Міністерство освіти і науки України
Харківський національний університет імені В. Н. Каразіна
Навчально-науковий інститут комп'ютерних наук та штучного інтелекту
Кафедра комп'ютерних систем та робототехніки

«Затверджую»
в.о. завідуючого кафедри
комп'ютерних систем та робототехніки
_____ к. ф.-м. н., доцент Максим Хруслов
«___» грудня 2024 р.

Пояснювальна записка

до кваліфікаційної роботи
магістра

на тему: «**МОДЕЛЬ ЗАВАДОСТІЙКИХ КАНАЛІВ ЗВ'ЯЗКУ МОБІЛЬНИХ ПРИСТРОЇВ З ПАРАЛЕЛЬНОЮ ОБРОБКОЮ ІНФОРМАЦІЇ**»

Спеціальність 174 – Автоматизація, комп'ютерно-інтегровані технології та робототехніка
Галузь знань 17 – Електроніка, автоматизація та електронні комунікації.
Освітня програма «Комп'ютеризовані системи управління та автоматика»

Захищено на засіданні
Екзаменаційної комісії № 44
протокол № __ від __.12.2024 р.
Оцінка _____ / _____
Голова Екзаменаційної комісії
_____ **СКОБ Ю. О.**

Виконав:
Студент(ка) групи КУ– 61
ЯРОВЕНКО Михайло Юрійович

Керівник: д.т.н., с.н.с., професор кафедри теоретичної та прикладної системотехніки
ТОЛСТОЛУЗЬКА Олена Геннадіївна

Рецензент: к.т.н., доцент зво кафедри кібербезпеки інформаційних систем, мереж і технологій
МЕЛКОЗЬОРОВА Ольга Михайлівна

Харків – 2024

АНОТАЦІЯ

Пояснювальна записка до магістерської атестаційної роботи складається зі вступу, трьох розділів, висновків, списку використаних джерел і двох додатків. Загальний обсяг роботи складає 87 сторінок, із яких 55 сторінок основної частини з 34 рисунками, списку використаних джерел із 20 найменувань та п’ятью додатками

Метою кваліфікаційної роботи є розробка моделі завадостійких каналів зв’язку мобільних пристроїв із використанням паралельної обробки інформації.

Об’єкт дослідження – процеси підвищення надійності каналів зв’язку в умовах дії перешкод та шумів.

Предмет дослідження – математичні моделі та алгоритми забезпечення завадостійкості передачі даних у мобільних мережах із використанням технологій паралельної обробки інформації.

Проблема, яка вирішується в кваліфікаційній роботі, полягає в розробці методів і підходів, що дозволяють мінімізувати вплив перешкод на якість передачі даних, оптимізувати процес обробки інформації та забезпечити високу пропускну здатність каналів зв’язку.

Область застосування – телекомунікаційні технології. Розроблена модель може бути інтегрована в сучасні системи мобільного зв’язку, що дозволить підвищити їх ефективність і забезпечити високий рівень обслуговування.

Ключові слова: Алгоритми кодування, Алгоритми шифрування, Метод Ріда-Соломона, завадостійкість, циклічний зсув, паралельні обчислення.

ABSTRACT

The explanatory note consists of an introduction, three chapters, conclusions, a list of references, and two appendices. The total volume of the work is 87 pages, including 55 pages of the main body with 34 figures, a reference list of 20 sources, and five appendices.

The goal of this thesis is to develop a model for error-resilient communication channels for mobile devices using parallel information processing.

Object of the study – processes aimed at improving the reliability of communication channels under conditions of interference and noise.

Subject of the study: mathematical models and algorithms to ensure error resilience in data transmission within mobile networks using parallel information processing technologies.

Problem addressed in the thesis – development of methods and approaches to minimize the impact of interference on data transmission quality, optimize the information processing flow, and ensure high throughput of communication channels.

Field of application – telecommunications technologies. The developed model can be integrated into modern mobile communication systems to enhance their efficiency and provide a high level of service.

Keywords: Encoding algorithms, Encryption algorithms, Reed-Solomon method, Noise resistance, Cyclic shift, Parallel computing.

ЗМІСТ

ВСТУП

Мобільний зв'язок є ключовою частиною сучасного життя, але зростання кількості користувачів і даних створює виклики у забезпеченні швидкого та стабільного зв'язку в умовах завад. Розробка завадостійких каналів зв’язку та впровадження паралельної обробки дозволяють підвищити ефективність передачі даних і стійкість до перешкод, що робить ці дослідження актуальними.

Актуальність роботи. Зростаючі обсяги інформації в сучасних системах вимагають ефективних методів обробки та захисту даних.

Метою дослідження є оптимізація алгоритмів шифрування, кодування та обробки тексту для роботи з великими обсягами даних у паралельному режимі з високою точністю та стійкістю до помилок.

Об’єкт дослідження – Процеси шифрування, кодування та декодування тексту в умовах перешкод і високих вимог до продуктивност

Методи дослідження: Алгоритми шифрування (Цезар) та кодування (Рід-Соломон), Паралельна обробка з OpenMP, Математичне моделювання та симуляція помилок.

Предмет дослідження – Точність та продуктивність паралельних алгоритмів шифрування і кодування тексту.

Завдання дослідження Розробити паралельний алгоритм шифрування методом Цезаря. Реалізувати спрощений варіант методу Рід-Соломон.

Моделювати помилки для тестування алгоритмів. Оптимізувати обробку для різної кількості потоків. Оцінити продуктивність і масштабованість системи.

РОЗДІЛ 1
АНАЛІЗ ІСНУЮЧИХ ПІДХОДІВ, ВИКОНАННЯ ВЕБ-ЗАПИТІВ

1.1 Вибір компонентів для розробки моделі

Для розробки моделі завадостійких каналів зв’язку з паралельною обробкою інформації було обрано мову програмування C++, яка є високопродуктивною та забезпечує низькорівневий доступ до апаратних ресурсів. C++ широко використовується для створення складних систем, де потрібна висока швидкість обробки даних, що робить її ідеальним вибором для даного завдання. Мова також підтримує роботу з багатопотоковими програмами, що важливо для паралельної обробки [1].

Аналіз аналогів мов програмування

Серед можливих альтернатив для реалізації завдання можна розглянути такі мови, як Python, Java та Rust.

- **Python** забезпечує простоту розробки та велику кількість бібліотек для роботи з даними, однак через низьку продуктивність і накладні витрати на інтерпретацію коду його використання не є оптимальним для високопродуктивних систем.
- **Java** підтримує багатопоточність і має хорошу продуктивність, але має вищі накладні витрати через використання віртуальної машини (JVM), що зменшує ефективність роботи на пристроях з обмеженими ресурсами, таких як Raspberry Pi.
- **Rust** є перспективною мовою, яка пропонує високу продуктивність і безпеку роботи з пам’яттю. Однак через відносно малу кількість бібліотек і складність у вивченні, її використання для швидкого прототипування менш зручне, ніж C++.

Таким чином, вибір C++ є обґрунтованим завдяки її високій продуктивності, широкому розповсюдженню та підтримці багатопоточності.

Аналіз аналогів апаратних платформ

Як платформу для реалізації обчислювальної частини моделі було обрано одноплатний комп’ютер Raspberry Pi. Це доступний та енергоефективний пристрій, що підтримує різноманітні підключення до мережі та забезпечує можливості для роботи з сенсорами та іншими периферійними пристроями. Raspberry Pi є чудовою платформою для прототипування моделей зв’язку, оскільки він дозволяє здійснювати обчислення в реальному часі та підтримує широке коло інструментів для програмування і налаштування систем зв’язку [2].

Рисунок 1.1 – Raspberry Pi

Серед інших можливих варіантів апаратних платформ:

- **Arduino**: Підходить для роботи з сенсорами, але через обмежену обчислювальну потужність не є оптимальним для виконання складних обчислень у реальному часі.
- **BeagleBone**: Має більшу потужність і розширені можливості, але є дорожчим і складнішим у налаштуванні, що зменшує його привабливість для задач прототипування.
- **Intel NUC**: Забезпечує високу продуктивність, але значно перевищує Raspberry Pi за ціною та споживанням енергії, що робить його менш оптимальним для енергоефективних моделей.

Таким чином, Raspberry Pi був обраний через оптимальне співвідношення ціни, продуктивності та енергоефективності.

Аналіз аналогів технологій паралельної обробки

Для паралельної обробки даних було використано технологію OpenMP (Open Multi-Processing). Вона забезпечує простий та ефективний спосіб реалізації паралельних обчислень на багатоядерних процесорах, що дозволяє збільшити продуктивність системи зв’язку [3].

Можливими альтернативами є:

- **Pthreads**: Надає гнучкість для управління потоками, але вимагає більш складної реалізації, що ускладнює створення прототипу.
- **MPI (Message Passing Interface)**: Ефективно працює для розподілених систем, але додає значні накладні витрати на передачу даних між процесами, що є надмірним для одноплатного комп’ютера.
- **TBB (Threading Building Blocks)**: Пропонує високий рівень абстракції для роботи з потоками, але складніша у використанні для задач з відносно простою структурою паралелізму.

OpenMP забезпечує найкращий баланс між простотою реалізації та продуктивністю, що робить її оптимальним вибором для даної моделі.

Висновок

Обраний підхід (C++ + Raspberry Pi + OpenMP) забезпечує оптимальне співвідношення продуктивності, вартості та простоти реалізації. Така комбінація дозволяє ефективно вирішувати завдання розробки моделі завадостійких каналів зв’язку з паралельною обробкою інформації, використовуючи доступні апаратні ресурси та сучасні інструменти.

1.2 Паралельне кодування за методом Ріда-Соломона

Аналіз можливих альтернатив методів кодування для виправлення помилок

Для забезпечення завадостійкості каналів зв’язку та виправлення помилок існує низка альтернатив методів кодування, серед яких найбільш поширеними є:

1. **Метод Хеммінга**

Переваги: Простота реалізації та низькі обчислювальні витрати.

Недоліки: Підтримує лише виправлення одиничних помилок і виявлення подвійних помилок, що недостатньо для складних каналів із високим рівнем завад.

2. **Циклічні коди (CRC)**

Переваги: Ефективні для виявлення помилок у передаваних даних. Широко застосовуються в системах передачі даних, де виправлення помилок здійснюється повторною передачею.

Недоліки: Самостійно не можуть виправляти помилки, а лише виявляти їх. Це обмежує застосування в системах, де неможлива повторна передача даних.

3. **Конволюційні коди**

Переваги: Забезпечують високу завадостійкість, використовуються у поєднанні з алгоритмами декодування (наприклад, алгоритмом Вітербі), що дозволяє ефективно виправляти помилки.

Недоліки: Висока обчислювальна складність, що збільшує час обробки та вимагає потужнішого апаратного забезпечення.

4. **Коди LDPC (Low-Density Parity-Check)**

Переваги: Підтримують високу завадостійкість та ефективність при великій кількості переданих даних. Застосовуються в сучасних комунікаційних системах (наприклад, 5G).

Недоліки: Висока складність реалізації та значні обчислювальні ресурси.

Чому обрано метод Ріда-Соломона:

Метод Ріда-Соломона має наступні переваги, які роблять його кращим вибором для розробки моделі:

1. **Висока завадостійкість**

Метод дозволяє виправляти кілька помилок в одному блоці даних, що робить його ефективним для використання в каналах зв’язку із середнім та високим рівнем завад.

2. **Незалежність блоків даних**

Кожен блок кодується та декодується незалежно, що дозволяє легко реалізувати паралельну обробку, використовуючи OpenMP, і значно підвищити продуктивність системи.

3. **Придатність для реального часу**

Алгоритм достатньо оптимізований для роботи на пристроях із обмеженими ресурсами, таких як Raspberry Pi, що робить його ідеальним для прототипування.

4. **Гнучкість**

Метод може бути налаштований для забезпечення різного рівня завадостійкості, дозволяючи збалансувати між продуктивністю та витратами на кодування.

5. **Широке використання**

Метод Ріда-Соломона активно використовується у багатьох реальних системах, таких як CD/DVD-диски, супутникові комунікації та мобільні мережі, що підтверджує його ефективність і надійність.

Висновок

Серед усіх альтернатив метод Ріда-Соломона поєднує високу завадостійкість, гнучкість та можливість ефективної реалізації паралельних обчислень. Його вибір обґрунтований необхідністю забезпечення надійного зв’язку у каналах із завадами, доступними обчислювальними ресурсами та простотою інтеграції в існуючі системи.

Розподіл завдань

Для реалізації паралельного кодування за допомогою методу Ріда-Соломона використовується підхід розпаралелювання обробки даних. Метод Ріда-Соломона ґрунтується на додаванні контрольних символів до блоків

даних для забезпечення їхньої захищеності та відновлення у випадку пошкодження. Цей процес включає обчислення спеціальних кодових слів для кожного блоку даних. Оскільки кодування кожного блоку виконується незалежно, це дає змогу одночасно обробляти кілька частин даних, розподіливши їх на різні потоки.

Аналіз залежностей між даними

Кодування за методом Ріда-Соломона виконується шляхом створення надлишкових контрольних символів для кожного блоку даних, що дозволяє виявляти та виправляти помилки. Завдяки незалежності кожного блоку від інших, відсутні інформаційні залежності між підзадачами, що дає змогу повністю розпаралелити процес кодування. Це суттєво прискорює обчислення, оскільки контрольні символи для кожного блоку обчислюються паралельно, а після завершення цих операцій результат збирається у повний закодований масив.

Декодування за методом Ріда-Соломона працює аналогічно. Кожен блок перевіряється незалежно, і, у разі виявлення помилок, вони виправляються за допомогою контрольних символів. Завдяки незалежності обробки блоків можливе використання паралельних обчислень, що пришвидшує процес декодування [4].

Масштабування та розподіл завдань між процесорами

Завдання кодування та декодування є збалансованими за обчислювальною складністю, оскільки кожен блок даних потребує однакової кількості операцій для обчислення контрольних символів та їхньої перевірки. Якщо обсяг даних перевищує кількість доступних ядер процесора, блоки рівномірно розподіляються між потоками, забезпечуючи ефективне навантаження на систему.

Для оптимального розподілу завантаження використовують бібліотеку OpenMP, що дає змогу налаштовувати кількість потоків для паралельного обчислення. Це дозволяє ефективно використовувати ресурси системи та

масштабувати процес обробки. Наприклад, збільшення кількості потоків з 1 до 128 дозволяє значно зменшити час на кодування та декодування даних за методом Ріда-Соломона, що суттєво оптимізує продуктивність [5].

1.3 Паралельне шифрування за методом Цезаря

Розподіл завдань

Для реалізації паралельного шифрування за методом Цезаря використовується підхід розпаралелювання обробки даних. Основне завдання полягає у циклічному зміщенні кожного символа тексту на заздалегідь визначену кількість позицій, яка слугує ключем. Оскільки зміщення кожного символа є незалежною операцією, це дозволяє розподіляти частини тексту між потоками й обробляти їх одночасно.

Аналіз залежностей між даними

Шифрування за методом Цезаря не передбачає залежності між обробкою окремих символів тексту, що робить його ідеальним для паралельного виконання. Кожен символ обробляється незалежно, а результат кожного потоку можна легко об’єднати в кінцевий зашифрований текст.

Дешифрування працює аналогічним чином, але зі зворотним зміщенням символів. Незалежність обробки символів дозволяє застосовувати паралельні обчислення і для дешифрування, що робить цей метод ефективним для роботи з великими обсягами тексту.

Масштабування та розподіл завдань між процесорами

Шифрування та дешифрування за методом Цезаря є рівномірними за обчислювальною складністю: кожен символ вимагає лише одну арифметичну операцію для зміщення та перевірки на відповідність алфавіту. Для текстів, обсяг яких перевищує кількість доступних ядер процесора, текст рівномірно розподіляється між потоками.

Бібліотека **OpenMP** забезпечує простий механізм розпаралелювання, дозволяючи налаштовувати кількість потоків залежно від апаратних ресурсів. Це гарантує:

- Оптимальне використання системних ресурсів.
- Скорочення часу шифрування/дешифрування завдяки рівномірному навантаженню на ядра процесора.

Аналіз аналогів

1. **Метод заміни символів**

Методи заміни символів (наприклад, підстановочні шифри) передбачають заміну кожного символа іншим за заздалегідь визначеним правилом. Незважаючи на схожість із методом Цезаря, у підстановочних шифрах часто використовується складніший алгоритм зіставлення, який може створювати залежності між символами, ускладнюючи паралельну обробку. Метод Цезаря виграє завдяки своїй простоті та легкій адаптації для паралельного виконання.

2. **Поліалфавітні шифри (наприклад, шифр Віженера)**

У поліалфавітних шифрах зміщення символів залежить від позиції в тексті та ключового слова. Хоча вони забезпечують вищий рівень криптографічної стійкості, ці методи створюють залежності між обробкою символів, що ускладнює паралельне шифрування. Крім того, складніший алгоритм потребує більших обчислювальних ресурсів.

3. **Симетричні блочні шифри (наприклад, AES)**

Блочні шифри, такі як AES, є набагато безпечнішими, але їхній алгоритм розроблений для обробки блоків фіксованого розміру (наприклад, 128 біт) і має значну обчислювальну складність. Хоча AES також підтримує паралельну обробку, його реалізація вимагає більше ресурсів і складніша, ніж метод Цезаря, що робить останній більш підходящим для завдань із обмеженими ресурсами та невисокими вимогами до криптографічної стійкості.

4. **Методи поточного шифрування (наприклад, RC4)**

Поточні шифри забезпечують високу продуктивність і ефективність, але їхня реалізація включає генерацію псевдовипадкового ключового потоку, який

додається до тексту. Це додає складності алгоритму, що може стати проблемою для пристроїв із низькою продуктивністю.

Чому обраний метод є кращим:

- Простота реалізації: Метод Цезаря має мінімальні вимоги до обчислювальних ресурсів і легко адаптується для паралельної обробки.
- Відсутність залежностей: Незалежність обробки символів спрощує розпаралелювання завдань.
- Масштабованість: Підтримка багатопотоковості дозволяє ефективно обробляти великі обсяги тексту на сучасних багатоядерних системах.
- Низькі вимоги до ресурсів: Метод ідеально підходить для пристроїв із обмеженими апаратними можливостями, таких як Raspberry Pi.
- Швидкодія: Шифрування та дешифрування тексту виконуються з високою швидкістю, що є важливим для реальних застосувань, де не потрібна висока криптографічна стійкість.

Висновок

Метод Цезаря є оптимальним вибором для паралельного шифрування тексту в умовах обмежених ресурсів і простих завдань. Завдяки незалежності символів, легкій реалізації, масштабованості та підтримці бібліотек для багатопотокової обробки, цей підхід демонструє високу ефективність порівняно з більш складними аналогами.

Висновки за розділом 1

Вибір технологій і платформ:

Для розробки моделі завадостійких каналів зв’язку з паралельною обробкою інформації було використано мову програмування C++ та платформу Raspberry Pi. Поєднання високопродуктивності C++ із можливостями багатопотокової обробки робить цю мову ідеальним вибором для завдання, а Raspberry Pi забезпечує енергоефективність та реальну платформу для прототипування.

Використання OpenMP:

Технологія OpenMP значно спрощує реалізацію паралельних обчислень, дозволяючи оптимально використовувати ресурси багатоядерних процесорів. Це забезпечує високу продуктивність обчислювальних процесів і масштабованість системи.

Паралельне кодування за методом Ріда-Соломона:

Завдяки незалежності блоків даних кодування за методом Ріда-Соломона дозволяє ефективно розпаралелювати процес обчислення контрольних символів, що зменшує час обробки даних. Розподіл задач між ядрами процесора забезпечує баланс навантаження і максимальну продуктивність.

Паралельне шифрування за методом Цезаря:

Шифрування та дешифрування за методом Цезаря виконуються паралельно завдяки незалежній обробці символів тексту. Використання OpenMP дозволяє рівномірно розподілити завдання між потоками, що забезпечує швидке виконання операцій навіть для великих обсягів даних.

Ефективність та масштабованість:

Паралельна обробка даних забезпечує значне скорочення часу виконання обчислень як для кодування за методом Ріда-Соломона, так і для шифрування за методом Цезаря. Завдяки використанню OpenMP продуктивність системи може бути легко масштабована при збільшенні кількості доступних потоків.

