POLITECHNIKA OPOLSKA	**Politechnika Opolska** **Wydział Elektrotechniki, Automatyki i Informatyki** **Instytut Informatyki**
Rok akademicki	2023/2024
Przedmiot	Programowanie współbieżne i rozproszone
Forma zajęć	Laboratorium
Prowadzący zajęcia	mgr inż. Maciej Walczak
Grupa	**3**

Zadanie nr 2

Nazwisko i imię	Nr indeksu
Jakub Nieśmiała	**101374**

Uwagi

1. Charakterystyka zadania

Zadanie polega na stworzeniu programu w C++, który równolegle wykonuje mnożenie dwóch macierzy używając trzech różnych technologii: OpenMP, biblioteki wątków w C++, oraz MPI. Program pobiera macierze wejściowe z plików CSV, wykonuje mnożenie macierzy zgodnie z wybraną metodą numeryczną, a następnie zapisuje wynik do nowego pliku CSV. Dodatkowo, program musi obsługiwać parametry przekazywane przez wiersz poleceń oraz umożliwiać kontrolę liczby wątków lub procesów używanych przez program.

$$c_{i,j} = \sum_{r=1}^{m} a_{i,r} \cdot b_{r,j}$$

2. Opis środowiska

1. **Procesor**: Intel Core i5-11400H
    - Model procesora: Intel Core i5-11400H
    - Liczba rdzeni fizycznych: 6
    - Liczba procesorów logicznych(wątków): 12
    - Częstotliwość bazowa: 2.7 GHz
2. **Pamięć RAM**: 16 GB DDR4
    - Pojemność: 16 GB
    - Typ: DDR4
    - Prędkość: Standardowa szybkość dla pamięci DDR4
3. **System operacyjny**: Windows 11
    - Wersja systemu operacyjnego: Windows 11 Home 23H2
    - Architektura systemu: x64 (64-bitowa)
4. **Oprogramowanie i narzędzia programistyczne**:
    - Środowisko programistyczne: Visual Studio
    - Wersja Visual Studio: 2022

- Język programowania: C++
- Biblioteki i narzędzia:
  - MPI 4.1.1
  - OpenMP: 5.0.
  - C++ Threads: C++17

3. Opis zastosowanych technologii oraz algorytmu

Algorytm mnożenia macierzy w programie zawiera kilka kluczowych etapów:

1. **Wczytanie danych**: Pobranie macierzy wejściowych AA i BB z plików CSV oraz sprawdzenie ich wymiarów.
2. **Przygotowanie wyniku**: Stworzenie macierzy wynikowej CC o odpowiednich wymiarach.
3. **Podział obciążenia**: Rozdzielenie macierzy AA na fragmenty do przetworzenia przez poszczególne wątki lub procesy, zapewniając równomierne obciążenie.
4. **Wykonanie obliczeń równolegle**: Mnożenie macierzy w sposób równoległy przy użyciu odpowiedniej technologii (OpenMP, MPI lub biblioteki wątków w C++).
5. **Zbieranie wyników**: Łączenie częściowych wyników i zapisanie ich do macierzy wynikowej CC.
6. **Zapisanie wyniku do pliku**: Zapisanie macierzy CC do pliku CSV.

Sposób dekompozycji i dystrybucji zadania oraz zasoby współdzielone przedstawione są poniżej dla każdego wariantu zrównoleglania kodu:

1. **MPI**
   - Proces o rankingu 0, czyli master, odpowiada za pobranie wartości z wiersza poleceń. Te wartości reprezentują parametry zadania. Proces master konwertuje te wartości na odpowiedni typ danych i format (spłaszczanie macierzy B aby przesłać ją jako wektor nie tablice) oraz rozgłasza parametry zadania do wszystkich procesów. Następnie algorytm wykonuje się równolegle przez wiele procesów MPI, gdzie każdy oblicza wyniki mnożenia macierzy dla swojego podprzedziału macierzy. Współdzielone są dane wejściowe (parametry zadania), które są rozgłaszane na wszystkie procesy oraz macierz wynikowa C, jednak każdy proces zapisuje wyniki dla swojego zakresu macierzy przez co procesy nie kolidują ze sobą zapisując dane wynikowe.

