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:

$$E(x) = (ax + b)\ mod\ m$$

$$D(x) = a^{-1} * (x - b)\ mod\ m$$

Де a і b – деякі цілі взаємно прості числа, m – довжина алфавіту, a^{-1} – число, що задовольняє наступному: $a * a^{-1}\ mod\ m = 1$

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();
```

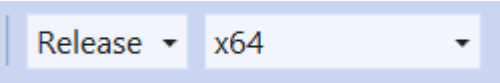
```
    if (ProcRank == 0) {
        std::cout << len << "     " << stop - start << std::endl;
    }

    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

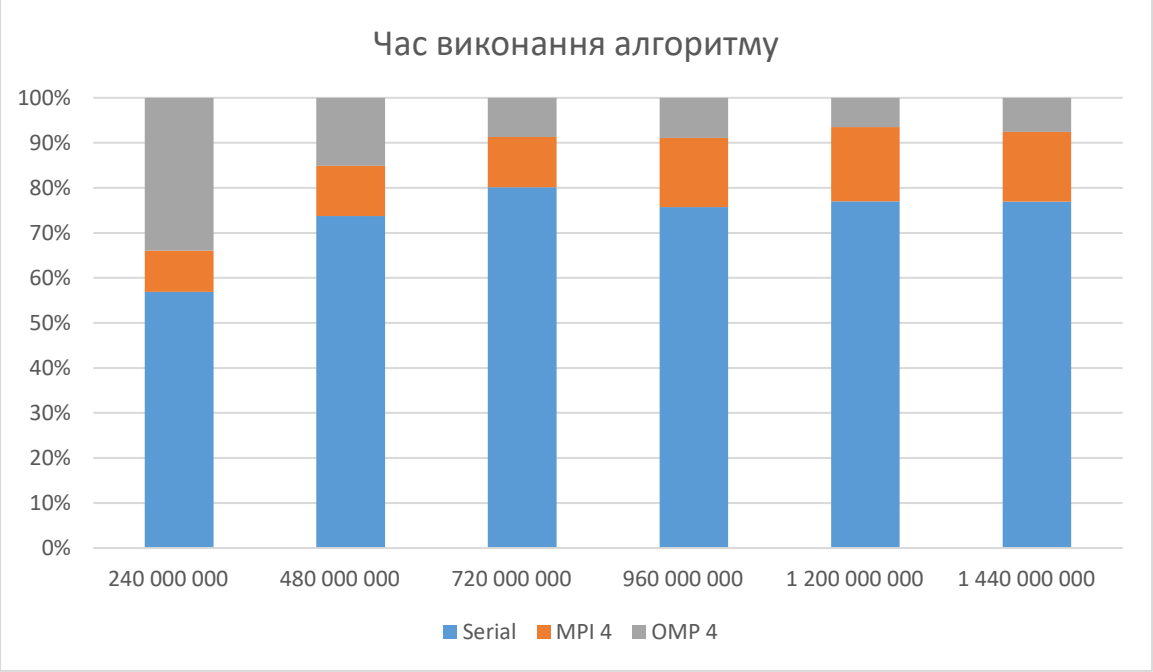
Для того, щоб отримати більш об’єктивні результати тестування, усі проєкти збираються в Release-конфігурації.



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

AffineCipherLabMPI....	3388	Running	ieris	11	1,055,816 K	Disabled
AffineCipherLabMPI....	17848	Running	ieris	12	3,165,196 K	Disabled

