Michał Szczygieł	Aleksander Śmierciak	17/06/2014

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 zrównoleglania. Ponadto, mnożenie macierzy samo w sobie jest bardziej skomplikowane algorytmicznie niż ich dodawanie.

Jaki wpływ na zrównoleglenie będzie miało zastosowanie poniższej dyrektywy OpenMP:

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

a jaki dyrektywa:

```
#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 błedy kompilacji.

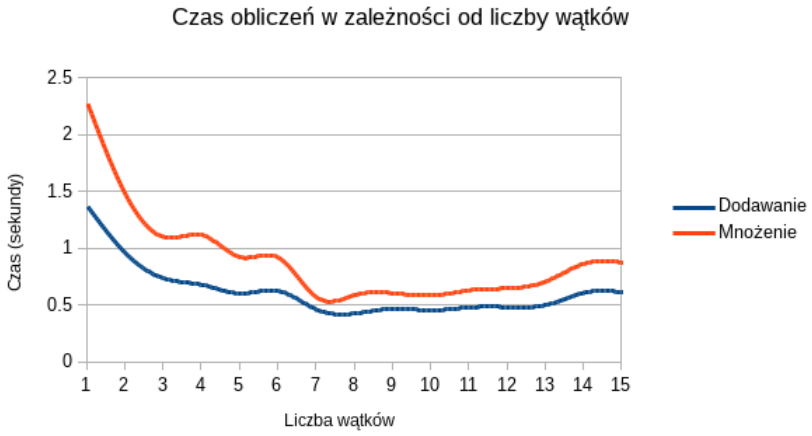
Pierwsza postać dyrektywy określa wszystkie zmienne obszaru równoległego jako współdzielone między wątkami. Druga postać dyrektywy określa zmienne A, B i C jako współdzielone, zmienną rozmiar jako prywatną o wartości wprowadzanej do obszaru równoległego i zmienne i, j jako zmienne prywatne bez wartości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

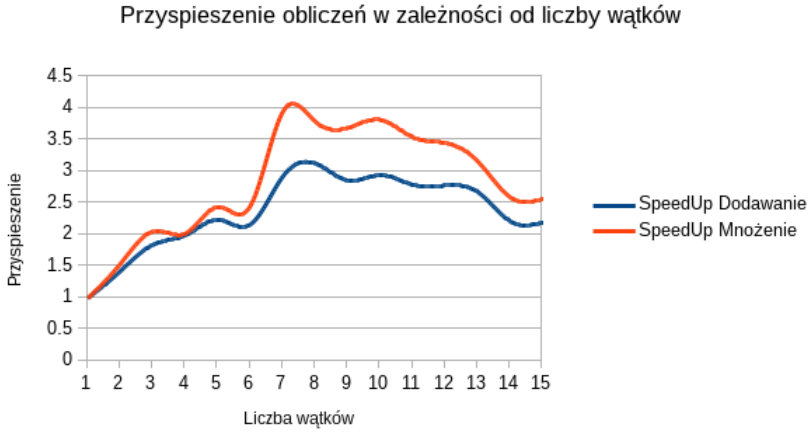
Operacje na macierzach zapisane w pseudokodzie:

```
function performMatrixOperation(...)
{
    TimePoint start = std::chrono::system_clock::now();
    C = matrixOperation(A, B, threadCount);
    TimePoint end = std::chrono::system_clock::now();

    cout << ((Duration) (end - start)).count();
}
[...]
function sumMatrices(...)
{
    fillMatrixWithZeros(*C, A.size());

    #pragma omp parallel for default(none) shared(A, B, C) num_threads(
        threadCount)
    for (unsigned int k = 0; k < 10000; ++k)
        for (unsigned int i = 0; i < A.size(); i++)
            C->at(i) = A.at(i) + B.at(i);
    return C;
```

```
19 }
20
21 function multiplyMatrices(...)
22 {
23    #pragma omp parallel for default(none) shared(A, B, C) firstprivate(
          rowSize) num_threads(threadCount)
24    for (unsigned int k = 0; k < 10000; ++k)
25       for (unsigned int i = 0; i < rowSize; ++i)
26          for (unsigned int j = 0; j < rowSize; ++j)
27             C->at(i * rowSize + j) = A.at(i * rowSize + j)
28                   * B.at(i + j * rowSize);
29    return C;
30 }
31
32 [...]
33 int main(int argc, char* argv[])
34 {
35    ...
36    performMatrixOperation(A, B, threadCount, sumMatrices);
37    performMatrixOperation(A, B, threadCount, multiplyMatrices);
38    ...
39 }
```

Poniżej wykresy przedstawiający rezultat przeprowadzonych operacji na wątkach.





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
We wcześniejszej implementacji kodu dodatkowe pętle wydłużające czas działania znajdowały się ponad sekcją równoległą. Negatywnie odbijało się to na wydajności programu - nie można było zaobserwować przyrostu wydajności wraz z rosnącą liczbą rdzeni na spodziewanym poziomie.