Alfa - Macierze

Programowanie Współbieżne

„Programowanie współbieżne to jak równoległe gotowanie w kuchni - czasem musisz obsługiwać kilka garnków naraz, ale jeśli nie uważasz, to łatwo możesz spalić zupę i utracić równowagę w życiu."

Stworzone przez

Michał Miziołek

Kraków
2023

Spis treści

Rozdział 1

Wstęp

1.1 Motyw

Głównym celem tego projektu jest zoptymalizowanie zadania Alfa z Podstaw Programowania oraz zaimplementowanie zrówloeglnych wersji za pomocą openMP oraz threads.

1.2 Treść

Jesteś informatykiem pracującym w firmie Bajtomania. Obecne Twoje zlecenie złożył szalony profesor Bonifacy Malkontent. Zajmuje się on astronomią i wykonuje całą masę różnych obliczeń (właściwie nie wiadomo po co...). Naukowiec potrzebuje programu do wykonywania operacji na macierzach.

Operacje do zaimplementowania zdefiniowane są następująco:

- **ADD** - dodawanie dwóch macierzy
- **MULTIPLY** - mnożenie dwóch macierzy
- **POWER** p - podnoszenie macierzy do potęgi p
- **DETERMINANT** - obliczenie wyznacznika macierzy

Dla szybszego działania programu podczas potęgowania macierzy należy zastosować algorytm szybkiego potęgowania.

1.3 disclaimers

W owym projekcie będziemy mierzyć i porównywać czasy działania poszczególnych funkcji. Tak więc, będziemy mieli wgląd jak wprowadzenie współbieżności wpływa na operacje na macierzach z pominięciem czasu wczytywania i wypisywania.

Algorytmy były przeprowadzane na różnych ilościach wątków - (1, 4, 8, 16, 32, 64), a wszystkie testy były uruchamiane na studencie

1.4 Linki

github

Rozdział 2

Sumowanie

2.1 Implementacja Sekwencyjna

2.1.1 Opis algorytmu

Najbardziej standardowy sposób dodawania dwóch macierzy, w której iterujemy się kolejno po wierszach i dodajemy wartości z jednej macierzy do drugiej.

W przyjętej implementacji dodajemy macierz B do A, aby nie tworzyć nowej macierzy (usprawnienie).

2.1.2 Pseudokod

```
bool operator+(Matrix& A, const Matrix& B) {
    if (A.width != B.width || A.height != B.height) return false;

    for (int i = 0; i < A.height; ++i) {
        for (int j = 0; j < A.width; ++j) {
            A.matrix[i][j] += B.matrix[i][j];
        }
    }

    return true;
}
```

Listing 2.1: Dodawanie macierzy

2.1.3 Analiza złożoności obliczeniowej

Jak widać dodawanie jest liniowe względem input'u i nie jesteśmy w stanie tego przyspieszyć. Dlatego najlepszym rozwiązaniem jest O(n*m), gdzie n i m są odpowiednio szerokością i długością macierzy.

2.2 Implementacja Współbieżna

2.2.1 Opis algorytmu

Każde z pól macierzy wynikowej jest otrzymywane jako suma unikalnych pól macierzy wejściowych. Dlatego możemy łatwo zrównoleglić dodawanie poprzez dodawanie każdego z wieszy w różnych wątkach.

2.2.2 Pseudokod

2.2.2.1 OpenMP

```
bool operator+(Matrix& A, const Matrix& B) {
    if (A.width != B.width || A.height != B.height) return false;

    #pragma omp parallel for
    for (int i = 0; i < A.height; ++i) {
        for (int j = 0; j < A.width; ++j) {
            A.matrix[i][j] += B.matrix[i][j];
        }
    }

    return true;
}
```

Listing 2.2: Dodawanie macierzy - openMP

2.2.2.2 Threads

```
bool operator+(Matrix& A, const Matrix& B) {
    if (A.width != B.width || A.height != B.height) return false;

    std::vector<std::thread> threads(N_THREADS);

    for (int t = 0; t < N_THREADS; ++t) {
        threads[t] = std::thread([&](int startRow) {
            for (int i = startRow; i < A.height; i += N_THREADS) {
                for (int j = 0; j < A.width; ++j) {
                    A.matrix[i][j] += B.matrix[i][j];
                }
            }
        }, t);
    }

    for (auto& th : threads) {
        if (th.joinable()) th.join();
    }

    return true;
}
```

Listing 2.3: Dodawanie macierzy - threads

2.2.3 Analiza złożoności obliczeniowej

O(n*m), jednak możemy zbić stałą do teoretycznych p razy, gdzie p jest ilością wątków.

