МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

## Факультет информационных технологий и робототехники

Кафедра программного обеспечения вычислительной техники

и автоматизированных систем

**Отчет по лабораторной работе № 5**

по дисциплине: ”Распределенная и параллельная обработка данных”

на тему: ***”*** ***Свертывание графа распараллеленности операций”***

Выполнил**:** студент группы 10701214 Зубарев А. А.

Принял**:** проф. Прихожий А.А.

Минск 2017

# Лабораторная работа 5. Свертывание графа распараллеленности операций.

**Цель работы:** Изучение процесс Свертывания графа распараллеленности операций.

## Задание

Спроектировать программу для вычисления шагов свертывания графа, вычисленного по заданной таблице смежности графа, количеству операций, типов, таблицы типов, количеству процессоров каждого типа и времени выполнения операции каждого типа.

**Теоретическая часть**

Граф распараллеленности операций определяется как GR=(V,R), где V – множество вершин, являющихся частичными последовательно-параллельными планами и, в частном случае, операциями; R – множество ребер, описывающих отношение распараллеленности частичных планов (операций). Граф является неориентированным. Две вершины i,k ϵ V соединены ребром, если соответствующие частичные планы выполняются параллельно, если вершины не соединены ребром, соответствующие планы выполняются параллельно. Граф является взвешенным. Каждой вершине i ϵ N ставится в соответствие два параметра: время Ti выполнения частичного плана (в частном случае, это время ti выполнения операции) и вектор bi=(bi1,…,bim) чисел bij процессоров типов j=1,…,m, необходимых для выполнения плана.

Исходный граф G0R на множестве вершин-операций строится по графу GH. В граф G0R вводится ребро (i,k), если на ориентированном графе GH существует путь, соединяющий операции i и k, в противном случае ребро не вводится.

Граф распараллеленности является исходными данными для синтеза последовательно-параллельного плана методом свертывания. Свертывание графа выполняется посредством пошагового склеивания пары вершин, при этом в графе появляются более сложные вершины, которым соответствуют частичные последовательно-параллельные планы и которые также метятся двумя интегрированными величинами:

1. временем Ti выполнения частичного плана;
2. вектором bi чисел процессоров различных типов, необходимых для реализации частичного плана.

**Исходные данные:**

1. Граф GH предшествования операций.

2. Времена ti, i ϵ N выполнения операций.

3. Время tstep шага управления.

4. Число pi доступных процессоров типа i=1,…,Types.

**Результирующие данные:**

1. Шаги свертывания.

2. Граф на каждом шаге.

**Описание алгоритма:**

Рассмотрим последовательно-параллельное планирование с целью минимизации времени решения задачи при заданных ограничениях на ресурсы. Ограничения представим вектором bmax = (b1max,…,bmmax) максимального числа используемых процессоров каждого типа. Свертывание графа GR посредством склеивания пар вершин. Пусть на шаге s планирования в графе для склеивания выбраны вершины x и y. В результате склеивания x и y образуется новая вершина с именем xy. Граф преобразуется путем удаления вершин x и y, удаления множества ребер, инцидентных этим вершинам, а также путем добавления вершины xy и добавления ребер, инцидентных этой вершине и вершинам из множества вершин, смежных с вершиной x в графе . Если вершины x и y соединены ребром, они склеиваются параллельно, при этом новой вершине соответствует параллельный план par(x,y). Если вершины не соединены ребром, они склеиваются последователь- но, при этом новой вершине соответствует последовательный план seq(x,y). Время выполнения Txy и вектор bxy для новой вершины xy вычисляются следующим образом:

* 1. если вершины склеиваются параллельно, то и ;
  2. 2) если вершины склеиваются последовательно, то и .

