Тестовое задание

Задачи на работу с lib MPI
Боязитов Вадим, 932125
18.04.2025

Задание 1

1. Нулевой процесс передает на все остальные процессы совой номер, используя функции двухточечного обмена. Каждый процесс принимает сообщение от процесса с номером ноль и выдает на экран свой номер и принятое сообщение.

Алгоритм

1. Подготовка

Создать переменные для получения информации о текущем исполняемом процессе.

Создать переменные для приема данных от других процессов.

Инициализация MPI.

2. Условие

Если процесс 0, то отправить в цикле остальным сообщение.

Иначе ждать сообщение от процесса 0.

3. Завершить работу

Программная реализация

Окружение Linix. Язык C.

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv) {
    int rank, size;
    int received_data;
    MPI_Status status;

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

    if (rank == 0) {
        // Процесс 0 отправляет свой номер всем остальным процессам
        for (int i = 1; i < size; i++) {
            MPI_Send(&rank, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
        }
    } else {
        // Остальные процессы принимают сообщение от процесса 0
        MPI_Recv(&received_data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
        printf("Process %d received message: %d\n", rank, received_data);
    }

    // Завершение работы с MPI
    MPI_Finalize();
    return 0;
}
```

Тестирование

```
katet_3@DESKTOP-RBO1289:/mnt/d/code/4course_2sem/parallel_programming/test$ make run TARGET=task_1
mpirun --mca btl_vader_single_copy_mechanism none -np 4 --oversubscribe ./task_1.out
Process 3 received message: 0
Process 1 received message: 0
Process 2 received message: 0
```

