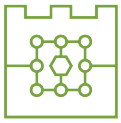


Wydział Informatyki
i Telekomunikacji

Zrównoleglenie algorytmu obliczania średniej arytmetycznej

PROJEKT Z PRZEDMIOTU PRiR
Informatyka, gr. CY-01 sem. II, studia stacjonarne II st.

Agata Bajorska,
Dariusz Cabała,
Armand Pajor

21.12.2021

Spis treści

1 Wprowadzenie

Celem projektu jest przedstawienie różnic w czasie obliczania średniej arytmetycznej pewnego zbioru liczb w różnych modelach programowania równoległego. Na samym początku zosatał przedstawiony program wykonujący obliczenia sekwencyjnie. W dalszej części projektu został przedstawiony model z wymianą komunikatów oraz model wirtualnej pamięci wspólnej. Jako ostatni zaimplementowany został model hybrydowy.

1.1 Treść zadania

Dany jest zbiór liczb całkowitych przynajmniej 5-cyfrowych. Oblicz średnią arytmetyczną liczb spełniających warunki: liczba należy do zadanego przedziału oraz suma cyfr liczby jest większa od ustalonej wartości.

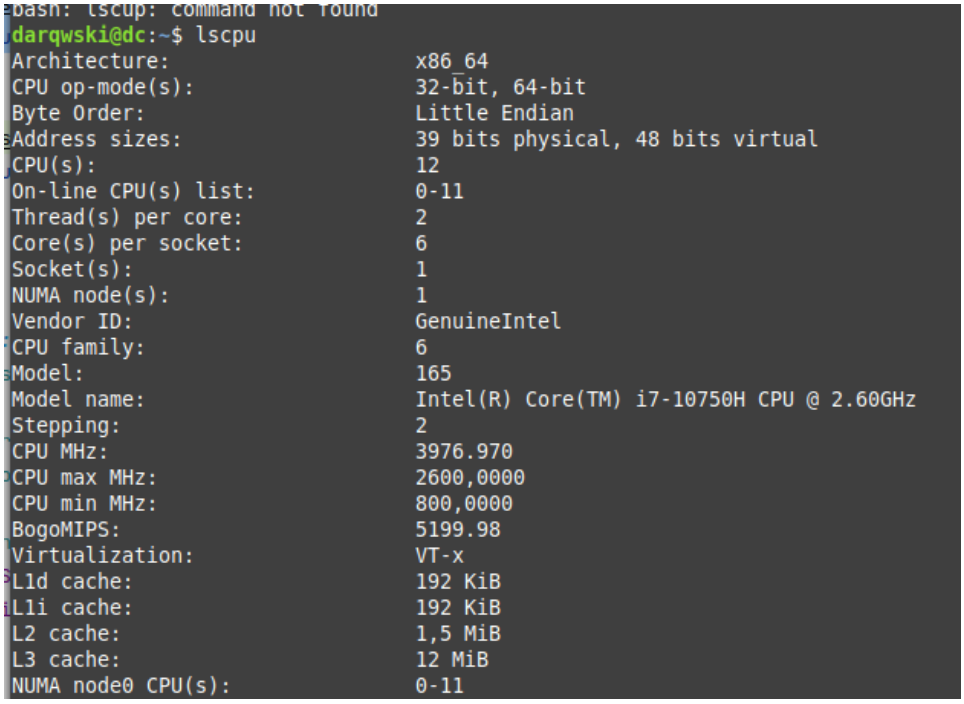
1.2 Dane wejściowe

Każda z wersji algorytmu posiada jednakowe dane wejściowe. Są one losowane generatorem pseudolosowych liczb. Poniżej przykład funkcji generującej liczby w wersji MPI:

```
1  int QUANTITY_IN_DATASET = 2000000;
2
3  void fillArrayWithRandomNumbers(int * array,
4      int minNumber, int maxNumber)
5  {
6      srand(time(NULL));
7
8      int minimalValueInDataset = 1;
9      int maximalValueInDataset = 10000000;
10     int quantityOfDataset = 5000000;
11
12     for(unsigned long int i =0;
13     i<QUANTITY_IN_DATASET;i++)
14     {
15         int random = rand();
16         array[i] = (abs(random)+minNumber)%maxNumber;
17     }
18 }
```

1.3 Parametry maszyny wykonującej obliczenia

Wszystkie implementacje algorytmu zostały uruchomione i przetestowane na maszynie, której parametry przedstawione są na poniższym rysunku.



