КПІ ім. Ігоря Сікорського
Інститут прикладного системного аналізу
Кафедра Системного проектування

Курсова Робота
з дисципліни
Паралельні обчислення

Виконав:
Студент групи ДА-01
ННК «ІПСА»
Зарицький Кирило Андрійович
Варіант № 9

Київ — 2023

Зміст

Вступ

Теоретичні Відомості

2.1 Інвертований індекс

Інвестований індекс – індекс бази даних, що зберігає асоціації з вмісту, як наприклад слова та числа, в їх розташування, таких як таблиці та документи (на відміну від прямого індексу, що ассоціює з документів в їх вміст). Основне призначення інвестованого індексу – швидкий текстовий пошук за рахунок повільного розширення індексу. Цей тип індексу активно використовується в пошукових системах. Існує два основні варіанти індексу: інвертований індекс рівня запису, що містить набір посилань на документи для кожного слова, та інвертований індекс рівня вмісту, де додатково додається позиція слова в документі.

2.2 Пул потоків

Модель Системи

3.1 Загальна Модель Системи

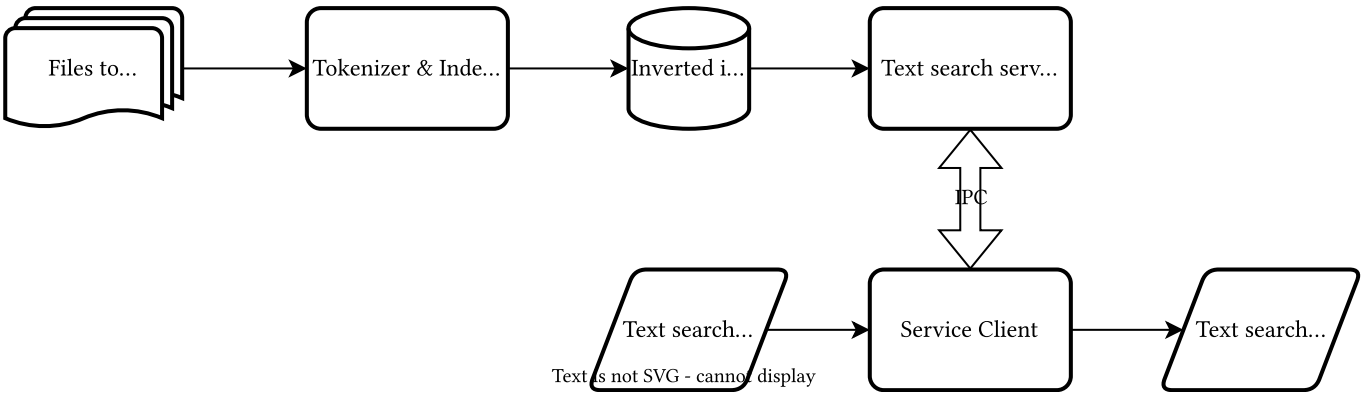

Figure 1: Загальна модель системи

На рисунку Figure 1 зображено загальну модель системи. Маємо файли, що токенізуються і індексуються. Результат зберігається в інвертованому індексі та надходить до системи надання сервісу, що через IPC надає сервіс клієнту перетворюючи текстовий пошуковий запит у відповідний список документів.