Параллельное склеивание вершин увеличивает число используемых процессоров, последовательное склеивание увеличивает время выполнения плана. Параллельное склеивание вершин не должно приводить к нарушению ограничений на максимальное число процессоров, описываемых вектором . Если нарушение ограничений имеет место, ребро может быть заранее удалено из графа . Различный порядок склеивания пар вершин приводит к построению различных последовательно-параллельных планов. Существенное влияние на качество синтезируемого плана оказывает потеря ребер, обусловленная с одной стороны, особенностями механизма склеиванием вершин и, с другой стороны, учетом ограничения на число процессоров bmax. Потеря ребра при склеивании вершин x и y происходит тогда, когда существует вершина z, смежная с вершиной x и не смежная с вершиной y. В результате такого склеивания новая вершины xy и вершина z не соединены ребром. Это означает потерю потенциального параллелизма между вершинами xy и z, что может сказаться отрицательно на минимизации времени выполнения синтезируемого плана. В силу сказанного, предпочтительно необходимо выбирать пары вершины, склеивание которых минимизирует потерю ребер. Склеивание пар вершин и свертывание графа распараллеленности завершается тогда, когда граф включает одну вершину. Этой вершине соответствует суперпозиция функций seq и par, описывающая синтезированный последовательно-параллельный план.

### Код программы

using System;

using System.Drawing;

using System.Windows.Forms;

namespace L5\_RiPOD

{

public partial class Form1 : Form

{

private ConvolutionGraph convolution;

public int CurrentStep { get; set; }

public Form1()

{

InitializeComponent();

}

private void button1\_Click(object sender, EventArgs e)

{

convolution = new ConvolutionGraph();

convolution.File\_Load("start\_param.txt");

convolution.Planning();

WriteResults();

}

private void WriteResults()

{

ClearView();

richTextBox1.Text = "Количество шагов: " + (convolution.StepList.Count - 1);

for(int i = 1; i < convolution.StepList.Count; i++)

{

richTextBox1.Text += "\n";

richTextBox1.Text += "Шаг " + i + ": " + convolution.SeqParFunc[i - 1];

}

ViewResults();

textBox1.Text = "";

textBox1.Text = "0";

CurrentStep = 0;

buttonPrev.Enabled = false;

}

private void ViewResults()

{

// типы операций

dataGridView1.ColumnCount = convolution.TypesCount + 1;

dataGridView1.RowCount = convolution.OperationsCount + 1;

dataGridView1.Rows[0].Cells[0].Value = "Operation \\ type";

dataGridView1.ColumnHeadersVisible = false;

dataGridView1.RowHeadersVisible = false;

for (int i = -1; i < convolution.TypesCount; i++)

{

dataGridView1.Rows[0].Cells[i + 1].Style.BackColor = Color.FromArgb(110, 110, 110);

dataGridView1.Rows[0].Cells[i + 1].Style.ForeColor = Color.FromArgb(255, 255, 255);

if (i > -1)

{

dataGridView1.Rows[0].Cells[i + 1].Value = i + 1;

dataGridView1.Columns[i + 1].Width = 50;

}

}

for (int i = 0; i < convolution.OperationsCount; i++)

{

dataGridView1.Rows[i + 1].Cells[0].Style.BackColor = Color.FromArgb(110, 110, 110);

dataGridView1.Rows[i + 1].Cells[0].Style.ForeColor = Color.FromArgb(255, 255, 255);

dataGridView1.Rows[i + 1].Cells[0].Value = i + 1;

}

for (int i = 0; i < convolution.TypesCount; i++)

{

for (int k = 0; k < convolution.ArrayTypes[i].Length; k++)

{

dataGridView1.Rows[convolution.ArrayTypes[i][k]].Cells[i + 1].Style.BackColor = Color.FromArgb(82, 97, 160);

}

}

// матрица смежности

dataGridView2.ColumnCount = convolution.OperationsCount + 1;

dataGridView2.RowCount = convolution.OperationsCount + 1;

dataGridView2.ColumnHeadersVisible = false;

dataGridView2.RowHeadersVisible = false;

for (int i = 0; i < convolution.OperationsCount + 1; i++)

{

// по вертикали

dataGridView2.Rows[i].Cells[0].Style.BackColor = Color.FromArgb(110, 110, 110);

dataGridView2.Rows[i].Cells[0].Style.ForeColor = Color.FromArgb(255, 255, 255);

dataGridView2.Rows[i].Cells[0].Value = i;

