POLITECHNIKA KRAKOWSKA
im. T. Kościuszki
Wydział Informatyki i Telekomunikacji

Kierunek studiów: Informatyka
Specjalność: Data Science

PROGRAMOWANIE RÓWNOLEGŁE I ROZPROSZONE

Sprawozdanie z projektu

Obliczanie średniej arytmetycznej liczb zmiennoprzecinkowych
z zadanego przedziału

Maria Guz
Paweł Midura

Spis Treści

1 Wprowadzenie **3**
1.1 Opis problemu . . . 3
1.2 Parametry maszyny . . . 3
1.3 OpenMP . . . 4
1.4 MPI . . . 4
1.5 Podejście hybrydowe . . . 4

2 Implementacja **4**
2.1 Generacja danych . . . 4
2.2 Bez zrównoleglenia . . . 5
2.3 OpenMP . . . 7
2.4 MPI . . . 8
2.5 OpenMP + MPI . . . 10

3 Porównanie **13**
3.1 Wykresy . . . 13
3.1.1 Zależności czasu od ilości danych . . . 13
3.1.2 Zależności czasu od liczby wątków . . . 13
3.1.3 Wykresy przyspieszenia . . . 16
3.2 Wnioski . . . 18

1 Wprowadzenie

1.1 Opis problemu

Celem projektu jest porównanie czasu obliczeń potrzebnych do wyliczenia średniej arytmetycznej liczb z zadanego przedziału wartości w zbiorze liczb zmiennoprzecinkowych. Program ma mieć możliwość zmiany ilości wątków, na których jest wykonywany (n). Porównanie czasu obliczeń dotyczy następujących implementacji:

- sekwencyjna (bez zrównoleglenia)
- z użyciem interfejsu *OpenMP*
- z użyciem interefejsu *MPI*
- hybrydowa - *OpenMP* + *MPI*

Wyniki porównania zostaną przedstawione na wykresach przyspieszenia[1].

1.2 Parametry maszyny

System operacyjny

Wersja	Windows 10 Education
Typ	64-bitowy

Procesor

Rodzina	AMD Ryzen
Seria	Ryzen 3 3100
Liczba rdzeni fizycznych	4
Liczba wątków	8
Taktowanie rdzenia	3.9 GHz

Pamięć operacyjna

Ilość pamięci RAM	16 GB
Szybkość	2666 MHz

[1] wykresy przyspieszenia służą do porównania czasu, który potrzebny jest do wykonania obliczeń programu napisanego sekwencyjnie do czasu obliczeń wykonywanych równolegle na n procesorach. Na osi poziomej jest zatem liczba procesorów (p), a na osi pionowej jest przyspieszenie - iloraz czasu obliczeń sekwencyjnych do czasu wykonywania obliczeń równoległych na n procesorach

1.3 OpenMP

OpenMP (ang. *Open Multi-Processing*) jest wieloplatformowym interfejsem programowania aplikacji. Umożliwia tworzenie programów komputerowych dla wieloprocesorowych systemów z pamięcią dzieloną. Można go wykorzystywać m.in. w języku C++. Jest dostosowany do wielu architektur (m.in. Unix i Microsoft Windows). Składa się ze zbioru dyrektyw kompilatora, bibliotek oraz zmiennych środowiskowych, które wpływają na sposób wykonywania się programu. Charakterystycznymi cechami OpenMP są przenośność, skalowalność, elastyczność i prostota użycia[2].

1.4 MPI

MPI (ang. *Message Passing Interface*), tłumaczone jako "interfejs transmisji wiadomości", to "protokół komunikacyjny będący standardem przesyłania komunikatów pomiędzy procesami programów równoległych działających na jednym lub więcej komputerach"[3]. MPI charakteryzuje się jakością, skalowalnością oraz przenośnością. Najczęściej standard MPI implementowany jest w postaci bibliotek np. dla języka C++. Interfejs ten specyfikuje, jak w dowolnej implementacji mają się zachowywać jego elementy.

1.5 Podejście hybrydowe

Do tworzenia aplikacji równoległych działających na wieloprocesorowych węzłach klastrów komputerowych można zastosować rozwiązanie hybrydowe. W takim podejściu programy są uruchamiane na klastrach komputerowych pod kontrolą interfejsu MPI, a do zrównoleglenia pracy węzłów klastrów wykorzystuje się OpenMP.