2.3 Porównanie

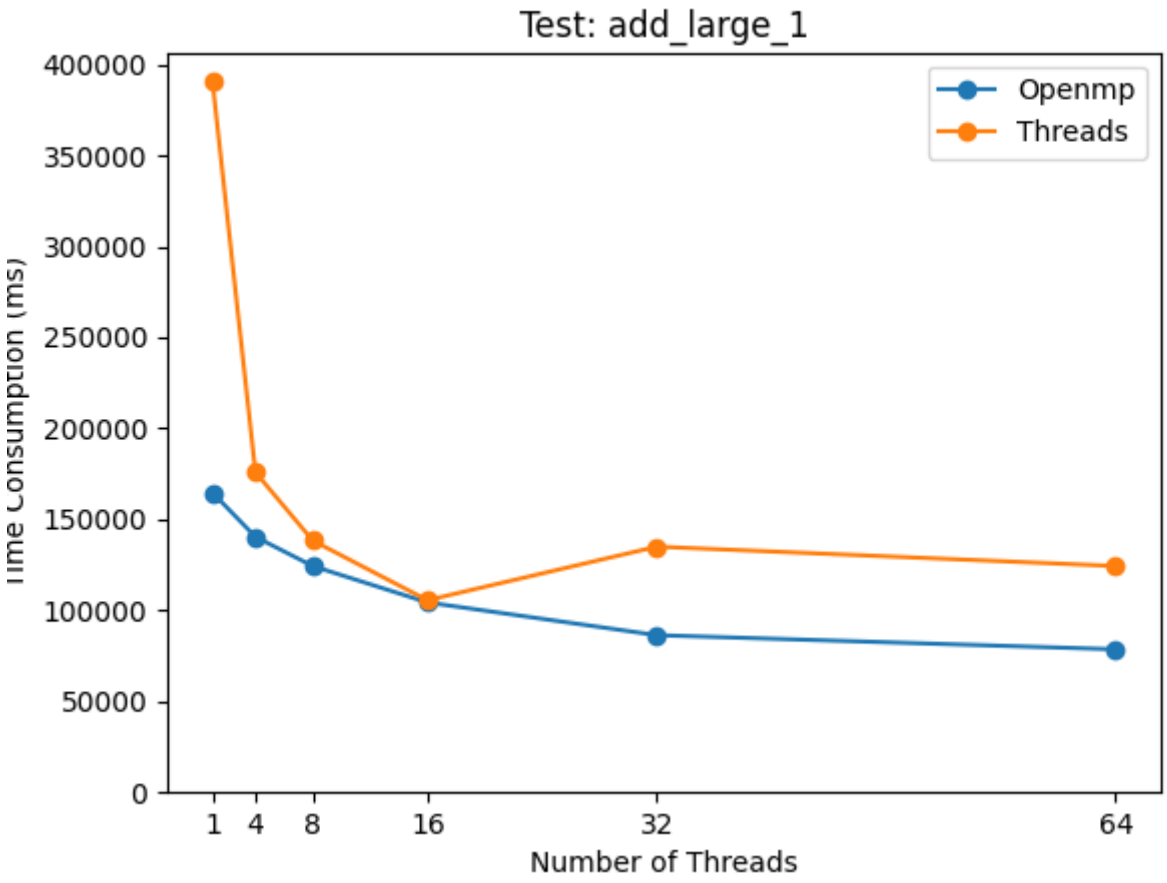
2.3.1 Obserwacje

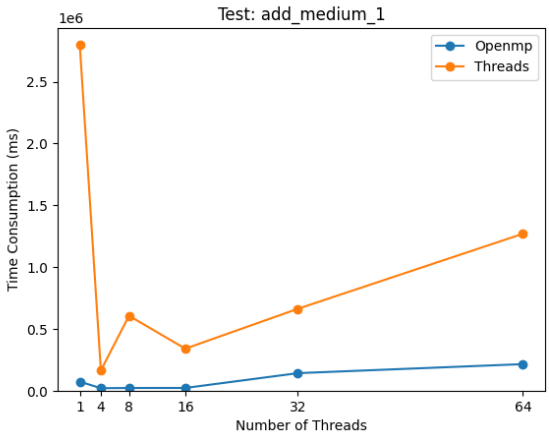
Jak już ustaliliśmy złożoność dodawania macierzy jest liniowa względem wczytywania danych, tak więc współbieżna implementacja algorytmu nie wypłynie aż tak bardzo na całkowity wynik. Dlatego też będziemy mierzyć długość działania po wywołaniu metody, a nie całkowite działanie programu.

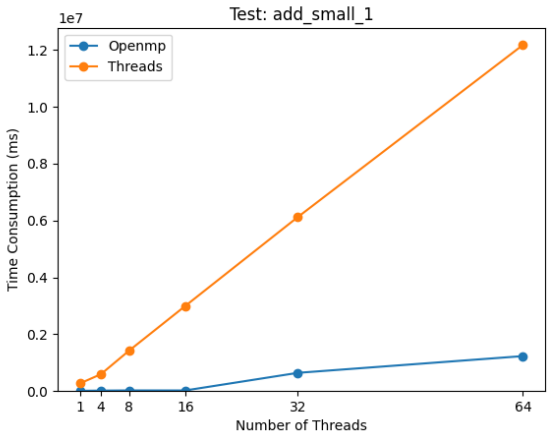
Jak widać, dla dużych testów otrzymujemy przyspieszenie rzędu dwukrotności, lecz dla mniejszych macierzy (rzędu n=100) otrzymujemy delitakne przyspieszenie dla kilku wątków, jednak przy większej ilości wywoływanie wątków i działanie na nich jest znacznie dłuższe niż dla pojedynczego wątku.

Owy problem wynika z narzutu tworzenia wątków.

2.3.2 Wykresy




Test: add_medium_1
1e6
Openmp
Threads
Time Consumption (ms)
Number of Threads
0.0
0.5
1.0
1.5
2.0
2.5
1
4
8
16
32
64


Test: add_small_1
1e7
Openmp
Threads
Time Consumption (ms)
Number of Threads
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1
4
8
16
32
64

Rozdział 3

Mnożenie

3.1 Implementacja Sekwencyjna

3.1.1 Opis algorytmu

Iterujemy się kolejno po rzędach i kolumnach macierzy i mnożymy unitarnie każdy z pozycji. Tradycyjne mnożenie wynikające z definicji mnożenia:

$$\mathrm{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad \mathrm{B} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{bmatrix}$$

$$\mathrm{AB} = \begin{bmatrix} \sum_{k=1}^{n} a_{1k}b_{k1} & \sum_{k=1}^{n} a_{1k}b_{k2} & \cdots & \sum_{k=1}^{n} a_{1k}b_{kp} \\ \sum_{k=1}^{n} a_{2k}b_{k1} & \sum_{k=1}^{n} a_{2k}b_{k2} & \cdots & \sum_{k=1}^{n} a_{2k}b_{kp} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{k=1}^{n} a_{mk}b_{k1} & \sum_{k=1}^{n} a_{mk}b_{k2} & \cdots & \sum_{k=1}^{n} a_{mk}b_{kp} \end{bmatrix}$$

3.1.2 Pseudokod

```
bool operator*(Matrix& A, const Matrix& B) {
    if (A.width != B.height) return false;

    Matrix result;
    result.width = B.width;
    result.height = A.height;
    result.matrix.resize(result.height,
       std::vector<int>(result.width, 0));

    for (int i = 0; i < A.height; ++i) {
        for (int j = 0; j < B.width; ++j) {
            int sum = 0;
            for (int k = 0; k < A.width; ++k) {
                sum += A.matrix[i][k] * B.matrix[k][j];
            }
            result.matrix[i][j] = sum;
        }
    }
```

```
18
19      A = result;
20
21      return true;
22  }
```

Listing 3.1: Mnożenie macierzy

3.1.3 Analiza złożoności obliczeniowej

Mnożenie ma złożoność O(n^3), gdzie n jest dłuższym bokiem macierzy.

3.2 Implementacja Współbieżna

3.2.1 Opis algorytmu

Zrównoleglamy zewnętrzną pętlę, gdyż każde pole wynikowe możemy obliczać niezależnie od innych pól. Pomimo tego, że odczyt może być zależny od tych samych pól macierzy wynikowych to jako iż nie dokonujemy modyfikacji możemy robić to równolegle.