// по горизонтали

dataGridView2.Rows[0].Cells[i].Style.ForeColor = Color.FromArgb(255, 255, 255);

dataGridView2.Rows[0].Cells[i].Style.BackColor = Color.FromArgb(110, 110, 110);

dataGridView2.Rows[0].Cells[i].Value = i;

dataGridView2.Columns[i].Width = 50;

}

dataGridView2.Rows[0].Cells[0].Value = "Operation \\ operation";

for (int i = 0; i < convolution.OperationsCount; i++)

{

for (int k = 0; k < convolution.ArrayH[i].Length; k++)

{

if (convolution.ArrayH[i][k] == 1)

{

dataGridView2.Rows[i + 1].Cells[k + 1].Style.BackColor = Color.FromArgb(82, 97, 160);

}

}

}

}

private void ClearView()

{

dataGridView1.ColumnCount = 1;

dataGridView1.RowCount = 1;

dataGridView2.ColumnCount = 1;

dataGridView2.RowCount = 1;

}

private void buttonPrev\_Click(object sender, EventArgs e)

{

int value = 0;

if (!Int32.TryParse(textBox1.Text, out value))

{

value = 0;

}

else

{

value--;

}

if (value == 0)

{

buttonPrev.Enabled = false;

}

else

{

buttonPrev.Enabled = true;

}

textBox1.Text = value.ToString();

label3.Text = "Граф на шаге " + value + ":";

}

private void buttonNext\_Click(object sender, EventArgs e)

{

int value = 0;

if (!Int32.TryParse(textBox1.Text, out value))

{

value = 0;

}

else

{

value++;

}

if (value == convolution.StepList.Count)

{

buttonNext.Enabled = false;

}

else

{

buttonNext.Enabled = true;

}

textBox1.Text = value.ToString();

label3.Text = "Граф на шаге " + value + ":";

}

private void textBox1\_TextChanged(object sender, EventArgs e)

{

int value = 0;

if (!Int32.TryParse(textBox1.Text, out value))

{

value = 0;

}

if (value == 0)

{

buttonPrev.Enabled = false;

}

else

{

buttonPrev.Enabled = true;

}

if (value == convolution.StepList.Count - 1)

{

buttonNext.Enabled = false;

}

else

{

buttonNext.Enabled = true;

}

pictureBox1.Refresh();

}

private void pictureBox1\_Paint(object sender, PaintEventArgs e)

{

int index;

if(Int32.TryParse(textBox1.Text, out index))

{

DrawGraph dgGraph = new DrawGraph((PictureBox) sender,

convolution.StepList[index], e.Graphics);

dgGraph.Draw();

}

}

}

}

**Operation.cs**

using System.Collections.Generic;

namespace L5\_RiPOD

{

struct Operation

{

public string Value { get; set; }

public int TimeCalculation { get; set; }

public string StateBlock { get; set; }

public List<int> VectorList;

public List<string> ConnectionList;

}

}

**ConvolutionGraph.cs**

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

namespace L5\_RiPOD

{

class ConvolutionGraph

{

public int OperationsCount;

public int TypesCount;

public int[][] ArrayTypes;

private int[] ProcessorsByTypesCount;

private int[] TimeCalculationByTypes;

public int[][] ArrayH;

public List<List<int>> List\_chains = new List<List<int>>();

public List<List<Operation>> StepList = new List<List<Operation>>();

public List<string> SeqParFunc = new List<string>();

public ConvolutionGraph()

{

OperationsCount = 0;

TypesCount = 0;

ArrayH = null;

ArrayTypes = null;

ProcessorsByTypesCount = null;

}

public void File\_Load(string filestr)

{

// получение данных из файла

StreamReader file = new StreamReader(filestr);

//step 1 чтение кол-ва операций

string buff = "";

buff = file.ReadLine();

buff = Find\_value(buff);

OperationsCount = Convert.ToInt32(buff);

//step 2 чтение кол-ва типов операций

buff = "";

buff = file.ReadLine();

buff = Find\_value(buff);