2 Implementacja

2.1 Generacja danych

Program przetestowano na 3 różnych zbiorach. Należało wygenerować dane zawierające 1 000 000, 2 000 000 oraz 3 000 000 liczb zmiennoprzecinkowych. Dane zapisywano do tablicy typu *double*.

```
double* generate_data(int n, int low, int high) {
        double* data = new double[n];
        for (int i = 0; i < n; i++) {
```

[2]https://pl.wikipedia.org/wiki/OpenMP
[3]https://pl.wikipedia.org/wiki/Message_Passing_Interface

```
                data[i] = ((double)(rand() % 10000) / 10000.0) * (
                    high - low) + low;
        }
        return data;
}
```

```
#define THREADS 4
#define SIZE 3e6
#define RANGE 10000
#define LB 2000
#define UB 5000

...

        data = generate_data(SIZE, 0, RANGE);
```

2.2 Bez zrównoleglenia

Obliczanie średniej z podzbioru liczb zmiennoprzecinkowych określonego dwoma wartościami odpowiadającymi najmniejszej i największej wartości docelowego podzbioru:

```
double* computeAVG_values(int n, double* data, int val_min, int
    val_max) {

        auto beginTime = std::chrono::high_resolution_clock::now()
            ;

        double avg = 0, sum = 0;
        int counter = 0;

        float min = *std::min_element(data, data + n);
        float max = *std::max_element(data, data + n);
        //std::cout << "\nMin = " << min << "\nMax = " << max;

        if (val_min < min || val_max > max) {
                std::cout << "\n(Srednia z zakresu wartosci) ERROR
                    : Wartosci poza zakresem.\n";
                return NULL;
        }
        else {
                if (val_min > val_max) {
                        std::cout << "\n(Srednia z zakresu
                            wartosci) ERROR: Wartosc min > max.\n";
                        return NULL;
                }
```

```
                else {
                        for (int i = 0; i < n; i++) {
                                if (data[i] >= val_min && data[i]
                                    <= val_max) {
                                        sum += data[i];
                                        counter++;
                                }
                        }

                        avg = sum / counter;
//                      std::cout << "\nSrednia (zakres wartosci)
    = " << avg;

                        auto endTime = std::chrono::
                            high_resolution_clock::now();
                        auto time = std::chrono::duration_cast<std
                            ::chrono::milliseconds>(endTime -
                            beginTime);

                        double* results = new double[4];

                        results[0] = avg;
                        results[1] = sum;
                        results[2] = counter;
                        results[3] = time.count() * 1e-3;

                        return results;
                }
        }

}
```

Funkcja przyjmuje parametry: *n* - rozmiar zbioru (ilość liczb), *data* - zbiór, *val_min* - wartość minimalna, *val_max* - wartość maksymalna. Po sprawdzeniu czy podane parametry są poprawne (czy *max* nie jest mniejsze od *min* oraz czy wartości mieszczą się w zakresie wartości zbioru oryginalnego) funkcja oblicza średnią określonego podzbioru przez zsumowanie danych wartości i podzielenie wyniku przez ich ilość. Wewnątrz funkcji sprawdzany jest czas wykonywania obliczeń. Funkcja zwraca tablicę z informacją na temat wartości średniej, sumy elementów podzbioru, ich ilości oraz czasu wykonywania.

main:

```
        double* results_seq = computeAVG_values(SIZE, data, LB, UB
            );

                //Wypisanie rezultatow
                printf("\nCzas Sekwencyjne: %f", results_seq[3]);
```

```
                printf("\nAVG Sekwencyjne: %.2f", results_seq[0]);
```

Sprawdzana jest średnia (i czas jej obliczenia) dla wartości z zakresu od 2 000 do 5 000.

2.3 OpenMP

W przypadku wykorzystywania interfejsu OpenMP, zastosowano przedstawioną wyżej funkcję, jednak nie do obliczenia średniej całego podzbioru, a średnich części powstałych z podziału podzbioru na wątki. Ilość wątków determinuje ilość podzbiorów.

```
    double* sums_OpenMP = new double[THREADS];
    double* counts_OpenMP = new double[THREADS];

    auto beginTime = std::chrono::high_resolution_clock::now();

    omp_set_num_threads(THREADS);

#pragma omp parallel
        {
                //pobranie ID watku
                int tread_num = omp_get_thread_num();

                // obliczenie przedzialow dla ktorych srednie beda
                    liczone rownolegle
                int begin = round(double(SIZE / THREADS) *
                   tread_num);
                int end = round(double(SIZE / THREADS) * tread_num
                    + SIZE / THREADS);
                int subsize = end - begin;

                //podzial zbioru
                double* splitted_data = new double[subsize];
                splitted_data = split_data(data, begin, end);

                //obliczenie wynikow w kazdym watku
                double* results = computeAVG_values(subsize,
                   splitted_data, LB, UB);

                sums_OpenMP[tread_num] = results[1];
                counts_OpenMP[tread_num] = results[2];

        }

        //obliczenie sredniej ogolnej
        double sum_OpenMP = 0;
```

```
        double count_OpenMP = 0;

        for (int i = 0; i < THREADS; i++) {
                sum_OpenMP = sum_OpenMP + sums_OpenMP[i];
                count_OpenMP = count_OpenMP + counts_OpenMP[i];
        }
        double avg_OpenMP = sum_OpenMP / count_OpenMP;

    auto endTime = std::chrono::high_resolution_clock::now();
        auto time = std::chrono::duration_cast<std::chrono::
           milliseconds>(endTime - beginTime);

        //Wypisanie rezultatow
        printf("\nCzas OpenMP: %f", time.count() * 1e-3);
        printf("\nAVG OpenMP: %.2f", avg_OpenMP);
}
```

Sprawdzono poprawność obliczeń porównując wyniku procedury sekwencyjnej z tą wykorzystującą interfejs OpenMP.

```
Czas Sekwencyjne: 0,062000
AVG Sekwencyjne: 3410,99
Czas OpenMP: 0,025000
AVG OpenMP: 3410,99
```

Dla 4 wątków czas wykonywania obliczeń przy wykorzystaniu OpenMP jest krótszy. Szczegółowe zestawienie wyników można znaleźć w sekcji *Porównanie*.