```
katet_3@DESKTOP-RBO1289:/mnt/d/code/4course_2sem/parallel_programming/test$ make run TARGET=task_1
mpirun --mca btl_vader_single_copy_mechanism none -np 4 --oversubscribe ./task_1.out
Process 2 received message: 0
Process 3 received message: 0
Process 1 received message: 0
```

Вывод

Посредством функций MPI, получилось с коммутировать работу разных процессов.

Задание 2

2. Передать случайное число из диапазона [0,9] по кругу начиная с процесса с номером ноль. При этом каждый процесс должен добавлять к получаемому числу свой номер и отправлять дальше. Задачу решить с использованием функции двухточечного обмена. В конце работы программы процесс с номер ноль выдает на экран исходное случайное число и полученное им число. Все остальные процессы выдают на экран свой номер и переданное им значение.

Алгоритм

1. Подготовка

Создать переменные для получения информации о текущем исполняемом процессе.

Создать переменные для приема данных от других процессов.

Создать переменную для случайного числа.

Инициализация MPI

2. Генерация случайного числа

Если процесс 0, то сгенерировать число и запомнить в переменной.

3. Передача числа по кругу

Если Процесс 0:

Отправляет случайное число следующему процессу.

Ожидает число от последнего процесса.

Иначе:

Процесс ждет сообщение от предыдущего процесса.

Текущий процесс отправляет сумму полученого числа со своим номером, следующему процессу.

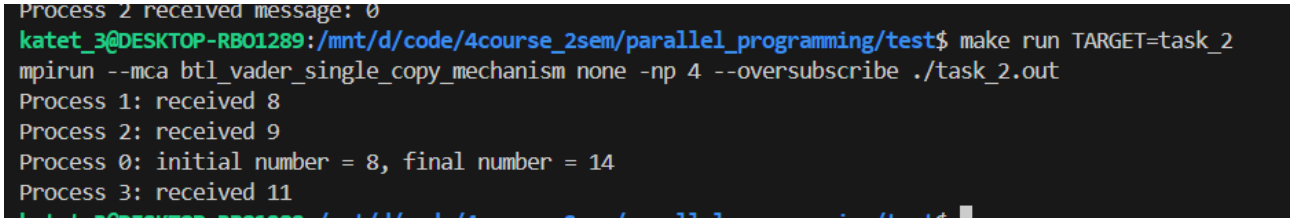
4. Завершить работу

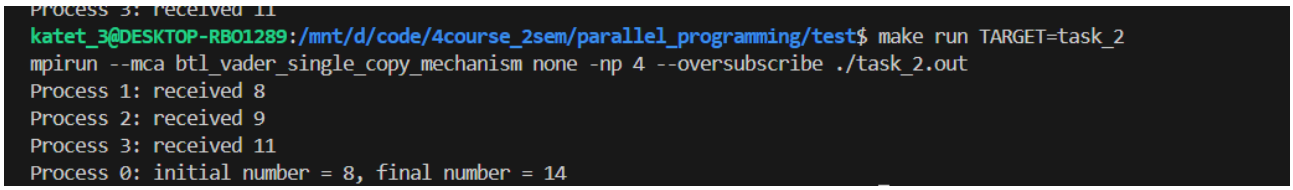
Программная реализация

Окружение Linix. Язык C.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>

int main(int argc, char** argv) {
    int rank, size;
    int number, initial_number;
    MPI_Status status;

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

    // Инициализация генератора случайных чисел (только процесс 0)
    if (rank == 0) {
        srand(time(NULL));
        initial_number = rand() % 10;  // Сохраняем исходное число
        number = initial_number;       // Рабочая переменная для передачи
    }

    // Передача по кругу
    if (rank == 0) {
        // Процесс 0 отправляет число следующему процессу
        MPI_Send(&number, 1, MPI_INT, (rank + 1) % size, 0, MPI_COMM_WORLD);

        // Затем получает число от последнего процесса
        MPI_Recv(&number, 1, MPI_INT, size - 1, 0, MPI_COMM_WORLD, &status);

        // Вывод исходного и итогового числа
        printf("Process %d: initial number = %d, final number = %d\n",
               rank, initial_number, number);
    } else {
        // Все остальные процессы получают число, добавляют свой rank и отправляют дальше
        MPI_Recv(&number, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, &status);
        printf("Process %d: received %d\n", rank, number);

        number += rank;
        MPI_Send(&number, 1, MPI_INT, (rank + 1) % size, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

Тестирование





Вывод

Посредством функций MPI, получилось с коммутировать работу разных процессов.

Задание 3

3. Написать параллельную программу умножения матрицы на вектор y=Ax. При условии, что исходная матрица «A» и вектор «x» хранятся и заполняются случайным числами на процессе с номером ноль. Можно использовать переменные A_global и x_global для их хранения. Необходимо используя операции коллективного взаимодействия MPI_Bcast() и MPI_Scatter() разослать вектор x_global на все процессы и распределить по процессам матрицу A_global.
   Далее каждый процесс умножит распределенные ему строки матрицы «A» на вектор «x» и получит свою часть вектора «y». В результате работы программы каждый процесс должен располагать в своем распоряжении целиком вектором «y», а не его частью. Для этого использовать операцию MPI_Allgather().
   Для проверки правильности работы параллельной программы на процессе с номером ноль умножить последовательно A_global на x_global и сравнить результат с полученным в результате работы параллельного алгоритма «y».

Алгоритм

1. Подготовка

Инициализация MPI

Создать переменные для получения информации о текущем исполняемом процессе.

Создать переменные для умножения вектора и матрицы.

2. Генерация случайной матрицы и вектора

Если процесс 0, то сгенерировать матрицу и вектор в переменной.

3. Отправить вектор другим процессам.

4. Распределить строки матрицы между процессами

5. Каждый процесс умножает свои строки на вектор

6. Собрать полученные части вектора

7. Вывод результата с 0 процесса

8. Освобождение памяти

9. Завершить работу

Программная реализация

Окружение Linix. Язык C.

```
34      MPI_Init(&argc, &argv);
35
36      int rank, size;
37      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
38      MPI_Comm_size(MPI_COMM_WORLD, &size);
39
40      // Размеры матрицы и вектора, строки, столбцы
41      const int N = 8;
42      const int M = 8;
43
44      double* A_global = NULL;
45      double* x_global = (double*)malloc(M * sizeof(double));
46
```

```
47      // Процесс 0 заполняет матрицу и вектор случайными числами
48      if (rank == 0) {
49          A_global = (double*)malloc(N * M * sizeof(double));
50          srand(time(NULL));
51          fill_random_matrix(A_global, N, M);
52          fill_random_vector(x_global, M);
53
54          printf("Matrix A:\n");
55          for (int i = 0; i < N; ++i) {
56              for (int j = 0; j < M; ++j) {
57                  printf("%.4f ", A_global[i * M + j]);
58              }
59              printf("\n");
60          }
61
62          printf("\nVector x:\n");
63          for (int j = 0; j < M; ++j) {
64              printf("%.4f ", x_global[j]);
65          }
66          printf("\n");
67      }
```

