Krystian Plackowski

„Rozwiązanie równania Laplace'a programowaniem równoległym w C++"

1. Cel.

Rozwiązać równanie różniczkowe:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 y}{\partial y^2} = -1 \; in \; \Omega$$

$$u = 0 \; on \; \partial\Omega$$

Gdzie:

$$\Omega =< 0{,}1 >\times< 0{,}1 >$$

Czyli Ω jest kwadratem o boku 1.

Jest to tak zwane równanie Laplace'a.

Rozwiązanie tego równania opisuje np. profil prędkości płynu lepkiego w przepływie laminarnym przez nieskończony kanał o przekroju kwadratowym.

2. Dyskretyzacja, siatka i podejście jednowątkowe.

Stosujemy metodę różnic skończonych, dzieląc każdy z boków kwadratu na N elementów. Daje to siatkę o rozmiarze N^2. Elementy siatki numerujemy od $0 \; do \; N-1$, jak na rys. 2.1.

0	1	2	3	4	5	6	7	8	9
1	0	0	0	0	0	0	0	0	0
2	0								0
3	0								0
4	0								0
5	0								0
6	0								0
7	0								0
8	0								0
9	0	0	0	0	0	0	0	0	0

Rys. 2.1. – siatka dla $N = 10$ z uzupełnionym warunkiem brzegowym

Dyskretyzacja polega na rozpisaniu równania różniczkowego tak, aby dla każdego elementu (i, j) uzyskać zależność między sąsiednimi elementami: $(i-1, j), (i+1, j), (i, j-1), (i, j+1)$ – rys. 2.2. Korzystając ze wzoru na dyskretyzację II pochodnych, gdzie $h = \frac{1}{N}$ jest odległością między środkami kolejnych elementów, dostajemy:

$$\frac{u_{i-1,j} - 2\,u_{i,j} + u_{i+1,j}}{h} + \frac{u_{i,j-1} - 2\,u_{i,j} + u_{i,j+1}}{h} = -1$$

Co można przekształcić do:

$$u_{i,j} = \frac{h^2 + u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1}}{4}$$

0	1	2	3	i-1	i	i+1	7	8	9
1	0	0	0	0	0	0	0	0	0
2	0								0
3	0								0
j-1	0				(i,j-1)				0
j	0			(i-1,j)	(i,j)	(i+1,j)			0
j+1	0				(i,j+1)				0
7	0								0
8	0								0
9	0	0	0	0	0	0	0	0	0

Rys. 2.2. – element (i, j) i jego elementy sąsiednie

Stosując uzyskany wzór na wszystkich „wewnętrznych” elementach siatki (czyli pomijając brzegi) dostajemy <u>główną część</u> 1 kroku iteracji:

$dla\ (1 \leq i, j \leq N-2)\ \ wykonaj$:

$$u_{new}[i][j] := (h * h + u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1])\ /\ 4$$

Po wykonaniu pętli powyższej pętli przypisujemy dla wszystkich elementów tablicy u wartości elementów tablicy u_{new}, ale ze współczynnikiem relaksacji $\alpha = 0{,}5$. Zastosowanie współczynnika relaksacji przyspiesza zbieżność rozwiązania:

$$u := \alpha * u + (1 - \alpha *\ u_{new})$$

Dodatkowo, interesuje nas zmierzenie, jak blisko jesteśmy zadowalającego rozwiązania (a więc ile kroków iteracji będziemy jeszcze potrzebować po wykonaniu aktualnej iteracji). W tym celu wprowadzamy parametr residuum ϵ. Mierzy on zbieżność rozwiązania poprzez porównanie dwóch kolejnych iteracji:

$$\epsilon = \sqrt{\sum_{1 \le i,j \le N-2} \left[u_{i,j}^{new} - u_{i,j}\right]^2}$$

Całość 1 iteracji prezentuje się fragmentem kodu:

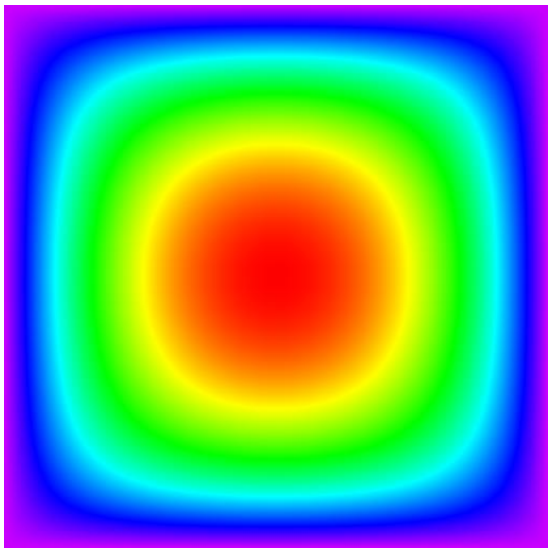
```
// wykonaj 1 iteracje na kazdym "wewnetrznym" elemencie tablicy:
//
eps = 0;

for( i = 1; i <= N-2; i++)
    for( j = 1; j <= N-2; j++) {

        u_new[i][j] = ( h*h + u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1]) / 4.;

        eps += ( u[i][j] - u_new[i][j]) * ( u[i][j] - u_new[i][j]);
    }


// **u_new := **u
//
for( i = 1; i <= N-2; i++)
    for( j = 1; j <= N-2; j++)
        u[i][j] = alfa * u[i][j] + (1-alfa) * u_new[i][j];

eps = sqrt(eps);
```

