Mnożenie macierzy przez wektor

Dane typu całkowitego (int)

Tomasz Rybiński - dokument
Nikita Lysiuk – kod sekwencyjny i równoległy
Kacper Górak - kod sekwencyjny i równoległy

Informacje na temat architektury testowej

Model procesora: Intel Xeon CPU E5-2670 v3
Ilość rdzeni na socket: 12
Ilość wątków na rdzeń: 2
Ilość socketów: 2
Maksymalne taktowanie pojedyńczego rdzenia: 3.100MHz
Taktowanie podstawowe pozostałych rdzeni: 2.300Mhz
Pamięć podkręczna: 30720KiB
System operacyny: AlmaLinux 8.7
Ilości pamięci RAM: 125GiB
Werska kernela: 4.18.0-425.13.1.el8_7.x86_64
Wersja kompilatora: g++ (GCC) 8.5.0 20210514 (Red Hat 8.5.0-20)

PROGRAM SEKWENCYJNY

Kod programu sekwencyjnego:
Sposób kompilacji: **g++ -fopenmp ./matrix_2D_vector_single.cpp**
Doświadczenie zostało przeprowadzone na dwóch macierzach testowych

```cpp
#include <iostream>
#include <random>
#include <vector>
#include <cmath>
#include "omp.h"

int main() {
    constexpr int N = 100'000;
    constexpr int M = 145'000;

    std::vector<std::vector<long long>> matrix;
    try {
        matrix.reserve(N);
        for (int i = 0; i < N; ++i) {
            matrix.emplace_back(std::vector<long long>(M));
        }
    } catch (const std::bad_alloc& e) {
        std::cerr << "Manual allocation failed: " << e.what() << std::endl;
        return 1;
    }
    std::vector<long long> vector(M);
    std::vector<long long> result(N);
```

Ten blok kodu tworzy dwuwymiarową macierz o wymiarach *N × M*, wektor o rozmiarze *M* oraz wektor wynikowy o rozmiarze *N*, wykorzystując *std::vector*. Użyto bloku *try-catch*, aby przechwycić ewentualny wyjątek *std::bad_alloc*, który może wystąpić przy alokacji dużej ilości pamięci.

```cpp
    std::random_device rd;

    std::mt19937 gen(rd());
    std::uniform_int_distribution<long long> dis(-1'000'000'000, 1'000'000'000);
    //std::uniform_int_distribution<long long> dis(-10, 10);

    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < M; ++j) {
            matrix[i][j] = dis(gen);
        }
    }

    for (int i = 0; i < M; ++i) {
        vector[i] = dis(gen);
    }
```

Ten fragment kodu odpowiada za wypełnienie macierzy oraz wektora losowymi liczbami całkowitymi z zakresu od −1 000 000 000 do 1 000 000 000. Użyto *std::random_device* jako źródła entropii do inicjalizacji generatora liczb pseudolosowych *std::mt19937*. *std::uniform_int_distribution* zapewnia równomierny rozkład wartości w zadanym przedziale.
Każdy element macierzy i wektora jest przypisywany w pętli, przy czym każda wartość jest generowana osobno.

```cpp
    double start_time = omp_get_wtime();
```

Zapisuje czas rozpoczęcia obliczeń przy użyciu funkcji omp_get_wtime(), która zwraca aktualny czas ścienny (wall time) w sekundach.

```cpp
    for (int i = 0; i < N; i++) {
        long long sum = 0;
        for (int j = 0; j < M; j++) {
            long long val = matrix[i][j] * vector[j];
            double tmp = std::sin(static_cast<double>(val));
            int int_val = static_cast<int>(tmp * 1000);

            sum += int_val;
        }
        result[i] = sum;
    }
```

Ta pętla wykonuje mnożenie macierzy przez wektor: dla każdego wiersza macierzy obliczana jest suma iloczynów odpowiadających elementów wiersza i wektora. Każdy iloczyn jest następnie przekształcany na wartość zmiennoprzecinkową, przetwarzany funkcją *sin()* w celu dodania kosztu obliczeniowego, a następnie skalowany (× 1000) i rzutowany na typ całkowity. Wynikowa suma jest zapisywana w wektorze.

```cpp
    double end_time = omp_get_wtime();

    std::cout << "\nMatrix " << N << "x" << M << "\n";
    std::cout << "\nExecution Time (OpenMP): " << (end_time - start_time) << " se

    return 0;
}
```

Zapisuje czas zakończenia obliczeń, umożliwiając późniejsze obliczenie całkowitego czasu wykonania fragmentu kodu.

