PR-2023L

Projekt semestralny - Prace zaliczeniowe

Inżynieria obliczeniowa, ICM UW

prowadząca dr Dorota Dąbrowska

Filip Rusiecki

Nr indeksu 451709

Zadanie A4

Napisać program rozwiązujący równanie ciepła (iteracje Jakobiego) w trzech wersjach:

- używając tylko OpenMP (dla CPU, tak jak na zajęciach),
- używając tylko MPI,
- łacząc MPI i OpenMP.

Porównać wersje. Dobrać odpowiednio duże dane.

Exercise 9 (Jacobi iteration)

Implement Jacobi iterations for parameters:

- *size* 256×256*:*

```
#define N 256              // number of rows/columns
```

- $\varepsilon = 0.00006$

```
#define EPS 0.00006        // error val for stop criterion
#define MAX_ITER 1000000   // max number of iterations
```

- *boundary temperatures*

```
#define VAL_UPPPER 70.0    // upper side values
#define VAL_LOWER 1.0      // lower side values
#define VAL_LEFT 20.0      // left side values
#define VAL_RIGHT 50.0     // right side values
```

Exercise 9 (continued)

- *Measure the time using OpenMP time measurement function.*
- *Only measure the time of iterations: do not include creating arrays, filling boundaries, saving the array to a file.*
- *Print error values every 10000 iterations. You should get values like these:*

```
iter = 0  1.388944489831610
iter = 10000  0.001083229753157
iter = 20000  0.000479983971215
iter = 30000  0.000223512298666
iter = 40000  0.000104583646264
iter = 47321  0.000059998830854
```

W tym sprawozdaniu zajmiemy się rozwiązaniem równania ciepła, wykorzystując trzy różne metody obliczeń równoległych: za pomocą OpenMP, MPI oraz połączenia obu tych technologii. Równanie ciepła opisuje, jak rozchodzi się ciepło w danym materiale w czasie i żeby je rozwiązać, stosuje się techniki numeryczne, w tym wypadku iteracje Jakobiego.
Problem polega na tym, że obliczenia te mogą być bardzo czasochłonne, zwłaszcza przy większych danych, dlatego ważne jest przyspieszenie ich przez równoległe wykonywanie na wielu procesorach lub procesach. OpenMP działa na jednym komputerze i wykorzystuje wiele rdzeni procesora, natomiast MPI pozwala na podział pracy między różne komputery w sieci.

Połączenie OpenMP i MPI daje możliwość maksymalnego wykorzystania zasobów, zarówno na poziomie wielu komputerów, jak i poszczególnych rdzeni na każdym z nich.

W sprawozdaniu porównamy, jak każda z tych metod radzi sobie z obliczeniami, testując je na dużych zestawach danych i sprawdzając, która metoda jest najefektywniejsza.

Dobrano tablice 1024 na 1024 by była odpowiednia duża tablica 2d przy liczeniu wymiany ciepła.
Poniżej zaprezentowano wyniki dla obliczenia sekwencyjnego tego problemu. Epsilon mniejszy niż 0.00006 został uzyskany dla 181254 iteracji wykonanych obliczeń na tablicy 2d.

Sekwencyjnie programowi napisanemu tak by działał jak najszybciej i tak zajęło to 1473 sekundy, to około 25 min, więc bardzo dużo. Lecz można obliczyć, że 181254 iteracji na tablicy 1024 na 1024 to realnie 181254 * 1022 * 1022, czyli około 200 miliardów obliczeń (mnożenie i suma) na zmiennych typu double plus kolejne 200 miliardów obliczania epsilona, razem to będzie około 500 miliardów obliczeń zmiennoprzecinkowych. Także w takim przypadku zrozumiałe jest czemu nawet współczesnemu komputerowi zajmuje to tyle czasu.

```
okeanos-login1 PROJEKT/OPENMP> cat heat_seq.txt

 current eps = 0.691131 in 0 iteration
 current eps = 0.000599 in 10000 iteration
 current eps = 0.000352 in 20000 iteration
 current eps = 0.000257 in 30000 iteration
 current eps = 0.000206 in 40000 iteration
 current eps = 0.000173 in 50000 iteration
 current eps = 0.000149 in 60000 iteration
 current eps = 0.000132 in 70000 iteration
 current eps = 0.000119 in 80000 iteration
 current eps = 0.000109 in 90000 iteration
 current eps = 0.000100 in 100000 iteration
 current eps = 0.000093 in 110000 iteration
 current eps = 0.000086 in 120000 iteration
 current eps = 0.000081 in 130000 iteration
 current eps = 0.000076 in 140000 iteration
 current eps = 0.000071 in 150000 iteration
 current eps = 0.000067 in 160000 iteration
 current eps = 0.000064 in 170000 iteration
 current eps = 0.000060 in 180000 iteration
 Number of iterations = 181254
 final eps = 0.000060 Work took 1472.760042 seconds
1472.72user 0.00system 24:32.79elapsed 99%CPU (0avgtext+0avgdata 18568maxresident)k
176inputs+0outputs (1major+4222minor)pagefaults 0swaps
```

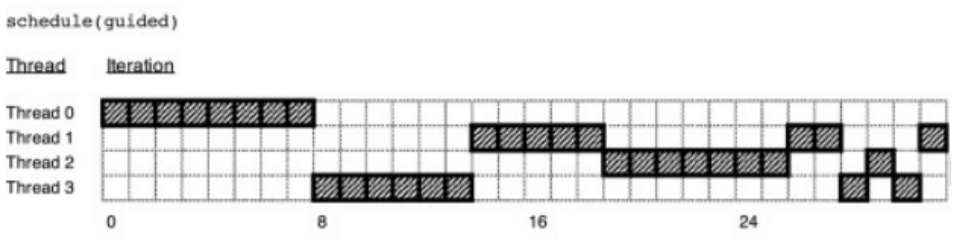
Najpierw jest wypełniana tablica wartościami brzegowymi (temperatura stałą), a później reszta wypełniana jest zerami. Tablice są statyczne, nie są alokowane dynamiczną pamięcią, bo na testach dawało to lekką przewagę. Tworzone są 2 jednakowe tablice, w których będą odbywać się obliczenia jest to zrobione po to, by nie było potrzeby kopiowania starych wyników, potrzebnych do następnej iteracji obliczeń. W ten sposób oszczędzamy bardzo dużo czasu na tym, że od razu mamy dostęp do wyników i nie trzeba ich nigdzie zapisywać. Obliczenia są wykonywane naprzemiennie raz pierwsza tablica dostaje nowe wyniki, czyli średnia temperaturę z wszystkich swoich sąsiadów, a później w następnej iteracji w 2 tablicy są wykonywane obliczenia średniej temperatury wykorzystując już uzyskane wyniki z poprzedniej iteracji z 1 tablicy i tak dalej i tak dalej, aż uzyskamy odpowiedni epsilon.

W przypadku obliczeń dla OpenMP zastosowano pragmę i start obliczeń od inicjalizacji tablic, ponieważ już przy tak dużych tablicach nawet wypełnienie ich zerami i wartościami zajmie sporo czasu

```
// Inicjalizacja tablic
#pragma omp parallel default(none) shared(arr, old_arr, temp)
{
    #pragma omp for schedule(guided)
    for (int i = 0; i < N; ++i) {
        fillRow(N, arr, i);
        fillRow(N, old_arr, i);
    }

    fillSides(arr);
    fillSides(old_arr);
```

Użyto do tego rozprowadzenia pomiędzy wątki pracy za sprawą dyrektywy guided.

Iteracje są najpierw dzielone na bloki o rozmiarze proporcjonalnym do liczby dostępnych wątków. Każdy wątek dostaje dużą porcję pracy na początku, a po jej zakończeniu pobiera kolejną, mniejszą porcję. Rozmiar kolejnych porcji stopniowo maleje, co pozwala lepiej równoważyć obciążenie między wątkami, zwłaszcza gdy zadania mają różne czasy wykonania. Dzięki temu wątki, które szybciej kończą pracę, mogą dynamicznie pobierać dodatkowe zadania, co minimalizuje czas oczekiwania pozostałych.



iterates not assigned yet	block assigned
32	$\lceil 32:4 \rceil = 8$
$32 - 8 = 24$	$\lceil 24:4 \rceil = 6$
$24 - 6 = 18$	$\lceil 18:4 \rceil = 5$
$18 - 5 = 13$	$\lceil 13:4 \rceil = 4$
$13 - 4 = 9$	$\lceil 9:4 \rceil = 3$
$9 - 3 = 6$	$\lceil 6:4 \rceil = 2$
$6 - 2 = 4$	$\lceil 4:4 \rceil = 1$
$4 - 1 = 3$	$\lceil 3:4 \rceil = 1$
$3 - 1 = 2$	$\lceil 2:4 \rceil = 1$
$2 - 1 = 1$	$\lceil 1:4 \rceil = 1$

Fig. from Peter S. Pacheco and Matthew Malensek (2022). An introduction to parallel programming

Później pragma jest używana w obliczeniach temperatury na płytce. Najpierw ustawiana jest bariera by każdy wątek zaczynał równo z każdym następną iterację, później używana jest pragma omp single, jest to użyte po to by pierwszy wątek, który dostanie się do zmiennej współdzielonej temp zmienił jej wartość na 0 na początku pętli, w przypadku braku czegoś takiego inny wątek mógłby później dojść do tej linijki, a inny już mógłby obliczać i dodawać do temp swoje obliczenia, a ten opóźniony by dostał się do zmiennej temp i ja znowu wyzerował

```
for (int t = 0; t < MAX_ITER && temp > EPS; ++t) {

    #pragma omp barrier
    #pragma omp single
    {
        temp = 0;
    }

    switch (t % 2 == 0) {
        case 0:
            #pragma omp for collapse(2) reduction(+:temp)
            for (int i = 1; i < N - 1; ++i) {
                for (int j = 1; j < N - 1; ++j) {
                    arr[i][j] = (old_arr[i + 1][j] + old_arr[i - 1][j] + old_arr[i][j + 1] + old_arr[i][j - 1]) / 4;
                    temp += (arr[i][j] - old_arr[i][j]) * (arr[i][j] - old_arr[i][j]);
                }
            }
            break;

        case 1:
            #pragma omp for collapse(2) reduction(+:temp)
            for (int i = 1; i < N - 1; ++i) {
                for (int j = 1; j < N - 1; ++j) {
                    old_arr[i][j] = (arr[i + 1][j] + arr[i - 1][j] + arr[i][j + 1] + arr[i][j - 1]) / 4;
                    temp += (arr[i][j] - old_arr[i][j]) * (arr[i][j] - old_arr[i][j]);
                }
            }
            break;
    }
```

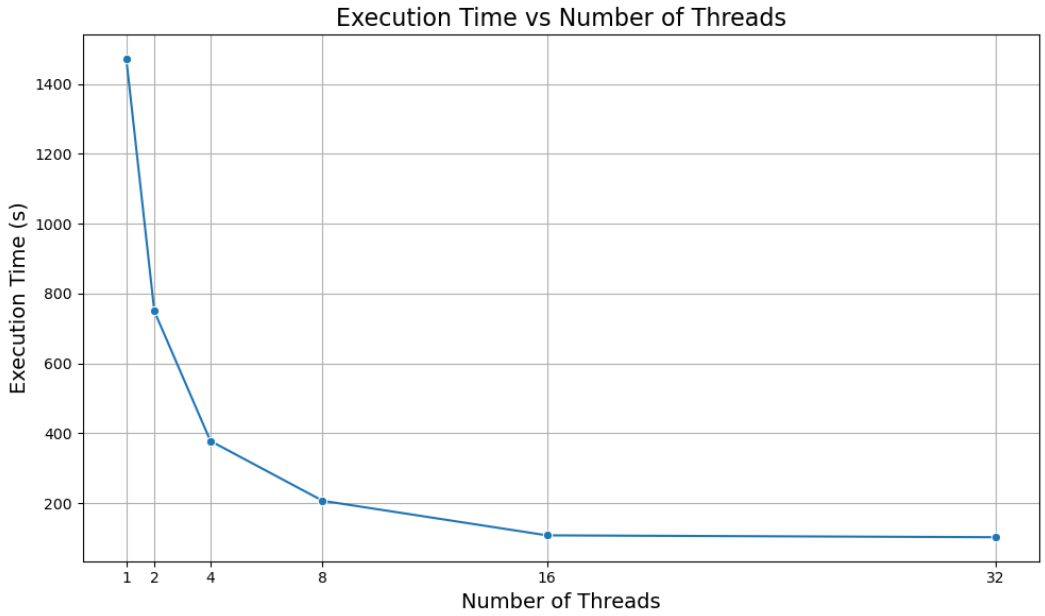
Później używana jest dyrektywa collapse(2) co powoduje, że rozdziela dla wątków podwójnego for'a tak jakby był 1 tablicą o długości (N-1) * (N-1) dzięki temu możemy użyć pragmy for na aż 2 pętlach, a nie tylko na 1. Robimy też redukcje, sumę za pomocą pragmy na temp, by jeszcze szybciej wykonywać obliczenia równoległe.

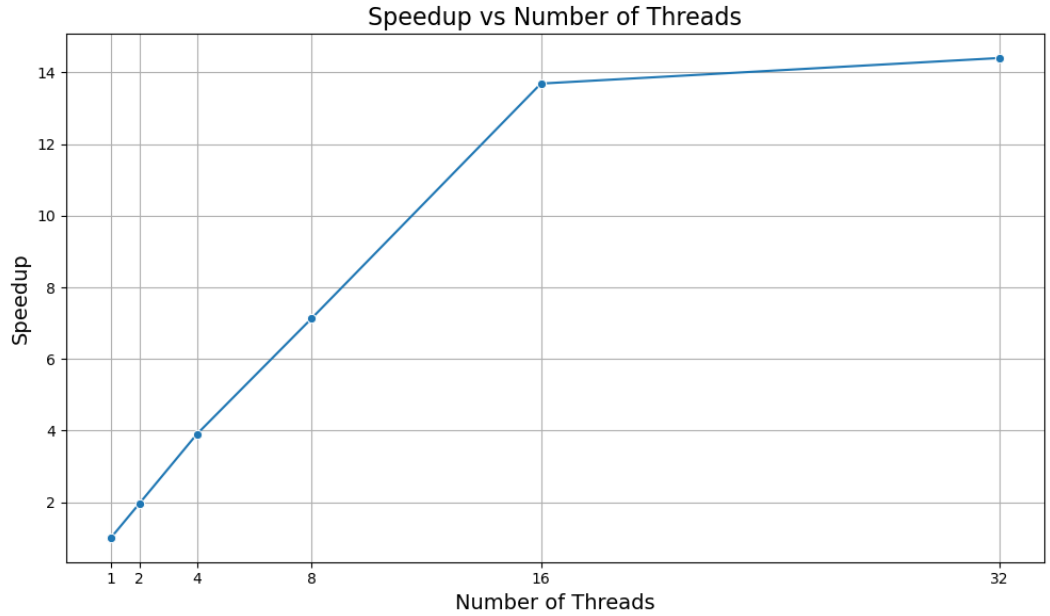
```
Uruchamianie programu z 8 watkami

 current eps = 0.691131 in 0 iteration
 current eps = 0.000599 in 10000 iteration
 current eps = 0.000352 in 20000 iteration
 current eps = 0.000257 in 30000 iteration
 current eps = 0.000206 in 40000 iteration
 current eps = 0.000173 in 50000 iteration
 current eps = 0.000149 in 60000 iteration
 current eps = 0.000132 in 70000 iteration
 current eps = 0.000119 in 80000 iteration
 current eps = 0.000109 in 90000 iteration
 current eps = 0.000100 in 100000 iteration
 current eps = 0.000093 in 110000 iteration
 current eps = 0.000086 in 120000 iteration
 current eps = 0.000081 in 130000 iteration
 current eps = 0.000076 in 140000 iteration
 current eps = 0.000071 in 150000 iteration
 current eps = 0.000067 in 160000 iteration
 current eps = 0.000064 in 170000 iteration
 current eps = 0.000060 in 180000 iteration
 Number of iterations = 181254
 final eps = 0.000060 Work took 206.602618 seconds
1652.48user 0.28system 3:26.60elapsed 799%CPU (0avgtext+0avgdata 18636maxresident)k
```

Przykładowy wynik przy użyciu 8 wątków za pomocą OpenMP. Iteracji musi być wiadomo tyle samo bo jest wykonywane dokładnie te samo obliczenie, natomiast czas został zmniejszony o ponad 7 razy! Czyli zamiast 25 minut, dzięki programowaniu równoległemu uzyskaliśmy wynik w zaledwie 3 i poł minuty.