3.2.2 Pseudokod

3.2.2.1 OpenMP

```
1  bool operator*(Matrix& A, const Matrix& B) {
2      if (A.width != B.height) return false;
3
4      Matrix result;
5      result.width = B.width;
6      result.height = A.height;
7      result.matrix.resize(result.height,
           std::vector<int>(result.width, 0));
8
9      #pragma omp parallel for
10     for (int i = 0; i < A.height; ++i) {
11         for (int j = 0; j < B.width; ++j) {
12             int sum = 0;
13             #pragma omp parallel for reduction(+:sum)
14             for (int k = 0; k < A.width; ++k) {
15                 sum += A.matrix[i][k] * B.matrix[k][j];
16             }
17             result.matrix[i][j] = sum;
18         }
19     }
20
21     A = result;
22
23     return true;
24 }
```

Listing 3.2: Mnożenie macierzy - openMP

3.2.2.2 Threads

```cpp
bool operator*(Matrix& A, const Matrix& B) {
    if (A.width != B.height) return false;

    Matrix result;
    result.width = B.width;
    result.height = A.height;
    result.matrix.resize(result.height,
       std::vector<int>(result.width, 0));

    std::vector<std::thread> threads(N_THREADS);

    for (int t = 0; t < N_THREADS; ++t) {
        threads[t] = std::thread([&](int startRow) {
            for (int i = startRow; i < A.height; i += N_THREADS) {
                for (int j = 0; j < B.width; ++j) {
                    int sum = 0;
                    for (int k = 0; k < A.width; ++k) {
                        sum += A.matrix[i][k] * B.matrix[k][j];
                    }
                    result.matrix[i][j] = sum;
                }
            }
        }, t);
    }

    for (auto& th : threads) {
        if (th.joinable()) th.join();
    }

    A = result;

    return true;
}
```

Listing 3.3: Mnożenie macierzy - threads

3.2.3 Analiza złożoności obliczeniowej

O(n^3), jednak możemy zbić stałą do teoretycznych p razy, gdyż zrównolegliliśmy zewnętrzną pętlę, gdzie p jest ilością wątków.

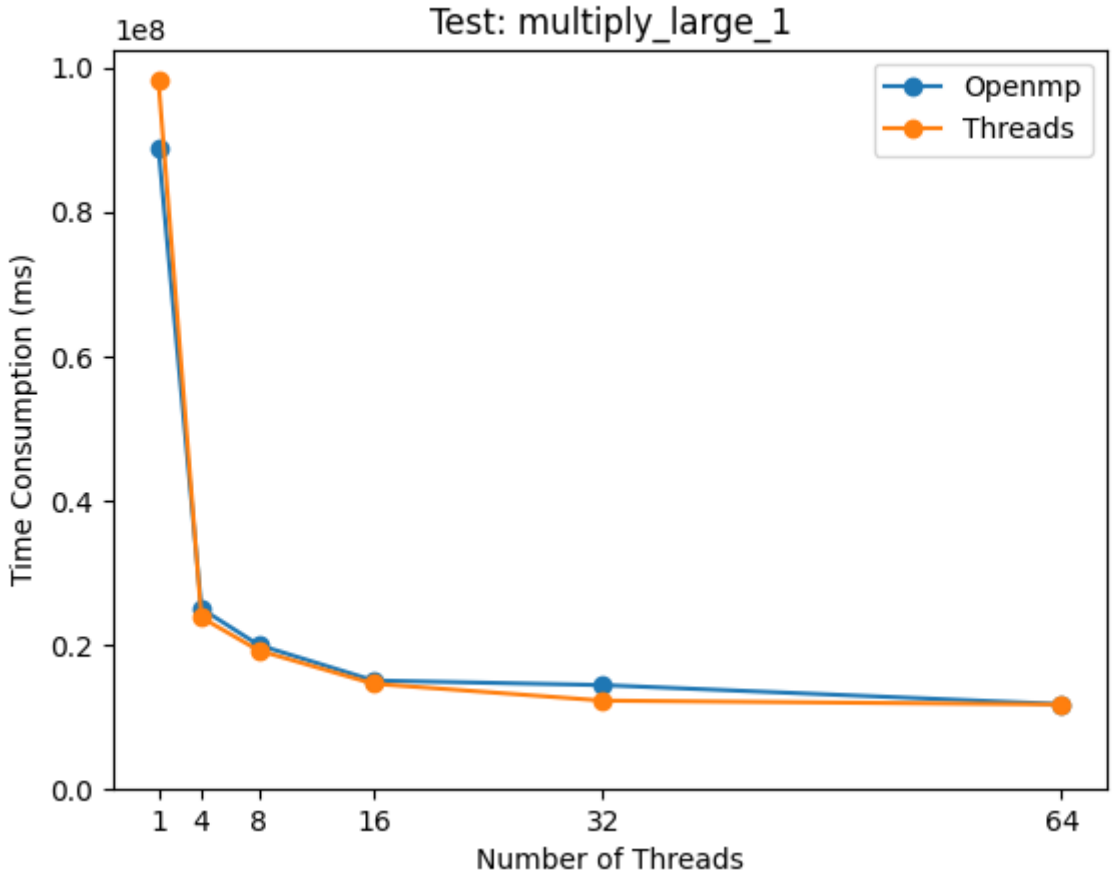
3.3 Porównanie

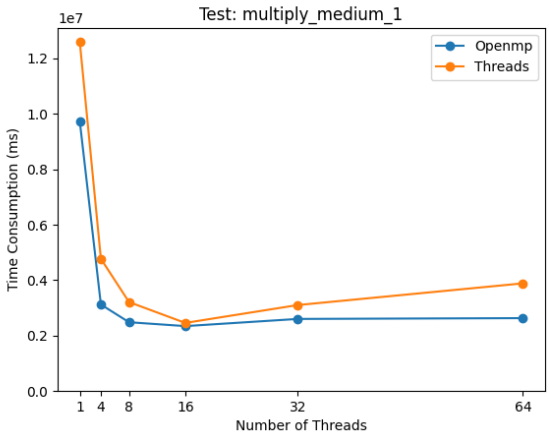
3.3.1 Obserwacje

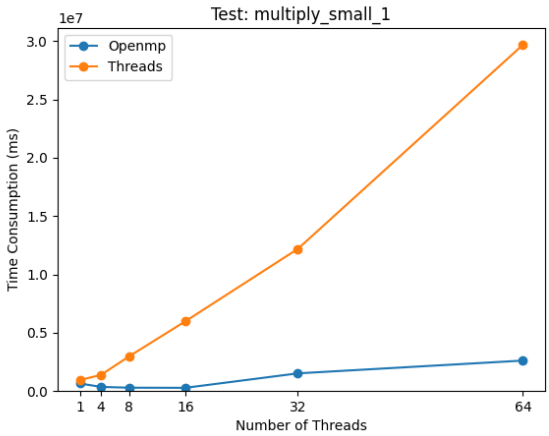
Jak widać, dla dużych testów otrzymujemy przyspieszenie rzędu pięciokrotności, lecz dla mniejszych macierzy (rzędu 100 lub mniej) otrzymujemy delikatne przyspieszenie dla kilku wątków, jednak przy większej ilości wywoływanie wątków i działanie na nich jest znacznie dłuższe niż dla pojedynczego wątku.