Test nr 1. 2’000 x 4’000 / Test nr 2. 100’000 x150’000

Wyniki dla programu sekwencyjnego

Czas dla **2000 x 4000**	Czas dla **100000 x150000**
0.839403 (s)	451.756 (s) ~ 7.5 minuty

Wyniki te posłużą nam zaraz jako punkt odniesienia do programu równoległego

PROGRAM RÓWNOLEGŁY

Sposób kompilacji: **g++ -fopenmp ./matrix_2D_vector.cpp**
Program równoległy był uruchamiany za pomocą skryptu batch
za pomocą którego w pętli był uruchamiany program równoległy z odpowiednią ilością wątków,
która była ustalana za pomocą zmiennej środowiskowej **OMP_NUM_THREADS**

```
for threads in {1..24}; do
    echo "Running with $threads thread(s)..." >> "$LOG_FILE"

    OMP_NUM_THREADS=$threads /usr/bin/time -v "$EXECUTABLE" > "$OUT_FILE" 2> "$TEMP_FILE"

    N=$(grep "Matrix" "$OUT_FILE" | awk '{split($2,a,"x"); print a[1]}')
    M=$(grep "Matrix" "$OUT_FILE" | awk '{split($2,a,"x"); print a[2]}')

    exec_time=$(grep "Execution Time (OpenMP)" "$OUT_FILE" | awk -F ': ' '{print $2}')
    threads_used=$(grep "Threads used" "$OUT_FILE" | awk -F ': ' '{print $2}')

    max_mem=$(grep "Maximum resident set size" "$TEMP_FILE" | awk -F': ' '{print $2}')
    cpu_percent=$(grep "Percent of CPU this job got" "$TEMP_FILE" | awk -F': ' '{print $2}')

    error_msg=$(grep -iE "error|failed|exception" "$TEMP_FILE" | head -n 1)
    if [ -n "$error_msg" ]; then
        echo "Error: $error_msg" >> "$LOG_FILE"
    fi

    echo "Matrix size: $N x $M | Threads: $threads_used | Execution time: $exec_time | Max Mem: ${max_mem} KB | CPU: $cpu_percent" >> "$LOG_FILE"
    echo "_____" >> "$LOG_FILE"
done


rm "$TEMP_FILE" "$OUT_FILE"
```

Kod programu równoległego

```cpp
#include <iostream>
#include <random>
#include <vector>
#include <cmath>
#include "omp.h"

int main() {
    constexpr int N = 100'000;
    constexpr int M = 145'000;

    std::vector<std::vector<long long>> matrix;
    try {
        matrix.reserve(N);
        for (int i = 0; i < N; ++i) {
            matrix.emplace_back(std::vector<long long>(M));
        }
    } catch (const std::bad_alloc& e) {
        std::cerr << "Manual allocation failed: " << e.what() << std::endl;
        return 1;
    }
    std::vector<long long> vector(M);
    std::vector<long long> result(N);

    std::random_device rd;

    #pragma omp parallel
    {
        std::mt19937 gen(rd());
        std::uniform_int_distribution<long long> dis(-1'000'000'000, 1'000'000'000);
        //std::uniform_int_distribution<long long> dis(-10, 10);

        #pragma omp for collapse(2)
        for (int i = 0; i < N; ++i) {
            for (int j = 0; j < M; ++j) {
                matrix[i][j] = dis(gen);
            }
        }

        #pragma omp for
        for (int i = 0; i < M; ++i) {
            vector[i] = dis(gen);
        }
    }

    double start_time = omp_get_wtime();

    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        long long sum = 0;
        for (int j = 0; j < M; j++) {
            long long val = matrix[i][j] * vector[j];
            double tmp = std::sin(static_cast<double>(val));
            int int_val = static_cast<int>(tmp * 1000);

            sum += int_val;
        }
        result[i] = sum;
    }

    double end_time = omp_get_wtime();

    std::cout << "\nMatrix " << N << "x" << M << "\n";
    std::cout << "\nExecution Time (OpenMP): " << (end_time - start_time) << " seconds\n";

    // Część kodu wyświetlająca liczbę użytych wątków
    #pragma omp parallel
    {
        #pragma omp single
        {
            // Funkcja omp_get_num_threads() zwraca liczbę uruchomionych wątków
            std::cout << "Threads used: " << omp_get_num_threads() << std::endl;
        }
    }

    return 0;
}
```