2.4 MPI

MPI pozwala na tworzenie procesów oraz komunikację między nimi. Po kompilacji programu należy go uruchomić za pomocą komendy mpiexec z argumentem -n, który odpowiada za ilość stworzonych procesów. Dzięki takiemu podejściu program jest w łatwy sposób skalowalny bez konieczności kompilacji.

Program działa w następujący sposób. Najpierw obliczany jest podział tablicy między wątki. Długość tablicy dzielona jest na ilość procesów, a następnie liczone jest dopełnienie. Iloczyn rozmiaru i ilości procesów liczących jest odejmowany od rozmiaru tablicy. Ma to znaczenie w przypadku gdy rozmiar tablicy jest niepodzielny przez ilość procesów. Następnie kawałki tablicy są rozsyłane do procesów potomnych gdzie wyliczane są sumy oraz zliczane jest ilość wartości należących do zadanego przedziału. Po obliczeniach wyniki wysyłane są do procesu 0 gdzie dokonywane jest ostateczne wyliczenie średniej.

```
    MPI_Barrier(MPI_COMM_WORLD);
```

```
double start = MPI_Wtime();
int count = SIZE/(size-1);
int rest = SIZE - count * (size-1);
double * local_array;
if (rank == 0) {
    int cnt;
for(int dest = 1; dest < size; ++dest)
{
    if(dest < size - 1) {
        cnt = count;
    } else {
        cnt = count + rest;
    }

    MPI_Send(&data[(dest - 1) * cnt], cnt, MPI_DOUBLE,
       dest, tag, MPI_COMM_WORLD);
    //printf("\nP0 sent a %d elements to P%d.", cnt, dest)
       ;
}
}

if (rank != 0) {
    int cnt;
    if(rank == size - 1){
    cnt = count + rest;
        local_array = new double [cnt];
    } else {
        cnt = count;
        local_array = new double [cnt];
    }
MPI_Recv(local_array, cnt, MPI_DOUBLE, 0, tag,
   MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    double* results = computeAVG_values(cnt, local_array,
       LB, UB);
MPI_Send(&results[0], 4, MPI_DOUBLE, 0, tag,
   MPI_COMM_WORLD);

}
MPI_Barrier(MPI_COMM_WORLD);


if (rank == 0) {
    double sum = 0;
    double count_in_tread = 0;
for(int src = 1; src < size; ++src)
{
    double* results = new double[4];
    MPI_Recv(results, 4, MPI_DOUBLE, src, tag,
       MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
            sum += results[1];
            count_in_tread += results[2];
        }

        double avg_MPI = sum / count_in_tread;
        double end = MPI_Wtime();
        printf("\nCzas MPI: %f", end - start);
        printf("\nAVG MPI: %.2f", avg_MPI);
    }
```

```
Czas Sekwencyjne: 0,062000
AVG Sekwencyjne: 3410,99
Czas OpenMP: 0,025000
AVG OpenMP: 3410,99
Czas MPI: 0,020163
AVG MPI: 3410,99
```

Porównując wynik obliczeń za pomocą MPI z OpenMP i metodą sekwencyjną widać, że wyniki są zgodne, co świadczy o poprawności działania programu.

2.5 OpenMP + MPI

Dzięki połączeniu wyżej wymienionych metod możliwe jest osiągnięcie mniejszego czasu obliczeń. Program jest kombinacją wcześniej zaprezentowanych rozwiązań. W ramach każdego procesu (MPI) obliczenia są dodatkowo zrównoleglane za pomocą wątków(OpenMP).

