|  |  |  |
| --- | --- | --- |
| **Imię i Nazwisko**  Patryk Twardosz | **Kierunek**  Informatyka Techniczna | **Rok studiów i grupa**  I rok, Gr. 9 |
| **Data**  18.01.2025r. | **Numer i temat sprawozdania**  Lab 11, 12 i 13 – MPI i pomiary wydajności | |

**Lab11:**

**Cel:**

* Opanowanie podstaw programowania z przesyłaniem komunikatów MPI.

**Zadanie:**

1. Przygotowanie projektu
2. Uzupełnienie programu o przesyłanie adresu internetowego węzła nadawcy.

if(size>1){

    if( rank != 0 ){ dest=0; tag=0;

      MPI\_Send( hostname, HOSTNAME\_MAX, MPI\_CHAR, dest, tag,

MPI\_COMM\_WORLD );

    } else {

      char recvhostname[HOSTNAME\_MAX];

      for( i=1; i<size; i++ ) {

        MPI\_Recv( recvhostname, HOSTNAME\_MAX, MPI\_CHAR,

MPI\_ANY\_SOURCE, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, &status );

        printf("Proces %d wysłał hostname: %s\n",

status.MPI\_SOURCE, recvhostname);

      }

    }

  }

  else {

    printf("Pojedynczy proces o randze: %d (brak komunikatów)\n", rank);

  }

Wszystkie procesy potomne otrzymują nazwę hosta nadrzędnego

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

1. Opracowanie programu propagującego komunikaty w konwencji pierścienia

if (rank == 0) {

    value = 1;

    printf("Proces %d inicjuje sztafetę z liczbą %d\n", rank, value);

    MPI\_Send(&value, 1, MPI\_INT, next, 0, MPI\_COMM\_WORLD);

  }

  else {

    MPI\_Recv(&value, 1, MPI\_INT, prev, 0, MPI\_COMM\_WORLD, &status);

    printf("Proces %d odebrał liczbę %d od procesu %d\n", rank, value, status.MPI\_SOURCE);

    value++;

    if (rank != size - 1)

      MPI\_Send(&value, 1, MPI\_INT, next, 0, MPI\_COMM\_WORLD);

  }

Obraz zawierający tekst, zrzut ekranu, Czcionka, numer

Opis wygenerowany automatycznie

1. Przesyłanie struktur między procesami

typedef struct {

    char name[50];

    float pi;

    int num;

} Data;

// Pakowanie danych

 Data data = {"Patryk", 21.37, 8080};

        void\* buffer = malloc(BUFFER\_SIZE);

        int position = 0;

        MPI\_Pack(data.name, 50, MPI\_CHAR, buffer, BUFFER\_SIZE, &position, MPI\_COMM\_WORLD);

        MPI\_Pack(&data.pi, 1, MPI\_FLOAT, buffer, BUFFER\_SIZE, &position, MPI\_COMM\_WORLD);

        MPI\_Pack(&data.num, 1, MPI\_INT, buffer, BUFFER\_SIZE, &position, MPI\_COMM\_WORLD);

        MPI\_Send(buffer, position, MPI\_PACKED, next, tag, MPI\_COMM\_WORLD);

        free(buffer);

// Rozpakowanie danych

 void \*buffer = malloc(BUFFER\_SIZE);

        Data recData;

        int position = 0;

        MPI\_Recv(buffer, BUFFER\_SIZE, MPI\_PACKED, prev, tag, MPI\_COMM\_WORLD, &status);

        MPI\_Unpack(buffer, BUFFER\_SIZE, &position, recData.name, 50, MPI\_CHAR, MPI\_COMM\_WORLD);

        MPI\_Unpack(buffer, BUFFER\_SIZE, &position, &recData.pi, 1, MPI\_FLOAT, MPI\_COMM\_WORLD);