Rysunek 1: Specyfikacja maszyny docelowej

2 Wersja sekwencyjna

2.1 Wprowadzenie teoretyczne

Wersja sekwencyjna programu charakteryzuje się tym, że jest uruchamiana na jednym procesorze maszyny. Kolejne instrukcje programu wykonują się kolejno jedna po drugiej, w taki sposób że w jednej jednostce czasu wykonywana jest tylko jedna instrukcja.

```cpp
double getAveregeWithValidationParallel(std::vector<int> numbers, int lowerLimit, int upperLimit, int minimalDigitsSum, int threads = 1)
{
    unsigned long sum = 0;
    int quantity = 0;

    double time = omp_get_wtime();
    int numbers_size = numbers.size();


    for (int index = 0; index < numbers_size; index++)
    {
        int number = numbers[index];
        if (number >= lowerLimit && number <= upperLimit)
        {
            std::string strNumber = std::to_string(number);
            int actualDigitsSum = 0;

            for (char digit : strNumber)
                actualDigitsSum += digit - '0';

            if (actualDigitsSum > minimalDigitsSum)
            {
                sum += number;
                quantity++;
            }
        }
    }


    double measuredTime = omp_get_wtime() - time;


    std::cout << measuredTime<<std::endl;

    return (double)sum / (double)quantity;
}
```

Rysunek 2: Najważniejsza funkcja sekwencyjnego algorytmu

3 Wersja w MPI

3.1 Wprowadzenie teoretyczne

Model z wymianą komunikatów (Message Passing Interface) charakteryzuje się podziałem probelmu na podproblemy, które są opracowywane przez odrębne procesy. Podproblemami w naszym zadaniu były podzbiory wejściowego zbioru danych. Zbiór wejściowy został podzielony na niezależne fragmenty, dla których wykonywane były niezależne obliczenia średniej arytmetycznej przez poszczególne procesy.

W procesie zerowym zostały wygenerowane dane wejściowe i zapisane do tablicy. Każdy proces obliczał dla fragmentu tablicy średnią arytmetyczną liczb o zdefiniowanym zakresie, wykorzystując pętlę for.

W modelu z wymianą komunikatów dane wymieniane pomiędzy procesami przesyłane są za pomocą komunikatów. Implementacja programu w MPI wykorzystuje przesył komunikatów typu jeden do jeden. Dzięki zastosowaniu funkcji MPI_Send() oraz MPI_Recv() referencja do wejściowego zbioru danych oraz obliczane sumy cząstkowe były przekazywane pomiędzy procesami co umożliwiło obliczenie średniej arytmetycznej liczb w tablicy.

```
51
52      if(plotRank == 0){
53          fillArrayWithRandomNumbers(dataset,minNumber, maxNumber);
54      }
55      if(plotRank == 0){
56
57          start = MPI_Wtime();
58
59          float sumOfAllPlots = 0;
60
61          for(int i = MAIN_PLOT_RANK+1;i<plotsAmount;i++){
62              MPI_Send(&dataset,QUANTITY_IN_DATASET,MPI_INT,i,tag,MPI_COMM_WORLD);
63
64          }
65
66          for(int i = MAIN_PLOT_RANK+1;i<plotsAmount;i++){
67              MPI_Recv(&resultToReceive,QUANTITY_IN_DATASET,MPI_INT,i,tag,MPI_COMM_WORLD,&status);
68              sumOfAllPlots+=resultToReceive;
69          }
70
71          float resultFromAllPlots = sumOfAllPlots/(plotsAmount-1);
72
73          int plotSize = ceil(QUANTITY_IN_DATASET/(plotsAmount-1));
74
75          end = MPI_Wtime();
76
77
78          printf("ForGrep%f\n", (end-start)/1000);
79      } else {
80          int plotSize = ceil(QUANTITY_IN_DATASET/(plotsAmount-1));
81          int sumInPlot = 0;
82
83          MPI_Recv(&dataset,QUANTITY_IN_DATASET,MPI_INT,MAIN_PLOT_RANK,tag,MPI_COMM_WORLD,&status);
84          int matchedNumbers = 0;
85
86          for(int i=plotSize*(plotRank-1);i<(plotRank)*plotSize;i++){
87              if(dataset[i]<minNumber){
88                  continue;
89              }
90              if(maxNumber<dataset[i]){
91                  continue;
92              }
93
94              if(sumDigitsInNumber(dataset[i])<minSumOfDigits){
95                  continue;
96              }
97              matchedNumbers++;
98              sumInPlot+=dataset[i];
99          }
100
101         resultToSend = (sumInPlot/matchedNumbers);
102
103         MPI_Send(&resultToSend,1,MPI_INT,MAIN_PLOT_RANK,tag,MPI_COMM_WORLD);
104     }
105
106     MPI_Finalize();
```