TypesCount = Convert.ToInt32(buff);

//step 3 чтение операций каждого типа

ArrayTypes = new int[TypesCount][];

for (int i = 0; i < TypesCount; i++)

{

buff = "";

buff = file.ReadLine();

buff = Find\_value(buff);

ArrayTypes[i] = GetArray(buff);

}

//step 4 чтение количества процессоров каждого типа

ProcessorsByTypesCount = new int[TypesCount];

for (int i = 0; i < TypesCount; i++)

{

buff = "";

buff = file.ReadLine();

buff = Find\_value(buff);

ProcessorsByTypesCount[i] = Convert.ToInt32(buff);

}

//step 5 чтение времени выполнения операции каждого типа

TimeCalculationByTypes = new int[TypesCount];

for (int i = 0; i < TypesCount; i++)

{

buff = "";

buff = file.ReadLine();

buff = Find\_value(buff);

TimeCalculationByTypes[i] = Convert.ToInt32(buff);

}

//step 6 чтение таблицы смежности

buff = "";

buff = file.ReadLine();

ArrayH = new int[OperationsCount][];

for (int i = 0; i < OperationsCount; i++)

{

buff = "";

buff = file.ReadLine();

ArrayH[i] = GetArray(buff);

}

file.Close();

}

public void Planning()

{

GetChains();

BeginnerGraph();

Convolution();

}

#region Получение цепочек

private void GetChains()

{

// построить цепи зависимостей

List<int> listUse = new List<int>();

List<int> chain = new List<int>();

int operation = 0;

while (listUse.Count < OperationsCount)

{

operation = 0;

while (listUse.Contains(operation))

{

operation++;

}

chain.Clear();

chain.Add(operation);

listUse.Add(operation);

while (true)

{

// найти в строке таблицы смежности следующее направление

bool index = false;

for (int i = 0; i < ArrayH[operation].Length; i++)

{

if (ArrayH[operation][i] == 1)

{

index = true;

operation = i;

break;

}

}

if (index) // если найдено следующее значение в цепи

{

chain.Add(operation);

if (!listUse.Contains(operation))

{

listUse.Add(operation);

}

}

else

{

break;

}

}

List\_chains.Add(new List<int>(chain));

}

}

private string Find\_value(string s)

{

int index = s.IndexOf(':') + 2;

return s.Substring(index); // извлечение подстроки с указанной позиции и до конца

}

private int[] GetArray(string s)

{

string[] nums = s.Split(' ');

int[] arr = new int[nums.Length];

for (int i = 0; i < nums.Length; i++)

{

arr[i] = int.Parse(nums[i]);

}

return arr;

}

#endregion

#region Начальный граф

// Постройка начального графа

private void BeginnerGraph()

{

List<Operation> operations = new List<Operation>();

for (int i = 0; i < OperationsCount; i++)

{

Operation op = new Operation();

int type = FindType(i + 1);

op.Value = (i + 1).ToString();

op.StateBlock = op.Value.ToString();

op.TimeCalculation = TimeCalculationByTypes[type];

op.VectorList = new List<int>();

op.ConnectionList = new List<string>();

for (int k = 0; k < TypesCount; k++)

{

op.VectorList.Add(k == type ? 1 : 0);

}

List<int> list = FindChains(i);

foreach (var chain in List\_chains)

{

foreach (var t in chain)

{

if (!list.Contains(t) && !op.ConnectionList.Contains((t + 1).ToString()))

{

op.ConnectionList.Add((t + 1).ToString());

}

}

}

op.ConnectionList.Sort();

operations.Add(op);

}

StepList.Add(operations);

}

// Поиск типа операции по ее значению

private int FindType(int value)

{

int res = -1;

for (int i = 0; i < TypesCount; i++)

{

if (ArrayTypes[i].Contains(value))

{

res = i;

break;

}

}

return res;

}

// Поиск всех цепочек, в которых присутствует операция с указанным значением

private List<int> FindChains(int value)

{

List<int> list = new List<int>();

foreach (var t in List\_chains)

