Grupa lab.
5

Data ćwiczenia
11.12.2023

Labratorium 10

Imie i nazwisko
Łukasz Sawina

Cel

Celem ćwiczenia było pogłębienie umiejętności pisania programów równoległych w środowisku OpenMP oraz umiejętność rozpoznania zależności zmiennych w programie.

Zmienne

W pierwszym zadaniu do wykonania pojawiał się program, który dla każdego wątku wykonywał pewne obliczenia na zmiennych z różnymi typami dostępu (private, fisrtprivate, shared). Niestety w programie pojawiał się problem, ponieważ równolegle działające wątki potrafiły sobie namieszać w zmiennych i wyniki były niepoprawne.

Naszym głównym zadaniem było znalezienie oraz wyeliminowanie zależności wśród zmiennych.

Pierwsza zależność jaka się pojawia w kodzie to:

```
//przed
d_local_private = a_shared + c_firstprivate;

//po
d_local_private = a_temp + c_firstprivate;
```

W kodzie zmienna a_shared jest umieszczona w klauzuli *shared()*, czyli wszystkie wątki ją współdzielą. Może się pojawić sytuacja, w której jeden wątek zacznie zmieniać wartość a_shared, a inny wątek akurat będzie chciał na niej coś zrobić, jest to zależność RAW. Rozwiązanie tego problemu jest proste, można utworzyć przed sekcją równoległą zmienną tymczasową, która będzie posiadała początkową wartość a_shared i następnie przekazać ją do sekcji krytycznej. W taki sposób każdy wątek dostanie oryginalną pierwszą wartość a_shared, a nie z ryzykiem zmiany.

Kolejną zależnością jest działąnie w pętli na zmiennej a_shared.

```
//przed
   for(i=0;i<10;i++){
     a_shared ++;
   }

//po
#pragma omp critical
   for(i=0;i<10;i++){
     a_shared ++;
   }
```

W trzym przypadku może pojawić się dużo problemów, dokładniej mówiąc pojawia się zależność RAW oraz WAR. Najlepiej to można zobaczyć rozpisując jak wyglądają poszczególne iteracje pętli.

```
a_shared = a_shared + 1;
a_shared = a_shared + 1;
a_shared = a_shared + 1;
a_shared = a_shared + 1;
```

Kolorem czerwonym widać zależność WAR, zielonym RAW, a niebieskim WAW. Te zależności pojawiają się wielokrotnie, w każdej iteracji pętli tak naprawdę. Najlepszym rozwiazaniem tego problemu jest dodanie sekcji krytycznej, tak aby tylko jeden wątek mógł zmieniać w danej chwili wartość a_shared.

W przypadku zmiennej c_firstprivate nie pojawiają się żadne problemy, ponieważ każdy wątek otrzyma własną kopię tej zmiennej, a wykorzystując *firstprivate()* nadajemy jej od razu wartość początkową. Problem mógłby się pojawić jednak gdybyśmy zamiast *firstprivate()* użyli *private()*. Wtedy zamiast normalnych wartości otrzymamy „Śmieci”, ponieważ zmienna c_firstprivate nie miałąby wartości początkowej.

W przypadku zmiennej e_atomic zależność jaka pojawia się w tym kodzie jest podobna do pierwszej, tutaj również mamy zmienną *shared()* i wykonujemy na niej operacje. Tym razem jednak zamiast sekcji krytycznej używamy dyrektywy atomic.

```
//przed
for(i=0;i<10;i++){
  e_atomic+=omp_get_thread_num();
}

//po
for(i=0;i<10;i++){
  #pragma omp atomic
  e_atomic+=omp_get_thread_num();
}
```

Ostatnia zależność pojawia się przy końcowym wypisywaniu, tutaj wyświetlamy wartości zmiennej zaraz po ich zmianie, dlatego mamy do czynienia z zależnością RAW. Do usunięcia tej zależności można wykorzystać barierę oraz sekcję krytyczną, W ten sposób poprawimy czytelność kodu oraz