2. **OpenMP**
   - Zadanie jest rozdzielane na mniejsze części, które są przetwarzane równolegle. Każdy fragment macierzy *A* jest obliczany przez osobny wątek, który generuje odpowiedni fragment macierzy *C*. Zastosowany został mechanizm synchronizacji

#pragma omp atomic który umożliwia wielu wątkom dostęp do tej samej zmiennej (każdy na swoim zakresie komórek).

3. **C++ thread Library**
   - Dekompozycja i dystrybucja: Wątki są tworzone w zależności od liczby wątków podanych przez użytkownika. Każdy wątek oblicza wartości dla swojego fragmentu macierzy oraz zapisuje je w macierzy wynikowej. Każdy wątek pracuje na swoim zakresie macierzy i nie ma sytuacji w której dwa wątki próbują zapisać dane do jednej komórki tablicy przez co nie ma konieczności używania mutex do pojedynczego dostępu do danych.

Ponadto w implementacji zastosowane zostały różne mechanizmy MPI, OpenMP oraz C++ Thread Library umożliwiające zrównoleglenie obliczeń. Opisy wykorzystanych mechanizmów dla każdego wariantu programu przedstawione są poniżej:

1. **MPI**
   - MPI_Init - Mechanizm inicjujący MPI, który przygotowuje środowisko MPI do pracy,
   - MPI_Comm_rank - Pobiera numer identyfikacyjny (rank) bieżącego procesu wewnątrz komunikatora MPI_COMM_WORLD,
   - MPI_Comm_size - Pobiera liczbę procesów biorących udział w komunikacji wewnątrz komunikatora MPI_COMM_WORLD,
   - MPI_Bcast - Mechanizm rozgłaszania, który przesyła dane z jednego procesu (w tym przypadku procesu o ranku 0) do wszystkich innych procesów,
   - MPI_Scatterv(): Rozdziela dane z jednego procesu na wiele procesów w równomierny sposób. W tym programie jest wykorzystane do rozproszenia fragmentów macierzy A na wszystkie procesy.
   - MPI_Gatherv(): Zbiera dane z wielu procesów i zbiera je w jednym procesie. W programie jest używane do zebrania wyników obliczeń z każdego procesu do procesu o ranku 0, aby utworzyć macierz wynikową C.
   - MPI_Barrier - Mechanizm synchronizacji, który wymusza oczekiwanie na wszystkich procesach do momentu, gdy każdy z nich osiągnie daną linijkę kodu,
   - MPI_Reduce - Mechanizm redukcji, który zbiera dane z różnych procesów i wykonuje operację na tych danych (np. sumę),
   - MPI_Finalize() - Mechanizm kończący pracę z MPI, czyszczący i zamykający wszystkie zasoby używane przez MPI.

2. **OpenMP**
   - #pragma omp parallel for - Ta dyrektywa tworzy region równoległy, gdzie każdy wątek wykonuje pętlę for równolegle,
   - collapse(collapseLevel) – dyrektywa ta pozwala na zagnieżdżanie wielu pętli w jednej równoległej sekcji, zwiększając poziom równoległości.
   - schedule(scheduleType) - Różne harmonogramy, takie jak static, dynamic i guided, kontrolują sposób przydziału iteracji pętli do wątków.

- omp atomic – dyrektywa ta zapewnia atomowe wykonanie operacji na zmiennych w celu uniknięcia problemów z dostępem wielu wątków do tych samych danych.
- omp_get_wtime() - Służy do pobrania czasu wykonywania się programu.

3. **C++ thread Library**
   - thread - Jest to klasa reprezentująca pojedynczy wątek wykonawczy. W kodzie użyto klasy std::thread do tworzenia i zarządzania wieloma wątkami, które są odpowiedzialne za równoległe obliczenia numeryczne,
   - mutex - jest klasą reprezentującą mutex (muteks), który służy do synchronizacji dostępu do współdzielonych zasobów przez wiele wątków. W kodzie użyto mutexów, aby zabezpieczyć dostęp do współdzielonych zmiennych, takich jak suma całkowania, zapobiegając w ten sposób równoczesnemu zapisowi przez wiele wątków.