Również wynika to z faktu narzutu tworzenia o obsługi wątków.

3.3.2 Wykresy


Test: multiply_large_1
1e8
1.0
0.8
0.6
0.4
0.2
0.0
Time Consumption (ms)
Openmp
Threads
1
4
8
16
32
64
Number of Threads


Test: multiply_medium_1
1e7
Openmp
Threads
Time Consumption (ms)
Number of Threads
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1
4
8
16
32
64


Test: multiply_small_1
1e7
Openmp
Threads
Time Consumption (ms)
Number of Threads
0.0
0.5
1.0
1.5
2.0
2.5
3.0
1
4
8
16
32
64

Rozdział 4

Potęgowanie

4.1 Implementacja Sekwencyjna

4.1.1 Opis algorytmu

W poniższej implementacji wykorzystałem algorytm szybkiego potęgowania.

Założeniem algorytmu szybkiego potęgowania jest zredukowanie liczby mnożeń macierzy poprzez wykorzystanie własności potęgowania. Algorytm można opisać następująco:

1. Inicjalizacja macierzy wynikowej R jako macierzy jednostkowej o tych samych wymiarach co macierz A.
2. Dla każdej iteracji, sprawdzamy, czy aktualna wartość wykładnika p jest nieparzysta:
   - Jeśli $p \mod 2 = 1$, mnożymy macierz wynikową R przez macierz A.
3. Następnie, kwadratujemy macierz A (tj. $A = A \times A$) i dzielimy wykładnik p przez 2.
4. Powtarzamy kroki 2 i 3, dopóki p jest większe od 0.
5. Wynik znajduje się w macierzy R.

Algorytm wykorzystuje podejście rekurencyjne, gdzie potęgowanie macierzy A do wykładnika p można przedstawić jako:

$$A^p = \begin{cases} R \times A, & \text{jeśli } p \mod 2 = 1 \\ R, & \text{w przeciwnym przypadku} \end{cases}$$

gdzie R to macierz wynikowa aktualizowana w każdej iteracji algorytmu.

4.1.2 Pseudokod

```
bool operator^(Matrix& A, long long p) {
    if (A.width != A.height) {
        return false;
    }

    Matrix result;
    result.set_identity(A.width);

```

```
9       while (p > 0) {
10          if (p % 2 == 1) {
11              result * A;
12          }
13
14          A * A;
15          p /= 2;
16      }
17
18      A = result;
19      return true;
20  }
```

Listing 4.1: Potęgowanie macierzy

4.1.3 Analiza złożoności obliczeniowej

Złożoność czasowa algorytmu szybkiego potęgowania macierzy jest logarytmiczna względem wykładnika - $log(p)$.

4.2 Implementacja Współbieżna

4.2.1 Brak

4.2.1.1 Opis algorytmu

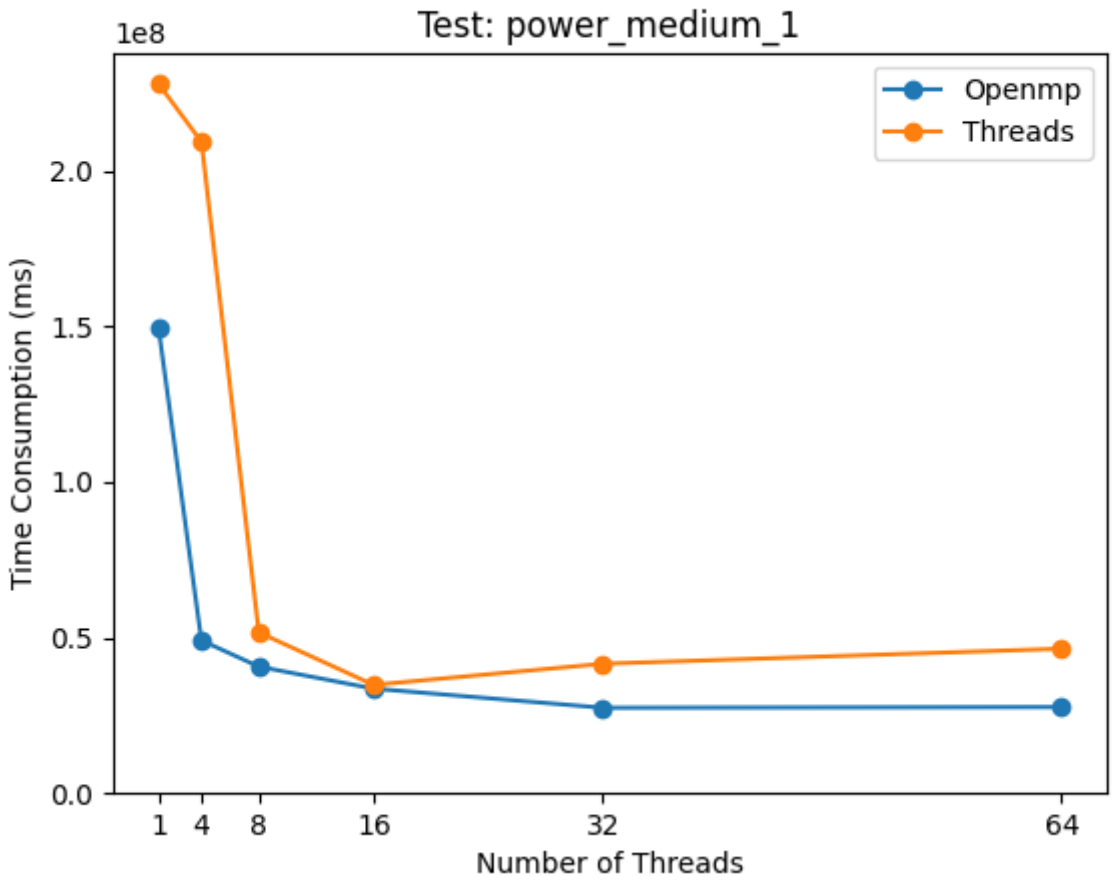
Jak już zauważyliśmy algorytm szybkiego potęgowania jest logarytmiczny, dlatego jego optymalizacja wielowątkowa byłaby zbędna i tylko spowolniłaby działanie. Dlatego optymalizujemy jedynie mnożenie.