#include "omp.h" dodaje do programu możliwość korzystania z funkcji i dyrektyw biblioteki OpenMP.

#pragma omp parallel tworzy równoległą sekcję kodu, która będzie wykonywana jednocześnie przez wiele wątków. Każdy wątek tworzy własną instancję generatora liczb losowych (std::mt19937), aby uniknąć konfliktów współbieżności.
Następnie:

- #pragma omp for collapse(2) rozdziela dwie zagnieżdżone pętle (for (i)... for (j)...) pomiędzy dostępne wątki, co przyspiesza inicjalizację całej macierzy matrix[i][j] losowymi wartościami.
- #pragma omp for rozdziela wypełnianie wektora vector[i] między wątki.

Dzięki zastosowaniu #pragma omp parallel i odpowiednich dyrektyw for, oba procesy — wypełnianie macierzy i wektora — są wykonywane współbieżnie, co znacznie skraca czas inicjalizacji przy dużych rozmiarach danych (N × M, M).

#pragma omp parallel for powoduje równoległe wykonanie pętli
for (int i = 0; i < N; i++), w której każdy wątek oblicza osobny wiersz wyników.
Dla każdego wiersza macierzy (matrix[i]) obliczana jest suma iloczynów elementów wiersza i odpowiadających im wartości wektora (vector[j]).
Każdy iloczyn jest dodatkowo przekształcany:

- rzutowany na double,
- przepuszczany przez funkcję sin(),
- mnożony przez 1000,
- konwertowany na int.

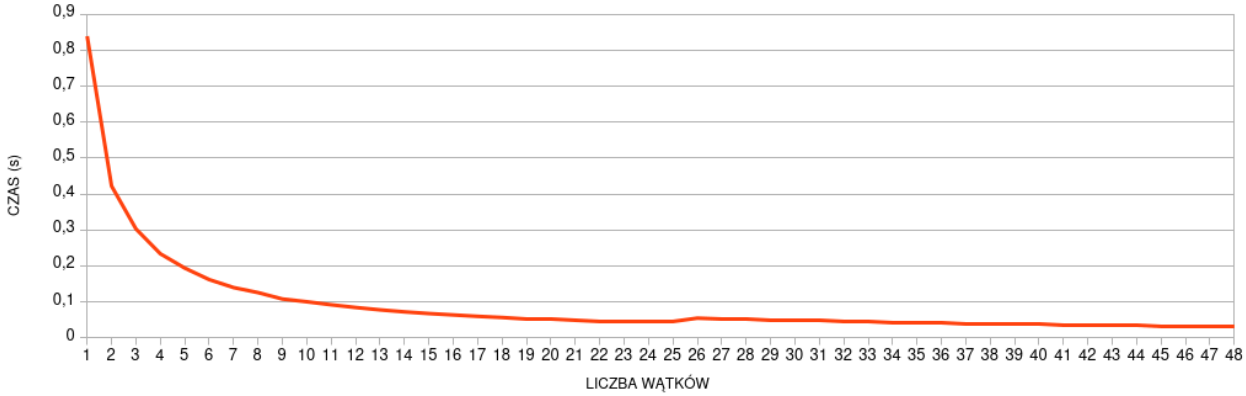
Te operacje matematyczne celowo zwiększają złożoność obliczeniową, aby lepiej pokazać wpływ równoległości (OpenMP) na czas wykonania.
Wynik (sum) zapisywany jest w wektorze result[i].

