Grupa lab.
5

Data ćwiczenia
04.12.2023

Labratorium 9

Imie i nazwisko
Łukasz Sawina

Cel

Celem ćwiczenia było nauczenie się tworzenia oraz implementowania programów równoległych z wykorzystaniem OpenMP.

OpenMP Pętle Simple

Pierwszy program służył do pokazania jak wygląda praca z OpenMP. W programie obliczana jest suma wyrazów tablicy na dwa sposoby. Pierwszy synchroniczny, a drugi do uzupełnienia przez nas równoległy. Ilość wątków w tym programie jak i w kolejnych będzie określana przez zmienne środowiskowe OMP_NUM_THREADS. Dla pierwszego programu przy 4 wątkach musimy zrównoleglić petlę obliczającą sumę tablicy o 18 elementach.

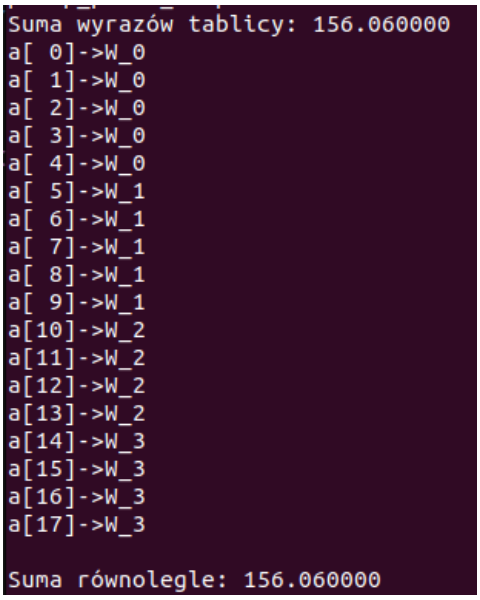
```
  #pragma omp parallel for schedule(dynamic) default(none) shared(a)
reduction(+:suma_parallel) ordered

  for(int i=0;i<WYMIAR;i++) {
    int id_w = omp_get_thread_num();
    suma_parallel += a[i];
    #pragma omp ordered
    printf("a[%2d]->W_%1d  \n",i,id_w);
  }
```

Jak widać przed pętlą wykorzystany jest dyrektywę *parallel for,* parametry w klauzuli *schedule* będą poniżej tłumaczone lepiej, dlatego teraz pomijam jej wytłumaczenie. Klauzula *defalut(none)* wymusza na programiście ręczne określanie dostępu do zmiennych przez wątki, w tym przypadku przy pomocy *shared(a)* informujemy, że wątki będą współdzieliły dostęp do tablicy a, co jest oczywiste, ponieważ to właśnie w niej znajdują się dane do zsumowania. Dodatkowo przy pomocy *reduction(+:suma_parallel)* określamy, że na zmiennej *suma_parallel* będziemy mogli bezpieczenie wykonywać operację dodawania. Działa to na zasadzie tworzenia dla każdego wątku własnej kopii zmiennej *suma_parallel* i operacji na niej, a na końcu w sekcji krytycznej zostanie zrobiona suma poszczególnych prywatnych zmiennych. Mechanizm podobny do tego co robiliśmy na wielu zajęciach, a zautomatyzowany przez jedną klauzulę. Na końcu pojawia się klauzula ordered, która informuje, że w pętli będzie wykonywane coś w kolejności, w naszym przypadku jest to wyświetlanie informacji jaki wątek bierze wartość z jakiego indeksu tablicy.

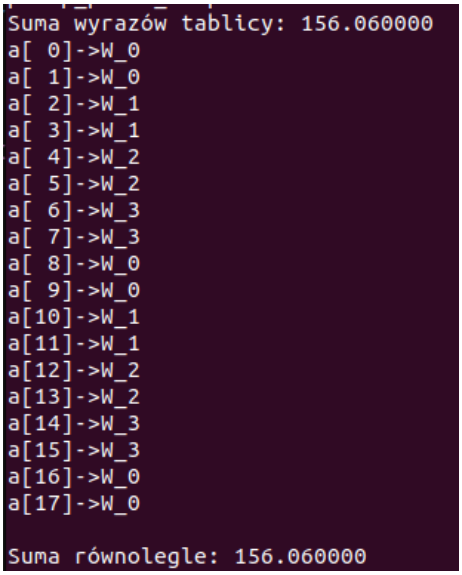
Klauzula schedule() określa jak wątki będą przydzielane do iteracji oraz ile iteracji dostanie jaki wątek. Na rzecz tych zajęć poznaliśmy tylko dwa rodzaje *static* oraz *dynamic*. W przypadku static wątki z góry mają określoną kolejność, iteracje będą wykonywane kolejno przez wątki, domyślnie ilość iteracji zostanie podzielona po równo między wątkami. Dlatego w tym przypadku każdy wątek dostanie po 6 iteracji do wykonania.

