Politechnika Wrocławska

Wydział Informatyki i Telekomunikacji

Obliczenia Wysokiej Wydajności

Środa, 13:15-15:00

Łamanie zahaszowanych haseł metodą słownikową

Autorzy:

Aleksandra Rozmus, 252954

Paweł Struczewski, 252950

Prowadzący:

dr inż. Radosław Idzikowski

9 stycznia 2024

Politechnika
Wrocławska

Spis treści

1 Temat projektu **2**

2 Skład zespołu **2**

3 Zakres projektu **2**

4 Wyliczenia teoretycznego przyśpieszenia **3**

5 Implementacja **4**

5.1 Metoda sekwencyjna 5

5.2 Metoda równoległa 5

6 Badania **7**

6.1 Pierwszy zbiór danych 7

6.2 Drugi zbiór danych 8

7 Wnioski **10**

1 Temat projektu

Tematem projektu jest stworzenie wielowątkowego współbieżnego programu do łamania zahaszowanych haseł metodą słownikową z wykorzystaniem algorytmu MD5.

2 Skład zespołu

Zespół składa się z dwóch osób:

- Aleksandra Rozmus - 252954
- Paweł Struczewski - 252950

3 Zakres projektu

Podczas tego projektu stworzony zostanie program polegający na łamaniu zahaszowanych haseł. Hasła są generowane metodą skrótów kryptograficznych MD5. Program zawiera listy zaszyfrowanych haseł do łamania oraz słownik, który zawierać będzie dużą liczbę słów różnej długości. Lista haseł będzie typowo mniejsza (maksymalnie 1000 jednowyrazowych haseł) i będą one miały stałą długość 32 znaków.

Zaimplementowani producenci haseł będą tworzyli hasła w następującym formacie:

- wyłącznie małe litery
- pierwsza wielka litera i reszta małe litery
- wyłącznie wielkie litery

Do każdego hasła przed zahaszowaniem dodana może zostać cyfra na początku wyrazu, na końcu lub w obu wymienionych miejscach.

Każde kolejno znalezione złamane hasło powinno zostać wyświetlone na ekranie.

Projekt podzielony został na 3 etapy:

- Etap 1 - Implementacja metody sekwencyjnej
- Etap 2 - Implementacja metody równoległej
- Etap 3 - Badania - porównanie metody sekwencyjnej i równoległej

W ramach implementacji metody równoległej stworzony zostanie wątek główny, którego zadaniem jest wczytanie do pamięci globalnej programu podanej przez użytkownika listy zaszyfrowanych haseł do łamania, oraz podanego (lub domyślnego) słownika. Następnie wątek główny powinien uruchomić zestaw wątków "producentów". Każdy z tych producentów ma za zadanie próbować złamać wszystkie hasła, stosując różne metody budowy haseł, aby proces łamania haseł mógł działać równolegle. Każdy producent będzie kolejno tworzyć potencjalne hasła przy użyciu swojej unikalnej metody, a następnie obliczać 32-znakowy skrót kryptograficzny MD5 dla każdego z tych haseł. Następnie porówna ten skrót z całym zestawem zaszyfrowanych haseł. Jeśli którykolwiek z tych zaszyfrowanych haseł zgadza się z obliczonym skrótem, to oznacza, że odnaleziono oryginalne hasło. To hasło zostanie przekazane do wątku konsumenta, który je zarejestruje i oznaczy odpowiednie zaszyfrowane hasło w globalnej tablicy, aby uniknąć dalszych prób jego porównywania. Wątek konsumenta powinien wyświetlać na ekranie każde kolejno znalezione złamane hasło w miarę otrzymywania ich od producentów.

Ilość działających wątków będzie dostosowywana do ilości rdzeni maszyny, na której uruchomiony jest program.

4 Wyliczenia teoretycznego przyśpieszenia

Teoretyczne przyśpieszenie programu wielowątkowego w zadaniu łamania zahaszowanych haseł za pomocą metody słownikowej może być wyliczone przy użyciu tzw. prawa Amdahla.

Wzór wykorzystany do wykonania obliczeń:

$$Pr = \frac{1}{(1-P) + \frac{P}{N}}$$

gdzie:

P - Proporcja programu, która może podlegać zrównolegleniu,
N - liczba procesorów.