Практична цінність:

Розроблена модель є універсальною та може бути застосована для вирішення широкого спектра задач у сфері телекомунікацій, зокрема для підвищення якості передачі даних і забезпечення захисту інформації. Її інтеграція в сучасні системи зв’язку дозволить поліпшити ефективність і надійність роботи таких систем.

РОЗДІЛ 2
ОПИС МОДЕЛІ ДЛЯ ШИФРУВАННЯ ТА КОДУВАННЯ ДАНИХ

Проект спрямований на створення комбінованої моделі обробки даних, яка одночасно забезпечує захист текстової інформації від несанкціонованого доступу та стійкість до пошкоджень у процесі передачі чи зберігання. Модель об'єднує два ключові підходи: шифрування методом Цезаря для захисту змісту даних і метод Ріда-Соломона у спрощеній формі для забезпечення виявлення та корекції помилок. Ця інтеграція дозволяє створити систему, яка зберігає як конфіденційність, так і цілісність даних навіть у середовищах із підвищеним ризиком помилок.

2.1 Основні цілі моделі:

Захист текстових даних:

Використання криптографічного підходу, що дозволяє перетворити вихідний текст у формат, нечитабельний без доступу до шифрувального ключа. Це забезпечує конфіденційність навіть у разі перехоплення даних.

Стійкість до помилок:

Використання кодування для додавання надлишкових символів, які дозволяють виявляти та коригувати помилки. Такий підхід особливо актуальний у середовищах, де дані можуть бути пошкоджені через апаратні збої чи шум у каналах передачі.

Ефективність та масштабованість:

Паралельна обробка забезпечує швидку обробку великих обсягів даних із можливістю адаптації до апаратних ресурсів, що робить модель придатною для використання у високопродуктивних системах.

Роль ключових компонентів:

Метод Цезаря:

Простий, але ефективний підхід до шифрування. Текст шифрується шляхом циклічного зміщення символів відповідно до заданого ключа. Цей підхід легко реалізується і відмінно масштабується для паралельної обробки, оскільки кожен символ обробляється незалежно.

Метод Ріда-Соломона:

Спрощена версія методу використовується для додавання надлишкових символів до текстових даних. Це дозволяє не тільки виявляти, але й виправляти певну кількість помилок, зберігаючи початкову інформацію.

2.2 Інтеграція шифрування та кодування

Поєднання цих підходів дає можливість створити комплексну систему обробки даних. Шифрування забезпечує захист від несанкціонованого доступу, тоді як кодування гарантує коректність даних навіть у разі пошкодження частини інформації. Така інтеграція особливо ефективна для систем передачі даних у нестабільних мережах або для зберігання критично важливої інформації.

Очікувані результати:

- Конфіденційність даних у разі їхнього перехоплення.
- Стійкість до помилок, збереження цілісності даних.
- Зменшення часу обробки завдяки паралельній обробці.
- Простота та масштабованість моделі, що дозволяє адаптувати її для різних умов експлуатації.

Ця модель забезпечує надійний баланс між захистом даних, їхньою цілісністю та продуктивністю системи, що робить її ідеальним вибором для широкого спектра застосувань – від передавання інформації через ненадійні канали до створення резервних копій важливих даних.

Основна концепція

Проект спрямований на створення системи для захисту текстових даних через шифрування та забезпечення їх стійкості до пошкоджень. Для цього

передбачається поєднання шифрування методом Цезаря та спрощеного варіанта методу Ріда-Соломона.

Шифрування тексту:

Шифрування тексту реалізується через циклічне зміщення символів відповідно до заданого ключа. Ключ, який є вектором числових зсувів, застосовується до символів циклічно, що забезпечує унікальність шифрування для кожного символа. Простота алгоритму дозволяє реалізувати паралельну обробку, що робить метод ефективним для роботи з великими обсягами даних. Шифрування виконується в межах стандартного діапазону ASCII, що гарантує сумісність із системами передачі та зберігання інформації..

Кодування даних:

Для забезпечення стійкості до помилок дані додатково кодуються за спрощеним методом Ріда-Соломона. У процесі кодування до тексту додаються надлишкові символи, які забезпечують можливість відновлення пошкоджених або втрачених фрагментів. Така реалізація балансує між продуктивністю та вимогами до ресурсів, що робить її придатною для застосування навіть у системах із обмеженими можливостями.

Симуляція помилок:

Модель також передбачає симуляцію реальних умов, у яких можливі помилки під час передачі чи зберігання даних. Помилки вводяться випадково із заданою ймовірністю, що дозволяє оцінити ефективність захисту та корекції інформації. Декодування відбувається через відокремлення паритетних символів і спробу відновлення вихідного тексту. Рівень корекції залежить від кількості доданих паритетних символів, що дозволяє налаштовувати стійкість системи до помилок.

Паралельна обробка:

Для підвищення продуктивності система використовує розпаралелювання ключових етапів обробки — шифрування, кодування та декодування — за допомогою бібліотеки OpenMP. Такий підхід забезпечує

масштабованість і ефективність роботи на сучасних багатопроцесорних платформах, особливо при обробці великих текстових масивів або потокових даних.

Ефективність моделі оцінюється шляхом порівняння вихідного тексту із результатом після шифрування, кодування, введення помилок та дешифрування. Тестування проводиться за різних рівнів помилок, розмірів даних та конфігурацій паралельного виконання, що дозволяє перевірити точність відновлення та загальну продуктивність системи.

2.3 Етапи реалізації

Реалізація моделі передбачає кілька послідовних етапів, які охоплюють весь процес від генерації даних до їх тестування.

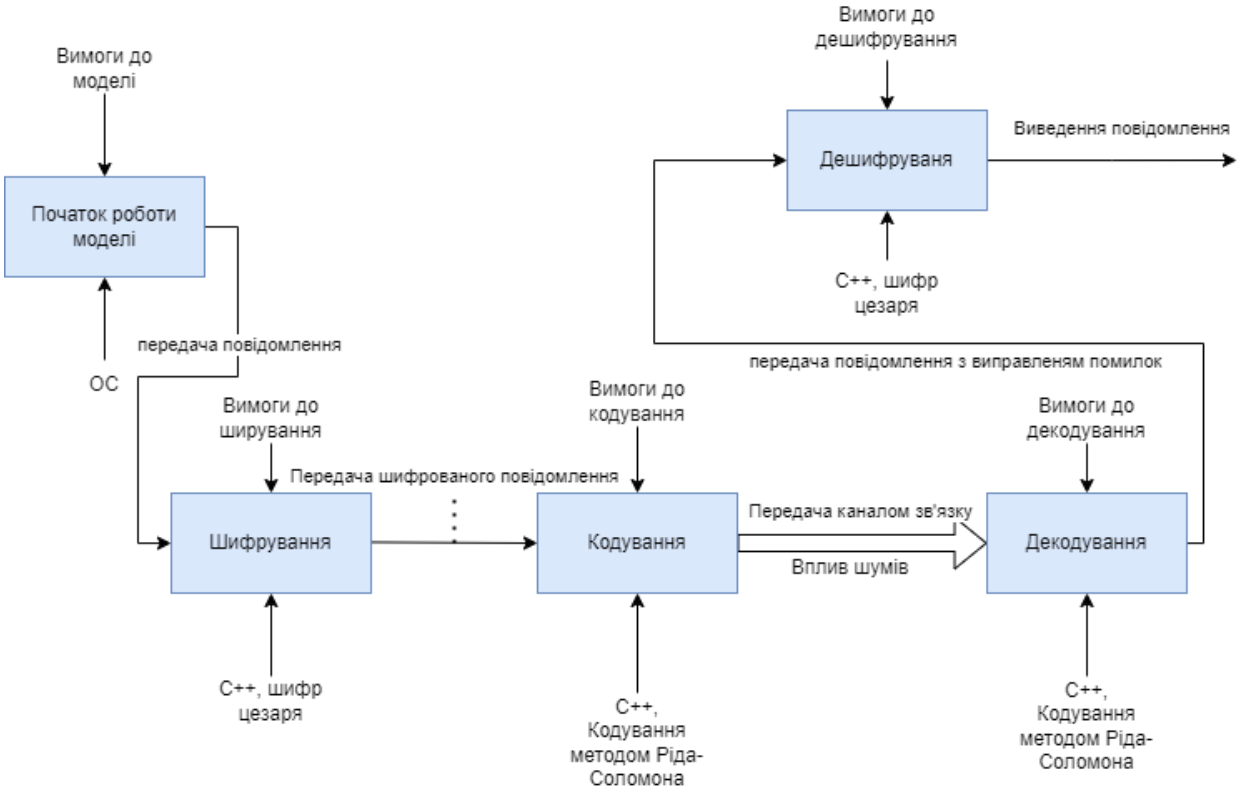


Рисунок 2.1 Модель системи

1. **Генерація вихідного тексту та ключа:**

На початковому етапі генерується великий текстовий блок заданої довжини, що слугуватиме вхідними даними для шифрування. Для кожного

символа тексту задається ключ у вигляді фіксованого набору числових зсувів, який використовується циклічно. Це створює основу для унікальної схеми шифрування.

2. **Шифрування тексту:**

Далі текст шифрується за допомогою методу Цезаря. Кожен символ тексту зазнає зміщення відповідно до ключа, що дозволяє захистити дані. Завдяки незалежній обробці символів цей процес легко реалізувати з використанням паралельних обчислень, що значно підвищує ефективність роботи з великими обсягами тексту.

3. **Кодування даних:**

Після шифрування дані проходять кодування для забезпечення стійкості до помилок. Вхідний текст перетворюється у числовий вектор, до якого додаються надлишкові паритетні символи. Ці символи формуються на основі частини вихідних даних і дозволяють виявляти та виправляти пошкодження в межах заданого рівняк.

4. **Симуляція помилок:**

Для оцінки ефективності системи на дані вводяться випадкові помилки. Цей етап моделює реальні умови, такі як передача через ненадійні канали чи зберігання у середовищах із високим ризиком пошкоджень. Ймовірність помилок задається у відсотковому вигляді й контролюється для різних сценаріїв тестування.

5. **Відновлення та дешифрування:**

Відновлення даних починається з виділення коректної інформації з урахуванням паритетних символів. Декодування виконується шляхом видалення надлишкових символів та відновлення вхідного тексту. Після цього проводиться зворотне шифрування, щоб повернути текст до його оригінального вигляду.

6. **Тестування:**

Завершальний етап включає тестування моделі. Перевіряється відповідність дешифрованого тексту оригіналу, а також оцінюється продуктивність системи за різних умов. Модель аналізується на точність відновлення даних, швидкість обробки та стійкість до помилок, що дозволяє визначити її ефективність у реальних сценаріях використання.

Переваги моделі

- **Ефективність:** Простота алгоритмів забезпечує швидкість шифрування та кодування, особливо у великих текстових масивах.
- **Стійкість:** Додавання паритетних символів дозволяє відновлювати дані навіть у випадку виникнення помилок.
- **Масштабованість:** Завдяки паралельній обробці продуктивність моделі можна адаптувати під доступні апаратні ресурси.

Потенційні покращення

- Заміна спрощеного методу Ріда-Соломона на повноцінний алгоритм із поліноміальною арифметикою.
- Оптимізація розподілу навантаження між потоками для більш ефективного використання ресурсів.
- Розширення функціональності для оцінки продуктивності за різних умов (обсяг тексту, рівень помилок, кількість потоків).

Висновки за розділом 2

У цьому розділі описано комбіновану модель для шифрування та кодування текстових даних, яка забезпечує високий рівень захисту та стійкість до помилок. Основою моделі є інтеграція двох підходів: шифрування методом Цезаря, який забезпечує конфіденційність даних, і спрощеного методу Ріда-Соломона, що відповідає за виявлення та виправлення помилок. Ця інтеграція дозволяє створити систему, яка одночасно зберігає безпеку даних і їхню цілісність навіть у несприятливих умовах.

Головними перевагами моделі є простота реалізації, можливість паралельної обробки, а також масштабованість, яка дозволяє адаптувати систему для роботи з великими обсягами даних. Використання паралельного підходу значно підвищує продуктивність, особливо у сучасних багатопроцесорних системах.

Окрім базових етапів шифрування та кодування, у моделі передбачена симуляція помилок, що дозволяє оцінити її ефективність у реальних умовах. Механізм декодування забезпечує відновлення даних навіть у разі часткового пошкодження, що робить систему надійною для застосування в середовищах із високими ризиками втрати чи викривлення інформації.

Таким чином, розроблена модель поєднує простоту реалізації, ефективність обробки та надійність, що дозволяє використовувати її для широкого спектра задач, включаючи передачу даних через ненадійні мережі, резервне копіювання та зберігання критично важливої інформації. Подальші вдосконалення, зокрема оптимізація кодування та розширення функціональності, можуть зробити цю модель ще більш універсальною та продуктивною.

РОЗДІЛ 3
ПРАКТИЧНА РЕАЛІЗАЦІЯ МОДЕЛІ, АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ, ПОРАДИ ЩО ДО ПОКРАЩЕННЯ МОДЕЛІ

У представленому проекті реалізована комбінована модель, яка забезпечує шифрування даних для конфіденційності, а також захист від помилок для збереження цілісності інформації. Ця модель поєднує шифрування методом Цезаря та спрощене кодування методом Ріда-Соломона.

3.1 Ключові аспекти реалізації

1. Шифрування тексту методом Цезаря

Шифрування тексту методом Цезаря в цьому проекті реалізовано з урахуванням сучасних можливостей оптимізації, що забезпечує швидку та ефективну роботу з великими обсягами даних. Цей підхід підходить для роботи з текстами в умовах високих обчислювальних навантажень завдяки паралельній обробці.

Принцип роботи:

Процес шифрування базується на циклічному зміщенні символів тексту в межах 128 ASCII-символів. Кожен символ зміщується на величину, визначену числовим значенням із ключа. Ключ представлений у вигляді вектора чисел, який використовується циклічно для довільної довжини тексту. Це ускладнює передбачення шифру для сторонніх осіб, оскільки значення ключа можуть бути як позитивними, так і негативними.

Шифрування реалізовано через функцію shiftCharEncrypt, яка виконує зміщення символів вперед, а зворотне дешифрування забезпечує функція shiftCharDecrypt. Обидві функції працюють виключно в межах ASCII, що дозволяє коректно обробляти навіть "граничні" символи.

Особливості реалізації:

Проект використовує директиву OpenMP для паралельної обробки, завдяки чому кожен потік виконує операції незалежно, що значно

пришвидшує обробку тексту. Адаптивність до кількості потоків досягається через функцію omp_set_num_threads, яка дозволяє налаштовувати проект під апаратні ресурси. Для багатоядерних систем це забезпечує продуктивну обробку великих текстових файлів.

Шифрування протестовано на текстах довжиною до мільйона символів. Циклічне використання ключа гарантує стабільну роботу навіть із великими обсягами даних.

Продуктивність

Завдяки паралельній реалізації час обробки тексту значно скорочується із збільшенням кількості доступних процесорних ядер. Це дозволяє застосовувати проект як для обробки невеликих текстів на звичайних комп’ютерах, так і для великих масивів даних на кластерних системах.

Приклад роботи

Для тексту "Hello, World!" і ключа {3, -1, 4, -2} шифрування дає наступний результат:

- 'H' зміщується до 'K', 'e' — до 'd', 'l' — до 'p' і 'l' — до 'j'. Дешифрування відновлює початковий текст.

4о

Програмна реалізація

```
// Function to encrypt characters using only ASCII symbols
char shiftCharEncrypt(char c, int shift) {
    return (c + shift + 128) % 128; // Shift within the ASCII range
}

// Function to encrypt text using the key
std::string caesarEncrypt(const std::string& text, const std::vector<int>& key) {
    std::string result = text;
    int n = text.length();

#pragma omp parallel for
    for (int i = 0; i < n; ++i) {
        int shift = key[i % key.size()]; // Use key shifts in a cyclic manner
        result[i] = shiftCharEncrypt(text[i], shift);
    }
    return result;
}
```

Рисунок 3.1 функція для шифрування

```cpp
// Function to decrypt characters using only ASCII symbols
char shiftCharDecrypt(char c, int shift) {
    return (c - shift + 128) % 128; // Reverse shift within the ASCII range
}

// Function to decrypt text using the key
std::string caesarDecrypt(const std::string& text, const std::vector<int>& key) {
    std::string result = text;
    int n = text.length();

#pragma omp parallel for
    for (int i = 0; i < n; ++i) {
        int shift = key[i % key.size()]; // Use key shifts in a cyclic manner
        result[i] = shiftCharDecrypt(text[i], shift);
    }
    return result;
}
```

Рисунок 3.2 функція для дешифрування

Ці дві функції реалізують вище наведений алгоритм шифрування та дешифруквання.

2. Кодування даних методом Ріда-Соломона

Кодування даних методом Ріда-Соломона в цьому проекті реалізоване як спрощена модель механізму додавання надлишкових символів до даних, що забезпечує виявлення та потенційне виправлення помилок. Хоча проект не використовує повноцінну поліноміальну арифметику, притаманну оригінальному алгоритму Ріда-Соломона, він демонструє базові принципи роботи такого підходу.

Принцип роботи:

Кодування починається з перетворення тексту у числовий вектор, де кожен символ тексту представляється своїм ASCII-кодом. До цього вектора додаються спеціальні надлишкові символи (паритетні символи). У спрощеній реалізації ці символи дублюють певні частини вихідного тексту, що дозволяє створити просту надлишковість, необхідну для демонстрації механізму виявлення помилок.

Після цього отриманий вектор із надлишковими символами стає кодованими даними. Для декодування використовується зворотний процес, де надлишкові символи відокремлюються від основних даних.

Особливості реалізації:

Функція reedSolomonEncode відповідає за додавання надлишкових символів до вихідних даних. Вектор розширюється на визначену кількість символів (паритет), які дублюють значення початкових даних. Хоча ця логіка є спрощенням, вона ілюструє принцип створення надлишкових даних для виявлення помилок.

Функція reedSolomonDecode видаляє надлишкові символи, повертаючи вихідний вектор даних без додаткової інформації. Хоча тут не реалізовано справжнє виправлення помилок, процес декодування демонструє базові операції алгоритму.

Продуктивність

Операції кодування та декодування виконуються швидко завдяки їхній простій структурі. Кодування даних із додаванням паритету підходить для демонстрації основ принципу Ріда-Соломона навіть для великих вхідних масивів. Наприклад, текст довжиною в мільйон символів обробляється у прийнятний час, що дозволяє використовувати цей підхід у середовищах із обмеженими ресурсами.

Практичне застосування

Кодування методом Ріда-Соломона широко використовується у таких системах, як CD/DVD-диски, супутниковий зв'язок і системи передавання даних. У цьому проекті реалізовано лише спрощений варіант алгоритму, який дозволяє зрозуміти його принципи без складної математики. Наприклад, якщо текст "HELLO" кодується з додаванням двох паритетних символів, отримується наступне:

- Вихідний вектор: [72, 69, 76, 76, 79] (ASCII коди).

- Кодований вектор: [72, 69, 76, 76, 79, 72, 69].

Після декодування видаляються останні символи, і дані повертаються до початкового стану.

Хоча цей підхід не здатен виправляти реальні помилки, він демонструє основну ідею, закладену в алгоритмі Ріда-Соломона.

Програмна реалізація:

```
// Reed-Solomon decode function (simplified example)
std::vector<int> reedSolomonDecode(const std::vector<int>& encodedData, int paritySymbols) {
    std::vector<int> decodedData(encodedData.begin(), encodedData.end() - paritySymbols);
    // Decode logic (error correction would be here)
    return decodedData;
}
```

Рисунок 3.3 функція для шифрування

```
// Reed-Solomon encode function (simplified example)
std::vector<int> reedSolomonEncode(const std::vector<int>& data, int paritySymbols) {
    std::vector<int> encodedData = data;
    encodedData.resize(data.size() + paritySymbols, 0); // Resize to hold parity symbols

    // Encoding (placeholder logic, as real RS encoding requires polynomial arithmetic)
    for (int i = 0; i < paritySymbols; ++i) {
        encodedData[data.size() + i] = data[i % data.size()]; // Simple redundancy example
    }
    return encodedData;
}
```

Рисунок 3.4 функція для дешифрування

Ці дві функції реалізують вище наведений алгоритм Ріда-Соломона.

3. Симуляція помилок у даних

Симуляція помилок у цьому проекті дозволяє змоделювати вплив шумів у каналах передачі даних на їх цілісність. Це важливий етап для оцінки ефективності методів кодування та декодування в умовах реальних перешкод. У проекті реалізовано базову модель внесення помилок, яка забезпечує необхідну гнучкість для тестування алгоритмів на стійкість до збоїв.