4. Opis scenariuszy testowych i metodologii badań

Scenariusze testowe:

1. Poprawność wyników:

   • Sprawdzenie zgodności wyników z obliczeń sekwencyjnych dla małych danych.

   • Porównanie wyników dla różnych przypadków testowych.

2. Wydajność:

   • Pomiar czasu wykonania dla różnych liczb wątków lub procesów.

   • Porównanie czasów wykonania dla różnych implementacji.

3. Skalowalność:

   • Badanie czasu wykonania dla rosnącej liczby wątków lub procesów.

   • Analiza wydajności w zależności od zmiennej liczby wątków lub procesów.

Metodologia badań:

1. Testowanie poprawności wyników:

   • Porównanie wyników z obliczeniami sekwencyjnymi dla małych danych.

   • Analiza wyników dla różnych przypadków testowych.

2. Testowanie wydajności:

   • Pomiar czasu wykonania dla różnych konfiguracji wątków lub procesów.

• Porównanie czasów wykonania między implementacjami.

3. Testowanie skalowalności:

• Badanie czasu wykonania dla rosnącej liczby zasobów.

• Analiza wydajności w zależności od liczby wątków lub procesów.

5. Wyniki badań

Dla każdej implementacji programu zostało wykonane 10 testów dla każdej wielkości zadania, a następnie został obliczony średni czas wykonywania danego przypadku przedstawione poniżej w tabelkach.

Tabela z średnimi czasami wykonania dla MPI:

Macierz	Liczba wątków			
	1	4	8	12
50x50	2,1527 ms	2,5782 ms	2,4723 ms	2,4296 ms
100x100	13,7582 ms	7,3086 ms	6,7824 ms	6,4736 ms
200x200	99,0278 ms	41,4976 ms	27,6621 ms	25,9149 ms
300x300	340,4860 ms	109,3580 ms	80,9206 ms	73,1592 ms

Tabela z średnimi czasami wykonania dla klasy Threads:

Macierz	Liczba wątków			
	1	4	8	12
50x50	20,9375 ms	19,5183 ms	19,4723 ms	19,728 ms
100x100	83,4933 ms	71,1156 ms	69,5136 ms	68,5795 ms
200x200	391,9727 ms	295,4542 ms	287,2716 ms	276,1173 ms
300x300	1064,1486 ms	711,2188 ms	667,3812 ms	641,1458 ms

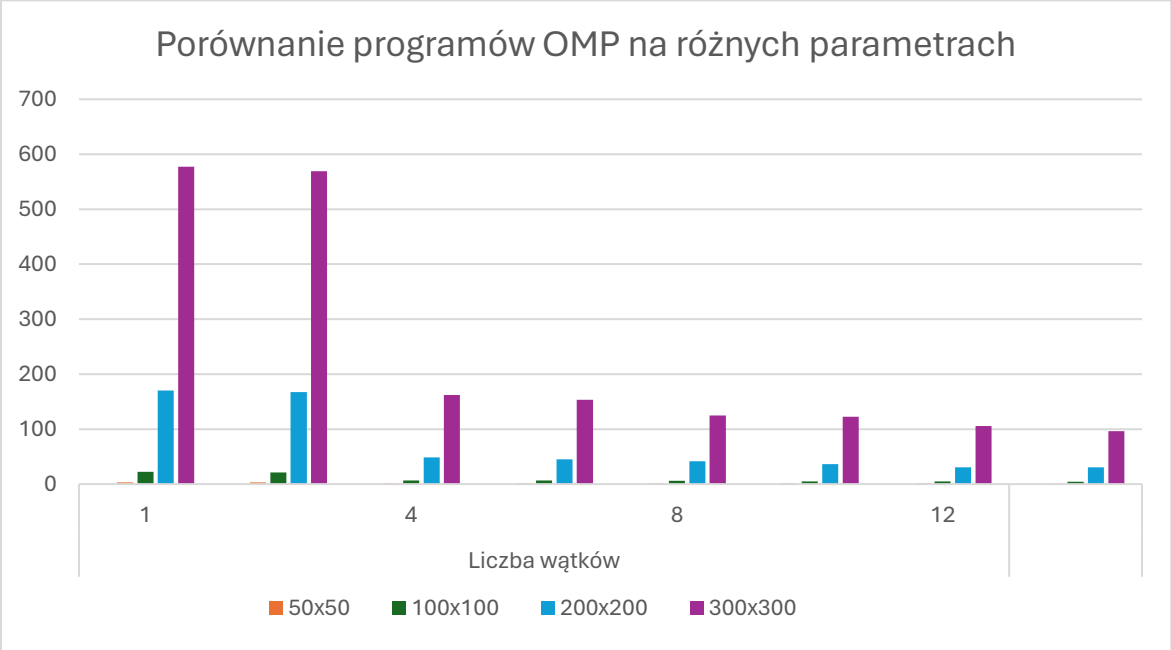
Tabela z średnimi czasami wykonania dla OMP z domyślną wartością schedule(static) oraz możliwością zrównoleglenia tylko 1 pętli:

Macierz	Liczba wątków			
	1	4	8	12
50x50	2,9313 ms	1,4159 ms	1,5493 ms	1,6121 ms
100x100	22,1478 ms	6,8449 ms	6,2630 ms	4,7237 ms
200x200	170.495 ms	48,3625 ms	41,5093 ms	30,5363 ms
300x300	577,2200 ms	162,2763 ms	124,6636 ms	95,7508 ms

Tabela z średnimi czasami wykonania dla OMP z najoptymalniejszymi wartościami czyli: schedule(guided) oraz możliwością zrównoleglenia do 3 pętli:

Macierz	Liczba wątków			
	1	4	8	12
50x50	2,9285 ms	1,1453 ms	1,1237 ms	0,9659 ms
100x100	21,0026 ms	6,3631 ms	5,1155 ms	4,3257 ms
200x200	167,1045 ms	45,0817 ms	36,0835 ms	30,7244 ms
300x300	569,2436 ms	153,2065 ms	122,6923 ms	96,3011 ms

Porównanie czasów wykonywania programu OMP z standardowym oraz optymalnym zestawem parametrów(dla danej ilości wątków najpierw standardowy później optymalny):

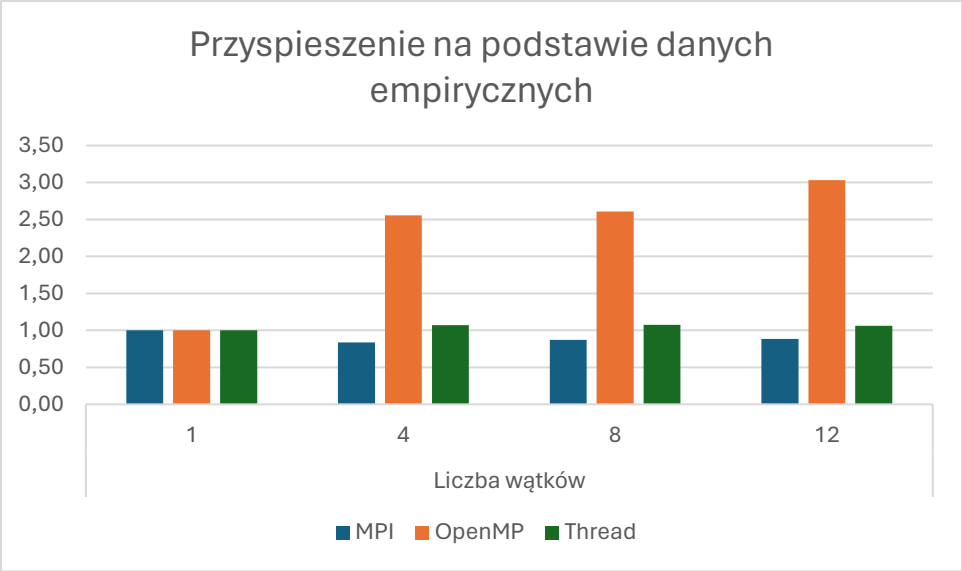


Ze względu na minimalnie szybszy czas wykonania w reszcie obliczeń uwzględniane będą pomiary z optymalnego zestawu parametrów dla programu OMP.

