

Міністерство освіти і науки України

Національний технічний університет України

«Київський політехнічний інститут ім. Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

**Лабораторна робота №8**

“MPI”

з дисципліни “Програмування для Паралельних Компьютерних Систем”

Виконав:

студент 3 курсу групи ІО-52

Бояршин Ігор

Номер заліковки: 5207

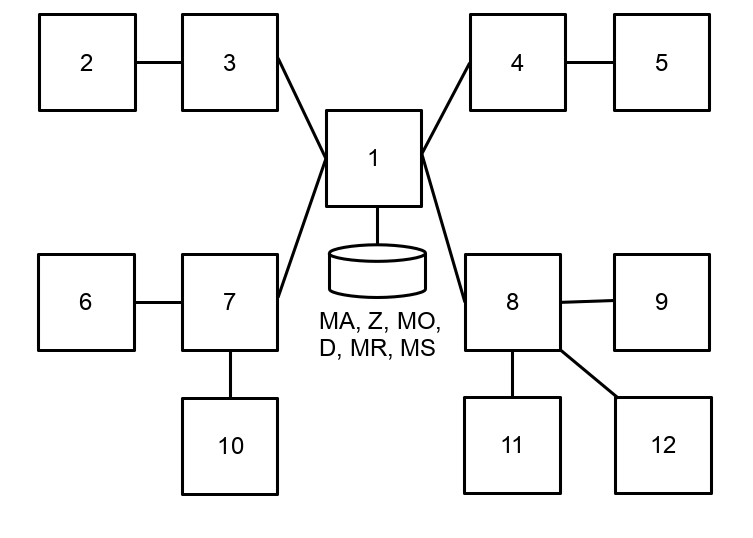
Перевірив:

Корочкін О.В.

Київ 2018 р.

**Техническое задание**

Структура паралельной компъютерной системы с общей памятью:



Выражение для подсчёта:

MA = min(Z) \* MO + d\*(MR\*MS)

Язык программирования:

C++

Библиотека параллельного программирования:

MPI

Средства взаимодействия задач:

Посылка сообщений

**Выполнение работы**

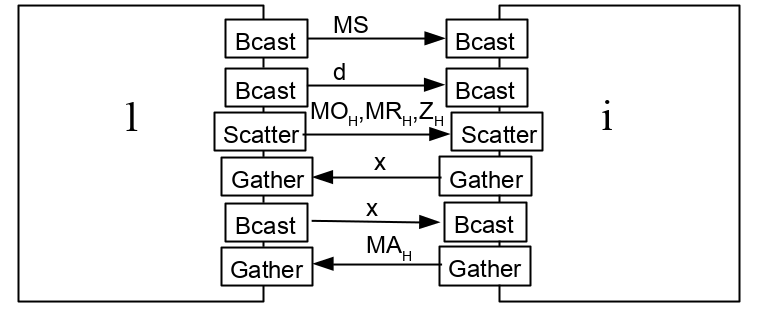
**Этап 1:** Разработка параллельного математического алгоритма

1. xi = min(ZH) , i = (1..P), P – количество потоков, P = 13
2. x = min(x; xi) , i = (1..P)
3. MAH = x\*MOH + d\*(MRH\*MS)

**Этап 2:** Разработка алгоритмов потоков

|  |  |
| --- | --- |
| # | **Ti (1 <= i <= 12)** |
| 1 | **Если** i=1 **то**:   * **Ввод** Z, MO, d, MR, MS |
| 2 | **Если** i=1 **то**:   * **Передать** в T2..12: ZH, MOH, d, MRH, MS   **Иначе**:   * **Принять** из T1: ZH, MOH, d, MRH, MS |
| 3 | **Счёт**: xi = min(ZH) |
| 4 | **Если** i=1 **то**:   * **Принять** из Т2..12: xi * **Счёт**: x = min(x; xi)   **Иначе**:   * **Передать** в Т1: xi |
| 5 | **Если** i=1 **то**:   * **Передать** в Т2..12: x   **Иначе**:   * **Принять** из Т1: х |
| 6 | **Счёт**: MAH = x\*MOH + d\*(MRH\*MS) |
| 7 | **Если** i=1 **то**:   * **Принять** из Т2..12: MAH   **Иначе**:   * **Передать** в Т1: MAH |
| 8 | **Если** i=1 **то**:   * **Вывод** MA |