        MPI\_Unpack(buffer, BUFFER\_SIZE, &position, &recData.num, 1, MPI\_INT, MPI\_COMM\_WORLD);

        printf("Proces %d otrzymał dane: %s, %d, %.2f\n", rank, recData.name, recData.num, recData.pi);

        free(buffer);

1. Wykonanie procesowania potokowaego struktury

    void \*buffer = malloc(BUFFER\_SIZE);

        int position = 0;

        MPI\_Pack(dataset[i].text, 50, MPI\_CHAR, buffer, BUFFER\_SIZE, &position, MPI\_COMM\_WORLD);

        MPI\_Pack(&dataset[i].a\_count, 1, MPI\_INT, buffer, BUFFER\_SIZE, &position, MPI\_COMM\_WORLD);

        MPI\_Pack(&dataset[i].length, 1, MPI\_INT, buffer, BUFFER\_SIZE, &position, MPI\_COMM\_WORLD);

        MPI\_Pack(&dataset[i].is\_done, 1, MPI\_INT, buffer, BUFFER\_SIZE, &position, MPI\_COMM\_WORLD);

        MPI\_Send(buffer, position, MPI\_PACKED, next, tag, MPI\_COMM\_WORLD);

        free(buffer);

        buffer = malloc(BUFFER\_SIZE);

        position = 0;

        MPI\_Recv(buffer, BUFFER\_SIZE, MPI\_PACKED, prev, tag, MPI\_COMM\_WORLD, &status);

        MPI\_Unpack(buffer, BUFFER\_SIZE, &position, dataset[i].text, 50, MPI\_CHAR, MPI\_COMM\_WORLD);

        MPI\_Unpack(buffer, BUFFER\_SIZE, &position, &dataset[i].a\_count, 1, MPI\_INT, MPI\_COMM\_WORLD);

        MPI\_Unpack(buffer, BUFFER\_SIZE, &position, &dataset[i].length, 1, MPI\_INT, MPI\_COMM\_WORLD);

        MPI\_Unpack(buffer, BUFFER\_SIZE, &position, &dataset[i].is\_done, 1, MPI\_INT, MPI\_COMM\_WORLD);

        free(buffer);

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

if(rank == size - 1) {

          break;

        } else {

          void \*buffer = malloc(BUFFER\_SIZE);

          int position = 0;

          MPI\_Pack(data.text, 50, MPI\_CHAR, buffer, BUFFER\_SIZE, &position, MPI\_COMM\_WORLD);

          MPI\_Pack(&data.a\_count, 1, MPI\_INT, buffer, BUFFER\_SIZE, &position, MPI\_COMM\_WORLD);

          MPI\_Pack(&data.length, 1, MPI\_INT, buffer, BUFFER\_SIZE, &position, MPI\_COMM\_WORLD);

          MPI\_Pack(&data.is\_done, 1, MPI\_INT, buffer, BUFFER\_SIZE, &position, MPI\_COMM\_WORLD);

          MPI\_Send(buffer, position, MPI\_PACKED, next, tag, MPI\_COMM\_WORLD);

          break;

        }

      }

      if (rank == 1) {

        count\_a\_in\_text(&data);

      } else if (rank == 2) {

        add\_a(&data);

      } else if (rank == 3) {

        calc\_len(&data);

      }

      buffer = malloc(BUFFER\_SIZE);

      position = 0;

      MPI\_Pack(data.text, 50, MPI\_CHAR, buffer, BUFFER\_SIZE, &position, MPI\_COMM\_WORLD);

      MPI\_Pack(&data.a\_count, 1, MPI\_INT, buffer, BUFFER\_SIZE, &position, MPI\_COMM\_WORLD);

      MPI\_Pack(&data.length, 1, MPI\_INT, buffer, BUFFER\_SIZE, &position, MPI\_COMM\_WORLD);

      MPI\_Pack(&data.is\_done, 1, MPI\_INT, buffer, BUFFER\_SIZE, &position, MPI\_COMM\_WORLD);