Kod 2.1. – odpowiedni tylko przy podejściu jednowątkowym

Iteracje powinny być powtarzane do momentu, gdy wartość ϵ po którejś iteracji będzie mniejsza niż założone minimum, np. $\mathbf{10}^{-5}$. Wówczas w tablicy u będzie znajdowało się rozwiązanie równania różniczkowego.

Rozwiązanie może zostać wypisane do pliku z rozszerzeniem *„.bmp”* jako mapa kolorów, przy użyciu wbudowanej funkcji ***void rysuj_kolorowy_wykres(string s, double **pole)*** która jako pierwszy argument przyjmuje nazwę docelowego pliku wraz z roszerzeniem (np. *„laplasjan.bmp”*), a jako drugi argument tablicę o rozmiarze $N \times N$, numerowaną od $(0, N-1)$, która ma zostać wypisana. Funkcja znajduje maksymalną i minimalną wartość w tablicy, następnie przydziela kolor czerwony wartości maksymalnej i fioletowy wartości minimalnej i wypisuje plik graficzny. Pozostałe wartości otrzymują kolory wynikające z liniowego przejścia wartości *Hue* w modelu kolorów *HSV* (https://en.wikipedia.org/wiki/HSL_and_HSV).

Rysunek rys. 2.3. pokazuje docelowy efekt obliczeń, jaki chcemy uzyskać.



Rys. 2.3. – poprawny rozkład prędkości w kwadracie zapisany jako mapa kolorów *HSV*

3. Podejście równoległe.

Mając do dyspozycji łączną liczbę **MODS** wątków, które będą wykonywać obliczenia w tym samym czasie, chcemy podzielić tablicę $N \times N$ na **MODS** fragmentów, każdy z fragmentów przekazać innemu wątkowi, przeprowadzić obliczenia na każdym z fragmentów, następnie odebrać wyniki obliczeń z każdego wątku i spoić w jeden wynik.

Poniżej analiza po kolei fragmentów kodu napisanego w C++ wykonywującego powyższy proces:

a) zmienne globalne.

```
static int N = 250;                          // ilosc podzialow boku kwadratu
static double dokladnosc = 1e-5;             // dokladnosc obliczenia
static double h = 1./N;                      // odleglosc oczek siatki
static double alfa = 0.5;                    // wspolczynnik relaksacji
```

Kod 3a)

Zwiększanie wartości zmiennej $\boldsymbol{N}$ powoduje zwiększenie ilości obliczeń w programie, a więc wydłużenie jego czasu działania. Jest to ilość podziałów boku kwadratu.

Zmniejszenie wartości zmiennej ***dokladnosc*** powoje przyspieszenie działania programu, gdyż zmniejsza to ilość wykonywanych iteracji.

Wartość zmiennej $\boldsymbol{h}$ nie może być zmieniana. Jest to odległość środków 2 kolejnych elementów siatki.

Zmiana wartości zmiennej ***alfa*** może przyspieszyć lub spowolnić program. Jest to współczynnik relaksacji używany w iteracji.

b) pobranie numeru wątku i ilości wszystkich wątków oraz nazwy aktualnej instancji.

```
//  Initialize MPI.
MPI_Init ( &argc, &argv );


//  Get the number of processes.
//
MPI_Comm_size ( MPI_COMM_WORLD, &MODS );
//
//  Get the individual process ID.
//
MPI_Comm_rank ( MPI_COMM_WORLD, &id );
//
//  Get Processor name:
//
MPI_Get_processor_name( procName, &nameLen );
```

Kod 3b)

Funkcja ***MPI_Init(…)*** jest wymagana początku programu.

Funkcja ***MPI_Comm_size(…)*** zwraca ilość wątków do zmiennej ***MODS***

Funkcja ***MPI_Comm_rank(…)*** zwraca numer wątku do zmiennej ***id*** i przyjmuje wartości w przedziale $(0, MODS - 1)$.