3.2 Діаграма Діяльності

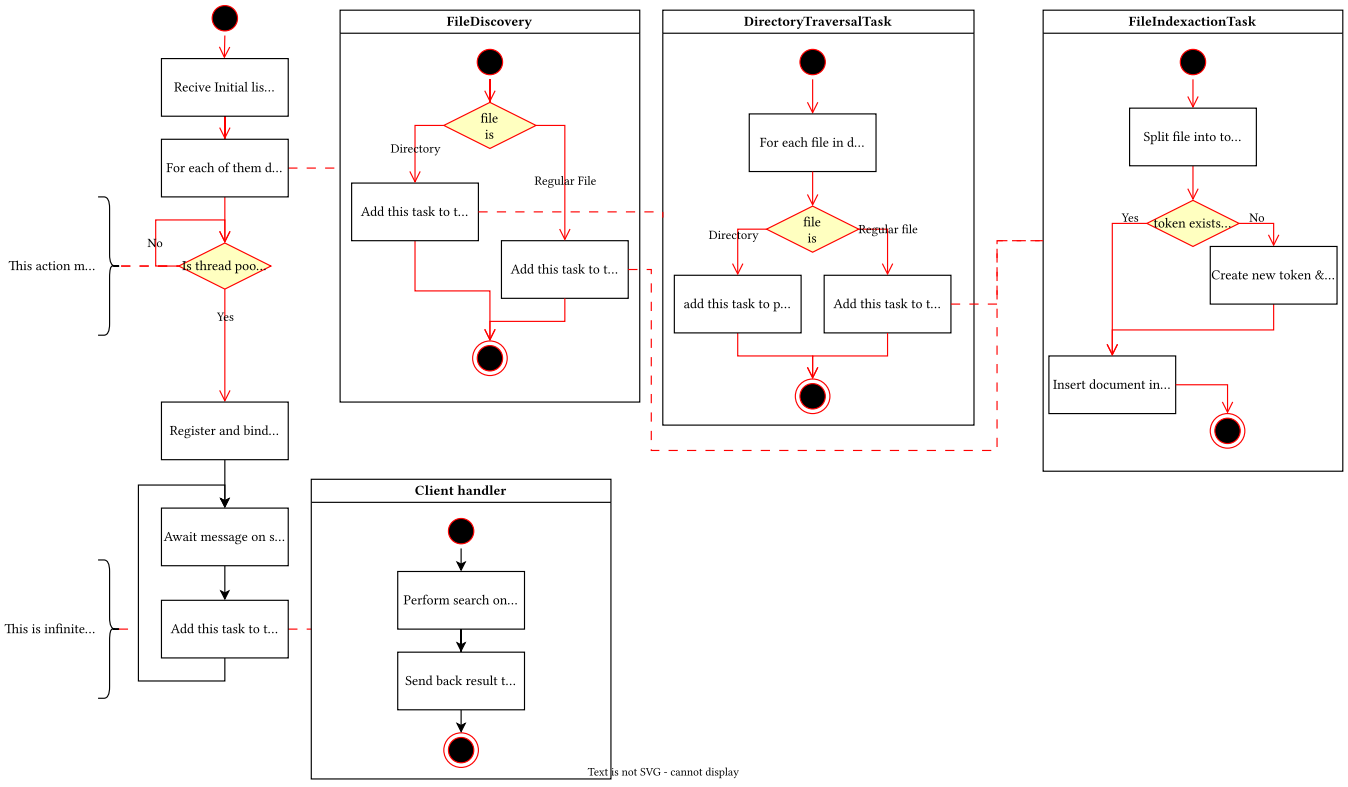

Figure 2: Діаграма Діяльності

На рисунку Figure 2 зображено діаграму діяльності моделі системи. Паралельна побудова індексу реалізована за рахунок паралельної обробки файлів.

Так як попередньо розподіл файлів в директоріях невідомий, було обрано метод асинхроного виконання функцій, зображений у формі процедур DirectoryTraversalTask та FileIndexationTask.

Такий підхід дозволяє розподіляти виконання файлів під час рекурсивного обходу наданих файлів. Недоліком підходу є відсутність розпаралелення при обробці самих файлів (процедура FileIndexationTask). Це може спричинити неоптимальне навантаження системи, коли серед файлів є велика дисперсія їх розмірів. Відсутність розпаралелення обробки директорій також може породжувати неоптимальний розподіл навантаження при достатньо великій кількості файлів в одній директорії.

Єдиним місцем синхронізації отриманих процедур буде додавання пар “слово-документ” до індексу. Відповідно слід мінімізувати кількість роздільних редагувань індексу. Пропонується виконати це за рахунок додавань групами. Після завершення побудови індексу його синхронізація не потрібна, так-як він буде лише читатись.

Паралельної обробки клієнтів виконується за рахунок асинхроного виклику процедури ClientHandler. Це просте рішення дозволить обробляти клієнтів паралельно. Воно є оптимальним для великої кількості клієнтів, але є неоптимальним, якщо клієнтів мало (у випадку пулу потоків, менша ніж кількість потоків).

Асинхроне виконання процедур пропонується виконати за рахунок пулу потоків.

