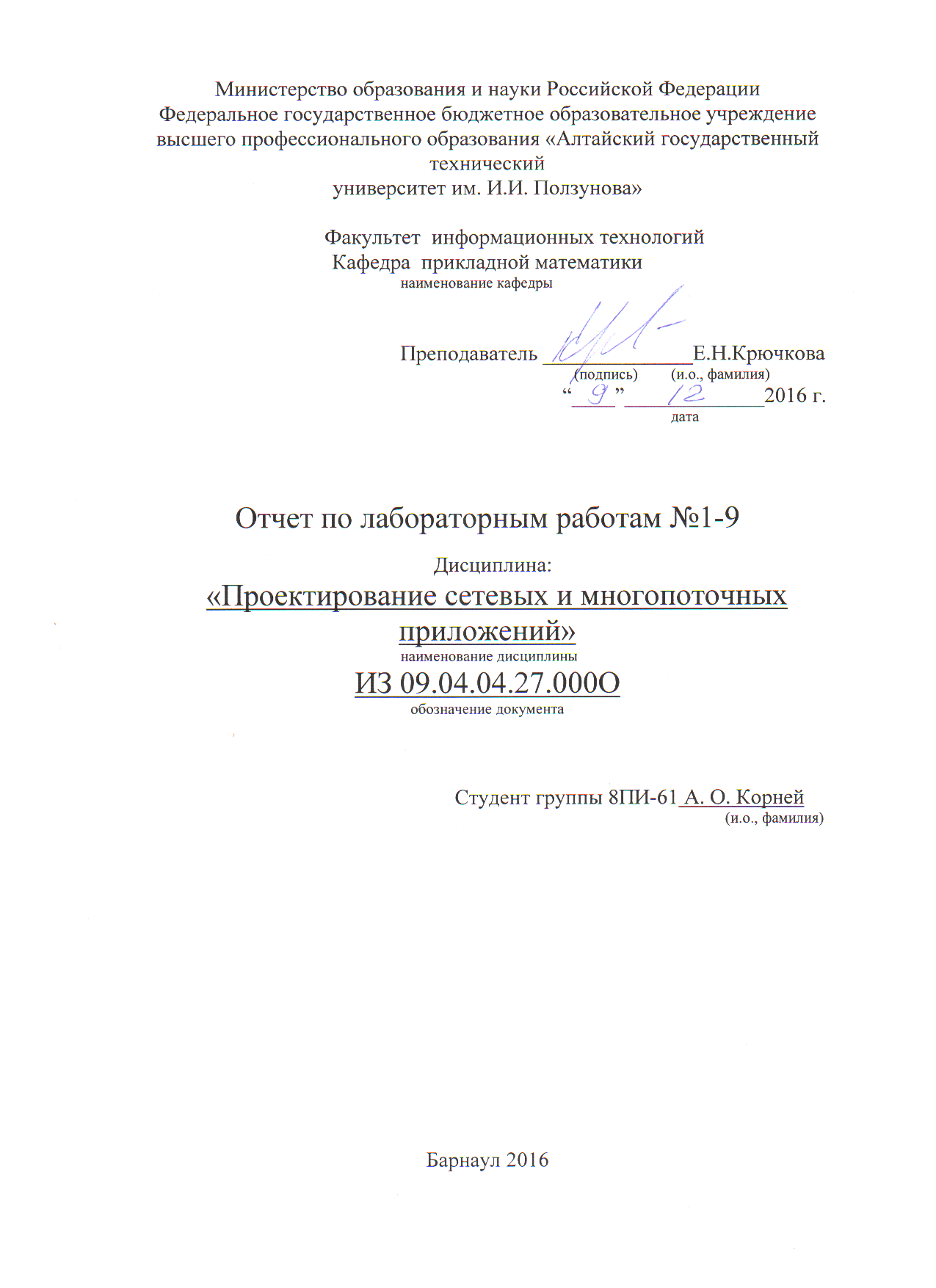
****

# Лабораторная работа №2

## Параллельные вычисления с использованием OpenMP

### Задание:

Реализовать параллельный генетический алгоритм для решения задачи оптимального разбиения графа на K подграфов. Параллельные вычисления реализовать средствами OpenMP

### Решение

Пусть дан взвешенный полносвязный неориентированный граф G=(V,E), каждой вершине v, принадлежащей V, и каждому ребру e, принадлежащему E, которого приписан вес. Задача оптимального разделения графа состоит в разбиении его вершин на непересекающиеся подмножества с максимально близкими суммарными весами вершин и минимальным суммарным весом ребер, проходящих между полученными подмножествами вершин.