      MPI\_Send(buffer, position, MPI\_PACKED, next, tag, MPI\_COMM\_WORLD);

      free(buffer);

**Wnioski:**

* **Opanowanie podstaw MPI**  
  Przeprowadzono ćwiczenia umożliwiające poznanie podstawowych operacji z przesyłaniem komunikatów w środowisku MPI, takich jak inicjalizacja, przesyłanie danych między procesami oraz finalizacja.
* **Realizacja modelu sztafety w MPI**  
  Opracowanie programu propagującego komunikaty w konwencji pierścienia pozwoliło na zrozumienie mechanizmów komunikacji między procesami. Szczególnie istotne było wyznaczenie ról poszczególnych procesów (poprzednika i następcy) oraz modyfikacja danych przesyłanych w pierścieniu.
* **Praktyczne zastosowanie struktur danych w MPI**  
  Stworzenie "bogatej" struktury danych w języku C, a następnie jej przesyłanie za pomocą typu MPI\_PACKED, umożliwiło pogłębienie umiejętności związanych z zaawansowaną obsługą danych w środowisku MPI.
* **Zastosowanie przetwarzania potokowego**  
  Rozwinięcie programu do realizacji przetwarzania potokowego pozwoliło na wdrożenie schematu, w którym wiele danych jest przetwarzanych jednocześnie w sposób równoległy. To ćwiczenie pokazało potencjalne przyspieszenie wynikające z równoległości.
* **Efektywność i testowanie działania**  
  Programy były uruchamiane i testowane na różnych liczbach procesów, co umożliwiło ocenę poprawności działania oraz porównanie wyników w zależności od parametrów uruchomienia.
* **Nowy typ danych w MPI**  
  Użycie funkcji MPI\_Type\_create\_struct do utworzenia nowego typu danych dla struktury w języku C podkreśliło możliwości dostosowywania komunikacji w MPI do specyficznych wymagań aplikacji.

**Lab12:**

**Cel:**

* Doskonalenie podstaw programowania z przesyłaniem komunikatów MPI

**Zadanie:**

1. Przygotowanie projektu
2. Analiza sekwencyjnego obliczania liczby PI
3. Zrównoleglenie obliczeń liczby PI

if (rank == 0) {

    printf("Podaj maksymalną liczbę wyrazów do obliczenia przybliżenia PI\n");

    scanf("%d", &max\_liczba\_wyrazow);

  }

  // Rozsyłanie liczby iteracji do wszystkich procesów

  MPI\_Bcast(&max\_liczba\_wyrazow, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

  // Obliczanie zakresów iteracji dla każdego procesu

  int my\_start = rank \* (max\_liczba\_wyrazow / size);

  int my\_end = (rank == size - 1) ? max\_liczba\_wyrazow : my\_start + (max\_liczba\_wyrazow / size);

  for (int i = my\_start; i < my\_end; i++) {

    int j = 1 + 4 \* i;

    local\_sum\_plus += 1.0 / j;

    local\_sum\_minus += 1.0 / (j + 2.0);

  }

  // Redukcja wyników lokalnych do procesu 0

  MPI\_Reduce(&local\_sum\_plus, &global\_sum\_plus, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

  MPI\_Reduce(&local\_sum\_minus, &global\_sum\_minus, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

  if (rank == 0) {

    SCALAR pi\_approx = 4 \* (global\_sum\_plus - global\_sum\_minus);

    printf("PI obliczone: \t\t\t%20.15lf\n", pi\_approx);

    printf("PI z biblioteki matematycznej: \t%20.15lf\n", M\_PI);

  }

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

1. Zmniejszenie ilości wymaganych do równoległych obliczeń iloczynu macierzy i wektora, użycie Bcast i Scatter

    MPI\_Scatter( a, WYMIAR\*n\_wier, MPI\_DOUBLE, a\_local, WYMIAR\*n\_wier, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD );

    MPI\_Bcast(x, WYMIAR, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

Około 50 linii kodu zostało zastąpione 2. Pokazuje to rozbudowanie biblioteki MPI jednocześnie dając możliwość na szczegółową implementacje przy jego użyciu.