3.3 Протокол Міжпроцесної Взаємодії

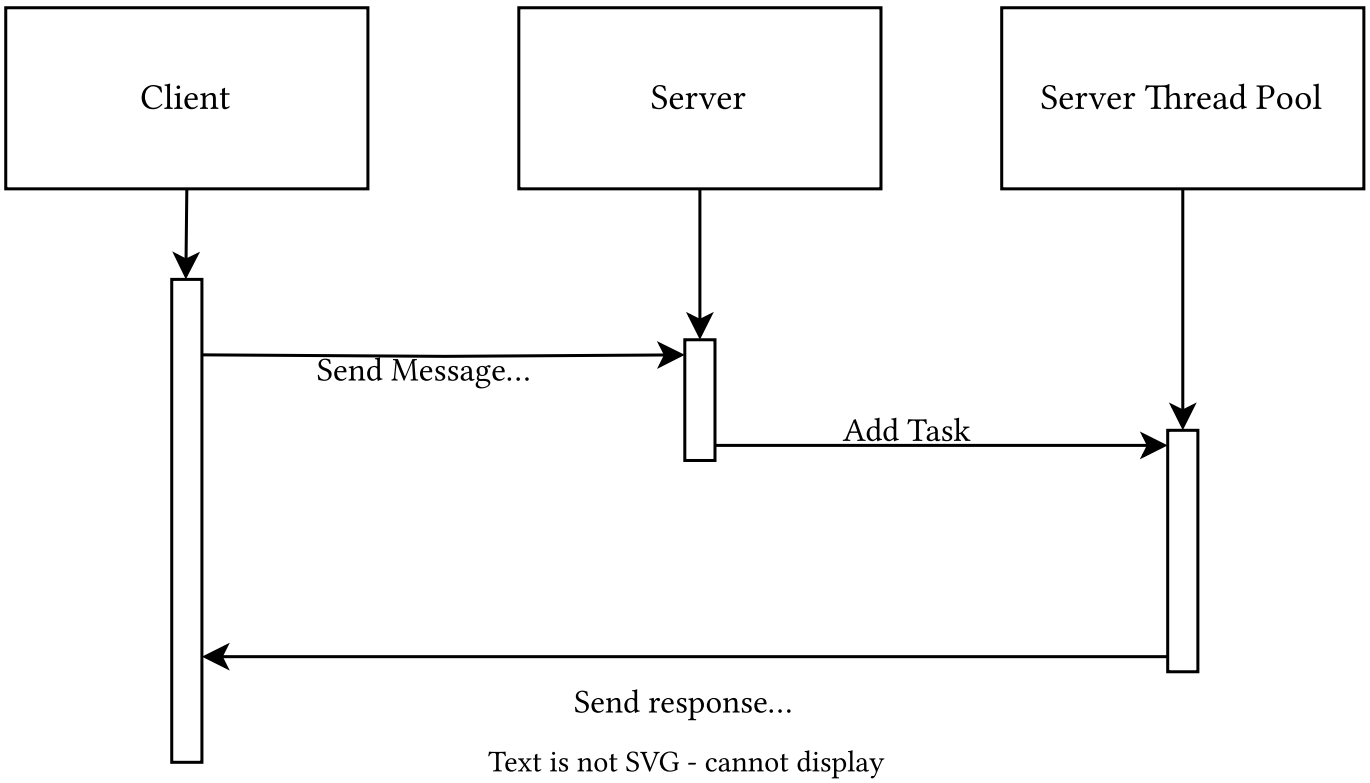

Figure 3: Протокол Міжпроцесної Взаємодії

На рисунку Figure 3 зображено міжпроцесну взаємодію. Клієнт надсилає повідомлення, вміст якого – текст пошукового запиту. Сервер отримує повідомлення, створює процедуру в пулі потоків, що оброблює запит та надсилає результат клієнту. Пакування пропонується зробити простим записом C подібної строки в тіло пакету. Для комунікації можна використати, що TCP, що UDP. У випадку TCP клієнт спочатку підключиться до сервера, відповідно сервер вже починаючи з цього моменту може відділити обробку в підпроцес. В розробленій реалізації якраз і використовується протокол TCP.

Опис Реалізації

В цьому розділі буде розглянуто реалізацію моделі системи. Далі буде розглянуто структуру проекту, опис його модулів та синхронізації в цих модулях. Також в кінці розділу є інструкція з інсталювання.

Загалом проект був розроблений на мові програмування C++ для Unix подібних систем (в першу чергу Linux).