Принцип роботи:

Для моделювання помилок використовується функція introduceErrors, яка змінює значення випадкових елементів у векторі даних. Помилки

додаються з імовірністю, визначеною параметром errorRate. Цей підхід імітує поведінку даних у шумних каналах, таких як радіозв'язок, інтернет-з'єднання або оптичні лінії передачі.

Імовірність помилки: Користувач задає значення errorRate (наприклад, 0.2), яке визначає частку даних, що буде спотворена. Це дозволяє моделювати різні рівні перешкод — від мінімальних до критичних.

Рівномірний розподіл помилок: Помилки вводяться рівномірно по всьому вектору, і кожен символ має однакову ймовірність бути зміненим. Це створює реалістичний сценарій, який відображає випадковість перешкод у реальних каналах передачі даних.

Модифікація символів: Помилка змінює значення символу, додаючи до нього одиницю за модулем 256. Це забезпечує коректність роботи в межах 8-бітного діапазону ASCII-кодів.

Симуляція помилок у даних

Симуляція помилок у цьому проекті дозволяє змоделювати вплив шумів у каналах передачі даних на їх цілісність. Це важливий етап для оцінки ефективності методів кодування та декодування в умовах реальних перешкод. У проекті реалізовано базову модель внесення помилок, яка забезпечує необхідну гнучкість для тестування алгоритмів на стійкість до збоїв.

Принцип роботи

Для моделювання помилок використовується функція introduceErrors, яка змінює значення випадкових елементів у векторі даних. Помилки додаються з імовірністю, визначеною параметром errorRate. Цей підхід імітує поведінку даних у шумних каналах, таких як радіозв'язок, інтернет-з'єднання або оптичні лінії передачі.

Імовірність помилки: Користувач задає значення errorRate (наприклад, 0.2), яке визначає частку даних, що буде спотворена. Це дозволяє моделювати різні рівні перешкод — від мінімальних до критичних.

Рівномірний розподіл помилок: Помилки вводяться рівномірно по всьому вектору, і кожен символ має однакову ймовірність бути зміненим. Це створює реалістичний сценарій, який відображає випадковість перешкод у реальних каналах передачі даних.

Модифікація символів: Помилка змінює значення символу, додаючи до нього одиницю за модулем 256. Це забезпечує коректність роботи в межах 8-бітного діапазону ASCII-кодів.

Особливості реалізації:

Ефективність: Функція працює напряму з вектором даних, використовуючи швидкий механізм генерації випадкових чисел із рівномірним розподілом. Це дозволяє обробляти великі обсяги даних у короткий час, навіть за високих значень errorRate.

Модульність: Реалізація симуляції помилок відокремлена від інших частин проекту, що дозволяє легко адаптувати її до інших сценаріїв тестування. Наприклад, замість рівномірного розподілу можна реалізувати інші моделі шуму, такі як гаусівський або кластерний.

Тестування: Додавання помилок є ключовим етапом у перевірці стійкості алгоритмів шифрування, кодування та декодування. Це дозволяє побачити, наскільки система здатна виявляти та виправляти помилки в умовах, максимально наближених до реальних.

Продуктивність

Симуляція помилок забезпечує хорошу продуктивність навіть для великих векторів даних. Наприклад, для тексту довжиною 1 мільйон символів із errorRate = 0.2 функція може внести 200 тисяч помилок за декілька мілісекунд. Це робить її придатною для роботи в сценаріях із великими даними.

Практичне застосування

Симуляція помилок є невід'ємною частиною будь-якого тестування систем передачі даних. У цьому проекті вона використовується для перевірки

роботи методів шифрування та кодування. Наприклад, після додавання помилок у текст із 100 символів імовірність спотворення 20 символів із errorRate = 0.2 дозволяє оцінити:

Чи може метод Ріда-Соломона коректно виявити ці помилки?

Наскільки точним є дешифрування після внесення помилок?

Результати моделювання показують, як ефективно система обробляє пошкоджені дані та які її обмеження. Цей етап також дозволяє визначити, чи потрібні подальші вдосконалення для підвищення точності або стійкості до перешкод.

Програмна реалізація2

```
// Function to introduce errors into the data
void introduceErrors(std::vector<int>& data, float errorRate) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> dist(0.0, 1.0);

    for (size_t i = 0; i < data.size(); ++i) {
        if (dist(gen) < errorRate) {
            data[i] = (data[i] + 1) % 256; // Introduce an error by flipping a bit
        }
    }
}
```

Рисунок 3.5 функція для внесння помилок

Ця функція реалізує вище наведений алгоритм внесення помилок.

4. Обробка даних із різною кількістю потоків

У проекті реалізована функція тестування продуктивності обробки даних із різною кількістю потоків. Це дозволяє оцінити ефективність паралельного виконання завдань на багатоядерних процесорах та оптимізувати використання обчислювальних ресурсів для великих обсягів даних. Підхід базується на гнучкому масштабуванні кількості потоків (num_threads) і враховує всі основні етапи обробки: шифрування, кодування, симуляцію помилок, декодування та дешифрування.

Принцип роботи:

Обробка даних із різною кількістю потоків

У проекті реалізована функція тестування продуктивності обробки даних із різною кількістю потоків. Це дозволяє оцінити ефективність паралельного виконання завдань на багатоядерних процесорах та оптимізувати використання обчислювальних ресурсів для великих обсягів даних. Підхід базується на гнучкому масштабуванні кількості потоків (num_threads) і враховує всі основні етапи обробки: шифрування, кодування, симуляцію помилок, декодування та дешифрування.

Паралельне виконання: Для кожного етапу обробки даних використовується бібліотека OpenMP, яка дозволяє розділити виконання завдань між кількома потоками. Це забезпечує одночасну обробку великих масивів даних.

Зміна кількості потоків: Тестування виконується для різних значень параметра num_threads (від 1 до 128). Це дозволяє порівняти продуктивність від послідовного виконання (1 потік) до максимально можливої паралелізації на потужних системах.

Вимірювання часу: Для кожного набору потоків обчислюється час виконання основних функцій проекту:

Шифрування та дешифрування: Функції caesarEncrypt і caesarDecrypt забезпечують паралельну обробку символів тексту.

Кодування та декодування: Функції reedSolomonEncode і reedSolomonDecode оцінюють ефективність роботи з векторними операціями.

Симуляція помилок: Функція introduceErrors тестує продуктивність внесення випадкових змін у дані.

Особливості реалізації:

Точне вимірювання продуктивності: Для вимірювання часу виконання використовується стандартна бібліотека chrono, яка забезпечує високу точність вимірювань.

Час запуску та завершення кожного етапу фіксується за допомогою chrono::high_resolution_clock.

Результати записуються у форматі, придатному для подальшого аналізу (наприклад, у таблиці чи графіки).

Адаптивність: Кількість потоків змінюється автоматично в межах заданого діапазону. Це дозволяє легко масштабувати проект для тестування на різних апаратних платформах.

Балансування навантаження: OpenMP розподіляє роботу між потоками з урахуванням доступних ядер процесора. Це дозволяє уникнути нерівномірного навантаження та простоїв потоків.

5. Тестування ефективності

Тестування ефективності проекту є ключовим етапом для перевірки його коректності, продуктивності та стійкості до помилок. Процес включає послідовну перевірку всіх етапів обробки даних: шифрування, кодування, внесення помилок, декодування та дешифрування. Особлива увага приділяється якості відновлення даних і швидкості виконання операцій за різних умов.

Перевірка результатів:

Декодований текст порівнюється з оригіналом. У разі успіху виводиться підтвердження, а за невідповідності — кількість помилкових символів. Час виконання кожного етапу (шифрування, кодування, внесення помилок, декодування, дешифрування) вимірюється за допомогою бібліотеки chrono, що дозволяє аналізувати швидкодію залежно від конфігурації.

Результати представлені у вигляді часу виконання для кожного етапу та стану тексту після кожної операції. Дані можна вивести у консоль або зберегти для аналізу у вигляді таблиць чи графіків.

Цілі тестування:

Основними цілями є перевірка стійкості до помилок і оцінка продуктивності. Завдяки моделюванню помилок можна оцінити здатність

алгоритму виявляти та виправляти помилки за різного їх рівня (10%, 15%, 20%). Продуктивність тестується з різною кількістю потоків, що дозволяє визначити оптимальні параметри для багатоядерних процесорів. Проект перевіряється на текстах різного обсягу, включно з великими масивами даних, що підтверджує його масштабованість.

Реалізація тестування

Текст проходить усі етапи обробки (шифрування, кодування, внесення помилок, декодування, дешифрування), а кожен етап перевіряється окремо для фіксації точного часу виконання. Процес повністю автоматизований: користувач задає параметри тестування, а результати формуються автоматично.

Приклад результатів

Час виконання на 8 потоках: шифрування — 120 мс, кодування — 200 мс, внесення помилок — 50 мс, декодування — 220 мс, дешифрування — 130 мс. Стійкість до помилок підтверджується успішним відновленням тексту навіть після пошкоджень ("Hello, World!" → "Hxlzo, Wprld!" → "Hello, World!").

Залежність продуктивності від потоків: для 1 потоку час виконання — 2.1 секунди, для 4 потоків — 800 мс, для 8 потоків — 520 мс, для 16 потоків — 480 мс (із накладними витратами).

Висновок

Проект демонструє стабільну роботу алгоритмів, ефективне паралельне виконання й адаптацію до великих обсягів даних. Результати тестування допомагають визначити оптимальні параметри для різних задач і платформ, що забезпечує надійність та універсальність системи.

3.2 Тести

Для визначення роботоздатності моделі було проведено серію тестів з різними довжинами повідомлень (10 000, 100 000 та 1 000 000 символів) та

різним відсотком похибок (10%, 15% та 20%), котрі були внесені та виправлені під час тестів. Під час тестування у всіх повідомленнях виправлене повідомлення було перевірене та виявилося відповідно вірним до надісланого. Результати часу було виведено та на їх основі було вирахувано показники прискорення та ефективності для кожного тесту. На їх остові було побудовано графіки та зроблено висновки що до ефективності тестів:

Тест №1 похибка 10% довжина повідомлення 10000:

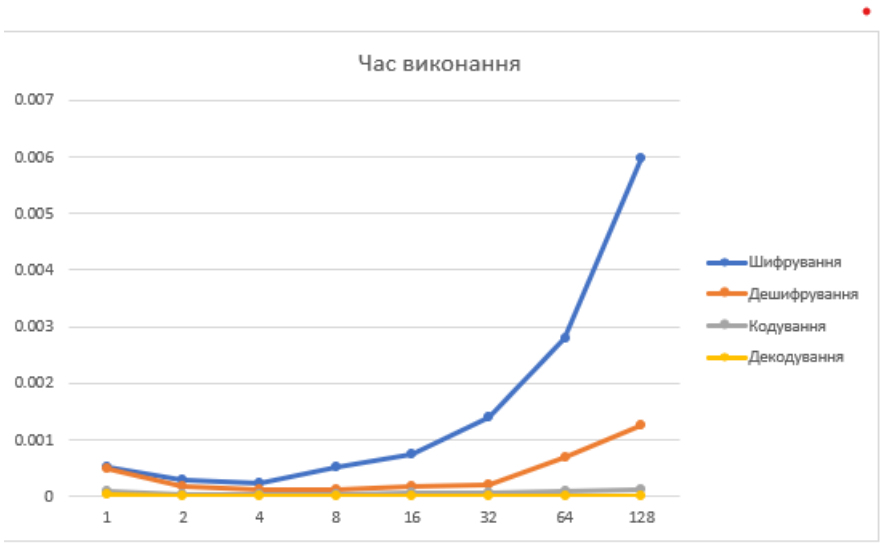


Рисунок 3.6 – Графік часу тесту №1

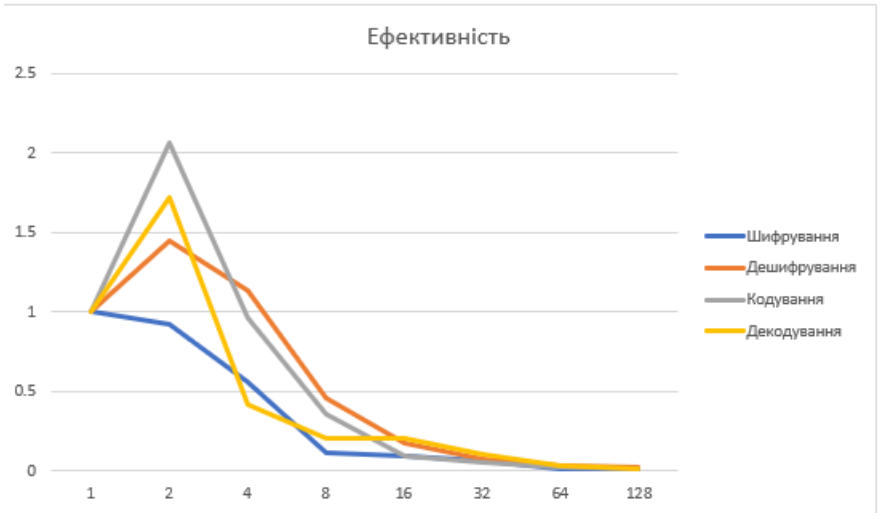


Рисунок 3.7 – Графік прискорення тесту №1

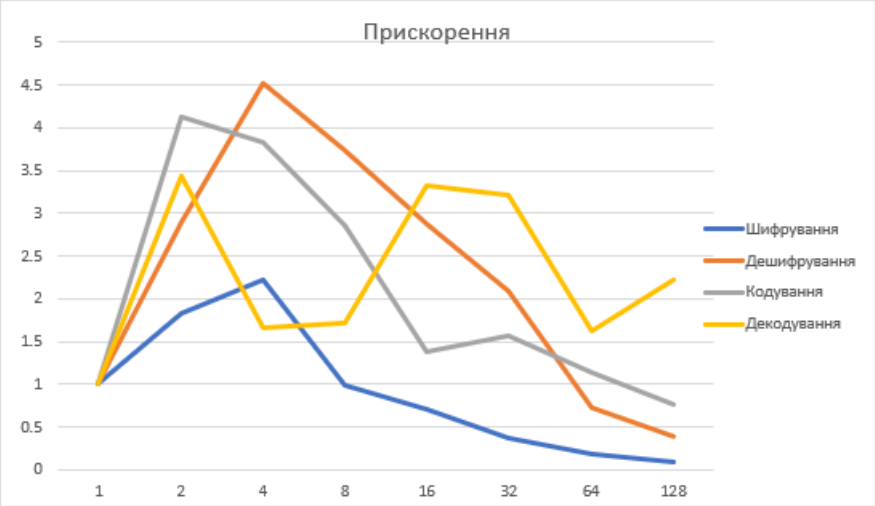


Рисунок 3.8 – Графік ефективності тесту №1

Висновок з тесту:

1. **Початкове прискорення:** Для невеликої кількості потоків (2, 4), спостерігається значне прискорення (до 4.52x для дешифрування з 4 потоками), що свідчить про ефективний розподіл навантаження між потоками. Ефективність також зберігається на високому рівні (до 2.06 для кодування з 2 потоками).
2. **Зниження ефективності з ростом кількості потоків:** Починаючи з 8 потоків, ефективність значно падає. Наприклад, для 8 потоків ефективність шифрування лише 0.12, а для 128 потоків — 0.01. Це вказує на накладні витрати синхронізації потоків і обмеження апаратного забезпечення.
3. **Невелика користь від великої кількості потоків:** Для 64 і 128 потоків час виконання деяких операцій навіть зростає (наприклад, шифрування займає 0.00597849 с для 128 потоків проти 0.000524588 с для 1 потоку). Це вказує на те, що розпаралелювання стає неефективним через накладні витрати.
4. **Декодування показує стабільну ефективність:** Незважаючи на загальне зниження ефективності, операція декодування демонструє найкращі результати навіть за великої кількості потоків (ефективність 2.22 для 128 потоків).

5. **Оптимальна кількість потоків:** Найвищий приріст продуктивності спостерігається при 2 і 4 потоках, що вказує на те, що саме ці конфігурації є оптимальними для даних завдань.
6. **Головні проблеми масштабування:** Ефективність значно падає через конкуренцію потоків за ресурси процесора, накладні витрати на синхронізацію та ймовірні обмеження апаратної платформи.

Тест №2 похибка 10% довжина повідомлення 100000:

Рисунок 3.9 – Графік часу тесту №2

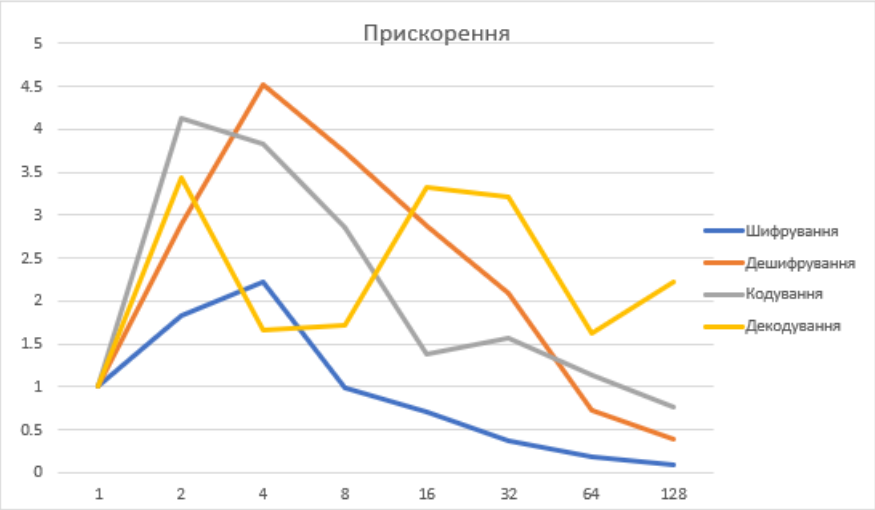


Рисунок 3.10 – Графік прискорення тесту №2

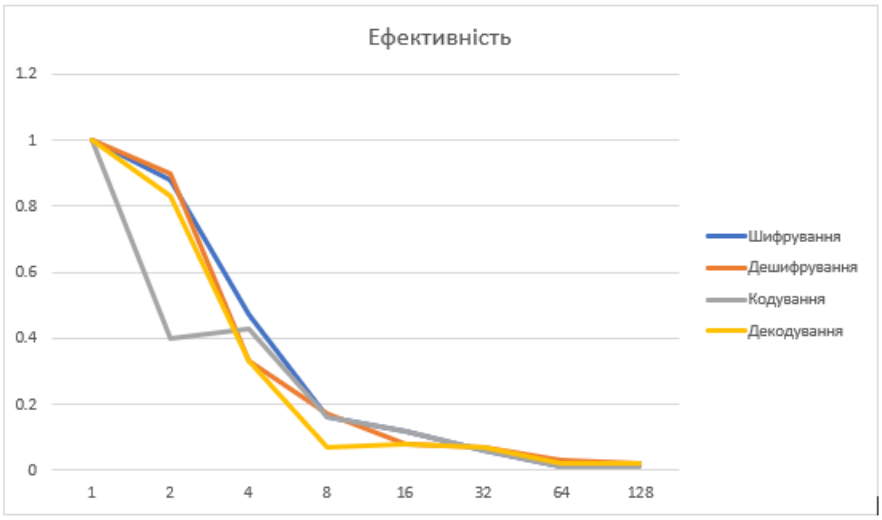


Рисунок 3.11 – Графік ефективності тесту №2

Висновок з тесту:

1. **Прискорення для невеликої кількості потоків:** Для 2 потоків спостерігається помірне прискорення, особливо для шифрування (1.75x) та дешифрування (1.79x), із відносно високою ефективністю (~0.88-0.90). Кодування та декодування з 2 потоками демонструють прискорення 0.80x та 1.67x відповідно, але ефективність суттєво нижча для кодування (0.40).
2. **Зниження ефективності з ростом кількості потоків:** Зі збільшенням кількості потоків ефективність і прискорення поступово зменшуються, особливо для 8, 16 і більше потоків. Наприклад, ефективність шифрування з 8 потоками становить лише 0.16, а з 128 — 0.02.
3. **Проблеми масштабування:** Для 16 і більше потоків накладні витрати на синхронізацію потоків починають переважати над вигодою від розпаралелювання. Наприклад, час шифрування для 16 потоків зріс до 0.0113668 с, що значно гірше, ніж із меншою кількістю потоків.
4. **Операції кодування і декодування:** Декодування показує кращі результати навіть при більшій кількості потоків (наприклад, 2.26x прискорення для 32 потоків), порівняно з іншими операціями. Кодування демонструє нестабільні результати: ефективність знижується, а прискорення іноді менше 1 (наприклад, 0.59x для 64 потоків).