Funkcja ***MPI_Get_processor_name(…)*** zwraca nazwę instacji do tablicy **procName** typu char i długość tej nazwy do zmiennej **nameLen.** Jeśli program zostanie uruchomiony na 2 komputerach, to nazwy instancji będą się między nimi różnić.

Blok ***int main (int argc, char *argv[])*** jest uruchamiany tyle razy, ile jest wątków. Poszczególne uruchomenia odróżniają od siebie wartości zmiennych ***id*** i ***procName****. W celu wykonania jakiejś operacji na tylko jednym, konkretnym wątku, niezbędne jest użycie klamr w poniższy sposób:

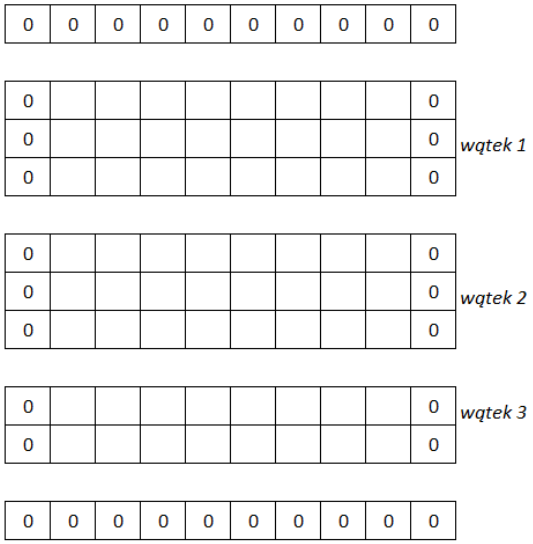
```
if( id == 0) {

    t = GetTickCount();
}
```

Kod 3.2. – wykonanie operacji tylko na wątku nr $q = 0$. Uruchomienia ***int main(…)*** z pozostałych wątków zignorują tę operację.

c) rozdzielenie tablicy $N \times N$ na wątki.

```
// tablica rozmiarow tablic, ktore otrzymaja poszczegolne watki: (znana wszystkim watkom)
//
int *size = new int[MODS];


// ustal rozmiary fragmentow tablic, ktore zostana przekazane poszczegolnym watkom:
//
for( q = 0; q < MODS; q++)
    size[q] = (N-2) / MODS;

// pierwsze kilka watkow dostanie o 1 wiekszy rozmiar niz pozostale:
//
for( q = 0; q < (N-2) - (N-2) / MODS * MODS; q++)
    size[q]++;
```

Kod. 3c)



Rys. 3.1. – wizualny podział tablicy dla $N = 10$ na 3 wątki. $8 = 3 + 3 + 2$

Wiesze pierwszy i ostatni tablicy u nie zostaną przekazane żadnemu wątkowi, gdyż nie wymagają one modyfikacji (zawsze są zerowe). Dlatego do rozdzielenia zostaje $N - 2$ wierszy.

Tablica ***size[q]*** ma zwracać ilość wierszy, która zostanie przekazana wątkowi nr $\boldsymbol{q}$.

W pierwszej chwili chwili ustalamy wartość ***size[q]*** dla każdego wątku na $\left\lfloor \frac{N-2}{MODS} \right\rfloor$.

Następnie liczymy resztę, czyli ilość wątków które nie zostały rozdzielone: $R = (N-2) - \left\lfloor \frac{N-2}{MODS} \right\rfloor$.
Następnie rozmiary wątków numerowanych od 0 do $R - 1$ zwiększamy o 1.

Przykład: rozdzielić tablicę dla $N = 10$ na 3 wątki ($MODS = 3$). W pierwszej chwili 8 wierszy rozejdzie się na kolejne wątki w stosunku $2{:}\,2{:}\,2$. Następnie liczymy $R = 2$ i modyfikujemy rozkład do $3{:}\,3{:}\,2$ – rys. 3.1.

d) sposób przechowywania fragmentów tablicy przez osobne wątki.

```
// stworz tablice **u dla watku, powiekszona o 2 wiersze (gorny i dolny):
//
double **u = new double * [size[id]+2];

for( i = 0; i <= size[id]+1; i++)
    u[i] = new double[N];
```

