Sprawozdanie

Zastosowanie MPI oraz OpenMP w procesie generowania skrótów MD5 dla kolejnych kombinacji zbioru znaków

Programowanie Równoległe i Rozproszone

Łukasz Niedzielski, Dominik Madej

15 stycznia 2025

Spis treści

1 Wprowadzenie **2**

2 Opis algorytmu **2**
2.1 Generowanie kolejnych kombinacji bez powtórzeń 2

3 Implementacja algorytmu **3**
3.1 Wersja sekwencyjna 4
3.2 Wersja zrównoleglenia OpenMP 5
3.3 Wersja zrównoleglenia MPI 6
3.4 Wersja MPI + OpenMP 7

4 Wyniki i analiza **8**
4.1 Parametry środowiska uruchomieniowego 8
4.2 Metodologia pomiarów 8
4.3 Wyniki 8

5 Wnioski **9**

1 Wprowadzenie

Celem pracy było zaimplementowanie i analiza algorytmu obliczania i porównywania skrótów MD5 dla kolejnych kombinacji znaków o ustalonej długości z podanego alfabetu. Jednym z zastosowań tego algorytmu jest łamanie haseł poprzez szukanie wiadomości, której skrót jest taki sam jak szukanego hasła. W sprawozdaniu opisano implementację algorytmu w wersji sekwencyjnej oraz równoległej (z wykorzystaniem OpenMP, MPI oraz kombinacji MPI+OpenMP). Przedstawiono także uzyskane wyniki w postaci wykresów oraz omówiono wpływ zastosowanych metod zrównoleglenia na wydajność programu.

2 Opis algorytmu

Algorytm polega na:

1. Generowaniu kolejnych kombinacji bez powtórzeń znaków o określonej długości z ustalonego alfabetu.
2. Obliczaniu skrótu MD5 dla każdej kombinacji.
3. Porównywaniu uzyskanego skrótu z poszukiwanym skrótem.
4. Zakończeniu działania programu po znalezieniu pasującej kombinacji lub po sprawdzeniu określonej liczby kombinacji.

Dane wejściowe algorytmu to:

- Długość wiadomości (np. 8 znaków),
- Alfabet (np. `abcdefghijklmnopqrstuvwxyz`),
- Skrót MD5 poszukiwanej wiadomości (np. `9e65ff77204283d4a951e12d2bb8357e`),
- Liczba wiadomości do sprawdzenia (np. 2^{24}).

Dane wyjściowe to znaleziona wiadomość lub informacja o jej braku.

2.1 Generowanie kolejnych kombinacji bez powtórzeń

Sposób generowania kolejnych kombinacji znaków opiera się na reprezentacji pozycyjnej (podobnej do systemu liczbowego). Każdy znak w kombinacji jest traktowany jako "cyfra", a alfabet jako "podstawa" tego systemu. Iterując przez wszystkie możliwe liczby w takim systemie, generujemy kombinacje w sposób deterministyczny i przewidywalny. Dzięki temu można w prosty sposób podzielić zakres kombinacji między różne wątki lub procesy, co jest kluczowe w implementacji równoległej.

Możliwość zastosowania tego sposobu wynika z jego deterministycznej natury, która gwarantuje wygenerowanie wszystkich możliwych kombinacji przy minimalnym narzucie obliczeniowym. Dodatkowo reprezentacja pozycyjna pozwala na łatwe przeliczanie zakresów kombinacji w środowiskach równoległych, co znacznie upraszcza implementację zrównoleglenia algorytmu.