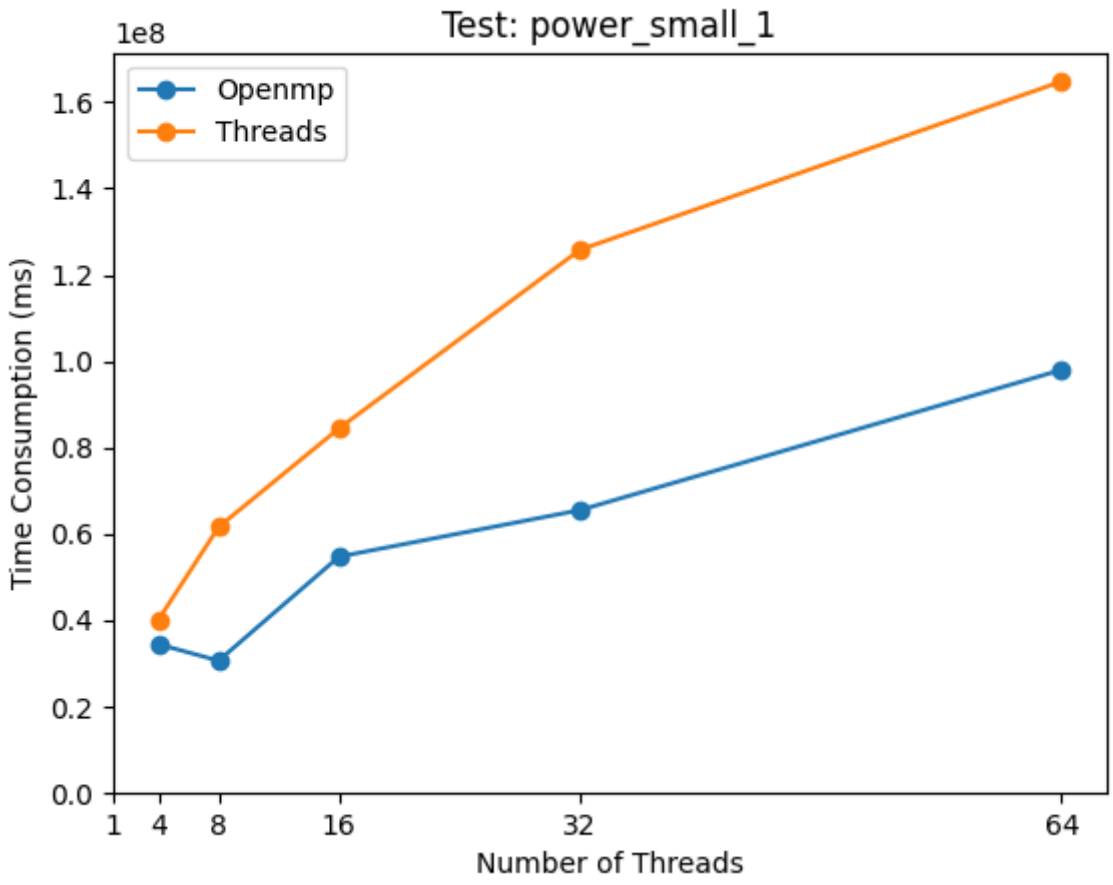
4.2.1.2 Obserwacje

Dla testu, który ma potęgę rzędu 2^{10}, głównym ciężarem jest obliczanie iloczynu macierzy, tak więc wykres będzie przypominać zależność jak w przypadku analizy mnożenia macierzy (zobacz pierwszy wykres).

Dla testu, w którym jest wiele małych macierzy lecz o sporym wykładniku rzędu 2^{63} oczywiście nie zachodzi żadna optymalizacja, a mnożenie macierzy można traktować jak sporą stałą. Dlatego obie implementacje wraz ze wzrostem ilości wątków tylko spowalniają działanie programu ze względu na narzut związany z obsługą wątków (zobasz drugi wykres).

4.2.1.3 Wykresy


Test: power_medium_1
1e8
Openmp
Threads
2.0
1.5
1.0
0.5
0.0
Time Consumption (ms)
1
4
8
16
32
64
Number of Threads


Test: power_small_1
1e8
Openmp
Threads
Time Consumption (ms)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
1
4
8
16
32
64
Number of Threads

4.2.2 Rekurencja

4.2.2.1 Opis algorytmu

Aby upewnić, że na pewno nie opłaca się optymalizować szybkiego potęgowania zaimplementowałem wersję rekurencyjną i ją zrównolegliłem.

Algorytm szybkiego potęgowania macierzy wykorzystuje rekurencję do podziału problemu na mniejsze części. Oto podstawowe kroki algorytmu:

1. Jeśli wykładnik *exp* jest równy 0, ustawiamy macierz wynikową *result* jako macierz jednostkową rozmiaru *size* i kończymy działanie funkcji.
2. W przeciwnym przypadku sprawdzamy, czy *exp* jest nieparzyste. Jeśli tak, obniżamy wykładnik o 1 i wywołujemy funkcję rekurencyjnie, a następnie mnożymy wynik przez macierz bazową *base*.
3. Jeśli *exp* jest parzyste, dzielimy wykładnik przez 2 i również wywołujemy funkcję rekurencyjnie, a następnie kwadratujemy uzyskany wynik (mnożymy go sam przez siebie).
4. Wynik jest zapisywany w macierzy *result*.

Algorytm można przedstawić w formie wzoru rekurencyjnego:

$$\text{matrix\_power}(A,p) = \begin{cases} I, & \text{jeśli } p = 0 \\ \text{matrix\_power}(A, p-1) \times A, & \text{jeśli } p \text{ nieparzyste} \\ B \times B, & \text{gdzie } B = \text{matrix\_power}(A, p/2), \text{jeśli } p \text{ parzyste} \end{cases}$$

gdzie:

- A jest macierzą bazową,
- p jest wykładnikiem potęgi,
- I jest macierzą jednostkową.