Wyniki dla użycia OpenMP:

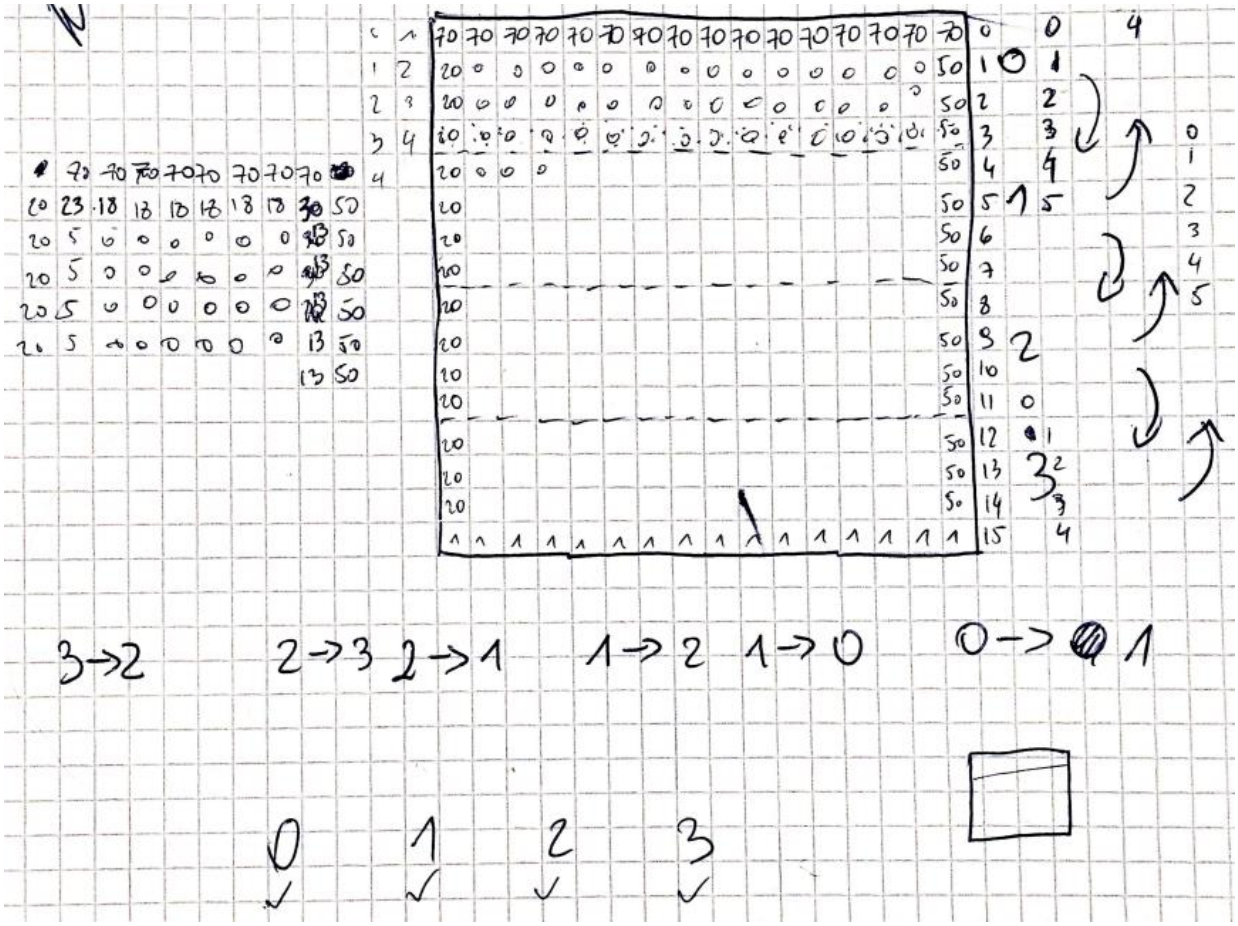
(1 wątek - 1472.760042 sek, 2 wątki - 749.682428 sek, 4 wątki - 377.874595 sek, 8 wątków - 206.602618 sek, 16 wątków - 107.577956 sek, 32 wątki - 102.243094 sek)

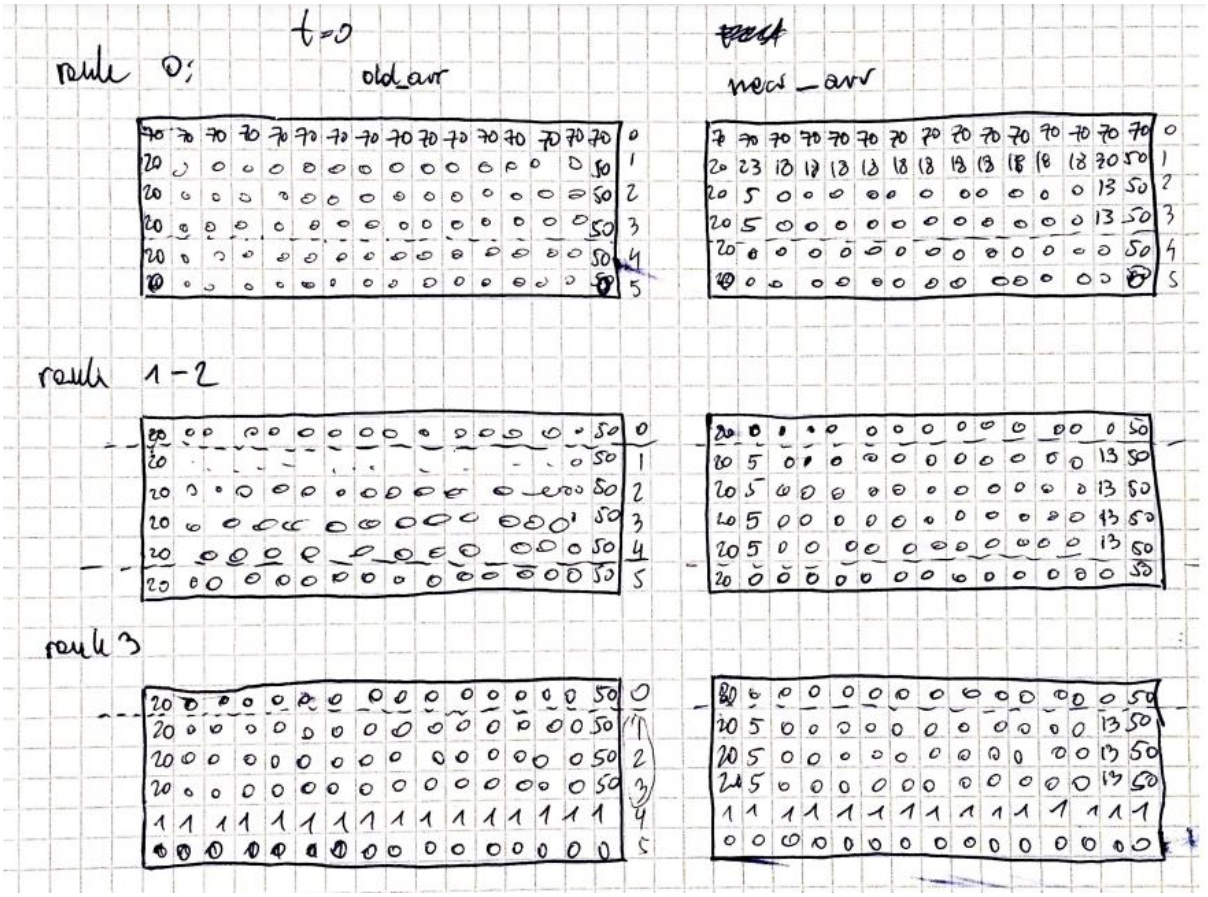




Jak widać do 4 wątków wartość speedup'u jest bardzo zbliżona do teoretycznej, czyli liczbie wątków, przy 8 już ta wartość jest mniejsza od teoretycznej. Przy 8, 16 wątkach mamy największą korzyść z użycia OpenMP jest przełamanie na wykresie czasu. Wyniki pokazują, że zwiększanie liczby wątków od 1 do 16 znacznie skraca czas obliczeń, co przekłada się na wyraźne przyspieszenie. Jednak przy 32 wątkach czas wykonania nie spada już znacząco w porównaniu do 16 wątków, co można przypisać kilku czynnikom. Wzrost liczby wątków zwiększa koszty synchronizacji i komunikacji między nimi, co może zmniejszyć korzyści z dalszego podziału pracy. Procesor może być też już gorzej zoptymalizowany na poziomie kompilatora do użycia większej liczby wątków (kompilacja na gcc)

W przypadku obliczeń dla MPI sprawa się ma inaczej, ponieważ obliczenia są wykonywane na wielu procesorach, a nie tak jak w OpenMp na 1. Przede wszystkim najbardziej kosztowna czasowo jest komunikacja pomiędzy procesami, każda wymiana danych oraz porozumiewanie się ze sobą przechodzi przez sieć, więc jest powolna w porównaniu do przesyłu danych na płycie głównej. Moim pomysłem jak stworzyć, jak najszybszy program liczący wymianę ciepła w MPI. Jest to by każdy oddzielny proces tworzył swoją lokalna mała tablice (duża tablica podzielona na liczbę procesów), a wymieniał się danymi na brzegach z sąsiadami. Proces zerowy i ostatni są szczególnymi przypadkami, ponieważ mają tylko 1 sąsiada, reszta procesów zachowuje się praktycznie tak samo. Także każdy proces ma tablice [1024/n_proc + 2] [1024] wymiana pomiędzy procesami jest tablicy 1 rzędu 1024 wartości. Tak samo jak w OpenMp i sekwencyjnym użyte są 2 tablice, które są zamieniane w zależności od iteracji.





Po każdej pojedynczej iteracji zliczany jest globalny epsilon za pomocą funkcji MPI Allreduce i sprawdzane jest, czy osiągnął epsilon docelową wartość

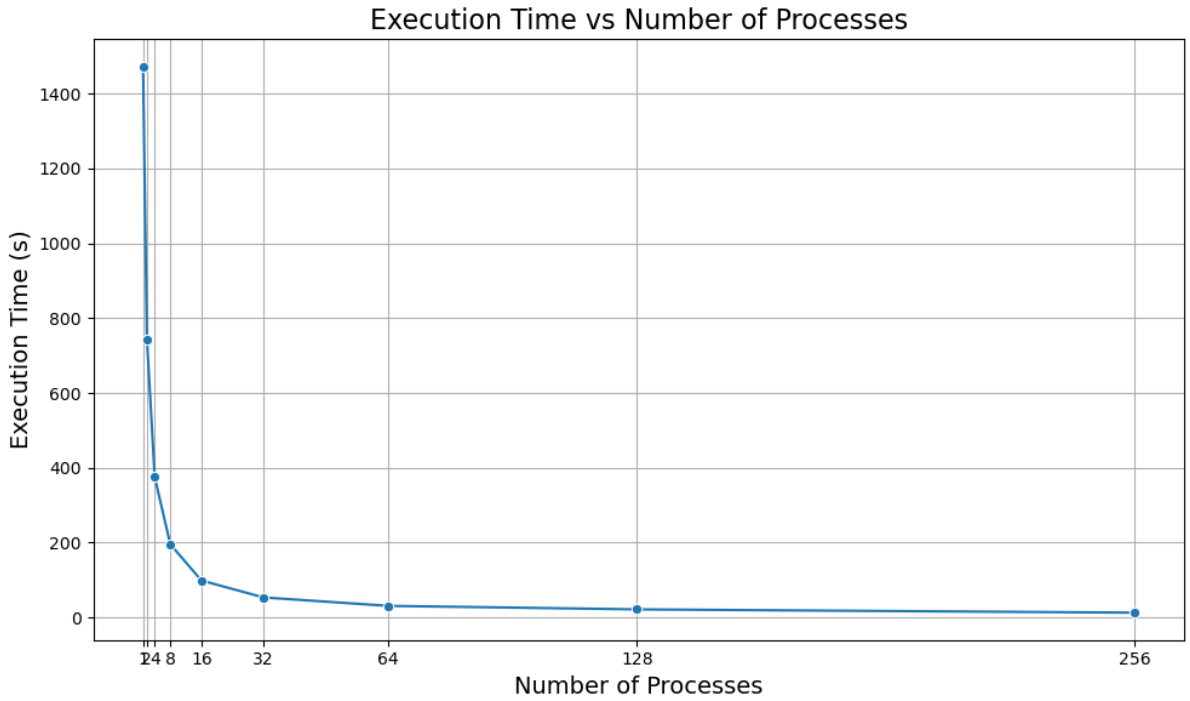
```
// Redukcja globalna dla sprawdzenia konwergencji
MPI_Allreduce(&local_error, &global_error, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
global_error = sqrt(global_error / ((N-2) * (N-2)));
```

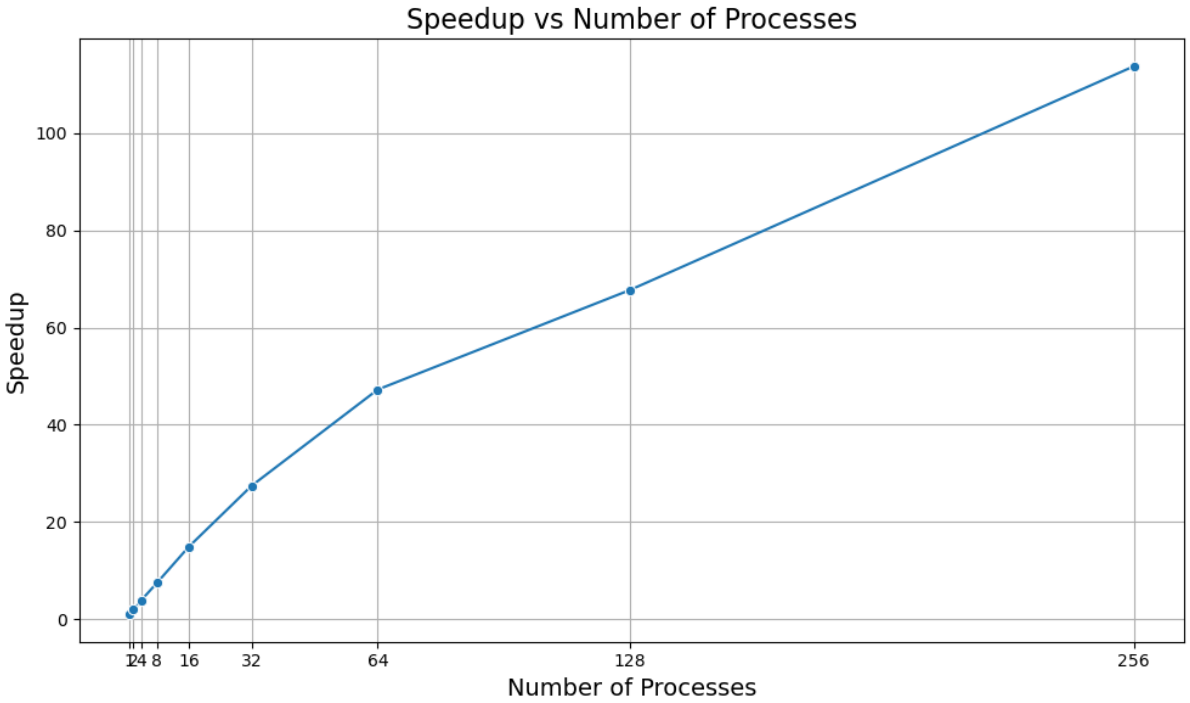
```
okeanos-login1 PROJEKT/MPI> cat MPI_nb_4Proc.out
Iteration 0, global error: 0.691131
Iteration 10000, global error: 0.000599
Iteration 20000, global error: 0.000352
Iteration 30000, global error: 0.000257
Iteration 40000, global error: 0.000206
Iteration 50000, global error: 0.000173
Iteration 60000, global error: 0.000149
Iteration 70000, global error: 0.000132
Iteration 80000, global error: 0.000119
Iteration 90000, global error: 0.000109
Iteration 100000, global error: 0.000100
Iteration 110000, global error: 0.000093
Iteration 120000, global error: 0.000086
Iteration 130000, global error: 0.000081
Iteration 140000, global error: 0.000076
Iteration 150000, global error: 0.000071
Iteration 160000, global error: 0.000067
Iteration 170000, global error: 0.000064
Iteration 180000, global error: 0.000060
Converged after 181254 iterations, final error: 0.000060
Total execution time: 375.256965 seconds
```

Przykładowe wyniki dla 4 procesów

Wyniki dla MPI:
(2 procesory - 743.745835 sek, 4 proc. - 375.256965, 8 proc. - 195.904888 sek, 16 proc. - 98.978608 sek, 32 proc. - 53.622652 sek, 64 proc. - 31.168730 sek, 128 proc - 21.745246 sek, 256 proc - 12.942392 sek)

Dla liczby procesorów od 2 do 16 używałem tyle node'ów co procesorów (1 proces per node), a dla 32 i powyżej 16 node'ów i kolejno 2,4,8 i 16 procesów per node.



