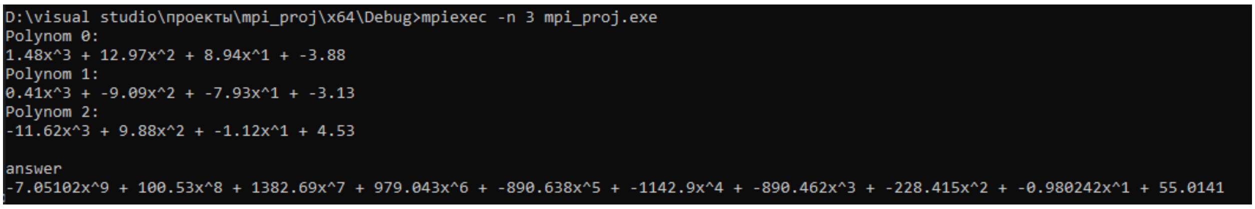
Лабораторная работа №4

Производные типы данных в MPI

Код: https://pastebin.com/sz136Jfe

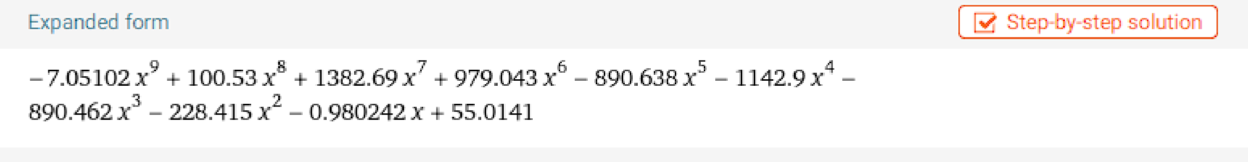
Результаты работы:

3 полинома 3 степени:



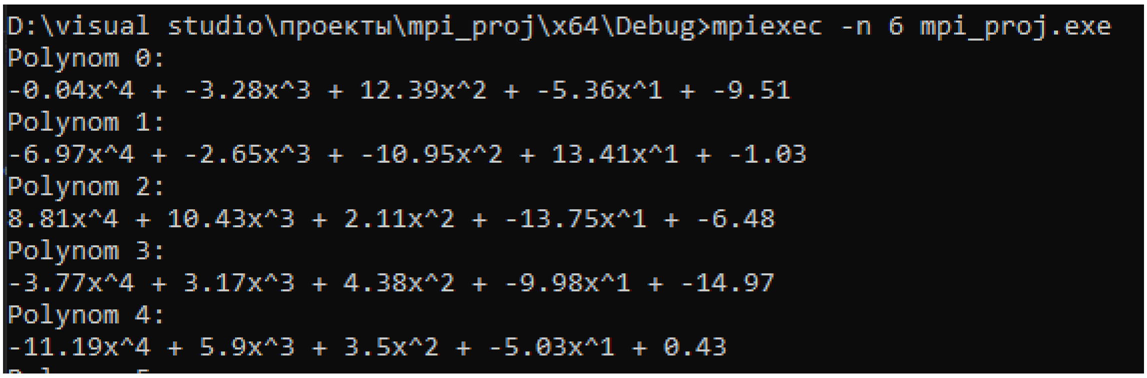
```
D:\visual studio\проекты\mpi_proj\x64\Debug>mpiexec -n 3 mpi_proj.exe
Polynom 0:
1.48x^3 + 12.97x^2 + 8.94x^1 + -3.88
Polynom 1:
0.41x^3 + -9.09x^2 + -7.93x^1 + -3.13
Polynom 2:
-11.62x^3 + 9.88x^2 + -1.12x^1 + 4.53

answer
-7.05102x^9 + 100.53x^8 + 1382.69x^7 + 979.043x^6 + -890.638x^5 + -1142.9x^4 + -890.462x^3 + -228.415x^2 + -0.980242x^1 + 55.0141
```

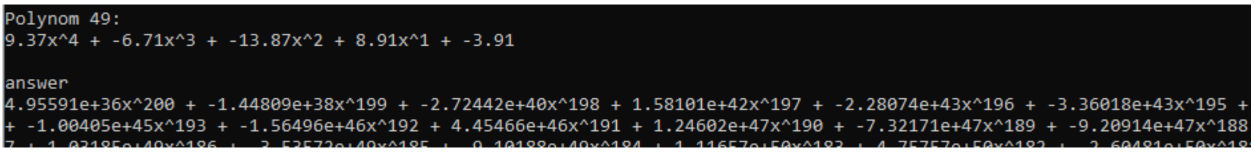


$$-7.05102\,x^9 + 100.53\,x^8 + 1382.69\,x^7 + 979.043\,x^6 - 890.638\,x^5 - 1142.9\,x^4 - 890.462\,x^3 - 228.415\,x^2 - 0.980242\,x + 55.0141$$