Для решения данной задачи при помощи генетического алгоритма необходимо описать организм, а так же процессы мутации и скрещивания и способ оценки приспособленности

**Организм**

В задачах разбиения графов, как правило, в качестве организма выбирается массив чисел, размер которого совпадает с числом вершин в графе, а значение j-ой позиции обозначает номер подграфа, к которому относится данная вершина.

**Мутация**

Процесс мутации опишем как случайное изменение каждого значения в массиве на новое, таким образом получается абсолютно новое разбиение.

**Скрещивание**

Скрещивание будем производить классическим способом обмена генов

**Оценка приспособленности**

При оценке качества разбиения начисляются следующие штрафы:

* отклонение суммарного веса вершин подграфа от среднего
* вес очередного граничного ребра

**Алгоритм**

Для решения задачи была выбрана островная модель генетического алгоритма. Она характеризуется наличием нескольких отдельных популяций – островов. Эти популяции развиваются независимо и только изредка происходит обмен представителями между популяциями. Данная модель генетического алгоритма обладает следующими свойствами:

* Наличие нескольких популяций, как правило одинакового фиксированного размера.
* Фиксированная разрядность генов.
* Любые комбинации стратегий отбора и формирования следующего поколения в каждой популяции. Можно сделать так, что в разных популяциях будут использоваться разные комбинации стратегий, хотя даже один вариант дает разнообразные решения на различных "островах".
* Ограничений на тип кроссовера и мутации нет.
* Случайный обмен особями между "островами". Если миграция будет слишком активной, то особенности островной модели будут сглажены и она будет не очень сильно отличаться от моделей ГА без параллелизма.

В данной работе задача решалась классическим генетическим алгоритмом, а так же островным вариантом (5 островов, обмен генами дважды за эволюционный процесс, внедрение трех лучших особей во все острова случайным образом)

При одинаковом суммарном размере популяции и одинаковом числе поколений на островной модели достигается лучшее решение, чем в классическом варианте. Оценка приспособленности в среднем выше на 15-18%.

**Код программы:**

#pragma once

#include <stdlib.h>

#include <set>

#include <random>

#include <limits.h>

#include "Graph.h"

struct Organizm

{

int chromoSize;

int \*chromo;

double fitness;

Organizm(int size, int K)

{

chromoSize = size;

chromo = new int[size];

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

{

chromo[i] = rand() % K;

}

fitness = 0;

}

void Print()

{

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

{

//cout << chromo[i];

//cout << " ";

}

//cout << endl;

}

bool operator<(Organizm r)

{

return fitness < r.fitness;

}

Organizm\* Clone()

{

Organizm\* clone = new Organizm(chromoSize);

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

{

clone->chromo[i] = chromo[i];

}

clone->fitness = fitness;

return clone;

}

~Organizm()

{

//delete [] chromo;

}

private:

Organizm(int size)

{

chromoSize = size;

chromo = new int[size];

fitness = 0;

}

};

class GeneticTaskSolver

{

int \_crossoverCount,

\_populationSize,

\_chromosomeVariability; // subgraphs count

double \_p\_m, // mutation probability

\_p\_vm, // probability of after-mutation selection for better

\_p\_v;

vector<Organizm\*> \_population;

vector<Organizm\*>::iterator \_champion;

vector<Organizm\*>::iterator \_outsider;

Graph \*\_target;

void RunCrossover();

void RunMutation();

void RunSelection();

void CollectStats();

public:

GeneticTaskSolver(Graph \*target, int populationSize, int crossoverCount, int chromosomeVariability);

void GeneratePopulation();

void RunEvolution(int generationCount, double pm, double pvm, double pv);

Organizm\* GetBest();

void Inject(Organizm \*candidate);

~GeneticTaskSolver();

};

#pragma once

#include <stdlib.h>

#include <iostream>

#include <map>

#include "Vertex.h"

using namespace std;

class Graph

{

double \_summaryVertexWeight;

map<int, Vertex\*> \_verticesList;

public:

void AddVertex(int number, double weight);

void AddEdge(int num\_from, int num\_to, double weight);

double EvaluatePartitioning(int\* partitionDescription, int partsCount);

int GetVerticesCount();

void GenerateRandom(int verticiesCount, double density);