4.1 Структура Проекту

Побудова, інсталювання та тестування проекту виконано за допомогою системи CMake. Її конфігураційні файли розташовані в кореневій директорії проекту та директорії tests.

В директоріях include та src розташовані файли в яких описані класи проекту (в “include/*.h” – опис класів, а в “src/*. cpp” – реалізації). Така місткова система є стандартною практикою в написанні C++ коду.

В директорії exec розташованні функції main відповідних виконавчих файлів. В проекті збирається 4 програми:

1. CWServer – програма, що збирає індекс та потім відкриває інтернет сокет для надання сервісу.
2. CWClient – програма-клієнт, що отримує сервіс.
3. CWMultiClient – програма, що робить стрес-експеримент запускаючи N потоків, що звертаються до сервісу одночасно.
4. tests/CommonCxxTests – тести системи. В проекті є 5 модульних тестів, що можуть бути виконані в цій програмі (зауважимо, що тести не виводять дані, тому для перевірки корректності їх виконання слід дивитись на код, що вони повертають (0 = успіх)):
    1. invertedindex_write_multithread_test – тест на багатопоточний запис в індекс.

2. invertedindex_write_singlethread_test – тест на однопоточний запис в індекс.
3. threadpool_completion_test – тест на завершення виконання всіх завдань в пулі потоків.
4. threadpool_subscriber_test – тест на перевірку підписки на завершення роботи пулу потоків.
5. threadpool_thread_count_test – тест на запуск правильної кількості робочих потоків.

В директоріях test_data та eval_data розташовані дані для побудови індексу. Перший – невеликий набір даних для наочної перевірки роботи програми. Другий – набір даних за варіантом курсової роботи. Зауважимо, що користувач може сам надавати програмі дані.

В директорії doc розташована документація проекту. В директорії models – діаграми моделі системи.

4.2 Опис Модулів

Прогама має в собі 2 класи, 2 незалежні функції та головну функцію.

4.2.1 InvertedIndex

defined in <invertedindex.h>

```
class InvertedIndex;
```

Реалізація інвертованого індексу. В середині він з себе представляє індивідуально реалізовану hash невпорядкований набір (hast-таблиця) з пар слово – набір назв документів в яких воно з'являється(завдяки реалізації набір документів відсортований).

4.2.1.1 (constructor)

```
InvertedIndex(const std::size_t initialSize = defaultInitialSize, const float loadFactor = defaultLoadFactor);
```

Будує клас з заданим початковим розміром та заданим loadFactor.

- **initialSize** – початковий розмір масиву backets hash-таблиці.
- **loadFactor** – loadFactor hash-таблиці. Якщо цей вираз вірний:

$$\frac{\text{elements in array}}{\text{array size}} > \text{loadFactor}$$

то розмір масиву збільшується в двічі.

4.2.1.2 insert

```
void insert(const std::string token, const std::string document);
```

Додає пару **token**-docu*ment до індексу. Цей метод блокуючий! Виконання цієї операції неможливе після виклику **finish**.

- **token** – слово, з яким асоційовується документ.
- **document** – назва документу, що буде асоційований зі словом.

4.2.1.3 insertBatch

```
void insertBatch(const std::vector<std::pair<std::string,std::string>>&
pairs);
```

Додає серію пар **token-document** до індексу. Цей метод блокуючий! Виконання цієї операції неможливе після виклику **finish**.

- **pairs** – масив пар слово-документ, що будуть додані до масиву.

4.2.1.4 find

```
bool find(const std::string token);
```

Шукає слово **token** в масиві. Виконання цієї операції неможливе перед викликом **finish**.

- **token** – слово, що треба знайти.
- **Повертає** – *true*, якщо слово є в індексі та *false* в іншому випадку.

4.2.1.5 read

```
const std::set<std::string>& read(const std::string token);
```

Зчитує документи в яких зустрічається слово **token**. Виконання цієї операції неможливе перед викликом **finish**.

- **token** – слово, що треба знайти.
- **Повертає** – набір документів, в яких слово присутнє в індексі; кидає *std::exception* в іншому випадку.

4.2.1.6 finish

```
void finish();
```

Завершує редагування класу та відкриває його для читання.

4.2.2 ThreadPool

defined in <threadpool.h>

```
class ThreadPool;
```

Реалізація пулу потоків. В середині використовує пріоритетну чергу для завдань.