Program działa w następujący sposób. Najpierw w procesie 0 rozsyłane są kawałki tablicy do procesów potomnych, które liczą średnie. Procesy potomne dzielą tablicę na podzakresy na których będą operować wątki stworzone w poszczególnych procesach. Następnie otwierane są wątki w uprzednio zdefiniowanej ilości i w każdym wątku obliczane są sumy i zliczane ilości wartości. Po tym etapie wyliczenia wątków są agregowane do zbiorczego rezultatu procesu. Następnie wyniki są odsyłane do procesu 0 w którym wyliczana jest ostateczna wartość. Jak widać w przypadku metody hybrydowej dochodzi do podwójnego podziału tablicy - raz na poziomie procesów a raz na poziomie wątków.

```
    MPI_Barrier(MPI_COMM_WORLD);
    start = MPI_Wtime();
    if (rank == 0) {
        int cnt;
        for(int dest = 1; dest < size; ++dest)
        {
            if(dest < size - 1) {
                cnt = count;
```

```
            } else {
                cnt = count + rest;
            }

            MPI_Send(&data[(dest - 1) * cnt], cnt, MPI_DOUBLE,
                dest, tag, MPI_COMM_WORLD);
            //printf("\nP0 sent a %d elements to P%d.", cnt, dest)
                ;
        }
    }

    if (rank != 0) {
        int cnt;
        if(rank == size - 1){
            cnt = count + rest;
            local_array = new double [cnt];
        } else {
            cnt = count;
            local_array = new double [cnt];
        }
        MPI_Recv(local_array, cnt, MPI_DOUBLE, 0, tag,
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        omp_set_num_threads(THREADS);
        double* sums_hyb = new double[THREADS];
        double* counts_hyb = new double[THREADS];
#pragma omp parallel
        {
            //pobranie ID watku
            int tread_num = omp_get_thread_num();

            // obliczenie przedzialow dla ktorych srednie beda
                liczone rownolegle
            int begin = round(double(cnt / THREADS) * tread_num);
            int end = round(double(cnt / THREADS) * tread_num +
                cnt / THREADS);
            int subsize = end - begin;

            //podzial zbioru
            double* splitted_data = split_data(local_array, begin,
                end);

            //obliczenie wynikow w kazdym watku
            double* results = computeAVG_values(subsize,
                splitted_data, LB, UB);

            sums_hyb[tread_num] = results[1];
            counts_hyb[tread_num] = results[2];
```

```
        }

        //obliczenie sredniej ogolnej
        double results[2] = {0};
        for (int i = 0; i < THREADS; i++) {
            results[0] += sums_hyb[i];
            results[1] += counts_hyb[i];
        }

        MPI_Send(&results[0], 2, MPI_DOUBLE, 0, tag,
           MPI_COMM_WORLD);

    }
    MPI_Barrier(MPI_COMM_WORLD);


    if (rank == 0) {
        double sum = 0;
        double count_in_tread = 0;
        for(int src = 1; src < size; ++src)
        {
            double* results = new double[4];
            MPI_Recv(results, 2, MPI_DOUBLE, src, tag,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            sum += results[0];
            count_in_tread += results[1];
        }

        double avg_hyb = sum / count_in_tread;
        double end = MPI_Wtime();
        printf("\nCzas HYBRID: %f", end - start);
        printf("\nAVG HYBRID: %.2f", avg_hyb);
    }
```

```
Czas Sekwencyjne: 0,062000
AVG Sekwencyjne: 3410,99
Czas OpenMP: 0,025000
AVG OpenMP: 3410,99
Czas MPI: 0,020163
AVG MPI: 3410,99
Czas HYBRID: 0,023468
AVG HYBRID: 3410,99
```

Porównując wyniki metody hybrydowej z wynikami poprzednich rozwiązań można wnioskować, że program działa poprawnie.

3 Porównanie

3.1 Wykresy

Niżej przedstawione wykresy zostały wygenerowane dla zbiorów 1 000 000, 2 000 000 i 3 000 000 elementowych, zawierających liczby zmiennoprzecinkowe z zakresu 0 - 10 000. W każdym zbiorze liczona była średnia wartości z zakresu od 2 000 do 5 000. Sprawdzono czasy wykonania obliczeń dla różnej ilości procesów i wątków.

3.1.1 Zależności czasu od ilości danych

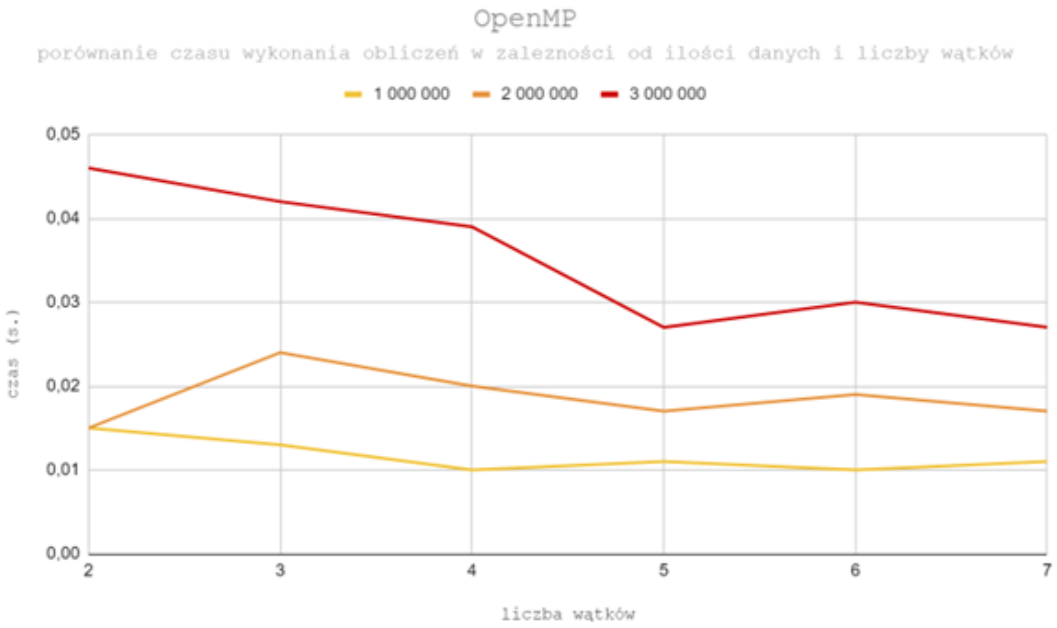
Dla implementacji wykorzystującej obliczenia sekwencyjne sprawdzono dodatkowo jak ilość danych wpływa na czas wykonywana obliczeń.

