Grupa lab.
5

Data ćwiczenia
08.01.2024

Labratorium 12

Imie i nazwisko
Łukasz Sawina

Cel

Celem ćwiczenia było doskonalenie podstaw programowania równoległego z wykorzystaniem MPI oraz poznanie komunikacji grupowej w tej technologii.

Obliczanie PI

W tym zadaniu musimy obliczyć liczbę PI korzystając z szeregu Leibniza:

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \frac{\pi}{4}$$

Na początku otrzymujemy przykładowy program obliczający PI szeregowo. Naszym zadaniem jest przekształcić ten program, tak aby obliczanie było zrównoleglone, dodatkowo chcemy aby każdy proces brał udział w obliczeniach, dlatego dokonamy dekompozycji blokowej. Proces o randze 0 będzie miał jeszcze dwie dodatkowe funkcjonalności, na początku programu przyjmuje od użytkownika ilość wyrazów z jakiej ma policzyć nasz szereg oraz na końcu otrzymać pośrednie wyniki każdego z procesów i podać ostateczny wynik.

Na początku programu tworzymy standardową strukturę inicjacji komunikacji w MPI:

```
int rank, ranksent, size;

MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );

int max_liczba_wyrazow=0;
int root = 0;
```

Tworzymy zmienne pomocnicze takie jak size – ilość procesów w komunikacji, rank – numer danego procesu. Na stępnie inicjalizujemy naszą komunikację, pobieramy numer rank procesu oraz rozmiar komunikacji.

Na przechowywanie ilości wyrazów naszego szeregu utworzona i zainicjalizowana została zmienna *max_liczba_wyrazow*, dodatkowo do przechowywania informacji, który proces jest naszym głównym (pobierającym dane oraz wyświetlającym na końcu) istnieje zmienna *root*, która ma wartość 0.

Następnie chcemy, aby główny proces wyświetlił użytkownikowi komunikat oraz pobrał ilość wyrazów szeregu.

```
if( rank == root )
{
  printf("Podaj maksymalną liczbę wyrazów do obliczenia przybliżenia PI\n");
  scanf("%d", &max_liczba_wyrazow);
}

MPI_Bcast( &max_liczba_wyrazow, 1, MPI_INT, root, MPI_COMM_WORLD );
```

Tworzymy sobie warunek sprawdzający rangę procesu, następnie proces 0 wyświetla i pobiera dane.

Przy pomocy *MPI_Bcast()* przesyłamy z procesu głównego do wszystkich pobraną wartość, w taki sposób przy pomocy jednej funkcji każdy proces otrzyma te same dane i nie musimy tworzyć skomplikowanej komunikacji między głównym procesem a pozostałymi.

Funkcja MPI_Bcast() przyjmuje jako parametry kolejno:

- Wskaźnik na przesyłane dane, bufor danych
- Ilość elementów w buforze
- Typ danych, oczywiście zdefiniowane jako typy MPI
- Numer procesu, który rozgłasza dane
- Komunikację, w ramach, której odbywa się rozgłoszenie

Teraz, gdy wszystkie procesy wiedzą jaki jest rozmiar szeregu mogą wykonać obliczenia, ponieważ wykonujemy dekompozycję blokową obliczamy rozmiar bloku dla każdego procesu i konkretne indeksy startu oraz końca.

```
int n_loc = ceil(max_liczba_wyrazow/size);

int my_start = rank*n_loc;
int my_end = (rank + 1) * n_loc;
if (rank == size - 1) my_end = max_liczba_wyrazow;

double suma_plus=0.0;
double suma_minus=0.0;
int i=0;
for(i = my_start; i<my_end; i++){
  int j = 1 + 4*i;
  suma_plus += 1.0/j;
  suma_minus += 1.0/(j+2.0);
}

double pi_approx_lok = 4*(suma_plus-suma_minus);
double pi_approx = 0.0;
```

Każdy proces oblicza standardowo swój początek oraz koniec bloku, dodatkowo dla ostatniego procesu końcem jest rozmiar podany przez użytkownika. Następnie w pętli obliczane są sumy i po przekształceniu wzoru obliczana jest wartość fragmentu, a wynik zapisywany jest w zmiennej *pi_approx_lok*. Zmienna pi_approx wykorzystana jest do przechowywania wyniku końcowego naszego programu.

Teraz każdy z procesów musi przesłać swój lokalny wynik do procesu 0, a następnie wartości te muszą zostać zsumowane i wyświetlone. Wykonać to można przy pomocy MPI_Reduce(), funkcja ta ułatwia nam przesłanie oraz zsumowanie wszystkich wyników do jednej zmiennej i dzięki temu nie musimy tego pisać ręcznie.

