Министерство образования и науки Российской Федерации
Федеральное государственное автономное
образовательное учреждение высшего образования
**«Национальный исследовательский Нижегородский
государственный университет им. Н.И. Лобачевского» (ННГУ)**

Институт информационных технологий, математики и механики

Кафедра математического обеспечения и суперкомпьютерных технологий
Прикладная математика и информатика

Задача оптимального разделения графов. Деление с учетом связности.

Выполнил:

Лалыкин О.В.

Нижний Новгород
2017 г.

Содержание

Введение

Математические модели в виде графов широко используются при моделировании разнообразных явлений, процессов и систем. Как результат, многие теоретические и реальные прикладные задачи могут быть решены при помощи тех или иных процедур анализа графовых моделей. Среди множества этих процедур может быть выделен некоторый определенный набор типовых алгоритмов обработки графов.

Граф является основным объектом изучения математической теории графов. Он является совокупностью непустого множества вершин и связей между вершинами. Объекты представляются как вершины, или узлы графа, а связи — как дуги, или рёбра. Многие структуры, представляющие практический интерес в математике и информатике, могут быть представлены графами.

Существуют задачи обработки данных, в которых области расчетов аппроксимируются двумерными или трехмерными вычислительными сетками. Получение результатов в таких задачах сводится, как правило, к выполнению тех или иных процедур обработки для каждого элемента (узла) сети. При этом в ходе вычислений между соседними элементами сети может происходить передача результатов обработки. Эффективное решение таких задач на многопроцессорных системах с распределенной памятью предполагает разделение сети между процессорами таким образом, чтобы каждому из процессоров выделялось примерно равное число элементов сети, а межпроцессорные коммуникации, необходимые для выполнения информационного обмена между соседними элементами, были минимальными.

Такие задачи разделения сети между процессорами могут быть сведены к проблеме оптимального разделения графа.

Задача оптимального разделения графов сама может являться предметом распараллеливания. Это бывает необходимо в тех случаях, когда вычислительной мощности и объема оперативной памяти обычных компьютеров недостаточно для эффективного решения задачи.

В рамках лабораторной работы реализован комбинаторный метод разделения графа – деление с учетом связности с применением API MPI. Проведены эксперименты для сравнения их быстродействия, а также для анализа масштабируемости.

Постановка задачи

Требования

В программе должна быть организована работа алгоритма – деление графа с учетом связности.

Реализация должна содержать:

1. Последовательную версию алгоритма.

2. Параллельную версию алгоритма.

В программе необходимо:

1. Выполнить проверку совпадения результатов последовательной и параллельной реализаций.

2. Продемонстрировать корректность работы алгоритмов на задаче/задачах малой размерности.

3. Обеспечить генерацию данных для задач произвольной размерности.

4. Вывести время работы последовательного и параллельного алгоритмов. Ускорение параллельного алгоритма.

Постановка задачи оптимального разделения графов [1]

Пусть дан взвешенный неориентированный граф $G = (V, E)$, каждой вершине $v \in V$ и каждому ребру $l \in E$ 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.

[1] Источник – Литература[1]

Метод решения

Способ задания входных данных

Обрабатываемый граф задается симметричной матрицей смежности. Элементами матрицы смежности являются признаки наличия дуг между вершинами. Если две вершины являются смежными, то соответствующее им значение в матрице равно единице. Если вершины не соединены дугой, то признак её отсутствия – нуль.

Пример плотного графа с 14-ю вершинами:

Рисунок 1. Матрица смежности 14x14

Описание алгоритма [2]

При разделении графа информационная зависимость между разделенными подграфами будет меньше, если смежные вершины будут находиться в одном подграфе. Алгоритм деления графов с учетом связности пытается достичь этого, последовательно добавляя к формируемому подграфу соседей. На каждой итерации алгоритма происходит разделение графа на 2 части. Таким образом, разделение графа на требуемое число частей достигается путем рекурсивного применения алгоритма.

Общая схема алгоритма деления графов с учетом связности:

1. Выбирается начальная вершина, ей присваивается номер 0.
2. Все связанные с ней вершины нумеруются единицей.
3. Все связанные с предыдущими и непронумерованные до сих пор вершины нумеруются следующим по порядку номером.
4. Процесс продолжается до тех пор, пока не будут пронумерованы все вершины.
5. Граф делится на n частей в порядке нумерации.