Wykres zależności czasu od ilości danych w implementacji sekwencyjnej

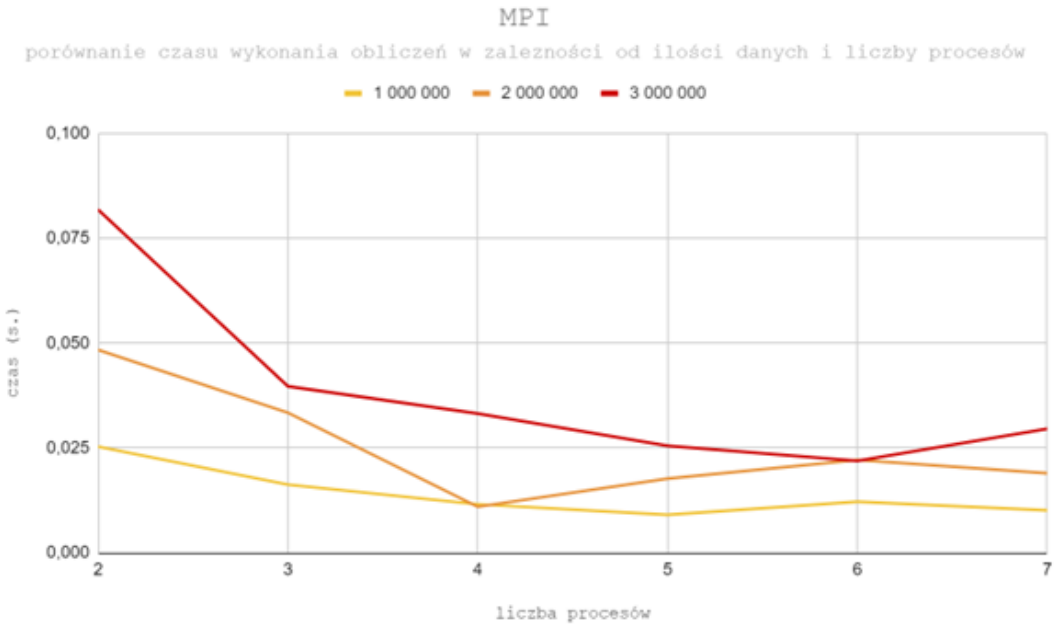
Nie jest niczym zaskakującym, że czas wykonania obliczeń rośnie wraz z ilością danych.

3.1.2 Zależności czasu od liczby wątków

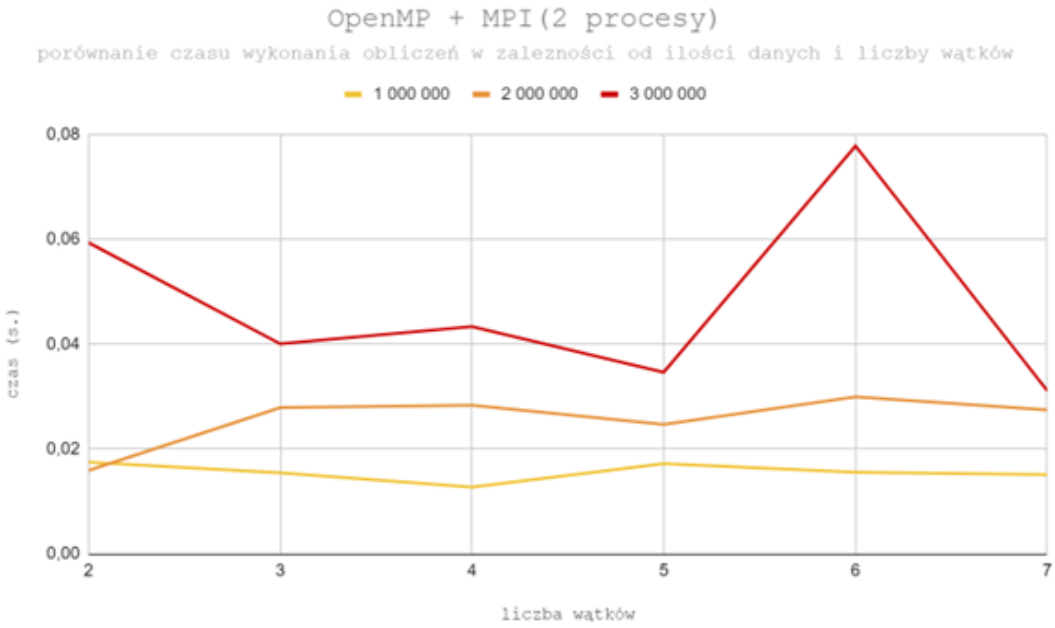
Dla implementacji wykorzystującej interfejsy OpenMP i MPI i dla różnej wielkości zbiorów sprawdzono jak ilość wątków/procesów wpływa na czas wykonywana obliczeń. To samo sprawdzono dla implementacji hybrydowej biorąc pod uwagę 2, 4 i 8 procesów (MPI).