Następnie nasz proces 0 wyświetla wynik obliczony przez program oraz dla porównania wartość z biblioteki matematycznej.

```
  MPI_Reduce( &pi_approx_lok, &pi_approx, 1, MPI_DOUBLE, MPI_SUM, root,
MPI_COMM_WORLD);

  if( rank == root )
  {
    printf("PI obliczone: \t\t\t%20.15lf\n", pi_approx);
    printf("PI z biblioteki matematycznej: \t%20.15lf\n", M_PI);
  }
```

Funkcja *MPI_Reduce* daje nie tylko opcję sumowania, przy jej pomocy możemy również przemnożyć wartości, pobrać wartość najmniejszą/największą ze wszystkich przekazanych, wykonać operacje bitowe oraz logiczne.

Funkcja *MPI_Reduce* przyjmuje jako parametry kolejno:

- Bufor danych do wysłania,
- Bufor gdzie zapisać dane,
- Liczba elementów przesyłanych,
- Typ danych
- Operacja na danych
- Numer procesu, który ma przyjąć wynik,
- Komunikacja

Wynikiem naszego programu dla przykładowo 1000 wyrazów jest:

```
Podaj maksymalną liczbę wyrazów do obliczenia przybliżenia PI
1000
PI obliczone:                   3.141092653621041
PI z biblioteki matematycznej:  3.141592653589793
```

Do uruchomienia programu można wykorzystać jeden z plików makefile z poprzednich zajęć. Wynik oczywiście nie jest zbyt dokładny, zwiększając ilość wyrazów zwiększymy dokładność wyniku, jednak pokazuje nam, że udało nam się zrównoleglić nasz program oraz wykorzystując mechanizm komunikacji grupowej zmiejszyć objętość kodu, przez wykorzystanie *MPI_Bcast()* oraz *MPI_Reduce()*.

Mnożenie macierz-wektor

W kolejnym zadaniu otrzymaliśmy program obliczający mnożenie macierzy przez wektor. Program posiada w sobie trzy warianty, jeden sekwencyjny oraz dwa równoległe, a naszym zadaniem było usprawnienie programu, przez wykorzystanie komunikacji grupowej.

W pierwszej kolejności musimy zrównoleglić przekazywanie wyników ze wszystkich procesów do jednego, który będzie analizował wyniki i porównywał je z rzeczywistymi. W gotowym programie rozwiązanie to jest wykonane przy pomocy wywołania w pętli *MPI_Recv()* oraz przesyłania danych przy pomocy *MPI_Send()*.

```
    if(rank>0){
      MPI_Send( z, n_wier, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD );
    } else {
      for(i=1;i<size;i++){
                MPI_Recv( &z[i*n_wier], n_wier, MPI_DOUBLE, i, tag, MPI_COMM_WORLD,
 &status  );
       }
    }
```

Jak widać taka komunikacja jest mało optymalna, pojawia się przede wszystkim problem z wąskim gardłem, którym jest proces 0, dodatkowo przez wykorzystanie MPI_Send oraz MPI_Recv pojawiają się blokady, ponieważ są to funkcje blokujące, MPI_Recv nie pójdzie dalej, dopóki nie otrzyma komunikatu.

Ten kod można zmienić na bardziej optymalny wykorzystując przykładowo funkcję MPI_Gather(), która zbierze dane ze wszystkich procesów i zapisze je w buforze odbiorcy.

```
    MPI_Gather(z, n_wier, MPI_DOUBLE, z, n_wier, MPI_DOUBLE, 0,
 MPI_COMM_WORLD);
```

Funkcja *MPI_Gather* przyjmuje jako parametry kolejno:

- Bufor danych do wysłania,
- Ilość wysyłanych danych,
- Typ wysyłanych danych,
- Bufor na odebrane dane,
- Ilość elementów w buforze na odebrane dane,
- Typ odebranych danych,
- Numer procesu, który odbiera dane,
- Komunikacja

Dzięki wykorzystaniu tej funkcji zmniejszamy ilość kodu z kilku linijek kodu do jednej.

Tą samą operację możemy wykonać podobnie, ale optymalniej z wykorzystaniem MPI_IN_PLACE. Bez tej stałej dane są kopiowane do buforów, a z wykorzystaniem jej pozwalamy , aby dane nie były kopiowane i abyśmy mogli operować na oryginalnych danych.

