Grupa 1	Kamil Sacha, Konrad Szwedo	03/11/2016

Zadanie 2 - Mnożenie macierzy OpenMPI

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
2  if (totalProcesses != 1) {
3
4  starttime = MPI_Wtime();
5
6
7  for(int destination = 1; destination < totalProcesses; destination++){
8
9     elemsForProc = getElemsCountForProcess(matrixSize, totalProcesses,
          destination);
10    rowsOffset += elemsForProc / matrixSize;
11
12    MPI_Send(&elemsForProc,                    1,    MPI_INT, destination,
          MSG_FROM_MASTER, MPI_COMM_WORLD);
13    MPI_Send(&rowsOffset,                      1,    MPI_INT, destination,
          MSG_FROM_MASTER, MPI_COMM_WORLD);
14    MPI_Send(&A[rowsOffset][0],    elemsForProc, MPI_FLOAT, destination,
          MSG_FROM_MASTER, MPI_COMM_WORLD);
15    MPI_Send(&B[0][0], matrixSize * matrixSize, MPI_FLOAT, destination,
          MSG_FROM_MASTER, MPI_COMM_WORLD);
16 }
17
18
19 multiplyPartOfMatrix(A, B, C, matrixSize, 0, getElemsCountForProcess(
       matrixSize, totalProcesses, MASTER));
20
21
22 for (int source = 1; source < totalProcesses; source++) {
23    MPI_Recv(&elemsForProc,                   1, MPI_INT,    source,
          MSG_FROM_WORKER, MPI_COMM_WORLD, &status);
24    MPI_Recv(&rowsOffset,                     1, MPI_INT,    source,
          MSG_FROM_WORKER, MPI_COMM_WORLD, &status);
25    MPI_Recv(&C[rowsOffset][0], elemsForProc , MPI_DOUBLE, source,
          MSG_FROM_WORKER, MPI_COMM_WORLD, &status);
26 }
27
28 endtime    = MPI_Wtime();
29
30 printf("Czas %.0f ms\n", (endtime - starttime) * 1000);
31 }
```

Listing kodu 1: Część programu odpowiedzialna za wysyłanie macierzy A oraz B i odbiór wyników z reszty procesów

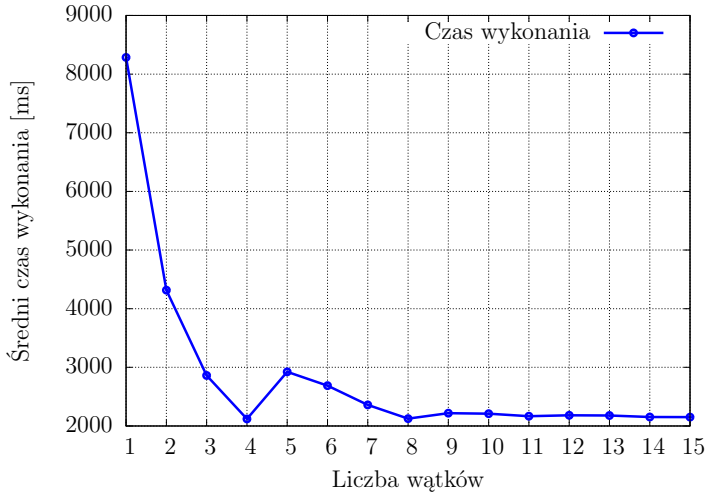
Zrównoleglenie problemu mnożenia macierzy zostało wykonane w środowisku OpenMPI, jest to środowisko do wymiany wiadomości/komunikatów między procesami.

Powyższy fragment kodu przedstawia część odpowiedzialną za odbieranie, liczenie, oraz wysyłanie danych do procesu mastera - zarządcy. Algorytm działa w następujący sposób: Gdy program uruchomiony jest jako posiadający maksimum 1 proces, najpierw inicjalizowane są macierze A, B, C, następnie generowane liczby wypełniające je a na końcu cały blok instrukcji mnożenia macierzy jest wykonywany przez pojedynczy proces. Gdy liczba procesów jest większa niż dwa, proces główny master rozsyła pozostałym całą macierz B, część macierzy A, oraz liczbę elementów które ma policzyć i liczbę wierszy (macierzy A). Procesy są blokowane funkcją czekającą na dane **MPI_Recv(...)**.

Gdy otrzymają dane wykonywana jest funkcja: **multiplyPartOfMatrix(A, B, C, matrixSize, rowsOffset, elemsForProc)** która to oblicza swoją część macierzy i zapisuje ją do tablicy **C**. Następnie procesy inne niż master, wysyłają z powrotem dane do procesu mastera który to czeka na złączenie wszystkich wyników w swoim egzemplarzu tablicy **C**. Proces master po wysłaniu danych, przystępuje do liczenia, dzięki temu podejściu każdy proces ma swój udział w obliczeniach i nie dochodzi do sytuacji, że jeden proces byłby wykorzystywany tylko na generowanie, wysyłanie i dalsze oczekiwanie na wyniki.

Wyniki

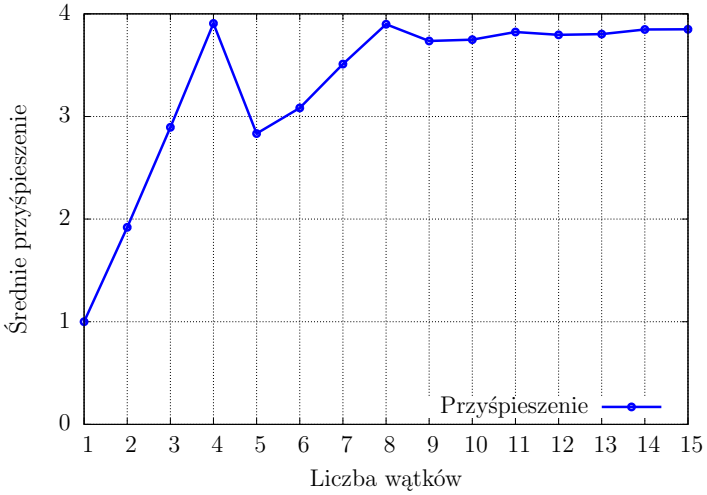
-10pt



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

Program za pomocą biblioteki openMPI 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.

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



Rysunek 2: Wykres zależności przyśpieszenia od liczby procesów. 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 MPI *gcxx* z flagą *-O3* w celu optymalizacji kodu i tym samym przyśpieszenia wykonania programu.

Wnioski

Jak możemy zauważyć skalowalność czasowa jest liniowa do wartości 4 procesów. Wiążę się to z kilkoma czynnikami, jednym z czynników jest to, że dla większej ilości procesów powstaje narzut współbieżności związany z tworzeniem, synchronizacją, i zarządzaniem poszczególnymi procesami. Drugim ważniejszym czynnikiem jest narzut związany z wymianą danych, przykładowo dla macierzy rozmiarów (100x100) zawierającą 1E4 elementów, gdzie każdy element jest liczbą zmiennoprzecinkową typu float, na samo wysłanie wyników do procesu mastera musimy wymienić 4E8 bajtów czyli około 400 megabajtów pamięci.