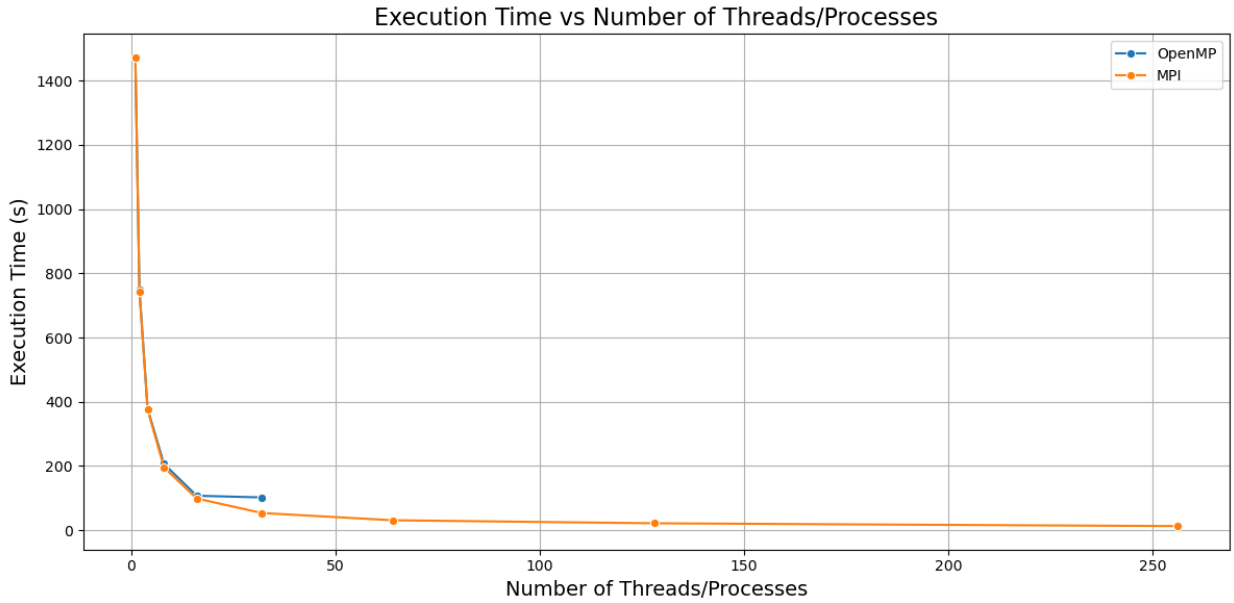


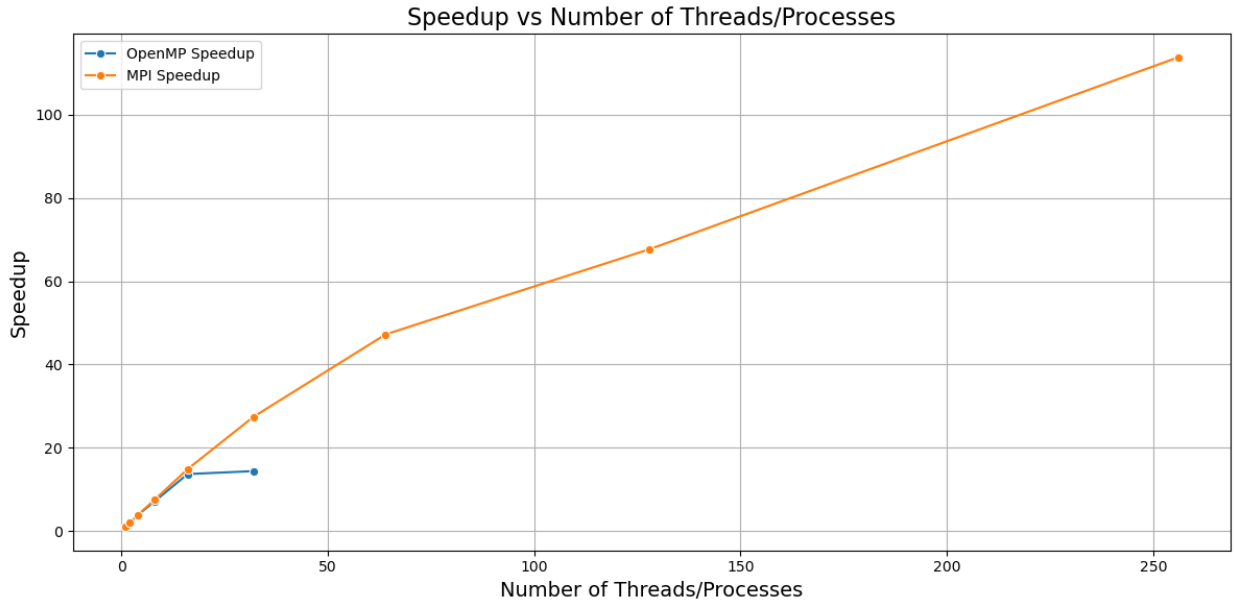
Jak widzimy, tutaj jest moc optymalizacji MPI dla procesów powyżej 16, możemy uzyskać tak genialny czas jak 13 sekund i speedup rzędu 115! Naprawdę zaskakujący wynik i fakt, że MPI tak dobrze się skaluje na więcej procesów.

```
okeanos-login2 PROJEKT/MPI> cat MPI_nb_256Proc.out
Iteration 0, global error: 0.691131
Iteration 10000, global error: 0.000599
Iteration 20000, global error: 0.000352
Iteration 30000, global error: 0.000257
Iteration 40000, global error: 0.000206
Iteration 50000, global error: 0.000173
Iteration 60000, global error: 0.000149
Iteration 70000, global error: 0.000132
Iteration 80000, global error: 0.000119
Iteration 90000, global error: 0.000109
Iteration 100000, global error: 0.000100
Iteration 110000, global error: 0.000093
Iteration 120000, global error: 0.000086
Iteration 130000, global error: 0.000081
Iteration 140000, global error: 0.000076
Iteration 150000, global error: 0.000071
Iteration 160000, global error: 0.000067
Iteration 170000, global error: 0.000064
Iteration 180000, global error: 0.000060
Converged after 181254 iterations, final error: 0.000060
Total execution time: 12.942392 seconds
```

Wyniki wskazują, że MPI wykazuje lepsze skalowanie w porównaniu do OpenMP, zwłaszcza przy dużej liczbie procesów. MPI efektywnie zarządza równoległymi obliczeniami na wielu węzłach, co pozwala na znaczące skrócenie czasu wykonania przy zwiększaniu liczby procesów. To dobra skalowalność wynika z zaawansowanych technik komunikacji międzyprocesowej, które minimalizują koszty komunikacji i optymalizują transfery danych. Z kolei OpenMP, które działa na pojedynczym węźle, doświadcza spadku efektywności przy zwiększaniu liczby wątków z powodu rosnących kosztów synchronizacji i lokalności pamięci. W miarę dodawania wątków, koszty związane z zarządzaniem i synchronizacją stają się bardziej znaczące, co ogranicza dalsze przyspieszenie. Ponadto, liczba rdzeni na jednym węźle może stać się wąskim gardłem w porównaniu do liczby procesów w systemie MPI. Dla dużych problemów rozproszonych, MPI jest bardziej efektywne, co przekłada się na lepszą wydajność przy dużej liczbie procesów. W rezultacie, MPI jest lepiej przystosowane do skalowania w środowiskach o dużej liczbie procesów, podczas gdy OpenMP lepiej sprawdza się w kontekście mniejszych, jednordzeniowych obliczeń.

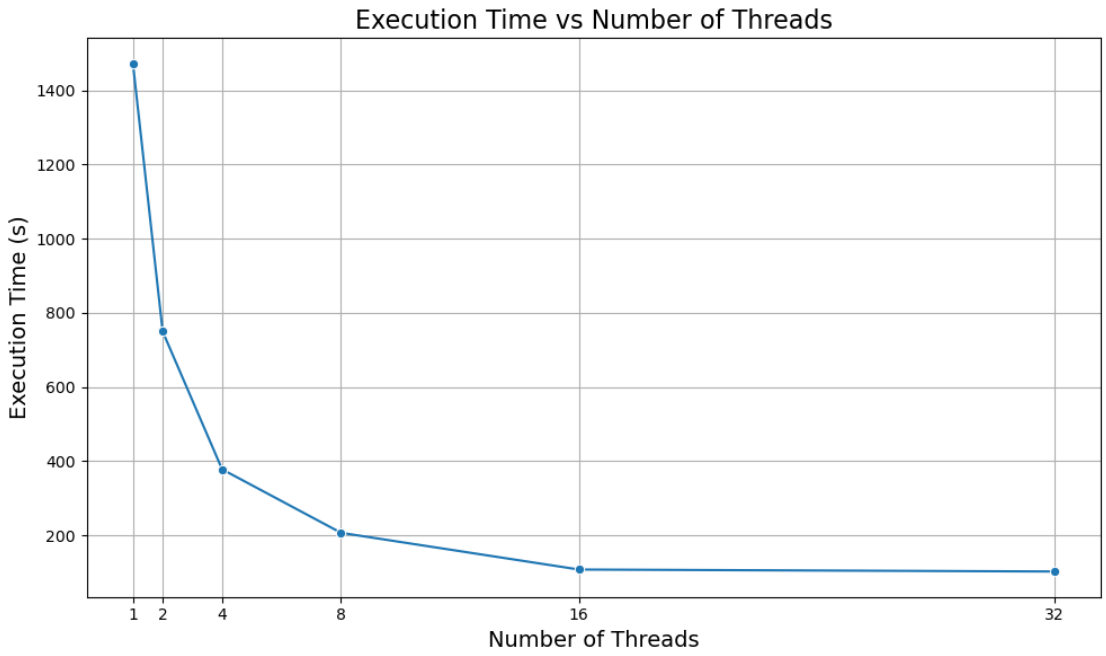
Wyniki MPI vs OpenMP:

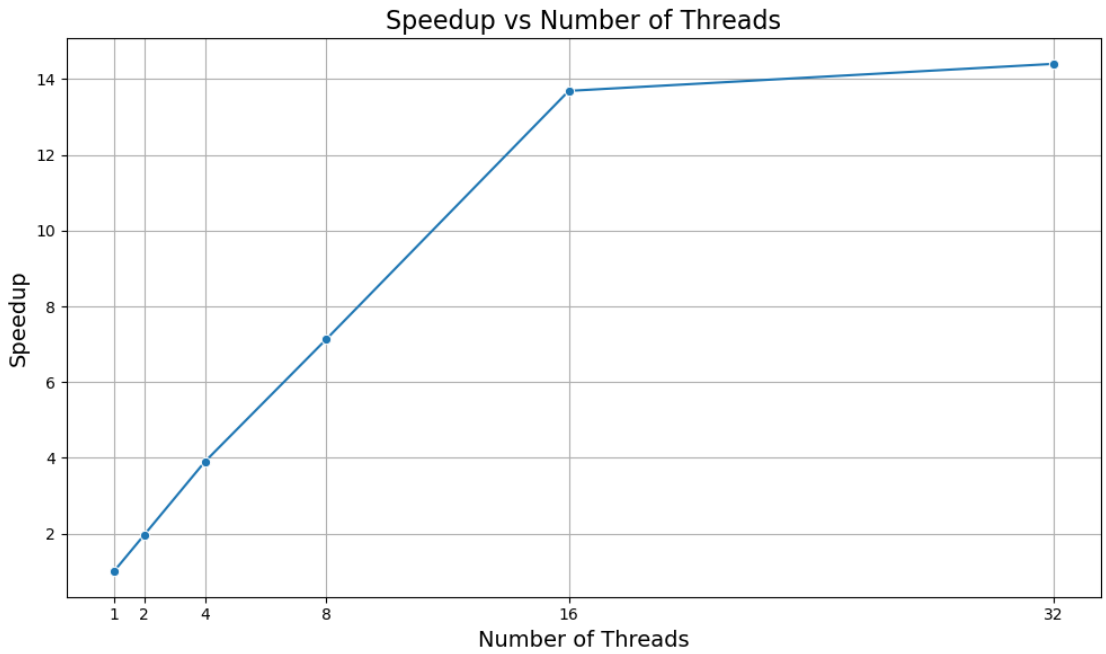




Widać, że OpenMP był jak równy z równym (pomimo, że dalej wolniejszy - liczba procesów vs wątków) aż do 16 wątków, później jednak MPI odskakuje i dalej jest bardzo dobrze skalowalny z większą liczbą procesów, OpenMP dla większej liczby wątków nie. MPI pomimo, że trudniejszy w napisaniu przynosi bardzo duże zyski końcowe.

Obliczenia hybrydowe dla połączenia MPI z OpenMP okazały się klęską, nie potrafiłem uzyskać odpowiedniej prędkości, próbowałem na 2 różne sposoby, na początku myślałem, że połączenie OpenMP z MPI jest banalne, wystarczy odpalić Multi-threading na już istniejącym kodzie MPI i gotowe. Nic bardziej mylnego, czysto teoretycznie odpalało, ale z niesatysfakcjonującymi wynikami. Wyniki były około 3-4 razy wolniejsze niż by to było sensowne w teorii (speedup to wątki razy procesy).



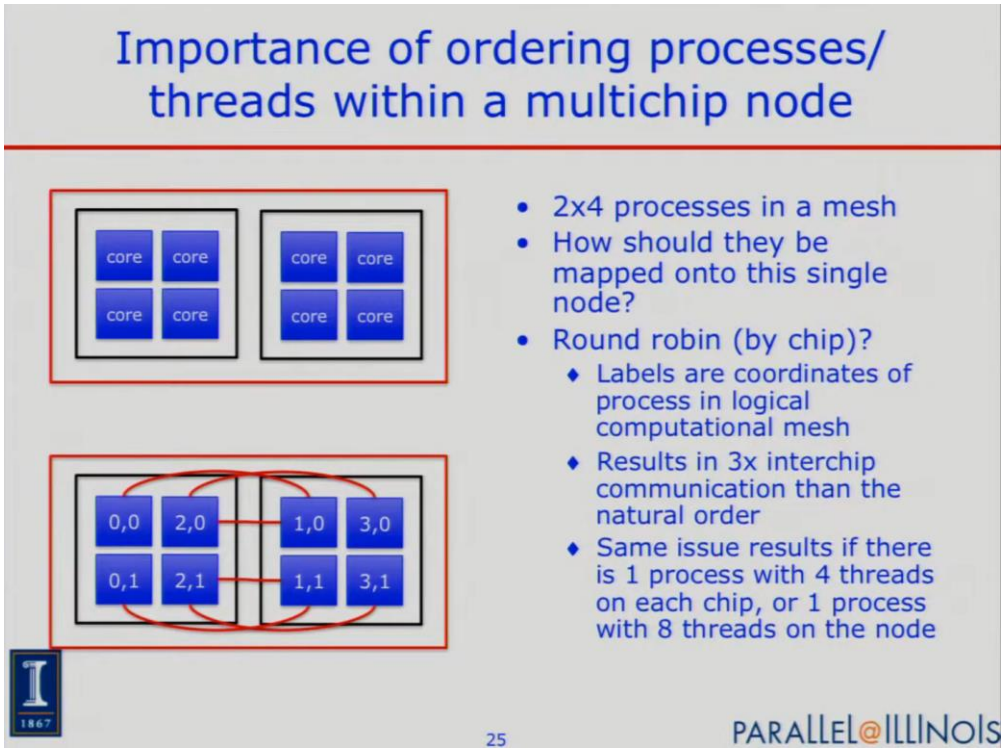


Wyniki dla MPI + OpenMP dla użycia 4 procesów z MPI i różnej liczby wątków, jak widać, wyniki są tylko trochę lepsze niż dla samego OpenMP.