3 Implementacja algorytmu

Przygotowane implementacje korzystają z poniższej procedury w celu sprawdzenia `hashes_to_check` kolejnych kombinacji, zaczynając od kombinacji o numerze `combination_number`.

```
struct search_result check_batch(size_t combination_number,
   size_t length, size_t hashes_to_check, uint8_t
   orginal_hash[16]) {
    char *password = NULL;
    size_t indices[PASSWORD_LENGTH] = {};
    get_nth_combinaiton(indices, combination_number, length);

    int pos = length - 1;
    char buffer[PASSWORD_LENGTH + 1] = {0};
    uint8_t result[16];

    size_t checked_hashes = 0;
    while (checked_hashes < hashes_to_check && pos >= 0) {
        // ustawienie kolejnej kombinacji z powtórzeniami
        next_combination(buffer, indices, &pos);

        // wyznaczenie skrótu kombinacji
        md5String(buffer, length, result);
        checked_hashes++;

        // sprawdzanie czy skróty się zgadzają
        if (memcmp(orginal_hash, result, 16) == 0) {
            // znaleziono pasującą wiadomość - koniec algorytmu
            password = clone(buffer);
            break;
        }
        // czyszczenie wiadomosci
        memset(buffer, 0, PASSWORD_LENGTH);
    }

    struct search_result search_result = {.password = password,
       .checked_passwords = checked_hashes};

    return search_result;
}
```

Listing 1: Sprawdzanie partii wiadomości.

3.1 Wersja sekwencyjna

W wersji sekwencyjnej algorytm przetwarza wiadomości w partiach, z których każda ma określoną wielkość (`BATCH_SIZE`). Generowanie kombinacji znaków, obliczanie skrótów MD5 oraz porównywanie wyników odbywa się w jednym wątku. Przykładowy kod implementacji przedstawiono poniżej:

```
struct search_result check_hashes(size_t length, size_t
   hashes_to_check, uint8_t orginal_hash[16]) {

    struct search_result search_result = {.password = NULL,
       .checked_passwords = 0};

    int finished = 0;
    size_t total_checked_hashes = 0;

    // podział na partie o rozmiarze BATCH_SIZE
    for (size_t n = 0; n <= hashes_to_check / BATCH_SIZE; n++) {
        if (finished == 1) {
            continue;
        }

        size_t combination_number = BATCH_SIZE * n;
        size_t to_check = min(BATCH_SIZE, hashes_to_check -
           combination_number);

        search_result = check_batch(combination_number, length,
           to_check, orginal_hash);
        total_checked_hashes += search_result.checked_passwords;

        if (search_result.password != NULL) {
            finished = 1;
            printf("finished: %p\n", search_result.password);
        }
        if (total_checked_hashes > hashes_to_check) {
            finished = 1;
        }
    }

    return search_result;
}
```

3.2 Wersja zrównoleglenia OpenMP

Algorytm sekwencyjny został zrównoleglony z wykorzystaniem dyrektywy `#pragma omp parallel for`, która umożliwia równoczesne przetwarzanie wielu partii wiadomości na różnych wątkach. Implementacja z wykorzystaniem OpenMP:

```
struct search_result check_hashes(size_t length, size_t
   hashes_to_check, uint8_t orginal_hash[16]) {

    struct search_result search_result = {.password = NULL,
                                          .checked_passwords =
                                             0};

    int finished = 0;
    size_t total_checked_hashes = 0;

    // podział na partie o rozmiarze BATCH_SIZE
#pragma omp parallel for shared(total_checked_hashes, finished)
    for (size_t n = 0; n <= hashes_to_check / BATCH_SIZE; n++) {
        if (finished == 1) {
            continue;
        }

        size_t combination_number = BATCH_SIZE * n;
        size_t to_check = min(BATCH_SIZE, hashes_to_check -
           combination_number);

        search_result = check_batch(combination_number, length,
           to_check, orginal_hash);
#pragma omp atomic
        total_checked_hashes += search_result.checked_passwords;

        if (search_result.password != NULL) {
            finished = 1;
            printf("finished: %p\n", search_result.password);
        }
    }

    search_result.checked_passwords = total_checked_hashes;

    return search_result;
}
```