Для улучшения качества разбиения в качестве начальной нужно выбирать граничную вершину - одну из двух вершин, максимально удаленных друг от друга.

Поиск оптимальной начальной вершины:

1. Выбирается случайная вершина, которой присваивается нулевой номер.
2. Остальные вершины нумеруются в соответствии с алгоритмом деления с учетом связности.
3. В качестве начальной берется вершина с наибольшим номером, она будет являться граничной.

[2] Источник – Литература[2]

Алгоритм деления с учетом связности гарантирует, что все вершины одного подграфа будут связаны между собой.

В результате, после одной итерации может быть получено следующее разбиение:

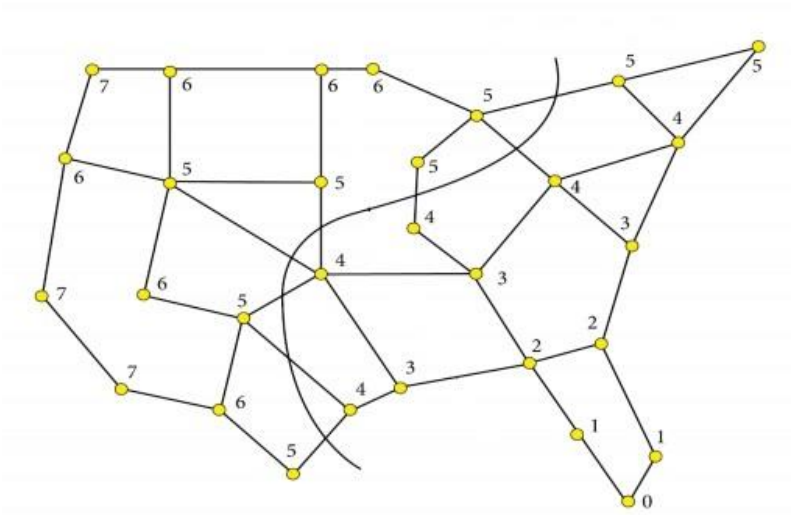

Рисунок 2. Пример разделения графа

Схема распараллеливания

Распределение задач по процессам происходит по аналогии с работой функции MPI_Bcast:

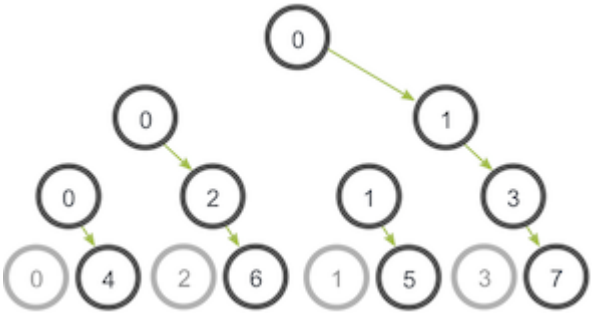

Рисунок 3. Дерево распределения MPI_Bcast

Для любого активного процесса справедлив порядок действий:

1. Процесс, получивший матрицу смежности, производит одну итерацию алгоритма деления.
2. Отправляет одну из двух полученных частей исходного графа другому незанятому процессу.
3. Сохраняет вторую часть разделения и повторяет действия сначала уже над ней.

Использование сети удваивается на каждом последующем этапе древовидной коммуникации, пока все процессы не получат данные.

В данной реализации процесс распределения начинается с нулевого процесса.

Когда распределение данных по всем процессам будет завершено, тогда будут производиться деления всех частей исходного графа параллельно.

Руководство пользователя

Запуск программы осуществляется вводом следующих параметров:

- Количество процессов
- Количество вершин исходного графа
- Количество получаемых разбиений
- Плотность графа

Соответственно, должна быть введена команда вида:
<Директория> mpiexec -n <Количество процессов> G.exe <Количество вершин> <Количество разбиений> <% дуг в графе>

В результате работы программы будет выведены следующие показатели:

- Номер процесса и соответствующий ему размер матрицы после полного распределения, а также количество разбиений, которые он получит после самостоятельного выполнения алгоритма.
- Размер исходной матрицы смежности
- Общее число разбиений
- Количество используемых процессов
- Плотность графа
- Время последовательной обработки
- Время параллельной обработки

```
C:\Users\Oleg\Desktop\LAB3PP\Graph1\G\Release>mpiexec -n 4 G 9015 255
Procrank = 3 Size = 2226 -> 63 matrices
Procrank = 2 Size = 2263 -> 64 matrices
Procrank = 1 Size = 2263 -> 64 matrices
Procrank = 0 Size = 2263 -> 64 matrices

Size of adjacency matrix: 9015
Number of dividual matrices: 255
Number of processes: 4
Density of graph: 75%

Time Serial: 1.38871

Time Parallel: 0.299521

Acceleration: 4.63643
```

