

# Programowanie Współbieżne 2024/2025 - Projekt

Piotr Pasula

Motywacją do wyboru Z5 jako zadania do projektu było użycie, już wcześniej, wielu optymalizacji takich jak operacje bitowe i pragma zamim moje submit dostał OK na satori. Chciałem zobaczyć jak bardzo uda się przyspieszyć rozwiązanie do tego zadania.

W moim rozwiązaniu zadania Z5 z kursu PP2023/2024 wykorzystywałem maski 16-bitowe do kodowania informacji o pracownikach. Decyzja była uargumentowana możliwością wcześniejszego obliczenia (w rozsądnym czasie) tablicy wielkości  $2^{16}$ , która miała zapisaną liczbę zapalonych bitów masek 16-bitowych.

Z taką tablicą nie było potrzeby wywoływania *popcount*, za każdym razem kiedy chcieliśmy sprawdzić ilość wspólnych zapalonych bitów 2 masek.

## Optymalizacje programu jednowątkowego

W wersji zoptymalizowanej główną zmianą było zwiększenie maski z 16-bitów do 64-bitów, a co za tym idzie, pozbycie się tablicy z preprocesowanymi danymi. Przygotowanie takiej tablicy, nawet dla maski 32-bitowej, przy użyciu wielu procesorów to za dużo pracy.

Jedyną opcją jest liczenie zapalonych bitów na bieżąco przy użyciu *popcountll*. Ta zmiana przyniosła przyspieszenie rzędu 1.5-2.5x na małych i średnich testach, oraz 3-4x na dużych testach.

To znaczy, że dostęp do pamięci cache jest wolniejszy niż zliczanie liczby bitów *popcountll*, który został dodatkowo zoptymalizowany przez kompilator przy użyciu *#pragma GCC target("popcnt")*.

## Wersje wielowątkowe - std::thread i OpenMP

Dominującymi pętlami programu o złożoności  $O(n^2m)$  są, i właśnie je będziemy chcieli zrównoleglić

```
1. for (i = 0; i < n; i++)
    2. for (j = i + 1; j < n; j++)
        3. for (l = 0; l < bit_itr(=m/64); l++)
            __builtin_popcountll((bit[i][l]) & (bit[j][l]))
```

Przez  $T$  oznaczmy liczbę dostępnych wątków dla programu.

Pętla 3. nie nadaje się do zrównoleglenia, ponieważ praca dla pojedynczego wątku to około  $\frac{m}{64T}$ . Ilość pracy będzie zbyt mała dla sensownego działania wątku, a także pojawi się potrzeba częstej synchronizować wątków, co za tym idzie, czas bezczynności wielu wątków czekających na skończenie pracy przez ostatni wątek.

Pętla 2. jest już znacznie lepszym kandydatem do zrównoleglenia ponieważ praca dla pojedynczego wątku będzie wynosić około  $\frac{n}{64} \frac{m}{T}$ , jednak zrównoleglenie pętli 1. dało najlepsze wyniki w przypadku std::thread, jak i OpenMP.

W przypadku std::thread, dzielimy pętlę 1. na  $T$ , rozłącznych, równych części pętli, którymi zajmują się dane thready. Zauważmy, że w takim rozwiązaniu ilość pracy do wykonania nie będzie dokładnie taka sama, ze względu na to, że ilość iteracji w pętli 2. jest zależna od obecnej iteracji w pętli 1.

Natomiast w przypadku OpenMP, nad pętlą dodajemy jedynie *#pragma omp parallel for num\_threads(T)*.

Optymalizowanie innych pętli programu takich jak kodowanie maski bitowej pracownika,

```
for (i = 0; i < n; i++)
    cin >> s;
    --> for (j = 0; j < m; j++)
        bit[i][j >> 6] |= (((unsigned long long)s[j] - '0') << (j & 0x3F));
```

lub podliczania specjalnych pracowników,

```
for (i = 0; i < n; i++)
    if (count[i] >= u) special++;
```

nie przyniosłoby, żadnych dobrych rezultatów ze względu na małą pracę do rozdysponowania na wątki, a także potrzebę synchronizacji wątków.

# Wyniki

Wszystkie testy były przeprowadzane na serwerze studenckim z procesorem

Intel(R) Xeon(R) Gold 6154 @ 3.00GHz

Programy były testowane na 100 losowych zestawach paczek. Każdy zestaw posiadał 3 paczki:

$Z$  - ilość testów w paczce,  $n$  - ilość pracowników i  $m$  - ilość specjalizacji

- small -  $Z = 1000$ ,  $n \leq 300$ ,  $m \leq 3000$

- medium -  $Z = 100$ ,  $n \leq 1500$ ,  $m \leq 8000$

- big -  $Z = 10$ ,  $n \leq 5000$ ,  $m \leq 50000$

Wyniki każdego zestawu zostały zapisane w pliku *results.txt*, które potem uśredniłem.

	small	medium	big
Orginal z5	1.598	7.782	56.374
Single-threaded z5	0.92	2.637	15.673
4 C++ Threads	1.546	2.102	8.438
4 OpenMP Threads	0.79	1.658	8.253
16 C++ Threads	2.136	1.53	3.847
16 OpenMP Threads	0.747	1.116	3.701
64 C++ Threads	4.023	1.795	2.66
64 OpenMP Threads	1.187	1.262	2.552

Dla dużych zestawów, czas wykonania programów zmniejsza się pierwiastkowo od ilości użytych wątku, co było oczekiwanym rezultatem.

Std::thread i OpenMP osiągają podobne czasy, które różnią się od siebie zaledwie o kilka procent.

Dla mniejszych testów, osiągnięte przyspieszenia nie są aż tak imponujące, a nawet pojawiają się anomalie, w postaci zwiększenia czasu wykonania programu wraz ze zwiększeniem liczby wątków.

Sz szczególnie kiepsko radzi sobie std::thread, który jest zmuszony do pracy z dokładną ilością wątków, natomiast OpenMP może zmniejszyć liczbę wątków podanych w dyrektywie *num\_threads()*, jeśli uzna to za stosowne.

## Obsługa *Makefile*

*make gen* - generuje paczki testów zgodnie z parametrami podanymi w plikach folderu */tests/gen*, gdzie 1 liczba w pliku oznacza  $Z$ , 2- $n$  i 3- $m$ .

*make z5XYZ* - kompiluje i włącza podany program z daną ilością wątków, używając danych z */tests/input* i zapisuje wyniki w odpowiednich plikach w */tests/output*

*make all* - kompiluje i włącza w sekwencji wszystkie programy