Grupa 1	Kamil Sacha, Konrad Szwedo	17/11/2016

Zadanie 1 - Mnożenie macierzy OpenMP

Naszym zadaniem laboratoryjnym było napisanie współbieżnego programu, który miał obliczyć iloczyn dwóch macierzy wstępnie wypełnionymi liczbami według wzorów podanych w zadaniu. Głównym celem było stworzenia rozwiązania które miało pokazywać wpływ tworzenia oprogramowania na szybkość obliczeń.

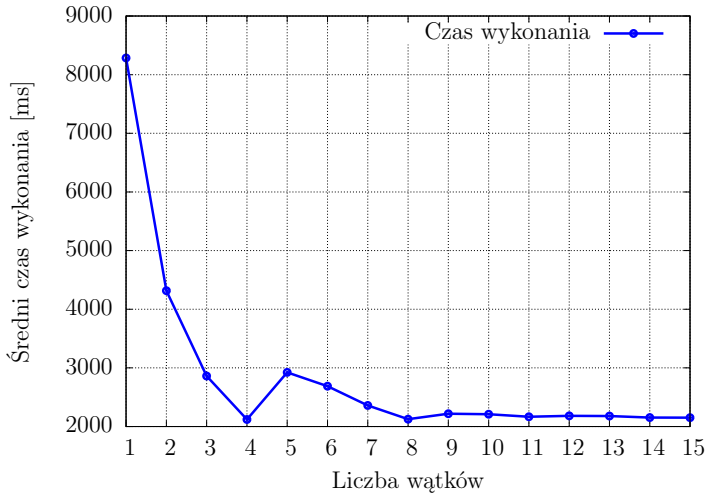
```
#pragma omp parallel for private(i, j, k) firstprivate(matrixSize)
    num_threads(threadCount)
for ( i = 0; i < matrixSize; i++) {
    for ( k = 0; k < matrixSize; k++) {
        for ( j = 0; j < matrixSize; j++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Listing kodu 1: Główna pętla programu - mnożenie macierzy

Do zrównoleglenia mnożenia macierzy została wykorzystana dyrektywa **omp parallel for**, która to dzięki wykorzystaniu dyrektywy pomocniczej **num_threads()** dzieli równomiernie operacje iteracji pomiędzy dostępnymi wątkami. Wykorzystując dyrektywę **firstprivate()** określamy, że zmienna *matrixsize* ma zostać zainicjowana przed użyciem jej przez wątek oraz ma służyć jako zmienna prywatna, oznacza to, że każdy wątek pracuje na swojej kopii tej zmiennej.

Wyniki

-10pt



Rysunek 1: Wykres zależności czasu wykonania od liczby wątków. Dla rozmiaru 10^6 danych macierzy.

Program za pomocą biblioteki openMP zrównoleglony został poprawnie, o czym świadczy wykres 1.
Niewielkie załamanie liniowości wykresu związane jest najprawdopodobniej z wykorzystywaniem technologi *HyperThreading*. Inną przyczyną może być też testowanie na serwerze który nie pracuje tylko i wyłącznie dla nas (wykonuje on też swoje zadania co wpływa na wynik, nie mamy 100% dostępu do czasu procesora.

Jaki wpływ na zrównoleglenie będzie miało zastosowanie poniższej dyrektywy OpenMP

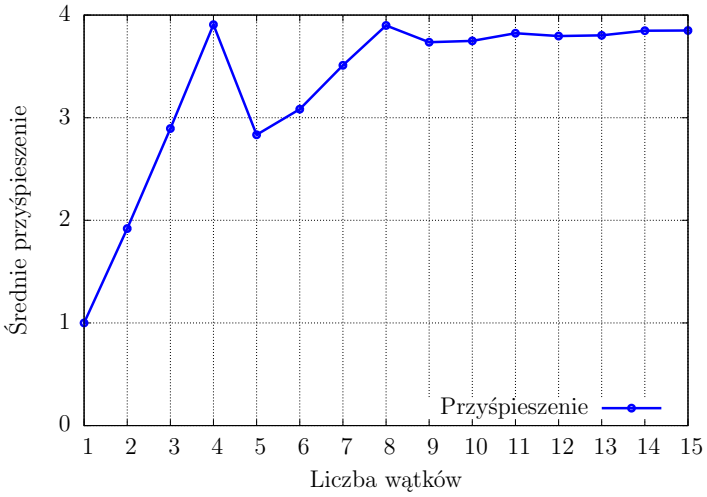
Dla pierwszej dyrektywy uzyskujemy wolniejsze zrównoleglenie ponieważ wszystkie zmienne są współdzielone.

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

a jaki dyrektywa

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

Największą wydajność uzyskujemy dla 4 wątków, naszym zdaniem związane jest to z tym, że w takim układzie każdy rdzeń procesora dostaje swój wątek i wykorzystuje go maksymalnie.



Rysunek 2: Wykres zależności przyśpieszenia od liczby wątków. Dla rozmiaru 10^6 danych macierzy.

Procedura testowania

Do przetestowania planowanego wzrostu wydajności został napisany skrypt pracujący pod powłoką BASH. Dla zadanego rozmiaru macierzy w naszym przypadku 10^6 oblicza średnią z 10 uruchomień naszego programu macierzomp (jako rozmiar macierzy przyjmujemy całkowitą ilość elementów jednej macierzy, wynika z tego, że macierz jest jest rozmiaru n x n gdzie n $=\sqrt{matrixSize}$, n jest zawsze liczbą całkowitą). W Ostatnim etapie generowany jest wykres w programie gnuplot. Program kompiluje się poprawnie poprzez kompilator *g++* z flagą *-O3* w celu optymalizacji kodu i tym samym przyśpieszenia wykonania programu.

Wnioski

Jak możemy zauważyć skalowalność czasowa jest liniowa do wartości 4 wątków. Wiążę się to z kilkoma czynnikami, jednym z czynników jest to, że dla większej ilości wątków powstaje narzut współbieżności związany z tworzeniem, synchronizacją, i zarządzaniem poszczególnymi wątkami.

Innym naszym zdaniem istotnym czynnikiem jest to, że program testowany był na serwerze pod adresem: cuda.iti.pk.edu.pl gdzie pracuje Intel® Core™ I7-950. Procesor ten wyposażony jest w 4 rdzenie fizyczne, które wraz z technologią Hyper-Threading tworzą 8 procesorów wirtualnych (widzianych dla systemu jako 8 procesorów fizycznych). Według informacji z portali zajmujących się testowaniem sprzętu, całkowity zysk ze sto-

sowania tej technologi jest podawany jako maksymalnie 10 - 30%. Dlatego też nie uzyskujemy skalowania liniowego rozpatrując czas wykonania dla powyżej 4 wątków.