Рисунок 4. Пример обработки графа с 9015 вершинами

При малом числе вершин исходного графа дополнительно проводится проверка корректности:

```
0 1 1 1 1          Last graph of Serial LND:  011101110
1 0 1 1 0
1 1 0 1 1          Last graph of Parallel LND:011101110
1 1 1 0 1
1 0 1 1 0          Algorithm is correct

0 1
1 0

0 1 1
1 0 1
1 1 0
```

Рисунок 5. Проверка совпадения результатов

Руководство программиста

Структура данных:

`int Size` – число вершин графа

`int countMatr` – количество получаемых разбиений

`int density` – значение плотности графа

`int **Matr` – матрица смежности

`int *traceP` – вектор перестановок для разделения

`bool last` – признак режима разделения

`int rank` – ранг текущего процесса

`int h` – высота дерева распределения

Реализации методов:

`void Init(int Size,int countMatr,int density,int **Matr,int **MatrS,int *traceP,int *traceS )` – Инициализация всех переменных и заполнение матрицы смежности случайными признаками наличия дуги с заданной плотностью.

`void Cut(int **Matr, int* trace, int Size, int countMatr, bool last, int rank)` – Выполнение разделения заданного графа. Если указатель `last` = false, то функция выполнит только одну итерацию разделения. Если `last` = true, то будут производиться деления, пока не будет получено `countMatr` разбиений.

`void Tree(int **  Size, int countMatr,int k,int*trace,  int root,  int h)` – Построение дерева распределения для процессов с соответствующими вызовами функции `Cut`.

`void PrintLastMatr(int **Matr, int Size)` – Вывод на экран указанной матрицы смежности

`int main(int argc, char* argv[])` – Стартовая функция программы, инициирует проведение последовательной и параллельной версий алгоритма. Производит проверку корректности и выводит результаты на экран.

Подтверждение корректности

Выявлены две оценки корректности работы программы:

- Оценка допустимости распределения задач по процессам – сумма размеров разделений совпадает с размером исходной матрицы смежности.
- Оценка верности разбиения – последнее разделение, полученное в параллельной версии совпадает с последовательной.

Результаты экспериментов

Для проведения экспериментов использовался один из кластеров ННГУ, тесты проводились на ядрах. Кластер имеет архитектуру вычислительных систем семейства «СКИФ» и включает в себя:

1. 64 двухпроцессорных двухъядерных сервера Intel Xeon 3.2 GHz, 4 Gb;
2. Сетевое оборудование на основе Gigabit Ethernet;
3. Пиковая производительность составляет 3 Тфлопс

При проведении 3 экспериментов изменялись по очереди следующие параметры:

- Число вершин
- Число разбиений
- Плотность графа

Для опорного эксперимента использовались параметры: 15000;256;75

Число вершин		15000	10000	15000	15000
Число разбиений		256	256	64	256
Плотность графа (%)		75	75	75	15
Среднее время последовательного алгоритма		5.5631	2.2856	5.8443	5.90055
2 процесса	Время работы(с)	2.7549	0.8470	2.5059	2.5991
	Ускорение	2.0193	1.9368	2.3174	2.2788
4 процесса	Время работы(с)	1.2910	0.6327	1.3456	1.36533
	Ускорение	4.2817	3.6123	4.3430	4.3217
8 процессов	Время работы(с)	0.7496	0.3944	0.7665	0.7440
	Ускорение	7.4233	5.7842	7.6227	7.9371
16 процессов	Время работы(с)	0.4733	0.2880	0.5136	0.4895
	Ускорение	11.7842	7.9143	11.3749	12.0499