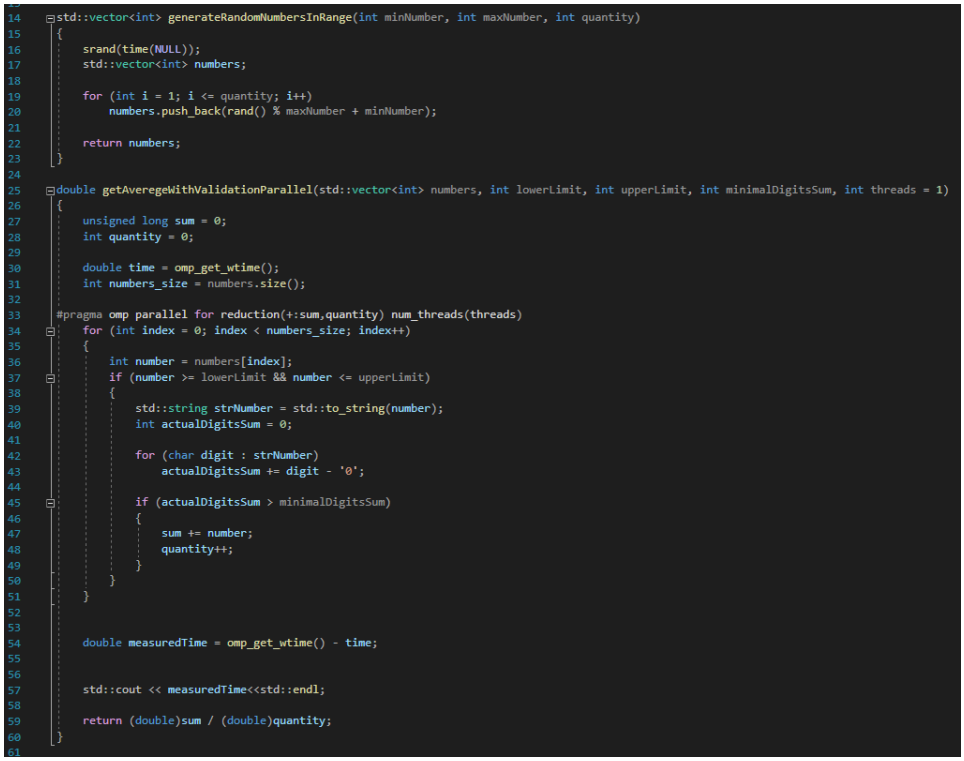
Rysunek 3: Fragment kodu algorytmu MPI

4 Wersja w OpenMP

4.1 Wprowadzenie teoretyczne

Model pamięci wspólnej algorytmu obliczającego średnią liczb został wykonany w standardzie OpenMP (Open Multi-Processing). Standard ten wykorzystuje pracę na wielu wątkach oraz pamięć współdzieloną. Kod algorytmu został napisany w języku wysokiego poziomu C++.

