

| Akademia Nauk Stosowanych w Nowym Sączu | | |
|---|---------------------|-------------|
| Programowanie współbieżne i rozproszone | | |
| PWIR_10 | | |
| Imię i Nazwisko: Karol Wolski | Ocena sprawozdania: | Zaliczenie: |
| | | |
| Data: 09.05.2023 | Grupa: L3 | |

PWIR_05_01.cpp

Zad 1

Dopisz dyrektywę num_threads do programu z PWIR_02_01.cpp. Przetestuj czas wykonywania programu dla dwóch i więcej wątków.

```

1  #include <stdio>
2  #include <stdint>
3  #include <stdlib>
4  #include <chrono>
5  #include <assert.h>
6  #include <windows.h>
7  #include <omp.h>
8
9  void DoSomethingFast() {
10     Sleep(1000);
11 }
12
13 void DoSomethingLong() {
14     Sleep(6000);
15 }
16
17 int main() {
18     uint8_t id;
19
20     auto start = std::chrono::high_resolution_clock::now();
21     DoSomethingFast();
22     auto end = std::chrono::high_resolution_clock::now();
23
24     printf("Szybkie wykonanie: %llu ms\r\n",
25           std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count());
26
27     start = std::chrono::high_resolution_clock::now();
28     DoSomethingLong();
29     end = std::chrono::high_resolution_clock::now();
30
31     printf("Długie wykonanie: %llu ms\r\n",
32           std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count());
33
34     start = std::chrono::high_resolution_clock::now();
35     #pragma omp parallel num_threads(2) private(id)
36     {
37         id = omp_get_thread_num();
38
39         if (id % 2) {
40             DoSomethingLong();
41         }
42         else {
43             DoSomethingFast();
44         }
45
46         printf("Wątek %d zakończył pracę i oczekuje na barierze\n", id);
47
48         #pragma omp barrier
49         printf("Wątek %d zakończył pracę i zakończył już wykonanie\n", id);
50     }
51
52     end = std::chrono::high_resolution_clock::now();
53     printf("Równoległe wykonanie z 2 wątkami: %llu ms\r\n",
54           std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count());
55
56     start = std::chrono::high_resolution_clock::now();
57     #pragma omp parallel num_threads(4) private(id)
58     {
59         id = omp_get_thread_num();
60
61         if (id % 2) {
62             DoSomethingLong();
63         }
64         else {
65             DoSomethingFast();
66         }
67
68         printf("Wątek %d zakończył pracę i oczekuje na barierze\n", id);
69
70         #pragma omp barrier
71     }

```

```

72     printf("Watek %d zakonczyl prace i zakonczyl juz wykonanie\n", id);
73 }
74
75     end = std::chrono::high_resolution_clock::now();
76     printf("Rownolegle wykonanie z 4 watkami: %llu ms\n",
77           std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count());
78
79     start = std::chrono::high_resolution_clock::now();
80     #pragma omp parallel num_threads(8) private(id)
81     {
82         id = omp_get_thread_num();
83
84         if (id % 2) {
85             DoSomethingLong();
86         }
87         else {
88             DoSomethingFast();
89         }
90
91         printf("Watek %d zakonczyl prace i oczekuje na barierze\n", id);
92     }
93     #pragma omp barrier
94     printf("Watek %d zakonczyl prace i zakonczyl juz wykonanie\n", id);
95 }
96
97     end = std::chrono::high_resolution_clock::now();
98     printf("Rownolegle wykonanie z 8 watkami: %llu ms\n",
99           std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count());
100
101     return 0;
102 }
103
104