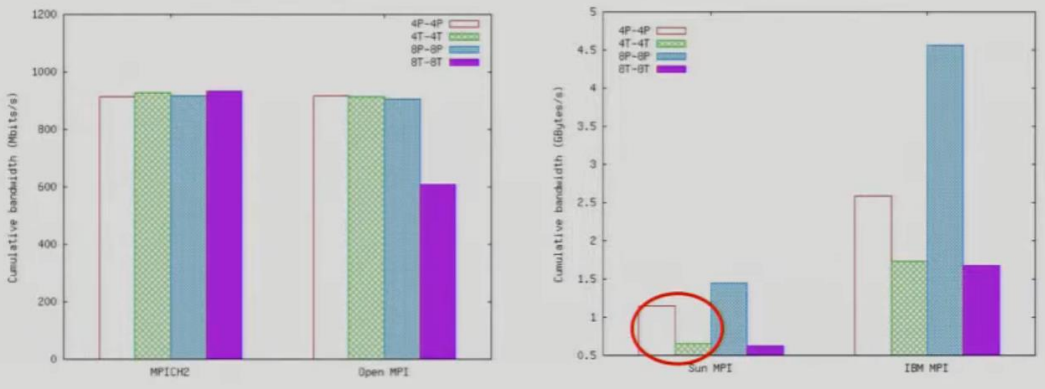
```
okeanos-login2 PROJEKT/OPENMP_MPI> cat MPI_DYNAMIC_16Proc_16thr.out
Iteration 0, global error: 0.691131
Iteration 10000, global error: 0.000599
Iteration 20000, global error: 0.000352
Iteration 30000, global error: 0.000257
Iteration 40000, global error: 0.000206
Iteration 50000, global error: 0.000173
Iteration 60000, global error: 0.000149
Iteration 70000, global error: 0.000132
Iteration 80000, global error: 0.000119
Iteration 90000, global error: 0.000109
Iteration 100000, global error: 0.000100
Iteration 110000, global error: 0.000093
Iteration 120000, global error: 0.000086
Iteration 130000, global error: 0.000081
Iteration 140000, global error: 0.000076
Iteration 150000, global error: 0.000071
Iteration 160000, global error: 0.000067
Iteration 170000, global error: 0.000064
Iteration 180000, global error: 0.000060
Converged after 181254 iterations, final error: 0.000060
Total execution time: 32.775673 seconds
```

Dla 16 procesów i 16 wątków uzyskujemy ładny wynik, ale zgodnie z teoria powinien być około 6 sekund, plus jak widzieliśmy lepiej i tak zrobić to samo samym MPI i podać większą liczbę procesów (Lepsza optymalizacja samego MPI)

Może być to spowodowane walka o środki pomiędzy MPI, a OpenMP (overhead). Kolejnym powodem może być fakt, że pragme wywoływałem w pętli iteracyjnej przez co duża ilość wywoływań kolejnych omp pragma parrallel mogła spowalniać wynik (Próbowałem też z użyciem MPI_THREAD_SERIALIZED i użyciem omp pragma single do wywoływań calli MPI, ale nie potrafiłem tego skompilować). Kolejnym możliwym wytłumaczeniem jest to co znalazłem w wykładzie uniwersytetu Illinois, że rozkład chipów ma znaczenie podczas komunikacji i 1 informacja może w najgorszym wypadku 3 razy dłużej przejść niż powinno to być normalnie (Mogłoby to tłumaczyć czemu mój program jest o około 3 razy wolniejszy niż powinien)



Concurrent Bandwidth Test



Lesson: Its hard to provide full performance from threads

(Recent results on current platforms show similar behavior)

PARALLEL@ILLINOIS

Tutaj widzimy, jak jest ciężko z przesyłem danych. Zbyt długo robiłem zadanie dla ambitnych i go nie skończyłem, z tego powodu też wykonałem 1 i drugie zadnie zwykłe nie dla ambitnych.

Zadanie S1

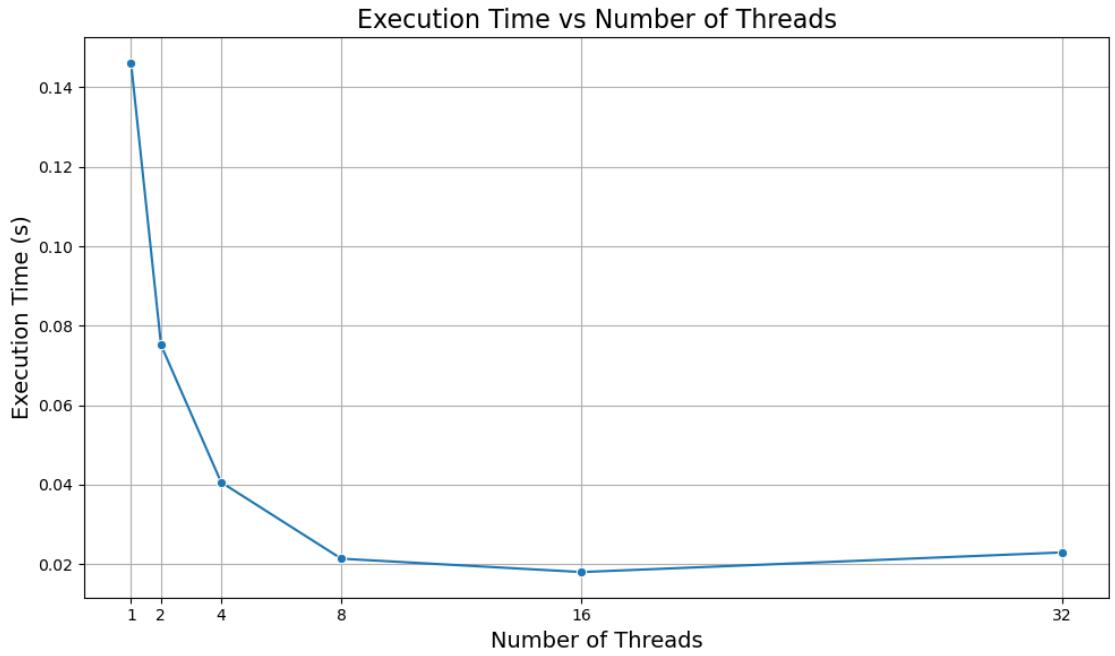
Napisać równoległą implementację operacji dodawania wektora B i wektora C. Wynik zapisywany jest w wektorze A, tzn.

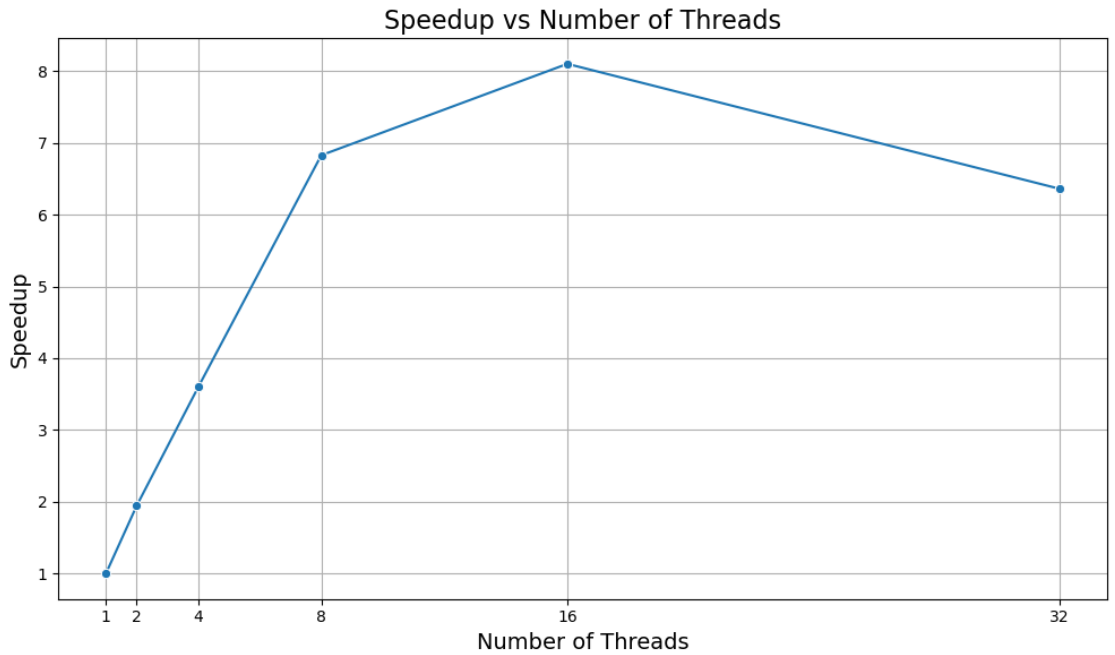
$$A[i]=B[i]+C[i] \text{ dla } i=0,1,\ldots,n-1 \text{ oraz } n \geq 2^{20}=1048576.$$

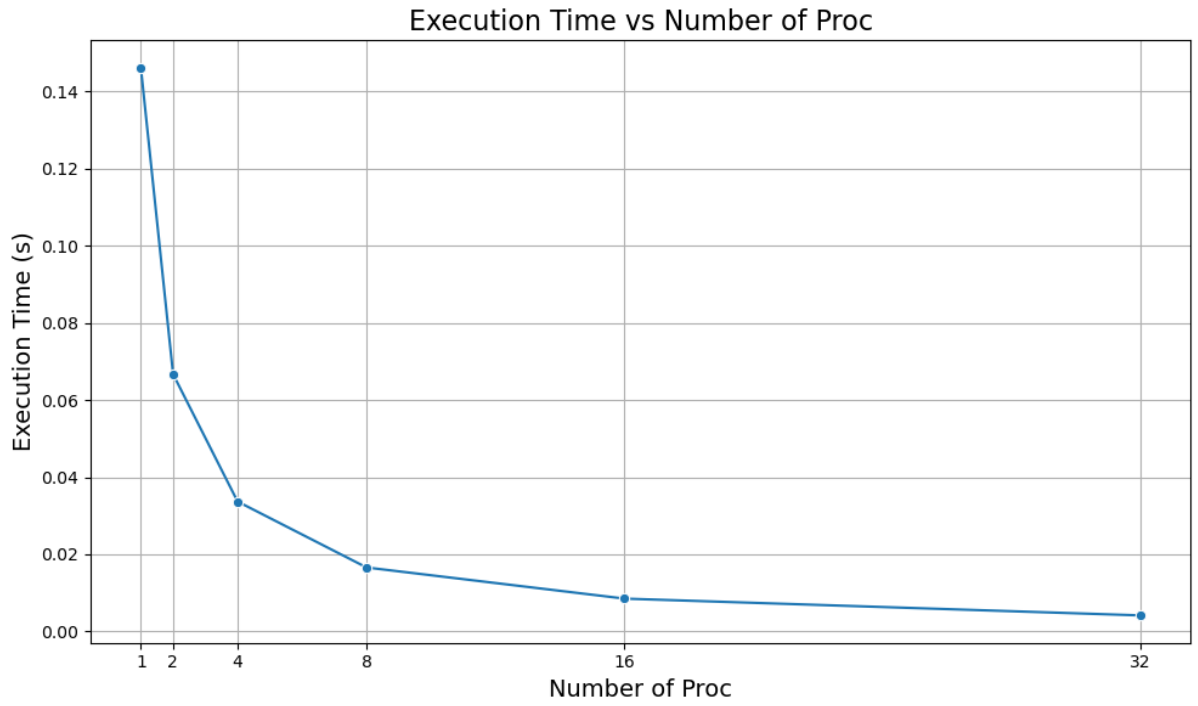
Wektory B i C należy wypełnić losowymi liczbami zmiennopozycyjnymi z przedziału [0,1].

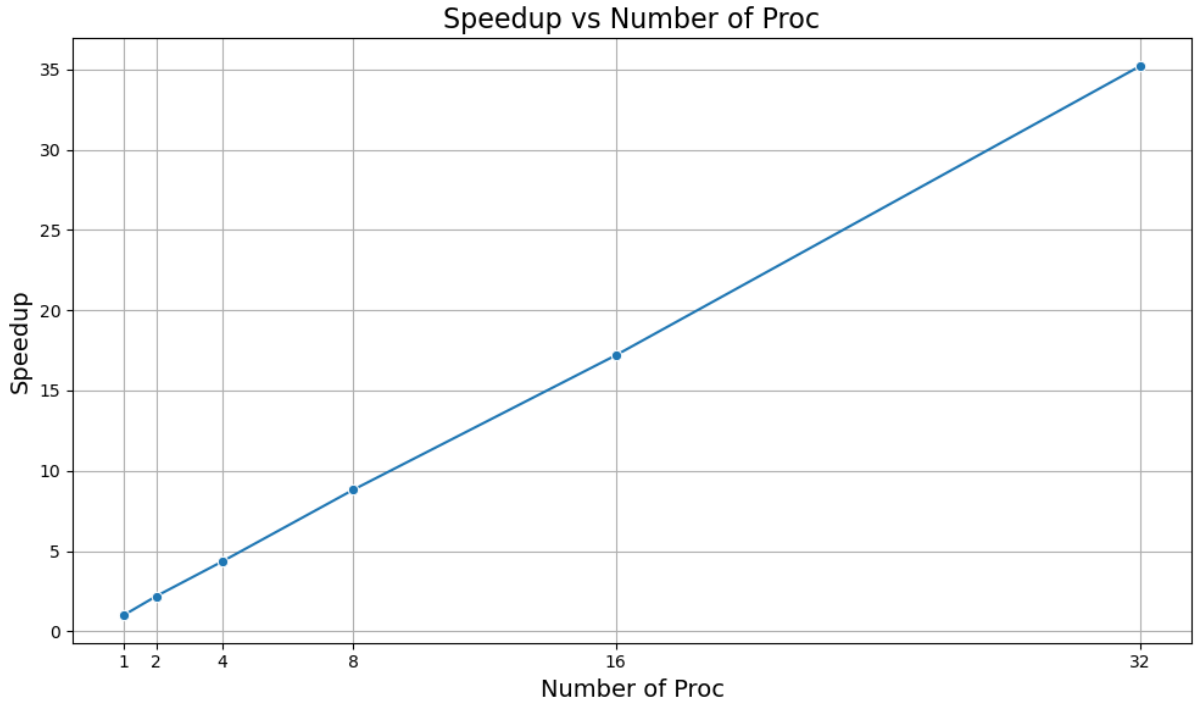
Jako wynik należy wypisać tylko A[0], B[0], C[0] i A[n-1], B[n-1], C[n-1]. Wynik powinien być wypisany tylko jeden raz.

Należy napisać dwa programy: pierwszy używając OpenMP oraz drugi używając MPI albo PCJ.