Implementacja algorytmu odróżnia się od MPI brakiem przesyłania komunikatów, ponieważ wykorzystana została pamięć wspólna. Kod jest tożsamy z wersją sekwencyjną, jednakże pętla algorytmu została zrównoleglona. Zrównoleglony kod algorytmu przedstawia rysunek poniżej.



```
std::vector<int> generateRandomNumbersInRange(int minNumber, int maxNumber, int quantity)
{
    srand(time(NULL));
    std::vector<int> numbers;

    for (int i = 1; i <= quantity; i++)
        numbers.push_back(rand() % maxNumber + minNumber);

    return numbers;
}

double getAveregeWithValidationParallel(std::vector<int> numbers, int lowerLimit, int upperLimit, int minimalDigitsSum, int threads = 1)
{
    unsigned long sum = 0;
    int quantity = 0;

    double time = omp_get_wtime();
    int numbers_size = numbers.size();

#pragma omp parallel for reduction(+:sum,quantity) num_threads(threads)
    for (int index = 0; index < numbers_size; index++)
    {
        int number = numbers[index];
        if (number >= lowerLimit && number <= upperLimit)
        {
            std::string strNumber = std::to_string(number);
            int actualDigitsSum = 0;

            for (char digit : strNumber)
                actualDigitsSum += digit - '0';

            if (actualDigitsSum > minimalDigitsSum)
            {
                sum += number;
                quantity++;
            }
        }
    }


    double measuredTime = omp_get_wtime() - time;


    std::cout << measuredTime<<std::endl;

    return (double)sum / (double)quantity;
}
```

Rysunek 4: Zrównoleglony fragment kodu algorytmu OpenMP

5 Implementacja Hybrydowa

5.1 Wprowadzenie teoretyczne

Implementacja hybrydowa łączy oba modele programowania równoległego: model z wymianą komunikatów i model pamięci wspólnej. Kod charakteryzuje się dwoma pozmiomami zrównoleglenia.

Kod programu początkowo rozdziela zadania na poszczególne procesy za pomocą funkcji MPI, natomiast w obrębie każdego procesu uruchamiane są dyrektywy OpenMP i następuje rozbicie podzadań na wątki (zrównoleglenie pętli). Na poniższym rysunku widoczne są dyrektywy kompilatora jak również funkcje do przesyłania komunikatów.

```
if(plotRank == 0){
    fillArrayWithRandomNumbers(dataset,minNumber, maxNumber);
}
if(plotRank == 0){

    start = MPI_Wtime();

    float sumOfAllPlots = 0;

    for(long long int i = MAIN_PLOT_RANK+1;i<plotsAmount;i++){
        MPI_Send(&dataset,QUANTITY_IN_DATASET,MPI_INT,i,tag,MPI_COMM_WORLD);

    }
    for(long long int i = MAIN_PLOT_RANK+1;i<plotsAmount;i++){
        MPI_Recv(&resultToReceive,QUANTITY_IN_DATASET,MPI_INT,i,tag,MPI_COMM_WORLD,&status);
        sumOfAllPlots+=resultToReceive;
    }

    float resultFromAllPlots = sumOfAllPlots/(plotsAmount-1);

    long long int plotSize = ceil(QUANTITY_IN_DATASET/(plotsAmount-1));

    end = MPI_Wtime();


    printf("ForGrep%f\n", (end-start)/1000);
} else {
    long long int plotSize = ceil(QUANTITY_IN_DATASET/(plotsAmount-1));
    long long int sumInPlot = 0;

    MPI_Recv(&dataset,QUANTITY_IN_DATASET,MPI_INT,MAIN_PLOT_RANK,tag,MPI_COMM_WORLD,&status);
    long long int matchedNumbers = 0;

    #pragma omp parallel for reduction(+:sumInPlot) reduction(+:matchedNumbers) num_threads(PLOTS)
    for(long long int i=plotSize*(plotRank-1);i<(plotRank)*plotSize;i++){
        if(dataset[i]<minNumber){
            continue;
        }
        if(maxNumber<dataset[i]){
            continue;
        }

        if(sumDigitsInNumber(dataset[i])<minSumOfDigits){
            continue;
        }
        matchedNumbers++;
        sumInPlot+=dataset[i];
    }

    resultToSend = (sumInPlot/matchedNumbers);

    MPI_Send(&resultToSend,1,MPI_INT,MAIN_PLOT_RANK,tag,MPI_COMM_WORLD);
}

MPI_Finalize();
}
```