Przykładowe wywołanie dla *schedule(static)*



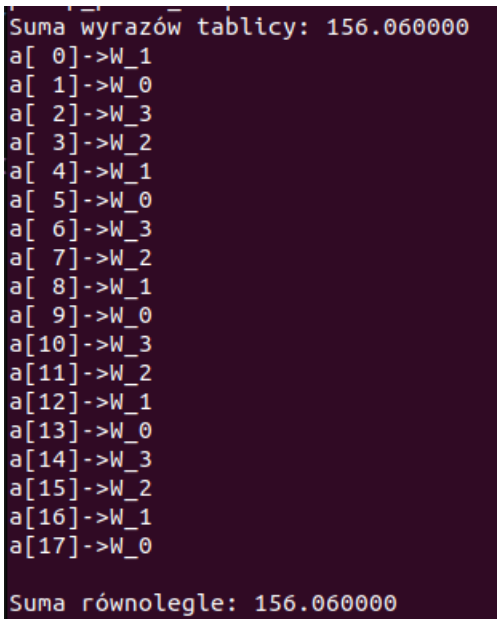
Do klauzuli schedule możemy również dodać ilość iteracji jaką ma wykonać wątek, wstawiając ją po przecinku i tak np. możemy ustawić, aby wątki dostawały nie po 6 a po 2 iteracje.

Przykładowe wywołanie dla *schedule(static,2)*



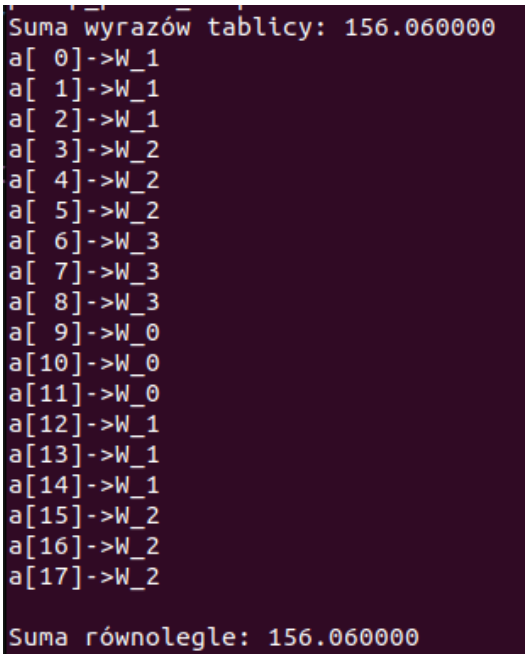
Troszeczkę inaczej to wygląda w przypadku *dynamic*, tutaj iteracje są przydzielane tak jak dostępne są wątki, nie ma z góry określonego, kto robi co. Przy samym dynamic bez określonego rozmiaru, domyślnie każdy wątek dostanie po jednej iteracji, tylko ich kolejność będzie zależna od tego jak szybko wykonają zadanie i kto będzie akurat wolny.