Po zakończeniu obliczeń, program wyświetla rozmiar macierzy oraz czas wykonania operacji (różnica między end_time a start_time).
Następnie:

- #pragma omp parallel uruchamia blok równoległy,
- #pragma omp single zapewnia, że tylko jeden wątek (spośród wszystkich uruchomionych) wykona zawartą w nim instrukcję,
- omp_get_num_threads() zwraca liczbę wątków użytych przez OpenMP, co pozwala sprawdzić stopień równoległości w praktyce.

Liczba wątków	Czas (s)	Przyśpieszenie
1	0,839403	1
2	0,422075	1,98875318367589
3	0,302588	2,77407894562904
3	0,233407	3,59630602338405
4	0,193572	4,33638646085178
5	0,161534	5,19644780665371
6	0,139382	6,02231995523095
7	0,125406	6,69348356537965
8	0,107521	7,80687493605900
9	0,0999239	8,40042272169121
10	0,091446	9,17922052358769
11	0,0838943	10,00548309003110
12	0,0773314	10,85462050344360
13	0,0717771	11,69457946893930
14	0,0670795	12,51355481182780
15	0,0631954	13,28265981384720
16	0,0592756	14,16102072353550
17	0,056112	14,95942044482460
18	0,0531684	15,78762949421080
19	0,0503708	16,66447624417320
20	0,0483591	17,35770516821030
21	0,0459162	18,28119487239800
22	0,0441495	19,01274080114160
23	0,0441495	19,01274080114160
24	0,0441495	19,01274080114160
25	0,0542739	15,46605274358390
26	0,0519308	16,16387577314430
27	0,0508992	16,49147727272730
28	0,0508992	17,04894292463270
29	0,0492349	17,49860849661350
30	0,0479697	17,95971594908650
31	0,0467381	18,43375160861030
32	0,0455362	18,86358222505880
33	0,0444986	19,59088935878230
34	0,0428466	20,19208148007380
35	0,0415709	20,54721387237960
36	0,0408524	21,21306238801520
37	0,0395701	21,79927803459200
38	0,038506	22,37757558469670
39	0,0375109	22,84898059177400
40	0,036737	23,61581809639290
41	0,0355441	24,08160864800350
42	0,0348566	24,62603414891740
43	0,034086	25,26457884994510
44	0,0332245	26,91161901952800
45	0,0325799	26,37243628412000
46	0,0318288	25,76444372143560
47	0,0311911	26,91161901952800
48	0,0307166	**27,32734091663790**