3.3 Wersja zrównoleglenia MPI

Zrównoleglenie z wykorzystaniem MPI polega na podziale zakresu kombinacji między procesy, które przetwarzają swoje części niezależnie. Przykładowy kod implementacji:

```
void run_tests(int rank, int size, size_t hashes_to_check,
   uint8_t orginal_hash[16]) {
    size_t hashes_per_process = hashes_to_check / size;

    // wyznaczanie pierwszej i ostatniej kombinacji dla danego
       procesu
    size_t start = rank * hashes_per_process;
    size_t end = (rank == size - 1) ? hashes_to_check
                                    : (start +
                                       hashes_per_process);
    size_t total_checked_hashes = 0;

    struct search_result search_result = {.password = NULL,
                                          .checked_passwords =
                                             0};

    for (size_t combination_number = start; combination_number <
       end;
         combination_number += BATCH_SIZE) {
        size_t to_check = min(BATCH_SIZE, end -
           combination_number);

        search_result =
            check_batch(combination_number, PASSWORD_LENGTH,
               to_check, orginal_hash);
        total_checked_hashes += search_result.checked_passwords;

        if (search_result.password != NULL) {
            break;
        }
    }

    if (search_result.password != NULL) {
        printf("Process %d found the password: %s\n", rank,
               search_result.password);
    }
}
```

3.4 Wersja MPI + OpenMP

Połączenie MPI i OpenMP umożliwia wykorzystanie wielu wątków w ramach jednego procesu MPI. Poniższa procedura wołana jest przez każdy z procesów, `rank` to numer procesu, `size` to ilość procesów;

```
void run_tests(int rank, int size, struct test_result *result,
   uint8_t orginal_hash[16]) {
    size_t hashes_per_process = PERMUTATIONS_TO_CHECK / size;

    size_t start = rank * hashes_per_process;
    size_t end = (rank == size - 1) ? PERMUTATIONS_TO_CHECK
                                    : (start +
                                       hashes_per_process);
    size_t total_checked_hashes = 0;

    struct search_result search_result = {.password = NULL,
                                          .checked_passwords =
                                             0};

    int i = 0;

    volatile int finished = 0;

#pragma omp parallel for shared(total_checked_hashes, finished)
    for (size_t permutation_number = start; permutation_number <
       end;
         permutation_number += BATCH_SIZE) {

        if (finished == 1) {
            continue;
        }

        size_t to_check = min(BATCH_SIZE, end -
           permutation_number);

        search_result =
            check_batch(permutation_number, PASSWORD_LENGTH,
               to_check);
        total_checked_hashes += search_result.checked_passwords;

        if (search_result.password != NULL) {
            finished = 1;
            printf("finished: %p\n", search_result.password);
        }
    }

    result->total_checked_hashes = total_checked_hashes;
}
```

4 Wyniki i analiza

Poniżej przedstawiono wyniki pomiarów czasu wykonania dla różnych wariantów algorytmu. Każdy program przetworzył 16 777 216 wiadomości, a pomiary wykonano na serwerze `torus` (szczegóły poniżej).

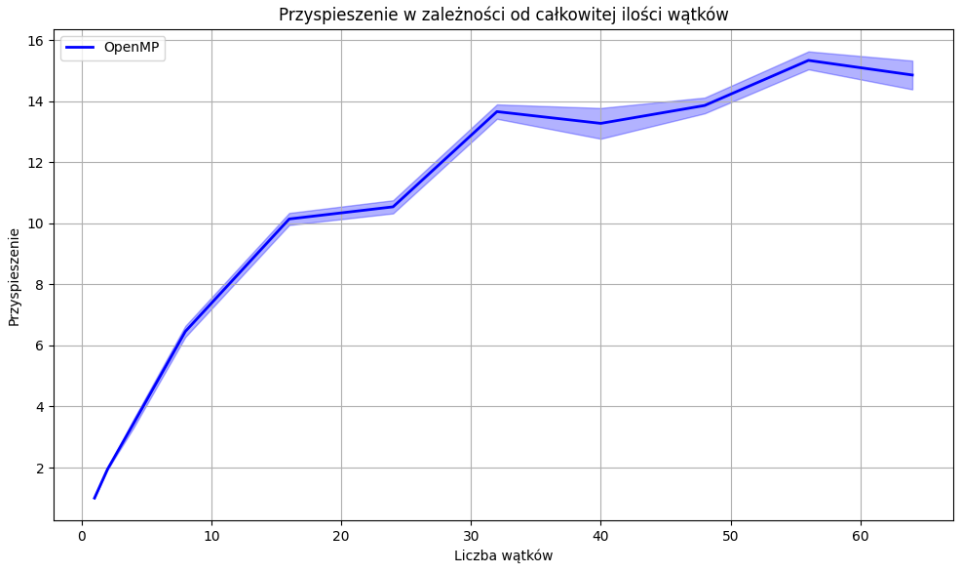
4.1 Parametry środowiska uruchomieniowego