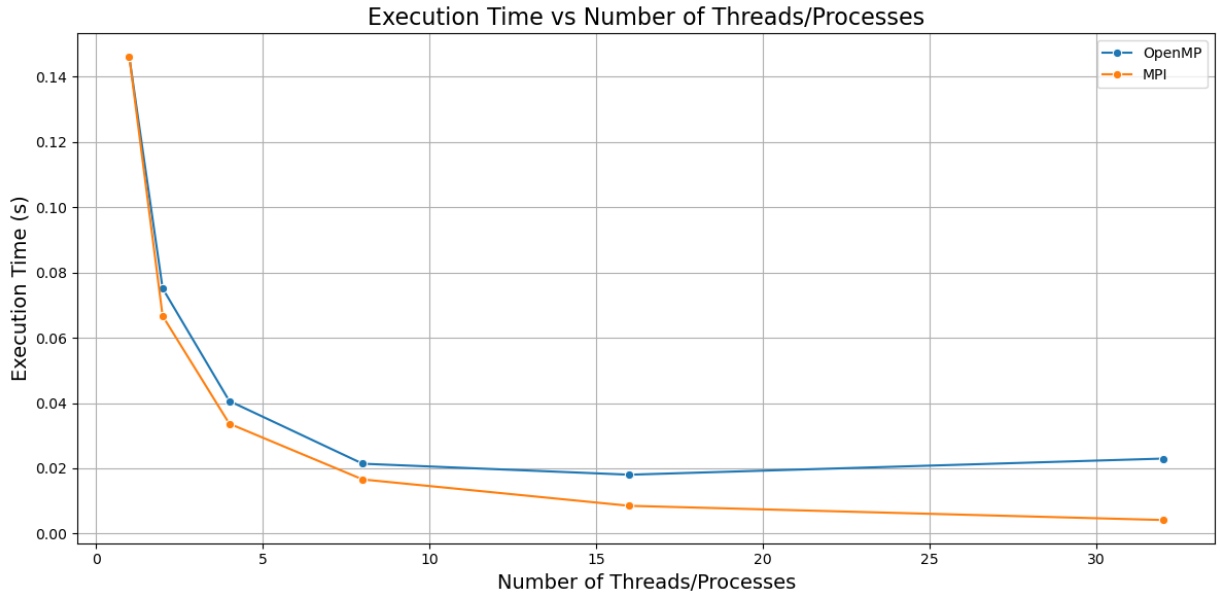


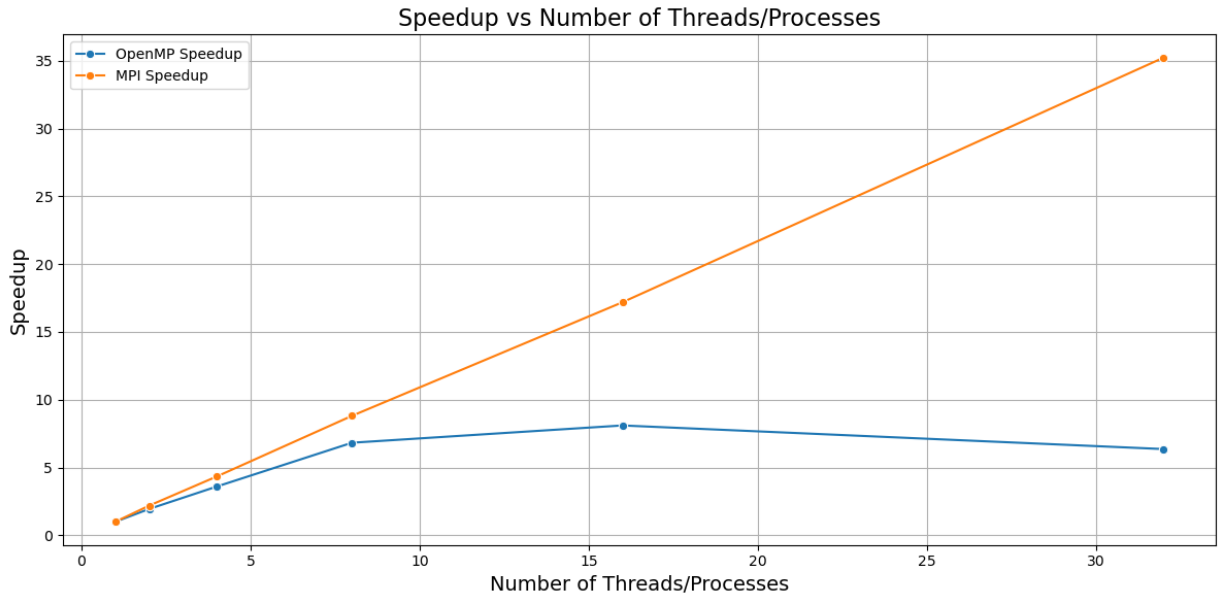






MPI o wiele lepiej się skalował, ponieważ nie używałem żadnej komunikacji. Jest też z tego powodu szybszy od OpenMP. MPI stworzyłem podobnie co zadanie dla ambitnych podzielilem tablice na lokalne i każdy proces miał lokalna mniejszą tablicę, na której wykonywał obliczenia. Zadanie nie prosiło nas, żeby jakikolwiek proces miał wszystkie dane, więc proces zerowy i ostatni po prostu wyświetliły swoje wyniki.





Na powyższych wykresach widać, że MPI był zdecydowanie lepszy dla tego zadania, skalował się całkowicie i prawie osiągał teoretyczne wyniki speedupu. Przykładowe wyniki:

```
Uruchamianie programu z 16 watkami
A[0] = 0.789217, B[0] = 0.286843, C[0] = 0.502374
A[6048575] = 1.430944, B[6048575] = 0.607463, C[6048575] = 0.823481
Elapsed time: 0.018037 seconds
0.24user 0.05system 0:00.02elapsed 1192%CPU (0avgtext+0avgdata 1169
136inputs+0outputs (1major+29693minor)pagefaults 0swaps
Uruchamianie programu z 32 watkami
A[0] = 0.887889, B[0] = 0.836424, C[0] = 0.051466
A[6048575] = 1.628289, B[6048575] = 0.706136, C[6048575] = 0.922154
Elapsed time: 0.022973 seconds
0.35user 0.11system 0:00.03elapsed 1512%CPU (0avgtext+0avgdata 1168
136inputs+0outputs (1major+29727minor)pagefaults 0swaps
okeanos-login2 PROJEKT/ZAD1>
okeanos-login2 PROJEKT/ZAD1> ls
zad1.c              zad1_MPI_2proc.out   zad1_MPI_8proc.out  zad1_
zad1.exe            zad1_MPI_32proc.out  zad1_MPI.c          zad1_
zad1_MPI_16proc.out  zad1_MPI_4proc.out   zad1_MPI.exe        zad1_
okeanos-login2 PROJEKT/ZAD1> cat zad1_MPI_
zad1_MPI_16proc.out  zad1_MPI_2proc.out   zad1_MPI_32proc.out  zad1_
okeanos-login2 PROJEKT/ZAD1> cat zad1_MPI_2proc.out
Czas wykonania operacji: 0.066685 sekund
Proces 0: A[0] = 1.050557, B[0] = 0.959376, C[0] = 0.091181
Proces 1: A[n-1] = 1.041083, B[n-1] = 0.401106, C[n-1] = 0.639978
okeanos-login2 PROJEKT/ZAD1> cat zad1_MPI_4proc.out
Czas wykonania operacji: 0.033675 sekund
Proces 0: A[0] = 0.926928, B[0] = 0.354987, C[0] = 0.571941
Proces 3: A[n-1] = 0.783854, B[n-1] = 0.701281, C[n-1] = 0.082574
```

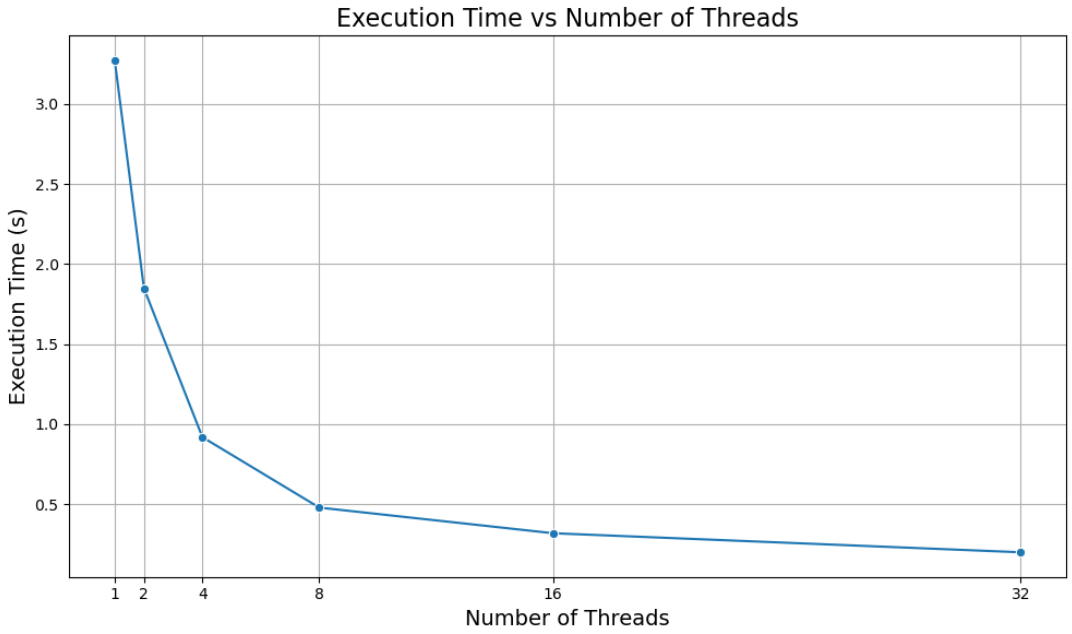
Zadanie S2

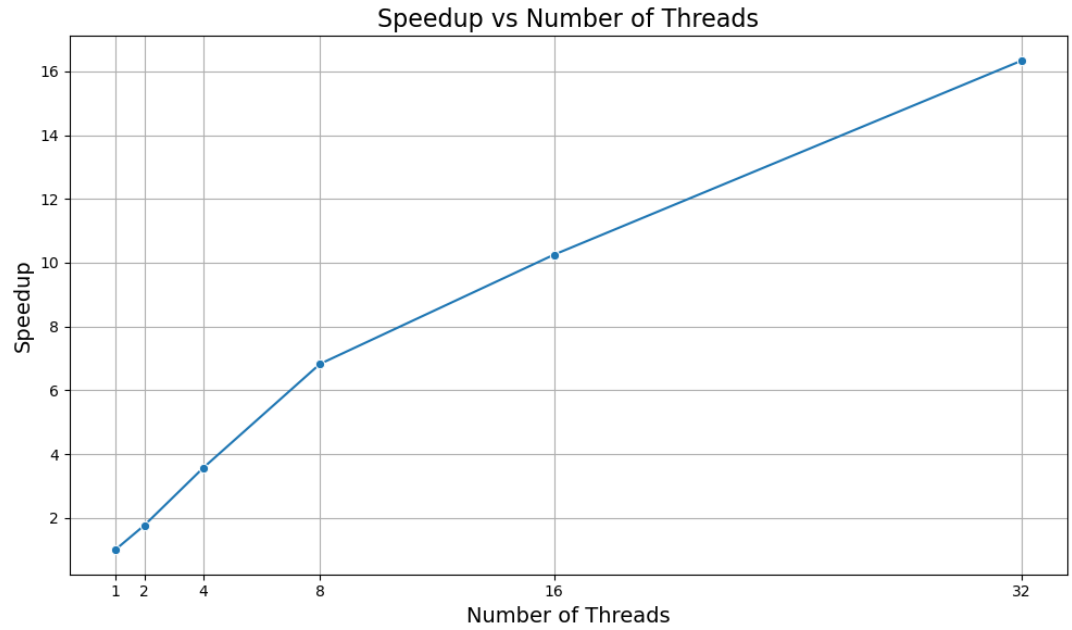
Napisać program równoległy, który wyznaczy tzw. normę drugą wektora A o długości n. Tablicę przechowującą wektor należy wypełnić losowymi liczbami zmiennopozycyjnymi z przedziału [0,1]. Wynik należy wypisać jednokrotnie na standardowe wyjście.

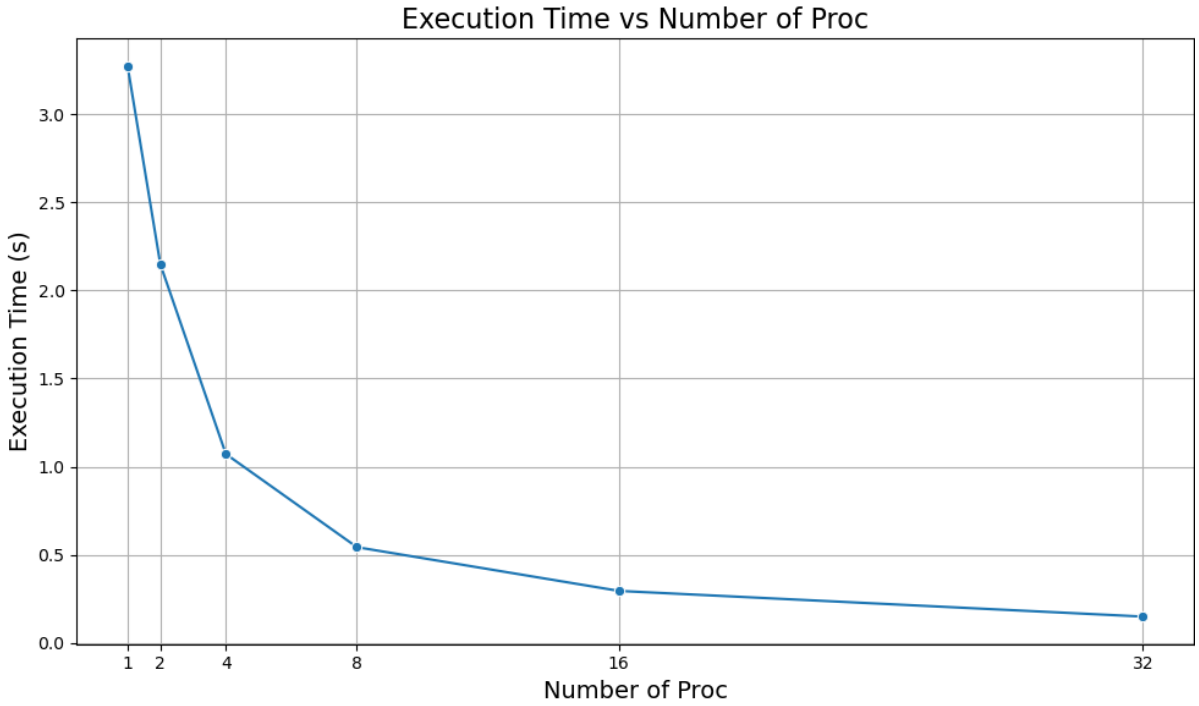
Norma druga wektora A długości n jest zdefiniowana jako

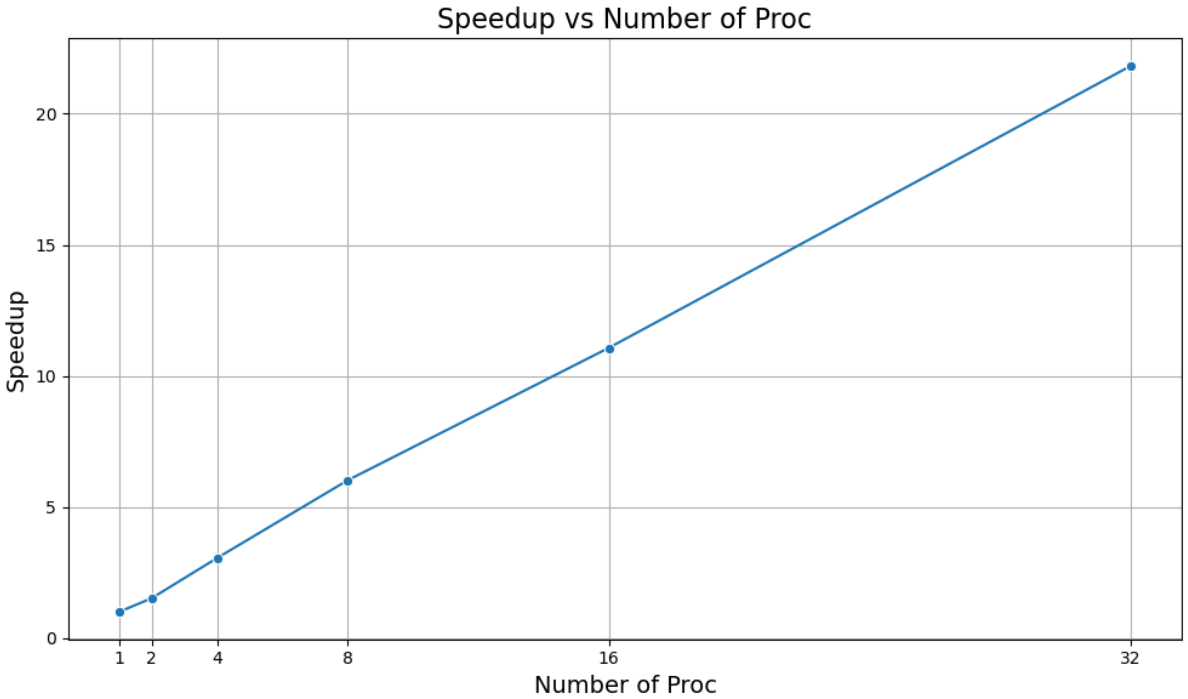
$$\sum i=0n-1A[i]2=A[0]2+A[1]2+...+A[n-1]2.$$

Należy dobrać n na tyle duże, by program wykonywany na jednym wątku/procesie trwał kilka sekund.