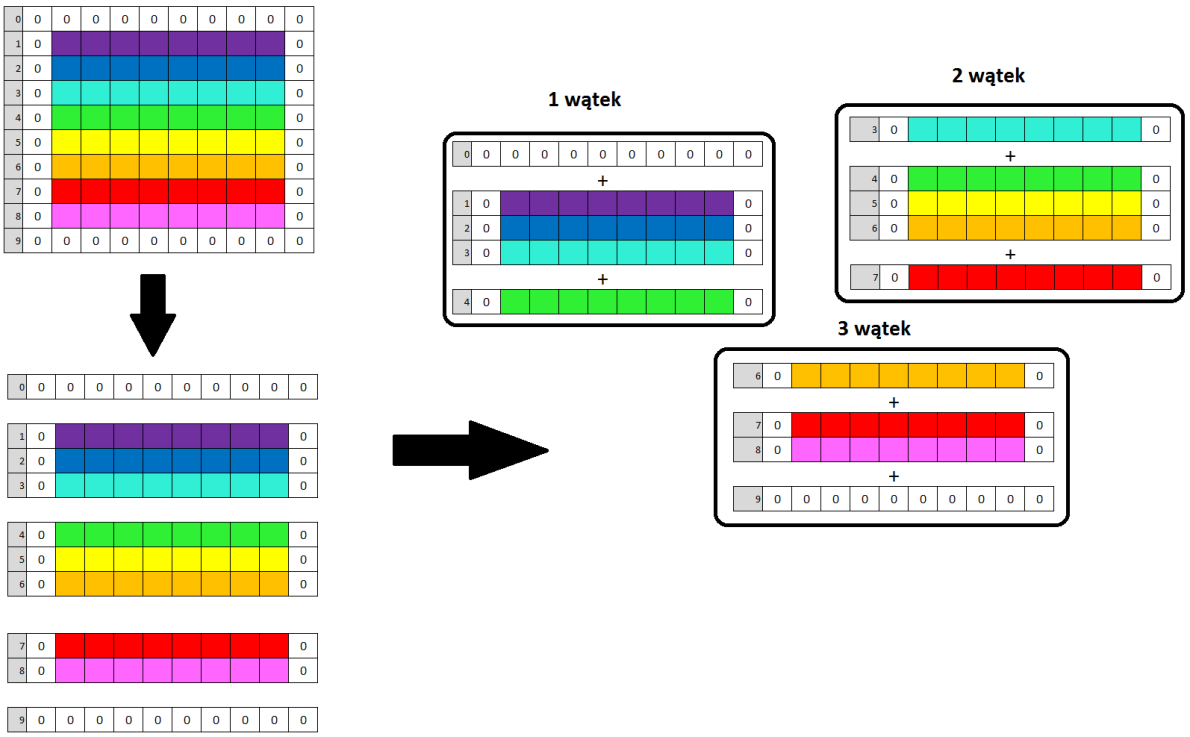
kod 3d)

Każdy wątek tworzy tablicę dynamiczną o rozmiarze ***size[id]+2***. Dodatkowe 2 wiersze są niezbędne, ze względu na to, iż do wykonania poniższego fragmentu iteracji:

$$dla\ (1 \le i \le size[q]), (1 \le j \le N-2)\ \ wykonaj:$$

$$u_{new}[i][j] := (h * h + u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1]) \ / \ 4$$

Jesteśmy zmuszeni odwoływać się do wierszy, które zostały przyporządkowane sąsiednim wątkom (1 wiersz wątku poprzedniego i 1 wiersz wątku następnego). Rozwiązaniem jest dodanie 2 wierszy do fragmentu tablicy, który przekażemy wątkowi. Dodatkowe wiersze posłużą jako warunki brzegowe podczas iteracji, a ich wartości zostaną ustalone przed iteracją. Wątek użyje tych wierszy tylko do odczytu i nie będzie modyfikował ich wartości podczas iteracji. Schematyczny sposób podziału tablicy wyjściowej do postaci, którą otrzymają wątki przedstawia rysunek rys. 3.2.



Rys. 3.2.

e) Funkcja *void foo(int q, int MODS, int size, double **u)*.

```
// obliczenia na watku nr q az do zbieznosci:
//
void foo( int q, int MODS, int size, double **u) {
    //          ^          ^
    // numer watku oraz ilosc watkow
```

Kod 3e)

Wewnątrz tej funkcji wykonywane iteracje na fragmencie tablicy wątku nr ***q***. ***MODS*** to ilość wszystkich wątków, ***size*** to przydzielona wątkowi ilość wierszy (bez pomocniczych dwóch wierszy), ***u*** to tablica zadeklarowana dynamicznie w ***int main(…)*** (dzięki temu będzie ją można dynamicznie usunąć wywołaniem wewnątrz ***int main(…)***).