{

if (t.Contains(value))

{

list.AddRange(t);

}

}

return list;

}

#endregion

#region Свертывание графа

// Свертывание

private void Convolution()

{

int step = 0;

List<Operation> opList = FullCloneObject(StepList[step]);

while (FindAnyConnections(opList) != 0) // пока не останется 1 операция в графе

{

step++;

opList = FullCloneObject(StepList[step - 1]); // предыдущее значение графа

int countMissedConnections = 0; // количество связей, разрешенных потерять

try

{

for (;;)

{

for (int i = 0; i < opList.Count; i++)

{

for (int k = 0; k < opList.Count - 1; k++)

{

if (i == k)

continue;

List<string> list = FindCommonConnections(opList[i], opList[k]);

int connect = Convert.ToInt32(HaveConnection(opList[i], opList[k]));

if (

(Math.Abs(Math.Max(opList[i].ConnectionList.Count, opList[k].ConnectionList.Count) -

(list.Count + connect))) > countMissedConnections)

{

continue;

}

else

{

opList = ConnectOperations(opList, i, k);

throw new Exception("Выход из 3 for-ов");

}

}

}

countMissedConnections++;

if (countMissedConnections > OperationsCount)

break;

}

}

catch (Exception e)

{

}

finally

{

// выделение памяти и созранение в StepList

List<Operation> memList = new List<Operation>();

memList = FullCloneObject(opList);

StepList.Add(memList);

}

}

// Соединить все оставшиеся вершины последовательно

while (opList.Count > 1)

{

opList = ConnectOperations(opList, 0, 1);

List<Operation> memList = new List<Operation>();

memList = opList.ToList();

StepList.Add(memList);

}

}

// глубокое копирование

private List<Operation> FullCloneObject(List<Operation> opList)

{

List<Operation> list = new List<Operation>();

foreach (var t in opList)

{

Operation aOperation = new Operation();

aOperation.Value = t.Value;

aOperation.StateBlock = t.StateBlock;

aOperation.TimeCalculation = t.TimeCalculation;

aOperation.ConnectionList = new List<string>();

foreach (var val in t.ConnectionList)

{

aOperation.ConnectionList.Add(val);

}

aOperation.VectorList = new List<int>();

foreach (var val in t.VectorList)

{

aOperation.VectorList.Add(val);

}

list.Add(aOperation);

}

return list;

}

// проверка на наличие связи между операциями

private bool HaveConnection(Operation op1, Operation op2)

{

if (op1.ConnectionList.Contains(op2.Value))

return true;

return false;

}

// поиск всех связей в графе

private int FindAnyConnections(List<Operation> opList)

{

int count = 0;

foreach (var op in opList)

{

count += op.ConnectionList.Count;

}

count = count / 2;

return count;

}

// поиск общих связей

private List<string> FindCommonConnections(Operation l1, Operation l2)

{

List<string> commonOperations = new List<string>();

foreach (var op in l1.ConnectionList)

{

if (op != l2.Value) // исключить проверку возможной связи между операциями

{

if (l2.ConnectionList.Contains(op))

{

commonOperations.Add(op);

}

}

}

return commonOperations;

}

// соединение операций в одну

private List<Operation> ConnectOperations(List<Operation> opList, int indexOp1, int indexOp2)

{

opList = ChangeConnectionsWithOperation(opList, opList[indexOp1], opList[indexOp2]);

opList = CalculateNewResources(opList, opList[indexOp1], opList[indexOp2]);

opList = DeleteConnectionsWithOperation(opList, opList[indexOp2]);

opList = DeleteAllInvalidConnections(opList);

return opList;

}

// удаление операции из графа со всеми связями

private List<Operation> DeleteConnectionsWithOperation(List<Operation> opList, Operation operation)

{

foreach (var op in opList)

{

if (op.Value != operation.Value)

{

op.ConnectionList.Remove(operation.Value);

}

}

opList.Remove(operation);

return opList;

}

// изменение соединений в графе в результате соединения операций

private List<Operation> ChangeConnectionsWithOperation(List<Operation> opList, Operation op1, Operation op2)