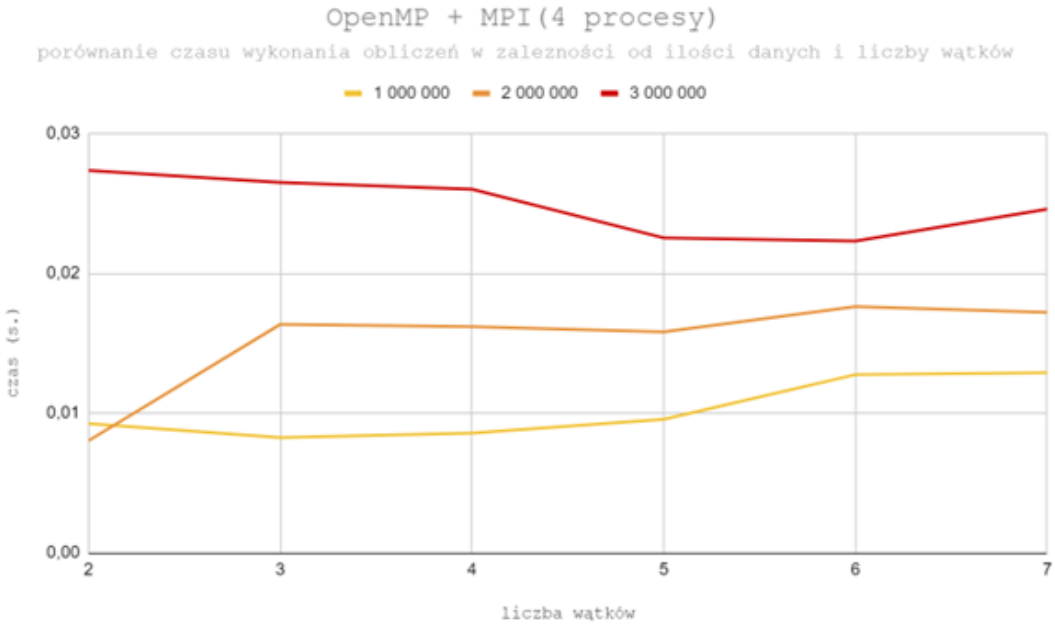
Wykres zależności czasu od ilości wątków - OpenMP wg. wielkości zbioru



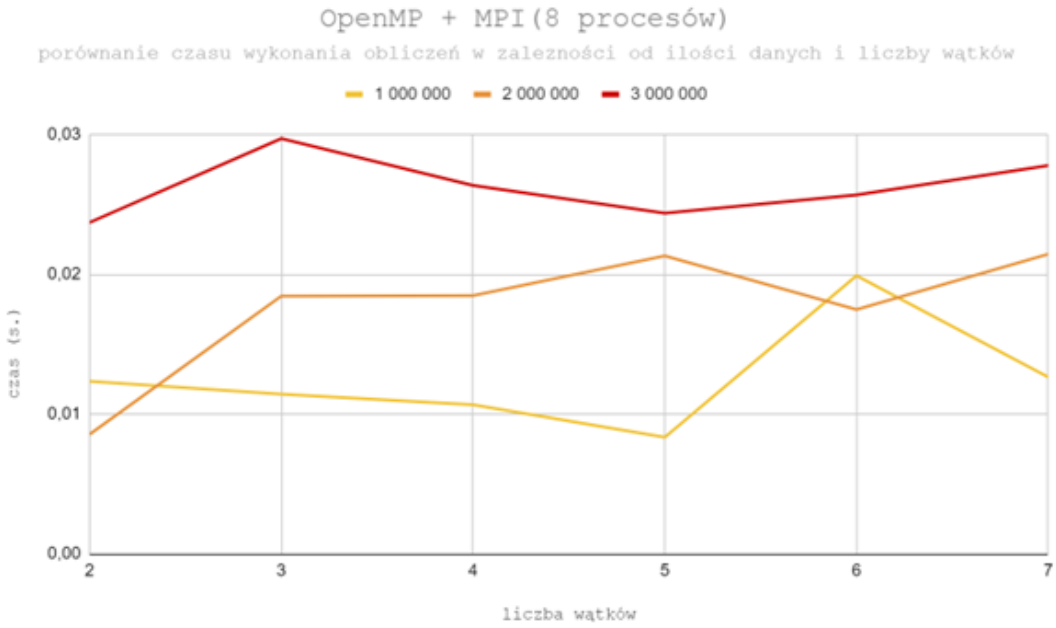
Wykres zależności czasu od ilości procesów - MPI wg. wielkości zbioru