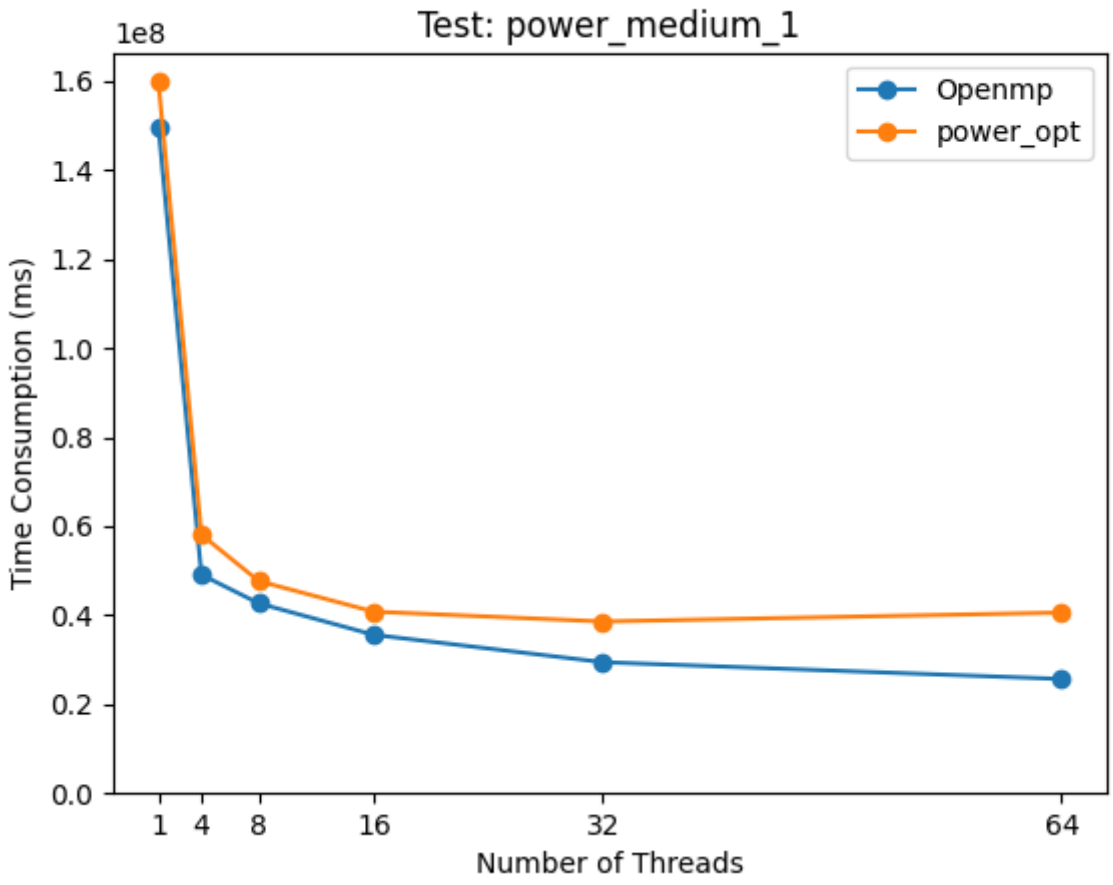
4.2.3 Pseudokod

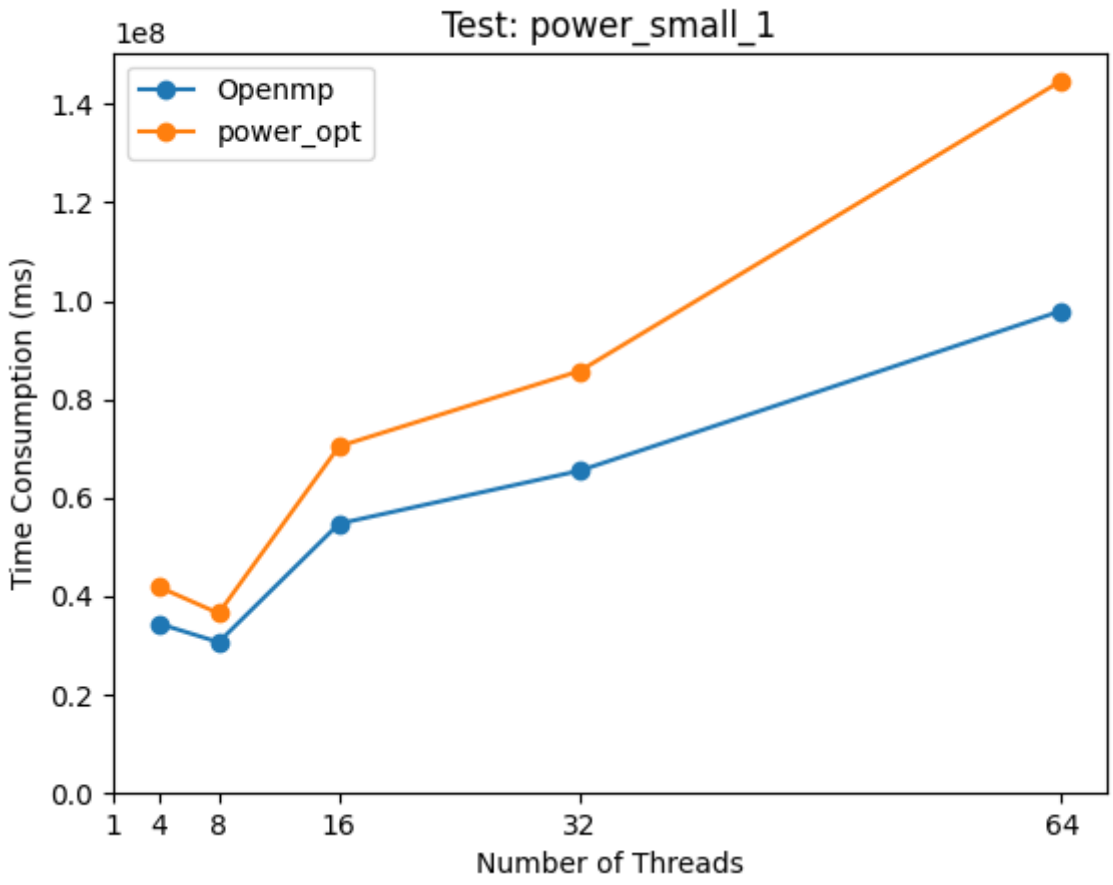
```
1  void matrix_power_recursive(const Matrix& base, long long exp, int
       size, Matrix& result) {
2      if (exp == 0) {
3          result.set_identity(size);
4          return;
5      }
6
7      Matrix temp_result;
8
9      if (exp % 2 == 1) {
10         #pragma omp task shared(base, temp_result) if(exp > 1)
11         matrix_power_recursive(base, exp - 1, size, temp_result);
12
13         #pragma omp taskwait
14         temp_result * base;
15     } else {
16         #pragma omp task shared(base, temp_result) if(exp > 1)
```

```
17          matrix_power_recursive(base, exp / 2, size, temp_result);
18
19          #pragma omp taskwait
20          temp_result * temp_result;
21      }
22      result = temp_result;
23  }
24
25  bool operator^(Matrix& A, long long p) {
26      if (A.width != A.height) {
27          return false;
28      }
29
30      Matrix result;
31      result.set_identity(A.width);
32
33      matrix_power_recursive(A, p, A.width, result);
34
35      A = result;  // Update the current instance with the result
36      return true;
37  }
```

Listing 4.2: Potęgowanie macierzy

4.2.3.1 Wykresy


Test: power_medium_1
1e8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Time Consumption (ms)
Openmp
power_opt
1
4
8
16
32
64
Number of Threads


Test: power_small_1
1e8
Openmp
power_opt
Time Consumption (ms)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1
4
8
16
32
64
Number of Threads

Rozdział 5

Wyznacznik

5.1 Implementacja Sekwencyjna

5.1.1 Opis algorytmu

Aby obliczyć wyznacznik, zdecydowałem się użyć dekompozycji LU, która dokonuje rozkładu macierzy na dwa czynniki:

$$A = LU$$

gdzie:

- L jest macierzą dolnotrójkątną (wszystkie elementy powyżej głównej przekątnej są zerowe),
- U jest macierzą górnotrójkątną (wszystkie elementy poniżej głównej przekątnej są zerowe).

