

Zadanie 1 - Macierze OMP

Proszę sprawdzić jak różni się czas mnożenia i dodawania macierzy. Który problem, dodawanie czy mnożenie wart jest zrównoleglenia?

Mnożenie jest operacją arytmetyczną bardziej kosztowną od dodawania, więc to właśnie ta operacja jest bardziej warta zrównoleglenia. Ponadto, mnożenie macierzy samo w sobie jest bardziej skomplikowane algorytmicznie niż ich dodawanie.

Jaki wpływ na zrównoleglenie będzie miało zastosowanie poniższej dyrektywy OpenMP:

```
1 #pragma omp parallel for default(shared)
```

a jaka dyrektywa:

```
1 #pragma omp parallel for default(none) shared(A, B, C) firstprivate(rozmiar) private(i, j)
```

Klauzula default określa tryb współdzielenia zmiennych w obszarze równoległym OpenMP. Wartość 'shared', która jest wartością domyślną, oznacza że każda zmienna znajdująca się w obszarze równoległym będzie traktowana jak gdyby była oznaczona klauzulą shared. Wartość 'none' oznacza, że każda zmienna zadeklarowana poza obszarem równoległym, nieokreślona jedną z klauzul: private, shared, reduction, firstprivate, lastprivate i użyta w tym obszarze spowoduje błędy kompilacji.

Pierwsza postać dyrektywy określa wszystkie zmienne obszaru równoległego jako współdzielone między wątkami. Druga postać dyrektywy określa zmienne A, B i C jako współdzielone, zmienną rozmiar jako prywatną o wartości wprowadzanej do obszaru równoległego i zmienne i, j jako zmienne prywatne bez wartości początkowej, a każde inne w obszarze równoległym wykorzystywane spowodują błędy kompilacji.

Poniżej został umieszczony wycinek kodu odpowiedzialny za mnożenie macierzy.

```
1 const DoubleMatrix *multiplyMatrices(const DoubleMatrix &A, const
   DoubleMatrix &B, unsigned int threadCount)
2 {
3     assertSizeEqual(A, B);
4
5     DoubleMatrix *C = new DoubleMatrix();
6     fillMatrixWithZeros(*C, A.size());
7     unsigned int rowSize = sqrt(A.size());
8
9     #pragma omp parallel for default(none) shared(A, B, C) firstprivate(
        rowSize) num_threads(threadCount)
10    for (unsigned int i = 0; i < rowSize; ++i)
11    {
12        //double local = 0.0;
13        for (unsigned int j = 0 ; j < rowSize; ++j)
14        {
15            C->at(i * rowSize + j) = A.at(i * rowSize + j) * B.at(i + j *
                rowSize);
16            //local += A.at(i * rowSize + j) * B.at(i + j * rowSize);
```

```

17     }
18 }
19
20 return C;
21 }

```

Poniżej został umieszczony wycinek kodu odpowiedzialny za sumowania macierzy.

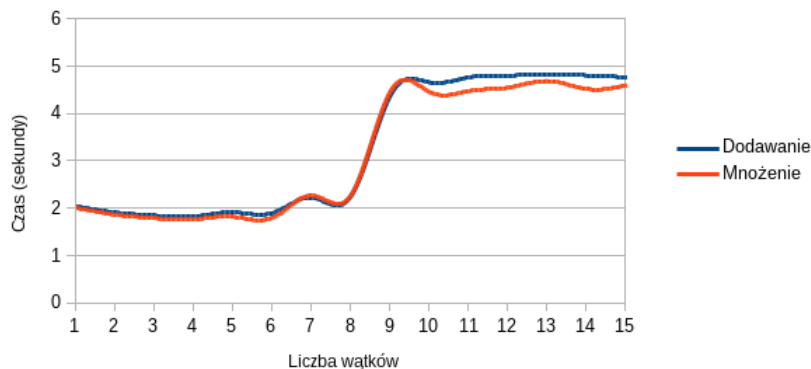
```

1 const DoubleMatrix *sumMatrices(const DoubleMatrix &A, const DoubleMatrix &
  B, unsigned int threadCount)
2 {
3     assertSizeEqual(A, B);
4
5     DoubleMatrix *C = new DoubleMatrix();
6     fillMatrixWithZeros(*C, A.size());
7
8     #pragma omp parallel for default(none) shared(A, B, C) num_threads(
9         threadCount)
10    for (unsigned int i = 0; i < A.size(); i++)
11    {
12        C->at(i) = A.at(i) + B.at(i);
13    }
14
15    return C;
16 }

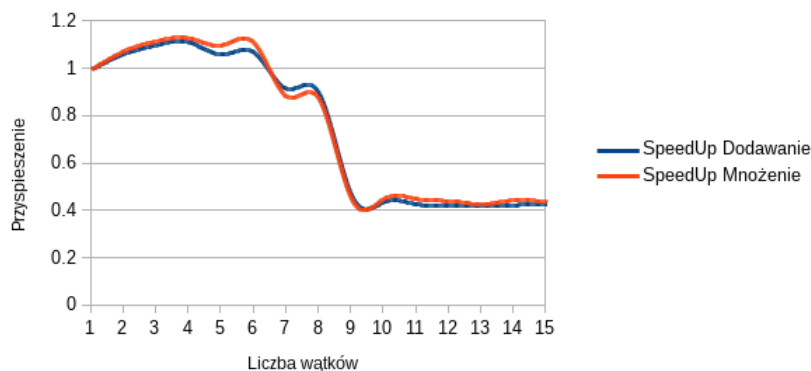
```

Poniżej wykresy przedstawiający rezultat przeprowadzonych operacji na wątkach.

Czas obliczeń w zależności od liczby wątków



Przyspieszenie obliczeń w zależności od liczby wątków



Wnioski

Przy użyciu OpenMP możemy zrównoleglić każdy powtarzalny fragment kodu uzyskując przy tym wysoką wydajność na maszynach wielordzeniowych. Wraz ze wzrostem liczby wątków obserwujemy spadek czasu potrzebnego na wykonanie programu. Ten wzrost nie jest jednak liniowy - dla pewnej liczby wątków funkcja przyspieszenia stabilizuje się. Jest to związane z osiągnięciem stanu, w którym tworzenie kolejnych wątków, ich synchronizacja i zarządzanie są operacjami o kosztach przewyższających zyski ze zrównoleglania wykonywanych obliczeń. Innym wytłumaczeniem - być może kolejnym - jest nieregularne obciążenie serwera, na którym pomiary były wykonywane