Przetwarzanie współbieżne

Laboratorium 7

Jan Bartula

# Zadanie 1

## Przygotowanie do zadania

Aby móc przystąpić do wykonywania zadań zostały przygotowane klasy pomocnicza CSVFile

Umożliwia ona łatwiejsze wypisywanie do plików **.csv** w celu zaimportowania wyników do EXCEL, który umożliwi prezentację wyników w formie wykresów.

## Generowanie liczb

std::random\_device rd{};

std::mt19937 gen{rd()};

static double random\_number(double min = -99, double max = 99) {

std::uniform\_real\_distribution<double> distribution(min, max);

return distribution(gen);

}

## Mnożenie wektorów:

double multiplicate(double \*v1, double \*v2, long int size) {

double product = 0;

for (long int i = 0; i < size; i++) {

product += v1[i] \* v2[i];

}

return product;

}

## Poszczególne wersje programów zawierają

Brak dyrektrywy

#pragma omp simd

#pragma omp parallel for schedule(static) num\_threads(6) reduction(+: product)

## Uruchomienie

Vector 1 generated

Vector 2 generated

SIZE: 10

SIZE: 100

SIZE: 1000

SIZE: 10000

SIZE: 100000

SIZE: 1000000

SIZE: 10000000

SIZE: 100000000

SIZE: 1000000000

### Wyniki

Jak możemy zauważyć dla wielkości wektorów do SIMD jest najszybszym rozwiązaniem.

Jednak dla większych wielkości wektorów, zrównoleglenie przy użyciu PARALLEL REDUCTION, na największej wielkości wektora jest to ponad 3 razy szybciej z użyciem REDUCTION.

### Tabela

|  |  |  |  |
| --- | --- | --- | --- |
| SIZE | NONE | SIMD | REDUCTION |
| 10 | 0.0000011 | 0.0000002 | 0.0002878 |
| 100 | 0.0000007 | 0.0000004 | 0.0000022 |
| 1000 | 0.0000030 | 0.0000025 | 0.0000023 |
| 10000 | 0.0000294 | 0.0000249 | 0.0000074 |
| 100000 | 0.0003936 | 0.0002404 | 0.0001736 |
| 1000000 | 0.0031423 | 0.0024408 | 0.0006840 |
| 10000000 | 0.0253162 | 0.0241869 | 0.0059833 |
| 100000000 | 0.2482280 | 0.2512420 | 0.0644440 |
| 1000000000 | 2.5155300 | 2.5193500 | 0.7095530 |

# Zadanie 2

## Funkcja obliczająca N-ty element ciągu Fibonacciego

long int fibonacci\_none(long int n) {  
 long int result, n\_1, n\_2;  
 if (n < 2) {  
 result = n;  
 } else {  
 n\_1 = fibonacci\_none(n - 1);  
 n\_2 = fibonacci\_none(n - 2);  
 result = n\_1 + n\_2;  
 }  
 return result;  
}

## Funkcja obliczająca N-ty element ciągu Fibonacciego z Task-ami

long int fibonacci\_task(long int n) {  
 long int result, n\_1, n\_2;  
 if (n < 2) {  
 result = n;  
 } else {  
#pragma omp task shared(n\_1)  
 n\_1 = fibonacci\_task(n - 1);  
  
#pragma omp task shared(n\_2)  
 n\_2 = fibonacci\_task(n - 2);  
  
#pragma omp taskwait result = n\_1 + n\_2;  
 }  
 return result;  
}

## Wykres

## Porównanie funkcji z taskami a bez nich (różnica w czasie)

## Podsumowanie

Jak możemy zauważyć dla elementów powyżej 31~33, różnica pomiędzy pętli z taskami a bez nich, zaczyna mocno spowalniać ze względu na generowanie olbrzymich ilości tasków.

## Funkcja obliczająca N-ty element ciągu Fibonacciego z Task-ami z ograniczeniem N=31

## Podsumowanie

Jak możemy zauważyć po zastosowaniu tej poprawki, funkcja z użyciem tasków do < 31 elementu ciągu, jest praktycznie tak samo szybkie jak całkowicie bez ich użycia.

Można stwierdzić, że zrównoleglenie dla ciągu Fibonacciego nie jest tak naprawdę potrzebne, bo nie przynosi większych korzyści.