W naszym przypadku część niepodlegająca zrównolegleniu to około 20%. Jest to część odpowiedzialna za np. operacje wczytywania i porównywania. Więc część programu podlegająca zrównolegleniu to około 80%. Przykładowe obliczenia dla komputera z dostępnymi 10 rdzeniami:

$$Pr = \frac{1}{(1-0,8) + \frac{0,8}{10}} = 3,57$$

Przyśpieszenie równoległe wyniesie około 3,57 dla komputera z dostępnymi 10 rdzeniami.

5 Implementacja

Program zaimplementowano przy pomocy języka C. Program korzysta z biblioteki OpenSSL do generowania funkcji skrótu MD5 Wykorzystany plik ze słownikiem słów został pobrany z internetu. Jego fragment zaprezentowano na listingu 1.

```
...
Aaronic
Aaronical
Aaronite
Aaronitic
Aaru
Ababa
Ababdeh
Ababua
Abadite
Abama
Abanic
Abantes
Abarambo
Abaris
Abasgi
...
```

Listing 1: Fragment słownika z pliku *dictionary.txt*

Na podstawie wspomnianego słownika przy pomocy kodu napisanego w języku python stworzono listę haseł zgodnie z założeniami opisanymi w rozdziale 3.

Następnie przystąpiono do implementacji metody sekwencyjnej. Utworzono kod odpowiedzialny za wczytanie zahaszowanych haseł z pliku i zapisywanie ich w dwuwymiarowej tablicy oraz wczytano słownik z pliku *dictionary.txt*, tworząc dynamiczną tablicę słów. Następnie zaimplementowano algorytm, zwany producentem, do generowania różnych kombinacji słów z uwzględnieniem wielkości liter oraz dodając cyfry. Porównywano zahaszowane hasła z otrzymanymi skrótami MD5, identyfikując złamane hasła.

W przypadku zgodności z zahaszowanym hasłem, program uznaje je za złamane, zapisując wyniki do pliku wraz z czasem potrzebnym na złamanie hasła. W pliku z hasłami hasło to zostaje także zastapione znakiem #, aby nie powielać prób jego złamania. Dodatkowo, program obsługuje sygnał SIGHUP, co pozwala na monitorowanie ilości złamanych haseł w trakcie działania programu.

Na końcu programu następuje zwalnianie pamięci zaalokowanej dynamicznie dla słownika i innych struktur danych.

5.1 Metoda sekwencyjna

Metoda sekwencyjna została zaimplementowana w taki sposób, że generowała wszystkie możliwe warianty zakładanych haseł dla danego słowa ze słownika i sprawdzała po kolei, czy hasło to znajduje się w tablicy przechowującej hasła do złamania. Proces ten był powtarzany dla każdego słowa ze słownika.

5.2 Metoda równoległa

Metoda równoległa została zaimplementowana analogicznie do metody sekwencyjnej, z tą różnica, że wykorzystano tutaj bibliotekę *pthread.h*. Użyto struktury danych i zmienne, takie jak mutexy, zmienne warunkowe, i globalne zmienne do synchronizacji i komunikacji między wątkami. Na początku inicjalizowana jest liczba wątków w zależności od komputera, na którym uruchomiony jest program.

Stworzono funkcję producenta (prod), która była wykorzystywana przez działające wątki i służyła do generowania możliwych wariantów haseł. Znaczące w metodzie równoległej było to, że funkcja producenta była uruchamiana z odpowiednim argumentem przez dany wątek, oznaczający wariant łamanego hasła (małe litery, wielkie litery itd.).