{

for (var i = 0; i < opList.Count; i++)

{

var op = opList[i];

if (op.Value != op1.Value && op.ConnectionList.Contains(op1.Value))

{

int index = opList[i].ConnectionList.IndexOf(op1.Value);

opList[i].ConnectionList[index] = op1.Value + op2.Value;

}

}

return opList;

}

// вычисление новых параметров вершины графа

private List<Operation> CalculateNewResources(List<Operation> opList, Operation op1, Operation op2)

{

string planFunc;

if (HaveConnection(op1, op2))

{

planFunc = "par";

}

else

{

planFunc = "seq";

}

int index = opList.IndexOf(op1);

op1 = WriteSeqParFunc(planFunc, op1, op2);

opList[index] = op1;

for (int i = 0; i < opList.Count; i++)

{

if (opList[i].Value == op1.Value)

{

var y = opList[i];

y.Value = op1.Value + op2.Value;

switch (planFunc)

{

case "par":

{

y.TimeCalculation = Math.Max(op1.TimeCalculation, op2.TimeCalculation);

y.VectorList = VectorSum(op1.VectorList, op2.VectorList);

break;

}

case "seq":

{

y.TimeCalculation = op1.TimeCalculation + op2.TimeCalculation;

y.VectorList = VectorMax(op1.VectorList, op2.VectorList);

break;

}

}

opList[i] = y;

}

}

return opList;

}

// запись операции в хранилище

private Operation WriteSeqParFunc(string planFunc, Operation op1, Operation op2)

{

// запись функции плана в хранилице

SeqParFunc.Add(planFunc + "(" + op1.StateBlock + ", " + op2.StateBlock + ")");

switch (planFunc)

{

case "seq":

{

op1.StateBlock = op1.StateBlock + "-" + op2.StateBlock;

break;

}

case "par":

{

op1.StateBlock = "[" + op1.StateBlock + "/" + op2.StateBlock + "]";

break;

}

}

return op1;

}

// суммирование векторов

private List<int> VectorSum(List<int> vector1, List<int> vector2)

{

for (int i = 0; i < vector1.Count; i++)

{

vector1[i] = vector1[i] + vector2[i];

}

return vector1;

}

// нахождение вектора максимумов

private List<int> VectorMax(List<int> vector1, List<int> vector2)

{

for (int i = 0; i < vector1.Count; i++)

{

vector1[i] = Math.Max(vector1[i], vector2[i]);

}

return vector1;

}

// сравнение суммы векторов с ограничением на процессоры

private bool Compare(List<int> obj1, List<int> obj2)

{

for (var i = 0; i < obj1.Count; i++)

{

if ((obj1[i] + obj2[i]) > ProcessorsByTypesCount[i])

{

return false;

}

}

return true;

}

// удаление всех связей, которые невозможны из-за ограничения на процессоры

private List<Operation> DeleteAllInvalidConnections(List<Operation> opList)

{

for (var i = 0; i < opList.Count; i++)

{

for (int k = 0; k < opList[i].ConnectionList.Count; k++)

{

int index = -1;

string val = opList[i].ConnectionList[k];

for (int t = 0; t < opList.Count; t++)

{

if (opList[t].Value == opList[i].ConnectionList[k])

{

index = t;

break;

}

}

if (!Compare(opList[i].VectorList, opList[index].VectorList))

{

opList[i].ConnectionList.Remove(val);

opList[index].ConnectionList.Remove(val);

k = -1;

}

}

}

return opList;

}

#endregion

}

}

**DrawGraph.cs**

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Windows.Forms;

namespace L5\_RiPOD

{

class DrawGraph

{

public PictureBox Picture { get; set; }

private Graphics g;

private Pen pen;

private int R;

private Point centerPoint;

private const int smallRadius = 20;

public List<Operation> opList { get; set; }

public DrawGraph(PictureBox pic, List<Operation> opList, Graphics graphics)

{

this.opList = opList;

Picture = pic;

g = graphics;

}

private void CalcCenter()

