Введение в параллельные вычисления

д.т.н. Мокрова Наталия Владиславовна

асп. Морунов Егор, Сырко Денис

пятница	ауд. 118, 119	
12:50 – 14:20	Лекция	
	1 неделя	2 неделя
14:35 - 17:55	Лаб – КС44	Лаб - КС40

Лекция 10. Моделирование и анализ параллельных вычислений

В.П. Гергель, Р.Г. Стронгин «Основы параллельных вычислений для многопроцессорных вычислительных систем»

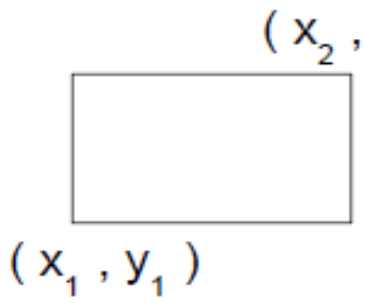
- http://www.software.unn.ac.ru/ccam/files/HTML_Version/index.html#Contents
- http://www.hpcc.unn.ru/files/HTML_Version/

N.I. Lobachevsky State University of Niznhi Novgorod

- http://www.hpcc.unn.ru/

Модель вычислений в виде графа

Граф алгоритма вычисления площади прямоугольника, заданного координатами двух противолежащих углов.



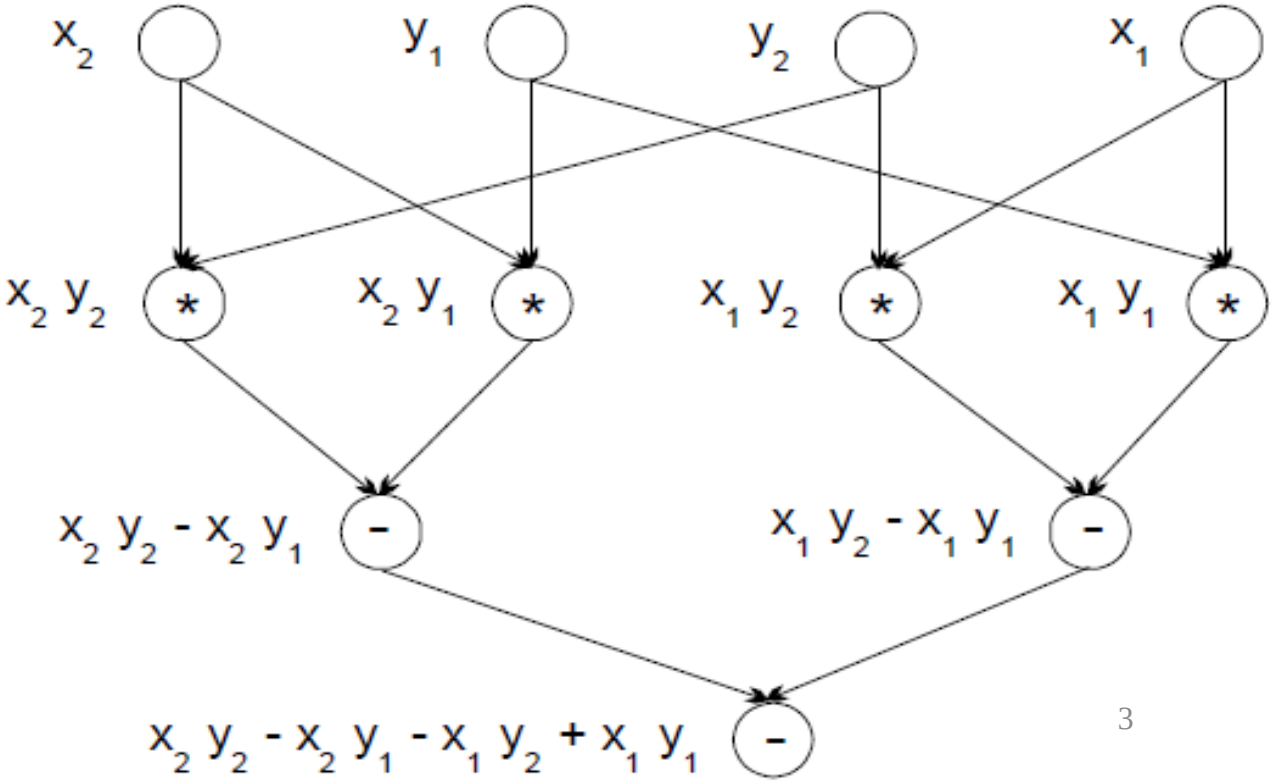
$$S = ((x_2 - x_1)(y_2 - y_1) = x_2 y_2 - x_2 y_1 - x_1 y_2 + x_1 y_1$$