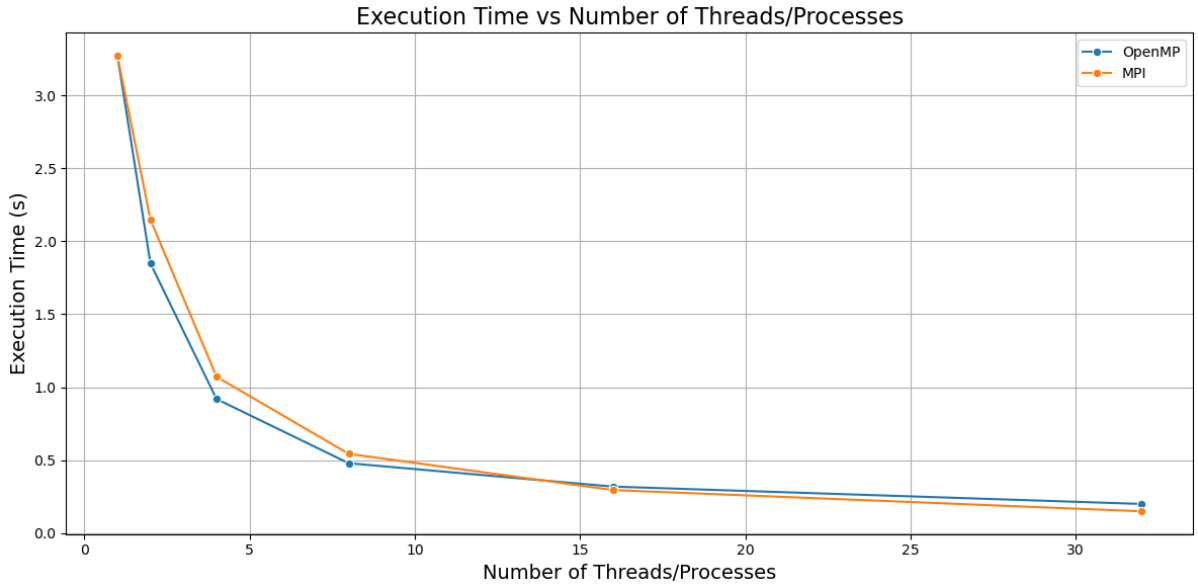


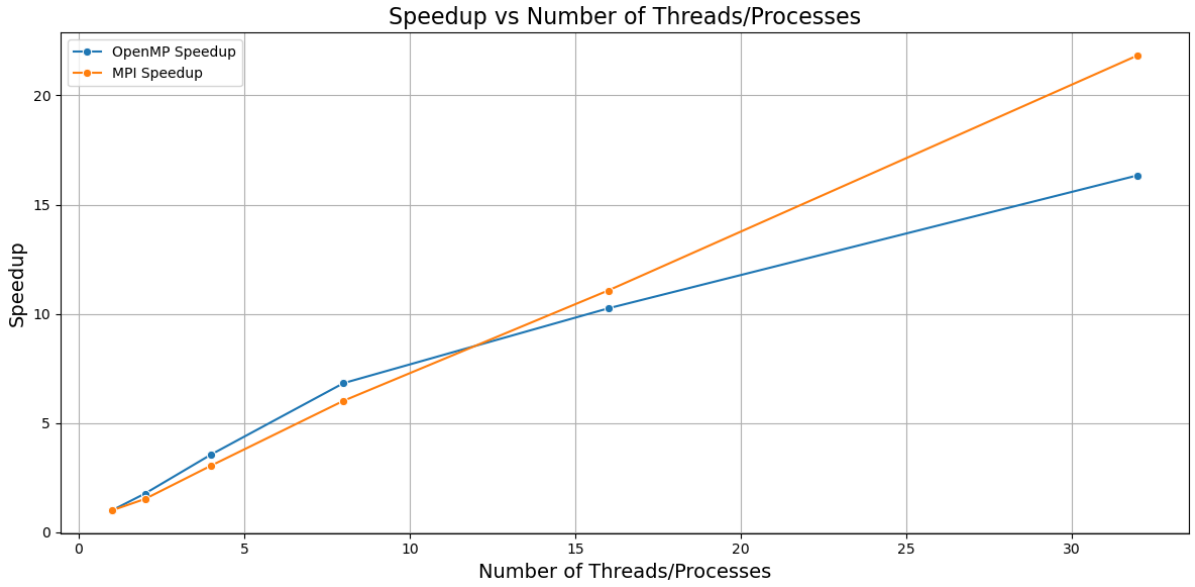






Zadanie 2 wykonałem analogicznie co dla ambitnych i S1, czyli w MPI użyłem tablic lokalnych, lecz w porównaniu do S1 tutaj na samym końcu musiałem zliczyć wszystkie wyniki z każdego z procesów za pomocą all reduce. To zaważa na prędkości szczególnie dla tak szybko wykonywanego programu. Komunikacja tutaj nam zwiększa czas, lecz MPI dalej wygrywa, jeżeli chodzi o skalowalność i im więcej procesów tym na jej korzyść.





Tak jak wspomniałem wyżej OpenMp wygrywa z MPI do około 16 wątków, później MPI ma przewagę skalowalności i jest szybsze od 16 wzwyż. Przykładowe wyniki:

```
Uruchamianie programu z 16 watkami
Norma druga wektora wynosi: 8164.965785
Elapsed time: 0.319246 seconds
3.46user 1.16system 0:00.39elapsed 1173%CPU (0avgtex
176inputs+0outputs (1major+586097minor)pagefaults 0s
Uruchamianie programu z 32 watkami
Norma druga wektora wynosi: 8164.965752
Elapsed time: 0.200280 seconds
3.85user 1.72system 0:00.27elapsed 2045%CPU (0avgtex
136inputs+0outputs (1major+586131minor)pagefaults 0s
okeanos-login2 PROJEKT/ZAD2>
okeanos-login2 PROJEKT/ZAD2> ls
zad1_OpenMP.out      zad2_MPI_32proc.out  zad2_mpi.c
zad2_MPI_16proc.out  zad2_MPI_4proc.out   zad2_mpi.e
zad2_MPI_2proc.out   zad2_MPI_8proc.out   zad2_mpi.s
okeanos-login2 PROJEKT/ZAD2> cat zad2_MPI_2proc.out
Norma druga wektora wynosi: 8164.995142
Elapsed time: 2.147929 seconds
okeanos-login2 PROJEKT/ZAD2> cat zad2_MPI_4proc.out
Norma druga wektora wynosi: 8164.699766
Elapsed time: 1.072135 seconds
```

Bibliografia:

1. https://youtu.be/2fpLLPQmjSs?feature=shared

2. https://stackoverflow.com/questions/3973665/how-do-i-use-rand-r-and-how-do-i-use-it-in-a-thread-safe-way

3.https://www.dcc.fc.up.pt/~ricroc/aulas/1516/cp/apontamentos/slides_mpi_openmp.pdf

Kody i slurm użyty do powyższego sprawozdania zostaną napisane na kolejnych stronach.

S1 OPENMP:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define N 6048576

int main() {
  float *B = malloc(N * sizeof(float));
  float *C = malloc(N * sizeof(float));
  float *A = malloc(N * sizeof(float));

  if (!B || !C || !A) {
    fprintf(stderr, "Memory allocation failed\n");
    return 1;
  }

  // Tablica nasionek dla wektorów B i C
  unsigned int *seeds_B = malloc(N * sizeof(unsigned int));
  unsigned int *seeds_C = malloc(N * sizeof(unsigned int));

  if (!seeds_B || !seeds_C) {
    fprintf(stderr, "Memory allocation for seeds failed\n");
    free(B);
    free(C);
    free(A);
    return 1;
  }
  double start_time, end_time;
  // Start measuring time
  start_time = omp_get_wtime();
  // Initialize seeds for B and C with different values
  #pragma omp parallel default(none) shared(A,B,C,seeds_B,seeds_C)
  {
```

```
    int thread_id = omp_get_num_threads() + omp_get_thread_num();
    unsigned int local_seed = 1234 + thread_id; // Unique seed per thread
    // printf("Thread %d has local seed %u\n", thread_id, local_seed);
    // Assign unique seeds for B and C within each thread
    #pragma omp for schedule(static)
    for (int i = 0; i < N; i++) {
      seeds_B[i] = local_seed + i;
      seeds_C[i] = local_seed + i + N;
    }
    // Fill B and C with random numbers in [0, 1) using unique seeds
    #pragma omp for schedule(static)
    for (int i = 0; i < N; i++) {
      B[i] = (float)rand_r(&seeds_B[i]) / RAND_MAX;
      C[i] = (float)rand_r(&seeds_C[i]) / RAND_MAX;
    }
  // Compute A = B + C
    #pragma omp for schedule(static)
    for (int i = 0; i < N; i++) {
      A[i] = B[i] + C[i];
    }
  }
  // End measuring time
  end_time = omp_get_wtime();
  // Print the required values
  printf("A[0] = %f, B[0] = %f, C[0] = %f\n", A[0], B[0], C[0]);
  printf("A[%d] = %f, B[%d] = %f, C[%d] = %f\n", N-1, A[N-1], N-1, B[N-1], N-1, C[N-1]);
  // Print elapsed time
  printf("Elapsed time: %f seconds\n", end_time - start_time);

  // Clean up
  free(B);
  free(C);
  free(A);
  free(seeds_B);
```

```
    free(seeds_C);


    return 0;
}
```

S1 OPENMP SLURM:

```
#!/bin/bash -l

#SBATCH --job-name="zad1" # any name you like

#SBATCH --output="zad1_OpenMP.out" # any name you like, the file with output

#SBATCH --nodes=1 # number of nodes

#SBATCH --mem=1000

#SBATCH --time=00:30:00 # not too much...

#SBATCH --account=g96-1882 # your grant id

#SBATCH --partition=okeanos


gcc -fopenmp zad1.c -o zad1.exe -lm


for j in 1 2 4 8 16 32

do

#SBATCH --cpus-per-task=$j

export OMP_NUM_THREADS=$j # the number of threads

echo "Uruchamianie programu z $j watkami"

srun time ./zad1.exe

Done
```

S1 MPI:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <time.h>

// Funkcja do generowania losowej liczby zmiennoprzecinkowej z przedzialu [0,1]
double random_double() {
  return (double)rand() / RAND_MAX;
}

int main(int argc, char** argv) {
  int rank, size;
  const int n = 3048576;
  double *A_local, *B_local, *C_local;
  int local_n;
  double start_time, end_time, total_time;

  // Inicjalizacja MPI
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  // Obliczenie ilosci elementów na proces
  local_n = n / size;

  // Alokacja pamieci dla lokalnych czesci wektorów
  A_local = (double*)malloc(local_n * sizeof(double));
  B_local = (double*)malloc(local_n * sizeof(double));
  C_local = (double*)malloc(local_n * sizeof(double));

  // Inicjalizacja generatora losowego
  srand(time(NULL) + rank);
```

```
 // Start pomiaru czasu
  if (rank == 0) {
     start_time = MPI_Wtime();
  }

  // Wypelnienie lokalnych wektorów B i C losowymi wartosciami
  for (int i = 0; i < local_n; i++) {
     B_local[i] = random_double();
     C_local[i] = random_double();
     A_local[i] = B_local[i] + C_local[i];
  }

  // Proces 0 wypisuje A[0], B[0], C[0]
  if (rank == 0) {
     end_time = MPI_Wtime();
     total_time = end_time - start_time;
     printf("Czas wykonania operacji: %f sekund\n", total_time);
     printf("Proces %d: A[0] = %f, B[0] = %f, C[0] = %f\n", rank, A_local[0], B_local[0], C_local[0]);



  }

  // Proces o ranku size-1 wypisuje A[n-1], B[n-1], C[n-1]
  if (rank == size - 1) {
     printf("Proces %d: A[n-1] = %f, B[n-1] = %f, C[n-1] = %f\n", rank, A_local[local_n - 1], B_local[local_n - 1],
C_local[local_n - 1]);
  }

  // Zwolnienie pamieci lokalnych wektorów
  free(A_local);
  free(B_local);
  free(C_local);
```

```
    // Zakonczenie MPI
    MPI_Finalize();

    return 0;
}
```

S1 MPI SLURM:

```
#!/bin/bash -l
#SBATCH --job-name="MPI" # any name you like
#SBATCH --nodes=16 # number of nodes
#SBATCH --ntasks=32 # number of processes
#SBATCH --ntasks-per-node=2 # how many processes per node
#SBATCH --mem=1000
#SBATCH --time=00:10:00 # not too much...
#SBATCH --account=g96-1882 # your grant id
#SBATCH --partition=okeanos
#SBATCH --output="zad1_MPI_32proc.out" # any name you like, the file with output
# CRAY MPICH version 7.7.10

# Zwiekszenie limitu stosu do 16 MB dla 2 PROC (lub "unlimited" dla pelnego wylaczenia limitu)
# ulimit -s 16384

cc zad1_MPI.c -o zad1_MPI.exe # not necessary if done earlier
srun ./zad1_MPI.exe
```

S2 OPENMP:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#define N 200000000

int main() {
  double *A = malloc(N * sizeof(double));

  if (!A) {
    fprintf(stderr, "Memory allocation failed\n");
    return 1;
  }

  // Tablica seedow dla wektora A
  unsigned int *seeds_A = malloc(N * sizeof(unsigned int));
  if (!seeds_A) {
    fprintf(stderr, "Memory allocation for seeds failed\n");
    free(A);
    return 1;
  }

  double start_time, end_time;

  // Zmienna do przechowania sumy kwadratów
  double sum = 0.0;

  // Start measuring time
  start_time = omp_get_wtime();

  // Równolegle inicjalizowanie seedow i wypelnianie tablicy losowymi wartosciami
```

```
#pragma omp parallel default(none) shared(A, seeds_A, sum)
{
   int thread_id = omp_get_num_threads() + omp_get_thread_num();
   unsigned int local_seed = 1234 + thread_id;  // Unikalne seed dla kazdego watku

   // Przypisanie unikalnych seedow do wektora A w kazdym watku
   #pragma omp for schedule(static)
   for (int i = 0; i < N; i++) {
      seeds_A[i] = local_seed + i;
   }

   // Wypelnianie A losowymi liczbami w przedziale [0, 1) przy uzyciu unikalnych seedow
   #pragma omp for schedule(static)
   for (int i = 0; i < N; i++) {
      A[i] = (double)rand_r(&seeds_A[i]) / RAND_MAX;
   }

   // Równolegle obliczenie sumy kwadratów
   #pragma omp for schedule(guided) reduction(+:sum)
   for (int i = 0; i < N; i++) {
      sum += A[i] * A[i];
   }
}
// Obliczenie normy drugiej
double norma_dwa = sqrt(sum);
// End measuring time
end_time = omp_get_wtime();
// Wypisanie wyniku
printf("Norma druga wektora wynosi: %f\n", norma_dwa);

// Wypisanie czasu dzialania programu
printf("Elapsed time: %f seconds\n", end_time - start_time);
```

```
    // Zwolnienie pamieci
    free(A);
    free(seeds_A);

    return 0;
}
```

S2 OPENMP SLURM:

```
#!/bin/bash -l

#SBATCH --job-name="zad2" # any name you like

#SBATCH --output="zad2_OpenMP.out" # any name you like, the file with output

#SBATCH --nodes=1 # number of nodes

#SBATCH --mem=30000

#SBATCH --time=00:10:00 # not too much...