50 полиномов 4 степени:



```
D:\visual studio\проекты\mpi_proj\x64\Debug>mpiexec -n 6 mpi_proj.exe
Polynom 0:
-0.04x^4 + -3.28x^3 + 12.39x^2 + -5.36x^1 + -9.51
Polynom 1:
-6.97x^4 + -2.65x^3 + -10.95x^2 + 13.41x^1 + -1.03
Polynom 2:
8.81x^4 + 10.43x^3 + 2.11x^2 + -13.75x^1 + -6.48
Polynom 3:
-3.77x^4 + 3.17x^3 + 4.38x^2 + -9.98x^1 + -14.97
Polynom 4:
-11.19x^4 + 5.9x^3 + 3.5x^2 + -5.03x^1 + 0.43
Polynom 5
```



```
Polynom 49:
9.37x^4 + -6.71x^3 + -13.87x^2 + 8.91x^1 + -3.91

answer
4.95591e+36x^200 + -1.44809e+38x^199 + -2.72442e+40x^198 + 1.58101e+42x^197 + -2.28074e+43x^196 + -3.36018e+43x^195 +
+ -1.00405e+45x^193 + -1.56496e+46x^192 + 4.45466e+46x^191 + 1.24602e+47x^190 + -7.32171e+47x^189 + -9.20914e+47x^188
7 + 1.03185e+49x^186 + -3.53572e+49x^185 + -9.10188e+49x^184 + 1.11657e+50x^183 + 4.75757e+50x^182 + -2.60481e+50x^18
```

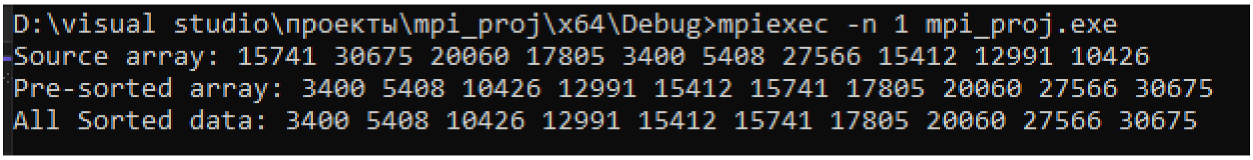
Лабораторная работа №5

Управление группами процессов и коммуникаторами

Код: https://pastebin.com/6HFCT6uw

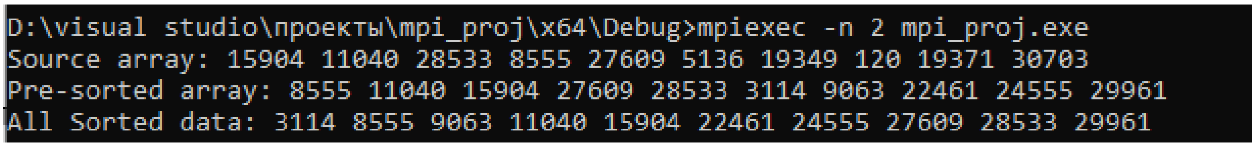
Результаты работы:

Один процесс:



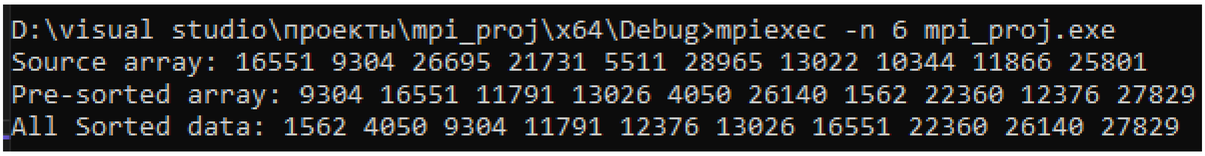
```
D:\visual studio\проекты\mpi_proj\x64\Debug>mpiexec -n 1 mpi_proj.exe
Source array: 15741 30675 20060 17805 3400 5408 27566 15412 12991 10426
Pre-sorted array: 3400 5408 10426 12991 15412 15741 17805 20060 27566 30675
All Sorted data: 3400 5408 10426 12991 15412 15741 17805 20060 27566 30675
```