- Procesor: Intel Xeon Processor (Skylake), 32 rdzenie, taktowanie 2.29 GHz,
- Pamięć: 62 GiB RAM,
- System operacyjny: Debian GNU/Linux 11 (bullseye),
- Kompilator: GCC 8.5,
- Wersja MPI: 4.1.0,
- Biblioteka OpenMP: wbudowana w GCC.

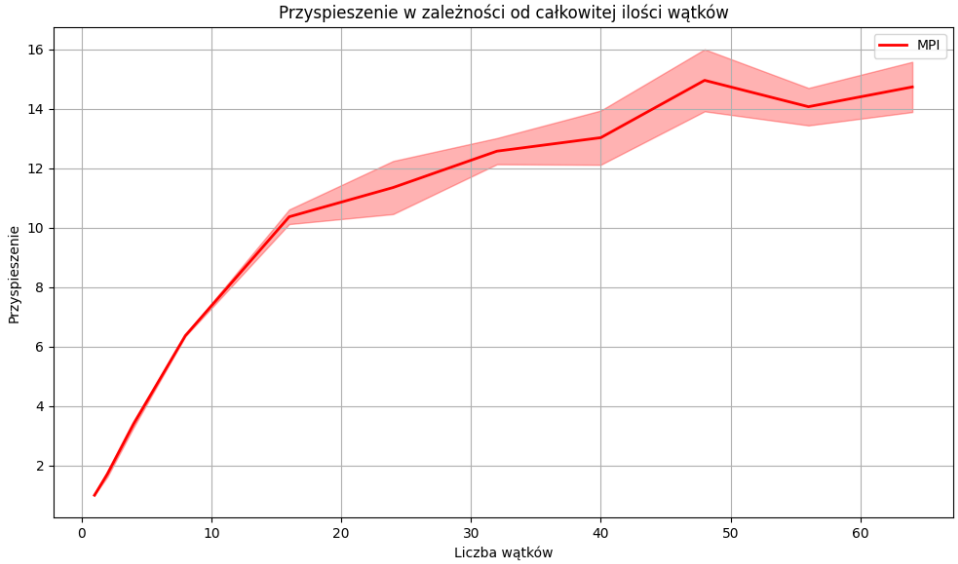
4.2 Metodologia pomiarów

Każda konfiguracja była uruchamiana 10 razy, a uzyskane czasy uśredniono oraz wyznaczono 95% przedziały ufności. Czas wykonania mierzono jako czas trwania głównej pętli przetwarzającej wiadomości. Wyniki przedstawiono na wykresach poniżej.