**Этап 3:** Разработка схемы взаимодействия потоков



**Этап 4:** Разработка программы

//-----------------------------------------------------------------------------

// Lab08: MPI. Graph

// Task: MA = min(Z)\*MO + d\*(MR\*MS)

// Author: Igor Boyarshin

// Date: 12.05.2018

//-----------------------------------------------------------------------------

#include <iostream>

#include <mpi.h>

//-----------------------------------------------------------------------------

// Constants

const static unsigned int N = 12;

const static unsigned int P = 12;

const static unsigned int H = N / P;

const static unsigned int STACK\_SIZE = 100000000;

const static unsigned int ALL\_H = P;

const static unsigned int OUTPUT\_THRESHOLD = 12;

const static bool VECTOR = true;

const static bool MATRIX = !VECTOR;

//-----------------------------------------------------------------------------

// Types

struct Vector {

public:

int \* elements;

const unsigned int hs;

const unsigned int size;

Vector(unsigned int hs) : hs(hs), size(hs \* H) {

elements = new int[size];

std::fill(elements, elements + size, 0);

}

~Vector() {

delete[] elements;

}

int& operator[](unsigned int index) {

return elements[index];

}

const int& operator[](unsigned int index) const {

return elements[index];

}

void\* getVoidPtr() {

return (void\*)(elements);

}

};

//-----------------------------------------------------------------------------

struct Matrix {

public:

int \* elements;

const unsigned int hs;

const unsigned int size;

Matrix(unsigned int hs)

: hs(hs), size(hs \* H \* N) {

elements = new int[size];

std::fill(elements, elements + size, 0);

}

~Matrix() {

delete[] elements;

}

int& operator[](unsigned int index) {

return elements[index];

}

const int& operator[](unsigned int index) const {

return elements[index];

}

void\* getVoidPtr() {

return (void\*)(elements);

}

};

//-----------------------------------------------------------------------------

// Functions

void fillVector(Vector& vector, unsigned int value);

void fillMatrix(Matrix& matrix, unsigned int value);

void outputVector(const Vector& vector);

void outputMatrix(const Matrix& matrix);

unsigned int getSizeFromHs(bool isVector, unsigned int sizeHs);

//-----------------------------------------------------------------------------

void ThreadFunction(unsigned int tid, MPI\_Comm comm) {

// Data

Matrix MA(tid == 0 ? ALL\_H : 1);

Matrix MO(tid == 0 ? ALL\_H : 1);

Matrix MR(tid == 0 ? ALL\_H : 1);

Matrix MS(ALL\_H);

Vector Z(tid == 0 ? ALL\_H : 1);

int d, x;

// Will hold blocks of MO, MR, Z of size H for each thread

int\* buff = new int[(tid == 0) ? (N\*N + N\*N + N) : (H\*N + H\*N + H)];

// Input

if (tid == 0) {

fillMatrix(MO, 1);

fillMatrix(MR, 1);

fillMatrix(MS, 1);

fillVector(Z, 1);

d = 1;

// Pack MO, MR, Z into single buffer

int shift = 0;

for (unsigned int i = 0; i < P; i++) {

memcpy((void\*)(buff + shift),

(void\*)(MO.elements + i \* H \* N),

H \* N \* sizeof(int));

shift += H \* N;

memcpy((void\*)(buff + shift),

(void\*)(MR.elements + i \* H \* N),

H \* N \* sizeof(int));

shift += H \* N;

memcpy((void\*)(buff + shift),

(void\*)(Z.elements + i \* H),

H \* sizeof(int));

shift += H;

}

}

// Distribute

MPI\_Bcast(MS.getVoidPtr(), getSizeFromHs(MATRIX, ALL\_H), MPI\_INT, 0, comm);