gop zabezpieczymy. Bariera zapewni nam, że wszystkie obliczenia będą wykonane przed wyświetleniem.

```
//przed
{
    printf("\nw obszarze równoległym: aktualna liczba watkow %d, moj ID %d\n",
            omp_get_num_threads(), omp_get_thread_num());
    printf("\ta_shared \t= %d\n", a_shared); // RAW i WAR przez inkrementacj w petli
    printf("\tb_private \t= %d\n", b_private);
    printf("\tc_firstprivate \t= %d\n", c_firstprivate);
    printf("\td_local_private = %d\n", d_local_private); // RAW zalezne od a_shared
    printf("\te_atomic \t= %d\n", e_atomic);// RAW i WAR przez zwiekszanie o
omp_get_thread_num()
}
//po
#pragma omp barrier
#pragma omp critical
{
 printf("\nw obszarze równoległym: aktualna liczba watkow %d, moj ID %d\n",
            omp_get_num_threads(), omp_get_thread_num());
 printf("\ta_shared \t= %d\n", a_shared); // RAW i WAR przez inkrementacj w petli
 printf("\tb_private \t= %d\n", b_private);
 printf("\tc_firstprivate \t= %d\n", c_firstprivate);
 printf("\td_local_private = %d\n", d_local_private); // RAW zalezne od a_shared
 printf("\te_atomic \t= %d\n", e_atomic);// RAW i WAR przez zwiekszanie o
omp_get_thread_num()
}
```

W taki oto sposób wynikiem programu dla przykładowo 5 wątków będzie:

```
Kompilator rozpoznaje dyrektywy OpenMP
przed wejsciem do obszaru rownoleglego -  nr_threads 1, thread ID 0
        a_shared        = 1
        b_private       = 2
        c_firstprivate  = 3
        e_atomic        = 5

w obszarze równoległym: aktualna liczba watkow 5, moj ID 2
        a_shared        = 51
        b_private       = 0
        c_firstprivate  = 23
        d_local_private = 4
        e_atomic        = 105

w obszarze równoległym: aktualna liczba watkow 5, moj ID 1
        a_shared        = 51
        b_private       = 0
        c_firstprivate  = 13
        d_local_private = 4
        e_atomic        = 105

w obszarze równoległym: aktualna liczba watkow 5, moj ID 0
        a_shared        = 51
        b_private       = 0
        c_firstprivate  = 3
        d_local_private = 4
        e_atomic        = 105

w obszarze równoległym: aktualna liczba watkow 5, moj ID 4
        a_shared        = 51
        b_private       = 0
        c_firstprivate  = 43
        d_local_private = 4
        e_atomic        = 105

w obszarze równoległym: aktualna liczba watkow 5, moj ID 3
        a_shared        = 51
        b_private       = 0
        c_firstprivate  = 33
        d_local_private = 4
        e_atomic        = 105
po zakonczeniu obszaru rownoleglego:
        a_shared        = 51
        b_private       = 2
        c_firstprivate  = 3
        e_atomic        = 105
```

Wartości zmiennych współdzielonych są identyczne we wszystkich wątkach oraz po zakończeniu programu.

Threadprivate

Przy pomocy threadprivate tworzy prywatne zmienne globalne dla wątków. W ten sposób możemy np. między sekcjami równoległymi przekazywać jakieś wartości w poszczególnych wątkach. Ma to jednak swoje minusy, ponieważ jeśli jedną sekcję równoległą wykonamy przy ilości wątków N, a drugą np. N+3 to te trzy dodatkowe wątki nie będą posiadały żadnej wartości w sobie. W wyniku programu będą posiadały po prostu wartosć 0.

```
static int f_threadprivate;

#pragma omp threadprivate(f_threadprivate)
#pragma omp parallel default(none) shared(a_temp, a_shared, e_atomic) private(b_private)
firstprivate(c_firstprivate )
  {
...
f_threadprivate = omp_get_thread_num();
```

W kodzie utworzona została zmienna statyczna f_threadprivate, która nastepnei przy pomocy dyrektywy *threadprivate()* została przekazana do sekcji równoległej. Tam przypisywane im są numery wątków.