Для описания информационных зависимостей в алгоритмах предложена модель "операции-операнды" (см. Bertsekas and Tsitsiklis (1989), Воеводин В.В. и Воеводин Вл.В. (2002))

Множество операций алгоритма и существующие информационные зависимости представлены в виде *ациклического ориентированного графа.*

G = (V, R), где V = { 1,…,V} – множество вершин (операции алгоритма), R – множество дуг (дуга r = (i, j), принадлежит графу только, если операция j использует результат выполнения операции i).

Разные схемы вычислений обладают различными возможностями для распараллеливания и при построении модели вычислений требуется выбор наиболее подходящей для параллельного исполнения вычислительной схемы алгоритма.



Показатели эффективности параллельного алгоритма

Ускорение (speedup), получаемое при использовании параллельного алгоритма для p процессоров, по сравнению с последовательным вариантом выполнения вычислений $S_p(n) = T_1(n) / T_p(n)$,

т.е. отношение времени решения задач на скалярной ЭВМ к времени выполнения параллельного алгоритма (n – количество входных данных задачи, использовано для параметризации вычислительной сложности).

Эффективность (efficiency) использования параллельным алгоритмом процессоров при решении задачи

$$E_p(n) = T_1(n) / (pT_p(n)) = S_p(n) / p$$

(величина эффективности определяет среднюю долю времени выполнения алгоритма, в течение которой процессоры реально используются для решения задачи).

Из приведенных соотношений в наилучшем случае $S_p(n) = p$ **и** $E_p(n) = 1$.

При выборе параллельного способа решения задачи может использоваться оценка ***стоимости*** (***cost***) вычислений, как произведение времени параллельного решения задачи и числа используемых процессоров.

Определяют понятие ***стоимостно-оптимального*** (***cost-optimal***) параллельного алгоритма как метода, стоимость которого пропорциональна времени выполнения наилучшего последовательного алгоритма.

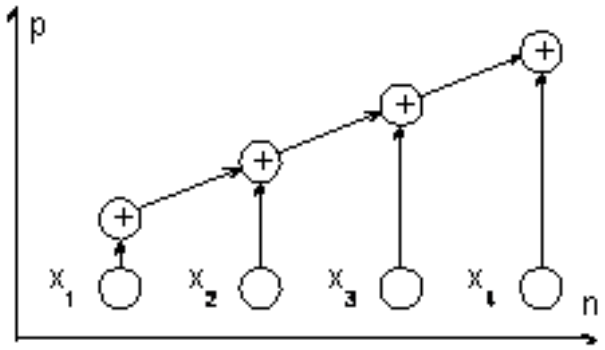
$$C_p = pT_p$$

Вычисление частных сумм

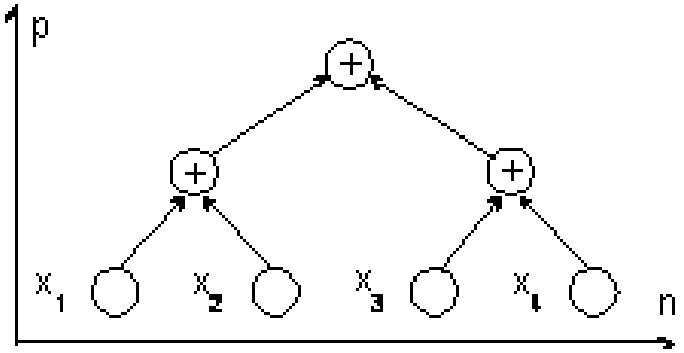
Задача нахождения частных сумм последовательности числовых значений

$$S_k = \sum_{i=1}^{k} x_i, \quad 1 \le k \le n,$$

Задача суммирования является частным случаем общей *задачи редукции.*



Последовательная вычислительная схема алгоритма суммирования.
Данный "стандартный" алгоритм суммирования допускает только строго последовательное исполнение и не может быть распараллелен.