Wyznacznik macierzy A można obliczyć jako produkt wyznaczników macierzy L i U:

$$\det(A) = \det(L) \times \det(U)$$

W moim przypadku L na przekątnej posiada same jedynki, więc pozostaje obliczyć iloczyn liczb na przekątnej macierzy U.

Załóżmy, że mamy macierz kwadratową A rozmiaru $n \times n$. Dekompozycja LU tej macierzy przebiega w następujących krokach:

1. Inicjalizacja macierzy L i U. Początkowo, L jest macierzą jednostkową (z jedynkami na głównej przekątnej), a U jest kopią macierzy A.
2. Dla każdego wiersza i od 1 do n, wykonaj następujące kroki:
   (a) Dla każdej kolumny j od i do n, oblicz elementy macierzy U w następujący sposób:

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} \times u_{kj}$$

(b) Dla każdej kolumny j od $i+1$ do n, oblicz elementy macierzy L w następujący sposób:

$$l_{ji} = \frac{1}{u_{ii}} \left(a_{ji} - \sum_{k=1}^{i-1} l_{jk} \times u_{ki} \right)$$

3. Po zakończeniu powyższych kroków, macierze L i U stanowią dekompozycję LU macierzy A.

Algorytm dekompozycji LU jest szczególnie efektywny dla macierzy gęstych i dużych rozmiarów. Pozwala on na uniknięcie bezpośredniego obliczania wyznaczników i inwersji macierzy, co jest operacjami obliczeniowo kosztownymi.

5.1.2 Pseudokod

```
void Matrix::luDecomposition(std::vector<std::vector<float>>& L,
                             std::vector<std::vector<float>>& U) const {
    int n = matrix.size();

    L.resize(n, std::vector<float>(n, 0));
    U.resize(n, std::vector<float>(n, 0));

    for (int i = 0; i < n; ++i) {
        L[i][i] = 1.0;  // Diagonal of L is 1
        for (int j = i; j < n; ++j) {
            U[i][j] = matrix[i][j];

            float sum = 0;
            for (int k = 0; k < i; ++k) {
                sum -= L[i][k] * U[k][j];
            }
            U[i][j] -= sum;
        }
        for (int j = i + 1; j < n; ++j) {
            L[j][i] = matrix[j][i];

            float sum = 0;
            for (int k = 0; k < i; ++k) {
                sum -= L[j][k] * U[k][i];
            }
            L[j][i] -= sum;

            L[j][i] /= U[i][i];
        }
    }
}

long long Matrix::determinant() {
    if (width != height) {
        return LONG_LONG_MIN;
    }

    std::vector<std::vector<float>> L, U;
    luDecomposition(L, U);
```

```
40
41      double det = 1.0;
42      for (int i = 0; i < width; ++i) {
43          det *= U[i][i];
44      }
45
46      return std::round(det);
47  }
```

Listing 5.1: Obliczanie wyznacznika

5.1.3 Analiza złożoności obliczeniowej

Złożoność czasowa dekompozycji LU jest rzędu $O(n^3)$. Wynika to z faktu, że dla każdego z n wierszy macierzy, algorytm wykonuje serię operacji na pozostałych $n - i$ elementach.

5.2 Implementacja Współbieżna

5.2.1 Opis algorytmu

Zrównoleglamy pętle wewnętrzne odpowiedzialne za obliczanie kolejnych wierszy (ścinanie do przekątnej).

5.2.2 Pseudokod

5.2.2.1 OpenMP

```
1  void Matrix::luDecomposition(std::vector<std::vector<float>>& L,
2                               std::vector<std::vector<float>>& U) const {
3      int n = matrix.size();
4
5      L.resize(n, std::vector<float>(n, 0));
6      U.resize(n, std::vector<float>(n, 0));
7
8      #pragma omp parallel for
9      for (int i = 0; i < n; ++i) {
10         L[i][i] = 1.0;  // Diagonal of L is 1
11
12         #pragma omp parallel for
13         for (int j = i; j < n; ++j) {
14             U[i][j] = matrix[i][j];
15
16             float sum = 0;
17             #pragma omp parallel for reduction(-:sum)
18             for (int k = 0; k < i; ++k) {
19                 sum -= L[i][k] * U[k][j];
20             }
21             U[i][j] -= sum;
22         }
23
24         #pragma omp parallel for
25         for (int j = i + 1; j < n; ++j) {
```

```
26              L[j][i] = matrix[j][i];
27
28
29              #pragma omp parallel for reduction(-:sum)
30              float sum = 0;
31              for (int k = 0; k < i; ++k) {
32                  sum -= L[j][k] * U[k][i];
33              }
34              L[j][i] -= sum;
35
36              L[j][i] /= U[i][i];
37          }
38      }
39  }
40
41  long long Matrix::determinant() {
42      if (width != height) {
43          return LONG_LONG_MIN;
44      }
45
46      std::vector<std::vector<float>> L, U;
47      luDecomposition(L, U);
48
49      double det = 1.0;
50      #pragma omp parallel for reduction(*:det)
51      for (int i = 0; i < width; ++i) {
52          det *= U[i][i];
53      }
54
55      return std::round(det);
56  }
```

Listing 5.2: Obliczanie wyznacznika - openMP

5.2.2.2 Threads

```
1  void Matrix::luDecomposition(std::vector<std::vector<float>>& L,
       std::vector<std::vector<float>>& U) const {
2      int n = matrix.size();
3      int n_threads = std::min(N_THREADS, n);
4      L.resize(n, std::vector<float>(n, 0));
5      U.resize(n, std::vector<float>(n, 0));
6
7      for (int i = 0; i < n; ++i) {
8          L[i][i] = 1.0;
9          std::vector<std::thread> threads(n_threads);
10
11         // First loop
12         for (int t = 0; t < n_threads; ++t) {
13             threads[t] = std::thread([&, t, i, n]() {
14                 for (int j = i + t; j < n; j += n_threads) {
15                     U[i][j] = matrix[i][j];
16                     float sum = 0;
17                     for (int k = 0; k < i; ++k) {
```

```
                        sum -= L[i][k] * U[k][j];
                    }
                    U[i][j] -= sum;
                }
            });
        }

        for (auto& th : threads) {
            if (th.joinable()) th.join();
        }

        // Second loop
        for (int t = 0; t < n_threads; ++t) {
            threads[t] = std::thread([&, t, i, n]() {
                for (int j = i + 1 + t; j < n; j += n_threads) {
                    L[j][i] = matrix[j][i];
                    float sum = 0;
                    for (int k = 0; k < i; ++k) {
                        sum -= L[j][k] * U[k][i];
                    }
                    L[j][i] -= sum;

                    L[j][i] /= U[i][i];
                }
            });
        }

        for (auto& th : threads) {
            if (th.joinable()) th.join();
        }
    }
}

long long Matrix::determinant() {
    if (width != height) {
        return LONG_LONG_MIN;
    }

    std::vector<std::vector<float>> L, U;
    luDecomposition(L, U);

    double det = 1.0;
    for (int i = 0; i < width; ++i) {
        det *= U[i][i];
    }

    return std::round(det);
}
```

Listing 5.3: Obliczanie wyznacznika - threads

5.2.3 Analiza złożoności obliczeniowej

O(n^3), jednak możemy zbić stałą do p razy, gdyż zrównolegliliśmy wewnętrzne pętle.