Na końcu programu w osobnej sekcji krytycznej wyświetlane są wartości f_threadprivate.

```
#pragma omp parallel num_threads(8)
printf("\nf_threadprivate = \t%d\n", f_threadprivate);
```

Jak widać pojawia się tutaj klauzula num_threads(8), która ustawia nam ilość wątków na 8, w systemie ustawione są obecnie 5 dlatego według tego co wcześniej napisałem powinny wystąpic 3 odatkowe 0.

```
po zakonczeniu obszaru rownoleglego:
        a_shared        = 51
        b_private       = 2
        c_firstprivate  = 3
        e_atomic        = 105

f_threadprivate =       2

f_threadprivate =       3

f_threadprivate =       0

f_threadprivate =       0

f_threadprivate =       4

f_threadprivate =       0

f_threadprivate =       1

f_threadprivate =       0
```

I tak też się stało, pojawiają się wartości od 0-4 ponieważ mamy 5 wątków numerowanych od 0 oraz dodatkowe 3 wątki, które otrzymały wartość 0.

Task sortowanie

Kolejne zadanie polega na wykorzystaniu dyrektywy Task do tworzenia zadań, które będą wykorzystywane do sortowania tablicy. Program, który analizujemy oraz modyfikujemy wykorzystuje kilka sposobów na sortowanie tablicy. Trzy z nich wykonują się sekwencyjne: systemowe – qsort, przez scalanie, sortowanie szybkie oraz trzy z nich są wariantami równoległymi sortowania przez scalanie.

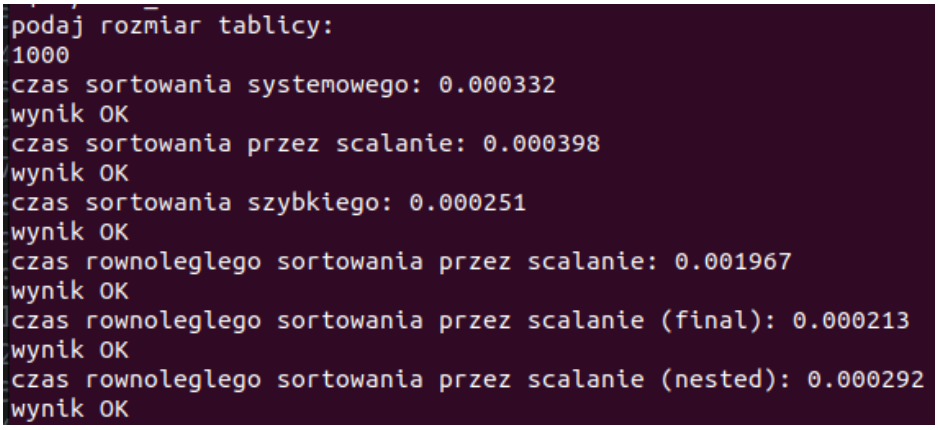
Naszym zadaniem było uzupełnienie sortowania przez scalanie w wersji równoległej z dodatkowym określeniem klauzuli *final()*.

```
if(p<r){
   poziom++;

   int q1=(p+r)/2;

#pragma omp task final( poziom>max_poziom ) default(none) firstprivate(A,p,r,q1,poziom)
   {
    if(omp_in_final())
      sortowanie_szybkie(A,p,q1);
    else
      merge_sort_openmp_2(A,p,q1,poziom);
   }

#pragma omp task final( poziom>max_poziom ) default(none) firstprivate(A,p,r,q1,poziom)
   {
    if(omp_in_final())
      sortowanie_szybkie(A,q1+1,r);
    else
    merge_sort_openmp_2(A,q1+1,r,poziom);
   }

#pragma omp taskwait

   scal(A,p,q1,r);

  }

  return;
```