Wykres zależności czasu od ilości wątków - OpenMP + MPI(2 procesy) wg. wielkości zbioru



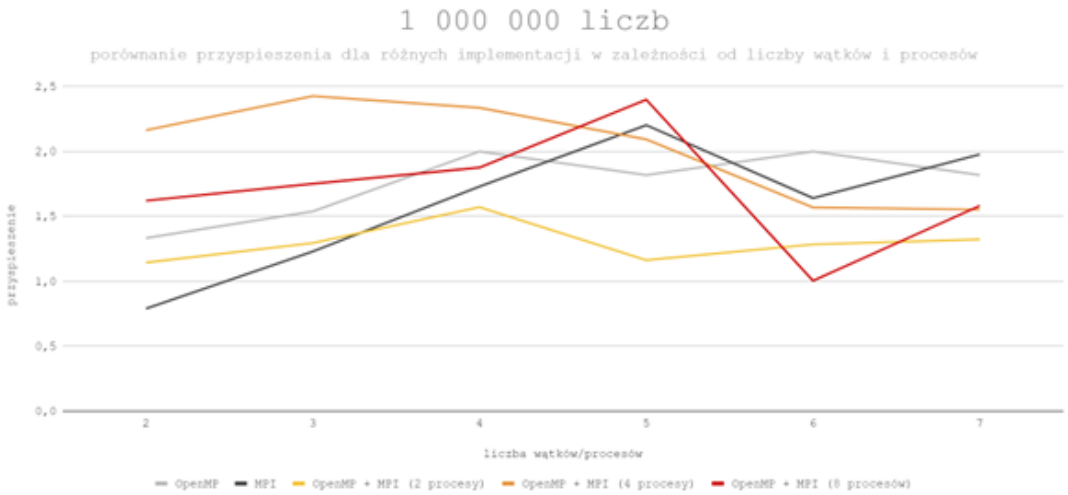
Wykres zależności czasu od ilości wątków - OpenMP + MPI(4 procesy) wg. wielkości zbioru



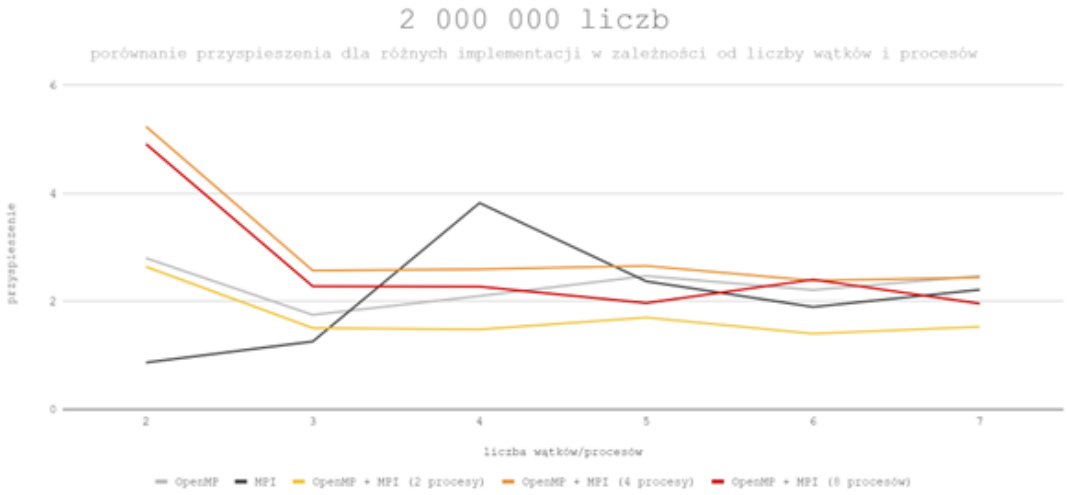
Wykres zależności czasu od ilości wątków - OpenMP + MPI(8 procesów) wg. wielkości zbioru