4.2.2.1 Task

```
typedef std::pair<int, std::function<const int()>> Task;
```

Тип "завдання" є важливим для додавання завдань в чергу. Представляє з себе пару з цілого числа та функції, що бере 0 аргументів та повертає ціле число.

4.2.2.2 (constructor)

```
 ThreadPool(unsigned int N, bool exitImmediatlyOnTerminate = false);
ThreadPool(const ThreadPool&) = delete;
ThreadPool(ThreadPool&& other) = delete;
ThreadPool& operator=(ThreadPool& rhs) = delete;
ThreadPool& operator=(ThreadPool&& rhs) = delete;
```

Будує клас з заданою кількістю потоків та стандартною поведінкою деструктора. Зауважимо, що copy та move семантика для класу заборонені (видалені).

- **N** – кількість потоків пулу потоків.

- **exitImmediatlyOnTerminate** – поведінка виходу. В залежності від обраного значення буде змінена поведінка деструктора.

4.2.2.3 (destructor)

```
~ThreadPool();
```

Зупиняє роботу класу. В залежності від значення флагу **exitImmediatlyOnTerminate**, заданому при створенні, буде викликано або terminateIm() для *true*, або terminate навпаки.

4.2.2.4 terminate/terminateIm

```
void terminate();
void terminateIm();
```

Зупиняє роботу класу. Всі потоки або відокремлюються при **terminateIm** або приєднуються до потоку виклику при **terminate**.

4.2.2.5 pause/unpause/pauseToggle

```
void pause();
void unpause();
void pauseToggle();
```

Призупиняє виконання наступних завдань. Ці функції не не призупиняють самі потоки, а лише не дають їм взяти нові завдання з черги.

4.2.2.6 addTask

```
void addTask(Task task);
```

Додає завдання в чергу.

- **task** – завдання, що буде додано.

4.2.2.7 removeTask

```
void removeTask();
```

Видаляє останнє завдання з черги.

4.2.2.8 currentQueueSize

```
unsigned int currentQueueSize();
```

Повертає поточну довжину черги.

4.2.2.9 currentThreadStatus

```
std::unordered_map<unsigned short, ThreadPool::threadStatusEnum>
currentThreadStatus();
```

Повертає поточний статус всіх потоків.

4.2.2.10 toString

```
static const char* toString(ThreadPool::threadStatusEnum v)
```

Повертає строку, що пояснює стан потоку. **v** – статус потоку, який треба пояснити.

4.2.2.11 numberOfThreds

```
unsigned int numberOfThreds()
```

Повертає кількість потоків в класі.

4.2.2.12 avgWaitTime

```
double avgWaitTime();
```

Повертає середній час очікування на завдання.

4.2.2.13 avgWaitTimeReset

```
void avgWaitTimeReset();
```

Скидує дані про середній час очікування.

4.2.2.14 avgQueueSize

```
double avgQueueSize();
```

Повертає середню довжину черги.

4.2.2.15 avgQueueSizeReset

```
void avgQueueSizeReset();
```

Скидує дані про середню довжину черги.

4.2.2.16 avgTaskCompletionTime

```
double avgTaskCompletionTime();
```

Повертає середній час виконання завдання.

4.2.2.17 avgTaskCompletionTimeReset

```
void avgTaskCompletionTimeReset();
```

Скидає дані про середній час виконання.

4.2.2.18 subscribeOnFinish

```
std::pair<std::set<std::function<void ()>>::iterator, bool>
subscribeOnFinish (std::function<void()> callback);
```

Підписується на подію завершення виконання. Ця подія відбувається коли всі потоки очікують елементу черги та черга пуста.

- **callback** – функція, що буде викликана за настання події.
- **Повертає** – пару з іттератору, що вказує на додану функцію та флагу, що показує чи був елемент доданий до списку підписників.

4.2.2.19 unsubscribeOnFinish

```
void unsubscribeOnFinish(std::set<std::function<void ()>>::iterator
itterator);
```

Відписатись від події завершення виконання.

- **itterator** – іттератор, що вказує на елемент, що був доданий.