Oszacowanie przyśpieszenia na podstawie danych empirycznych:

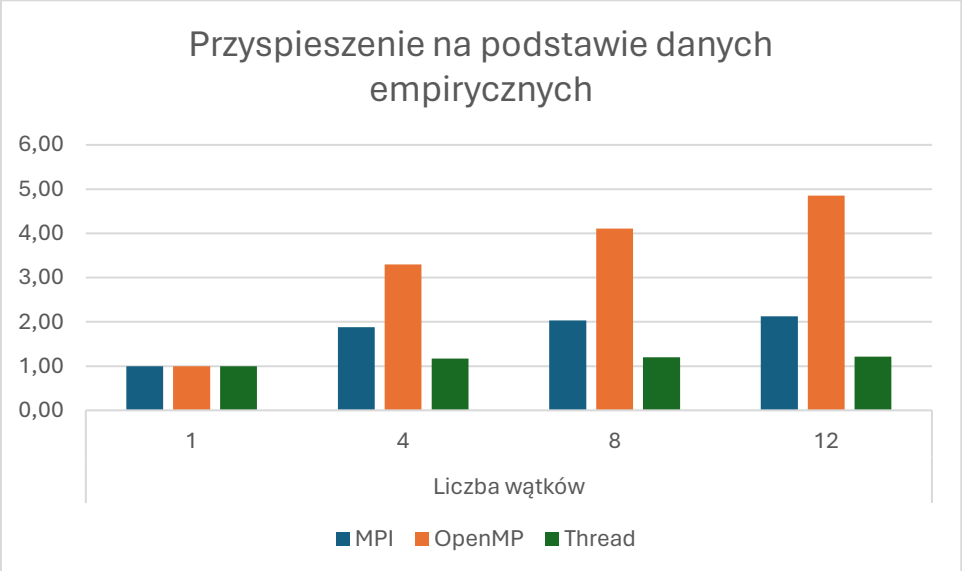
- Dla macierzy 50x50

	Przyspieszenie na podstawie danych empirycznych			
Liczba wątków	1	4	8	12
MPI	1,00	0,83	0,87	0,89
OpenMP	1,00	2,56	2,61	3,03
Threads	1,00	1,07	1,08	1,06



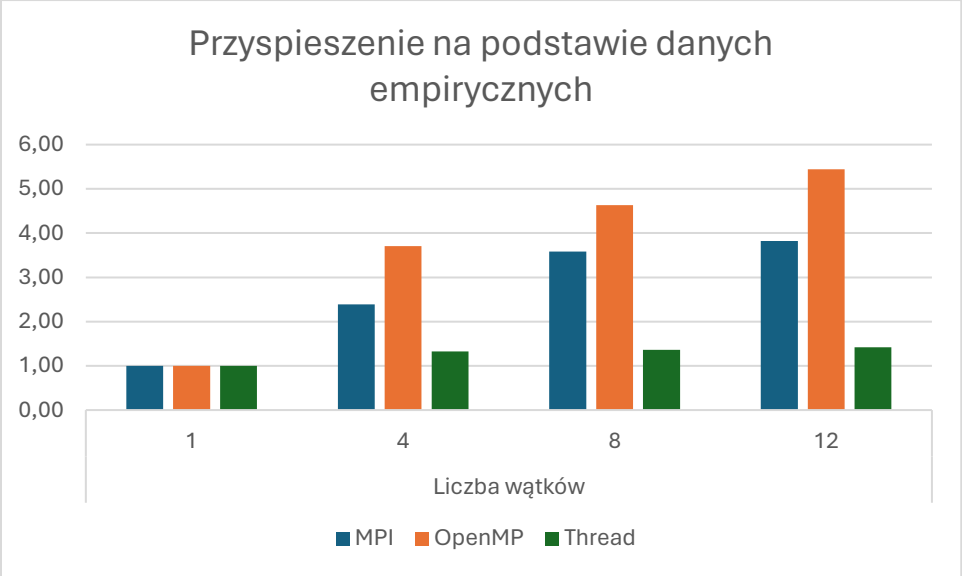
- Dla macierzy 100x100

	Przyspieszenie na podstawie danych empirycznych			
Liczba wątków	1	4	8	12
MPI	1,00	1,88	2,03	2,13
OpenMP	1,00	3,30	4,11	4,86
Threads	1,00	1,17	1,20	1,22