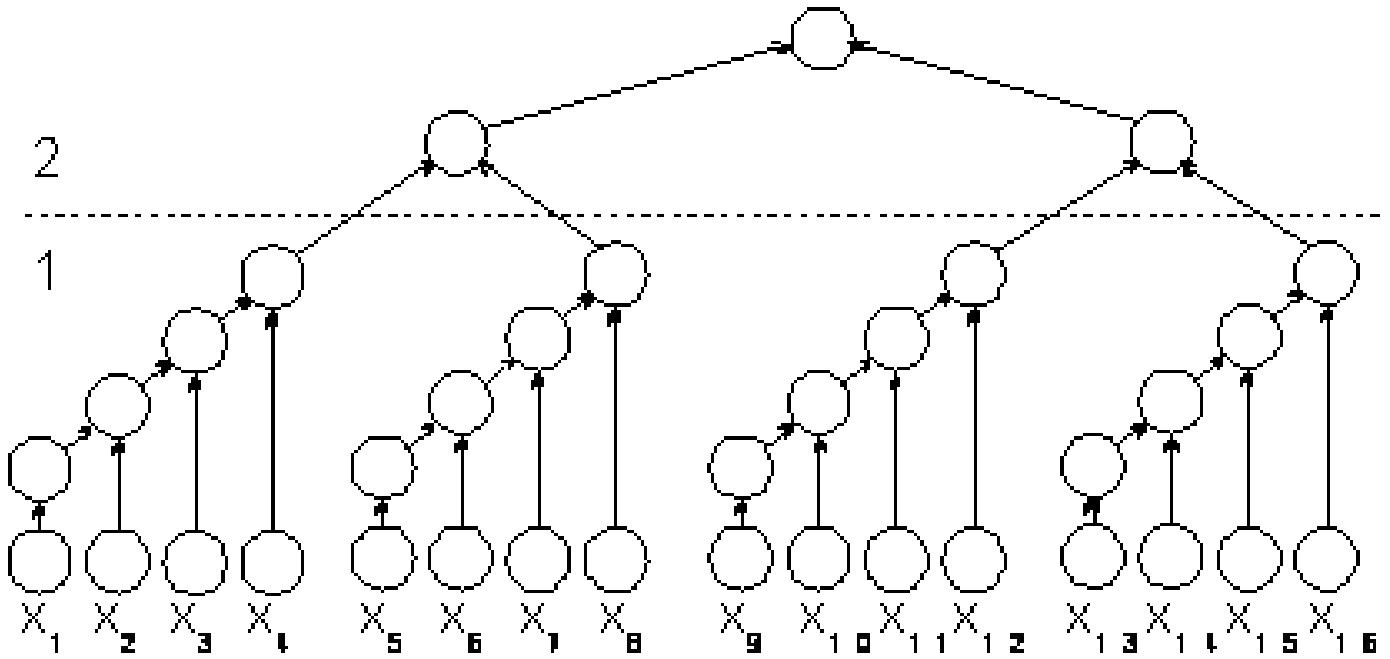


Каскадная схема алгоритма суммирования.
Необходимое для выполнения каскадной схемы количество процессоров p = n/2.
Эффективность использования процессоров уменьшается при увеличении количества суммируемых значений.

Модифицированная каскадная схема

В этом варианте каскадной схемы все проводимые вычисления подразделяется на два последовательно выполняемых этапа суммирования:

- на первом этапе вычислений все суммируемые значения подразделяются на $(n/\log_2 n)$ групп, в каждой из которых содержится $\log_2 n$ элементов;
  далее для каждой группы вычисляется сумма значений при помощи последовательного алгоритма суммирования;
  вычисления в каждой группе могут выполняться независимо друг от друга (т.е. параллельно – для этого необходимо наличие не менее процессоров $(n/\log_2 n)$);
- на втором этапе для полученных $(n/\log_2 n)$ сумм отдельных групп применяется обычная каскадная схема.