Graph();

~Graph();

};

#pragma once

#include <stdlib.h>

#include <iostream>

#include <map>

using namespace std;

class Vertex

{

int \_number;

double \_weight;

map<int, double> \_edges;

public:

Vertex(int number, double weight);

int GetNumber();

double GetWeight();

double GetEdgeWeight(int to);

const map<int, double>& GetEdges() const;

void AddEdge(Vertex \*to, double weight);

bool operator<(const Vertex & op\_right) const;

};

#include "stdafx.h"

#include "GeneticTaskSolver.h"

GeneticTaskSolver::GeneticTaskSolver(Graph \* target, int populationSize, int crossoverCount, int chromosomeVariability)

{

\_target = target;

\_populationSize = populationSize;

\_crossoverCount = crossoverCount;

\_chromosomeVariability = chromosomeVariability;

}

void GeneticTaskSolver::GeneratePopulation()

{

int graphSize = \_target->GetVerticesCount();

for (int i = 0; i < \_populationSize; i++)

{

Organizm \*org = new Organizm(graphSize, \_chromosomeVariability);

for (int chrom = 0; chrom < graphSize; chrom++)

{

org->chromo[chrom] = rand() % \_chromosomeVariability;

}

\_population.push\_back(org);

}

}

void GeneticTaskSolver::RunEvolution(int generationCount, double pm, double pvm, double pv)

{

\_p\_m = pm;

\_p\_v = pv;

\_p\_vm = pvm;

for (int g = 0; g < generationCount; g++)

{

RunCrossover();

RunMutation();

RunSelection();

CollectStats();

}

}

Organizm \* GeneticTaskSolver::GetBest()

{

return \*\_champion;

}

void GeneticTaskSolver::Inject(Organizm \* candidate)

{

//if (candidate->fitness > (\*\_champion)->fitness)

//{

auto outs = \*\_outsider;

\_population.erase(\_outsider);

\_population.push\_back(candidate->Clone());

//}

delete outs;

}

void GeneticTaskSolver::CollectStats()

{

double fitMax = -std::numeric\_limits<double>::max();

double fitMin = std::numeric\_limits<double>::max();

for (vector<Organizm\*>::iterator i = \_population.begin();

i != \_population.end();

i++)

{

if ((\*i)->fitness > fitMax)

{

fitMax = (\*i)->fitness;

\_champion = i;

}

if ((\*i)->fitness < fitMin)

{

fitMin = (\*i)->fitness;

\_outsider = i;

}

}

}

void GeneticTaskSolver::RunCrossover()

{

bool \*crossoverFlags = new bool[\_populationSize];

int parentA, parentB;

for (int cross = 0; cross < \_crossoverCount; cross++)

{

do

{

parentA = rand() % \_populationSize;

} while (!crossoverFlags[parentA]);

crossoverFlags[parentA] = false;

do

{

parentB = rand() % \_populationSize;

} while (!crossoverFlags[parentB]);

crossoverFlags[parentB] = false;

Organizm \*childA, \*childB;

childA = \_population[parentA]->Clone();

childB = \_population[parentB]->Clone();

//cout << "Parent A\n";

\_population[parentA]->Print();

//cout << "Parent B\n";

\_population[parentB]->Print();

int crossoverPosition = rand() % (childA->chromoSize - 2) + 1;

for (int i = crossoverPosition; i < childA->chromoSize; i++)

{

int buffer = childA->chromo[i];

childA->chromo[i] = childB->chromo[i];

childB->chromo[i] = buffer;

}

\_population.push\_back(childA);

\_population.push\_back(childB);

//cout << "Child A\n";

childA->Print();

//cout << "Child B\n";

childB->Print();

}

}

void GeneticTaskSolver::RunMutation()

{

for (vector<Organizm\*>::iterator pop\_iter = \_population.begin();

pop\_iter != \_population.end();

pop\_iter++)

{

double f = (double)rand() / RAND\_MAX;

if (f < \_p\_m)

{

//cout << "Try to mutate:\n";

Organizm\* clone = (\*pop\_iter)->Clone();

clone->Print();

for (int i = 0; i < clone->chromoSize; i++)

{

int oldValue = clone->chromo[i];

int newValue;

do

{

newValue = rand() % \_chromosomeVariability;

} while (newValue == oldValue);

clone->chromo[i] = newValue;