Przykładowe wywołanie dla *schedule(dynamic)*



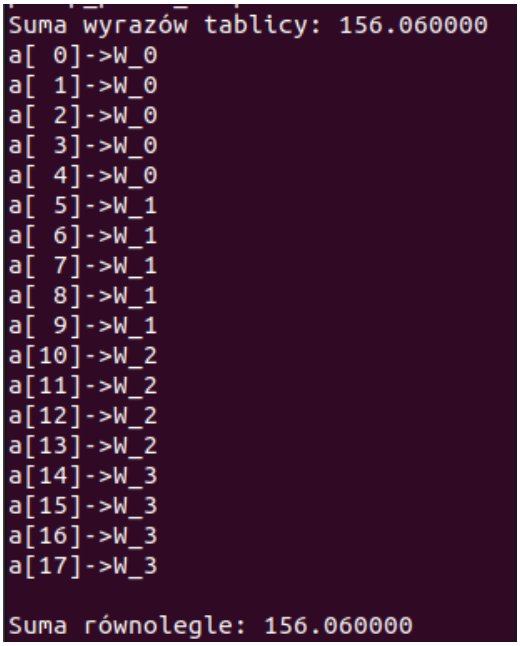
Jak widać kolejność wątków jest różna oraz każdy wątek dostał po jednej iteracji naraz. Przy określeniu wielkości możemy określić ile wątek ma dostać iteracji w danej chwili, ale kolejność wątków będzie ponownie dobierana automatycznie, zależnie od tego który jest wolny.

Przykładowe wywołanie dla *schedule(dynamic,3)*



W tym przypadku wątki dostały po 3 iteracjie, ale ich kolejność jest wymieszana.

W sytuacji jednak jak nie określimy klauzuli *schedule()* po uruchomieniu programu możemy zauważyć po wynikach, że wyglądają identycznie jak dla programu z *schedule(static)*, dlatego można określić, że jest to domyślny sposób podziału.



OpenMP Pętle

Kolejny program działa na podobnej zasadzie, tylko tym razem liczymy sumę tablicy 2D oraz operujemy na 3 wątkach. Na podstawie tego programu będziemy wykonywali różnego dekompozycje.

Dekompozycja wierszowa – zrównoleglenie pętli zewnętrznej

Na początku będziemy zrównoleglali zewnętrzną pętlę, przy dodatkowym określeniu, że każdy wątek dostanie statycznie 2 iteracje po wierszach.

```
double suma_parallel=0.0; int i,j;
  #pragma omp parallel for schedule(static,2) default(none) shared(a) private(j)
reduction(+:suma_parallel) ordered
  for(i=0;i<WYMIAR;i++) {
    int id_w = omp_get_thread_num();
    for(j=0;j<WYMIAR;j++) {
      suma_parallel += a[i][j];
      #pragma omp ordered
      printf("(%1d,%1d)-W_%1d ",i,j,omp_get_thread_num());
    }
    #pragma omp ordered
    printf("\n");
  }
```

Zrównoleglanie pętli się troszeczkę bardziej komplikujem, ponieważ teraz operujemy na dwóch pętlach. Ponieważ zrównoleglamy po zewnętrznej pętli, dlatego tylko tam pojawia się dyrektywa oraz kaluzule do zrównoleglania. Ponownie pojawia się *schedule()* do określenia podziału, *default(none)* do zresetowania dostępu do danych, *shared(a)* oraz ordered. Nową klauzulą jest

private(), określa ona jakie zmienne mają być prywatne dla wątków. W tym przypadku jest to zmienna j, która jest zmienną sterującą drugą pętlą. Nie możemy pozwolić na jej współdzielenie, ponieważ wątki by sobie nadpisywały ją i pomijały iteracje.

Wynikiem programu przy 3 wątkach jest:

Suma wyrazów tablicy: 913.500000

(0,0)-W_0 (0,1)-W_0 (0,2)-W_0 (0,3)-W_0 (0,4)-W_0 (0,5)-W_0 (0,6)-W_0 (0,7)-W_0 (0,8)-W_0 (0,9)-W_0

(1,0)-W_0 (1,1)-W_0 (1,2)-W_0 (1,3)-W_0 (1,4)-W_0 (1,5)-W_0 (1,6)-W_0 (1,7)-W_0 (1,8)-W_0 (1,9)-W_0

(2,0)-W_1 (2,1)-W_1 (2,2)-W_1 (2,3)-W_1 (2,4)-W_1 (2,5)-W_1 (2,6)-W_1 (2,7)-W_1 (2,8)-W_1 (2,9)-W_1

(3,0)-W_1 (3,1)-W_1 (3,2)-W_1 (3,3)-W_1 (3,4)-W_1 (3,5)-W_1 (3,6)-W_1 (3,7)-W_1 (3,8)-W_1 (3,9)-W_1

(4,0)-W_2 (4,1)-W_2 (4,2)-W_2 (4,3)-W_2 (4,4)-W_2 (4,5)-W_2 (4,6)-W_2 (4,7)-W_2 (4,8)-W_2 (4,9)-W_2

(5,0)-W_2 (5,1)-W_2 (5,2)-W_2 (5,3)-W_2 (5,4)-W_2 (5,5)-W_2 (5,6)-W_2 (5,7)-W_2 (5,8)-W_2 (5,9)-W_2

(6,0)-W_0 (6,1)-W_0 (6,2)-W_0 (6,3)-W_0 (6,4)-W_0 (6,5)-W_0 (6,6)-W_0 (6,7)-W_0 (6,8)-W_0 (6,9)-W_0

(7,0)-W_0 (7,1)-W_0 (7,2)-W_0 (7,3)-W_0 (7,4)-W_0 (7,5)-W_0 (7,6)-W_0 (7,7)-W_0 (7,8)-W_0 (7,9)-W_0

(8,0)-W_1 (8,1)-W_1 (8,2)-W_1 (8,3)-W_1 (8,4)-W_1 (8,5)-W_1 (8,6)-W_1 (8,7)-W_1 (8,8)-W_1 (8,9)-W_1

(9,0)-W_1 (9,1)-W_1 (9,2)-W_1 (9,3)-W_1 (9,4)-W_1 (9,5)-W_1 (9,6)-W_1 (9,7)-W_1 (9,8)-W_1 (9,9)-W_1

Suma wyrazw tablicy równolegle: 913.500000

Tym razem wynik nie jest w postaci zrzutu ekranu, ponieważ są na nim zaznaczone kolorami wątki, aby lepiej zauważyć sposób podziału. Jak widać otrzymaliśmy dekompozycję blokową po wierszach.

Dekompozycja kolumnowa – zrównoleglenie pętli wewnętrznej

W tym przypadku zrównoleglamy wewnętrzną pętlę wykorzystując schedule(dynamic). Główną zmianą w kodzie będzie miejsce dyrektywy oraz klauzul do zrównoleglania, jak również dostępu do zmiennych.

```
  for(i=0;i<WYMIAR;i++) {
    int id_w = omp_get_thread_num();
    #pragma omp ordered
    #pragma omp parallel for schedule(dynamic) default(none) shared(a, i)
reduction(+:suma_parallel) ordered
    for(j=0;j<WYMIAR;j++) {
      suma_parallel += a[i][j];
      #pragma omp ordered
      printf("(%1d,%1d)-W_%1d ",i,j,omp_get_thread_num());
    }
    #pragma omp barier
    printf("\n");
  }
```

W tym przypadku wątki muszą współdzielić dostęp do tablicy z danymi a oraz wartość iteratora i.

Wynikiem programu jest:

Suma wyrazów tablicy: 913.500000

(0,0)-W_1 (0,1)-W_0 (0,2)-W_2 (0,3)-W_1 (0,4)-W_0 (0,5)-W_2 (0,6)-W_1 (0,7)-W_0 (0,8)-W_2 (0,9)-W_1

(1,0)-W_2 (1,1)-W_0 (1,2)-W_1 (1,3)-W_2 (1,4)-W_0 (1,5)-W_1 (1,6)-W_2 (1,7)-W_0 (1,8)-W_1 (1,9)-W_2

(2,0)-W_1 (2,1)-W_0 (2,2)-W_2 (2,3)-W_1 (2,4)-W_0 (2,5)-W_2 (2,6)-W_1 (2,7)-W_0 (2,8)-W_2 (2,9)-W_1

(3,0)-W_1 (3,1)-W_0 (3,2)-W_2 (3,3)-W_1 (3,4)-W_0 (3,5)-W_2 (3,6)-W_1 (3,7)-W_0 (3,8)-W_2 (3,9)-W_1

(4,0)-W_2 (4,1)-W_0 (4,2)-W_1 (4,3)-W_2 (4,4)-W_0 (4,5)-W_1 (4,6)-W_2 (4,7)-W_0 (4,8)-W_1 (4,9)-W_2

(5,0)-W_1 (5,1)-W_0 (5,2)-W_2 (5,3)-W_1 (5,4)-W_0 (5,5)-W_2 (5,6)-W_1 (5,7)-W_0 (5,8)-W_2 (5,9)-W_1

(6,0)-W_1 (6,1)-W_0 (6,2)-W_2 (6,3)-W_1 (6,4)-W_0 (6,5)-W_2 (6,6)-W_1 (6,7)-W_0 (6,8)-W_2 (6,9)-W_1

(7,0)-W_1 (7,1)-W_0 (7,2)-W_2 (7,3)-W_1 (7,4)-W_0 (7,5)-W_2 (7,6)-W_1 (7,7)-W_0 (7,8)-W_2 (7,9)-W_1

(8,0)-W_1 (8,1)-W_0 (8,2)-W_2 (8,3)-W_1 (8,4)-W_0 (8,5)-W_2 (8,6)-W_1 (8,7)-W_0 (8,8)-W_2 (8,9)-W_1

(9,0)-W_1 (9,1)-W_0 (9,2)-W_2 (9,3)-W_1 (9,4)-W_0 (9,5)-W_2 (9,6)-W_1 (9,7)-W_0 (9,8)-W_2 (9,9)-W_1

Suma wyrazów tablicy równolegle: 913.500000

Jak można zauważyć w każdym wierszu pojawia się nam pewnego rodzaju podział cykliczny, wątki idą kolejno po sobie, jednak nie w kolejności tylko czasami pomieszane. W tym przypadku zaznaczę jedynie kilka pierwszych wierszy.

Dekompozycja kolumnowa – zrównoleglenie pętli zewnętrznej

W tym przypadku zamieniamy nasze iteratory miejscami, przez co pętla zewnętrzna jest pętlą po kolumnach, a wewnętrzna po wierszach. Dodatkowo dla tego przykładu nie będziemy korzystać z klauzuli *reduction()* tylko ręcznie napiszemy taki mechanizm.

```
  #pragma omp parallel for schedule(static) default(none) shared(suma_parallel, a)
private(i) ordered
  for(j=0;j<WYMIAR;j++) {
    int id_w = omp_get_thread_num();
    double suma_temp=0.0;
    for(i=0;i<WYMIAR;i++) {
      suma_temp += a[i][j];
      #pragma omp ordered
      printf("(%1d,%1d)-W_%1d ",i,j,omp_get_thread_num());
    }

    #pragma omp critical
    suma_parallel += suma_temp;

    #pragma omp ordered
    printf("\n");
  }
```

Zrównoleglenie pętli wygląda identycznie jak dla pierwszego wariantu, jednak brakuje w nim *reduction()*. Zamiast tego w każdej iteracji powstaje prywatna zmienna tymczasowa, do której dodawane są wartości oraz na końcu w sekcji krytycznej sumowane są do wynikowej sumy. Dodatkowo podział jest ustawiony na statyczny z domyślnym rozmiarem, dlatego możemy podejrzewać, że kolumny zostaną podzielone po równo między wątkami.

Wynikiem programu jest:

Suma wyrazów tablicy: 913.500000

(0,0)-W_0 (0,1)-W_0 (0,2)-W_0 (0,3)-W_0 (0,4)-W_0 (0,5)-W_0 (0,6)-W_0 (0,7)-W_0 (0,8)-W_0 (0,9)-W_0

(1,0)-W_0 (1,1)-W_0 (1,2)-W_0 (1,3)-W_0 (1,4)-W_0 (1,5)-W_0 (1,6)-W_0 (1,7)-W_0 (1,8)-W_0 (1,9)-W_0

(2,0)-W_0 (2,1)-W_0 (2,2)-W_0 (2,3)-W_0 (2,4)-W_0 (2,5)-W_0 (2,6)-W_0 (2,7)-W_0 (2,8)-W_0 (2,9)-W_0

(3,0)-W_0 (3,1)-W_0 (3,2)-W_0 (3,3)-W_0 (3,4)-W_0 (3,5)-W_0 (3,6)-W_0 (3,7)-W_0 (3,8)-W_0 (3,9)-W_0

(4,0)-W_1 (4,1)-W_1 (4,2)-W_1 (4,3)-W_1 (4,4)-W_1 (4,5)-W_1 (4,6)-W_1 (4,7)-W_1 (4,8)-W_1 (4,9)-W_1

(5,0)-W_1 (5,1)-W_1 (5,2)-W_1 (5,3)-W_1 (5,4)-W_1 (5,5)-W_1 (5,6)-W_1 (5,7)-W_1 (5,8)-W_1 (5,9)-W_1

(6,0)-W_1 (6,1)-W_1 (6,2)-W_1 (6,3)-W_1 (6,4)-W_1 (6,5)-W_1 (6,6)-W_1 (6,7)-W_1 (6,8)-W_1 (6,9)-W_1

(7,0)-W_2 (7,1)-W_2 (7,2)-W_2 (7,3)-W_2 (7,4)-W_2 (7,5)-W_2 (7,6)-W_2 (7,7)-W_2 (7,8)-W_2 (7,9)-W_2

(8,0)-W_2 (8,1)-W_2 (8,2)-W_2 (8,3)-W_2 (8,4)-W_2 (8,5)-W_2 (8,6)-W_2 (8,7)-W_2 (8,8)-W_2 (8,9)-W_2

(9,0)-W_2 (9,1)-W_2 (9,2)-W_2 (9,3)-W_2 (9,4)-W_2 (9,5)-W_2 (9,6)-W_2 (9,7)-W_2 (9,8)-W_2 (9,9)-W_2

Suma wyrazów tablicy równolegle: 913.500000

W wyniku otrzymaliśmy piękny podział na trzy części. Wygląd jest mylący, ale w tym przypadku otrzymaliśmy dekompozycję blokową dla kolumn, trzeba pamiętać o tym, że nasza tablica została odwrócona. Dlatego po lewej są kolumny a od góry wierwsze.

Dekompozycja 2D

Kolejnym sposobem podziału jest podział 2D, czyli zrównoleglamy jednocześnie obie pętle.

```
  omp_set_nested(1);

  double suma_parallel=0.0; int i,j;
  #pragma omp parallel for schedule(static) default(none) shared(a) private(j)
reduction(+:suma_parallel) ordered
  for(i=0;i<WYMIAR;i++) {
    int id_w = omp_get_thread_num();
    #pragma omp ordered
    #pragma omp parallel for schedule(static) default(none) shared(a,i,id_w)
reduction(+:suma_parallel) ordered
    for(j=0;j<WYMIAR;j++) {
      suma_parallel += a[i][j];
      #pragma omp ordered
      printf("(%1d,%1d)-W_[%1d,%1d] ",i,j,id_w,omp_get_thread_num());
    }
    #pragma omp ordered
    printf("\n");
  }
```

Najważniejsze co trzeba zrobić na początku to włączenie zagnieżdzania zrównoleglania przy pomocy *omp_set_nested(1)*. Dzięki temu wewnątrz zrównoleglonej pętli możemy zrównoleglic kolejną. W tym programie główną trudność sprawia określenie dostępu do zmiennych, pierwsza

pętla musi posiadać *shared(a)* oraz *private(j)*, tak jak było to w pierwszym wariancie. Druga pętla jest bardziej skomplikowana, w tym przypadku musi ona posiadać *schared(a,i,id_w)*.

Wynikiem programu jest:

Suma wyrazów tablicy: 913.500000

(0,0)-W_[0,0] (0,1)-W_[0,0] (0,2)-W_[0,0] (0,3)-W_[0,0] (0,4)-W_[0,1] (0,5)-W_[0,1] (0,6)-W_[0,1] (0,7)-W_[0,2] (0,8)-W_[0,2] (0,9)-W_[0,2]

(1,0)-W_[0,0] (1,1)-W_[0,0] (1,2)-W_[0,0] (1,3)-W_[0,0] (1,4)-W_[0,1] (1,5)-W_[0,1] (1,6)-W_[0,1] (1,7)-W_[0,2] (1,8)-W_[0,2] (1,9)-W_[0,2]

(2,0)-W_[0,0] (2,1)-W_[0,0] (2,2)-W_[0,0] (2,3)-W_[0,0] (2,4)-W_[0,1] (2,5)-W_[0,1] (2,6)-W_[0,1] (2,7)-W_[0,2] (2,8)-W_[0,2] (2,9)-W_[0,2]