#SBATCH --account=g96-1882 # your grant id

#SBATCH --partition=okeanos


gcc -fopenmp zad2_openmp.c -o zad2_openmp.exe -lm




for j in 1 2 4 8 16 32

do

#SBATCH --cpus-per-task=$j

export OMP_NUM_THREADS=$j # the number of threads

echo "Uruchamianie programu z $j watkami"

srun time ./zad2_openmp.exe

Done
```

S2 MPI:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

#define N 200000000  // Dlugosc wektora

int main(int argc, char** argv) {
   int rank, size;
   double *local_A = NULL;
   double local_sum = 0.0, global_sum = 0.0;
   double start_time, end_time;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   // Obliczanie lokalnej dlugosci tablicy
   int local_n = N / size;

   // Przydzielanie pamieci dla lokalnej tablicy
   local_A = malloc(local_n * sizeof(double));
   if (!local_A) {
      fprintf(stderr, "Memory allocation for local array failed\n");
      MPI_Abort(MPI_COMM_WORLD, 1);
   }

   // Rozpocznij pomiar czasu
   start_time = MPI_Wtime();

   // Inicjalizacja seeda dla kazdego procesu
```

```
unsigned int seed = 1234 + rank * 1000;  // Unikalny seed na podstawie ranku
srand(seed);  // Inicjalizacja generatora liczb losowych

// Wypelnianie lokalnej tablicy losowymi wartosciami
for (int i = 0; i < local_n; i++) {
   local_A[i] = (double)rand() / RAND_MAX;
}



// Oblicz lokalna sume kwadratów
for (int i = 0; i < local_n; i++) {
   local_sum += local_A[i] * local_A[i];
}

// Redukcja lokalnych sum kwadratów do sumy globalnej
MPI_Reduce(&local_sum, &global_sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

// Obliczanie i wypisywanie normy drugiej tylko na procesie 0
if (rank == 0) {
   double norma_dwa = sqrt(global_sum);
   end_time = MPI_Wtime();

   // Wypisanie wyniku
   printf("Norma druga wektora wynosi: %f\n", norma_dwa);
   printf("Elapsed time: %f seconds\n", end_time - start_time);
}

// Zwolnienie pamieci
free(local_A);

MPI_Finalize();
return 0;
```

```
}
```

S2 MPI SLURM:

```
#!/bin/bash -l

#SBATCH --job-name="MPI" # any name you like

#SBATCH --nodes=16 # number of nodes

#SBATCH --ntasks=32 # number of processes

#SBATCH --ntasks-per-node=2 # how many processes per node

#SBATCH --mem=1000

#SBATCH --time=00:10:00 # not too much...

#SBATCH --account=g96-1882 # your grant id

#SBATCH --partition=okeanos

#SBATCH --output="zad2_MPI_32proc.out" # any name you like, the file with output

# CRAY MPICH version 7.7.10



# Zwiekszenie limitu stosu do 16 MB dla 2 PROC (lub "unlimited" dla pelnego wylaczenia limitu)

# ulimit -s 16384



cc zad2_mpi.c -o zad2_mpi.exe # not necessary if done earlier

srun ./zad2_mpi.exe
```

A4 Heat Sekwencyjny:

```
#include <stdio.h>
#include <math.h>
#include <omp.h>
#include <string.h>

#define N 512  // number of rows/columns
#define MAX_ITER 1000000 // max number of iterations
#define EPS 0.00006 // error val for stop criterion

#define VAL_UPPPER 70.0 // upper side values
#define VAL_LOWER 1.0 // lower side values
#define VAL_LEFT 20.0 // left side values
#define VAL_RIGHT 50.0 // right side values

// Declare statically sized 2D arrays
double arr[N][N];
double old_arr[N][N];

void fillSides(double arr[N][N], int m, int n);
void fillRow(int n, double arr[N][N], int k);

int main()
{
  for (int i = 0; i < N; ++i) {
    fillRow(N, arr, i);
    fillRow(N, old_arr, i);
  }

  fillSides(arr, N, N);
  fillSides(old_arr, N, N);
```

```
double start, end;
start = omp_get_wtime();


for (int t = 0; t < MAX_ITER; ++t) {
  double temp = 0;


  switch (t % 2 == 0) {
  case 0:
    for (int i = 1; i < N - 1; ++i) {
      for (int j = 1; j < N - 1; ++j) {
        arr[i][j] = (old_arr[i + 1][j] + old_arr[i - 1][j] + old_arr[i][j + 1] + old_arr[i][j - 1]) / 4;
        temp += (arr[i][j] - old_arr[i][j]) * (arr[i][j] - old_arr[i][j]);
      }
    }
      break;


  case 1:
    for (int i = 1; i < N - 1; ++i) {
      for (int j = 1; j < N - 1; ++j) {
        old_arr[i][j] = (arr[i + 1][j] + arr[i - 1][j] + arr[i][j + 1] + arr[i][j - 1]) / 4;
        temp += (arr[i][j] - old_arr[i][j]) * (arr[i][j] - old_arr[i][j]);
      }
    }
      break;
  }


  temp = sqrt(temp / ((N - 2) * (N - 2)));
  if (t % 10000 == 0) {
    printf("\n current eps = %lf in %d iteration ", temp, t);
  }
  if (temp < EPS) {
    printf("\n Number of iterations = %d\n final eps = %lf ", t,temp);
    break;
```

```
        }
        // stopping criterion
    }

    end = omp_get_wtime();

    printf("Work took %lf seconds\n", end - start);

    return 0;
}

void fillRow(int n, double arr[N][N], int k) {
    for (int i = 0; i < n; ++i) {
        arr[k][i] = 0;
    }
}

void fillSides(double arr[N][N], int m, int n) {
    int k = n - 1;
    for (int i = 1; i < k; ++i) {
        arr[i][0] = VAL_LEFT;
    }
    for (int i = 1; i < k; ++i) {
        arr[i][k] = VAL_RIGHT;
    }
    k = m - 1;
    for (int j = 1; j < k; ++j) {
        arr[0][j] = VAL_UPPPER;
    }
    for (int j = 1; j < k; ++j) {
        arr[k][j] = VAL_LOWER;
    }
}
```

A4 HEAT OPENMP:

```
#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <omp.h>

#include <string.h>


#define N 1024  // liczba wierszy i kolumn

#define MAX_ITER 1000000 // maksymalna liczba iteracji

#define EPS 0.00006 // kryterium bledu do zatrzymania


#define VAL_UPPPER 70.0 // wartosci na górnej krawedzi

#define VAL_LOWER 1.0 // wartosci na dolnej krawedzi

#define VAL_LEFT 20.0 // wartosci na lewej krawedzi

#define VAL_RIGHT 50.0 // wartosci na prawej krawedzi


// Definicje statycznych tablic

double arr[N][N];

double old_arr[N][N];


void fillSides(double arr[N][N]);

void fillRow(int n, double arr[N][N], int k);

void copy_matrix(double arr[N][N], double old_arr[N][N]);


int main()

{

int nthreads = omp_get_num_threads();

        int id = omp_get_thread_num();

double start, end;

double temp = 1;

start = omp_get_wtime();


// Inicjalizacja tablic
```

```
#pragma omp parallel default(none) shared(arr, old_arr, temp)
{
#pragma omp for schedule(guided)
for (int i = 0; i < N; ++i) {
fillRow(N, arr, i);
fillRow(N, old_arr, i);
}

fillSides(arr);
fillSides(old_arr);

////////////////////////////////////////////////////////

for (int t = 0; t < MAX_ITER && temp > EPS; ++t) {

#pragma omp barrier
#pragma omp single
{
temp = 0;
}

switch (t % 2 == 0) {
case 0:
#pragma omp for collapse(2) reduction(+:temp)
for (int i = 1; i < N - 1; ++i) {
for (int j = 1; j < N - 1; ++j) {
arr[i][j] = (old_arr[i + 1][j] + old_arr[i - 1][j] + old_arr[i][j + 1] + old_arr[i][j - 1]) / 4;
temp += (arr[i][j] - old_arr[i][j]) * (arr[i][j] - old_arr[i][j]);
}
}
break;
```

```
case 1:
#pragma omp for collapse(2) reduction(+:temp)
for (int i = 1; i < N - 1; ++i) {
for (int j = 1; j < N - 1; ++j) {
old_arr[i][j] = (arr[i + 1][j] + arr[i - 1][j] + arr[i][j + 1] + arr[i][j - 1]) / 4;
temp += (arr[i][j] - old_arr[i][j]) * (arr[i][j] - old_arr[i][j]);
}
}
break;
}

#pragma omp single
{
temp = sqrt(temp / ((N - 2) * (N - 2)));
if (t % 10000 == 0) {
printf("\n current eps = %lf in %d iteration ", temp, t);
}
if (temp < EPS) {
                                                printf("\n Number of iterations = %d\n final eps = %lf ", t,temp);
            }
}
}
}

end = omp_get_wtime();

printf("Work took %lf seconds\n", end - start);

return 0;
}

void fillRow(int n, double arr[N][N], int k) {
for (int i = 0; i < n; ++i) {
```

```
arr[k][i] = 0;
}
}

void fillSides(double arr[N][N]) {
int k = N - 1;

for (int i = 1; i < k; ++i) {
arr[i][0] = VAL_LEFT;
}
for (int i = 1; i < k; ++i) {
arr[i][k] = VAL_RIGHT;
}
k = N - 1;
for (int j = 1; j < k; ++j) {
arr[0][j] = VAL_UPPPER;
}
for (int j = 1; j < k; ++j) {
arr[k][j] = VAL_LOWER;
}
}
```

A4 HEAT OPENMP SLURM:

```
#!/bin/bash -l

#SBATCH --job-name="heat_OpenMP" # any name you like

#SBATCH --output="OpenMP.out" # any name you like, the file with output

#SBATCH --nodes=1 # number of nodes

#SBATCH --mem=1000

#SBATCH --time=00:30:00 # not too much...

#SBATCH --account=g96-1882 # your grant id

#SBATCH --partition=okeanos

for j in 2 4 8 16 32

do

#SBATCH --cpus-per-task=$j

export OMP_NUM_THREADS=$j # the number of threads

echo "Uruchamianie programu z $j watkami"

srun time ./OpenMP.exe

done
```

A4 HEAT MPI:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#include <string.h>

#define N 1024  // liczba wierszy/kolumn
#define MAX_ITER 1000000 // maksymalna liczba iteracji
#define EPS 0.00006 // kryterium bledu dla warunku stopu

#define VAL_UPPPER 70.0 // górna wartosc brzegowa
#define VAL_LOWER 1.0 // dolna wartosc brzegowa
#define VAL_LEFT 20.0 // lewa wartosc brzegowa
#define VAL_RIGHT 50.0 // prawa wartosc brzegowa

void fillSides(double arr[][N], int local_n, int rank, int size);
void exchangeBoundaries(double local_arr[][N], int local_n, int rank, int size, MPI_Comm comm);
double calculateJacobi(double local_old[][N], double local_new[][N], int local_n, int rank, int size);
void save2D(double arr[][N], int m, int n, const char *filename);

int main(int argc, char *argv[])
{
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  // Obliczanie liczby wierszy dla kazdego procesu
  int base_rows = N / size;  // Podstawowa liczba wierszy
  int extra_row = (rank == 0 || rank == size - 1) ? 1 : 2;  // Rank 0 i ostatni maja 1 marginesowy, reszta po 2.
  int local_n = base_rows + extra_row;
```

```
// Sprawdzenie, czy local_n nie przekracza rozmiaru tablicy
if (local_n > N + 2) {
   printf("Error: local_n exceeds the array bounds for process %d\n", rank);
   MPI_Abort(MPI_COMM_WORLD, 1);
}

// Tworzenie lokalnych tablic dla kazdego procesu
double local_arr_0[local_n][N];  // Tablica lokalna nr 0
double local_arr_1[local_n][N];  // Tablica lokalna nr 1

double start_time, end_time;
MPI_Barrier(MPI_COMM_WORLD); // Synchronizacja procesów przed rozpoczeciem pomiaru czasu
start_time = MPI_Wtime();

// Inicjalizacja tablic na 0
for (int i = 0; i < local_n; ++i) {
   for (int j = 0; j < N; ++j) {
      local_arr_0[i][j] = 0.0;
      local_arr_1[i][j] = 0.0;
   }
}

// Wypelnienie tablic warunkami brzegowymi
fillSides(local_arr_0, local_n, rank, size);
fillSides(local_arr_1, local_n, rank, size);

double global_error = 1.0;
for (int iter = 0; iter < MAX_ITER && global_error > EPS; ++iter) {
   double local_error = 0.0;

   switch (iter % 2) {
      case 0:
```

```
            // Wymiana danych z sasiednimi procesami (dla local_arr_0)
            exchangeBoundaries(local_arr_0, local_n, rank, size, MPI_COMM_WORLD);
            // Obliczenia Jacobiego (z local_arr_0 do local_arr_1) przed wymiana granic
            local_error = calculateJacobi(local_arr_0, local_arr_1, local_n, rank, size);
            break;
        case 1:
            // Wymiana danych z sasiednimi procesami (dla local_arr_1)
            exchangeBoundaries(local_arr_1, local_n, rank, size, MPI_COMM_WORLD);
            // Obliczenia Jacobiego (z local_arr_1 do local_arr_0) przed wymiana granic
            local_error = calculateJacobi(local_arr_1, local_arr_0, local_n, rank, size);
            break;
    }

    // Redukcja globalna dla sprawdzenia konwergencji
    MPI_Allreduce(&local_error, &global_error, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    global_error = sqrt(global_error / ((N-2) * (N-2)));

    // Wyswietlanie co 10000 iteracji
    if (rank == 0 && iter % 10000 == 0) {
        printf("Iteration %d, global error: %lf\n", iter, global_error);
    }

    // Sprawdzenie warunku stopu
    if (global_error < EPS) {
        if (rank == 0) {
            printf("Converged after %d iterations, final error: %lf\n", iter, global_error);
        }
        break;
    }
}

MPI_Barrier(MPI_COMM_WORLD); // Synchronizacja procesów po zakonczeniu iteracji
end_time = MPI_Wtime();
```

```
    if (rank == 0) {
        printf("Total execution time: %lf seconds\n", end_time - start_time);
    }
    MPI_Finalize();
    return 0;
}

// Ustawianie wartosci brzegowych dla kazdego procesu
void fillSides(double arr[][N], int local_n, int rank, int size) {
    // Wypelnianie bocznych wartosci brzegowych
    for (int i = 0; i < local_n; ++i) {
        arr[i][0] = VAL_LEFT;
        arr[i][N - 1] = VAL_RIGHT;
    }

    // Proces 0 ustawia górna krawedz
    if (rank == 0) {
        for (int j = 0; j < N; ++j) {
            arr[0][j] = VAL_UPPPER;
        }
    }

    // Ostatni proces ustawia dolna krawedz
    if (rank == size - 1) {
        for (int j = 0; j < N; ++j) {
            arr[local_n - 1][j] = VAL_LOWER;
        }
    }
}

// Wymiana granicznych wierszy miedzy sasiednimi procesami z nieblokujaca komunikacja
void exchangeBoundaries(double local_arr[][N], int local_n, int rank, int size, MPI_Comm comm) {
```

```
    MPI_Request requests[4];
    int req_count = 0;

    // Asynchroniczna komunikacja z poprzednim procesem
    if (rank > 0) {
        MPI_Irecv(local_arr[0], N, MPI_DOUBLE, rank - 1, 0, comm, &requests[req_count++]);
        MPI_Isend(local_arr[1], N, MPI_DOUBLE, rank - 1, 0, comm, &requests[req_count++]);
    }

    // Asynchroniczna komunikacja z nastepnym procesem
    if (rank < size - 1) {
        MPI_Irecv(local_arr[local_n - 1], N, MPI_DOUBLE, rank + 1, 0, comm, &requests[req_count++]);
        MPI_Isend(local_arr[local_n - 2], N, MPI_DOUBLE, rank + 1, 0, comm, &requests[req_count++]);
    }

    // Oczekiwanie na zakonczenie wszystkich operacji
    MPI_Waitall(req_count, requests, MPI_STATUSES_IGNORE);
}

// Obliczanie jednej iteracji metody Jacobiego
double calculateJacobi(double local_old[][N], double local_new[][N], int local_n, int rank, int size) {
    double local_error = 0.0;

    for (int i = 1; i < local_n - 1; ++i) {
        for (int j = 1; j < N - 1; ++j) {
            local_new[i][j] = 0.25 * (local_old[i-1][j] + local_old[i+1][j] + local_old[i][j-1] + local_old[i][j+1]);
            local_error += (local_new[i][j] - local_old[i][j]) * (local_new[i][j] - local_old[i][j]);
        }
    }

    return local_error;
}
```

A4 HEAT MPI SLURM:

```
#!/bin/bash -l

#SBATCH --job-name="MPI" # any name you like

#SBATCH --nodes=16 # number of nodes

#SBATCH --ntasks=128 # number of processes

#SBATCH --ntasks-per-node=8 # how many processes per node

#SBATCH --mem=1000

#SBATCH --time=00:10:00 # not too much...

