Grupa 6	Dariusz Szczupak, Kamil Wanat	07/11/2018

Zadanie 4 - Liczby Pierwsze MPI

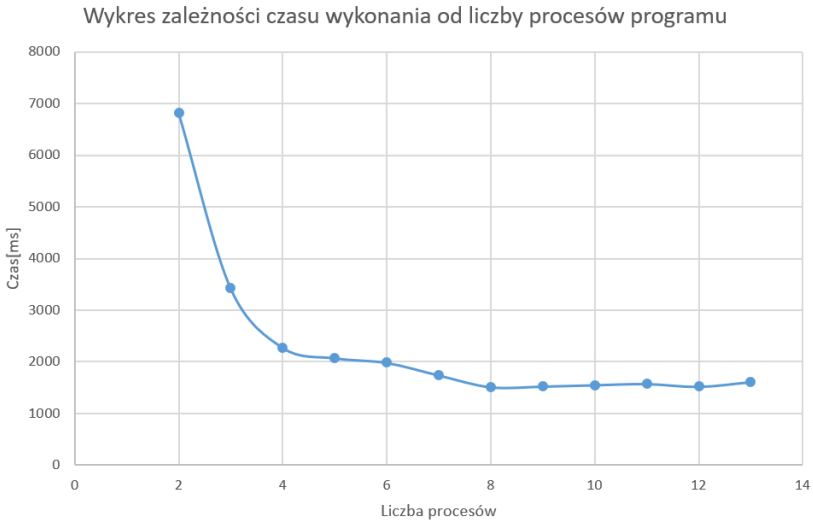
Zadanie laboratoryjne polegało na zaimplementowaniu programu sprawdzającego czy podane liczby są liczbami pierwszymi, czy złożonymi. Program w swoim działaniu wykorzystuje MPI(Message Passing Interface) - standard przesyłania komunikatów pomiędzy procesami, w celu jego zrównoleglenia. Problem został rozwiązany przy pomocy algorytmu "naiwnego". Jest to metoda nieco wolniesza niż metody probabilistyczne, jednak dająca zdecydowanie lepsze wyniki. Polega ona na dzieleniu testowanego elementu przez kolejne liczby od 2 do pierwiastka kwadratowego z tejże. W naszym programie podejście naiwne zostało nieco zmodyfikowane. Zamiast sprawdzać dzielniki od 2 do pierwiastka z każdej liczby, znajdujemy najpierw liczbę maksymalną, a następnie testujemy wszystkie liczby przy pomocy dzielnikow od 2 do sqrt(max). Ponadto nie testujemy każdej liczby osobno dla danego dzielnika, lecz cały zbiór.

```
for (int i = 1; i < size; i++) {
            if (i == (size - 1))
                end = tab.size();
            vector <number> templateTable(tab.begin() + begin, tab.begin()
               + end);
            begin = end;
            end += step;

            uint tableSize = templateTable.size();
            MPI_Send(&tableSize, 1, MPI_INT, i, 0, comm);
            MPI_Send(&sqr, 1, MPI_INT, i, 1, comm);
            MPI_Send(templateTable.data(), tableSize, myType, i, 2, comm);
        }
        vector <number> tab2;
        vector <number> concatTable;
for (int i = 1; i < size; i++) {

            MPI_Recv(&d, 1, MPI_INT, i, 0, comm, MPI_STATUS_IGNORE);
            tab2.resize(d);
            MPI_Recv(tab2.data(), d, myType, i, 1, comm, MPI_STATUS_IGNORE)
                ;
            concatTable.insert(concatTable.end(), tab2.begin(), tab2.end())
                ;
        }
```

Powyższy kod źródłowy przedstawia działanie zasadniczej części procesu zerowego (głównego). Jego zadaniem jest podział całego wektora liczb pierwszych na mniejsze podzbiory, a następnie ich rozesłanie do procesów o numerze rank wiekszym niż 0. Odpowedzialna jest za to pierwsza pętla for. Zakres liczb które wchodzą w skład danego podzbioru ustalają zmienne begin oraz end a ich ilość zmienna step. Wycięcie zbioru danych wykonywane jest w konstruktorze wektora templateTable. Do procesów potomnych wysyłane sa trzy wiadomości: pierwsza z rozmiarem wektora danych, druga z wartością pierwiastka do którego testujemy liczby oraz trzecia z danymi zawartymi w wektorze. Kolejną częścią programu jest odebranie oraz sklejenie danych z wektorem wynikowym. Program główny oczekuje na odebranie od procesów potomnych dwóch wiadomości: pierwsza z rozmiarem danych oraz druga z dynymi właściwymi zapisanymi do wektora tab2. Następnie dane te są doklejane za pomocą funkcji insert do wektora wynikowego - concatTable. Aby możliwe