Kod z wykorzystaniem MPI_IN_PLACE wygląda następująco:

```
    if (rank > 0) {
        MPI_Gather(z, n_wier, MPI_DOUBLE, z, n_wier, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
    } else {
        MPI_Gather(MPI_IN_PLACE, n_wier, MPI_DOUBLE, z, n_wier, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
    }
```

Używając MPI_IN_PLACE dla bufora wysyłającego, procesy o numerach większych od 0 przekazują swoje dane bezpośrednio do bufora odbiorczego na procesie 0, bez dodatkowej alokacji pamięci na procesie 0.

Kolejną rzeczą do zrównoleglenia było uzyskiwanie wyniku poprzez wykorzystanie *MPI_Reduce* oraz *MPI_Alltoall*.

```
    MPI_Reduce(z, y, WYMIAR, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    MPI_Alltoall(MPI_IN_PLACE, n_col, MPI_DOUBLE, y, n_col, MPI_DOUBLE,
MPI_COMM_WORLD);
```

Chcemy osiągnąć, aby wszystkie wątki przesłały swoje dane do procesu 0, dane zostały zsumowane oraz aby wszyskie dane zostały rozesłane pomiędzy wszystkimi procesami. Funkcja *MPI_Reduce()*, została już opisana wcześniej, dlatego pomijam jej opis tutaj.

Funkcja *MPI_Alltoall()*, odpowiedzialna jest za wymianę danych między wszystkimi procesami. Każdy proces wysyła blok danych do każdego innego procesu i jednocześnie otrzymuje blok od każdego innego procesu.

Funkcja *MPI_Alltoall* przyjmuje jako parametry kolejno:

- Bufor danych do wysłania,
- Ilość danych do wysłania,
- Typ danych do wysłania,
- Bufor na odebrane dane,
- Ilość odebranych danych,
- Typ odebranych danych,
- Komunikacja

Dzięki wykorzystaniu jednej funkcji pomijamy tworzenie zbędnego kodu, ponieważ przykładowy kod realizujący to samo, ale z wykorzystaniem *MPI_Recv* oraz *MPI_Send* wyglądałby następująco:

```
    for (int i = 0; i < size; i++) {
        if (i != rank) {
            MPI_Send(&send_data, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
        }
    }

    for (int i = 0; i < size; i++) {
        if (i != rank) {
            MPI_Recv(&recv_data[i], 1, MPI_INT, i, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        }
    }
```

Jak widać ilość kodu jest znacznie mniejsza oraz optymalność tego kodu będzie większa.

Po tych wszystkich zrównolegleniach po wykonaniu programu otrzymujemy:

```
poczatek (wykonanie sekwencyjne)
        czas wykonania (zaburzony przez MPI?): 0.000001, Gflop/s: 0.839344, GB/s
> 3.567213
Starting MPI matrix-vector product with block row decomposition!
Wersja rownolegla MPI z dekompozycją wierszową blokową
        czas wykonania: 0.000036, Gflop/s: 0.014028, GB/s> 0.059620
Starting MPI matrix-vector product with block column decomposition!
Werja rownolegla MPI z dekompozycją kolumnową blokową
        czas wykonania: 0.000021, Gflop/s: 0.024012, GB/s> 0.102049
```

Wnioski

Mechanizm komunikacja w MPI jest rzeczą bardzo pomocną, jednak w sytuacji gdy chcemy komunikować się z większą ilością procesów jednocześnie, wykorzystanie podstawowych *MPI_Send* oraz *MPI_Recv* traci sens, ponieważ nie są to optymalne, ani czytelne rozwiązania.

MPI udostępnia kilka mechanizmów do komunikacji grupowej, które bez problemu rozwiążą większość problemów, między innymi: *MPI_Bcast, MPI_Scatter, MPI_Gather, MPI_Alltoall* i tym podobne. Wszystkie te mechanizmy są zaimplementowa przez MPI, więc będą gwarantowały większą wydajność w komunikacji między procesami. Przy pomocy przykładowo funkcji *MPI_Bcast* oraz *MPI_Gather* zapewniają większą spójność danych, ponieważ mogą być one jednocześnie wysłane do wszystkich procesów lub pobierane ze wszystkich procesów jednocześnie.

Operacje grupowe są szczególnie przydatne w sytuacji, gdzie zadanie obejmuje równoczesne przetwarzanie danych w wielu procesach, a wyniki muszą być wymienione między procesami.

Wiele funkcji ma swoje odpowiedniki, które są poprzedzone prefiksem „All”, tak np. zamiast *MPI_Reduce()* możliwe jest wykorzystanie *MPI_Allreduce()*, który zadziała identycznie, z tym, że nie przyjmuje ona numeru procesu, do którego ma wysłać wynik, a wynik wysyła do wszystkich procesów w komunikacji. Podobnie funkcja MPI_Allgather(), dane są zbierane ze wszystkich procesów, a następnie wysyłane do wszystkich.