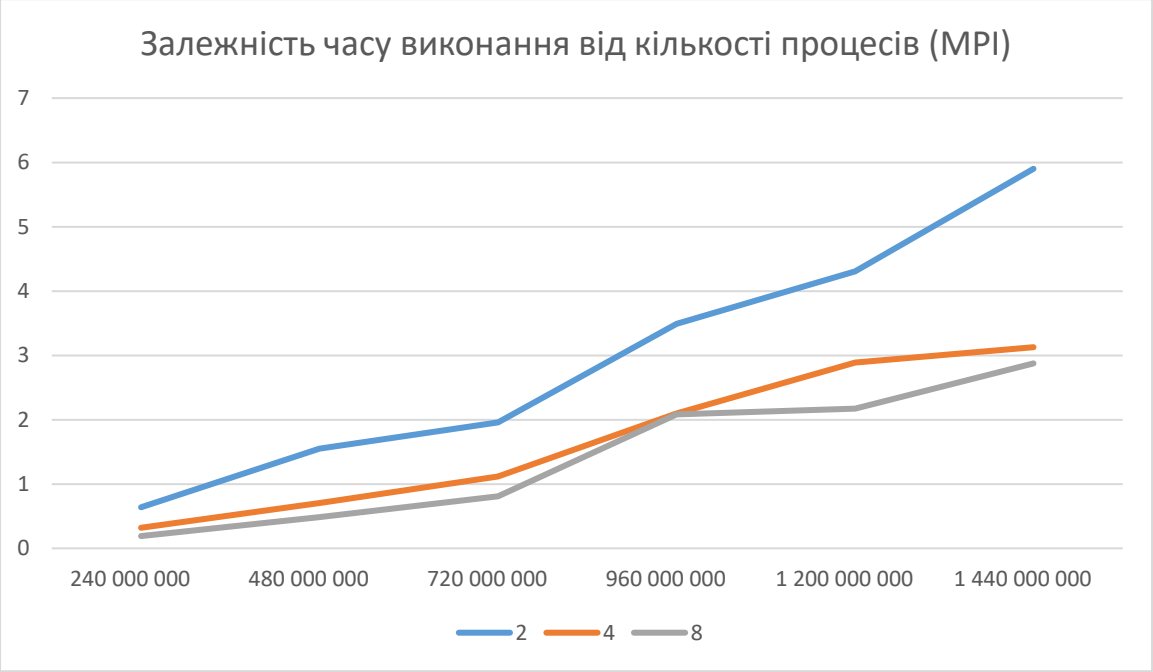
ProcessNum = 4

AffineCipherLabMPI....	16816	Running	ieris	08	352,780 K	Disabled
AffineCipherLabMPI....	6976	Running	ieris	10	352,780 K	Disabled
AffineCipherLabMPI....	20584	Running	ieris	10	1,759,028 K	Disabled
AffineCipherLabMPI....	18924	Running	ieris	11	352,780 K	Disabled

ProcessNum = 8

AffineCipherLabMPI....	11752	Running	ieris	01	118,432 K	Disabled
AffineCipherLabMPI....	23564	Running	ieris	03	199,968 K	Disabled
AffineCipherLabMPI....	7836	Running	ieris	02	177,000 K	Disabled
AffineCipherLabMPI....	24020	Running	ieris	09	352,800 K	Disabled
AffineCipherLabMPI....	3416	Running	ieris	05	177,008 K	Disabled
AffineCipherLabMPI....	11240	Running	ieris	07	177,008 K	Disabled
AffineCipherLabMPI....	14652	Running	ieris	07	176,992 K	Disabled
AffineCipherLabMPI....	24200	Running	ieris	03	1,583,260 K	Disabled

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

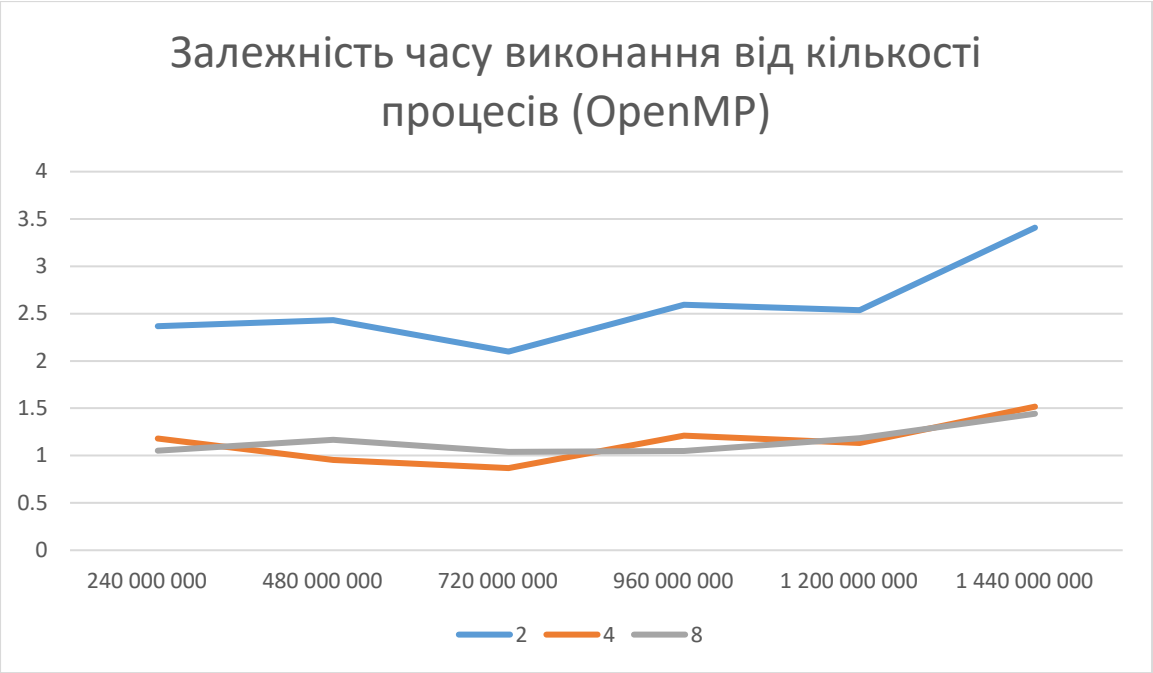


Паралельний алгоритм OpenMP

Name	PID	Status	Username	CPU	Memory (ac...	UAC virtualisati...
AffineCipherLabOMP....	24312	Running	ieris	12	3,860,008 K	Disabled

Кількість процесів	Довжина слова	Середній час виконання
2	240 000 000	2.36703
	480 000 000	2.43132
	720 000 000	2.09897
	960 000 000	2.59462
	1 200 000 000	2.53517
	1 440 000 000	3.4081
4	240 000 000	1.17893
	480 000 000	0.952545
	720 000 000	0.867947
	960 000 000	1.20954
	1 200 000 000	1.1346
	1 440 000 000	1.51718
8	240 000 000	1.05185
	480 000 000	1.16858
	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 C++ 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.