było wysyłanie wektora który zawiera w sobie typ złożony, musieliśmy zarejestrować taki typ w programie. Sposób rejestracji nowego typu MPI przedstawiony jest na poniższym listingu.

```
1 int lengths[2] = {1, 1};
    // ilosc zmiennych w strukturze
2   MPI_Aint offsets[2] = {offsetof(number, value), offsetof(number, prime)
      }; // przesuniecie bitowe zmiennych struktury
3   MPI_Datatype types[2] = {MPI_LONG,MPI::BOOL};
                                // podstawowe typy skladajace sie na nasz
      typ
4   MPI_Datatype myType;
                                                    // deklaracja
      nowego typu
5   MPI_Type_create_struct(2, lengths, offsets, types, &myType);
                // utworzenie nowego typu
6   MPI_Type_commit(&myType);
```

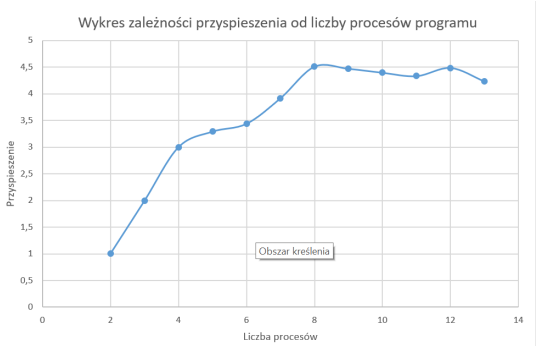


Rysunek 1: Wykres zależności czasu wykonywania od liczby procesów

Powyższy wykres przedstawia zależność czasu wykonywania się programu od liczby procesów z jaką został on uruchomiony. Wykres rozpoczyna sie od 2 procesów, ponieważ proces zerowy wykorzystywany jest jedynie do podziału zbioru na mniejsze części, wysłanie danych do procesów oraz ich odebranie i sklejenie. Wobec tego nie jest możliwe uruchomienie programu z parametrem n=1. Jak dobrze widać na wykresie czas nie zmniejsza się liniowo wraz ze wzrostem liczby wątków. Spowodowane jest to wzrostem czasu obliczeń dla procesu zerowego ze względu na większą ilość operacji komunikacji pomiędzy procesem zerowym a procesami potomnymi. Każda z takich operacji zajmuje pewną ilość czasu, tak więc im więcej procesów uruchomimy, tym większy stanie się czas wykonywania operacji wysyłania/odbierania danych. Podobnie jak w przypadku OpenMP najlepszy wynik uzyskujemy dla 8 procesów (tyle ile wątków dostarcza serwer CUDA). Późniejsze zwiększanie ilości procesów zwiększa nam nieznacznie czas wykonania obliczeń. Również tak jak w poprzednich zadaniach widoczny jest pewien moment w którym czas obliczeń jest na stałym poziomie, a następnie znów się zmniejsza. Taka sytuacja występuje dla 6 procesów. Powodowane jest to prawdopodobnie zwiększeniem narzutu na procesor

przez większą liczbę komunikatów oraz włączeniem oraz architekturą sewera CUDA (8 wątków przy 4 fizycznych rdzeniach z wykorzystaniem technologii HyperThreading).

Wykres przyspieszenia programu testującego liczby pierwsze pokazuje czy i jak udało się zrównoleglić działanie programu. Z tego wykresu możemy wyciągnąć analogiczne wnioski do tych, które zostały opisane pod wykresem zależności czasu wykonania od ilości wątków procesora. Tutaj również widać spadek przyspieszenia od liczby procesów - 5 wzwyż oraz wzrost czasu obliczeń dla liczby procesów powyżej 8.



Rysunek 2: Wykres przyspieszenia

Dane przeprowadzanych testów:

- Wzięto pod uwagę średnią pomiarów czasów wykonania programu dla liczby wątków z przedziału od 2 do 13. Program był uruchamiany dla pliku pobranego z internetu.
- Do mierzenia czasu wykorzystano MPIWtime.
- Testy zostały wykonane na serwerze cuda.iti.pk.edu.pl .
- W czasie testów na serwerze zalogowany był jedynie użytkownik testujący.