Od tego momentu przechodzimy z analizą kodu do wnętrza funkcji ***foo(…).***

f) pierwsze przybliżenie rozwiązania.

```
// pierwsze "przyblizenie" rozwiazania **u wraz z warunkami brzegowymi:
//
for( i = 0; i <= size+1; i++)
    for( j = 0; j < N; j++)
        u[i][j] = 0;
```

Kod 3f)

Wiersze pomocnicze zapisujemy wartościami 0. Jeśli nie zostaną zmodyfikowane, to pozostaną zerowe (będzie tak w przypadku pierwszego i ostatniego wątku). Pozostałym elementom również przydzielamy wartość 0, choć nie licząc elementów brzegowych o indeksach $j = 0$ *i* $j = N - 1$, mogą to być inne wartości.

g) rozpoczęcie iteracji.

```
eps = INF;
// wykonaj k iteracji:
//
for( k = 0; eps > dokladnosc; k++) {

    // wyslij watkom poprzedniemu i nastepnemu gorny i dolny wiersz LICZONEJ CZESCI tablicy:
    //
```

Kod 3g)

Wejdź do pętli i kontynuuj ją dopóki wartość zmiennej ***eps*** jest większa niż wartość ustalonej na początku programu zmiennej ***dokladnosc***.

h) Komunikacja między wątkami podczas obliczeń.

```
// wyslij watkom poprzedniemu i nastepnemu gorny i dolny wiersz LICZONEJ CZESCI tablicy:
//

if( q-1 >= 0)
    MPI_Isend ( u[1], N, MPI_DOUBLE, q-1, 1, MPI_COMM_WORLD, &request);
if( q+1 < MODS)
    MPI_Isend ( u[size], N, MPI_DOUBLE, q+1, 2, MPI_COMM_WORLD, &request);



// otrzymaj gorny i dolny wiersz od watkow poprzedniego i nastepnego: (TYLKO DO ODCZYTU)
//

if( q-1 >= 0)
    MPI_Recv ( u[0], N, MPI_DOUBLE, q-1, 1, MPI_COMM_WORLD, &status );
if( q+1 < MODS)
    MPI_Recv ( u[size+1], N, MPI_DOUBLE, q+1, 2, MPI_COMM_WORLD, &status );
```

Kod 3h)

Na początku każdej iteracji dany wątek przesyła wątkom kolejnemu i poprzedniego odpowiednie, swoje wiersze tablicy, nie czekając na odbiór przez te wątki. **Wyjątki**: pierwszy wątek nie przesyła wiersza wątki poprzedniemu, ostatni wątek nie przesyła wiersza wątkowi kolejnemu.

Następnie dany wątek odbiera odpowiednie 2 wiersze, które są mu wysyłane przez równoległe wątki. Dopóki dany wątek *q* nie odbierze w pełni tych wierszy, nie może przejść dalej w kodzie.

Tak, jak zostało opisane w c) każdy wątek potrzebuje informacji o 2 wierszach należących do sąsiednich wątków. Muszą one zostać odebrane przed wykonywaniem obliczeń.

MPI_Isend (u[1], N, MPI_DOUBLE, q-1, 1, MPI_COMM_WORLD, &request) – przesyła pierwszy obliczeniowy wiersz tablicy ***u***, mający ***N*** elementów typu double wątkowi poprzedniemu i nie czeka na odbiór tego wiersza przez ów wątek. Tag wysyłki = 1.

MPI_Isend (u[size], N, MPI_DOUBLE, q+1, 2, MPI_COMM_WORLD, &request) – przesyła ostatni obliczeniowy wiersz tablicy ***u*** wątkowi kolejnemu i nie czeka na odbiór wiersza przez ów wątek. Tag wysyłki = 2.

MPI_Recv (u[0], N, MPI_DOUBLE, q-1, 1, MPI_COMM_WORLD, &status) - odbiera od wątku poprzedniego wiersz mający ***N*** elementów i zapisuje go w tablicy ***u*** jako wiersz o indeksie 0. Czeka, dopóki wiersz mający ***N*** elementów nie zostanie w całości wysłany przez poprzedni wątek. Tag = 1 odbierze wysyłkę z tagiem = 1.

MPI_Recv (u[size+1], N, MPI_DOUBLE, q+1, 2, MPI_COMM_WORLD, &status) – odbiera od wątku następnego wiersz i zapisuje go w tablicy ***u*** jako wiersz o indeksie ***size+1***. Czeka, dopóki wiersz mający ***N*** elementów nie zostanie w całości wysłany przez kolejny wątek. Tag = 2 odbierze wysyłkę z tagiem = 2.

i) wykonanie iteracji na wszystkich elementach obliczeniowych przydzielonych danemu wątkowi.

```
// wykonaj 1 iteracje na kazdym "wewnetrznym" elemencie tablicy:
//
eps = 0;

for( i = 1; i <= size; i++)
    for( j = 1; j <= N-2; j++) {

        u_new[i][j] = ( h*h + u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1]) / 4.;

        eps += ( u[i][j] - u_new[i][j]) * ( u[i][j] - u_new[i][j]);
    }


// **u_new := **u
//
for( i = 1; i <= size; i++)
    for( j = 1; j <= N-2; j++)
        u[i][j] = alfa * u[i][j] + (1-alfa) * u_new[i][j];
```

Kod 3i)

Analog do kodu kod 2.1.

j) komunikacja między wątkami w celu obliczenia błędu iteracji.