Wykres dla programu równoległego 2000 x 4000



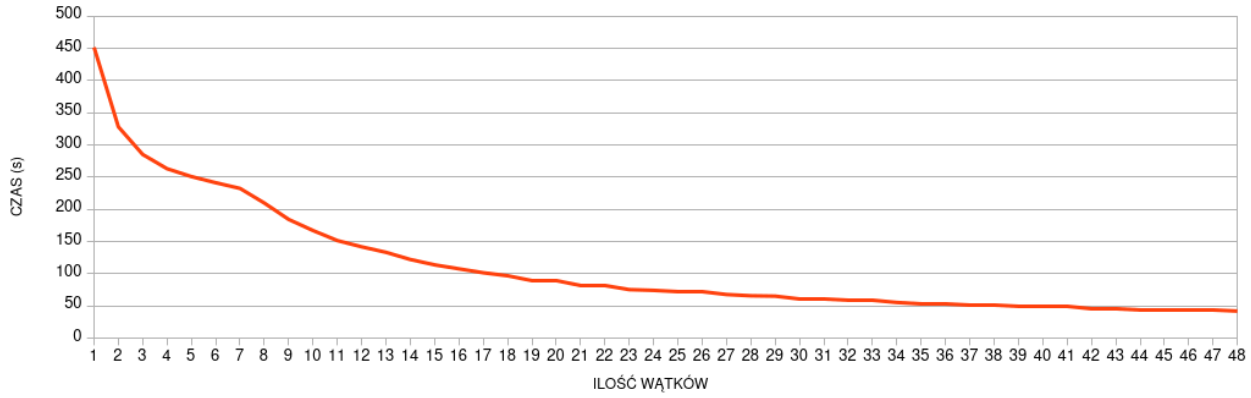
Wnioski dla małego zestawu danych

Jak widać na załaczonych danych problem bardzo dobrze zeskalował się dla niewielkiego zestawu danych. Benefity z wykorzystania wielu wątków do obliczeń okazały się bardzo zadowalające, co pozwoliło przyśpieszyć program 27-krotnie dzięku wykorzystaniu 48 wątków. Zmniejszając czas pracy z 0,839403 (s) do 0,0307166 (s)

Przejdźmy teraz do testu dla dużego zestawu danych
Macierz 100’000 x150’000

Liczba wątków	Czas (s)	Przyśpieszenie
1	451,756	1
2	328,338	1,37588704322984
3	285,369	1,58305912695492
4	263,234	1,71617648176147
5	250,98	1,7999681249502
6	241,43	1,87116762622706
7	232,586	1,94231811029039
8	209,932	2,15191585846846
9	184,41	2,44973699907814
10	167,117	2,70323186749403
11	151,538	2,98114004408135
12	141,844	3,18487916302417
13	133,284	3,38942408691216
14	122,159	3,69809837998019
15	113,815	3,96921319685454
16	107,657	4,19625291434835
17	101,483	4,45154360828907
18	96,8686	4,66359584013808
19	89,166	5,06646030998363
20	88,913	5,08087681216470
21	82,1607	5,49844390322867
22	81,8736	5,51772488323465
23	75,333	5,99678759640529
24	74,1925	6,08897125720255
25	71,9496	6,27878403771529
26	72,2721	6,25076620161861
27	67,7571	6,66728652790630
28	65,77	6,86872434240535
29	65,2121	6,92748738347638
30	60,9096	7,41682756084427
31	60,0615	7,52155707066923
32	59,0778	7,64679795117625
33	58,3506	7,74209691074299
34	55,3681	8,15913856534719
35	53,7035	8,41204018360070
36	53,6953	8,41332481613847
37	51,8469	8,71326926007148
38	50,4188	8,96007044991154
39	49,6051	9,10704746084576
40	48,5854	9,29818422818378
41	48,7111	9,27419007166744
42	46,2665	9,76421384803259
43	45,9828	9,82445610097689
44	43,4644	10,39370151204200
45	44,4877	10,15462700926320
46	44,1038	10,24301760846000
47	42,7668	10,56324064461220
48	41,9951	**10,75735026229250**

Wykres dla programu równoległego 100’000 x150’000



Wnioski dla dużego zestawu danych

Program na dużym zestawie danych również został wyraźnie przyśpieszony dzięki wykorzystaniu wielowątkowości. Zestaw ten nie skaluje się natomiast tak dobrze jak na małym zbiorze danych, przyśpieszenie nie jest tak znaczące jak w przypadku poprzedniego testu, natomiast i tak znacząco przyśpieszyło to nasze obliczenia.

Podsumowanie doświadczenia

Problem mnożenia macierzy przez wektor jest dobrym przykładem benefitów wykorzystania architektóry równoległej.
Zwiększenie ilości wątków wykorzystanej do obliczeń dobrze skaluję się z czasem potrzebnym do wykonania potrzebnych działań.
Zarówno dla małego jak i dużego zestawu danych czepiemy sporo korzyści, przyśpieszając naszą pracę.Warto natomiast podkreślić że współczynnik przyśpieszenia spada wraz z ilością przydzielonych wątków, spowodowane jest to chociażby koniecznością przydzielania operacji do wątku co jest wymagającym zadaniem czasowo dla sprzętu. Największe przyśpieszenie obserwujemy w zakresie od 1 do 8 wątków dla małego zestawu danych , oznacza to że później nie dostajemy już tak znaczącego przyśpieszenia, natomiast duży zestaw lepiej skaluje się dla większej ilości wątków, natomiast warto jest wykorzystać nadal większą ilość wątków dla zwiększenia efektywności obliczeń dla obu zestawów.