3.1.3 Wykresy przyspieszenia

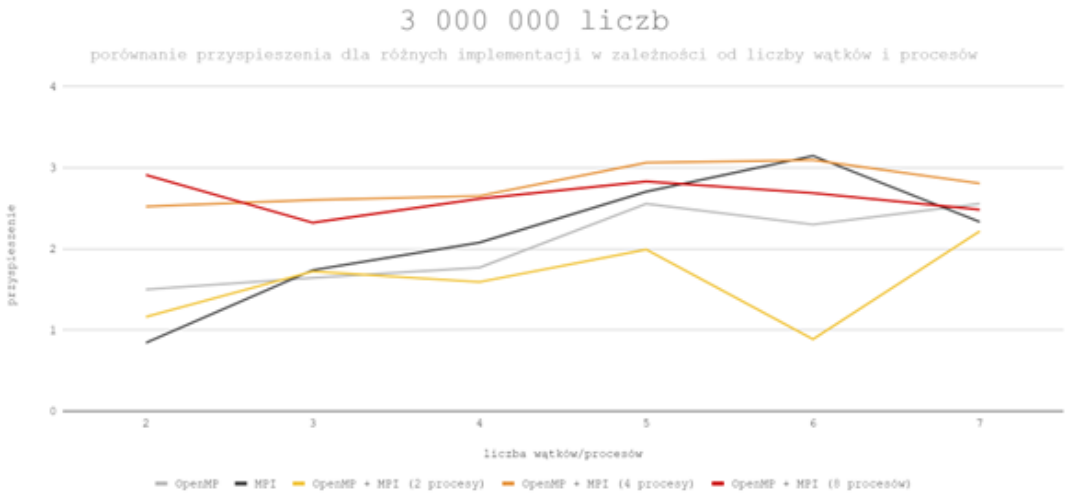
Wykresy przyspieszenia służą do porównania czasu, który potrzebny jest do wykonania obliczeń programu napisanego sekwencyjnie do czasu obliczeń wykonywanych równolegle na n procesorach. Na osi poziomej jest zatem liczba procesów (dla MPI)/wątków (dla OpenMP i metody hybrydowej), a na osi pionowej jest przyspieszenie - iloraz czasu obliczeń sekwencyjnych do czasu wykonywania obliczeń równoległych na n procesach/wątkach. Wygenerowano wykresy porównujące zastosowane implementacje dla 1 000 000, 2 000 000 i 3 000 000 elementowych zbiorów.



Wykres przyspieszenia dla 1 000 000 liczb i różnych implementacji w zależności od liczby wątków/procesów



Wykres przyspieszenia dla 2 000 000 liczb i różnych implementacji w zależności od liczby wątków/procesów



Wykres przyspieszenia dla 3 000 000 liczb i różnych implementacji w zależności od liczby wątków/procesów

3.2 Wnioski

Zdecydowanie najwięcej zalet programowania równoległego widać dopiero w przypadku zbioru o wielkości 3 mln. Największe zmiany czasu wykonywania w zależności od ilości wątków/procesów było widać w przypadku zastosowania samego OpenMP lub MPI. Czas wykonania wyraźnie spadał wraz ze wzrostem liczby wątków/procesów. W przypadku metody hybrydowej czas wykonywania utrzymywał się mniej więcej na stałym poziomie niezależnie od ilości wątków OpenMP i procesów MPI. Nie jest niczym zaskakującym, że czas wykonania dla każdego rodzaju implementacji najwyższy jest w przypadku 3 000 000 liczb.

Dla zbioru z 1 mln danych największe przyspieszenie uzyskano przy użyciu metod hybrydowych (4 i 8 procesów MPI) dla 3 i 5 wątków. W przypadku 2 procesów i metody wykorzystującej MPI obliczenia wykonywały się dłużej niż sekwencyjnie.

Podobnie w przypadku zbioru z 2 mln liczb w przypadku 2 procesów i metody wykorzystującej MPI obliczenia wykonywały się dłużej niż sekwencyjnie. Największe przyspieszenie (ok. 5 razy) otrzymano stosując implementacje hybrydowe (4 i 8 procesów MPI) przy 2 wątkach. Jest to równocześnie największe przyspieszenie jakie udało się uzyskać biorąc pod uwagę wszystkie obserwacje.

Dla zbioru 3 mln najlepsze przyspieszenie uzyskano również dla metody hybrydowej (4 i 8 procesów MPI). Było ono najbardziej stabilne wahające się od 2.5 do 3 razy niezależnie od ilości wątków OpenMP dla 4 i 8 procesów MPI. W przypad-

ku zastosowania MPI oraz OpenMP przyspieszenia wahały się od około 0.9 (dla 2 procesów dla MPI znów obliczenia wykonywały się dłużej) do 3 w najlepszym przypadku (6 procesów MPI). Zalety metody hybrydowej zdecydowanie lepiej widać w przypadku dużego zbioru danych, choć największe przyspieszenie osiągnęła metoda MPI dla 6 procesów. Wszystkie wykorzystane implementacje wykorzystujące zrównoleglenie obliczeń sprawdzają się znacznie lepiej niż obliczenia sekwencyjne (nie biorąc pod uwagę przypadków MPi dla 2 procesów).