Liczymy $\epsilon = \sqrt{\sum_{1 \le i,j \le N-2} \left[u_{i,j}^{new} - u_{i,j}\right]^2}$. Ale jest trudniej, bo mamy obliczenia równoległe.

```
// watek glowny sumuje eps nadesłane ze wszystkich watkow:
//
if( q != 0) {

    // najpierw wyslij glownemu watkowi swoja czesc sumy:
    //
    MPI_Send ( &eps, 1, MPI_DOUBLE, 0, 3, MPI_COMM_WORLD);

    // nastepnie odbierz od niego sume wszystkich:
    //
    MPI_Recv ( &eps, 1, MPI_DOUBLE, 0, 4, MPI_COMM_WORLD, &status);
}
else {

    // najpierw odbierz eps od wszystkich watkow i zsumuj:
    //
    for( i = 1; i < MODS; i++) {

        MPI_Recv ( &eps_rest, 1, MPI_DOUBLE, i, 3, MPI_COMM_WORLD, &status);
        eps += eps_rest;
    }


    // nastepnie odeslij kazdemu watkowi sume:
    //
    eps = sqrt(eps);

    for( i = 1; i < MODS; i++)
        MPI_Send ( &eps, 1, MPI_DOUBLE, i, 4, MPI_COMM_WORLD);
}
```

Kod 3j)

Aby tego dokonać, musimy wybrać jeden wątek, który będzie zbierał i sumował składniki sumy z pozostałych wątków (i dodawał swoją część), liczył √ i odsyłał wynik wszystkim wątkom. Toteż robi powyższy fragment kodu.

MPI_Send (&eps, 1, MPI_DOUBLE, 0, 3, MPI_COMM_WORLD) – wyślij swoją część sumy wątkowi nr 0. Wątek nr 0 tego nie robi.

MPI_Recv (&eps, 1, MPI_DOUBLE, 0, 4, MPI_COMM_WORLD, &status) – odbierz sumę sum od wątka nr 0. Wątek nr 0 tego nie robi.

MPI_Recv (&eps_rest, 1, MPI_DOUBLE, i, 3, MPI_COMM_WORLD, &status) – jako wątek główny odbierz część sumy od wątka nr ***i.***

MPI_Send (&eps, 1, MPI_DOUBLE, i, 4, MPI_COMM_WORLD) – jako wątek główny odeślij sumę sum wątkowi nr ***i.***

k) wysłanie swojej części tablicy wątkowi głównemu po uzyskaniu zbieżności.

```
// wyslij wyniki glownemu watkowi (nr 0):
//
if( q != 0)      // nie wysylaj do siebie ( watek 0 -/-> watek 0)
    for( i = 1; i <= size; i++)
        MPI_Send ( u[i], N, MPI_DOUBLE, 0, 5, MPI_COMM_WORLD);
```

Kod 3k)

Na koniec funkcji ***foo(…)*** wyślij wszystkie wiersze (z rozwiązaniem) wątkowi nr 0 (głównemu).

MPI_Send (u[i], N, MPI_DOUBLE, 0, 5, MPI_COMM_WORLD) – jako wątek inny niż główny wyślij wiersz nr ***i***, mający ***N*** elementów wątkowi nr 0. Tag wysyłki = 5.

*Od tego momentu wracamy z analizą kodu do bloku **int main(…)**, do miejsca gdzie wywołana została funkcja **foo(…).***

l) deklaracja tablicy zbierającej wyniki.

```
// stworz tablice dynamiczna do prezentacji wynikow:
//
if (id == 0) {

    double **v;

    v = new double * [N];

    for( i = 0; i < N; i++)
        v[i] = new double[N];
```

Kod 3l)

W wątku nr 0 (głównym) zadeklaruj tablicę pod nazwą v, która będzie tablicą wyników.

m) odebranie wyników od wszystkich wątków i połączenie w całość.

```
// zbierz wyniki ze wszystkich watkow:
//
k = 0;
for( j = 0; j < size[0]+1; j++) // najpierw z watku nr 1 (glownego)
    copy( u[j], u[j]+N, v[k++]);


for( i = 1; i < MODS; i++)       // potem z pozostalych watkow
    for( j = 0; j < size[i]; j++)
        MPI_Recv ( v[k++], N, MPI_DOUBLE, i, 5, MPI_COMM_WORLD, &status );
```

kod 3m)

Powyższy kod wykonuje się tylko na wątku nr 0 (głównym).