Рисунок 6. Таблица показателей кластера

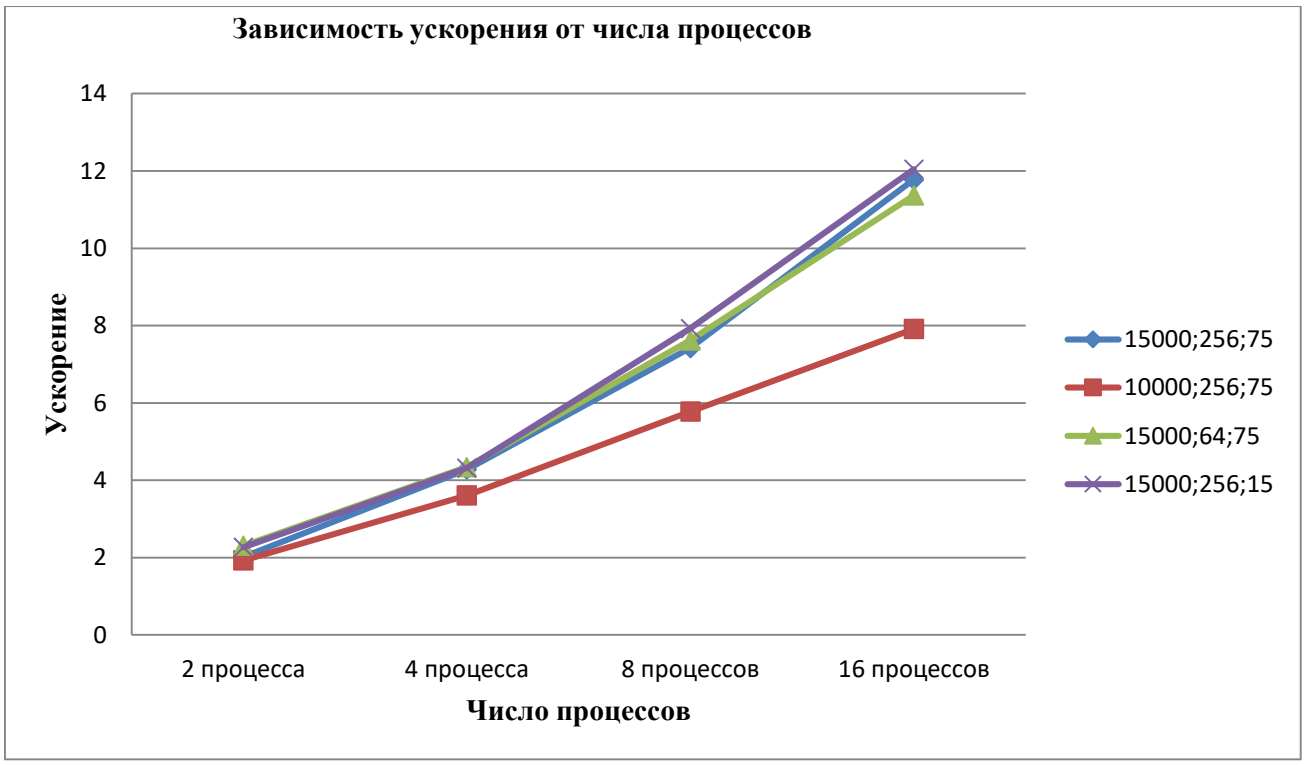

Рисунок 7. График показателей ускорения

Показатели, полученные при проведении экспериментов на кластере проясняют следующую зависимость - единственный параметр, значительно влияющий на скорость выполнения алгоритма, как нетрудно догадаться, является размер матрицы смежности. Остальные параметры не смогли привести к значительным изменениям. Это связано с тем, что реализованный алгоритм в любом случае будет обрабатывать всю матрицу целиком. Поэтому используемый метод деления графов лучше использовать для наиболее плотных, чтобы не терять эффективность. Если граф разрежен (плотность дуг 5-10%), то имеет смысл задавать его не матрицей, а списком смежностей.

Как видно из таблицы, на 2 и 4 процессах при размерности матрицы смежности 15000 было получено сверхлинейное ускорение. Такая ситуация возникла из-за неравномерности выполнения последовательной и параллельной программ. Когда задача выполнялась на одном процессе происходила сильная нехватка кэш-памяти, из-за чего приходилось хранить часть обрабатываемых данных в медленной внешней памяти. При выполнении задачи на нескольких процессах эта величина нехватки уменьшается, так как исходная матрица разделена на части и соответственно требует меньше памяти к каждому из них.

Пересылка данных не вносит значительных временных затрат, так как она организована параллельно по схеме MPI_Bcast.

Заключение

В результате выполнения лабораторной был реализован алгоритм, позволяющий обрабатывать граф.