4.2.3 HandleRegularFile

defined in “exec/CWServer.cpp”

```
const int HandleRegularFile(const std::filesystem::path filePath,
InvertedIndex& invIn);
```

Процедура, що переглядає всі слова в файлі та будує їх набір. З цього набору формуються пари слово-документ, що потім всі разом додаються до індексу

- **filePath** – шлях до файлу.
- **invertedIndex** – індекс, до якого слід додати слова з файлу.

4.2.4 HandleFile

defined in “exec/CWServer.cpp”

```
const int HandleFile(const std::filesystem::path filePath, ThreadPool&
threadPool, InvertedIndex& invIn);
```

Процедура, що оброблює файли. Якщо він директорія, то процедура переглядає його вміст додає рекурсивні виклики над цими файлами до пулу потоків. Якщо він звичайний файл, то додає процедуру HandleRegularFile над цим файлом до пулу потоків.

- **filePath** – шлях до файлу.
- **threadPool** – пул потоків, до якого слід додавати процедури.
- **invertedIndex** – індекс, що передається функції HandleRegularFile.

4.3 Опис синхронізації

Синхронізація системи має бути присутня лише в двох місцях: реалізації пулу потоків та реалізації запису в індекс.

Реалізація синхронізації пулу потоків є вирішення задачі розробників-споживачів. Виклики методів **addTask** тощо є розробником а потоки – споживачами. Ця задача вирішена використанням м’ютекса, що блокує будь-які зміни в спільних даних (в першу чергу – черги завдань) та умовної змінної.

Паралельний запис в індекс відсутній. Ця операція є виключно синхронною, реалізованою за рахунок єдиного мютекса на запис. Таким чином щоб мінімізувати сповільнення від викликів м’ютексу слід виконувати роздільні записи як можна рідше. Для цього був зроблений метод **insertBatch**, що за одну синхронізацію додає цілий масив елементів.

Також індекс має недопускати виконання запису та зчитування одночасно. Так як цього ніколи не відбувається в системі, це виконано методом finish, що відкриває індекс для читання та закриває для запису.

4.4 Керівництво з Встановлення Програми

Перед встановленням програми на комп'ютері має бути встановлено команди cmake та make. Програма підтримується **лише для Unix подібних систем**.

1. Скачайте релізний код з репозиторію та розархівуйте його.
2. В кореневій директорії отриманого коду створіть директорію build та перейдіть в неї.
3. З директорії build виконайте команду "`cmake ..`"
4. З директорії build виконайте команду "`make`"

Якщо обидві команди завершились без помилок з кодом 0, програму зібрано правильно і її можна знайти в директорії build (CWServer) разом з клієнтом (CWClient) та багатопоточним експериментом (CWMultiClient). Тести можна запустити з директорії build/tests/CommonCxxTests.

Виконання програми

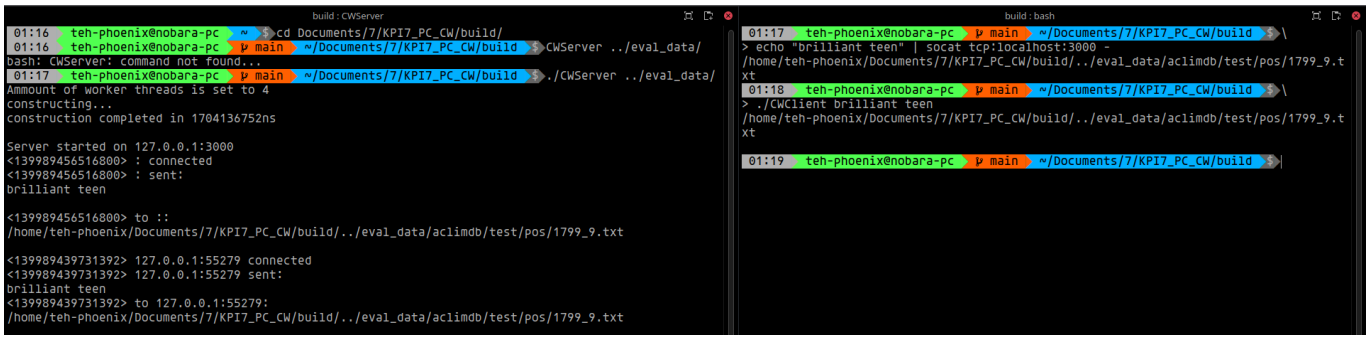

Figure 4: Виконання програми. Сервер викликається з програми socat та з програми CWClient