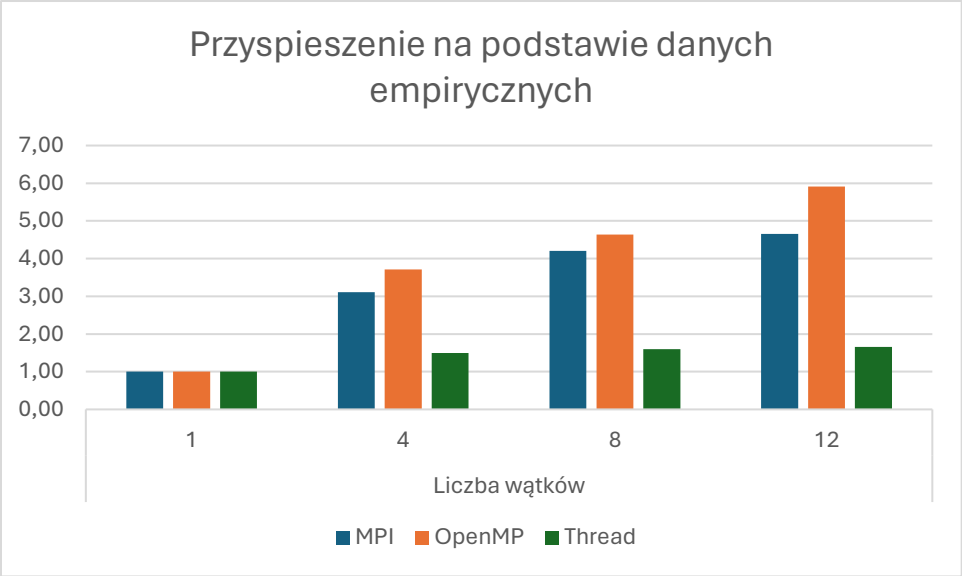
- Dla macierzy 200x200

	Przyspieszenie na podstawie danych empirycznych			
Liczba wątków	1	4	8	12
MPI	1,00	2,39	3,58	3,82
OpenMP	1,00	3,71	4,63	5,44
Threads	1,00	1,33	1,36	1,42



- Dla macierzy 300x300

	Przyspieszenie na podstawie danych empirycznych			
Liczba wątków	1	4	8	12
MPI	1,00	3,11	4,21	4,65
OpenMP	1,00	3,72	4,64	5,91
Threads	1,00	1,50	1,59	1,66



Prawo Amdahla wymaga określenia stopnia zrównoleglenia danego zadania oraz uwzględnienia części sekwencyjnej.

Przyśpieszenie można wyrazić wzorem:

$$S(p) = \frac{1}{(1-P)+\frac{P}{p}}$$

gdzie:

- S(p) to przyśpieszenie,

- P to stopień zrównoleglenia,

- p to liczba rdzeni lub wątków.

P= (Czas Sekwencyjny - Czas Równoległy) / Czas Sekwencyjny

Prawo Amdahla = 1/((1-P) + (P/Liczba wątków))

Przyśpieszenie z prawa Amdahla:

- Dla macierzy 50x50

	Przyspieszenie dla prawa Amdahla			
Liczba wątków	1	4	8	12

MPI	1,0000	0,8709	0,8998	0,9120
OpenMP	1,0000	1,8406	1,8595	2,0106
Thread	1,0000	1,0536	1,0554	1,0453

- Dla macierzy 100x100

	Przyspieszenie dla prawa Amdahla			
Liczba wątków	1	4	8	12
MPI	1,0000	0,9350	0,9301	0,9272
OpenMP	1,0000	1,4670	1,5278	1,5689
Thread	1,0000	1,0076	1,0086	1,0092