В процессе написания работы были использованы средства языка C++ и API MPI.

Были применены некоторые знания из дискретной математики.

Проведено тестирование программы на кластере ННГУ.

Описаны и продемонстрированы результаты и наблюдения.

Литература

1. Гергель В.П., Стронгин Р.Г Основы параллельных вычислений для многопроцессорных вычислительных систем Н. Новгород: Изд-во ННГУ, 2001
2. ТЕОРИЯ ГРАФОВ. Алексеев В.Е., Захарова Д.В. Электронное учебно-методическое пособие. – Нижний Новгород: Нижегородский госуниверситет, 2012.

Приложение

```
#include <iostream>
#include <fstream>
#include <math.h>
#include <stdlib.h>
#include <mpi.h>
#include <string>

using namespace std;

void Init(int Size,int countMatr,int density,int **Matr,int **MatrS,int *traceP,int
*traceS )
{
	for (int i = 0; i < Size; i++)
	{
		traceP[i] = i;
		traceS[i]=i;
	}
	for (int i = 0; i < Size; i++)
	{
		for (int j = i + 1; j < Size; j++)
		{
			if (i != j)
			{
				MatrS[i][j] = MatrS[j][i] =Matr[i][j] = Matr[j][i] = (rand() %
100 < density ? (1) : 0);// в 75% - дуги
			}
			else
			{
				MatrS[i][j] = MatrS[j][i]= Matr[i][j] = Matr[j][i] = 1;
			}
		}
	}
	for (int i = 0; i < Size; i++)
	{
		MatrS[i][i] = Matr[i][i] = 0;
	}
}
void PrintLastMatr(int **Matr, int size)
{
	for (int i = 0; i < size; i++)
	{
		for (int j = 0; j < size; j++)
		{
			cout << Matr[i][j] << " ";
		}
		cout << endl;

	}
	cout << endl;
}
void Cut(int **Matr, int * trace, int size, int countMatr, bool last,int rank)
{
	if(size<60)//вывод матриц + запись данных для проверки корректности
	{
		PrintLastMatr(Matr,size);


		if(rank==0)
		{

			FILE *f;
			string Buf;
			f = fopen("check.txt","w");
```

```
                ofstream fout1 ("check.txt");
                for (int i = 0; i <size; i++)
                {
                        for (int j = 0; j < size; j++)
                        {
                                fout1 << Matr[i][j];
                                fclose(f);
                        }
                }
        }
}


int *mas = new int[size];
int *part = new int[size];

part[0] = 0;
mas[0] = 0;
for (int i = 1; i < size; i++)
{
        mas[i] = 1;
        part[i]= -1;
}

int k = 1;//номер первой используемой вершины
int l = 0;

//первое нумерование вершин (поиск граничной вершины)

for (int i=1; i < size;) {
        for (int j = 0; j < size; j++) {
                if (l == i)
                        break;
                if (Matr[part[l]][j] == 1 && (mas[j] > mas[part[l]] + 1))
                {
                        part[i] = j;
                        mas[j] = mas[part[l]] + 1;
                        i++;
                }
        }
        if (l == i)
        {
                break;
        }
        else
        {
                l++;
        }
        k=i;
}

int start = part[k - 1];

for (int i = 0; i < size; i++)
{
        part[i] = i;
}

int temp = part[0];
part[start] = part[0];
part[0] = temp;


mas[start] = 0;
for (int i = 0; i < size; i++)
```

```
{
	mas[i] = 1;
}

l = 0;

int SizeSubMatr = size * (countMatr / 2) / countMatr;

//второе нумерование вершин (оптимальное, начинающееся от граничной вершины)
for (int i=1; i < SizeSubMatr;)
{
	for (int j = 0; j < size; j++)
		if (Matr[part[l]][part[j]] == 1 && (mas[part[j]] > mas[part[l]] + 1))
		{
			mas[part[j]] = mas[part[l]] + 1;

			int tmp = part[i];
			part[j] = part[i];
			part[i] = tmp;

			i++;
		}
		if (l == i)
		{
			break;
		}
		else
		{
			l++;
		}
}

int tmp;
//перезапись наличия дуг(значений матрицы смежности)
for (int i = 0; i < SizeSubMatr; i++)
{
	//одну половину матрицы смежности
	for (int j = 0; j < size; j++)
	{
		tmp = Matr[i][j];
		Matr[i][j] = Matr[part[i]][j];
		Matr[part[i]][j] = tmp;
	}
	//вторую (симметричную)
	for (int j = 0; j < size; j++)
	{
		tmp = Matr[j][i];
		Matr[j][i] = Matr[j][part[i]];