#SBATCH --account=g96-1882 # your grant id

#SBATCH --partition=okeanos

#SBATCH --output="MPI_nb_128Proc.out" # any name you like, the file with output

# CRAY MPICH version 7.7.10


# Zwiekszenie limitu stosu do 16 MB dla 2 PROC (lub "unlimited" dla pelnego wylaczenia limitu)

# ulimit -s 16384


cc MPI_heat_nb.c -o MPI_heat_nb.exe # not necessary if done earlier

srun ./MPI_heat_nb.exe
```

A4 HEAT MPI OPENMP Wersja 1:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#include <string.h>

#define N 1024  // liczba wierszy/kolumn
#define MAX_ITER 1000000 // maksymalna liczba iteracji
#define EPS 0.00006 // kryterium bledu dla warunku stopu

#define VAL_UPPPER 70.0 // górna wartosc brzegowa
#define VAL_LOWER 1.0 // dolna wartosc brzegowa
#define VAL_LEFT 20.0 // lewa wartosc brzegowa
#define VAL_RIGHT 50.0 // prawa wartosc brzegowa

void fillSides(double arr[][N], int local_n, int rank, int size);
void exchangeBoundaries(double local_arr[][N], int local_n, int rank, int size, MPI_Comm comm);
double calculateJacobi(double local_old[][N], double local_new[][N], int local_n);
void save2D(double arr[][N], int m, int n, const char *filename);

int main(int argc, char *argv[])
{
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);



  // Obliczanie liczby wierszy dla kazdego procesu
  int base_rows = N / size;  // Podstawowa liczba wierszy
  int extra_row = (rank == 0 || rank == size - 1) ? 1 : 2;  // Rank 0 i ostatni maja 1 marginesowy, reszta po 2.
  int local_n = base_rows + extra_row;
```

```
// Sprawdzenie, czy local_n nie przekracza rozmiaru tablicy
if (local_n > N + 2) {
   printf("Error: local_n exceeds the array bounds for process %d\n", rank);
   MPI_Abort(MPI_COMM_WORLD, 1);
}

// Tworzenie lokalnych tablic dla kazdego procesu
double local_arr_0[local_n][N];  // Tablica lokalna nr 0
double local_arr_1[local_n][N];  // Tablica lokalna nr 1

double start_time, end_time;
MPI_Barrier(MPI_COMM_WORLD); // Synchronizacja procesów przed rozpoczeciem pomiaru czasu
start_time = MPI_Wtime();

// Inicjalizacja tablic na 0
for (int i = 0; i < local_n; ++i) {
   for (int j = 0; j < N; ++j) {
      local_arr_0[i][j] = 0.0;
      local_arr_1[i][j] = 0.0;
   }
}

// Wypelnienie tablic warunkami brzegowymi
fillSides(local_arr_0, local_n, rank, size);
fillSides(local_arr_1, local_n, rank, size);

double global_error = 1.0;
for (int iter = 0; iter < MAX_ITER && global_error > EPS; ++iter) {
   double local_error = 0.0;

   switch (iter % 2) {
```

```
      case 0:
        // Wymiana danych z sasiednimi procesami (dla local_arr_0)
        exchangeBoundaries(local_arr_0, local_n, rank, size, MPI_COMM_WORLD);
        // Obliczenia Jacobiego (z local_arr_0 do local_arr_1) przed wymiana granic
        local_error = calculateJacobi(local_arr_0, local_arr_1, local_n);
        break;
      case 1:
        // Wymiana danych z sasiednimi procesami (dla local_arr_1)
        exchangeBoundaries(local_arr_1, local_n, rank, size, MPI_COMM_WORLD);
        // Obliczenia Jacobiego (z local_arr_1 do local_arr_0) przed wymiana granic
        local_error = calculateJacobi(local_arr_1, local_arr_0, local_n);
        break;
    }

    // Redukcja globalna dla sprawdzenia konwergencji
    MPI_Allreduce(&local_error, &global_error, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    global_error = sqrt(global_error / ((N-2) * (N-2)));

    // Wyswietlanie co 10000 iteracji
    if (rank == 0 && iter % 10000 == 0) {
      printf("Iteration %d, global error: %lf\n", iter, global_error);
    }

    // Sprawdzenie warunku stopu
    if (global_error < EPS) {
      if (rank == 0) {
        printf("Converged after %d iterations, final error: %lf\n", iter, global_error);
      }
      break;
    }
  }

  MPI_Barrier(MPI_COMM_WORLD); // Synchronizacja procesów po zakonczeniu iteracji
```

```
    end_time = MPI_Wtime();

    if (rank == 0) {
        printf("Total execution time: %lf seconds\n", end_time - start_time);
    }
    MPI_Finalize();
    return 0;
}

// Ustawianie wartosci brzegowych dla kazdego procesu
void fillSides(double arr[][N], int local_n, int rank, int size) {
    // Wypelnianie bocznych wartosci brzegowych
    for (int i = 0; i < local_n; ++i) {
        arr[i][0] = VAL_LEFT;
        arr[i][N - 1] = VAL_RIGHT;
    }

    // Proces 0 ustawia górna krawedz
    if (rank == 0) {
        for (int j = 0; j < N; ++j) {
            arr[0][j] = VAL_UPPPER;
        }
    }

    // Ostatni proces ustawia dolna krawedz
    if (rank == size - 1) {
        for (int j = 0; j < N; ++j) {
            arr[local_n - 1][j] = VAL_LOWER;
        }
    }
}

// Wymiana granicznych wierszy miedzy sasiednimi procesami z nieblokujaca komunikacja
```

```
void exchangeBoundaries(double local_arr[][N], int local_n, int rank, int size, MPI_Comm comm) {
  MPI_Request requests[4];
  int req_count = 0;

  // Asynchroniczna komunikacja z poprzednim procesem
  if (rank > 0) {
    MPI_Irecv(local_arr[0], N, MPI_DOUBLE, rank - 1, 0, comm, &requests[req_count++]);
    MPI_Isend(local_arr[1], N, MPI_DOUBLE, rank - 1, 0, comm, &requests[req_count++]);
  }

  // Asynchroniczna komunikacja z nastepnym procesem
  if (rank < size - 1) {
    MPI_Irecv(local_arr[local_n - 1], N, MPI_DOUBLE, rank + 1, 0, comm, &requests[req_count++]);
    MPI_Isend(local_arr[local_n - 2], N, MPI_DOUBLE, rank + 1, 0, comm, &requests[req_count++]);
  }

  // Oczekiwanie na zakonczenie wszystkich operacji
  MPI_Waitall(req_count, requests, MPI_STATUSES_IGNORE);
}

// Obliczanie jednej iteracji metody Jacobiego
double calculateJacobi(double local_old[][N], double local_new[][N], int local_n) {
  double local_error = 0.0;

  #pragma omp parallel default(none) shared(local_new, local_old, temp)
  {
    #pragma omp for collapse(2) reduction(+:local_error)
    for (int i = 1; i < local_n - 1; ++i) {
      for (int j = 1; j < N - 1; ++j) {
        local_new[i][j] = 0.25 * (local_old[i-1][j] + local_old[i+1][j] + local_old[i][j-1] + local_old[i][j+1]);
        local_error += (local_new[i][j] - local_old[i][j]) * (local_new[i][j] - local_old[i][j]);
      }
    }
```

```
        return local_error;
    }
}
```

A4 HEAT MPI OPENMP Wersja 2:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#include <string.h>
#include <omp.h>

#define N 1024  // liczba wierszy/kolumn
#define MAX_ITER 1000000 // maksymalna liczba iteracji
#define EPS 0.00006 // kryterium bledu dla warunku stopu

#define VAL_UPPPER 70.0 // górna wartosc brzegowa
#define VAL_LOWER 1.0 // dolna wartosc brzegowa
#define VAL_LEFT 20.0 // lewa wartosc brzegowa
#define VAL_RIGHT 50.0 // prawa wartosc brzegowa

void fillSides(double arr[][N], int local_n, int rank, int size);

int main(int argc, char *argv[])
{
  int rank, size, provided;

  // Inicjalizacja MPI z obsługą wątków
  MPI_Init_thread(&argc, &argv, MPI_THREAD_SERIALIZED, &provided);

  // Sprawdzanie, czy MPI obsługuje wymagany poziom wątków
  if (provided < MPI_THREAD_SERIALIZED) {
    fprintf(stderr, "Error: MPI does not provide MPI_THREAD_SERIALIZED\n");
    MPI_Abort(MPI_COMM_WORLD, 1);
  }

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
MPI_Comm_size(MPI_COMM_WORLD, &size);


// Obliczanie liczby wierszy dla kazdego procesu
int base_rows = N / size;  // Podstawowa liczba wierszy
int extra_row = (rank == 0 || rank == size - 1) ? 1 : 2;  // Rank 0 i ostatni maja 1 marginesowy, reszta po 2.
int local_n = base_rows + extra_row;
double local_error = 1.0;
omp_set_num_threads(2);

// Sprawdzenie, czy local_n nie przekracza rozmiaru tablicy
if (local_n > N + 2) {
   printf("Error: local_n exceeds the array bounds for process %d\n", rank);
   MPI_Abort(MPI_COMM_WORLD, 1);
}

// Tworzenie lokalnych tablic dla kazdego procesu
double local_arr_0[local_n][N];  // Tablica lokalna nr 0
double local_arr_1[local_n][N];  // Tablica lokalna nr 1

double start_time, end_time;
MPI_Barrier(MPI_COMM_WORLD); // Synchronizacja procesów przed rozpoczeciem pomiaru czasu
start_time = MPI_Wtime();

#pragma omp parallel default(none) shared(local_arr_0, local_arr_1, local_n, local_error, rank, size)
{
   #pragma omp for collapse(2) schedule(guided)
   for (int i = 0; i < local_n; ++i) {
      for (int j = 0; j < N; ++j) {
         local_arr_0[i][j] = 0.0;
         local_arr_1[i][j] = 0.0;
      }
   }
```

```
    // Wypelnienie tablic warunkami brzegowymi
    fillSides(local_arr_0, local_n, rank, size);
    fillSides(local_arr_1, local_n, rank, size);

    double global_error = 1.0;

    for (int iter = 0; iter < MAX_ITER && global_error > EPS; ++iter) {
      #pragma omp barrier
      #pragma omp single
      {
        local_error = 0.0;
      }

      switch (iter % 2) {
        case 0:
          // Wymiana danych z sasiednimi procesami (dla local_arr_0)
          #pragma omp single
          {
            MPI_Request requests[4];
            int req_count = 0;

            // Asynchroniczna komunikacja z poprzednim procesem
            if (rank > 0) {
              MPI_Irecv(local_arr_0[0], N, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD, &requests[req_count++]);
              MPI_Isend(local_arr_0[1], N, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD, &requests[req_count++]);
            }

            // Asynchroniczna komunikacja z nastepnym procesem
            if (rank < size - 1) {
              MPI_Irecv(local_arr_0[local_n - 1], N, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD,
&requests[req_count++]);
              MPI_Isend(local_arr_0[local_n - 2], N, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD,
&requests[req_count++]);
            }
```

```
            // Oczekiwanie na zakonczenie wszystkich operacji
            MPI_Waitall(req_count, requests, MPI_STATUSES_IGNORE);
        }

        #pragma omp barrier

        // Obliczenia Jacobiego (z local_arr_0 do local_arr_1)
        #pragma omp for collapse(2) reduction(+:local_error)
        for (int i = 1; i < local_n - 1; ++i) {
            for (int j = 1; j < N - 1; ++j) {
                local_arr_1[i][j] = 0.25 * (local_arr_0[i-1][j] + local_arr_0[i+1][j] + local_arr_0[i][j-1] + local_arr_0[i][j+1]);
                local_error += (local_arr_1[i][j] - local_arr_0[i][j]) * (local_arr_1[i][j] - local_arr_0[i][j]);
            }
        }
        break;
    case 1:
        // Wymiana danych z sasiednimi procesami (dla local_arr_1)
        #pragma omp single
        {
            MPI_Request requests[4];
            int req_count = 0;

            // Asynchroniczna komunikacja z poprzednim procesem
            if (rank > 0) {
                MPI_Irecv(local_arr_1[0], N, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD, &requests[req_count++]);
                MPI_Isend(local_arr_1[1], N, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD, &requests[req_count++]);
            }

            // Asynchroniczna komunikacja z nastepnym procesem
            if (rank < size - 1) {
                MPI_Irecv(local_arr_1[local_n - 1], N, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD,
&requests[req_count++]);
```

```
                MPI_Isend(local_arr_1[local_n - 2], N, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD,
&requests[req_count++]);
            }

            // Oczekiwanie na zakonczenie wszystkich operacji
            MPI_Waitall(req_count, requests, MPI_STATUSES_IGNORE);
        }
        #pragma omp barrier

        // Obliczenia Jacobiego (z local_arr_1 do local_arr_0)
        #pragma omp for collapse(2) reduction(+:local_error)
        for (int i = 1; i < local_n - 1; ++i) {
            for (int j = 1; j < N - 1; ++j) {
                local_arr_0[i][j] = 0.25 * (local_arr_1[i-1][j] + local_arr_1[i+1][j] + local_arr_1[i][j-1] + local_arr_1[i][j+1]);
                local_error += (local_arr_0[i][j] - local_arr_1[i][j]) * (local_arr_0[i][j] - local_arr_1[i][j]);
            }
        }
        break;
    }

    // Redukcja globalna dla sprawdzenia konwergencji
    #pragma omp single
    {
        MPI_Allreduce(&local_error, &global_error, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
        global_error = sqrt(global_error / ((N-2) * (N-2)));
    }

    // Wyswietlanie co 10000 iteracji
    if (rank == 0 && iter % 10000 == 0) {
        printf("Iteration %d, global error: %lf\n", iter, global_error);
    }

    // Sprawdzenie warunku stopu
    if (global_error < EPS) {
```

```
            if (rank == 0) {
                printf("Converged after %d iterations, final error: %lf\n", iter, global_error);
            }
            break;
        }
    }
  }

  MPI_Barrier(MPI_COMM_WORLD); // Synchronizacja procesów po zakonczeniu iteracji
  end_time = MPI_Wtime();

  if (rank == 0) {
    printf("Total execution time: %lf seconds\n", end_time - start_time);
  }

  MPI_Finalize();
  return 0;
}

// Ustawianie wartosci brzegowych dla kazdego procesu
void fillSides(double arr[][N], int local_n, int rank, int size) {
  // Wypelnianie bocznych wartosci brzegowych
  for (int i = 0; i < local_n; ++i) {
    arr[i][0] = VAL_LEFT;
    arr[i][N - 1] = VAL_RIGHT;
  }

  // Proces 0 ustawia górna krawedz
  if (rank == 0) {
    for (int j = 0; j < N; ++j) {
      arr[0][j] = VAL_UPPPER;
    }
  }
```

```
    // Ostatni proces ustawia dolna krawedz
    if (rank == size - 1) {
        for (int j = 0; j < N; ++j) {
            arr[local_n - 1][j] = VAL_LOWER;
        }
    }
}
```