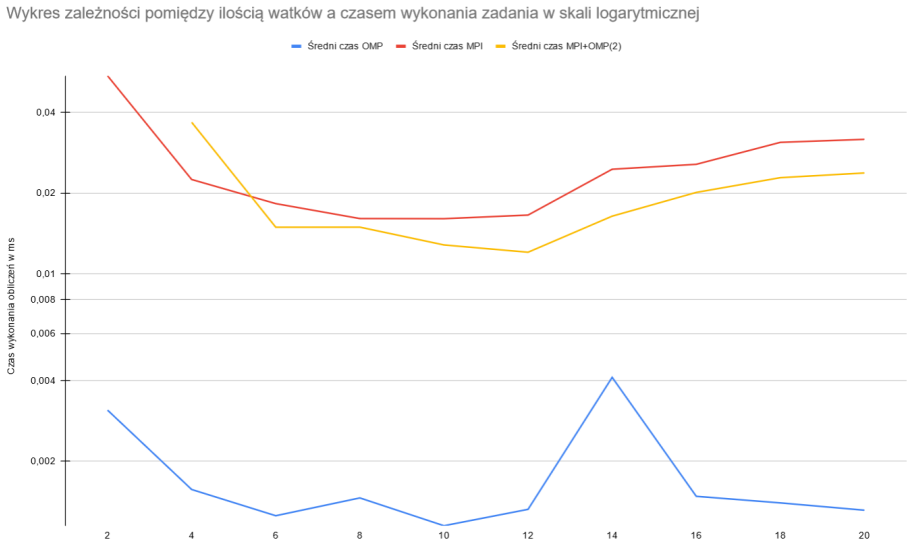
Rysunek 5: Fragment kodu implementacji hybrydowej

6 Czasy wykonania

Czasy wykonania algorytmów dla wcześniej opisanych modeli zostały przetestowane dla różnych danych wejściowych: dla 2 mln elementów w tablicy, dla 1 mln, dla 500 tysięcy oraz dla 250 tysięcy.

Wszystkie obliczenia oraz wykresy dołączone zostały do sprawozdania w arkuszu kalkulacyjnym Excel.

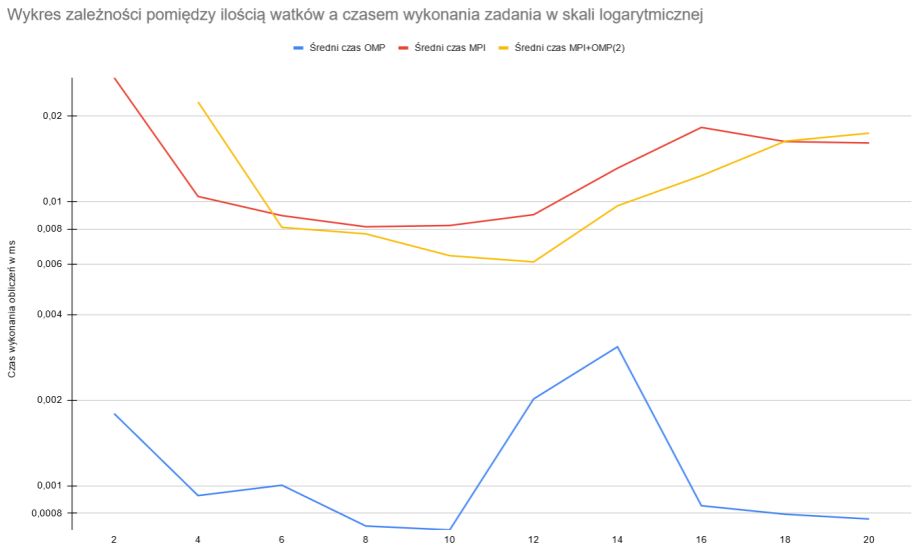
6.1 N=2000000



Rysunek 6: Wykres, N = 2mln

Dla 2 mln wartości w tablicy wejściowej optymalny jest algorytm OpenMP. Dla 2 wątków algorytm OpenMP osiągną sprawność równą w przybliżeniu 99,3185.