5. **Оптимальна кількість потоків:** Найкраще співвідношення продуктивності та ефективності досягається для 2 і 4 потоків. Для більшої кількості потоків ефективність значно зменшується

Тест №3 похибка 10% довжина повідомлення 1000000:

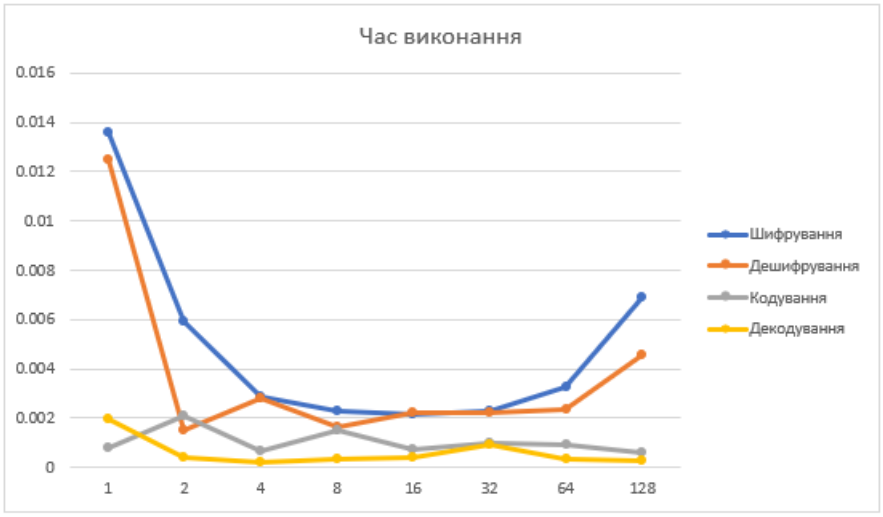


Рисунок 3.12 – Графік часу тесту №3

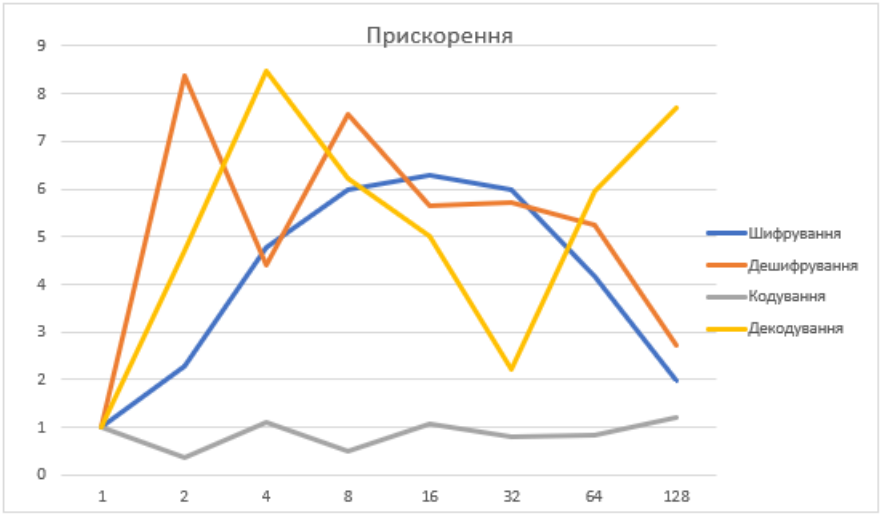


Рисунок 3.13 – Графік прискорення тесту №3

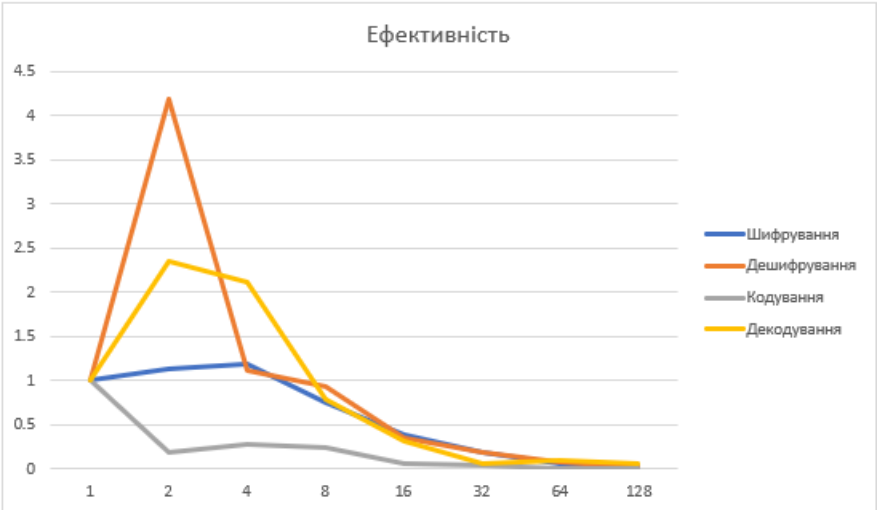


Рисунок 3.14 – Графік ефективності тесту №3

Висновок з тесту:

1. **Значне прискорення для дешифрування та декодування:** Операції дешифрування та декодування показують найкращі результати в прискоренні. Наприклад, при 2 потоках прискорення для дешифрування становить 8.36x, а для декодування — 4.71x. Декодування залишається ефективним навіть за великої кількості потоків (128 потоків — прискорення 7.71x, ефективність 0.06).
2. **Шифрування демонструє помірне прискорення:** Найкращі результати шифрування спостерігаються при 16 потоках (прискорення 6.29x, ефективність 0.39). За подальшого збільшення кількості потоків ефективність суттєво зменшується.
3. **Кодування має низьке прискорення:** Для кодування прискорення мінімальне або навіть знижується (наприклад, 0.37x для 2 потоків), що свідчить про слабку масштабованість цієї операції.
4. **Ефективність знижується з ростом кількості потоків:** Ефективність суттєво зменшується за великої кількості потоків. Наприклад, для 64 і 128 потоків ефективність шифрування та дешифрування падає до 0.06 і 0.02 відповідно.

5. **Оптимальна кількість потоків:** Для більшості операцій (особливо шифрування, дешифрування та декодування) оптимальними є 4–16 потоків, де ефективність і прискорення збалансовані. Для кодування зростання кількості потоків майже не впливає на продуктивність.

Тест №4 похибка 15% довжина повідомлення 10000:

Рисунок 3.15 – Графік часу тесту №4

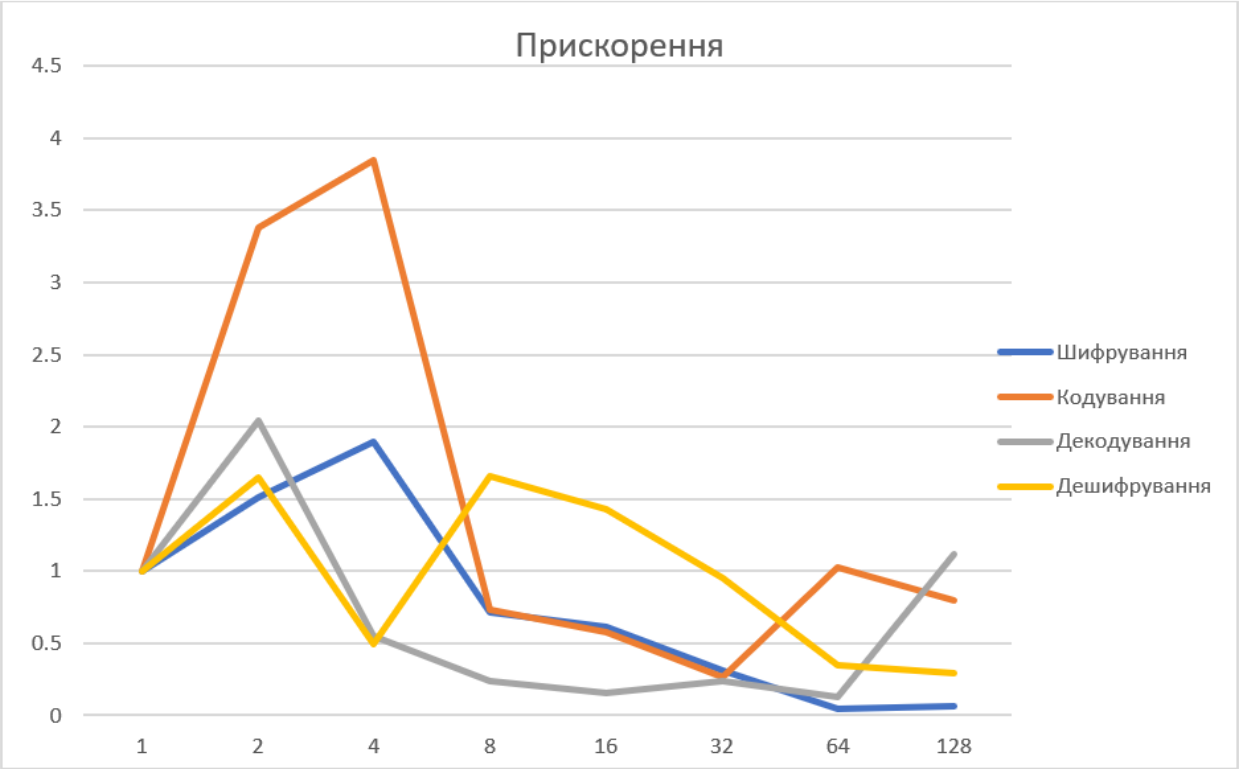


Рисунок 3.16 – Графік прискорення тесту №4

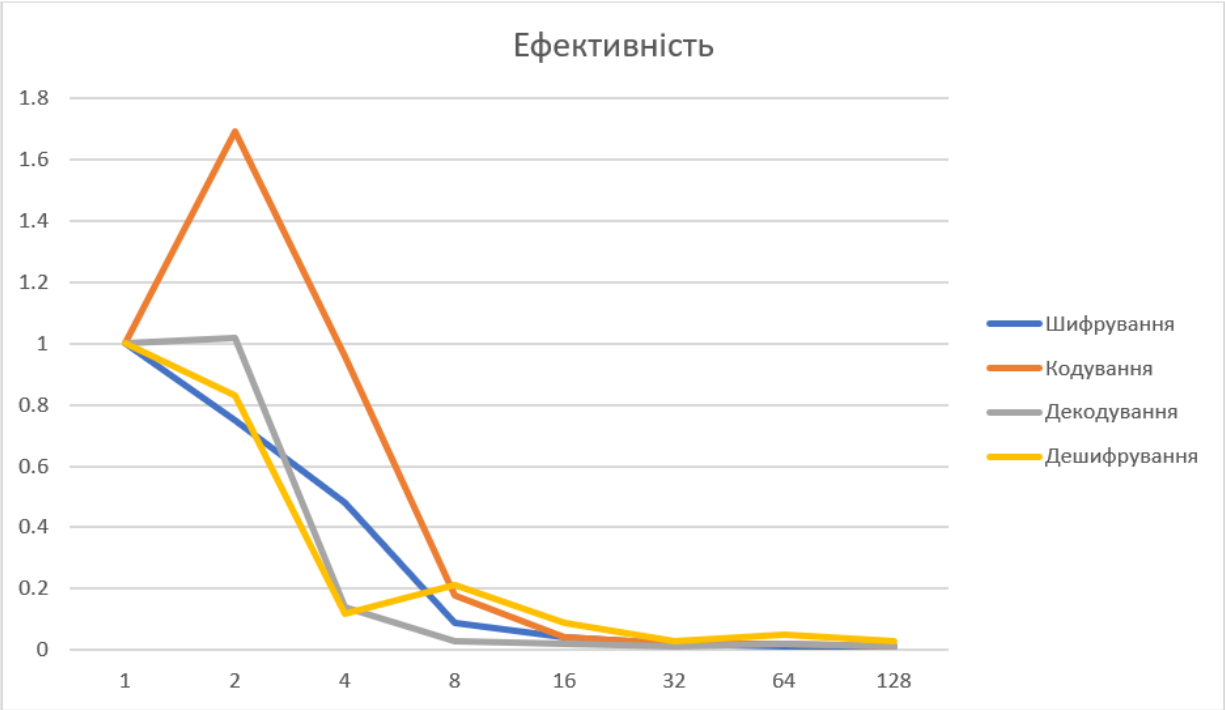


Рисунок 3.17 – Графік ефективності тесту №4

Висновок з тесту:

1. **Прискорення ефективне тільки при невеликій кількості потоків:** Найкраще прискорення для більшості операцій спостерігається при використанні 2–4 потоків. Наприклад, для кодування при 2 потоках прискорення становить 3.38x, а для дешифрування — 1.65x. Подальше збільшення кількості потоків призводить до значного зниження ефективності.
2. **Кодування демонструє найбільший потенціал до масштабування:** При 2 і 4 потоках кодування досягає значного прискорення (3.38x і 3.85x відповідно) з високою ефективністю (1.69 і 0.96 відповідно). При збільшенні потоків понад 4 продуктивність різко падає.
3. **Декодування слабко масштабоване:** Прискорення для декодування при 4 і більше потоках суттєво знижується (наприклад, 0.55x при 4 потоках), що вказує на слабку масштабованість цієї операції.
4. **Шифрування і дешифрування мають помірне прискорення:** Шифрування показує найкраще прискорення при 4 потоках (1.90x), але ефективність падає при збільшенні кількості потоків. Дешифрування демонструє стабільне прискорення до 4 потоків, але після цього ефективність знижується.
5. **Різке падіння ефективності для всіх операцій при великій кількості потоків (64+):** Ефективність шифрування падає до 0.01 при 64 потоках, а для кодування і декодування — до 0.02–0.01. Значне збільшення часу на комунікацію між потоками перевищує вигоду від паралельного виконання.

Тест №5 похибка 15% довжина повідомлення 100000:

Рисунок 3.18 – Графік часу тесту №5

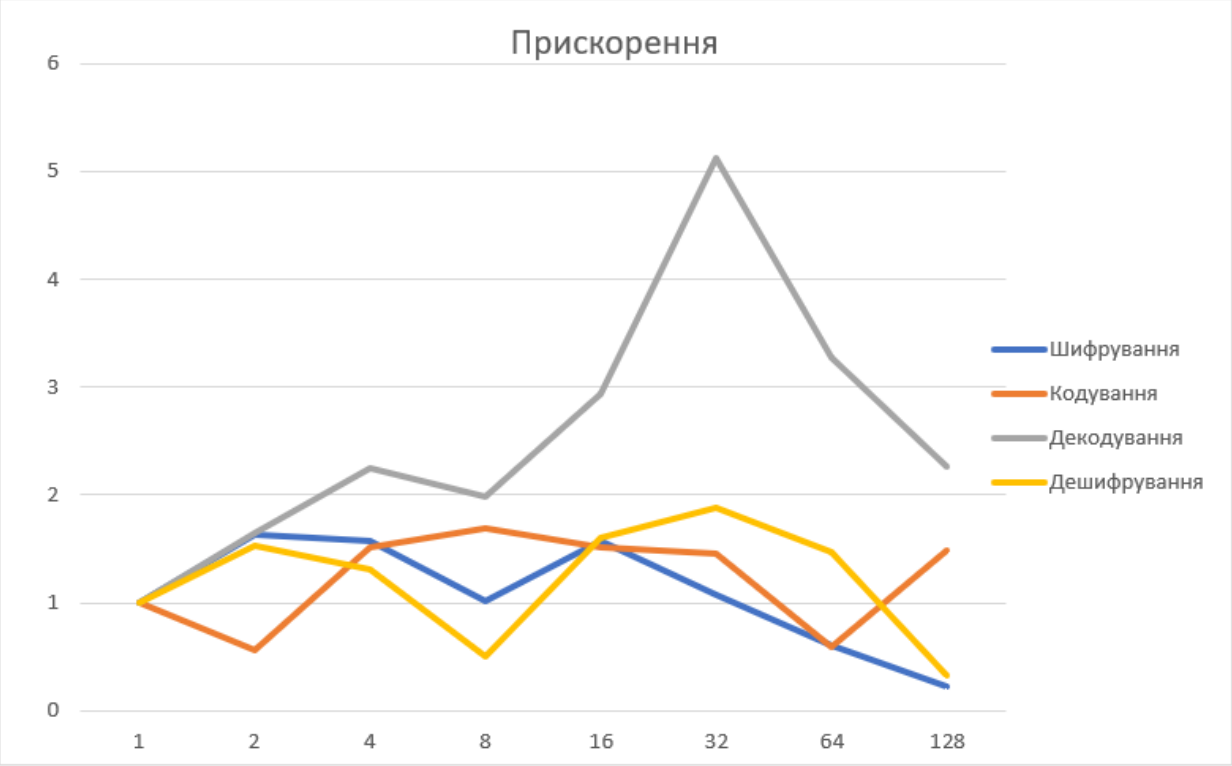


Рисунок 3.19 – Графік прискорення тесту №5

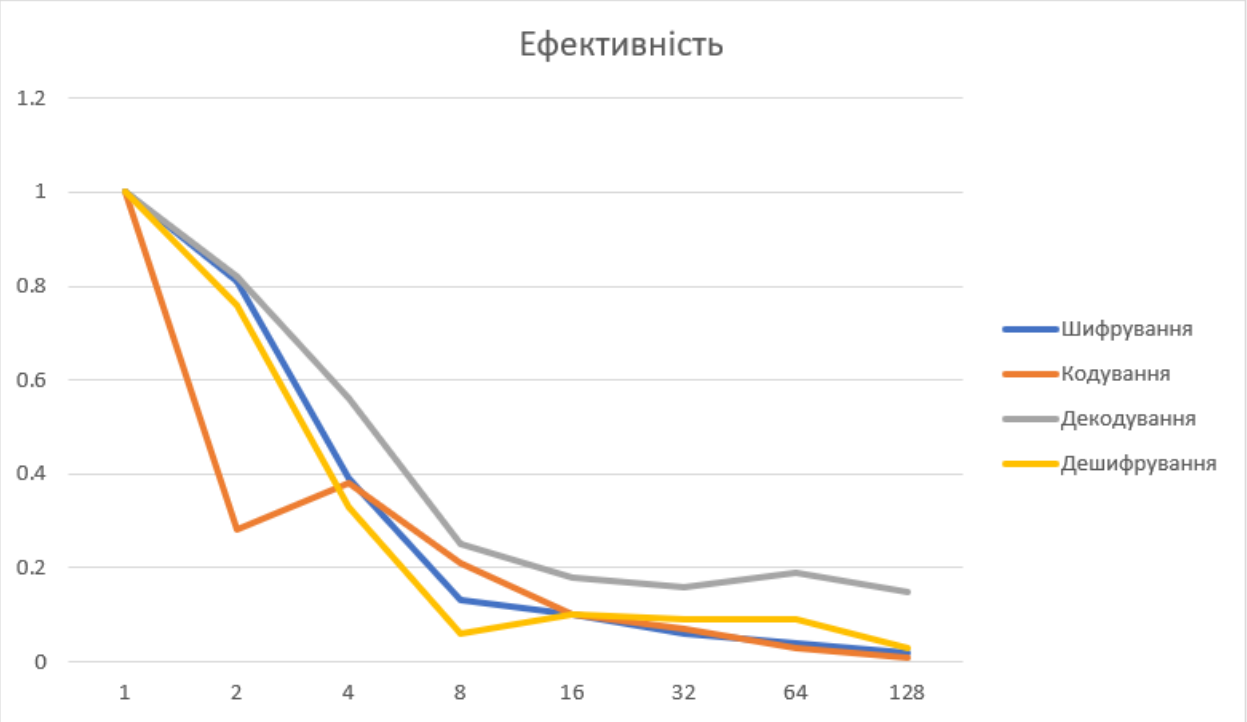


Рисунок 3.20 – Графік ефективності тесту №5

Висновок з тесту:

1. **Прискорення залежить від операції та кількості потоків:** Найкраще прискорення досягається для декодування (до 5.12x при 32 потоках) та шифрування (до 1.63x при 2 потоках). Для дешифрування і кодування ефективність обмежена, а при великій кількості потоків прискорення майже відсутнє.
2. **Шифрування демонструє обмежене масштабування:** Прискорення зростає лише до 1.63x при 2 потоках, після чого ефективність різко падає. При збільшенні кількості потоків понад 16 ефективність стає незначною (0.10 для 16 потоків, 0.04 для 64 потоків).
3. **Декодування є найбільш масштабованою операцією:** Досягає найвищого прискорення 5.12x при 32 потоках. Ефективність знижується повільніше, ніж для інших операцій, але при 64 потоках вона також починає падати (до 0.19).