Wykorzystano do tego stworzoną strukturę danych (thread_data), która reprezentowała dane przekazywane do wątków producentów (prod). Zawierała ona trzy pola:

- repetitions: ilość wątków dla danego producenta
- type: Typ kombinacji liter i cyfr przypisanej do danego wątku (1 - tylko wielkie litery, 2 - tylko małe litery, 3 - wielkie i małe litery).
- thread_number: Numer identyfikacyjny wątku producenta. Wątki dzieliły pracę poprzez przypisanie im różnych kombinacji liter i cyfr do sprawdzenia.

```
typedef struct {
   int repetitions;
   int type;
   int thread_number;
} thread_data;
```

Listing 2: Struktura przekazywana do wątków

Stworzono także wątek konsumenta (kon). który oczekiwał na sygnał od producenta, a następnie wyświetlał złamane hasło i zapisywał czas wykonania.

```
(...)
if((strcmp(t1,md5_1))==0 && passwords[i][0]!='#')
{
        pthread_mutex_lock(&lock);
        decodedpassword=malloc(strlen(letters)*sizeof(char));
        strcpy(decodedpassword, letters);
        counter++;
        taskid=type;
        task_number=thread_number;
        passwords[i][0]='#';
        pthread_cond_signal(&condvar);
        pthread_mutex_unlock(&lock);
        break;
}
else if ((strcmp(t1,md5_2))==0 && passwords[i][0]!='#'){
        pthread_mutex_lock(&lock);
        decodedpassword=malloc(strlen(letters_number)*sizeof(char));
        strcpy(decodedpassword, letters_number);
        counter++;
        taskid=type;
        task_number=thread_number;
        passwords[i][0]='#';
        pthread_cond_signal(&condvar);
        pthread_mutex_unlock(&lock);
        break;
}
(...)
```

Listing 3: Fragment funkcji producenta sprawdzający czy złamano hasło

```
thread_data threadData[NUM_THREADS];

(...)

for (int i = 0; i < NUM_THREADS; i++) {
        if (i == 0) {
                pthread_create(&threads[i], &attr, kon, (void*)t1);
        }
        else {
        pthread_create(&threads[i], &attr, prod, &threadData[i-1]);
        }
}
```