//int buffer = clone->chromo[i];

//clone->chromo[i] = clone->chromo[clone->chromoSize - i - 1];

//clone->chromo[clone->chromoSize - i - 1] = buffer;

}

(\*pop\_iter)->fitness = \_target->EvaluatePartitioning((\*pop\_iter)->chromo, \_chromosomeVariability);

clone->fitness = \_target->EvaluatePartitioning(clone->chromo, \_chromosomeVariability);

Organizm \*best = (\*pop\_iter)->fitness > clone->fitness

? \*pop\_iter

: clone;

Organizm \*worse = (\*pop\_iter)->fitness < clone->fitness

? \*pop\_iter

: clone;

f = (double)rand() / RAND\_MAX;

//cout << "New:\n";

clone->Print();

if (f < \_p\_vm)

{

//cout << "Best chosen\n";

\*pop\_iter = best;

delete worse;

}

else

{

//cout << "Worse chosen\n";

\*pop\_iter = worse;

delete best;

}

}

}

}

void GeneticTaskSolver::RunSelection()

{

for (vector<Organizm\*>::iterator i = \_population.begin();

i != \_population.end();

i++)

{

(\*i)->fitness = \_target->EvaluatePartitioning((\*i)->chromo, \_chromosomeVariability);

}

set<Organizm\*> done;

int selectionCount = \_crossoverCount \* 2;

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

{

Organizm\* forceA, \*forceB;

int indexA;

do

{

indexA = rand() % \_population.size();

forceA = \_population[indexA];

} while (done.find(forceA) != done.end());

done.insert(forceA);

int indexB;

do

{

indexB = rand() % \_population.size();

forceB = \_population[indexB];

} while (indexB == indexA

|| done.find(forceB) != done.end());

done.insert(forceB);

//forceA->fitness = \_target->EvaluatePartitioning(forceA->chromo, \_chromosomeVariability);

//forceB->fitness = \_target->EvaluatePartitioning(forceB->chromo, \_chromosomeVariability);

double f = (double)rand() / RAND\_MAX;

int best = forceA->fitness > forceB->fitness

? indexA

: indexB;

int worse = forceA->fitness < forceB->fitness

? indexA

: indexB;

if (f < \_p\_v)

{

Organizm \*o = \_population[worse];

\_population.erase(\_population.begin() + worse);

delete o;

}

else

{

Organizm \*o = \_population[best];

\_population.erase(\_population.begin() + best);

delete o;

}

}

done.clear();

}

GeneticTaskSolver::~GeneticTaskSolver()

{

}

#include "stdafx.h"

#include "Graph.h"

void Graph::AddVertex(int number, double weight)

{

if (\_verticesList.find(number) != \_verticesList.end())

{

//cout << "Sorry, vertex was already added";

return;

}

Vertex\* vertex = new Vertex(number, weight);

\_summaryVertexWeight += weight;

\_verticesList[number] = vertex;

}

void Graph::AddEdge(int num\_from, int num\_to, double weight)

{

\_verticesList[num\_from]->AddEdge(\_verticesList[num\_to], weight);

\_verticesList[num\_to]->AddEdge(\_verticesList[num\_from], weight);

}

double Graph::EvaluatePartitioning(int \* partitionDescription, int partsCount)

{

double \* partWeights = new double[partsCount];

double averageSubWeight = \_summaryVertexWeight / partsCount;

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

{

partWeights[i] = 0.0;

}

int vc = \_verticesList.size();

//partWeights contains calculated weights of subgraphs

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

{

partWeights[partitionDescription[i]] +=

\_verticesList.at(i)->GetWeight();

}

//calculating penalty for vertex weights of subraphs

double sqSum = 0.0;

//double sumDev = 0.0;

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

{

//sumDev += abs(partWeights[i] - averageSubWeight );

//for (int j = i + 1; j < partsCount; j++)

//{

sqSum += pow((partWeights[i] - averageSubWeight), 2);

//}

}

//calculating penalty for border edges weight

double borderWeight = 0;

for (map<int, Vertex\*>::iterator itr = \_verticesList.begin();

itr != \_verticesList.end();

itr++)

{

Vertex\* vertexFrom = itr->second;

map<int, double> edges = vertexFrom->GetEdges();

for (map<int, double>::iterator edgesItr = edges.begin();

edgesItr != edges.end();

edgesItr++)