Два процесса:



```
D:\visual studio\проекты\mpi_proj\x64\Debug>mpiexec -n 2 mpi_proj.exe
Source array: 15904 11040 28533 8555 27609 5136 19349 120 19371 30703
Pre-sorted array: 8555 11040 15904 27609 28533 3114 9063 22461 24555 29961
All Sorted data: 3114 8555 9063 11040 15904 22461 24555 27609 28533 29961
```

6 процессов:



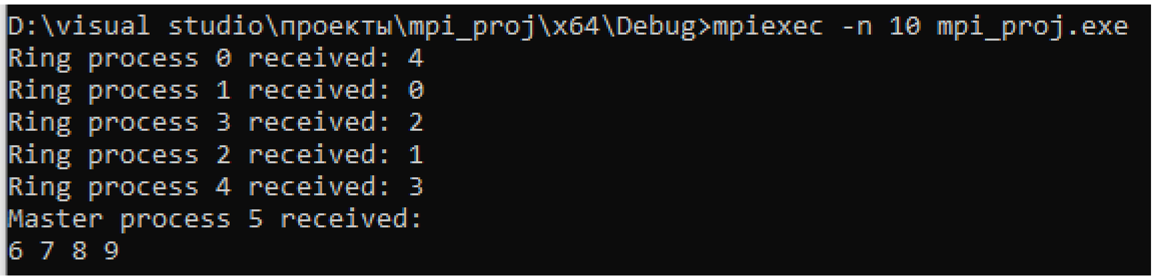
```
D:\visual studio\проекты\mpi_proj\x64\Debug>mpiexec -n 6 mpi_proj.exe
Source array: 16551 9304 26695 21731 5511 28965 13022 10344 11866 25801
Pre-sorted array: 9304 16551 11791 13026 4050 26140 1562 22360 12376 27829
All Sorted data: 1562 4050 9304 11791 12376 13026 16551 22360 26140 27829
```

Лабораторная работа №6

Виртуальные топологии

Код: https://pastebin.com/R95wxyMb

Результаты работы:



```
D:\visual studio\проекты\mpi_proj\x64\Debug>mpiexec -n 10 mpi_proj.exe
Ring process 0 received: 4
Ring process 1 received: 0
Ring process 3 received: 2
Ring process 2 received: 1
Ring process 4 received: 3
Master process 5 received:
6 7 8 9
```

Приложение. Коды.

1.

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

#include <cstddef>
#include <cstdio>
#include <iostream>
#include <random>

const int MAX_DEGREE = 500;
const int POLY_DEGREE = 5;
const int N = 5;

using namespace std;

typedef struct {
    int degree;
    double coefficients[MAX_DEGREE];
} Polynomial;

Polynomial multPolynoms(Polynomial a, Polynomial b) {
    Polynomial result;
    result.degree = a.degree + b.degree;
    for (int i = 0; i <= result.degree; i++) {
        result.coefficients[i] = 0.0;
    }
    for (int i = 0; i <= a.degree; i++) {
        for (int j = 0; j <= b.degree; j++) {
            result.coefficients[i + j] += a.coefficients[i] * b.coefficients[j];
        }
    }
    return result;
}

double generateCoeff(int index_to_randomize) {
    random_device rd;
    mt19937 gen(rd());
    uniform_real_distribution<> distrib(-15.0, 15.0);
    double random_number = distrib(gen);
    random_number = std::round(random_number * 100.0) / 100.0;
    return random_number;
}

Polynomial initPoly() {
    Polynomial poly;
    for (int i = 0; i < POLY_DEGREE; i++) {
        poly.coefficients[i] = generateCoeff(i);
    }
    poly.degree = POLY_DEGREE - 1;
    return poly;
}

void printPoly(Polynomial poly) {
    for (int i = 0; i < poly.degree + 1; i++) {
        if (i == poly.degree) {
            cout << poly.coefficients[i] << endl;
        }
        else {
            cout << poly.coefficients[i] << "x^" << poly.degree - i << " + ";
        }

    }
}

MPI_Datatype init_MPI_POLY() {
    MPI_Datatype MPI_POLY;
    int blocklengths[2] = { 1, MAX_DEGREE };
    MPI_Datatype types[2] = { MPI_INT, MPI_DOUBLE };
    MPI_Aint displacements[2] = { offsetof(Polynomial, degree), offsetof(Polynomial, coefficients) };
    MPI_Type_create_struct(2, blocklengths, displacements, types, &MPI_POLY);
    MPI_Type_commit(&MPI_POLY);
    return MPI_POLY;
}