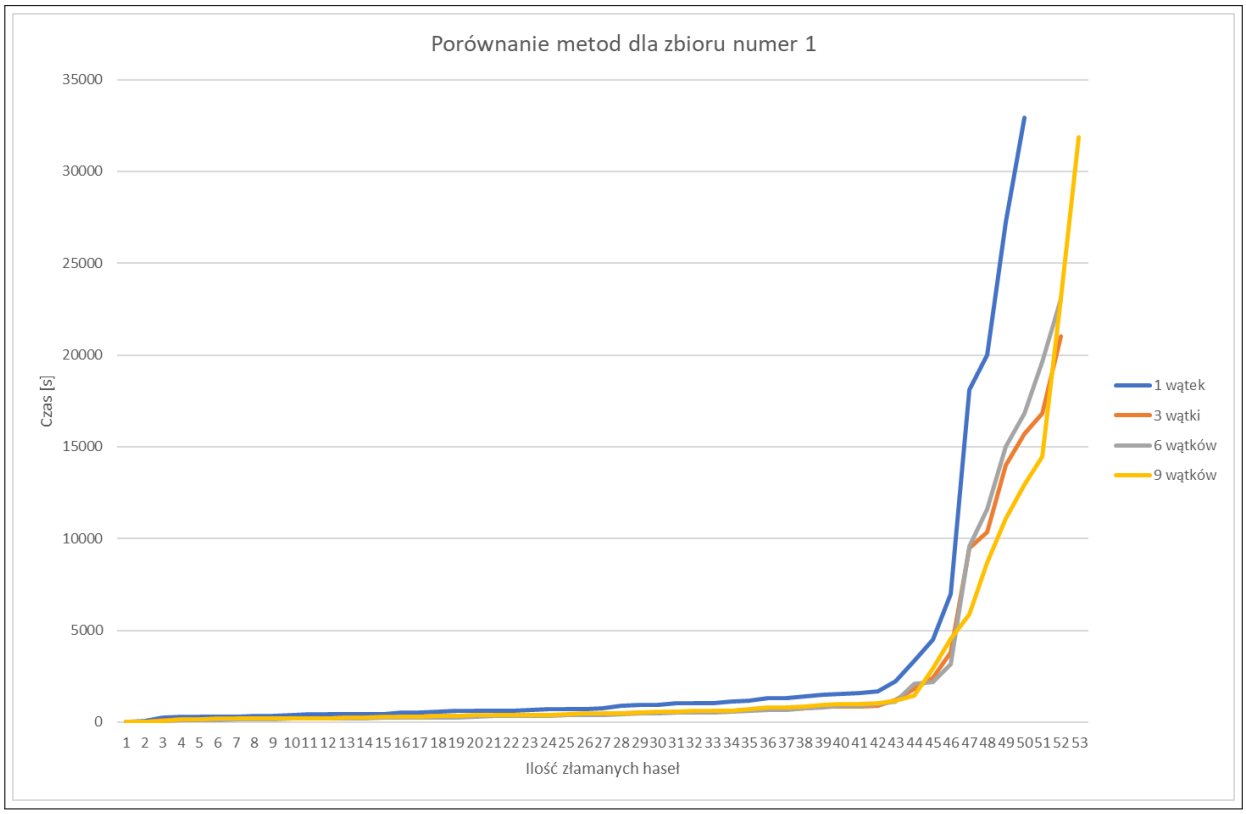
Listing 4: Uruchamianie wątków

6 Badania

W celu przeprowadzenia badań na podstawie wspomnianego słownika przy pomocy kodu napisanego w języku python stworzono listę 300 i 1000 haseł. Pierwszy plik zawierał 300 haseł, w których ilość cyfr była nieograniczona, natomiast plik z 1000 haseł zawierał hasła, w których występowały maksymalnie jednocyfrowe liczby nie większe niż 5. W celu sprawdzenia przyspieszenia programu w zależności od ilości wątków, zmieniano ilość działających wątków producenta, kolejno dla 1, 3, 6 oraz 9 wątków producenta.

6.1 Pierwszy zbiór danych

Pierwszy zbiór danych zawierał 300 zahaszowanych haseł zgodnie z założeniami opisanymi w rozdziale 3.



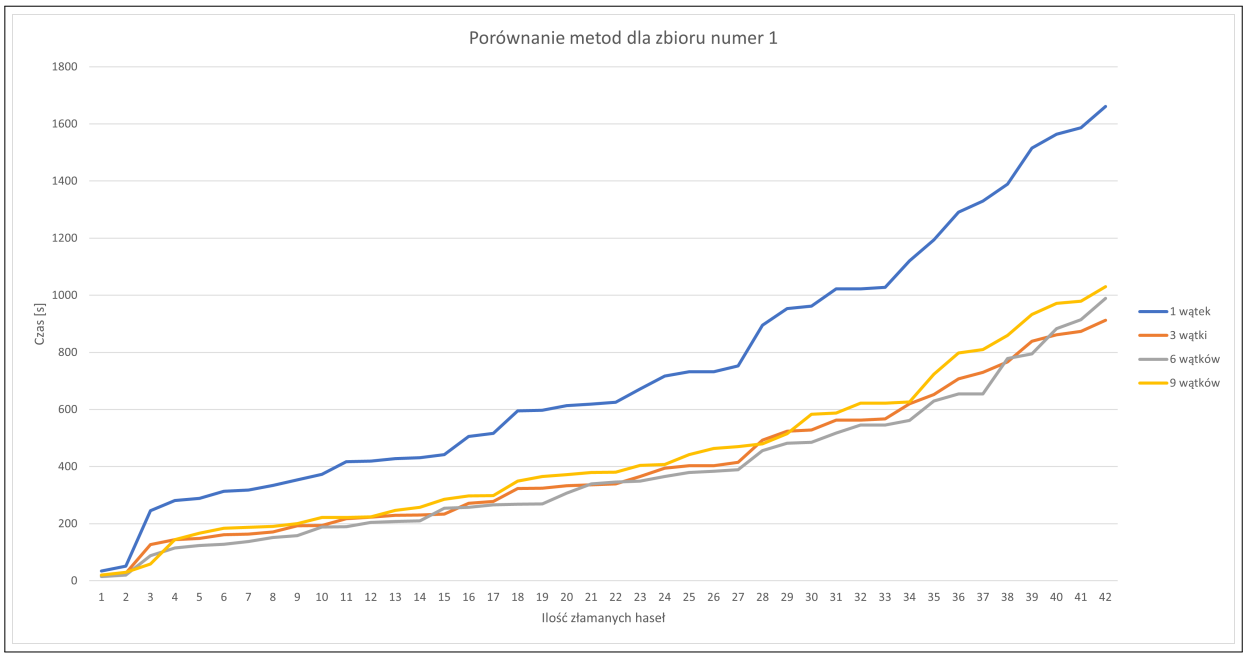
Rysunek 6.1: Wykres przedstawiający czas działania programu w zależności od ilości złamanych haseł.

