

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.

```

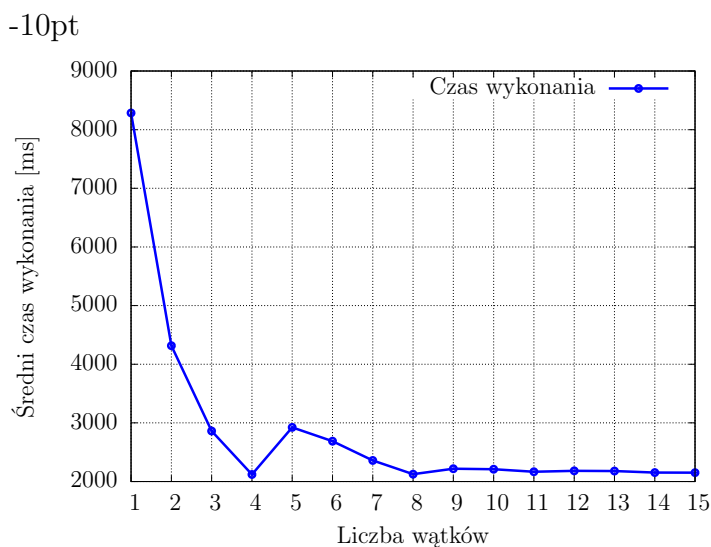
1 #pragma omp parallel for private(i, j, k) firstprivate(matrixSize)
   num_threads(threadCount)
2 for ( i = 0; i < matrixSize; i++) {
3     for ( k = 0; k < matrixSize; k++) {
4         for ( j = 0; j < matrixSize; j++) {
5             C[i][j] += A[i][k] * B[k][j];
6         }
7     }
8 }

```

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



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 technologii *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)
```

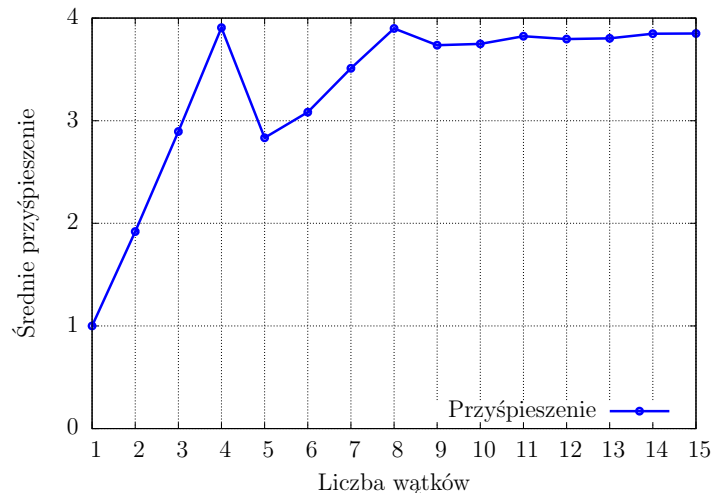
a jaka dyrektywa

```
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.

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 rozmiaru $n \times n$ gdzie $n = \sqrt{\text{matrixSize}}$, 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 przyspieszenia wykonania programu.



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

Wnioski

Jak możemy zauważyć skalowalność czasowa jest liniowa do wartości 4 wątków. Wiąże 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 technologii jest podawany jako maksymalnie 10 - 30%. Dlatego też nie używamy skalowania liniowego rozpatrując czas wykonania dla powyżej 4 wątków.