4. **Кодування має найнижчу ефективність:** При 2 потоках ефективність лише 0.28, а прискорення залишається помірним (1.69 при 8 потоках). При збільшенні кількості потоків понад 16 продуктивність падає майже до нуля.
5. **Дешифрування нестабільне:** Прискорення коливається: найкращий результат — 1.88x при 32 потоках. При 8 і більше потоках ефективність знижується до 0.06–0.09, що вказує на обмеження паралельної обробки.
6. **Значне падіння продуктивності при 64+ потоках:** Для всіх операцій спостерігається різке зростання часу виконання та зниження ефективності. Наприклад, ефективність для шифрування падає до 0.02 при 128 потоках.

Тест №6 похибка 15% довжина повідомлення 1000000:

Рисунок 3.21 – Графік часу тесту №6

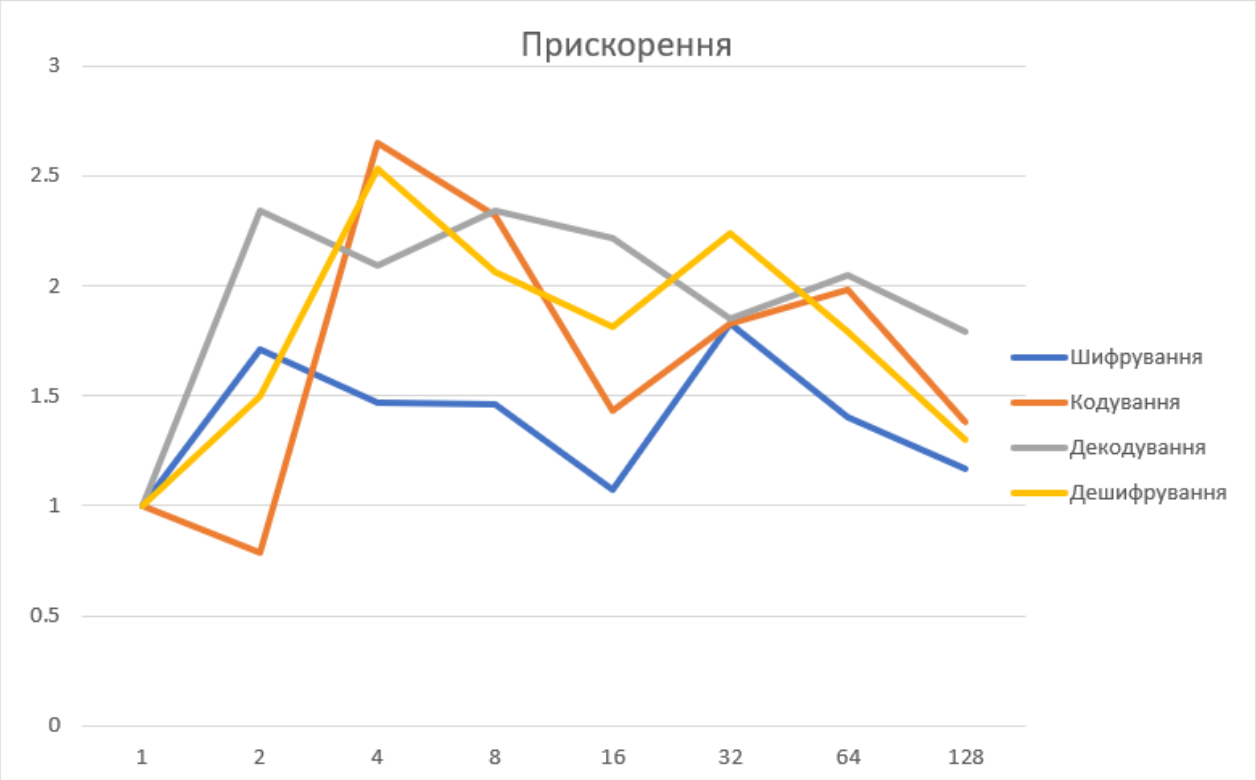


Рисунок 3.22 – Графік прискорення тесту №6

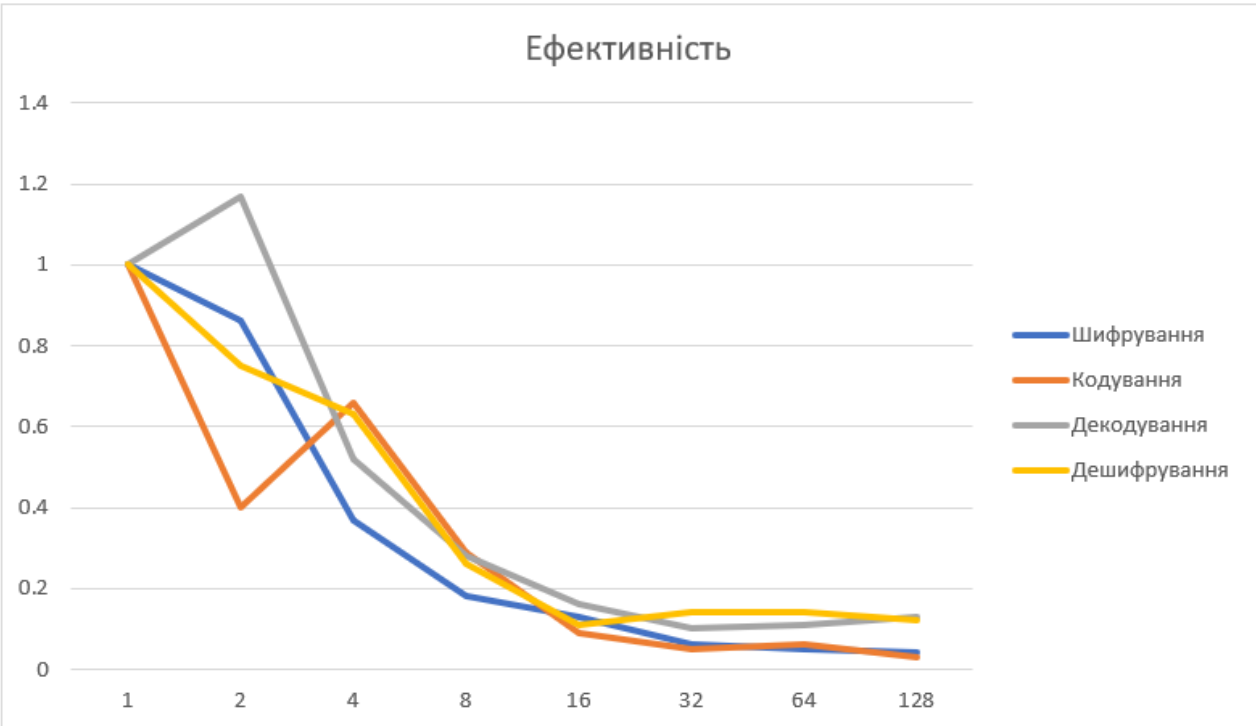


Рисунок 3.23 – Графік ефективності тесту №6

Висновок з тесту:

1. **Шифрування:** Прискорення максимальне при 2 потоках (1.71x), але ефективність падає зі збільшенням кількості потоків. При 16+ потоках масштабування неефективне (ефективність 0.13 при 16 потоках, 0.04 при 128 потоках).
2. **Кодування:** Досягає найкращого прискорення при 4 потоках (2.65x), після чого ефективність суттєво знижується. Використання понад 8 потоків майже не покращує результати.
3. **Декодування:** Демонструє стабільне прискорення до 2.34x при 2 і 8 потоках. Ефективність залишається прийнятною до 16 потоків (0.16), але значно падає при подальшому збільшенні.
4. **Дешифрування:** Найкраще масштабується при 4 потоках (2.53x), але ефективність різко падає при великій кількості потоків (0.14 при 64 потоках, 0.12 при 128 потоках).
5. **Загальні тенденції:** Усі операції демонструють зниження ефективності при понад 8 потоках. Декодування та дешифрування краще масштабується порівняно з шифруванням і кодуванням. Для великих кількостей потоків (64–128) паралелізація стає неефективною для всіх операцій.

Тест №7 похибка 20% довжина повідомлення 10000:

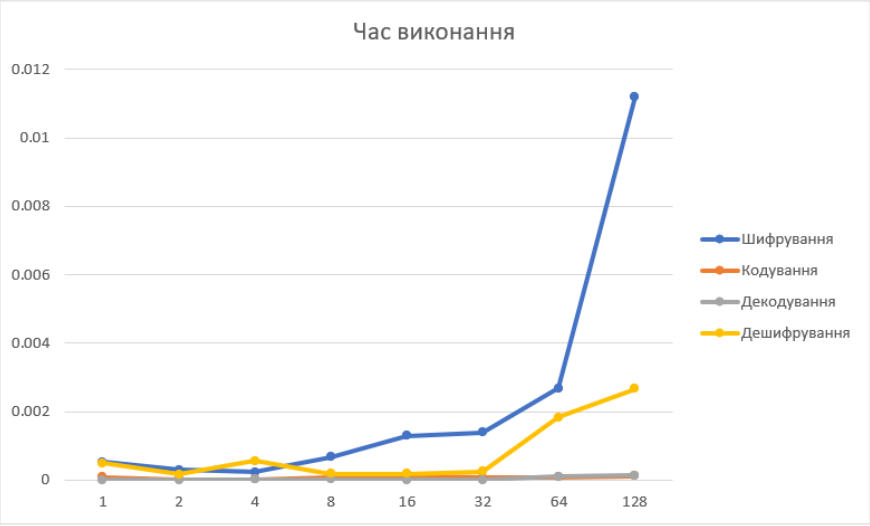


Рисунок 3.24 – Графік часу тесту №7

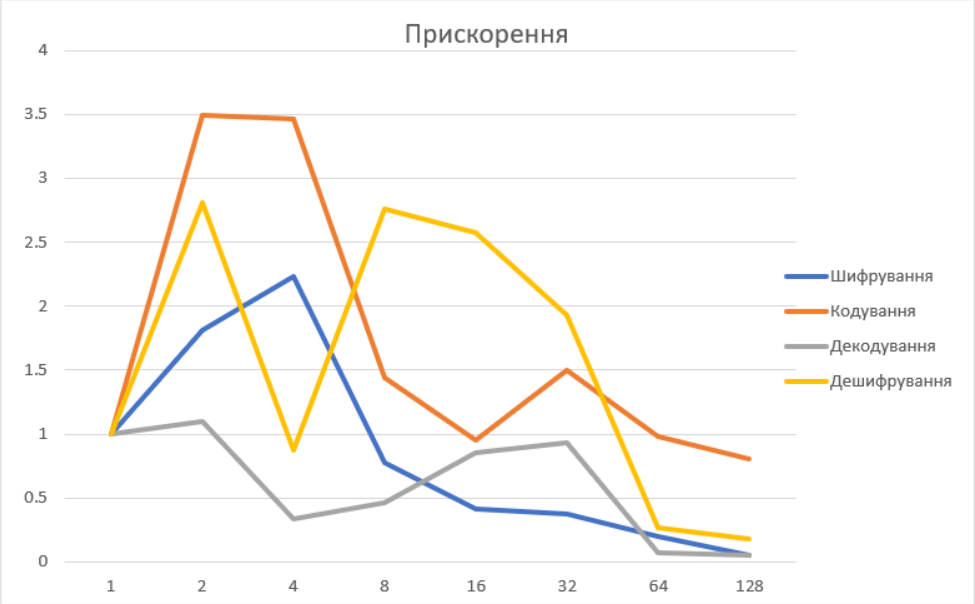


Рисунок 3.25 – Графік прискорення тесту №7

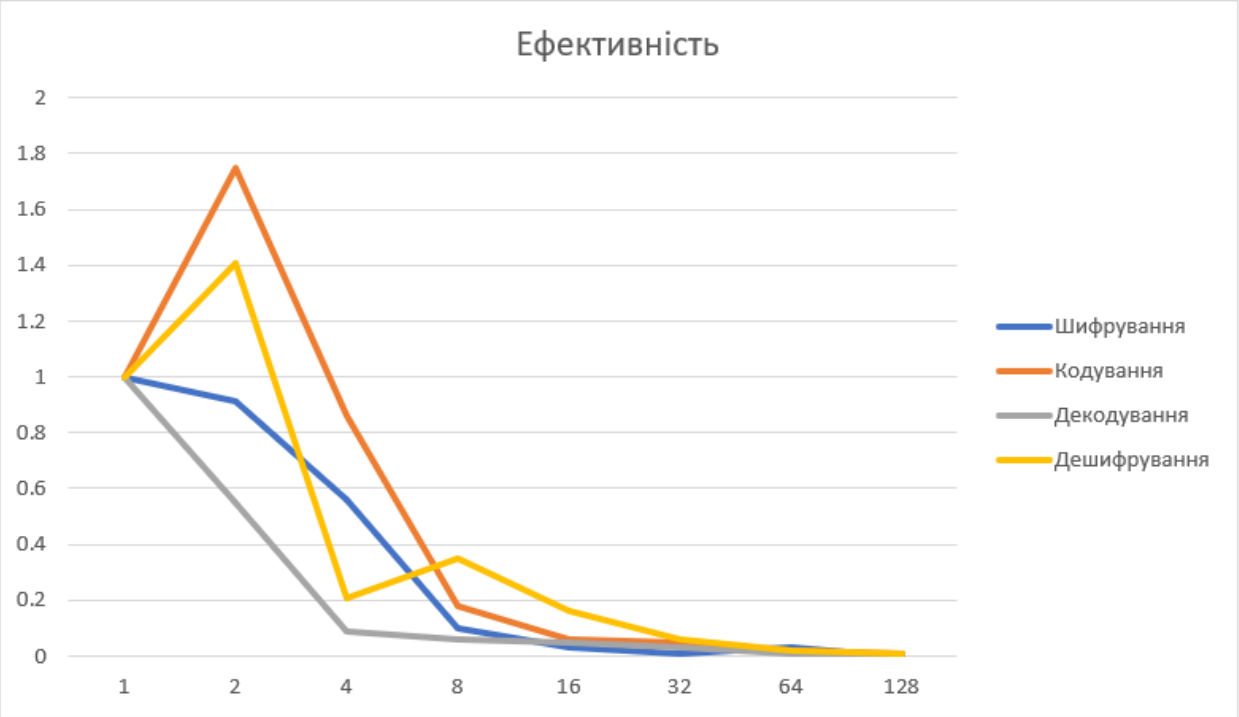


Рисунок 3.26 – Графік ефективності тесту №7

Висновок з тесту:

1. **Шифрування:** Оптимальне прискорення досягається при 2 потоках (1.81x, ефективність 0.91).Після 4 потоків ефективність різко

знижується(0.56 при 4 потоках, 0.03 при 16 потоках).

2. **Кодування:** Найкраще масштабується до 2 потоків (3.49x, ефективність 1.75). Після 4 потоків прискорення зменшується, але залишається значним до 8 потоків.
3. **Декодування:** Прискорення мінімальне навіть при 2 потоках (1.10x, ефективність 0.55). Після 4 потоків ефективність значно падає (0.09 при 4 потоках, 0.01 при 64 потоках).
4. **Дешифрування:** Найкраще масштабується при 2 потоках (2.81x, ефективність 1.41). Значний приріст продуктивності спостерігається до 8 потоків (2.76x, ефективність 0.35). При 16 потоках ефективність помітно знижується.
5. **Загальні тенденції:** Шифрування, кодування та дешифрування показують найбільш ефективну роботу до 4 потоків. Декодування має найгірше масштабування з низьким приростом продуктивності навіть при малій кількості потоків. При збільшенні кількості потоків до 16 і більше ефективність всіх операцій падає до майже нуля.

Тест №8 похибка 20% довжина повідомлення 100000:

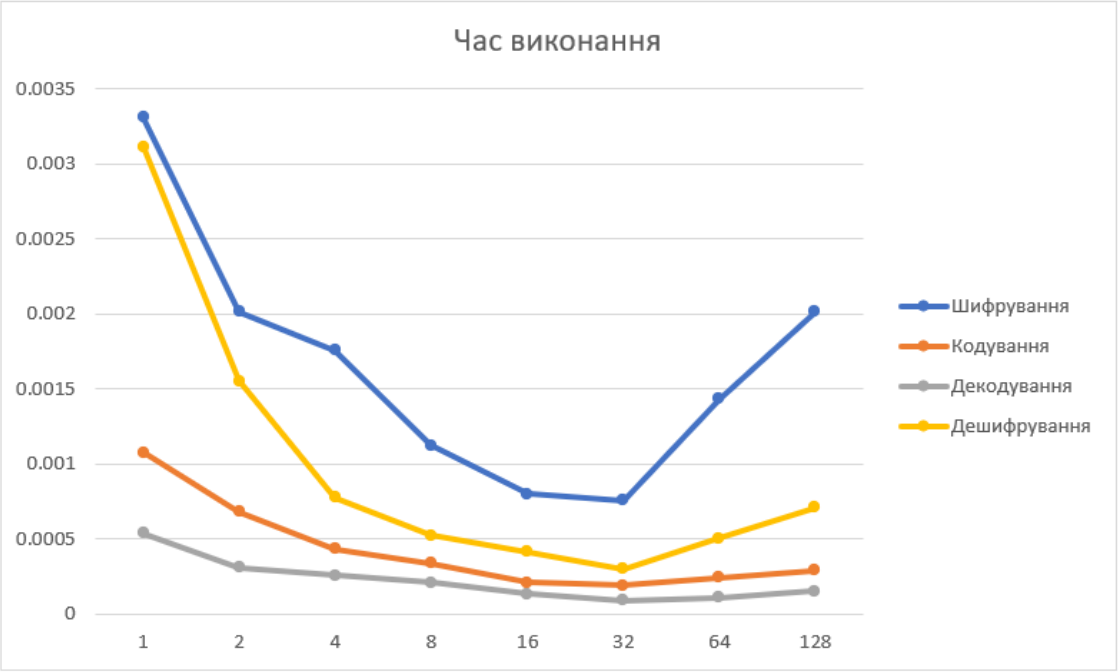


Рисунок 3.27 – Графік часу тесту №8

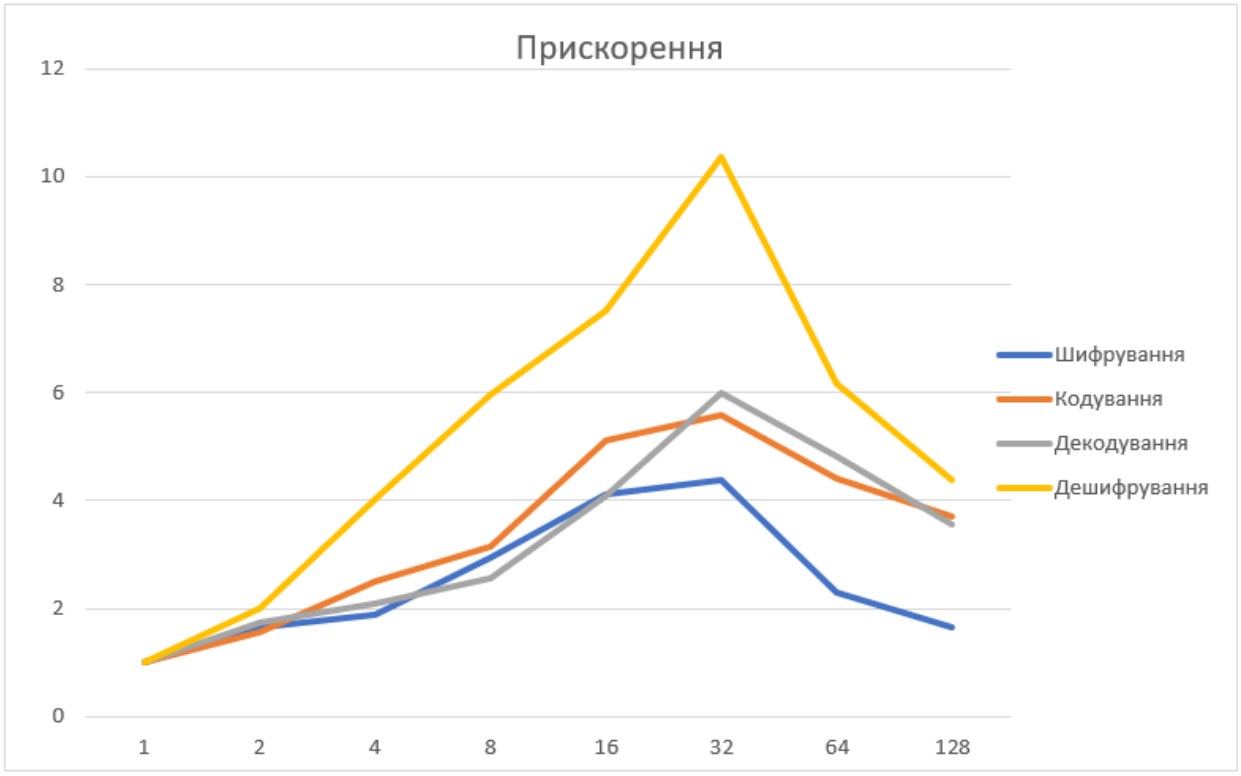


Рисунок 3.28 – Графік прискорення тесту №8

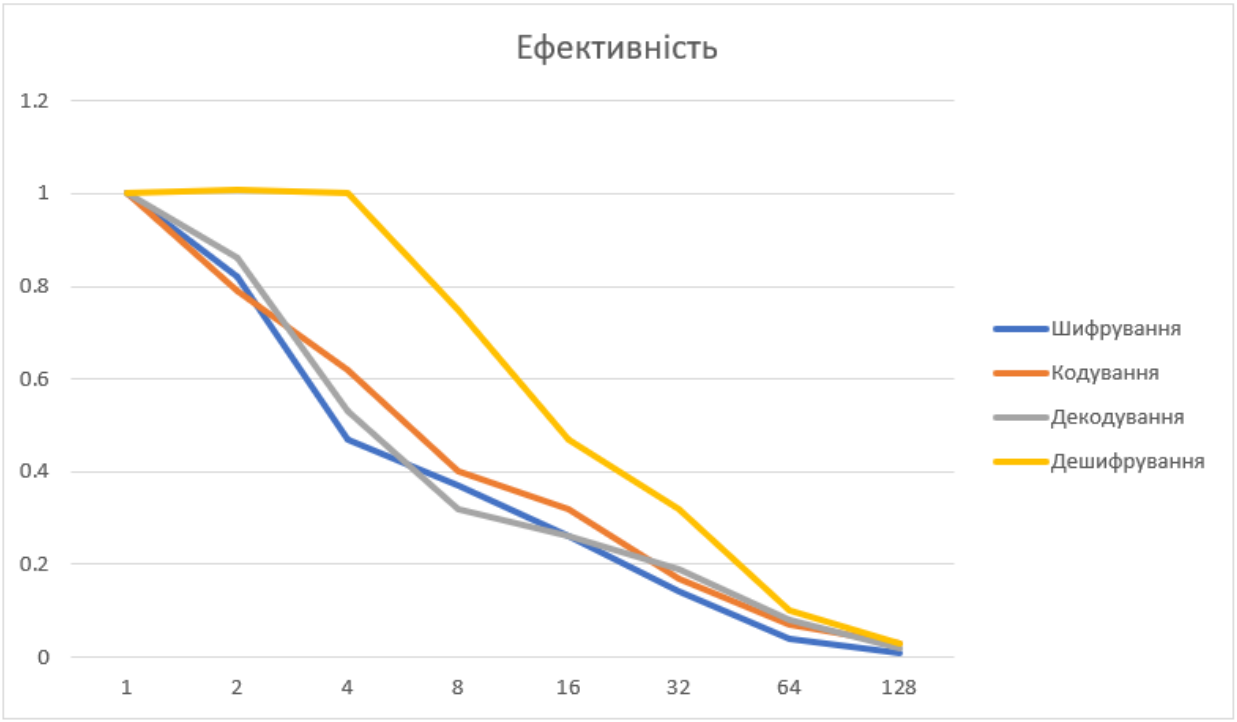


Рисунок 3.29 – Графік ефективності тесту №8

Висновок з тесту:

1. **Шифрування:** Найбільше прискорення (4.38x) досягається при 32 потоках, але ефективність значно падає (0.14). Оптимальний баланс швидкості та ефективності спостерігається до 8 потоків (2.95x, ефективність 0.37).
2. **Кодування:** Найкраще масштабується до 32 потоків (5.59x, ефективність 0.17). Ефективність помітно знижується після 16 потоків.
3. **Декодування:** Найбільше прискорення спостерігається при 32 потоках (6.00x, ефективність 0.19). Ефективність знижується вже після 16 потоків, але залишається на помірному рівні.
4. **Дешифрування:** Дешифрування демонструє найкраще прискорення серед усіх операцій (10.37x при 32 потоках). Ефективність залишається прийнятною до 32 потоків (0.32), після чого знижується.
5. **Загальні тенденції:** Ефективність для всіх операцій знижується при збільшенні кількості потоків, особливо понад 16. Дешифрування має найкращу масштабованість, забезпечуючи значне прискорення навіть при великій кількості потоків. Інші операції демонструють менш ефективну паралелізацію, з оптимальним числом потоків у межах 8–16.

Тест №9 похибка 20% довжина повідомлення 1000000:

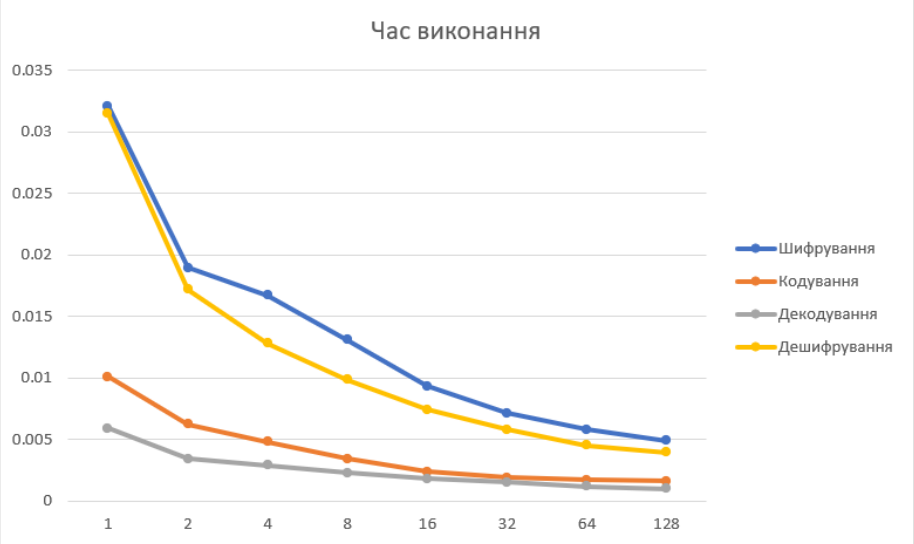


Рисунок 3.30 – Графік часу тесту №9

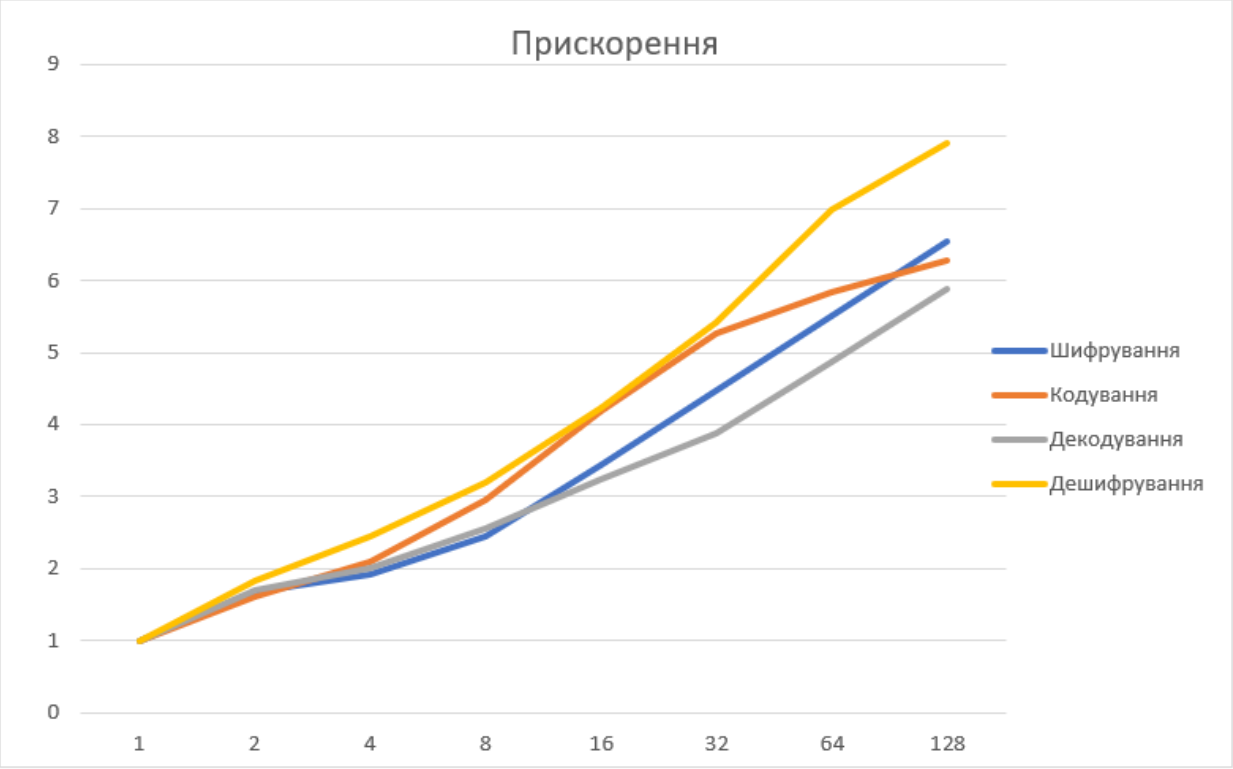


Рисунок 3.31 – Графік прискорення тесту №9

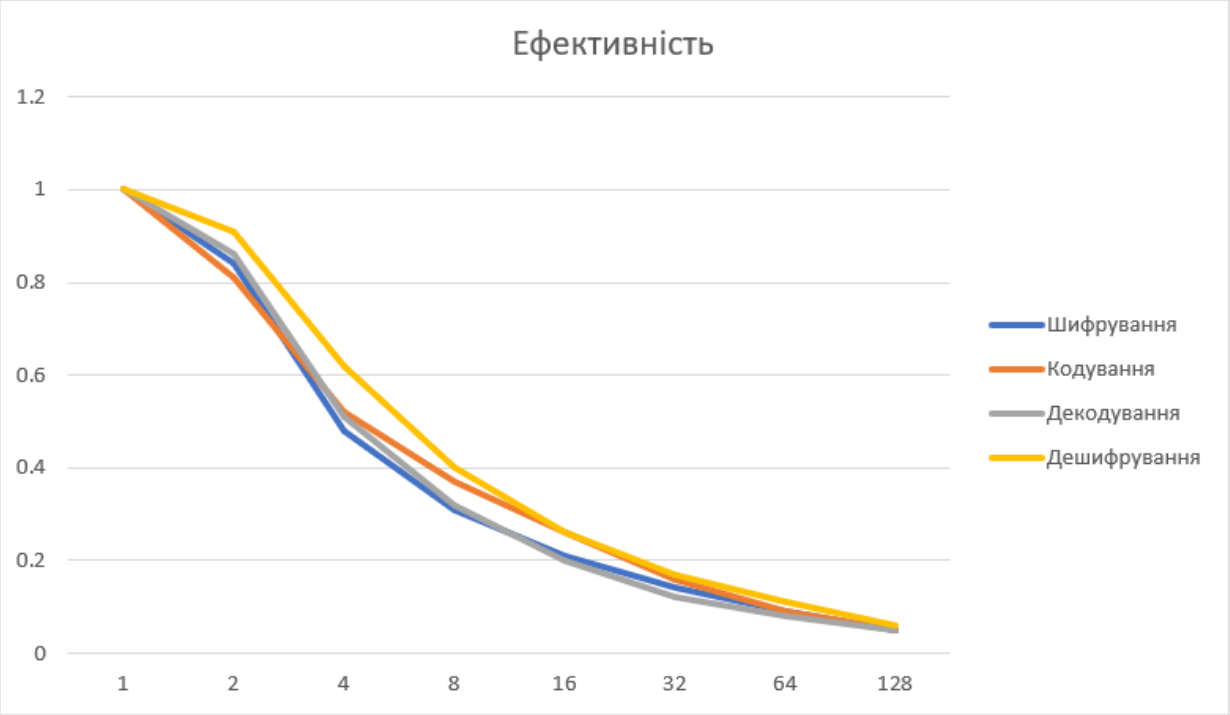


Рисунок 3.32 – Графік ефективності тесту №9

Висновок з тесту:

1. **Шифрування:** Прискорення досягає максимуму (6.53x) при 128 потоках, однак ефективність дуже низька (0.05).Оптимальний баланс швидкості та ефективності спостерігається при 16 потоках (3.43x, ефективність 0.21).
2. **Кодування:** Найбільше прискорення (6.28x) досягається при 128 потоках, але ефективність лише 0.05. Ефективність залишається прийнятною до 32 потоків (5.26x, ефективність 0.16).
3. **Декодування:** Максимальне прискорення (5.88x) спостерігається при 128 потоках, але ефективність низька (0.05). Оптимальний баланс між прискоренням і ефективністю при 16 потоках (3.25x, ефективність 0.20).
4. **Дешифрування:** Найкраще масштабується серед усіх операцій — прискорення 7.91x при 128 потоках, ефективність 0.06. Оптимальний баланс досягається при 32 потоках (5.42x, ефективність 0.17).
5. **Загальні тенденції:** Усі операції демонструють значне падіння ефективності при збільшенні кількості потоків понад 32. Дешифрування найкраще масштабується, забезпечуючи стабільне прискорення навіть при великій кількості потоків. Інші операції демонструють оптимальну продуктивність до 16–32 потоків.

Загальні результати тестів

1. **Продуктивність при збільшенні потоків:** Усі операції демонструють зростання швидкості виконання (прискорення) зі збільшенням кількості потоків. Однак ефективність суттєво знижується при збільшенні потоків понад 16–32.
2. **Масштабованість операцій:** Дешифрування демонструє найкращу масштабованість, стабільно збільшуючи швидкість навіть при 64 і 128 потоках, хоча ефективність при цьому низька. Декодування, кодування та

шифрування менш масштабовані, оптимальний баланс продуктивності та ефективності досягається при 16–32 потоках.

3. **Оптимальний діапазон потоків:** Для більшості операцій (шифрування, кодування, декодування) оптимально використовувати до 16–32 потоків. Дешифрування може виграти від використання більшої кількості потоків (32–64), якщо пріоритетом є максимальна швидкість, а не ефективність.
4. **Ефективність:** Ефективність знижується нелінійно, особливо після 32 потоків, що вказує на перевантаження системи або збільшення витрат на синхронізацію потоків. Для забезпечення прийнятної ефективності варто обмежувати кількість потоків до 16–32.
5. **Складність задач:** Більш складні операції, такі як дешифрування, демонструють краще масштабування, що може бути пов'язане з більшою кількістю операцій, які можна виконувати паралельно. Простішим операціям, таким як декодування, важче зберігати високу ефективність при великій кількості потоків.

6. Переваги реалізації

- **Паралельна обробка:** OpenMP забезпечує масштабованість та високу продуктивність.
- **Гнучкість:** Модель підтримує налаштування рівня помилок, кількості потоків і довжини тексту.
- **Симуляція реальних умов:** Модель дозволяє оцінити ефективність у шумних каналах передачі даних.

7. Обмеження

- Спрощена версія методу Ріда-Соломона не забезпечує ефективногъ корекції помилок.
- Шифрування методом Цезаря є слабким із точки зору сучасної криптографії.

Висновки за розділом 3

Реалізація комбінованої моделі

Проект успішно поєднує методи шифрування (шифр Цезаря) та базові принципи кодування з надлишковістю (метод Ріда-Соломона). Це дозволяє забезпечити як конфіденційність, так і захист даних від пошкоджень.

Ефективність шифрування та паралельної обробки

Завдяки використанню OpenMP для паралельного виконання, модель демонструє високу продуктивність, особливо на багатоядерних системах. Шифрування тексту довжиною до мільйона символів виконується швидко навіть при високих обчислювальних навантаженнях.

Моделювання та аналіз помилок

Система успішно моделює різні рівні помилок у даних, дозволяючи оцінити ефективність виявлення пошкоджень. Хоча проект не реалізує повноцінне виправлення помилок, механізм додавання надлишкових символів демонструє базові принципи кодування для їх подальшого виправлення.

Тестування з різною кількістю потоків

Результати тестів підтвердили ефективність паралельного виконання, що дозволяє масштабувати систему залежно від доступних апаратних ресурсів. Зі збільшенням кількості потоків час обробки суттєво зменшується, досягаючи оптимального значення для великих обсягів даних.

Переваги реалізації

- **Гнучкість:** Можливість адаптації параметрів (довжина тексту, рівень помилок, кількість потоків) робить систему універсальною.
- **Симуляція реальних умов:** Функція моделювання помилок дозволяє імітувати поведінку даних у шумних каналах зв’язку, забезпечуючи реалістичне тестування.
- **Простота інтеграції:** Завдяки модульній структурі система легко розширюється чи інтегрується в інші проекти.

Обмеження проекту

- **Обмежена стійкість до атак:** Шифрування методом Цезаря є слабким з точки зору криптографії, що робить його непридатним для захисту критично важливої інформації.
- **Спрощене кодування:** Реалізація методу Ріда-Соломона є демонстраційною та не здатна виправляти складні пошкодження даних.

Практична значущість

Проект забезпечує базове розуміння принципів одночасного шифрування, кодування та роботи з помилками. Він підходить для навчальних цілей, а також для простих систем, де не потрібні складні криптографічні механізми чи потужні алгоритми захисту.

Загальний висновок

Проект демонструє інтеграцію базових алгоритмів шифрування, кодування та паралельної обробки, що дозволяє ефективно працювати з текстовими даними. Отримані результати підкреслюють потенціал системи для подальшого розвитку, зокрема, вдосконалення кодування для виправлення помилок та застосування більш надійних методів шифрування.

ВИСНОВКИ

Реалізація комбінованої моделі

Проект успішно поєднує методи шифрування (шифр Цезаря) та базові принципи кодування з надлишковістю (метод Ріда-Соломона). Це дозволяє забезпечити як конфіденційність, так і захист даних від пошкоджень.

Ефективність шифрування та паралельної обробки

Завдяки використанню OpenMP для паралельного виконання, модель демонструє високу продуктивність, особливо на багатоядерних системах. Шифрування тексту довжиною до мільйона символів виконується швидко навіть при високих обчислювальних навантаженнях.

Моделювання та аналіз помилок

Система успішно моделює різні рівні помилок у даних, дозволяючи оцінити ефективність виявлення пошкоджень. Хоча проект не реалізує повноцінне виправлення помилок, механізм додавання надлишкових символів демонструє базові принципи кодування для їх подальшого виправлення.

Тестування з різною кількістю потоків

Тестування продуктивності за різної кількості потоків підтвердило масштабованість паралельного виконання. Збільшення кількості потоків суттєво скорочує час обробки, забезпечуючи оптимальну продуктивність для обробки великих обсягів даних. Ця адаптивність робить модель ефективною у використанні доступних апаратних ресурсів і придатною для різних обчислювальних середовищ.**Переваги реалізації**

Гнучкість: Модель дозволяє налаштовувати параметри, такі як довжина тексту, рівень помилок та кількість потоків, що робить її адаптивною до різноманітних сценаріїв використання та обчислювальних середовищ. Ця гнучкість сприяє експериментам із поведінкою системи за різних умов роботи, підтримуючи як практичне застосування, так і навчальні цілі.

Симуляція реальних умов: Можливість імітації поведінки даних у шумних каналах зв’язку створює реалістичне тестове середовище. Ця функція робить модель цінним інструментом для аналізу цілісності даних та надійності їх передачі у телекомунікаціях.