Na wykresie 6.1 wyraźnie widać, że od określonego momentu czas potrzebny do złamania hasła gwałtownie rośnie. Ten wzrost jest konsekwencją pojawienia się w zbiorze danych haseł zawierających większą liczbę przy haśle, na przykład "TECHNOLOGY51". Zwiększenie liczby przy haśle znacząco wydłuża czas, który program potrzebuje na jego złamanie.

Metody były aktywne wobec pierwszego zbioru danych przez okres 8 godzin i 20 minut. Zestawienie wyników uzyskanych z ich działania znajduje się w poniższej tabeli

	1 wątek	**3 wątki**	**6 wątków**	**9 wątków**
Ilość złamanych haseł	50	52	52	53

Celem bardziej szczegółowej analizy czasu potrzebnego na złamanie haseł przedstawionych na wykresie 6.1, postanowiono wykluczyć kilka ostatnich haseł, które znacząco wydłużały czas potrzebny do złamania pojedynczego hasła. Dzięki tej modyfikacji wykres prezentuje się w sposób bardziej klarowny.

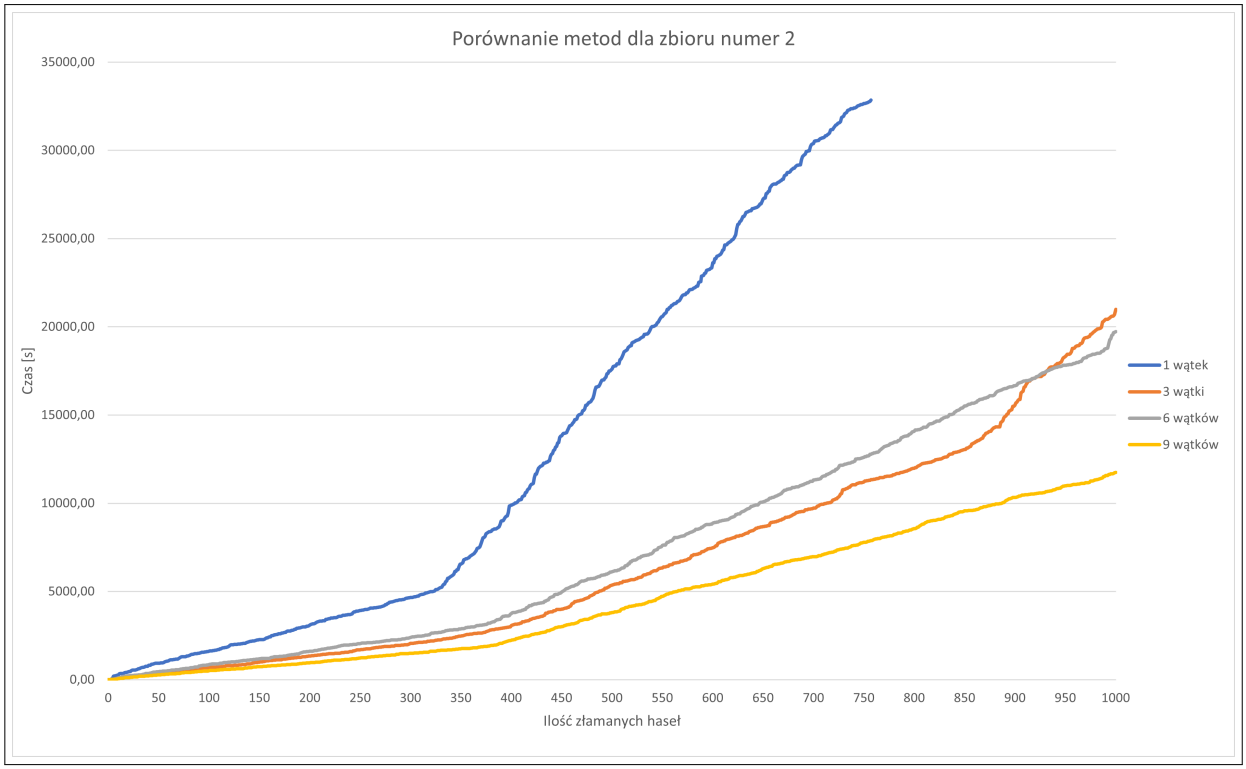


Rysunek 6.2: Wykres przedstawiający czas działania programu w zależności od ilości złamanych haseł.