Вычисление всех частных сумм

Проведем анализ возможных способов последовательной и параллельной организации вычислений (плакат 5 зад. 1).

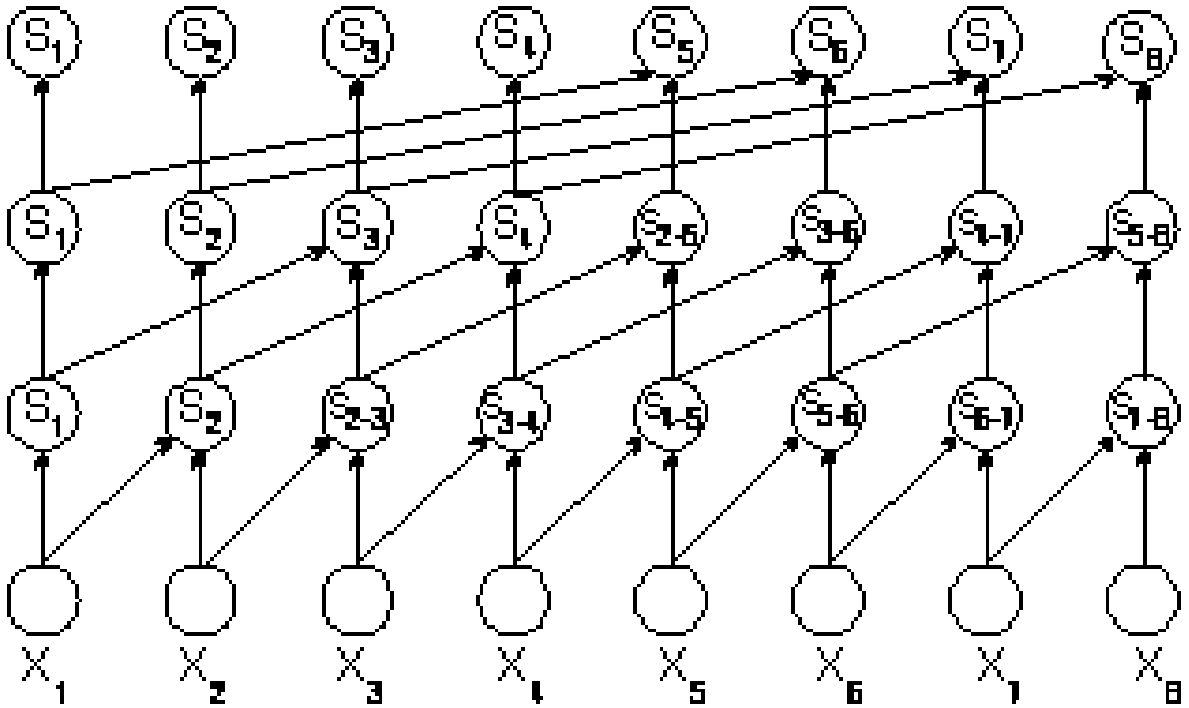
Вычисление всех частных сумм на скалярном компьютере может быть получено при помощи последовательного алгоритма суммирования при количестве операций (!) $T_1 = n$

При параллельном исполнении применение каскадной схемы в явном виде не приводит к результатам, достижение эффективного распараллеливания требует привлечения новых подходов для разработки параллельно-ориентированных алгоритмов решения задач.

Для рассматриваемой задачи нахождения всех частных сумм алгоритм, обеспечивающий получение результатов за $\log_2 n$ параллельных операций (как в случае вычисления общей суммы), может состоять в следующем:

- перед началом вычислений создается копия вектора суммируемых значений ($S = x$);
- далее на каждой итерации суммирования i, $1 \le i \le \log_2 n$ формируется вспомогательный вектор Q путем сдвига вправо вектора S на 2^{i-1} позиций (освобождающие при сдвиге позиции слева устанавливаются в нулевые значения); итерация алгоритма завершается параллельной операцией суммирования векторов S и Q: $S \leftarrow S + Q$

Схема параллельного алгоритма вычисления всех частных сумм



Величины S_{i-j} означают суммы значений от i до j элементов числовой последовательности

Параллельный алгоритм выполняется за $\log_2 n$ параллельных операций сложения. На каждой итерации алгоритма параллельно выполняются n скалярных операций сложения и, общее количество выполняемых скалярных операций

$$L_{пар} = n \log_2 n$$

(параллельный алгоритм содержит большее (!) количество операций по сравнению с последовательным способом суммирования).