{

centerPoint.X = Picture.Width / 2;

centerPoint.Y = Picture.Height / 2;

R = (int)Math.Round(2.0 / 3.0 \* Math.Min(centerPoint.X, centerPoint.Y));

}

public void Draw()

{

CalcCenter();

pen = new Pen(Color.Black);

Point oPoint = new Point();

double angle = 360.0 / opList.Count;

// конвертация в радианы

angle = angle \* Math.PI / 180;

for (int i = 0; i < opList.Count; i++)

{

oPoint = GetCenterCircle(centerPoint, i, angle);

DrawOperation(opList[i], oPoint, angle \* i);

DrawLinesOperationConnections(opList, i, oPoint, angle);

}

}

private Point GetCenterCircle(Point centerPoint, int indexOp, double angle)

{

Point oPoint = new Point();

double xAngle = Math.Sin(angle \* indexOp);

double yAngle = Math.Cos(angle \* indexOp);

xAngle \*= R;

yAngle \*= -R;

oPoint.X = (int)Math.Round(xAngle) + centerPoint.X;

oPoint.Y = (int)Math.Round(yAngle) + centerPoint.Y;

return oPoint;

}

private void DrawOperation(Operation op, Point center, double angle)

{

Point upLeftPoint = new Point();

upLeftPoint.X = center.X - smallRadius;

upLeftPoint.Y = center.Y - smallRadius;

Pen pen = new Pen(Color.Black);

Brush b = new SolidBrush(Color.Firebrick);

Font f = new Font("Times New Roman", 12);

g.DrawEllipse(pen, upLeftPoint.X, upLeftPoint.Y, smallRadius \* 2, smallRadius \* 2);

// StateBlock

g.DrawString(op.StateBlock, f, b, center.X - smallRadius/2 - 5\*op.StateBlock.Length/2, center.Y - smallRadius/2);

string s = op.TimeCalculation + " (";

for (int i = 0; i < op.VectorList.Count; i++)

{

if (i == op.VectorList.Count - 1)

{

s = s + op.VectorList[i] + ")";

}

else

{

s = s + op.VectorList[i] + ", ";

}

}

// строка состояния

int a = 5;

int len = 0;

if (angle > Math.PI)

{

len = s.Length;

}

upLeftPoint.X = (int)(Math.Sin(angle) \* (smallRadius + a)) + center.X - 7\* len;

upLeftPoint.Y = -(int)(Math.Cos(angle) \* (smallRadius + 3\*a)) + center.Y;

g.DrawString(s, f, b, upLeftPoint);

// линии

}

private void DrawLinesOperationConnections(List<Operation> opList, int indexOp, Point center, double angle)

{

Random r = new Random(indexOp \* DateTime.Now.Millisecond \* DateTime.Now.Second);

Pen pen = new Pen(Color.FromArgb(r.Next(0, 200), r.Next(0, 200), r.Next(0, 255)));

for (int i = 0; i < opList[indexOp].ConnectionList.Count; i++)

{

Point beginPoint = new Point();

Point endPoint = new Point();

int index = FindIndex(opList, opList[indexOp].ConnectionList[i]);

beginPoint = center;

endPoint = GetCenterCircle(centerPoint, index, angle);

g.DrawLine(pen, beginPoint, endPoint);

}

}

private int FindIndex(List<Operation> opList, string connection)

{

for (int i = 0; i < opList.Count; i++)

{

if (opList[i].Value == connection)

return i;

}

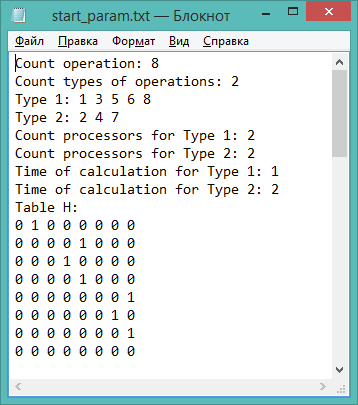
return -1;

}

}

}

### Скриншоты результатов



## Вывод

Изучен алгоритм свертывания графа распараллеленности операций.