Na wykresie 6.2 widoczna jest wyraźna dysproporcja w efektywności działania pomiędzy metodą sekwencyjną a metodą równoległą.

6.2 Drugi zbiór danych

Drugie zbiór danych składał się z 1000 zahaszowanych haseł, zgodnych z kryteriami przedstawionymi w rozdziale 3. W przypadku tego zbioru, liczby towarzyszące słowom mieszczą się w przedziale od 0 do 5.



Rysunek 6.3: Wykres przedstawiający czas działania programu w zależności od ilości złamanych haseł.

Metody kontynuowały swoje działanie aż do momentu złamania całego zestawu 1000 zahaszowanych haseł. Warto zauważyć, że metodzie sekwencyjnej udało się złamać 757 haseł w akceptowalnym czasie (9 godzin 7 minut). Szczegółowe rezultaty działania poszczególnych metod zostały przedstawione w poniższej tabeli.

	1 wątek	**3 wątki**	**6 wątków**	**9 wątków**
Ilość złamanych haseł	757	1000	1000	1000

Czas działania poszczególnych metod został przedstawiony w tabeli poniżej.

	1 wątek	**3 wątki**	**6 wątków**	**9 wątków**
Czas złamania 757 haseł	32841,32s	11330,27s	12782,84s	7890,74s
Czas złamania 1000 haseł	-	20981,89s	19731,15s	11740,45s

Na postawie czasu złamania 757 można wyliczyć przyśpieszenie empiryczne.

Dla 3 wątków producenta:

$$P_e = \frac{32841,32s}{11330,27s} \approx 2,9$$

Dla 6 wątków producenta:

$$P_e = \frac{32841,32s}{12782,84s} \approx 2,57$$

Dla 9 wątków producenta:

$$P_e = \frac{32841,32s}{7890,74s} \approx 4,16$$

Tabela porównywująca przyśpieszenie empiryczne z przyśpieszeniem teoretycznym.

	Przyśpieszenie teoretyczne	**Przyśpieszenie empiryczne**
4 wątki	2,5	2,9
7 wątków	3,18	2,57
10 wątków	3,57	4,16

7 Wnioski

- Porównanie między przyśpieszeniem empirycznym a teoretycznym dla różnych liczb wątków producenta wskazuje na zróżnicowane wyniki.
- Biorąc pod uwagę fakt, że metoda sekwencyjna nie złamała 1000 haseł, ze względu na swój czas działania, obliczone przyśpieszenia należy traktować jako przybliżone.
- W niektórych przypadkach, jak dla 3 wątków, przyśpieszenie empiryczne przewyższało wartości teoretyczne, co może wynikać z różnych czynników środowiskowych.
- W innych przypadkach, jak dla 7 wątków, uzyskane przyśpieszenie empiryczne było niższe od przewidywanego teoretycznie.
- Warto zauważyć, że dla 9 wątków uzyskane przyśpieszenie empiryczne było najwyższe, co może wskazywać na optymalizację działania algorytmu w przypadku wykorzystania większej liczby wątków.
- Wyraźnie zaobserwowano, że dodanie większej liczby znaków, takiej jak w przypadku hasła "TECHNOLOGY51", ma bezpośredni i wyraźny wpływ na czas potrzebny do złamania hasła. Im większa złożoność hasła (w postaci większej liczby przy słowie), tym czas potrzebny na jego złamanie wzrastał bardzo gwałtownie.
- Wraz z zwiększeniem ilości wątków w procesie złamywania haseł, obserwuje się wzrost przyśpieszenia empirycznego. Jednakże, ten wzrost przyśpieszenia jest powolny i wykazuje cechy

zbliżone do przyrostu logarytmicznego.

- Czas obliczeń może być związane z właściwościami sprzętowymi, architekturą algorytmu czy też specyficznymi warunkami środowiskowymi, które stają się bardziej widoczne przy wzroście ilości wątków.