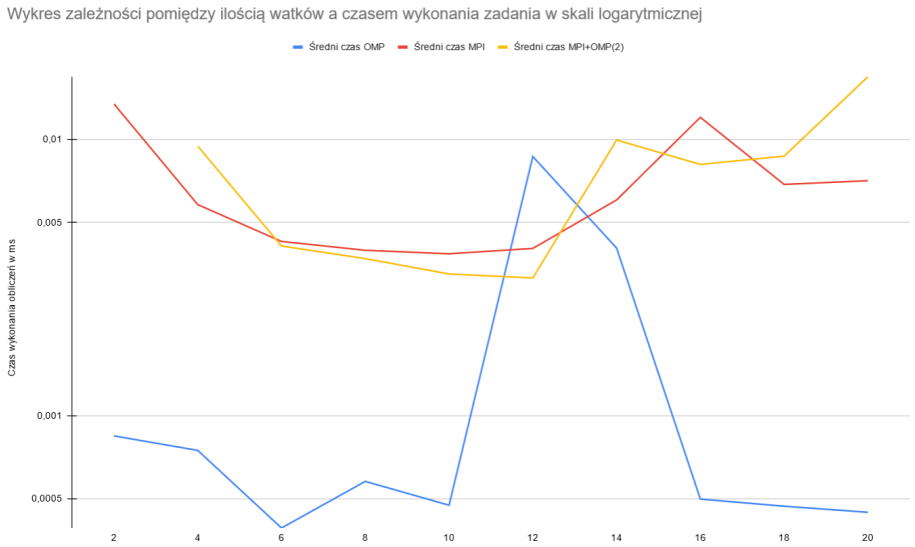
6.2 N=1000000



Rysunek 7: Wykres, N = 1mln

Dla 1 mln wartości w tablicy wejściowej optymalny jest algorytm OpenMP. W ogólnym przypadku (dla wątków od 2-20) lepszą sprawność od OpenMP wykazuje algorytm MPI. Zaskakująco wysokie przyspieszenie względne posiadał algorytm hybrydowy dla 12 wątków, aż 7,25.

6.3 N=500000

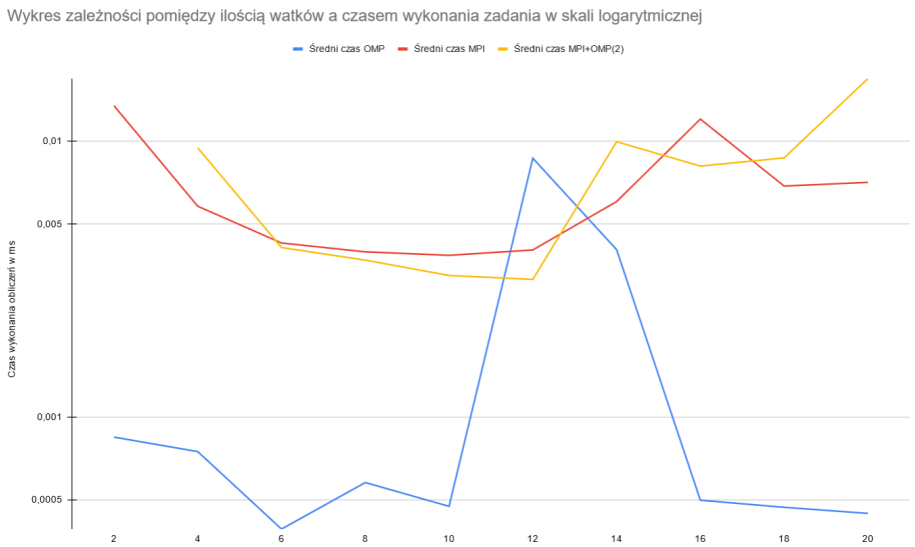


Rysunek 8: Wykres, N = 500 tys.

W ogólnym przypadku dla 500 tys. wartości w tablicy wejściowej optymalny jest algorytm OpenMP. Jednak dla 12 wątków najszybszy okazał się algorytm hybrydowy. Analizując sprawności algorytmów najwyższą wartość osiągnął model hybrydowy dla 6 wątków.

6.4 N=250000

W ogólnym przypadku dla 250 tys. wartości w tablicy wejściowej optymalny jest algorytm OpenMP. Jednak dla 12 wątków (tak jak w przypadku 500 tys. wartości) najszybszy okazał się algorytm hybrydowy. Analizując sprawności algorytmów najwyższą wartość osiągnął model MPI dla 2 wątków.



Rysunek 9: Wykres, N = 500 tys.

7 Podsumowanie

Dla wszystkich testowanych wartości N najszybszy w ogólności okazywał się algorytm OpenMP, jednak dla mniejszych próbek algorytm hybrydowy osiągał najszybciej wynik. Jednak najniższy średni czas wykonania algorytmu nie wiązał się z najwyższą sprawnością modelu.

8 Źródła

- Wykłady Dr Joanny Płażek - "OpenMP", "Modele Programowania Równoległego"
- Laboratoria Dr Filip Krużel - "OpenMP", "MPI"
- www.overleaf.com
- materiały MPI - tutorial
- stackoverflow.com
- navoica.pl