A')) + b) % 26) + 'A');

}

else {

c[i] = m[i];

}

}

}

void decryption(char\* c, char\*& m, int len) {

int a\_inverse = 0;

int flag = 0;

for (int i = 0; i < 26; i++) {

flag = (a \* i) % 26;

if (flag == 1) {

a\_inverse = i;

}

}

for (int i = 0; i < len; i++) {

if (c[i] != ' ') {

m[i] = (char)(((a\_inverse \* ((c[i] + 'A' - b)) % 26)) + 'A');

}

else {

m[i] = c[i];

}}}

Для MPI-реалізації використовуються ті ж самі функції, проте їх виконання розпаралелюється за допомогою відповідних функцій MPI:

char\* word;

char\* encoded;

char\* decoded;

int main(int argc, char\* argv[]) {

std::stringstream str(argv[argc - 1]);

int k;

str >> k;

MPI\_Init(&argc, &argv);

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

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

int len = 24 \* k \* 10000000;

if (ProcNum == 0)

{

word = new char[len];

encoded = new char[len];

decoded = new char[len];

GetRandWord(word, len);

}

int part = len / ProcNum;

char\* bufWord = new char[part];

char\* bufEncoded = new char[part];

char\* bufDecoded = new char[part];

if (ProcNum == 0)

{

MPI\_Scatter(word, part, MPI\_CHAR, bufWord, part, MPI\_CHAR, 0, MPI\_COMM\_WORLD);

}

double start = MPI\_Wtime();

encryption(bufWord, bufEncoded, part);

decryption(bufEncoded, bufDecoded, part);

double stop = MPI\_Wtime();

if (ProcNum == 0)

{

MPI\_Gather(bufEncoded, part, MPI\_CHAR, encoded, part, MPI\_CHAR, 0, MPI\_COMM\_WORLD);

MPI\_Gather(bufWord, part, MPI\_CHAR, word, part, MPI\_CHAR, 0, MPI\_COMM\_WORLD);

MPI\_Gather(bufDecoded, part, MPI\_CHAR, decoded, part, MPI\_CHAR, 0, MPI\_COMM\_WORLD);

}

MPI\_Finalize();

if (ProcRank == 0) {

std::cout << len << " " << stop - start << std::endl;

}

return 0;

}

Для OpenMP-реалізації функції шифрування-дешифрування дещо змінені для використання паралельного циклу for:

void encryption(char\* m, char\*& c, int len, int procNum) {

int part = len / procNum;

#pragma omp parallel for

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

{

int nlen = (i + 1) \* part;

for (int j = i \* part; j < nlen; j++) {

if (m[j] != ' ') {

c[j] = (char)((((a \* (m[i] - 'A')) + b) % 26) + 'A');

}

else {

c[j] = m[j];

}

}

}

}

void decryption(char\* c, char\*& m, int len, int procNum) {

int a\_inverse = 0;

int flag = 0;

for (int i = 0; i < 26; i++) {

flag = (a \* i) % 26;

if (flag == 1) {

a\_inverse = i;

}

}

int part = len / procNum;

#pragma omp parallel for

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

{

int nlen = (i + 1) \* part;

for (int j = i \* part; j < nlen; ++j) {

if (c[j] != ' ') {

m[j] = (char)(((a\_inverse \* ((c[j] + 'A' - b)) % 26)) + 'A');

}

else {

m[j] = c[j];

}

}

}

}

**Тестування реалізацій алгоритму**

Для того, щоб отримати більш об’єктивні результати тестування, усі проєкти збираються в Release-конфігурації.



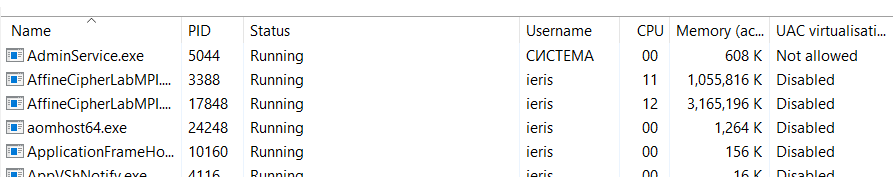
Наведені значення – середні для 10 тестів алгоритму на кожних вхідних даних.