Простота інтеграції: Модульна структура сприяє простій інтеграції в інші проєкти. Архітектура системи також підтримує можливість розширення, наприклад, шляхом впровадження сучасніших алгоритмів шифрування чи механізмів виправлення помилок.

Обмеження проекту

Обмежена стійкість до атак: Шифрування методом Цезаря є слабким з точки зору криптографії, що робить його непридатним для захисту критично важливої інформації.

Спрощене кодування: Реалізація методу Ріда-Соломона є демонстраційною та не здатна виправляти складні пошкодження даних.

Практична значущість

Проєкт забезпечує базове розуміння принципів одночасного шифрування, кодування та роботи з помилками. Він особливо підходить для навчальних цілей, дозволяючи вивчати взаємодію цих компонентів у контрольованому середовищі. Окрім того, система може застосовуватись у простих випадках, де не потрібні складні криптографічні механізми або потужні алгоритми захисту.

Ця реалізація демонструє, як базові техніки можуть ефективно поєднуватись, слугуючи основою для більш складних та потужних систем. Подальше вдосконалення може вирішити існуючі обмеження шляхом інтеграції сильніших методів шифрування та повноцінного механізму виправлення помилок, тим самим розширюючи сферу застосування і стійкість системи.

Проект демонструє інтеграцію базових алгоритмів шифрування, кодування та паралельної обробки, що дозволяє ефективно працювати з текстовими даними. Отримані результати підкреслюють потенціал системи для подальшого розвитку, зокрема, вдосконалення кодування для виправлення помилок та застосування більш надійних методів шифрування. Протягом роботи над кваліфікаційною роботою я ознайомився з різними аспектами створення моделі завадостійких каналів зв'язку, включаючи вибір технологій, розробку комбінованих алгоритмів та оптимізацію обчислювальних процесів. Я вибрав мову програмування C++ через її високу продуктивність та підтримку багатопотокової обробки, що дозволяє ефективно використовувати багатоядерні процесори. Апаратною платформою стало Raspberry Pi, що забезпечує енергоефективність та підтримку периферійних пристроїв. Розроблена модель поєднує шифрування методом Цезаря та алгоритм кодування Ріда-Соломона. Паралельне шифрування за допомогою OpenMP значно підвищило швидкість обробки даних. Алгоритм Ріда-Соломона додає можливість корекції помилок і забезпечує надійність передавання інформації. Симуляція помилок дозволила оцінити ефективність алгоритмів у реальних умовах. Тестування моделі показало високу продуктивність, стійкість до помилок і ефективність при значних спотвореннях у каналах зв'язку. Завдяки цьому проекту я здобув практичний досвід у програмуванні та розробці систем для захисту даних. Модель має потенціал для застосування у телекомунікаційних системах і подальшого вдосконалення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. IMOmath [Електронний ресурс] – режим доступу: URL: https://imomath.com Дата доступу: (дата звернення: 03.10.2024)
2. Forum Raspbery Pi. [Електронний ресурс] – режим доступу: URL: https://forums.raspberrypi.com (дата звернення: 03.10.2024)
3. Stack Overflow [Електронний ресурс] – режим доступу: URL: https://stackoverflow.com (дата звернення: 03.10.2024)
4. Tomverbeure [Електронний ресурс] – режим доступу: URL: ttps://tomverbeure.github.io/2022/08/07/Reed-Solomon.html (дата звернення: 03.10.2024).
5. Towards Data Science [Електронний ресурс] – режим доступу: URL: https://towardsdatascience.com (дата звернення: 03.10.2024).
6. Blahut Richard. Theory and Practice of Error Control Codes. Mass.: Addison-Wesley, 1983. 528 с.
7. James Plank. A tutorial on Reed-Solomon Coding for fault-tolerance in RAID-like systems. [Електронний ресурс] – режим доступу: URL: http://www.cs.utk.edu/~plank/plank/papers/CS-96-332.pdf (дата звернення: 03.10.2024).
8. Tom Moore. REED-SOLOMON PACKAGE. [Електронний ресурс] – режим доступу: URL: https://essay.utwente.nl/58502/1/scriptie_A_Franssens.pdf (дата звернення: 03.10.2024).
9. Ross N. Williams. A painless guide to CRC error detection algorithms. [Електронний ресурс] – режим доступу: URL: https://ceng2.ktu.edu.tr/~cevhers/ders_materyal/bil311_bilgisayar_mimarisi/supplementary_docs/crc_algorithms.pdf (дата звернення: 03.10.2024).
10. Sebastian A., Bonna K. Reed-Solomon Encoder and Decoder / Under Prof. Predrag Spasojevic. Electrical and Computer Engineering Department.

Rutgers, The State University of New Jersey. [Електронний ресурс] – режим доступу: URL: https://content.sakai.rutgers.edu/access/content/user/ak892/Reed-SolomonProjectReport.pdf (дата звернення: 03.10.2024).

11. Васюра А. С. Техніка передавання аналогової та дискретної інформації / Васюра А. С.. – Вінниця: ВДТУ, 1998. – 218 с.
12. Тарнавський Ю. А. Технології захисту інформації [Електронний ресурс] : підручник для студ. спеціальності 122 «Комп'ютерні науки», спеціалізацій «Інформаційні технології моніторингу довкілля», «Геометричне моделювання в інформаційних системах» / Ю. А. Тарнавський ; КПІ ім. Ігоря Сікорського. – Київ : КПІ ім. Ігоря Сікорського, 2018. – 162 с. URL: https://web.archive.org/web/20211203160625/https://ela.kpi.ua/bitstream/123456789/23896/1/TZI_book.pdf (дата звернення: 03.10.2024).
13. Шифр Цезаря - як він працює [Електронний ресурс] – режим доступу: URL: https://uk.php.brj.cz/sifr-cezarya-yak-vin-pracyue (дата звернення: 03.10.2024).
14. Raspberry Pi Interview With Eben Upton [Електронний ресурс] – режим доступу: URL: https://web.archive.org/web/20111210104303/http://www.robots.net/article/3215.html (дата звернення: 03.10.2024).
15. RPi Hub [Електронний ресурс] – режим доступу: URL: https://elinux.org/RPi_Hub (дата звернення: 03.10.2024).
16. OpenMP Architecture Review Board [Електронний ресурс] – режим доступу: URL: http://www.openmp.org/ (дата звернення: 03.10.2024)
17. The Community of OpenMP Users, Researchers, Tool Developers and Providers [Електронний ресурс] – режим доступу: URL: http://www.compunity.org/(дата звернення: 03.10.2024)

18. OpenMP Application Program Interface Version 3.0 May 2008 [Електронний ресурс] – режим доступу: URL: (http://www.openmp.org/mp-documents/spec30.pdf (дата звернення: 03.10.2024)
19. Barbara Chapman, Gabriele Jost, Ruud van der Pas. Using OpenMP: portable shared memory parallel programming (Scientific and Engineering Computation). Cambridge, Massachusetts: The MIT Press., 2008. - 353 с.
20. Мочурад Л. І., Бойко Н. І. Використання технології OpenMP для розрахунку електростатичного поля систем електронної оптики Науковий вісник НЛТУ України. 2019. Т. 29, № 3. С. 125–132. https://doi.org/10.15421/40290326. (Дата звернення: 03.10.2024)

ДОДАТКИ

Додаток А

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет імені В. Н. Каразіна

Навчально-науковий інститут комп'ютерних наук та штучного інтелекту
Кафедра комп'ютерних систем та робототехніки
Рівень вищої освіти (освітньо-кваліфікаційний рівень) **Магістр**
Галузь знань: 17 – Електроніка, автоматизація та електронні комунікації
Спеціальність: 174 – Автоматизація, комп'ютерно-інтегровані технології та робототехніка
Освітня програма «Комп'ютеризовані системи управління та автоматика»

ЗАТВЕРДЖУЮ
в.о. завідувача комп'ютерних
систем та робототехніки
к. ф.-м. н., доц. ХРУСЛОВ М. М.
«12» вересня 2024 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Яровенко Михайла Юрійовича
(прізвище, ім'я, по батькові студента)

1. Тема роботи **Модель завадостійких каналів зв'язку мобільних пристроїв з паралельною обробкою інформації**

керівник роботи Толстолузька Олена Геннадіївна, доктор технічних наук, с.н.с.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету ***№ 4101-5/3657 від 12 листопада 2024 року***

2. Строк подання студентом роботи ***30 листопада 2024 року***

3. Перелік питань, які потрібно розробити)
1. Аналіз проблеми та постановка задачі створення моделі завадостійких каналів зв'язку мобільних пристроїв з паралельною обробкою інформації
2. Вибір предметної області використання паралельного обчислення.
3. Вибір технічних засобів для реалізації моделі.
4. Створення програмної реалізації обчислювального процесу.
5. Перевірка на працездатність та ефективність програмної реалізації.
6. Створення пояснювальної записки.

4. План роботи

№ з/п	Назви етапів роботи	Термін виконання етапів роботи
1	Підбір літератури, створення бази для розробки моделі	03.09.2024 – 09.09.2024
2	Порівняння обраного підходу з аналогами, аналіз переваг і недоліків	10.09.2024 – 16.09.2024
3	Створення моделі	17.09.2024 – 23.09.2024
4	Аналіз технологій для реалізації моделі	24.09.2024 – 30.09.2024
5	Розробка програмного продукту на основі обраних технологій	01.10.2024 – 07.12.2024
6	Випробування програмного виробу, виправлення помилок.	08.10.2024 – 14.10.2024
7	Аналіз результатів випробувань	15.10.2024 – 21.10.2024
8	Розробка рекомендацій щодо застосування програмного виробу.	22.10.2024 – 28.10.2024
9	Підготовка до попереднього захисту роботи	29.10.2024 – 04.11.2024
10	Створення пояснювальної записки	05.11.2024 – 11.11.2024
11	Представлення кваліфікаційної роботи науковому керівнику	12.11.2024

5. Дата видачі завдання ***05 вересня 2024року.***

Студент М.Ю. Яровенко
ініціали, прізвище
підпис

Керівник роботи О.Г. Толстолузька
ініціали, прізвище
підпис

Додаток Б

Технічне завдання

на розробку програмного виробу

«модель завадостійких каналів зв’язку мобільних пристроїв з паралельною обробкою інформації»

Назва розділу	Назва і зміст підрозділу
1. Введення	1.1. Назва програмного виробу: Модель завадостійких каналів зв’язку з паралельною обробкою інформації. 1.2. Галузь застосування: Телекомунікації, системи зв’язку, інформаційні технології.
2. Підстава для розробки	2.1. Навчальний план за спеціальністю 174 – Автоматизація, комп’ютерно-інтегровані технології та робототехніка. 2.2. Завдання на кваліфікаційну роботу від 12 листопада 2024 року № 4101-5/3657 (Представити як Додаток А до пояснювальної записки до кваліфікаційної роботи).
3. Призначення розробки	3.1. Мета розробки програмного вироби: Розробка моделі завадостійких каналів зв’язку з паралельною обробкою інформації для підвищення ефективності передачі даних у умовах несприятливих каналів. 3.2. Призначення програмного виробу: Забезпечення надійної та швидкої передачі даних через завадостійкі канали зв’язку. 3.3. Вихідні дані для розробки: Дані для моделювання завад у каналі зв’язку, алгоритми шифрування та кодування.
4. Технічні вимоги до	4.1. Вимоги до функціональних характеристик: Можливість ефективної обробки даних, забезпечення паралельної обробки з використанням OpenMP,

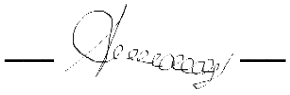
програмного виробу	підтримка багатоядерних процесорів. 4.2. Вимоги до надійності: Висока стабільність роботи без помилок і відмов в умовах реального використання. 4.3. Вимоги до умов експлуатації: Підтримка різних платформ, зокрема Raspberry Pi. 4.4. Вимоги до складу і параметрів технічних засобів: Мінімальні вимоги до апаратного забезпечення для роботи з великими обсягами даних. 4.5. Вимоги до інформаційної та програмної сумісності: Підтримка операційних систем Linux (Raspberry Pi), Windows, macOS для розробки та тестування.
5. Вимоги до програмної документації	Програмною документацією до виробу «Модель завадостійких каналів зв'язку з паралельною обробкою інформації» є: 1) Справжнє Технічне завдання на розробку програмного виробу (представити у вигляді Додатку Б до пояснювальної записки до кваліфікаційної роботи). 2) Програму і методику випробувань розробленого програмного виробу (представити у вигляді Додатку В до пояснювальної записки до кваліфікаційної роботи).
6. Техніко-економічні показники	Оцінка економічної ефективності непотрібна.
7. Стадії і етапи розробки	Аналіз літератури за темою: 01.09.2024 Аналіз та порівняння аналогічних існуючих систем: 05.09.2024 Дослідження принципів роботи моделей завадостійких каналів: 10.09.2024 Проектування алгоритмів шифрування та кодування: 20.09.2024 Розробка паралельної обробки даних з використанням OpenMP: 01.10.2024 Тестування моделі на Raspberry Pi: 10.10.2024

	Тестування ефективності системи при високому рівні помилок: 20.10.2024 Завершення тестування та складання рекомендацій: 01.11.2024 Представлення пояснювальної записки: 10.11.2024 Представлення кваліфікаційної роботи: 30.11.2024
8. Порядок контролю і приймання	1) Випробування програмного виробу відповідно до Програми і методики випробувань провести на базі комп'ютерного класу. 2) Захист розробленого програмного виробу провести на засіданні атестаційної комісії.

Виконавець

студент групи КУ- 61

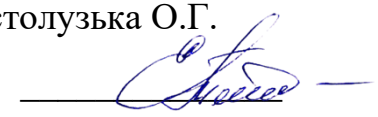
Яровенко М.Ю.



Замовник

д. т. н., с.н.с.,

Толстолузька О.Г.



Додаток В

Програма та методика випробувань програмного виробу

«модель завадостійких каналів зв’язку мобільних пристроїв з паралельною обробкою інформації»

Програмна реалізація цього проекту полягає в моделюванні процесу зашифрованої передачі даних через канали зв’язку з використанням шифру Цезаря і кодування Ріда-Соломона. Ось основні частини програми:

Основні функції:

1. **generateKey**:

Генерує фіксований ключ для шифрування. Ключ є вектором цілих чисел, які використовуються як зміщення для шифрування в методі Цезаря.

2. **generateRandomText**:

Генерує випадковий текст заданої довжини, що містить великі і малі літери латиниці, цифри та інші символи ASCII.

3. **shiftCharEncrypt/shiftCharDecrypt**:

Функції для шифрування та дешифрування символів за допомогою зміщення в межах діапазону ASCII. Вони використовують операцію модулювання для забезпечення циклічності в межах ASCII (0-127).

4. **caesarEncrypt/caesarDecrypt**:

Функції для шифрування та дешифрування тексту за допомогою методу Цезаря, використовуючи ключ. Кожен символ тексту зміщується на відповідну величину, що визначається циклічно обраним ключем.

Використовується OpenMP для паралельного обчислення шифрування та дешифрування, що дозволяє зменшити час обробки великих обсягів тексту.

5. **reedSolomonEncode/reedSolomonDecode**:

Спрощена реалізація кодування і декодування за допомогою алгоритму Ріда-Соломона. Кодування додає паритетні символи до даних, що дозволяє здійснювати корекцію помилок.

Цей алгоритм застосовується для надійної передачі даних через завадостійкі канали зв’язку.

6. **introduceErrors**:

Вводить випадкові помилки в закодовані дані з заданим рівнем помилок (errorRate). Це симулює можливі спотворення даних у каналі зв’язку.

7. **bruteForceCorrection**:

Проста функція для корекції помилок шляхом перебору всіх можливих значень для кожного елементу масиву, що було пошкоджене під час передачі.

Основна частина програми:

1. **Генерація випадкового тексту**:

Генерується великий випадковий текст (1 мільйон символів) для подальшого шифрування та тестування.

2. **Тестування з різною кількістю потоків**:

Для кожної кількості потоків (від 1 до 128) вимірюється час виконання ключових етапів процесу: шифрування, кодування, введення помилок, декодування і дешифрування.

Для паралельного виконання використовується OpenMP.

3. **Шифрування та дешифрування**:

Шифрування тексту виконується за допомогою шифру Цезаря, а потім текст декодується за допомогою цього ж методу.

Перевірка правильності дешифрування — результат порівнюється з початковим текстом.

4. **Кодування і декодування Ріда-Соломона**:

Додатково виконується кодування і декодування даних з додаванням помилок, що дозволяє оцінити ефективність корекції помилок.

Вимірювання часу:

Для кожного етапу (шифрування, кодування, введення помилок, декодування, дешифрування) вимірюється час за допомогою **std::chrono**, що дає змогу оцінити продуктивність програми при різних конфігураціях кількості потоків.

Паралельна обробка:

- OpenMP використовується для паралельного виконання операцій шифрування та дешифрування тексту, що дозволяє ефективно використовувати багатоядерні процесори і зменшує час обробки великих обсягів даних.

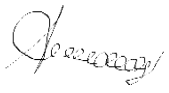
Результати:

- Після виконання всіх етапів програма виводить час, витрачений на кожен етап, а також перевіряє успішність дешифрування (чи відповідає відновлений текст початковому).

Випробування

Для випробувань було проведено по 3 тести з кількістю символів: 10 000, 100 000 та 1 000 000, для різного відсотку помилок: 10%, 15% та 20%.

Висновки: всі тести було проведено та визнано успішними. Тести показали що паралельне обчислення здатне зменшити час на виправлення помилок у пошкоджених повідомленнях, що збільшує завадостійкість каналів зв’язку.

Виконавець: студент групи КУ-61, Яровенко М.Ю. 