Najpierw przepisz zawartość tablicy ***u*** do tablicy ***v***. Tablica ***u*** zawiera część rozwiązania z pierwszymi wierszami wyniku. Pod indeksem 0 tablicy ***u*** kryje się tablica pomocnicza, wypełniona zerami – ją również przepisz. Zatem przepisane zostanie ***size[0]+1*** wierszy.

Następnie przepisuj do tablicy ***v*** po kolei wszystkie wiersze przesłane z pozostałych wątków – podpunkt k).

MPI_Recv (v[k++], N, MPI_DOUBLE, i, 5, MPI_COMM_WORLD, &status) – obierz 1 wiersz od wątku nr ***i*** i zapisz go jako kolejny wiersz w tablicy wyników ***v***. Łącznie do odebrania z wątku nr ***i*** jest ***size[i]*** wierszy.

Ostatni wiersz tablicy wyników ***v*** nie zostanie odebrany, ale został on wypełniony zerami na etapie deklaracji tablicy.

n) „wypisanie” tablicy *v* do pliku graficznego.

Dokonuje tego opisana już wcześniej w pkt. 2. funkcja ***void rysuj_kolorowy_wykres(string s, double **pole).***

o) Zakończenie wątku.

Dokonuje tego komenda ***MPI_Finalize().***

4. Wyniki obliczeń na klastrze dla różnych konfiguracji instancji.

Postanowiono porównać czas działania programu na 5 konfiguracjach:

a) 8 instancji każda po 1 wątek (n=8,ppn=1).

b) 8 instancji każda po 2 wątki (n=8,ppn=2).

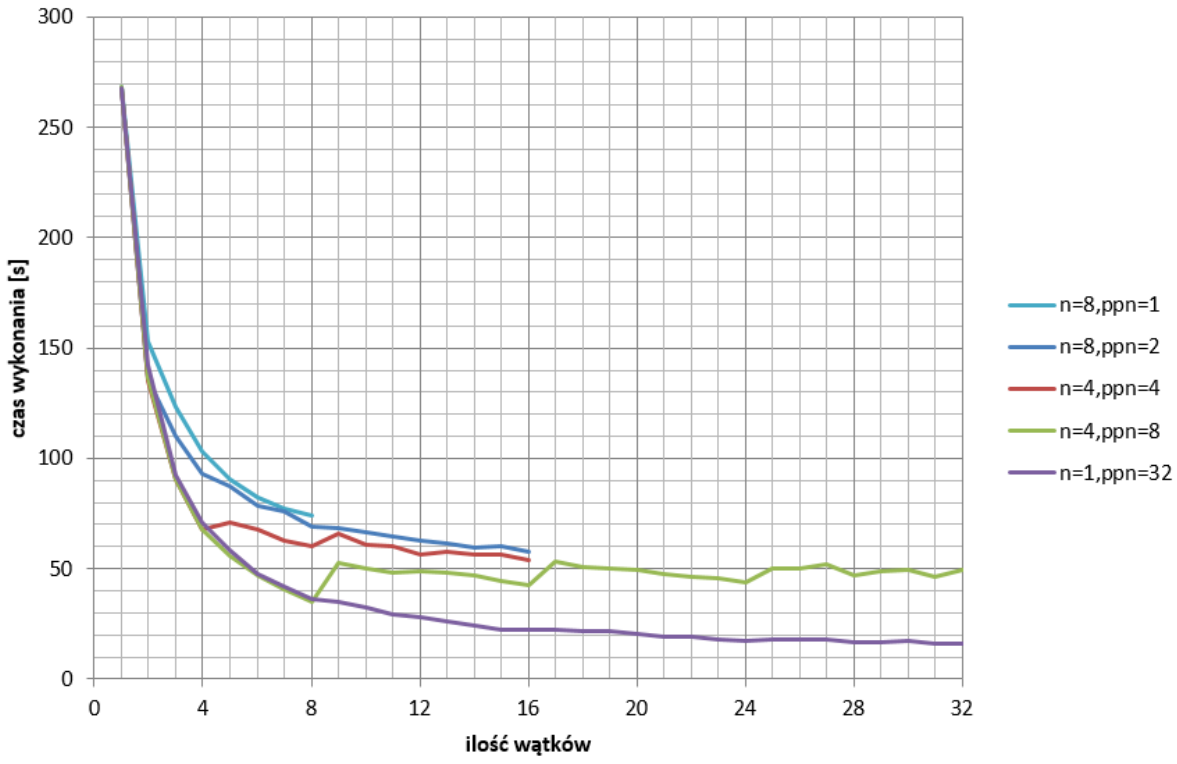
c) 4 instancje każda po 4 wątki (n=4,ppn=4).

d) 4 instancje każda po 8 wątków (n=4,ppn=8).