MPI_Datatype init_MPI_VECTOR(MPI_Datatype MPI_POLY) {
    MPI_Datatype MPI_POLY_VECTOR;
    MPI_Type_contiguous(N, MPI_POLY, &MPI_POLY_VECTOR);
    MPI_Type_commit(&MPI_POLY_VECTOR);
    return MPI_POLY_VECTOR;
}

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

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

    if (size - 1 > N) {
        cout << "Number of -n <number> must be bigger than N of polynoms!";
        return 0;
    }

    MPI_Datatype MPI_POLY = init_MPI_POLY();
    MPI_Datatype MPI_POLY_VECTOR = init_MPI_VECTOR(MPI_POLY);

    Polynomial result;
    Polynomial polys[N];

    if (rank == 0) {
        for (int i = 0; i < N; i++) {
            polys[i] = initPoly();
            cout << "Polynom " << i << ":" << endl;
            printPoly(polys[i]);
        }
        for (int i = 1; i < size; i++) {
            MPI_Send(&polys, 1, MPI_POLY_VECTOR, i, 1, MPI_COMM_WORLD);
        }

        MPI_Recv(&result, 1, MPI_POLY, 1, 1, MPI_COMM_WORLD, &status);
        Polynomial temp;
        for (int i = 2; i < size; i++) {
            MPI_Recv(&temp, 1, MPI_POLY, i, 1, MPI_COMM_WORLD, &status);
            result = multPolynoms(result, temp);
        }

        cout << "\nanswer" << endl;
        printPoly(result);

    }
    else {
        MPI_Recv(&polys, 1, MPI_POLY_VECTOR, 0, 1, MPI_COMM_WORLD, &status);
        int chunk_size = N / (size - 1);
        int chunk_start = chunk_size * (rank - 1);
        int chunk_end = ((rank - 1) == (size - 1) - 1) ? N : chunk_start + chunk_size;
        //cout << rank << " counts from " << chunk_start << " to " << chunk_end - 1 << endl;

        Polynomial poly_temp = polys[chunk_start];
        for (int i = chunk_start + 1; i < chunk_end; i++) {
            poly_temp = multPolynoms(poly_temp, polys[i]);
        }

        //printPoly(poly_temp);
        MPI_Send(&poly_temp, 1, MPI_POLY, 0, 1, MPI_COMM_WORLD);
    }

    MPI_Type_free(&MPI_POLY);
    MPI_Type_free(&MPI_POLY_VECTOR);
    MPI_Finalize();

    return 0;
}
```

2.

```
#include <mpi.h>

#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>

using namespace std;

int* mergeSortBlocks(int* PreSortedData, int dataSize, int LBLOCK) {
  // Calculate the number of blocks
  int numBlocks = (dataSize + LBLOCK - 1) / LBLOCK;

  // Create an array to store the merged data
  int* mergedData = new int[dataSize];

  // Create an array of pointers to the beginning of each block
  int** blockPointers = new int*[numBlocks];
  for (int i = 0; i < numBlocks; ++i) {
    blockPointers[i] = PreSortedData + i * LBLOCK;
  }

  // Create an array to store the indices of the elements in each block
  int* blockIndices = new int[numBlocks];
  for (int i = 0; i < numBlocks; ++i) {
    blockIndices[i] = 0;
  }

  int mergedIndex = 0;
  while (mergedIndex < dataSize) {
    int minVal = INT_MAX;
    int minBlockIndex = -1;

    // Find the minimum value among the blocks
    for (int i = 0; i < numBlocks; ++i) {
      // Check if we reached the end of the current block
      if (blockIndices[i] <
          (i == numBlocks - 1 ? dataSize - i * LBLOCK : LBLOCK)) {
        if (blockPointers[i][blockIndices[i]] < minVal) {
          minVal = blockPointers[i][blockIndices[i]];
          minBlockIndex = i;
        }
      }
    }

    // Add the minimum value to the merged array
    mergedData[mergedIndex++] = minVal;
    blockIndices[minBlockIndex]++;
  }

  // Clean up allocated memory
  delete[] blockPointers;
  delete[] blockIndices;

  return mergedData;
}

// Функция для сравнения элементов при сортировке
int cmp(const void* a, const void* b) { return (*(int*)a - *(int*)b); }

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

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

  const int N = 10;  // Количество элементов в массиве
  const int blockSize = N / size;  // Размер блока для каждого процесса
  const int remainder = N % size;  // Остаток от деления