```



Konsola debugowania progra



Szybkie wykonanie: 1009 ms

Dlugie wykonanie: 6012 ms

Watek 0 zakonczyl prace i oczekuje na barierze

Watek 1 zakonczyl prace i oczekuje na barierze

Watek 1 zakonczyl prace i zakonczyl juz wykonanie

Watek 0 zakonczyl prace i zakonczyl juz wykonanie

Rownolegle wykonanie z 2 watkami: 6013 ms

Watek 2 zakonczyl prace i oczekuje na barierze

Watek 0 zakonczyl prace i oczekuje na barierze

Watek 1 zakonczyl prace i oczekuje na barierze

Watek 3 zakonczyl prace i oczekuje na barierze

Watek 1 zakonczyl prace i zakonczyl juz wykonanie

Watek 3 zakonczyl prace i zakonczyl juz wykonanie

Watek 0 zakonczyl prace i zakonczyl juz wykonanie

Watek 2 zakonczyl prace i zakonczyl juz wykonanie

Rownolegle wykonanie z 4 watkami: 6013 ms

Watek 6 zakonczyl prace i oczekuje na barierze

Watek 2 zakonczyl prace i oczekuje na barierze

Watek 0 zakonczyl prace i oczekuje na barierze

Watek 4 zakonczyl prace i oczekuje na barierze

Watek 3 zakonczyl prace i oczekuje na barierze

Watek 1 zakonczyl prace i oczekuje na barierze

Watek 7 zakonczyl prace i oczekuje na barierze

Watek 5 zakonczyl prace i oczekuje na barierze

Watek 7 zakonczyl prace i zakonczyl juz wykonanie

Watek 1 zakonczyl prace i zakonczyl juz wykonanie

Watek 3 zakonczyl prace i zakonczyl juz wykonanie

Watek 5 zakonczyl prace i zakonczyl juz wykonanie

Watek 2 zakonczyl prace i zakonczyl juz wykonanie

Watek 4 zakonczyl prace i zakonczyl juz wykonanie

Watek 0 zakonczyl prace i zakonczyl juz wykonanie

Watek 6 zakonczyl prace i zakonczyl juz wykonanie

Rownolegle wykonanie z 8 watkami: 6014 ms

Wątek natrafiając na `#pragma omp barrier` zostaje wstrzymany i będzie czekać na pozostałe, dopóki każdy z wątków nie dotrze do bariery. Jest to bardzo przydatna dyrektywa pozwalająca na ewentualną synchronizację wątków.

PWIR_06_01.cpp

Zad 1

Przetestuj działanie _01 z klauzulą `nowait` oraz bez. Sprawdź również działanie na większej ilości wątków.

Program z klauzulą `nowait`:

2 wątki

```
Konsola debugowania progra x + v
Sections - Thread 0 working...
Iteration 0 execute thread 0.
Iteration 1 execute thread 0.
Iteration 2 execute thread 0.
Iteration 3 execute thread 0.
Iteration 4 execute thread 0.
Sections - Thread 1 working...
Iteration 5 execute thread 1.
Iteration 6 execute thread 1.
Iteration 7 execute thread 1.
Iteration 8 execute thread 1.
Iteration 9 execute thread 1.
Parallel normal way 6035 ms
```

4 wątki

```
Konsola debugowania progra x + v
Iteration 3 execute thread 1.
Iteration 8 execute thread 3.
Iteration 4 execute thread 1.
Iteration 9 execute thread 3.
Iteration 5 execute thread 1.
Sections - Thread 0 working...
Iteration 0 execute thread 0.
Iteration 1 execute thread 0.
Iteration 2 execute thread 0.
Sections - Thread 2 working...
Iteration 6 execute thread 2.
Iteration 7 execute thread 2.
Parallel normal way 4821 ms
```

Program bez klauzuli `nowait`:

2 wątki

```
Konsola debugowania progra x + v
Sections - Thread 0 working...
Sections - Thread 1 working...
Iteration 5 execute thread 1.
Iteration 0 execute thread 0.
Iteration 6 execute thread 1.
Iteration 1 execute thread 0.
Iteration 7 execute thread 1.
Iteration 2 execute thread 0.
Iteration 3 execute thread 0.
Iteration 8 execute thread 1.
Iteration 9 execute thread 1.
Iteration 4 execute thread 0.
Parallel normal way 6034 ms
```

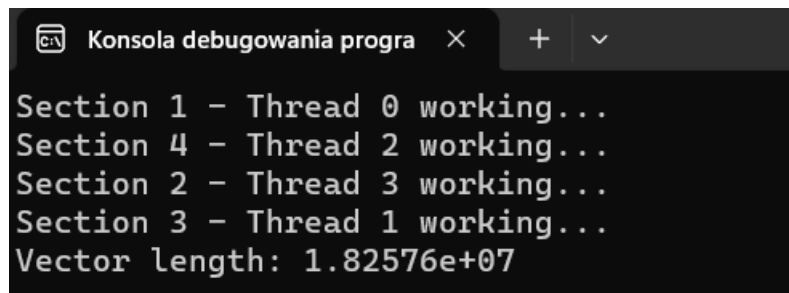
4 wątki

```
Konsola debugowania progra x + v
Sections - Thread 0 working...
Sections - Thread 1 working...
Iteration 3 execute thread 1.
Iteration 6 execute thread 2.
Iteration 0 execute thread 0.
Iteration 8 execute thread 3.
Iteration 1 execute thread 0.
Iteration 4 execute thread 1.
Iteration 9 execute thread 3.
Iteration 7 execute thread 2.
Iteration 2 execute thread 0.
Iteration 5 execute thread 1.
Parallel normal way 5226 ms
```