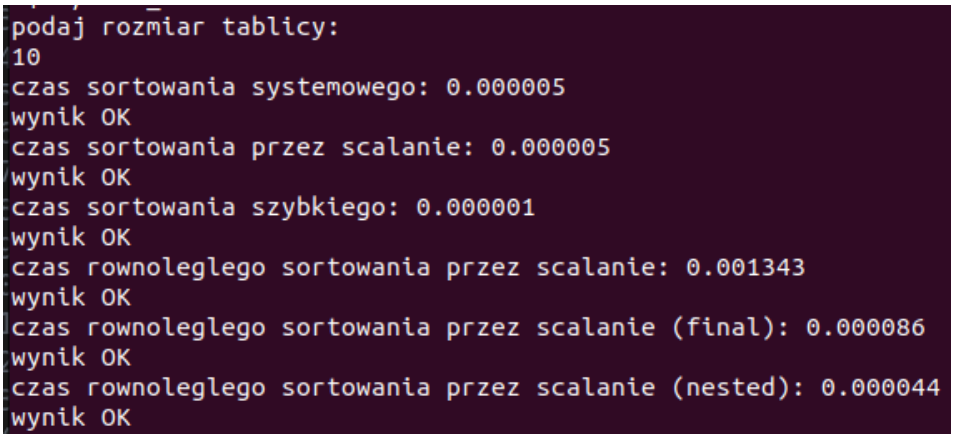
Sortowanie to polega na dzieleniu tablicy na mniejsze, aż osiągniemy tablice jednoelementowe, które następnie będą scalane odpowiednio, aby osiągnąć posortowaną tablicę. W tym przypadku każdy z podziałów wykonywany jest w przestrzeni równoległej z dyrektywą Task, służy ona do tworzenia nowego zadania, które będzie wykonywane równolegle z programem. Jak widać pojawia się również klauzula *final()*, która określa w którym momencie ma skończyć tworzenie nowych zadań. Tutaj służy to do wykorzystania szybkiego sortowania dla mniejszych tablic. Aby sprwadzić czy w naszym zadaniu jest prawdziwy warunek final można wykorzystać *omp_in_final()*.

Pojawia się również dyrektywa taskawait, która oczekuje na zakończenie zadania, aby pójść dalej, dzięki temu pomimo wykonywania równolegle zadań, zadanie rodzic nie skończy się aż jego podzadania nie wykonają swoich czynności.

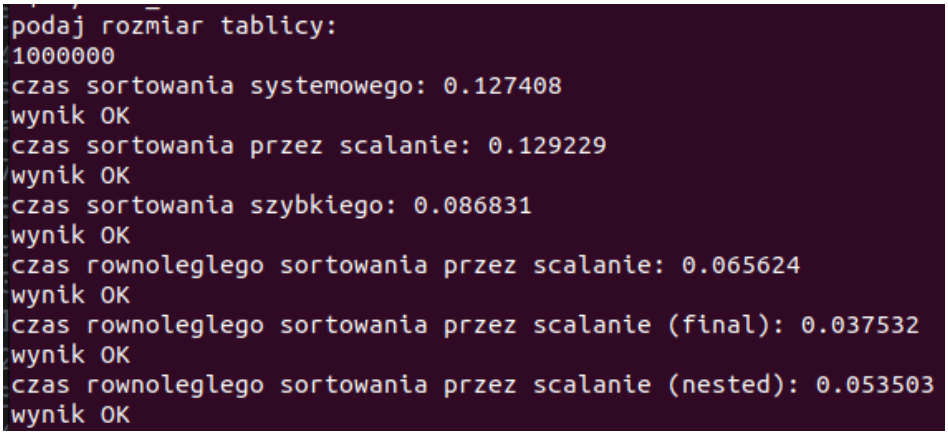
Wynikiem programu dla rozmiaru tablicy 1000:



Jak widać wyniki są OK, dodatkowo możemy porównać czasy wykonania każdej z metod sortowania. Jak można zauważyć dla takiego rozmiaru tablicy osiągneliśmy całkiem spore przyspieszenie dla metody równoległej przez scalanie z final oraz nested. Niestety dla równoległej przez scalanie czas jest gorszy niż dla sekwencyjnych metod. Jest to spowodowane tym, że tworzymy zadania dla wszystkich podziałów tablic, co przy małych tablicach jest nieopłacalne. Dlatego właśnie dla wersji final oraz nested, w których określamy moment stopu podziału mamy lepszy czas, małe tablice szybciej posortujemy sekwencyjnie niż równolegle przez brak narzutu na tworzenie wątków.



Dobrze to widać dla małej tablicy np. 10 elementów. Sekwencyjen metody są kilku/kilkunasto krotnie lepsze od równoległych, ponieważ nie tracą czasu na tworzenie wątków.



Jak widać dla bardzo dużych tablic wykonanie równoległe jest dużo bardziej opłacalne, prawie o połowę zmiejszamy czas sortowania, a w przypadku wykorzystania final aż czterokrotnie.

Można to podsumować, że wykorzystanie zadań do sortowania jest opłącalne w przypadku dużych rozmiarów danych, ponieważ narzut na tworzenie zadań jest niwelowany przez szybkość wykonania, jednak dla małych rozmiarów tworzy nam dodatkowe problemy i opóźnienia.