		Matr[j][part[i]] = tmp;
	}
}

//конечная операция на итерации алгоритма

//записываем вектор вершин для перестановки в порядке формирования двух графов
for (int i = 0; i < size; i++)
{
	part[i] = trace[part[i]];
}
//перезаписываем вектор вершин для перестановки
for (int i = 0; i < size; i++)
{
	trace[i] = part[i];
}
```

```
	//в итоге получим один вектор в котором содержится новый порядок создания 2 матриц смежности

	//если нужно делить до конца без сторонних перераспределений (последовательно на одном процессе)
	//функция работает рекурсивно
	if (last) //если сработал сигнал, что делим до конца
	{



		int **Submatr = 0;//создаем вторую парную матрицу
		//её парой будет часть исходной

		if (countMatr > 2)//если надо получить более 2х матриц смежности
		{
			Submatr = new int*[size - SizeSubMatr];
			for (int i = SizeSubMatr; i < size; i++)
			{
				Submatr[i - SizeSubMatr] = &Matr[i][SizeSubMatr];//записываем нужную часть от исходной матрицы
			}
		}


		if (countMatr == 3)//если нужно произвести ровно 1 деление (в итоге получим 2 новых графа)
		{
			Cut(Submatr, &trace[SizeSubMatr], size - SizeSubMatr, 2, last,rank);//делим только одну матрицу (вторая уже разделена)

		}

		if (countMatr > 3)
		{
			//продолжаем делить обе
			Cut(Matr, trace, SizeSubMatr, countMatr/2, last,rank);
			Cut(Submatr, &trace[SizeSubMatr], size - SizeSubMatr, countMatr - (countMatr/2), last,rank);
		}
	}
}
void Tree(int **  a, int n,int k,int*trace,  int root,  int h) {

	int rank;
	int ProcNum;
	int size = (int)(powl(2, h));
	int unnatureSize = (int)(powl(2, h)) + root;

	MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	int unnatureRank = rank, unnatureProcNum = ProcNum + root;
	unnatureRank = rank - root;

	if ((unnatureRank >= 0) && (unnatureRank < size)) {
		int unnatureReceiverRank = unnatureRank + unnatureSize , receiverRank = rank + size;

		if (unnatureReceiverRank < unnatureProcNum) {

			if (receiverRank < ProcNum) {

	// отправляем часть исходной матрицы на обработку другим процессом
			int n1 = n * (k / 2) / k;
			int k1 = k / 2;
```

```
                        MPI_Send(&n1, 1, MPI_INT,receiverRank, 0, MPI_COMM_WORLD);
                        MPI_Send(&k1, 1, MPI_INT, receiverRank, 0, MPI_COMM_WORLD);

                        for (int i = 0; i < n; i++)
                        {
                        MPI_Send(a[i], n1, MPI_INT, receiverRank, 0, MPI_COMM_WORLD);
                        }
                        MPI_Send(trace, n1, MPI_INT,receiverRank, 0, MPI_COMM_WORLD);


                        //перезапись текущей матрицы в процессе
                        //заменяем её на остаточную половину
                        int n2 = n - n1;
                        int k2 = k - k / 2;
                        int **b = new int*[n - n1];
                        int* buf=new int[n2];
                for (int i = n * (k / 2) / k; i < n; i++)
                {
                        buf[i - n * (k / 2) / k]=trace[i - n1];
                        b[i - n * (k / 2) / k] = &a[i][n1];
                }

trace = new int[n2];
  a = new int*[n2];
  for (int i = 0; i < n2; i++)
{
        trace[i]=buf[i];
a[i] = new int[n2];
  a[i]=b[i];
  }
                n=n2;
                k=k2;

                //если распределение завершено

                                if(ProcNum==size*2)
                        {
                                cout << "Procrank = " << rank << " Size = " << n << " -> " <<
k << " matrices "<<endl;
                                Cut(a, trace, n, k,true,rank);// обработка самостоятельно
имеющегося куска
                                return;
                        }

                        }


                }
                else {
                        return;
                }
        }
        else {
                        if ((unnatureRank >= size) && (unnatureRank < 2 * size))