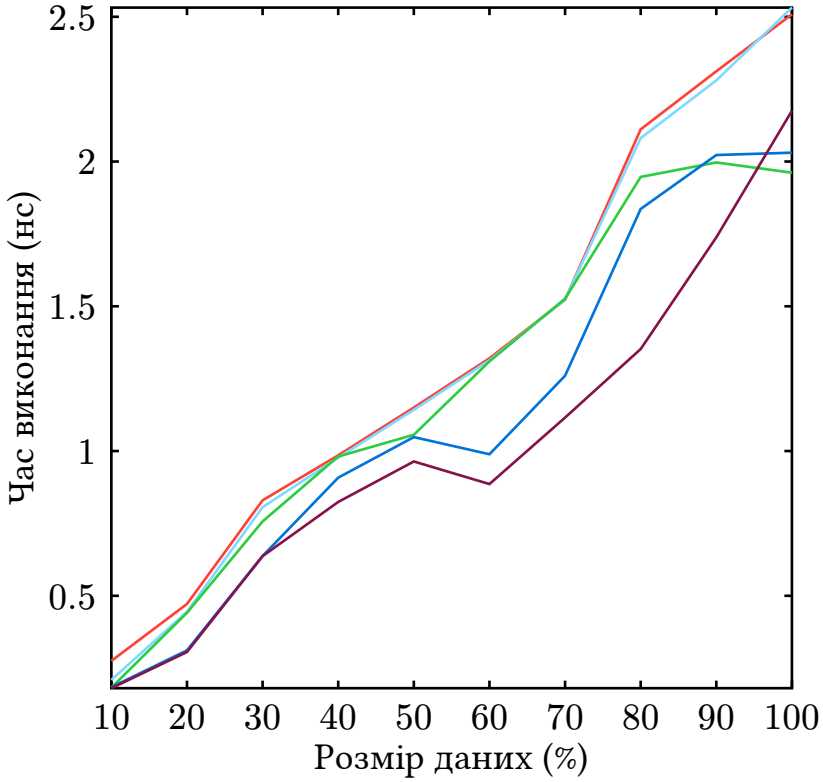

Figure 5: Час побудови в залежності від розміру даних для різної кількості потоків (red – 1, aqua – 2, green – 4, blue – 8, maroon – 16)

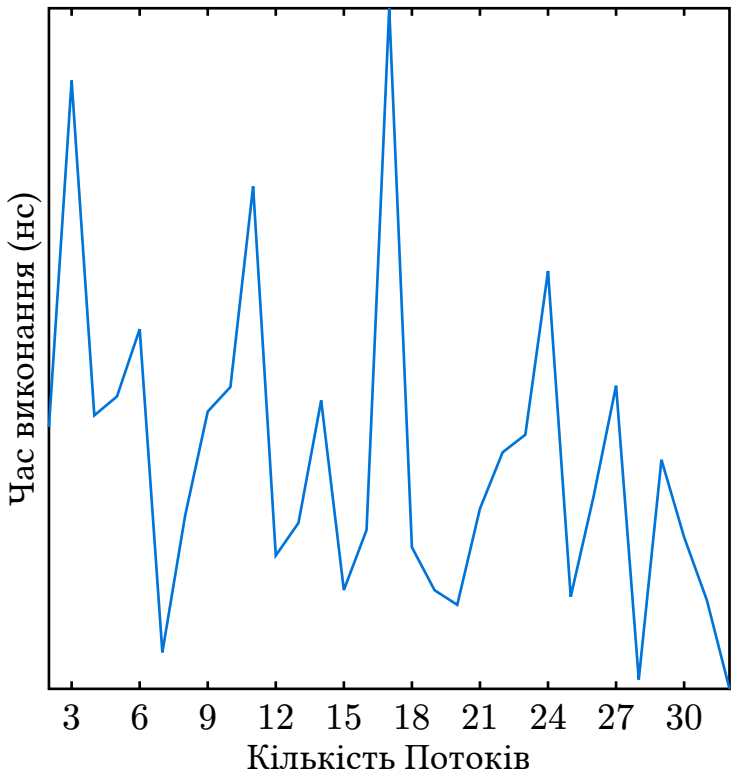

Figure 6: Час побудови в залежності від кількості потоків

Висновки