Task binary search

W tym programie poszukujemy największej wartości w tablicy, ponownie pojawia się kilka metod wykonania z czego część jest równoległa a część sekwencyjna. Naszym zadaniem było napisanie równoległego szukania liniowego maksylanej wartości z wykorzystaniem zadań Task.

```
  double a_max = A[p];
  double a_max_local = a_max;

#pragma omp parallel default(none) shared(a_max) firstprivate(A,p,k,a_max_local)
  {

#pragma omp single
    {
      int num_threads = omp_get_num_threads();
      float n = k-p+1;

      int num_tasks = 2*num_threads;
      int n_loc=ceil(n/num_tasks);

      for(int itask=0; itask<num_tasks; itask++){

          int p_task = p+itask*n_loc;
          if(p_task>k) {
            printf("Error in task decomposition! Exiting.\n");
            exit(0);
          }
          int k_task = p+(itask+1)*n_loc-1;
          if(k_task>k) k_task = k;

#pragma omp task default(none) shared(a_max,A) firstprivate(a_max_local,p_task,k_task)
          {
    a_max_local = search_max(A, p_task, k_task);

#pragma omp critical (cs_a_max)
    {
      if(a_max < a_max_local) a_max = a_max_local;
    }
          } // end task definition

      } // end loop over tasks
    } // end single region
  } // end parallel region

  return(a_max);
```

Na początku pojawia się nowa dyrektywa single, służy ona do okreslenia obliczeń/działań, które mają zostać wykonane tylko raz przez wątki, tab aby każdy wątek nie musiał ich wykonywać, a tylko jeden z nich. Okreslamy w nich podział tablicy na mniejsze elementy, które będą wykonywane przez kolejne zadania. Następnie wewnątrz pętli tworzone są zadania z określonymi dostępami do zmiennych. Zmienna A oraz a_max jest wspólna dla wszystkich wątków, a a_max_local, p_task oraz k_task prywatne dla każdego wątku. Przez to, że wykorzystaliśmy *fisrtprivate()* wartości tych zmiennych są przekazywane do lokalnych kopii w wątku. Nastepnie w każdym zadaniu wykonywana jest metoda *search_max()* na mniejszych tablicach.

Dodatkowym zadniem było napisanie równoległego binarnego szukania,

```
    if (level < max_level) {
      int q = (p + r) / 2;
      double a_max_1, a_max_2;

#pragma omp task final( level>max_level ) shared(a_max_1)
    {
      if(omp_in_final())
        a_max_1 = search_max(A,p,q);
      else
        a_max_1 = bin_search_max_task(A, p, q, level + 1);

    }

#pragma omp task final( level>max_level ) shared(a_max_2)
    {
      if(omp_in_final())
        a_max_2 = search_max(A,q + 1, r);
      else
        a_max_2 = bin_search_max_task(A, q + 1, r, level + 1);
    }

#pragma omp taskwait
      return (a_max_1 < a_max_2) ? a_max_2 : a_max_1;
    } else {
      return bin_search_max(A, p, r);
    }
```

Całość działa podobnie jak przykład sortowania przez scalanie, tutaj również każdy podział jest wykonywany w osobnym zadaniu oraz okreslony jest moment, kiedy ma skończyć podział przy pomocy *final()*.

Wynikiem programu jest:

```
maximal element 249999.950000
time for sequential linear search: 0.011770
maximal element 249999.950000
time for parallel linear search: 0.003175
maximal element 249999.950000
time for parallel linear search (tasks): 0.003863
maximal element 249999.950000
time for sequential binary search: 0.021816
maximal element 249999.950000
time for parallel binary search: 0.010145
```

Jak widać wszystkie metody znalazły ten sam maksymalny element, ale ważniejszy jest czas wykonania, Przy wykorzystaniu równoległych wersji otrzymujemy o ponad połowę szybszy czas wykonania.