  // Вычисляем начальный и конечный индекс для каждого процесса
  int start = blockSize * rank + std::min(rank, remainder);
  int end = start + blockSize + (rank < remainder ? 1 : 0);
```

```
// Генерируем случайные числа в промежутке от 0 до 100000
srand(time(nullptr) + rank);
int* data = new int[N];
for (int i = 0; i < N; ++i) {
  data[i] = rand() % 100001;
}

// Выводим исходный массив на экран
if (rank == 0) {
  cout << "Source array: ";
  for (int i = 0; i < N; ++i) {
    cout << data[i] << " ";
  }
  cout << endl;
}

int* ranks = new int[size];
for (int i = 0; i < size; ++i) {
  ranks[i] = i;
}

// Создаем коммуникатор для группы процессов
MPI_Group worldGroup, newGroup;
MPI_Comm_group(MPI_COMM_WORLD, &worldGroup);
MPI_Group_incl(worldGroup, size, ranks, &newGroup);
MPI_Comm newComm;
MPI_Comm_create(MPI_COMM_WORLD, newGroup, &newComm);

// Сортируем блоки данных каждого процесса
qsort(data + start, end - start, sizeof(int), cmp);

// Собираем отсортированные блоки данных
int* sortedData = nullptr;
if (rank == 0) {
  sortedData = new int[N];
}
```

```
  MPI_Gather(data + start, end - start, MPI_INT, sortedData, end - start,
             MPI_INT, 0, newComm);

  // Выводим отсортированный массив
  if (rank == 0) {
    cout << "Pre-sorted array: ";
    for (int i = 0; i < N; ++i) {
      cout << sortedData[i] << " ";
    }
    cout << endl;

    int* sortedDataAll = mergeSortBlocks(sortedData, N, blockSize);

    cout << "All Sorted data: ";
    for (int i = 0; i < N; ++i) {
      cout << sortedDataAll[i] << " ";
    }
    cout << endl;
    delete[] sortedDataAll;
  }

  delete[] data;
  delete[] sortedData;

  MPI_Finalize();

  return 0;
}
```

3.

```
#include <mpi.h>


#include <iostream>

#include <vector>


using namespace std;


int main(int argc, char** argv) {

  MPI_Init(&argc, &argv);


  int rank, size;

  MPI_Comm_size(MPI_COMM_WORLD, &size);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);


  if (size % 2 != 0) {

    if (rank == 0) {

      cerr << "Error: Number of processes must be even." << endl;

    }

    MPI_Finalize();

    return 1;

  }


  // Создаем коммуникаторы для подгрупп

  int ring_size = size / 2;

  int master_slave_size = size / 2;


  MPI_Comm ring_comm;

  MPI_Comm master_slave_comm;


  MPI_Comm_split(MPI_COMM_WORLD, rank < ring_size ? 0 : 1, rank, &ring_comm);

  MPI_Comm_split(MPI_COMM_WORLD, rank >= ring_size ? 0 : 1,

               rank % master_slave_size, &master_slave_comm);


  // Кольцевой обмен

  if (rank < ring_size) {

    int ring_rank;
    MPI_Comm_rank(ring_comm, &ring_rank);
    int data = rank;
    int next_rank = (ring_rank + 1) % ring_size;
    int prev_rank = (ring_rank + ring_size - 1) % ring_size;
    int received_data;
    MPI_Sendrecv(&data, 1, MPI_INT, next_rank, 0, &received_data, 1, MPI_INT,
                 prev_rank, 0, ring_comm, MPI_STATUS_IGNORE);
    cout << "Ring process " << rank << " received: " << received_data << endl;
  }

  // Master-slave обмен
  if (rank >= ring_size) {
    int master_slave_rank;
    MPI_Comm_rank(master_slave_comm, &master_slave_rank);
    int data = rank;

    if (master_slave_rank == 0) {  // Master
      vector<int> received_data(master_slave_size);
      for (int i = 1; i < master_slave_size; ++i) {
        MPI_Recv(&received_data[i], 1, MPI_INT, i, 0, master_slave_comm,
                 MPI_STATUS_IGNORE);
      }
      cout << "Master process " << rank << " received:" << endl;
      for (int i = 1; i < master_slave_size; i++) {
        cout << received_data[i] << " ";
      }
      cout << endl;
    } else {  // Slave
      MPI_Send(&data, 1, MPI_INT, 0, 0, master_slave_comm);
    }
  }

  MPI_Comm_free(&ring_comm);
  MPI_Comm_free(&master_slave_comm);
  MPI_Finalize();
  return 0;
}
```