(3,0)-W_[0,0] (3,1)-W_[0,0] (3,2)-W_[0,0] (3,3)-W_[0,0] (3,4)-W_[0,1] (3,5)-W_[0,1] (3,6)-W_[0,1] (3,7)-W_[0,2] (3,8)-W_[0,2] (3,9)-W_[0,2]

(4,0)-W_[1,0] (4,1)-W_[1,0] (4,2)-W_[1,0] (4,3)-W_[1,0] (4,4)-W_[1,1] (4,5)-W_[1,1] (4,6)-W_[1,1] (4,7)-W_[1,2] (4,8)-W_[1,2] (4,9)-W_[1,2]

(5,0)-W_[1,0] (5,1)-W_[1,0] (5,2)-W_[1,0] (5,3)-W_[1,0] (5,4)-W_[1,1] (5,5)-W_[1,1] (5,6)-W_[1,1] (5,7)-W_[1,2] (5,8)-W_[1,2] (5,9)-W_[1,2]

(6,0)-W_[1,0] (6,1)-W_[1,0] (6,2)-W_[1,0] (6,3)-W_[1,0] (6,4)-W_[1,1] (6,5)-W_[1,1] (6,6)-W_[1,1] (6,7)-W_[1,2] (6,8)-W_[1,2] (6,9)-W_[1,2]

(7,0)-W_[2,0] (7,1)-W_[2,0] (7,2)-W_[2,0] (7,3)-W_[2,0] (7,4)-W_[2,1] (7,5)-W_[2,1] (7,6)-W_[2,1] (7,7)-W_[2,2] (7,8)-W_[2,2] (7,9)-W_[2,2]

(8,0)-W_[2,0] (8,1)-W_[2,0] (8,2)-W_[2,0] (8,3)-W_[2,0] (8,4)-W_[2,1] (8,5)-W_[2,1] (8,6)-W_[2,1] (8,7)-W_[2,2] (8,8)-W_[2,2] (8,9)-W_[2,2]

(9,0)-W_[2,0] (9,1)-W_[2,0] (9,2)-W_[2,0] (9,3)-W_[2,0] (9,4)-W_[2,1] (9,5)-W_[2,1] (9,6)-W_[2,1] (9,7)-W_[2,2] (9,8)-W_[2,2] (9,9)-W_[2,2]

Suma wyrazów tablicy równolegle: 913.500000

Jak widać w wyniku całość zostałą podzielona na mniejsze bloczki, rozmiar tych bloków jest obliczony domyślnie, ponieważ dla jednej jak i drugiej pętli wykorzystana została klauzula *schedule(static)*.

Dekompozycja wierszowa – wykorzystanie tablicy

W tym programie wykonujemy dekompozycję wierszową, ale nasza tablica będzie odwrócona (zamienione miejscami i oraz j), dodatkowo nie korzystamy z reduction(), a tworzymy tablicę z wynikami dla poszczególnych wątków.

```
double tab_lok_sum[3];
 for(int i=0;i<3;i++)
   tab_lok_sum[i] = 0;

 for(j=0;j<WYMIAR;j++) {
  int id_w = omp_get_thread_num();
  #pragma omp ordered
  #pragma omp parallel for schedule(static,1) default(none) shared(a, tab_lok_sum, j,
id_w) ordered
   for(i=0;i<WYMIAR;i++) {
    tab_lok_sum[id_w] += a[i][j];
    #pragma omp ordered
    printf("(%1d,%1d)-W_%1d ",j,i,omp_get_thread_num());
   }
   printf("\n");
  }

 for(i = 0; i < 3; i++)
   suma_parallel += tab_lok_sum[i];
```

Wynikiem programu jest:

Suma wyrazów tablicy: 913.500000

(0,0)-W_0 (0,1)-W_1 (0,2)-W_2 (0,3)-W_0 (0,4)-W_1 (0,5)-W_2 (0,6)-W_0 (0,7)-W_1 (0,8)-W_2 (0,9)-W_0

(1,0)-W_0 (1,1)-W_1 (1,2)-W_2 (1,3)-W_0 (1,4)-W_1 (1,5)-W_2 (1,6)-W_0 (1,7)-W_1 (1,8)-W_2 (1,9)-W_0

(2,0)-W_0 (2,1)-W_1 (2,2)-W_2 (2,3)-W_0 (2,4)-W_1 (2,5)-W_2 (2,6)-W_0 (2,7)-W_1 (2,8)-W_2 (2,9)-W_0

(3,0)-W_0 (3,1)-W_1 (3,2)-W_2 (3,3)-W_0 (3,4)-W_1 (3,5)-W_2 (3,6)-W_0 (3,7)-W_1 (3,8)-W_2 (3,9)-W_0

(4,0)-W_0 (4,1)-W_1 (4,2)-W_2 (4,3)-W_0 (4,4)-W_1 (4,5)-W_2 (4,6)-W_0 (4,7)-W_1 (4,8)-W_2 (4,9)-W_0

(5,0)-W_0 (5,1)-W_1 (5,2)-W_2 (5,3)-W_0 (5,4)-W_1 (5,5)-W_2 (5,6)-W_0 (5,7)-W_1 (5,8)-W_2 (5,9)-W_0

(6,0)-W_0 (6,1)-W_1 (6,2)-W_2 (6,3)-W_0 (6,4)-W_1 (6,5)-W_2 (6,6)-W_0 (6,7)-W_1 (6,8)-W_2 (6,9)-W_0

(7,0)-W_0 (7,1)-W_1 (7,2)-W_2 (7,3)-W_0 (7,4)-W_1 (7,5)-W_2 (7,6)-W_0 (7,7)-W_1 (7,8)-W_2 (7,9)-W_0

(8,0)-W_0 (8,1)-W_1 (8,2)-W_2 (8,3)-W_0 (8,4)-W_1 (8,5)-W_2 (8,6)-W_0 (8,7)-W_1 (8,8)-W_2 (8,9)-W_0

(9,0)-W_0 (9,1)-W_1 (9,2)-W_2 (9,3)-W_0 (9,4)-W_1 (9,5)-W_2 (9,6)-W_0 (9,7)-W_1 (9,8)-W_2 (9,9)-W_0

Suma wyrazów tablicy równolegle: 913.500000

Ponownie otrzymaliśmy dekompozycję cykliczną po wierszach.

Wnioski

Wykorzystując mechanizm OpenMP możemy sobie ułatwić pracę, ponieważ wtedy to nie programista jest odpowiedzialny za tworzenie wątków, uruchamianie ich oraz zarządzanie, tylko sam mechanizm omp, naszym zadaniem jest tylko sterować nim ustawiajac odpowiednie klauzule oraz dyrektywy.

Chociaż składnia na początku może być bardzo niezrozumiała tak po krótkiej praktyce szybko da się zrozumieć jak poprwanie zrównoleglać program, jak przydzielać do niego zmienne oraz jak ustawiać wykorzystanie wątków.

Główne różnice między staic a dynamic można już zrozumieć przez ich nazwy, static będzie przydzielał statycznie zadania dla wątków, jak sprawdziliśmy wcześniej, domyślnie podzieli całą pulę zadań porówno pomiędzy wątki i zostaną one wykonane kolejno po sobie. W przypadku dynamic jest troszeczkę inaczej, mechanizm działania jest podobny do puli wątków, który dynamicznie przydziela zadania do wolnych wątków. W tym przypadku domyślnie otrzymują po jednym zadaniu.

OpenMP dodatkowo udostępnia bardzo przydatną klauzulę *reduction()*, która dużo rzeczy robi za nas, tworzenie prywatnych zmiennych, operacja na nich oraz sumowanie w sekcji krytycznej zajmuje kilka linijek kodu, a dzięki *reduction()* jest to dosłownie jedna kaluzula.

Oczywiście OpenMP daje nam opcje pracy z sekcją krytyczną, można to zrobić przy pomocy kilku różnych instrukcji, ale przede wszystkim dzięki *critical*, które określa nam sekcję krytyczną, OpenMP dodatkwo udostępnia mechanizm podobny do mutexów *omp_set_lock()* oraz *omp_unset_lock()*, dlatego nie jesteśmy ograniczeni jedynie do *reduction()*.

Z przydatnych rzeczy OpenMP udostępnia dyrektywy *ordered* do wykonywania w kolejności oraz *barier* do oczekiwania aż wszystkie wątki trafią do bariery i dopiero wtedy pójdą dalej.