**Wnioski:**

* **Obliczanie liczby π z szeregu Leibniza**  
  Zrównoleglenie obliczeń liczby π umożliwiło efektywne podziałanie pracy między procesy. Procesy indywidualnie obliczały swoje części sumy, a proces o randze 0 zbierał i sumował wyniki, co pokazało możliwości redukcji komunikacyjnej w MPI.
* **Równoległe mnożenie macierzy przez wektor**  
  Implementacja algorytmu mnożenia macierz-wektor pokazała znaczenie poprawnej dekompozycji danych i synchronizacji procesów. Testowanie wyników z użyciem komunikacji punkt-punkt oraz grupowej umożliwiło analizę różnic wydajności i czytelności kodu.
* **Optymalizacja przez komunikację grupową**  
  Zastąpienie wymiany punkt-punkt (MPI\_Send/MPI\_Recv) funkcjami grupowymi (MPI\_Bcast, MPI\_Gather, MPI\_Scatter itp.) znacząco uprościło kod i przyczyniło się do poprawy jego czytelności i wydajności, szczególnie przy większej liczbie procesów.

**Lab13:**

**Cel:**

* Doskonalenie umiejętności analizy wydajności programów równoległych

**Zadanie:**

1. Przygotowanie projektu
2. Przeprowadzenie testów dla różnych ilości wątków.
3. Zebranie danych w arkuszu.
4. Wykonanie wykresów.

Całka (OpenMP):

|  |  |  |  |
| --- | --- | --- | --- |
| Thread Count | Czas [s] | Przyspieszenie | Fektywność |
| 1 | 0,108229 | 1 | 1 |
| 2 | 0,057564 | 1,880150789 | 0,940075 |
| 4 | 0,038389 | 2,819271145 | 0,704818 |
| 8 | 0,023229 | 4,65921908 | 0,582402 |
| 16 | 0,02184 | 4,955540293 | 0,309721 |

Mnożenie Macierz – Wektor (MPI)

|  |  |  |  |
| --- | --- | --- | --- |
| Liczba procesów | Czas [s] | Przyspieszenie | Fektywność |
| 1 | 0,083597 | 1 | 1 |
| 2 | 0,115793 | 0,721952104 | 0,360976 |
| 4 | 0,042919 | 1,947785363 | 0,486946 |
| 8 | 0,039835 | 2,098581649 | 0,262323 |
| 16 | 0,073133 | 1,143081783 | 0,071443 |

**Wnioski:**

* **Całka (OpenMP):**
  + Zależność czasu wykonania od liczby wątków pokazuje poprawę wydajności do pewnego momentu, ale osiągnięcie idealnego przyspieszenia (linearnego wzrostu) nie jest możliwe.
  + Przy większej liczbie wątków narzut związany z synchronizacją oraz zarządzaniem wątkami zaczyna dominować, zmniejszając efektywność obliczeń.
  + Efektywność spada istotnie szybciej niż w przypadku mat\_vec, co wskazuje na potencjalne problemy z równomiernym podziałem pracy między wątki.
* **Mat\_vec (MPI):**
  + Wydajność zmniejsza się wraz ze wzrostem liczby procesów. Początkowy wzrost szybkości jest ograniczony narzutem komunikacyjnym i zarządzaniem procesami.
  + Dla niewielkiej liczby wątków (1–4), wzrost wydajności jest zauważalny, ale znacząco poniżej idealnego przyspieszenia. Wynika to z rosnącej trudności w efektywnym wykorzystaniu większej liczby rdzeni.
  + Wysoka liczba wątków prowadzi do efektu przeciążenia (overhead), co wpływa na spadek efektywności.
  + Skalowalność algorytmu jest ograniczona przez jego charakter (możliwe wąskie gardła w dostępie do pamięci lub synchronizacji).