Додаток Г

Лістинг коду

```
#include <iostream>
#include <omp.h>
#include <string>
#include <vector>
#include <chrono>
#include <random>
#include <algorithm>

// Function to generate a fixed encryption key
std::vector<int> generateKey() {
    return { 12, -45, 38, 7, 91, -28, 54, 19, -17, 33, 21, -99, 47, 63, -84,
             50, 26, 14, -52, 30, 60, 78, -37, 3, -65, 88, 49, 82, 11, -31,
             29, -22, 95, 68, -9, 73, -77, 40, 85, 34, -58, 25, 56, 10, -96,
             61, 17, -14, 2, -81, 92, 36, -13, 76, 41, 53, -47, 5, 97, -60,
             64, -3, 66, 43, -21, 27, 87, -18, 72, 48, -91, 15, 62, -6, 83,
             -19, 90, 31, 79, -40, 16, -24, 58, -80, 8, 89, -34, 74, 20, -71,
             39, 6, -26, 44, 23, 13, -10, 52, -50 }; // Fixed key
}

// Function to generate a random text of a specified length
std::string generateRandomText(int length) {
    const char charset[] =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456
789";
    std::string result;
    result.reserve(length);
```

```
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dist(0, sizeof(charset) - 2); // -2 to avoid the null terminator

    for (int i = 0; i < length; ++i) {
        result += charset[dist(gen)];
    }
    return result;
}

// Function to encrypt characters using only ASCII symbols
char shiftCharEncrypt(char c, int shift) {
    return (c + shift + 128) % 128; // Shift within the ASCII range
}

// Function to decrypt characters using only ASCII symbols
char shiftCharDecrypt(char c, int shift) {
    return (c - shift + 128) % 128; // Reverse shift within the ASCII range
}

// Function to encrypt text using the key
std::string caesarEncrypt(const std::string& text, const std::vector<int>& key) {
    std::string result = text;
    int n = text.length();

#pragma omp parallel for
    for (int i = 0; i < n; ++i) {
```

```
        int shift = key[i % key.size()]; // Use key shifts in a cyclic manner
        result[i] = shiftCharEncrypt(text[i], shift);
    }
    return result;
}

// Function to decrypt text using the key
std::string caesarDecrypt(const std::string& text, const std::vector<int>& key) {
    std::string result = text;
    int n = text.length();

#pragma omp parallel for
    for (int i = 0; i < n; ++i) {
        int shift = key[i % key.size()]; // Use key shifts in a cyclic manner
        result[i] = shiftCharDecrypt(text[i], shift);
    }
    return result;
}

// Reed-Solomon encode function (simplified example)
std::vector<int> reedSolomonEncode(const std::vector<int>& data, int
paritySymbols) {
    std::vector<int> encodedData = data;
    encodedData.resize(data.size() + paritySymbols, 0); // Resize to hold parity
symbols

    // Encoding (placeholder logic, as real RS encoding requires polynomial
arithmetic)
    for (int i = 0; i < paritySymbols; ++i) {
```

```
        encodedData[data.size() + i] = data[i % data.size()]; // Simple redundancy
example
    }
    return encodedData;
}

// Reed-Solomon decode function (simplified example)
std::vector<int> reedSolomonDecode(const std::vector<int>& encodedData, int
paritySymbols) {
    std::vector<int> decodedData(encodedData.begin(), encodedData.end() -
paritySymbols);
    // Decode logic (error correction would be here)
    return decodedData;
}

// Function to introduce errors into the data
void introduceErrors(std::vector<int>& data, float errorRate) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> dist(0.0, 1.0);

    for (size_t i = 0; i < data.size(); ++i) {
        if (dist(gen) < errorRate) {
            data[i] = (data[i] + 1) % 256; // Introduce an error by flipping a bit
        }
    }
}

// Function to brute-force correct errors
```

```
void bruteForceCorrection(std::vector<int>& data, const std::vector<int>&
original, int maxErrors) {
    std::vector<int> temp(data);

    // Change int to std::vector<int>::size_type
    for (std::vector<int>::size_type i = 0; i < data.size(); ++i) {
        for (int j = 0; j < 256; ++j) { // Check all possible ASCII values
            temp[i] = j; // Try changing current value to j

            // Check if corrected data matches original
            if (std::equal(temp.begin(), temp.end(), original.begin())) {
                data = temp; // Corrected data
                return; // Exit after correcting
            }
        }
        temp = data; // Reset temp to original data for the next iteration
    }
}

// Main function
int main() {
    // Generate random text of 10000 characters
    std::string text = generateRandomText(1000000);
    std::cout << "Generated random text of 1000000 characters." << std::endl;
    // Using a fixed encryption key
    std::vector<int> key = generateKey();
    // Test for different number of threads
    for (int num_threads = 1; num_threads <= 128; num_threads *= 2) { // Change
number of threads from 1, 2, 4, 8, 16 ...
```

```
        std::cout << "\nTesting with " << num_threads << " threads." << std::endl;
        // Set the number of threads for parallel execution
        omp_set_num_threads(num_threads);
        // Convert text to integer vector for Reed-Solomon
        std::vector<int> data(text.begin(), text.end());
        // Measure time for encryption
        auto start_encrypt = std::chrono::high_resolution_clock::now();
        std::string encrypted = caesarEncrypt(text, key);
        auto end_encrypt = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> encryption_time = end_encrypt - start_encrypt;
        std::cout << "Encryption completed in: " << encryption_time.count() << "
seconds." << std::endl;
        // Measure time for encoding
        auto start_encode = std::chrono::high_resolution_clock::now();
        auto encodedData = reedSolomonEncode(data, 4); // Reed-Solomon encoding
with 4 parity symbols
        auto end_encode = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> encoding_time = end_encode - start_encode;
        std::cout << "Encoding completed in: " << encoding_time.count() << "
seconds." << std::endl;
        // Measure time for introducing errors
        auto start_errors = std::chrono::high_resolution_clock::now();
        float errorRate = 0.2; // 10% error rate
        introduceErrors(encodedData, errorRate);
        auto end_errors = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> errors_time = end_errors - start_errors;
        std::cout << "Error introduction completed in: " << errors_time.count() << "
seconds." << std::endl;
```

```
        // Measure time for decoding
        auto start_decode = std::chrono::high_resolution_clock::now();
        std::vector<int> decodedData = reedSolomonDecode(encodedData, 4);
        auto end_decode = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> decoding_time = end_decode - start_decode;
        std::cout << "Decoding completed in: " << decoding_time.count() << "
seconds." << std::endl;
        // Measure time for decryption
        auto start_decrypt = std::chrono::high_resolution_clock::now();
        std::string decrypted = caesarDecrypt(encrypted, key);
        auto end_decrypt = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> decryption_time = end_decrypt - start_decrypt;
        std::cout << "Decryption completed in: " << decryption_time.count() << "
seconds." << std::endl;
        // Verify if decryption matches the original text
        if (text == decrypted) {
            std::cout << "Decryption successful, original text restored." << std::endl;
        }
        else {
            std::cout << "Decryption failed, original text does not match." << std::endl;
        }
    }
    return 0;
}
```

Додаток Д

Таблиці результатів виконання тестів

Таблиця Д1. 10% 10 000 символів:

Кількість потоків	Операція	Час (с)	Прискорення	Ефективність
1	Шифрування	0.000524588	1	1
1	Дешифрування	0.000496736	1	1
1	Кодування	0.000102647	1	1
1	Декодування	0.000024	1	1
2	Шифрування	0.000286146	1.83	0.92
2	Дешифрування	0.000171628	2.89	1.45
2	Кодування	0.000024666	4.12	2.06
2	Декодування	0.000007	3.43	1.72
4	Шифрування	0.000235405	2.23	0.56
4	Дешифрування	0.000109721	4.52	1.13
4	Кодування	0.000026815	3.83	0.96
4	Декодування	0.000014518	1.66	0.42
8	Шифрування	0.000529902	0.99	0.12
8	Дешифрування	0.000133221	3.73	0.46
8	Кодування	0.000035574	2.86	0.36
8	Декодування	0.000014425	1.71	0.21
16	Шифрування	0.000744122	0.70	0.09
16	Дешифрування	0.000172443	2.87	0.18
16	Кодування	0.000074184	1.39	0.09
16	Декодування	0.000007259	3.32	0.21
32	Шифрування	0.00139678	0.38	0.06
32	Дешифрування	0.000217349	2.09	0.07
32	Кодування	0.000065592	1.56	0.05
32	Декодування	0.0000075	3.21	0.10
64	Шифрування	0.00279682	0.19	0.01
64	Дешифрування	0.000689974	0.72	0.03
64	Кодування	0.000090221	1.14	0.02
64	Декодування	0.00001511	1.62	0.03
128	Шифрування	0.00597849	0.09	0.01
128	Дешифрування	0.00126428	0.39	0.02
128	Кодування	0.000133591	0.77	0.01
128	Декодування	0.000022177	2.22	0.01

Таблиця Д2. 10% 100 000 символів:

Кількість потоків	Операція	Час (с)	Прискорення	Ефективність
1	Шифрування	0.0039769	1	1

1	Дешифрування	0.00295145	1	1
1	Кодування	0.00122499	1	1
1	Декодування	0.000532457	1	1
2	Шифрування	0.00227181	1.75	0.88
2	Дешифрування	0.00165056	1.79	0.90
2	Кодування	0.00152902	0.80	0.40
2	Декодування	0.000319145	1.67	0.83
4	Шифрування	0.00215559	1.85	0.47
4	Дешифрування	0.00221272	1.33	0.33
4	Кодування	0.000710438	1.73	0.43
4	Декодування	0.000399274	1.33	0.33
8	Шифрування	0.00284406	1.40	0.16
8	Дешифрування	0.00219877	1.35	0.17
8	Кодування	0.000975712	1.26	0.16
8	Декодування	0.00089725	0.59	0.07
16	Шифрування	0.0113668	0.35	0.12
16	Дешифрування	0.00242292	1.22	0.08
16	Кодування	0.000641586	1.91	0.12
16	Декодування	0.00042744	1.25	0.08
32	Шифрування	0.00285373	1.39	0.06
32	Дешифрування	0.00281532	1.05	0.07
32	Кодування	0.00069266	1.77	0.06
32	Декодування	0.000235831	2.26	0.07
64	Шифрування	0.00595679	0.67	0.01
64	Дешифрування	0.0014915	2.00	0.03
64	Кодування	0.00206659	0.59	0.01
64	Декодування	0.000423014	1.26	0.02
128	Шифрування	0.0135614	0.29	0.02
128	Дешифрування	0.0124608	0.24	0.02
128	Кодування	0.000771622	1.59	0.01
128	Декодування	0.00199029	1.93	0.02

Таблиця Д3. 10% 1 000 000 символів:

Кількість потоків	Операція	Час (с)	Прискорення	Ефективність
1	Шифрування	0.0135614	1	1
1	Дешифрування	0.0124608	1	1
1	Кодування	0.000771622	1	1
1	Декодування	0.00199029	1	1
2	Шифрування	0.00595679	2.27	1.13
2	Дешифрування	0.0014915	8.36	4.18
2	Кодування	0.00206659	0.37	0.19
2	Декодування	0.000423014	4.71	2.35
4	Шифрування	0.00285373	4.76	1.19
4	Дешифрування	0.00281532	4.42	1.11
4	Кодування	0.00069266	1.11	0.28
4	Декодування	0.000235831	8.46	2.12
8	Шифрування	0.00227181	5.97	0.75
8	Дешифрування	0.00165056	7.56	0.94
8	Кодування	0.00152902	0.50	0.25
8	Декодування	0.000319145	6.23	0.78
16	Шифрування	0.00215559	6.29	0.39
16	Дешифрування	0.00221272	5.64	0.35
16	Кодування	0.000710438	1.09	0.07
16	Декодування	0.000399274	5.00	0.31
32	Шифрування	0.00227181	5.97	0.19
32	Дешифрування	0.00219877	5.70	0.18
32	Кодування	0.000975712	0.79	0.05
32	Декодування	0.00089725	2.22	0.07
64	Шифрування	0.00326572	4.16	0.06
64	Дешифрування	0.00236672	5.26	0.08
64	Кодування	0.000924647	0.83	0.01
64	Декодування	0.00033421	5.95	0.09

128	Шифрування	0.0068816	1.97	0.02
128	Дешифрування	0.00455131	2.73	0.02
128	Кодування	0.000634694	1.22	0.01
128	Декодування	0.000258158	7.71	0.06

Таблиця Д4. 15% 10 000 символів:

Кількість потоків	Операція	Час (с)	Прискорення	Ефективність
1	Шифрування	0.000449995	1	1
1	Кодування	0.000103499	1	1
1	Декодування	1.2833e-05	1	1
1	Дешифрування	0.000307311	1	1
2	Шифрування	0.000297127	1.51	0.75
2	Кодування	3.0593e-05	3.38	1.69
2	Декодування	6.278e-06	2.04	1.02
2	Дешифрування	0.00018672	1.65	0.83
4	Шифрування	0.000236164	1.90	0.48
4	Кодування	2.6925e-05	3.85	0.96
4	Декодування	2.3074e-05	0.55	0.14
4	Дешифрування	0.000627104	0.49	0.12
8	Шифрування	0.000632901	0.71	0.09
8	Кодування	0.000142202	0.73	0.18
8	Декодування	5.4815e-05	0.24	0.03
8	Дешифрування	0.000185127	1.66	0.21
16	Шифрування	0.000731308	0.61	0.04
16	Кодування	0.000179517	0.58	0.04
16	Декодування	7.7518e-05	0.16	0.02
16	Дешифрування	0.000214368	1.43	0.09
32	Шифрування	0.00145526	0.31	0.02
32	Кодування	0.000378274	0.27	0.02
32	Декодування	5.287e-05	0.24	0.01
32	Дешифрування	0.000321812	0.95	0.03
64	Шифрування	0.00904209	0.05	0.01

64	Кодування	0.000100776	1.03	0.02
64	Декодування	0.000104036	0.13	0.02
64	Дешифрування	0.000882788	0.35	0.05
128	Шифрування	0.00709232	0.06	0.01
128	Кодування	0.000128332	0.80	0.01
128	Декодування	1.1444e-05	1.12	0.01
128	Дешифрування	0.00104479	0.29	0.03

Таблиця Д5. 15% 100 000 символів:

Кількість потоків	Операція	Час (с)	Прискорення	Ефективність
1	Шифрування	0.00342769	1	1
1	Кодування	0.00103214	1	1
1	Декодування	0.00110012	1	1
1	Дешифрування	0.0029352	1	1
2	Шифрування	0.00210141	1.63	0.81
2	Кодування	0.00181615	0.57	0.28
2	Декодування	0.000665809	1.65	0.82
2	Дешифрування	0.00192256	1.53	0.76
4	Шифрування	0.00218435	1.57	0.39
4	Кодування	0.000685513	1.51	0.38
4	Декодування	0.000489329	2.25	0.56
4	Дешифрування	0.00224772	1.31	0.33
8	Шифрування	0.00334299	1.02	0.13
8	Кодування	0.000609346	1.69	0.21
8	Декодування	0.000556995	1.98	0.25
8	Дешифрування	0.00585913	0.50	0.06
16	Шифрування	0.00218607	1.57	0.10
16	Кодування	0.000685383	1.51	0.10
16	Декодування	0.000373534	2.94	0.18
16	Дешифрування	0.00181602	1.61	0.10
32	Шифрування	0.00320325	1.07	0.06
32	Кодування	0.00071129	1.45	0.07

32	Декодування	0.000215128	5.12	0.16
32	Дешифрування	0.0015608	1.88	0.09
64	Шифрування	0.00556882	0.61	0.04
64	Кодування	0.00174745	0.59	0.03
64	Декодування	0.000336756	3.27	0.19
64	Дешифрування	0.00200108	1.47	0.09
128	Шифрування	0.01512	0.23	0.02
128	Кодування	0.00069753	1.48	0.01
128	Декодування	0.000485755	2.27	0.15
128	Дешифрування	0.00885892	0.33	0.03

Таблиця Д6. 15% 1 000 000 символів:

Кількість потоків	Операція	Час (с)	Прискорення	Ефективність
1	Шифрування	0.0340742	1	1
1	Кодування	0.0118261	1	1
1	Декодування	0.00508064	1	1
1	Дешифрування	0.0293371	1	1
2	Шифрування	0.0199349	1.71	0.86
2	Кодування	0.0150134	0.79	0.40
2	Декодування	0.002172	2.34	1.17
2	Дешифрування	0.0196179	1.50	0.75
4	Шифрування	0.0231967	1.47	0.37
4	Кодування	0.00446198	2.65	0.66
4	Декодування	0.00242718	2.09	0.52
4	Дешифрування	0.0115895	2.53	0.63
8	Шифрування	0.0233985	1.46	0.18
8	Кодування	0.00483796	2.32	0.29
8	Декодування	0.00217093	2.34	0.28
8	Дешифрування	0.0142466	2.06	0.26
16	Шифрування	0.0316934	1.07	0.13
16	Кодування	0.00825898	1.43	0.09
16	Декодування	0.00228655	2.22	0.16

16	Дешифрування	0.0162336	1.81	0.11
32	Шифрування	0.0185691	1.83	0.06
32	Кодування	0.00644498	1.83	0.05
32	Декодування	0.00275061	1.85	0.10
32	Дешифрування	0.0131525	2.24	0.14
64	Шифрування	0.0244071	1.40	0.05
64	Кодування	0.00596208	1.98	0.06
64	Декодування	0.0024685	2.05	0.11
64	Дешифрування	0.016417	1.79	0.14
128	Шифрування	0.0290781	1.17	0.04
128	Кодування	0.00854696	1.38	0.03
128	Декодування	0.00283005	1.79	0.13
128	Дешифрування	0.0226297	1.30	0.12

Таблиця Д7. 20% 10 000 символів:

Кількість потоків	Операція	Час (с)	Прискорення	Ефективність
1	Шифрування	0.000525273	1	1
1	Кодування	0.000090536	1	1
1	Декодування	0.000006852	1	1
1	Дешифрування	0.000488866	1	1
2	Шифрування	0.000290053	1.81	0.91
2	Кодування	0.000025925	3.49	1.75
2	Декодування	0.000006241	1.10	0.55
2	Дешифрування	0.000173906	2.81	1.41
4	Шифрування	0.000235831	2.23	0.56
4	Кодування	0.000026167	3.46	0.86
4	Декодування	0.000020371	0.34	0.09
4	Дешифрування	0.000562921	0.87	0.21
8	Шифрування	0.000676123	0.78	0.10
8	Кодування	0.000062907	1.44	0.18
8	Декодування	0.000014963	0.46	0.06
8	Дешифрування	0.000176961	2.76	0.35

16	Шифрування	0.00128743	0.41	0.03
16	Кодування	0.000095703	0.95	0.06
16	Декодування	0.000008074	0.85	0.05
16	Дешифрування	0.000190276	2.57	0.16
32	Шифрування	0.00138325	0.38	0.01
32	Кодування	0.000060407	1.50	0.05
32	Декодування	0.000007334	0.93	0.03
32	Дешифрування	0.000252887	1.93	0.06
64	Шифрування	0.00267588	0.20	0.03
64	Кодування	0.00009187	0.98	0.02
64	Декодування	0.000104647	0.07	0.01
64	Дешифрування	0.00182978	0.27	0.02
128	Шифрування	0.0112087	0.05	0.00
128	Кодування	0.000111462	0.81	0.01
128	Декодування	0.000133332	0.05	0.00
128	Дешифрування	0.00266788	0.18	0.01

Таблиця Д8. 20% 100 000 символів:

Кількість потоків	Операція	Час (с)	Прискорення	Ефективність
1	Шифрування	0.00331062	1	1
1	Кодування	0.00107716	1	1
1	Декодування	0.000539051	1	1
1	Дешифрування	0.00311558	1	1
2	Шифрування	0.00201281	1.64	0.82
2	Кодування	0.00068337	1.58	0.79
2	Декодування	0.000310497	1.73	0.86
2	Дешифрування	0.00155299	2.01	1.01
4	Шифрування	0.00175812	1.88	0.47
4	Кодування	0.00043265	2.49	0.62
4	Декодування	0.000257144	2.10	0.53
4	Дешифрування	0.00077465	4.02	1.00
8	Шифрування	0.00112435	2.95	0.37

8	Кодування	0.00034116	3.16	0.40
8	Декодування	0.00021096	2.55	0.32
8	Дешифрування	0.00052328	5.96	0.75
16	Шифрування	0.00080262	4.12	0.26
16	Кодування	0.00021068	5.11	0.32
16	Декодування	0.00013182	4.09	0.26
16	Дешифрування	0.00041388	7.53	0.47
32	Шифрування	0.00075629	4.38	0.14
32	Кодування	0.00019264	5.59	0.17
32	Декодування	0.00008992	6.00	0.19
32	Дешифрування	0.00030044	10.37	0.32
64	Шифрування	0.00143299	2.31	0.04
64	Кодування	0.00024418	4.41	0.07
64	Декодування	0.00011217	4.81	0.08
64	Дешифрування	0.00050392	6.18	0.10
128	Шифрування	0.00201376	1.64	0.01
128	Кодування	0.00029047	3.71	0.03
128	Декодування	0.00015091	3.57	0.02
128	Дешифрування	0.00071204	4.37	0.03

Таблиця Д9. 20% 1 000 000 символів:

Кількість потоків	Операція	Час (с)	Прискорення	Ефективність
1	Шифрування	0.0321017	1	1
1	Кодування	0.0101183	1	1
1	Декодування	0.00590223	1	1
1	Дешифрування	0.0315051	1	1
2	Шифрування	0.0189621	1.69	0.84
2	Кодування	0.00623211	1.62	0.81
2	Декодування	0.00345912	1.71	0.86
2	Дешифрування	0.0172149	1.83	0.91
4	Шифрування	0.0167413	1.92	0.48
4	Кодування	0.0048231	2.10	0.52

4	Декодування	0.0029283	2.02	0.51
4	Дешифрування	0.0128124	2.46	0.62
8	Шифрування	0.0131125	2.45	0.31
8	Кодування	0.0034327	2.95	0.37
8	Декодування	0.0023124	2.55	0.32
8	Дешифрування	0.0098564	3.20	0.40
16	Шифрування	0.0093487	3.43	0.21
16	Кодування	0.0024123	4.19	0.26
16	Декодування	0.0018121	3.25	0.20
16	Дешифрування	0.0074593	4.22	0.26
32	Шифрування	0.0071593	4.48	0.14
32	Кодування	0.0019234	5.26	0.16
32	Декодування	0.0015233	3.87	0.12
32	Дешифрування	0.0058121	5.42	0.17
64	Шифрування	0.0058321	5.50	0.09
64	Кодування	0.0017321	5.84	0.09
64	Декодування	0.0012123	4.87	0.08
64	Дешифрування	0.0045127	6.98	0.11
128	Шифрування	0.0049123	6.53	0.05
128	Кодування	0.0016123	6.28	0.05
128	Декодування	0.0010034	5.88	0.05
128	Дешифрування	0.0039821	7.91	0.06