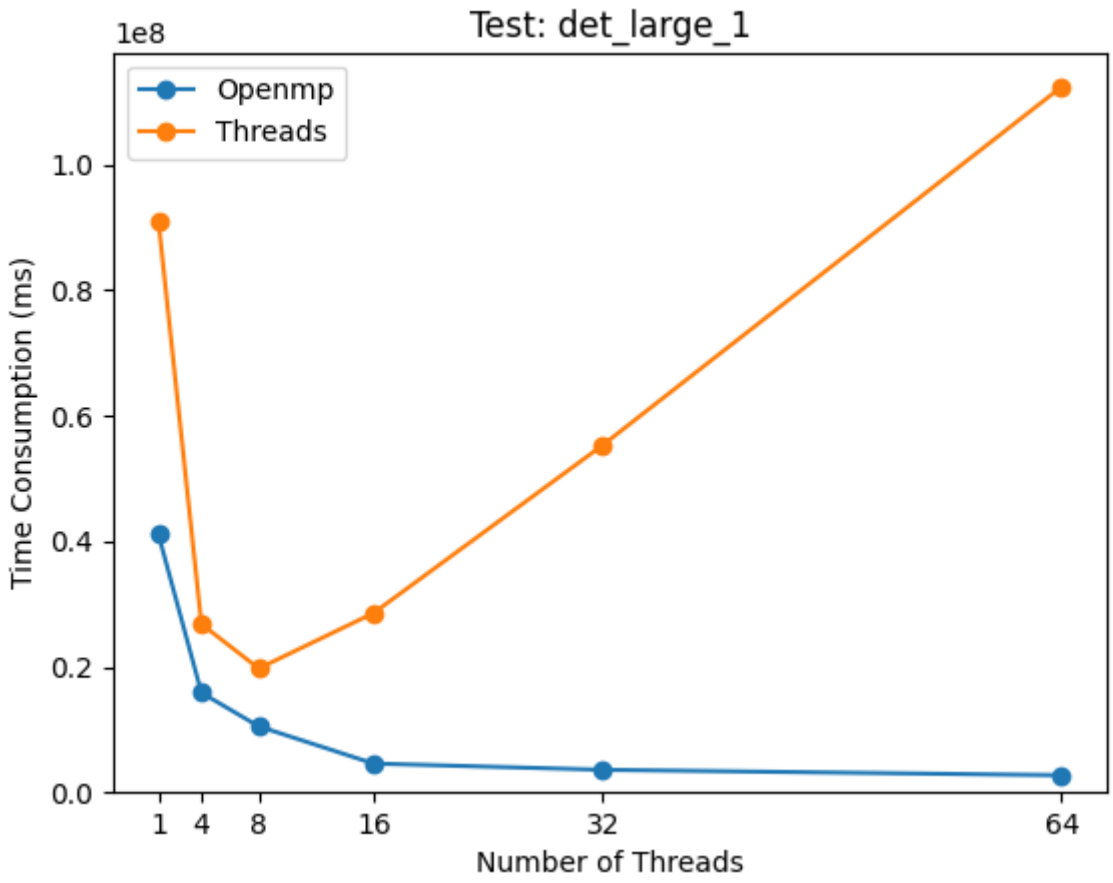
5.3 Porównanie

5.3.1 Obserwacje

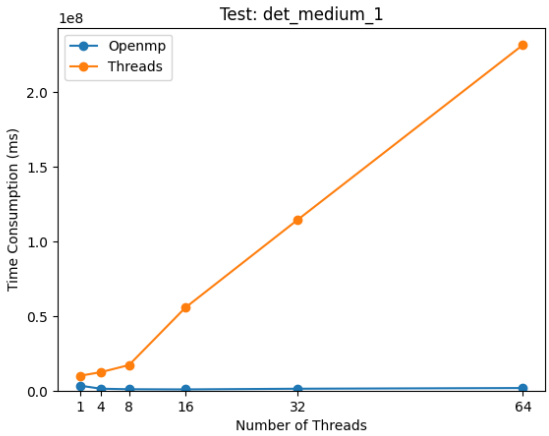
W teorii wykres powinien wyglądać podobnie do mnożenia macierzy, jednak jak widać na załączonym obrazku dla większniej ilośći wątków implementacja za pomocą threads daje niepoprawny wynik. Niestety ze względu na obciążenie studenta, musiałem dokonać testu dla determinant na miracle, na którym jest dostępne 16 wątków, dlatego przypuszczam, że odchylenie wynika z przymusu działania na wątkach systemowych, który wymaga kosztownych kernel-swtitch'y.

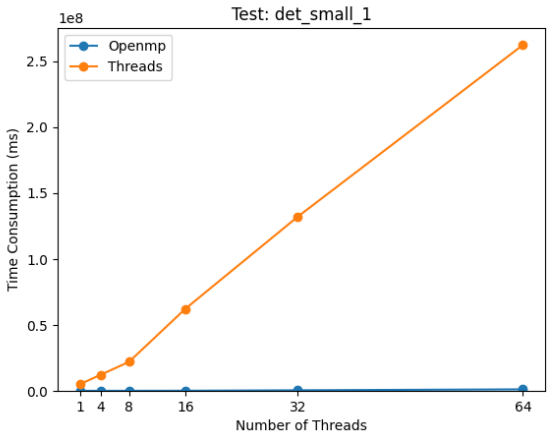
Jednak jak widać openMP nie miał z tym większych problemów, a wykres przypomina krzywą zakreślaną jak przy mnożeniu macierzy, czy potęgowaniu.

5.3.2 Wykresy



5.3.3 Wnioski


Test: det_medium_1
1e8
Openmp
Threads
Time Consumption (ms)
0.0
0.5
1.0
1.5
2.0
1
4
8
16
32
64
Number of Threads


Test: det_small_1
1e8
Openmp
Threads
Time Consumption (ms)
0.0
0.5
1.0
1.5
2.0
2.5
1
4
8
16
32
64
Number of Threads