MPI\_Bcast(&d, 1, MPI\_INT, 0, comm);

MPI\_Scatter((void\*)buff, 2\*H\*N + H, MPI\_INT,

(void\*)buff, 2\*H\*N + H, MPI\_INT,

0, comm);

// Collect

memcpy(MO.getVoidPtr(), (void\*)(buff), H \* N \* sizeof(int));

memcpy(MR.getVoidPtr(), (void\*)(buff + H \* N), H \* N \* sizeof(int));

memcpy(Z.getVoidPtr(), (void\*)(buff + 2 \* H \* N), H \* sizeof(int));

// Find local min

x = Z[0];

for (unsigned int i = 0; i < H; i++) {

if (Z[i] < x) {

x = Z[i];

}

}

// Collect mins

int mins[P];

MPI\_Gather(&x, 1, MPI\_INT, mins, 1, MPI\_INT, 0, comm);

// Find global min

if (tid == 0) {

for (unsigned int i = 0; i < P; i++) {

if (mins[i] < x) {

x = mins[i];

}

}

}

// Send global min

MPI\_Bcast(&x, 1, MPI\_INT, 0, comm);

// Main calculations

for (unsigned int h = 0; h < H; h++) {

for (unsigned int i = 0; i < N; i++) {

int elem = 0;

for (unsigned int j = 0; j < N; j++) {

elem += MR[h\*N + j] \* MS[j\*N + i];

}

MA[h \* N + i] = x \* MO[h \* N + i] + d \* elem;

}

}

// Collect result

MPI\_Gather(MA.getVoidPtr(), getSizeFromHs(MATRIX, 1), MPI\_INT,

MA.getVoidPtr(), getSizeFromHs(MATRIX, 1), MPI\_INT,

0, comm);

// Output

if (tid == 0) {

outputMatrix(MA);

}

}

//-----------------------------------------------------------------------------

int main() {

int size, rank;

MPI\_Init(0, 0);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

if (rank == 0) {

std::cout << ":> Started" << std::endl;

}

// Create graph

const int nodes = P;

int neigh[nodes] = {4, 1, 2, 2, 1, 1, 3, 4, 1, 1, 1, 1};

int indices[nodes];

int edges[] = {

3, 4, 7, 8,

3,

1, 2,

1, 5,

4,

7,

1, 6, 10,

1, 9, 11, 12,

8,

7,

8,

8

};

indices[0] = neigh[0];

for (unsigned int i = 1; i < nodes; i++) {

indices[i] = indices[i-1] + neigh[i];

}

const unsigned int edgesSize = neigh[nodes - 1];

// Convert from 1-based to 0-based

for (unsigned int i = 0; i < edgesSize; i++) edges[i]--;

MPI\_Comm comm;

MPI\_Graph\_create(MPI\_COMM\_WORLD, nodes, indices, edges, false, &comm);

ThreadFunction(rank, comm);

MPI\_Finalize();

std::cout << ":> Finished " << rank << std::endl;

return 0;

}

//-----------------------------------------------------------------------------

// Function definitions

void fillVector(Vector& vector, unsigned int value) {

for (unsigned int i = 0; i < vector.size; i++) {

vector[i] = value;

}

}

void fillMatrix(Matrix& matrix, unsigned int value) {

for (unsigned int i = 0; i < matrix.size; i++) {

matrix[i] = value;

}

}

void outputVector(const Vector& vector) {

for (unsigned int i = 0; i < vector.size; i++) {

std::cout << vector[i] << " ";

}

std::cout << std::endl;

}

void outputMatrix(const Matrix& matrix) {

for (unsigned int i = 0; i < matrix.hs \* H; i++) {

for (unsigned int j = 0; j < N; j++) {

std::cout << matrix[i \* N + j] << " ";

}

std::cout << std::endl;

}

}

unsigned int getSizeFromHs(bool isVector, unsigned int sizeHs) {

return sizeHs \* H \* (isVector ? 1 : N);

}