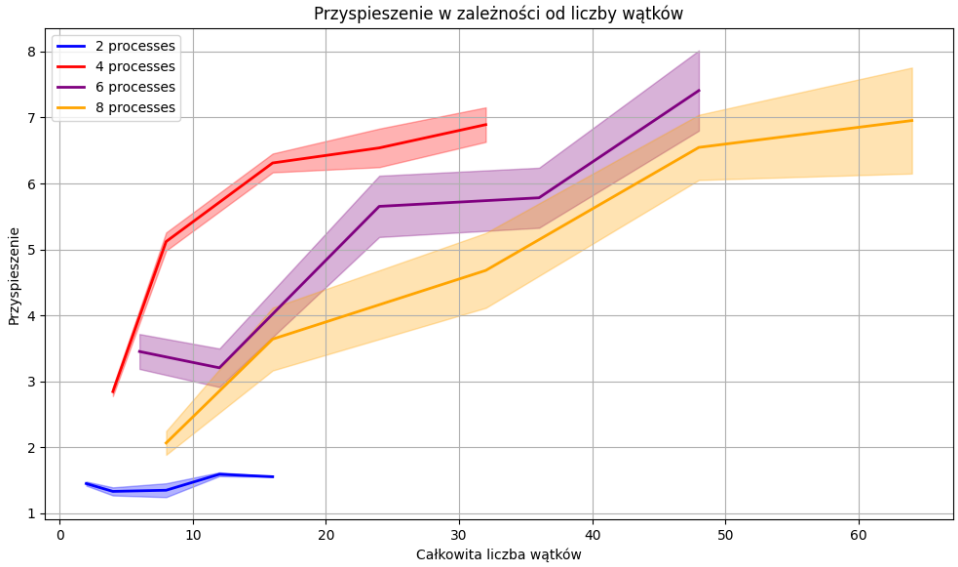
4.3 Wyniki



Rysunek 1: Przyspieszenie dla OpenMP



Rysunek 2: Przyspieszenie dla MPI



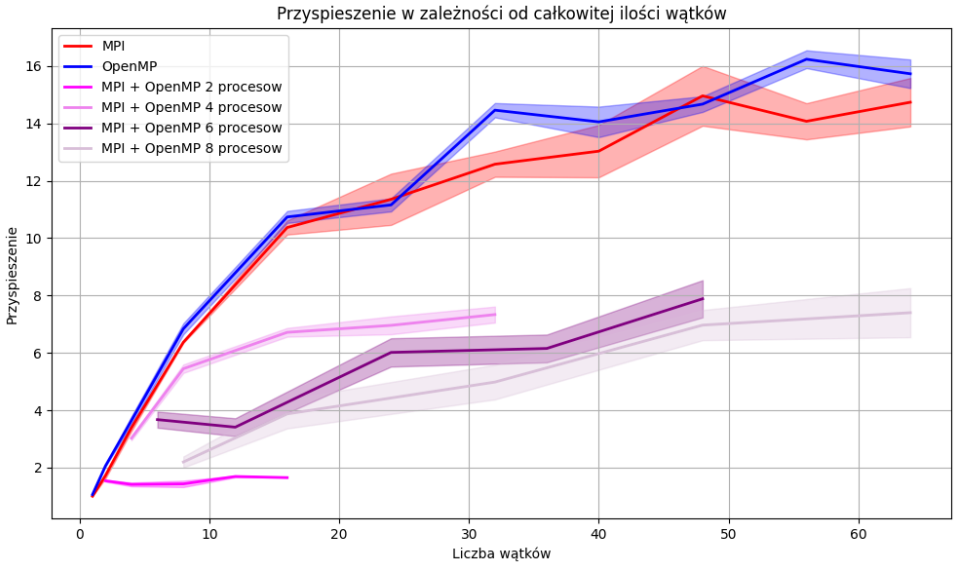
Rysunek 3: Przyspieszenie dla MPI+OpenMP

5 Wnioski

Najlepsze wyniki uzyskały implementacje stosujące MPI lub OpenMP.

Kombinacja MPI+OpenMP okazała się mniej wydajna ze względu na narzuty obu technologii, jednak umożliwia skalowanie algorytmu na wiele maszyn.

Dalsze optymalizacje mogą obejmować lepsze zarządzanie zasobami w konfiguracji MPI+OpenMP oraz dostosowanie wielkości partii przetwarzanych wiadomości.



Rysunek 4: Porównanie przyspieszeń dla wszystkich metod