{

int to = edgesItr->first;

double weight = edgesItr->second;

if (partitionDescription[vertexFrom->GetNumber()]

!= partitionDescription[to])

{

borderWeight += weight / 2;

}

}

}

double fitness = -sqSum - borderWeight;

//double fitness = -sumDev - log10(borderWeight);

delete[] partWeights;

return fitness;

}

int Graph::GetVerticesCount()

{

return \_verticesList.size();

}

void Graph::GenerateRandom(int verticiesCount, double density)

{

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

{

double w = 30 \* (double)rand() / RAND\_MAX;

this->AddVertex(i, w);

}

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

{

for (int j = i+1; j < verticiesCount; j++)

{

double needToAdd = (double)rand() / RAND\_MAX;

if (needToAdd < density)

{

this->AddEdge(i, j, 3 \* (double)rand() / RAND\_MAX);

}

}

}

}

Graph::Graph()

{

\_summaryVertexWeight = 0;

}

Graph::~Graph()

{

for (map<int, Vertex\*>::iterator i = \_verticesList.begin();

i != \_verticesList.end();

i++)

{

delete i->second;

}

}

#include "stdafx.h"

#include "Vertex.h"

Vertex::Vertex(int number, double weight)

{

\_number = number;

\_weight = weight;

}

int Vertex::GetNumber()

{

return \_number;

}

double Vertex::GetWeight()

{

return \_weight;

}

double Vertex::GetEdgeWeight(int to)

{

if (\_edges.find(to) == \_edges.end())

{

return 0;

}

return \_edges[to];

}

const map<int, double>& Vertex::GetEdges() const

{

return \_edges;

}

void Vertex::AddEdge(Vertex \* to, double weight)

{

\_edges[to->GetNumber()] = weight;

}

bool Vertex::operator<(const Vertex & op\_right) const

{

return this->\_number < op\_right.\_number;

}

// GAOpenMP.cpp : Defines the entry point for the console application.

//

#include "stdafx.h"

#include <algorithm>

#include <omp.h>

#include <time.h>

#include <stdlib.h>

#include <Windows.h>

#include "Graph.h"

#include "GeneticTaskSolver.h"

bool LessFitted(Organizm\* left, Organizm \*right)

{

return left->fitness > right->fitness;

}

int main()

{

#pragma region task parameters

int solversCount = 5;

int graphSize = 150;

double graphD = 0.3;

int fullPopulationSize = 100;

int fullCrossoverCount = 40;

int subgraphsCount = 15;

double pm = 0.04,

pvm = 0.99,

pv = 0.95;

int fullGenCount = 15;

int epochCount = 3;

int topCount = 3;

#pragma endregion

Graph\* gr = new Graph();

gr->GenerateRandom(graphSize, graphD);

#pragma region Single

GeneticTaskSolver\* solver = new GeneticTaskSolver(gr, fullPopulationSize, fullGenCount, subgraphsCount);

solver->GeneratePopulation();

solver->RunEvolution(fullGenCount, pm, pvm, pv);

cout << "single thread best = " << solver->GetBest()->fitness << endl;

#pragma endregion

#pragma region Parallel

srand(time(nullptr));

vector<Organizm\*> champs;

vector<GeneticTaskSolver\*> solvers;

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

{

solvers.push\_back(new GeneticTaskSolver(gr,

fullPopulationSize/solversCount,

fullCrossoverCount/solversCount,

subgraphsCount));

}

omp\_set\_num\_threads(solversCount);

#pragma omp parallel for

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

{

srand(int(time(NULL)) ^ omp\_get\_thread\_num());

solvers[i]->GeneratePopulation();

}

for (int g = 0; g < epochCount; g++)

{

//run partial evo

#pragma omp parallel for

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

{

srand(int(time(NULL)) ^ omp\_get\_thread\_num());

solvers[i]->RunEvolution(fullGenCount/epochCount, pm, pvm, pv);

}

//get best

champs.clear();

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

{

champs.push\_back(solvers[i]->GetBest()->Clone());

}

std::sort(champs.begin(), champs.end(), LessFitted);

//inject best

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

{

if (champs[i % topCount] != solvers[i]->GetBest())

{

solvers[i]->Inject(champs[i % topCount]);

}

else

{

solvers[i]->Inject(champs[(i + 1) % topCount]);

}

}

}

cout <<"openmp best = " << champs[0]->fitness;

#pragma endregion

return 0;

}