Klauzula NOWAIT umożliwia wątkowi kontynuowanie wykonywania regionu równoległego bez czekania na zakończenie regionu przez inne wątki w zespole. Innymi słowy, tłumi domniemaną barierę na końcu obszaru równoległego. W tym przypadku istnieje zmienna, która jest zapisywana wewnątrz regionu równoległego i odczytywana za regionem, bez żadnej BARIERY między końcem regionu równoległego a odczytem. Jak widzimy w przykładzie powyżej, użycie klauzuli NOWAIT znacznie przyspieszyło program, gdyż wątki nie musiały na siebie czekać. Korzystając z klauzuli nowait, wątki mogą pracować niezależnie od siebie, bez konieczności oczekiwania na innych wątków. Każdy wątek może przejść do swojego fragmentu kodu bez oczekiwania na resztę wątków.

Zad 2

```
1  #include <iostream>
2  #include <cmath>
3  #include <omp.h>
4
5  double calculateVectorLength(double* vector, int size) {
6      double length = 0.0;
7
8      #pragma omp parallel num_threads(4)
9      {
10         int tid = omp_get_thread_num();
11         double localLength = 0.0;
12
13         #pragma omp sections
14         {
15             #pragma omp section
16             {
17                 for (int i = 0; i < size / 4; i++) {
18                     localLength += vector[i] * vector[i];
19                 }
20                 std::cout << "Section 1 - Thread " << tid << " working..." << std::endl;
21             }
22
23             #pragma omp section
24             {
25                 for (int i = size / 4; i < size / 2; i++) {
26                     localLength += vector[i] * vector[i];
27                 }
28                 std::cout << "Section 2 - Thread " << tid << " working..." << std::endl;
29             }
30
31             #pragma omp section
32             {
33                 for (int i = size / 2; i < 3 * size / 4; i++) {
34                     localLength += vector[i] * vector[i];
35                 }
36                 std::cout << "Section 3 - Thread " << tid << " working..." << std::endl;
37             }
38
39             #pragma omp section
40             {
41                 for (int i = 3 * size / 4; i < size; i++) {
42                     localLength += vector[i] * vector[i];
43                 }
44                 std::cout << "Section 4 - Thread " << tid << " working..." << std::endl;
45             }
46         }
47
48         #pragma omp critical
49         {
50             length += localLength;
51         }
52
53         return std::sqrt(length);
54     }
55 }
56
57 int main() {
58     const int size = 100000;
59     double vector[size];
60
61     // Inicjalizacja wektora
62     for (int i = 0; i < size; i++) {
63         vector[i] = i + 1;
64     }
65
66     // Obliczenie długości wektora na czterech wątkach
67     double length = calculateVectorLength(vector, size);
68
69     std::cout << "Vector length: " << length << std::endl;
70
71     return 0;
72 }
```



```
Section 1 - Thread 0 working...
Section 4 - Thread 2 working...
Section 2 - Thread 3 working...
Section 3 - Thread 1 working...
Vector length: 1.82576e+07
```

W tym programie używamy dyrektywy `#pragma omp parallel` do utworzenia równoległego obszaru, w którym obliczana jest długość wektora. Używamy `num_threads(4)` aby ustawić liczbę wątków na cztery. Wewnątrz równoległego obszaru używamy dyrektywy `#pragma omp sections`, a wewnątrz niej umieszczamy cztery dyrektywy `#pragma omp section`. Każda sekcja wykonuje obliczenia dla odpowiedniego zakresu elementów wektora. Wyświetlamy także informacje o pracy poszczególnych wątków w każdej sekcji. Następnie, używamy dyrektywy `#pragma omp critical` do synchronizacji dostępu do zmiennej `length`, która przechowuje sumę lokalnych długości obliczonych przez wątki. Na końcu zwracamy pierwiastek kwadratowy z sumy długości wektora jako wynik obliczeń.