                        {
                                int senderRank = rank - size;// -3

                                if (senderRank >= 0 && ( rank > root))
                                {

                        MPI_Recv(&n, 1, MPI_INT,  senderRank, 0,
MPI_COMM_WORLD,MPI_STATUSES_IGNORE);
                        MPI_Recv(&k, 1, MPI_INT,  senderRank, 0,
MPI_COMM_WORLD,MPI_STATUSES_IGNORE);
```

```
                    a = new int*[n];
                    for (int i = 0; i < n; i++)
                    {a[i] = new int[n];}
                    trace = new int[n];
                    for (int i = 0; i < n; i++)
                    {
                    MPI_Recv(a[i], n, MPI_INT,  senderRank, 0,
MPI_COMM_WORLD,MPI_STATUSES_IGNORE);
                    }

                    MPI_Recv(trace, n, MPI_INT, senderRank, 0,
MPI_COMM_WORLD,MPI_STATUSES_IGNORE);


                                        if(ProcNum==size*2)

                    {
                            cout << "Procrank = " << rank << " Size = " << n << " -> " <<
k << " matrices "<<endl;
                            Cut(a, trace, n, k,true,rank);
                            return;
                    }
                            }
                    }
        }

        h++;//высота дерева распределения

        if (size > sqrtl(ProcNum))
        {
                return;
        }

        Tree(a, n,k,trace , root, h);

}

int main(int argc, char* argv[])
{
        int Procrank;
        int ProcNum;

        int Size;//число вершин графа

        int countMatr;//число разбиений

        //вектора перестановок
        int *trace = NULL;
        int *traceS= NULL;

        double t1=0, t2=0;
        double t1s=0, t2s=0;

        int density;//плотность дуг

        //матрицы смежности
        int **Matr = NULL;
        int **MatrS= NULL;

        int sizeCheck=0;
        int sizeCheckS=0;

        string CheckBuf;
        string CheckBufS;
```

```
	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &Procrank);
	MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);

		if (argc > 1)
		{	Size = atoi(argv[1]);}
		else
		{	Size = 15000;}
		if (argc > 2)
		{	countMatr = atoi(argv[2]);}
		else
		{	countMatr = 256;}
		if (argc > 3)
		{density = atoi(argv[3]);}
		else
			{density = 15;}

		if (Procrank == 0)
	{
		trace = new int[Size];
		traceS = new int[Size];

		Matr = new int*[Size];
		MatrS = new int*[Size];
		for (int i = 0; i <Size; i++)
			{
		Matr[i] = new int[Size];
	    MatrS[i] = new int[Size];
		}
		Init(Size,countMatr,density,Matr,MatrS,trace,traceS);

	t1s = MPI_Wtime();

	Cut(MatrS, traceS, Size, countMatr, true,0);

	t2s = MPI_Wtime();
if (Size<60)
{
FILE *t;
string bufS;
t = fopen("check.txt","r");
ifstream fin1 ("check.txt");
fin1 >> CheckBufS;
fclose(t);
}


	t1 = MPI_Wtime();
	}

Tree(Matr,Size,countMatr,trace , 0, 0);


	MPI_Barrier(MPI_COMM_WORLD);
	if (Procrank == 0)
	{
		t2 = MPI_Wtime();

	cout<< endl << "Size of adjacency matrix: " <<Size << endl;
	cout << "Number of dividual matrices: " << countMatr << endl;
	cout << "Number of processes: " << ProcNum << endl;
	cout<<"Density of graph: "<<density<<"%"<< endl;

	cout<< endl << "Time Serial: "<<t2s-t1s<<endl;
```

```
        cout << endl << "Time Parallel: " << t2 - t1 << endl;
        cout << endl <<"Acceleration: "<<(t2s-t1s)/(t2-t1)<<endl;

               delete[] *Matr;
               delete[]*MatrS;

if(Size<60)
{
FILE *t;
string bufS;
t = fopen("check.txt","r");
ifstream fin1 ("check.txt");
fin1 >> CheckBuf;

fclose(t);

cout << endl <<"Last graph of Serial LND:  "<< CheckBufS << endl;
cout << endl <<"Last graph of Parallel LND:" << CheckBuf << endl;

if(CheckBufS==CheckBuf){cout<< endl<<"Algorithm is correct";}
}


        }
        MPI_Finalize();
        return 0;
}
```