```
68
69      // Рассылаем вектор x всем процессам
70      MPI_Bcast(x_global, M, MPI_DOUBLE, 0, MPI_COMM_WORLD);
71
72      // Распределяем строки матрицы между процессами
73      int rows_per_process = N / size;
74      double* A_local = (double*)malloc(rows_per_process * M * sizeof(double));
75
76      MPI_Scatter(A_global, rows_per_process * M, MPI_DOUBLE,
77                  A_local, rows_per_process * M, MPI_DOUBLE,
78                  0, MPI_COMM_WORLD);
```

```
80      // Каждый процесс умножает свои строки на вектор
81      double* y_local = (double*)malloc(rows_per_process * sizeof(double));
82      for (int i = 0; i < rows_per_process; ++i) {
83          y_local[i] = 0.0;
84          for (int j = 0; j < M; ++j) {
85              y_local[i] += A_local[i * M + j] * x_global[j];
86          }
87      }
```

```
89      // Собираем все части вектора y на всех процессах
90      double* y_global = (double*)malloc(N * sizeof(double));
91      MPI_Allgather(y_local, rows_per_process, MPI_DOUBLE,
92                    y_global, rows_per_process, MPI_DOUBLE,
93                    MPI_COMM_WORLD);
```

```
95      // Проверка на процессе 0
96      if (rank == 0) {
97          double* y_seq = (double*)malloc(N * sizeof(double));
98          sequential_matrix_vector_mult(A_global, x_global, y_seq, N, M);
99
100         printf("\nSequential result:\n");
101         for (int i = 0; i < N; ++i) {
102             printf("%.4f ", y_seq[i]);
103         }
104         printf("\n");
105
106         printf("\nParallel result:\n");
107         for (int i = 0; i < N; ++i) {
108             printf("%.4f ", y_global[i]);
109         }
110         printf("\n");
111
```

```
111
112         // Проверка совпадения результатов
113         int correct = 1;
114         for (int i = 0; i < N; ++i) {
115             if (fabs(y_seq[i] - y_global[i]) > 1e-10) {
116                 correct = 0;
117                 break;
118             }
119         }
120
121         if (correct) {
122             printf("\nResults match!\n");
123         } else {
124             printf("\nResults don't match!\n");
125         }
126
127         free(y_seq);
128     }
129
```

```
130         if (rank == 0) {
131             free(A_global);
132         }
133         free(A_local);
134         free(y_local);
135         free(y_global);
136         free(x_global);
137
138         MPI_Finalize();
139         return 0;
140     }
```

Тестирование

```
katet_3@DESKTOP-RBO1289:/mnt/d/code/4course_2sem/parallel_programming/test$ make run TARGET=task_3
mpirun --mca btl_vader_single_copy_mechanism none -np 4 --oversubscribe ./task_3.out
Matrix A:
0.7955 0.5987 0.1287 0.6160 0.8038 0.7559 0.4392 0.2538
0.8010 0.2603 0.3939 0.0680 0.2089 0.3983 0.0670 0.3702
0.3085 0.9497 0.5739 0.5786 0.2160 0.7018 0.4794 0.3742
0.3899 0.5732 0.4810 0.7792 0.7806 0.1536 0.8404 0.5760
0.7522 0.9691 0.1920 0.5561 0.7251 0.6312 0.8098 0.5260
0.8915 0.2037 0.5940 0.1004 0.6020 0.6611 0.4706 0.9104
0.6108 0.0445 0.4890 0.8268 0.7463 0.9684 0.2010 0.1362
0.5416 0.6821 0.9154 0.3221 0.8356 0.7558 0.8981 0.5878
```

```
Vector x:
0.7250 0.0901 0.1439 0.4500 0.7213 0.9537 0.9760 0.6128

Sequential result:
2.8113 1.5142 2.1745 2.6370 3.1483 2.8773 2.6309 3.2913

Parallel result:
2.8113 1.5142 2.1745 2.6370 3.1483 2.8773 2.6309 3.2913

Results match!
```

Вывод

Посредством функций MPI, получилось с коммутировать работу разных процессов.