Mnożenie macierz – wektor

Dodatkowym zadaniem było zrównoleglenie mnożenia macierzy przez wektor, jak wiemy taka operacja może być czasochłonna przez ilość operacji jakie są wykonywane. Z 4 metod udało mi się zrównoleglić 3 z nich

```
void mat_vec_row_row_decomp(double* a, double* x, double* y, int n)
{
 int i,j;

#pragma omp parallel for default(none) private(j) shared(a, n) firstprivate(y,x)
 for(i=0;i<n;i++){
  y[i]=0.0;
  for(j=0;j<n;j++){
   y[i]+=a[n*i+j]*x[j];
  }
 }
}

void mat_vec_row_col_decomp(double* a, double* x, double* y, int n)
{
 int i,j;

 for(i=0;i<n;i++){
  double temp_y = 0.0;

  #pragma omp parallel for default(none) reduction(+:temp_y) shared(a, x, n, i) private(j)
  for(j = 0; j < n; j++) {
   temp_y += a[n * i + j] * x[j];
  }
  #pragma omp critical
  y[i] = temp_y;
 }
}
```

W przypadku funkcji mat_vec_row_row_decomp zrównoleglamy zewnętrzną funkcję po wierszach. Zasada działania jest bardzo podobna do poprzednich labolatoriów, dlatego nie będę jej szczegółowo opisywał. Największą różnicą jest jednak sposób przekazania x oraz y, każdy wątek otrzymuje swoją prywatną kopię wskaźników y i x, ale wskaźniki te wskazują na te same obszary pamięci co oryginalne wskaźniki przed wejściem do pętli. Oznacza to, że modyfikacje danych, do których wskaźniki y i x wskazują, będą widoczne w oryginalnych danych po zakończeniu pętli.

W przypadku *mat_vec_row_col_decomp()* zrównoleglona została wewnętrzna pętla, a obliczenia są wykonywane z *reduction()*. Dzięki temu każdy wątek otrzymuje własną kopię zmiennej *temp_y*, która pod koniec pętli jest sumowana w sekcji krytycznej do odpowiedniego elementu y.

```
void mat_vec_col_col_decomp(double* a, double* x, double* y, int n)
{

  int i,j;
#pragma omp parallel default(none) private(i, j) shared(a, x, y, n)
{
  double *y_local = malloc(n * sizeof(double));

   #pragma omp for
    for(i = 0; i < n; i++) {
        y_local[i] = 0.0;
    }

  #pragma omp for
  for(j=0;j<n;j++){
    for(i=0;i<n;i++){
      y_local[i]+=a[i+j*n]*x[j];
    }
  }

  #pragma omp critical
  {
      for(i = 0; i < n; i++) {
          y[i] += y_local[i];
      }
  }
}

}
```

W funkcji *mat_vec_col_col_decomp()* wykorzystujemy dodatkową tablicę localną y_local, w której zapisywane są lokalne sumy, a na końcu w krytycznej sekcji dodawane do odpowiedniego elementu tablicy y.

Wnioski

Jak widać przy wykonywaniu obliczeń równolegle trzeba zwrócić uwagę na zależności, nawet nieświadomie możemy napisać program, który będzie nam dawał dobre wyniki, jednak przy innej ilości wątków zaczną się pojawiać problemy z wynikami. Znalezienie tych zależności po zrozumieniu trzech podstawowych rodzajów RAW, WAR oraz WAW jest całkiem łatwe, jednak trzeba mieć świadomość o ich istnieniu.

Jeśli chcemy, aby zmienne przekazane do sekcji równoległej były prywatne, ale otrzymały początkową wartość możemy wykorzystać klauzulę firstprivate(), która tworzy lokalne kopie zmiennej i przypisuje jej poczatkową wartość, gdzie kaluzula private() tworzy tylko lokalne kopie bez wartości.

Threadprivate pozwoli nam utworzyć zmienne globalne dla wątków, które będą przekazywane pomiedzy róznymi sekcjami równoległymi, jednak musimy zwrócić uwagę, czy nie zmienia się ilosć wątków, ponieważ nowe wątki nie będą posiadały żadnej wartości.

Wykorzystanie zadań do wykonywania równolegle jakichś zadań jest praktyką często wykorzystywaną, dzięki temu możemy wykonać jakieś zadanie równolegle z naszym, tak aby nie zatrzymywało nam programu w oczekiwaniu na wynik. Dodatkowo możemy sterować równoległym wykonywaniem zadań i określać warunki kiedy jakaś operacja może być wykonana równolegle, a kiedy nie. W sytuacji, gdy wykonujemy kilka operacji równolegle, ale chcemy aby fragment obliczeń został wykonany tylko raz przez jeden wątek, a nie przez wszystkie możemy wykorzystać dyrektywę single.