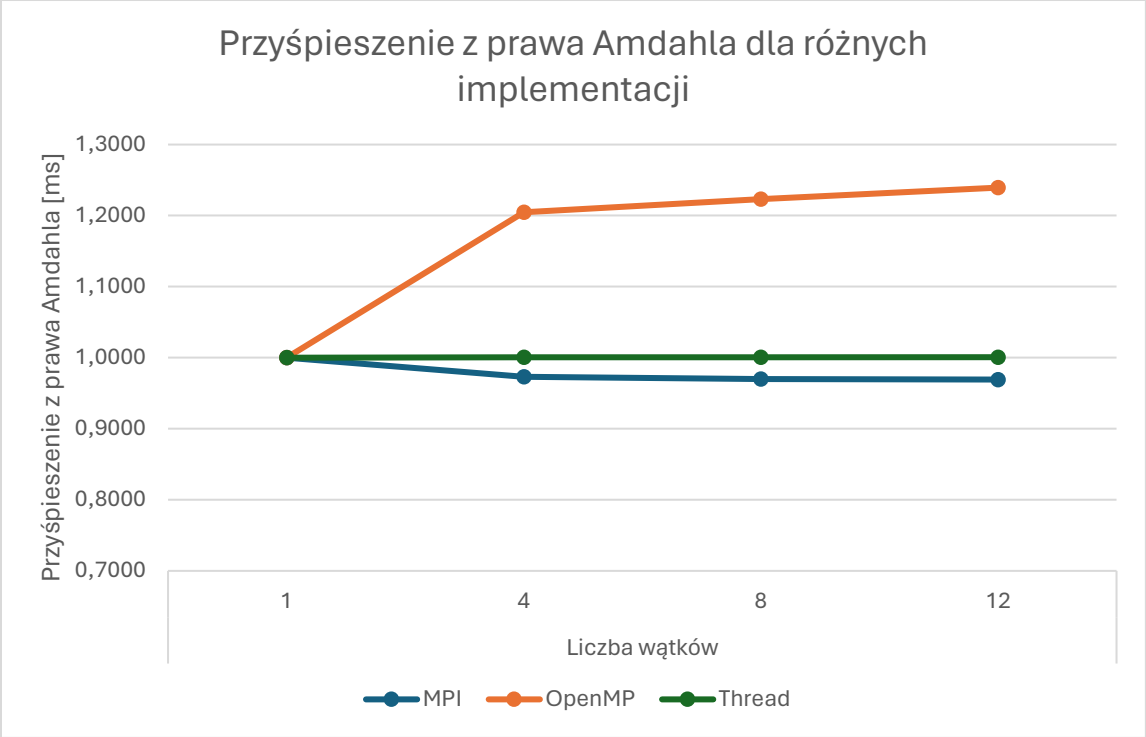
- Dla macierzy 200x200

	Przyspieszenie dla prawa Amdahla			
Liczba wątków	1	4	8	12
MPI	1,0000	0,9612	0,9523	0,9512
OpenMP	1,0000	1,3028	1,3326	1,3510
Thread	1,0000	1,0019	1,0020	1,0022

- Dla macierzy 300x300

	Przyspieszenie dla prawa Amdahla			
Liczba wątków	1	4	8	12
MPI	1,0000	0,9733	0,9701	0,9693
OpenMP	1,0000	1,2047	1,2230	1,2393
Thread	1,0000	1,0006	1,0007	1,0007

Wartości dla prawa Amdahla uzyskane dla największego zestawu danych przedstawione na wykresie:



Próg opłacalności realizacji równoległej:

Próg opłacalności implementacji równoległej można określić jako moment, kiedy przyspieszenie algorytmu równoległego przewyższa czas wykonania algorytmu sekwencyjnego dla określonego zestawu danych wejściowych.