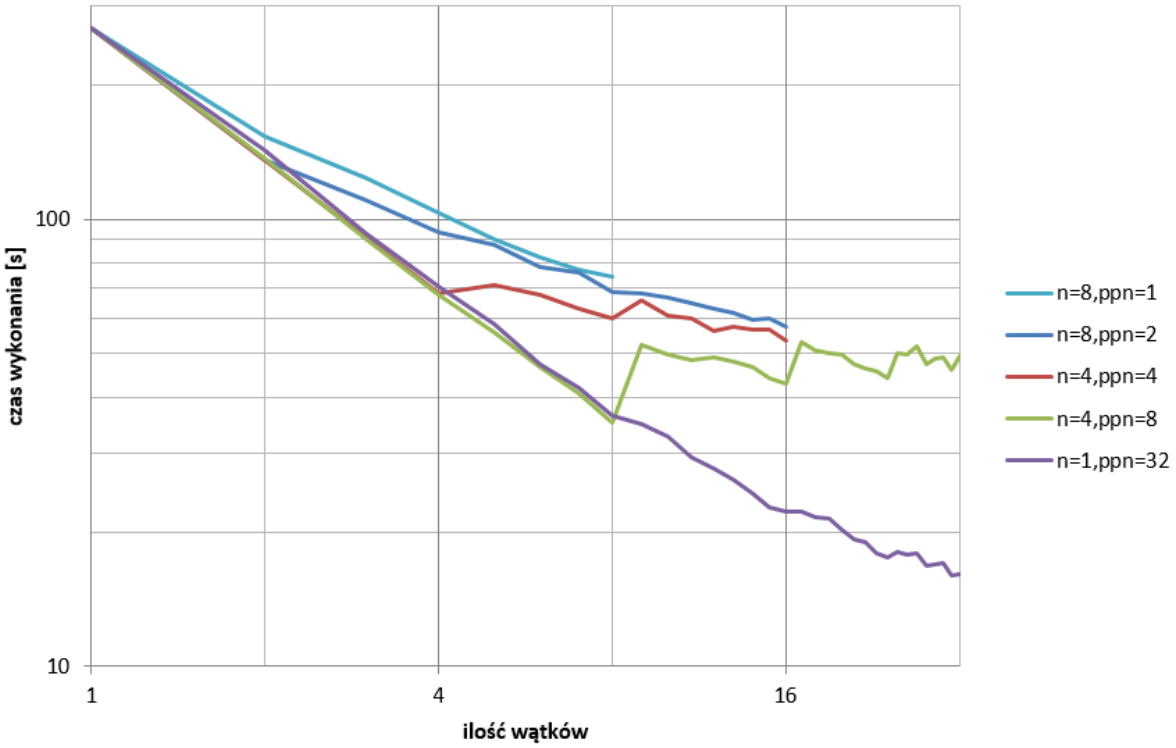
e) jedna szybka instancja 32-wątkowa (n=1,ppn=32).

Dla programu z parametrami $N = 250, \epsilon = 10^{-5}$.

Wyniki czasów obliczeń przedstawiono na **rys. 4.9. i rys. 4.10.** przy czym wyniki dla instancji 32-wątkowej zostały przeskalowane tak, aby jej czas wykonania na 1 wątku odpowiadał czasowi wykonania instancji 8-wątkowych na 1 wątku (ok. 267s).



Rys. 4.9. – czas obliczeń w zależności od liczby wątków dla różnych konfiguracji



Rys. 4.10. – czas obliczeń w zależności od liczby wątków dla różnych konfiguracji na skali logarytmicznej

Widzimy, że dla ilości wątków = 8 każda konfiguracja zwraca wyraźnie inny czas działania. Jest to spowodowane opóźnieniem wywołanym **komunikacją między instancjami**, która jest dużo wolniejsza niż komunikacja bezpośrednia między wątkami jednej instancji

Jeśli dysponujemy instancjami **1- lub 2-wątkowymi**, to zwiększanie liczby tych instancji **przyspiesza** działanie programu. Jednakże, w przypadku instancji **4-wątkowej zwiększanie liczby instancji jest nieopłacalne i prawie nie przyspiesza działania programu.** W przypadku instancji **8-wątkowych** zwiększenie liczby instancji powoduje **spowolnienie działania programu**. Najlepszym rozwiązaniem jest użycie jednej instancji o dużej ilości wątków (np. 32).

Czas obliczeń dla jednej instancji o dużej ilości wątków **skaluje się potęgowo** z ilością wątków, co widać jako linię prostą na **rys. 4.10.** Jeśli dochodzi czas wymiany informacji między kilkoma instancjami, to linia przestaje być prosta, a czas obliczeń jest gorszy niż potęgowy.