Тестування послідовного алгоритму

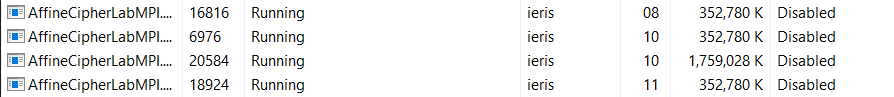
|  |  |
| --- | --- |
| Довжина слова | Середній час виконання |
| 240 000 000 | 1.97881 |
| 480 000 000 | 4.65517 |
| 720 000 000 | 8.02012 |
| 960 000 000 | 10.3173 |
| 1 200 000 000 | 13.4907 |
| 1 440 000 000 | 15.4788 |

Паралельний алгоритм MPI

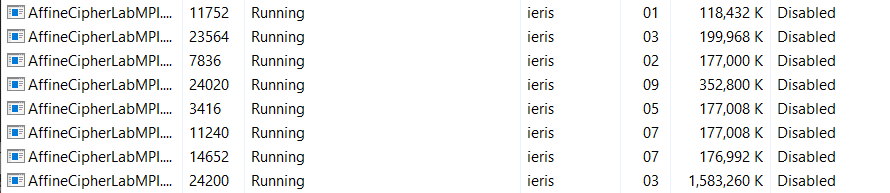
ProcessNum = 2



ProcessNum = 4



ProcessNum = 8



|  |  |  |
| --- | --- | --- |
| Кількість процесів | Довжина слова | Час виконання |
|  | 240 000 000 | 0.6381776 |
|  | 480 000 000 | 1.5497782 |
| 2 | 720 000 000 | 1.957042 |
|  | 960 000 000 | 3.491686 |
|  | 1 200 000 000 | 4.306297 |
|  | 1 440 000 000 | 5.900853 |
|  | 240 000 000 | 0.738065 |
|  | 480 000 000 | 1.27137 |
| 4 | 720 000 000 | 2.0163 |
|  | 960 000 000 | 2.74889 |
|  | 1 200 000 000 | 3.35451 |
|  | 1 440 000 000 | 4.0453 |
|  | 240 000 000 | 0.463297 |
| 8 | 480 000 000 | 0.915492 |
|  | 720 000 000 | 1.49814 |
|  | 960 000 000 | 2.08397 |
|  | 1 200 000 000 | 2.17067 |
|  | 1 440 000 000 | 2.87497 |

Паралельний алгоритм OpenMP



|  |  |  |
| --- | --- | --- |
| Кількість процесів | Довжина слова | Середній час виконання |
|  | 240 000 000 | 2.36703 |
|  | 480 000 000 | 2.43132 |
| 2 | 720 000 000 | 2.09897 |
|  | 960 000 000 | 2.59462 |
|  | 1 200 000 000 | 2.53517 |
|  | 1 440 000 000 | 3.4081 |
|  | 240 000 000 | 1.17893 |
|  | 480 000 000 | 0.952545 |
| 4 | 720 000 000 | 0.867947 |
|  | 960 000 000 | 1.20954 |
|  | 1 200 000 000 | 1.1346 |
|  | 1 440 000 000 | 1.51718 |
|  | 240 000 000 | 1.05185 |
|  | 480 000 000 | 1.16858 |
| 8 | 720 000 000 | 1.0392 |
|  | 960 000 000 | 1.04753 |
|  | 1 200 000 000 | 1.18492 |
|  | 1 440 000 000 | 1.44249 |

Висновки

Для отримання порівняльної характеристики різних підходів до імплементації послідовного й паралельного алгоритму пошуку Афінного шифру були запропоновані реалізації на мові С++ з використанням інтерфейсів MPI та OpenMP та без них.

Порівняння алгоритмів проводилося з використанням середніх значень результатів десяти запусків програми на кожному наборі вхідних даних. Для більшої об’єктивності порівняння всі програми збиралися в Release-конфігурації.

Послідовний алгоритм, очікувано, показав найбільший час виконання та стрімкий приріст часу зі збільшенням розміру вхідних даних. На найменших вхідних даних середній час виконання склав близько 2 секунд, а на найбільшому – більше 15 секунд.

MPI-реалізація показала найкращий результат на найменших вхідних даних. Утім, мала кратний приріст часу виконання з їх збільшенням. Це свідчить про неоптимальність використання MPI для такого розпаралелювання.

OMP-реалізація поступається попередній у роботі з невеликим обсягом даних, проте значно виграє з більшими даними. До того ж, при збільшенні вхідного масиву середній приріст часу виконання склав менше 0.5 секунд.

Виходячи із отриманих результатів, найоптимальнішим для даного алгоритму є використання інтерфейсу OpenMP.

OpenMP для міжпроцесної взаємодії використовує спільну пам’ять, а MPI – обмін повідомленнями. Даний алгоритм не потребує великої кількості обчислень, тому витрати часу на передачу даних у MPI відчутно уповільнюють алгоритм.