Próg opłacalności = czas wykonania sekwencyjnego algorytmu / (Czas wykonania sekwencyjnego - Czas wykonania równoległego)

- Dla macierzy 50x50

	Próg opłacalności [ms]			
Liczba wątków	1	4	8	12
MPI	-	-5,059224442	-6,7356	-7,7743
OpenMP	-	1,642272319	1,62262	1,49215
Thread	-	14,75302988	14,2899	17,3109

- Dla macierzy 100x100

	Próg opłacalności [ms]			
Liczba wątków	1	4	8	12
MPI	-	2,133186554	1,97228	1,88867
OpenMP	-	1,434643023	1,32199	1,25938
Thread	-	6,745461596	5,97247	5,59839

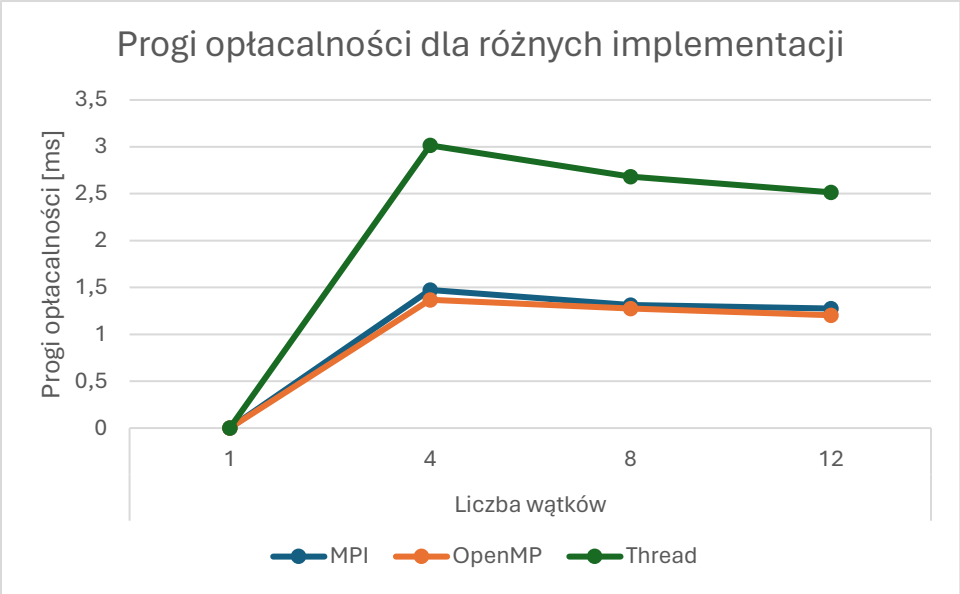
- Dla macierzy 200x200

	Próg opłacalności [ms]			
Liczba wątków	1	4	8	12
MPI	-	1,721318542	1,38761	1,35445
OpenMP	-	1,3694546	1,2754	1,22529
Thread	-	4,061114709	3,74373	3,38329

- Dla macierzy 300x300

	Próg opłacalności [ms]			
Liczba wątków	1	4	8	12
MPI	-	1,473149078	1,31175	1,27367
OpenMP	-	1,368250901	1,27476	1,20362
Thread	-	3,015184889	2,68205	2,5157

Progi opłacalności dla różnych implementacji programu przedstawione na wykresie poniżej:



6. Wnioski

Różne technologie równoległe, jak OpenMP, MPI i biblioteka wątków w C++, wykazują odmienne właściwości wydajnościowe w zależności od rodzaju operacji i rozmiaru danych. W przypadku mniejszych zbiorów danych oraz niewielkiej ilości wątków, implementacje równoległe mogą generować dodatkowe koszty związane z synchronizacją i zarządzaniem wątkami, co często prowadzi do ograniczonych korzyści wydajnościowych. Jednakże, wraz ze wzrostem rozmiaru danych i liczby wątków lub procesów, przyspieszenie staje się bardziej zauważalne. Dla większych zbiorów danych oraz większej liczby wątków, korzyści z równoległego przetwarzania stają się wyraźniejsze. W implementacji OpenMP najefektywniejszy okazał się zestaw parametrów schedule(guided) oraz collapse(3) który pozwala aby program przydzielał zadania do wątków dynamicznie, ale z tendencją do zmniejszania rozmiaru zadań w miarę postępującego wykonania oraz umożliwia programowi wykonanie równoległe do 3 petli.