Необходимое количество процессоров определяется количеством суммируемых значений $p = n$.

Эффективность параллельного решения

Показатели ускорения и эффективности параллельного алгоритма вычисления всех частных сумм можно оценить как:

Ускорение

$$S_p = T_1 / T_p = n / \log_2 n,$$

Эффективность

$$E_p = T_1 / pT_p = n / (p \log_2 n) = n / (n \log_2 n) = 1 / \log_2 n$$

Эффективность алгоритма уменьшается при увеличении числа суммируемых значений и при необходимости повышения величины этого показателя может оказаться полезной модификация алгоритма как и в случае с обычной каскадной схемой.

Умножение матрицы на вектор

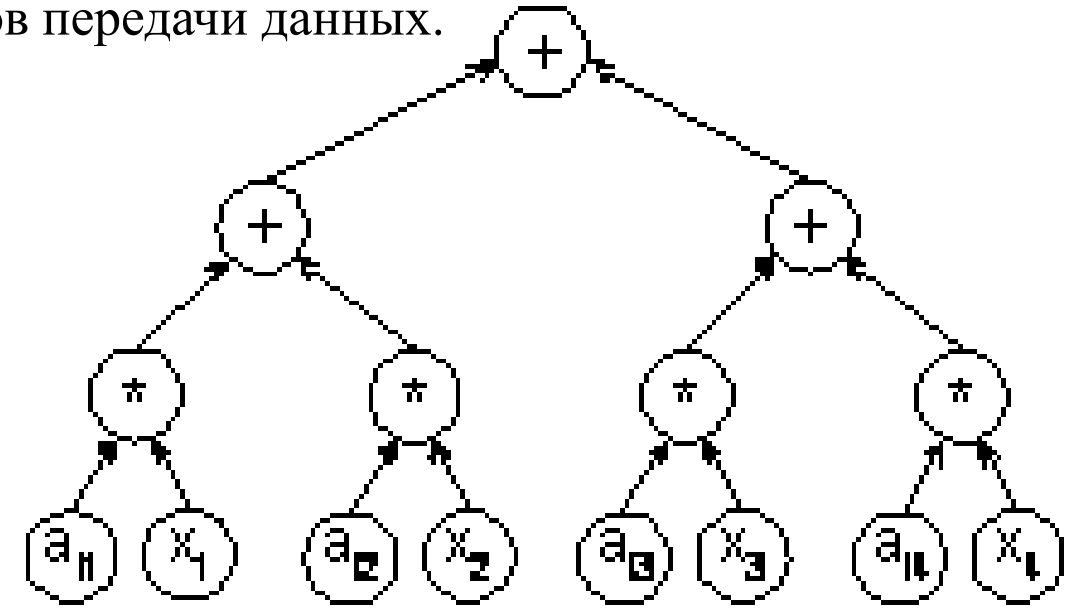
Задача умножения матрицы на вектор $y_i = \sum_{j=1}^{n} a_{ij} x_j , 1 \le i \le n$

Общее количество необходимых скалярных операций $T_1 = 2n^2$

Достижение максимально возможного быстродействия $p = n^2$.

Наиболее подходящими топологиями являются структуры, в которых обеспечивается быстрая передача данных (пути единичной длины) в каскадной схеме суммирования (см. рис.). Используются топологии с полной системой связей (**полный граф**) и **гиперкуб**. Другие топологии приводят к возрастанию коммуникационного времени из-за удлинения маршрутов передачи данных.

Применение вычислительной системы с топологией в виде прямоугольной **двумерной решетки** размера n x n приводит к простой и наглядной интерпретации выполняемых вычислений (структура сети соответствует структуре обрабатываемых данных).



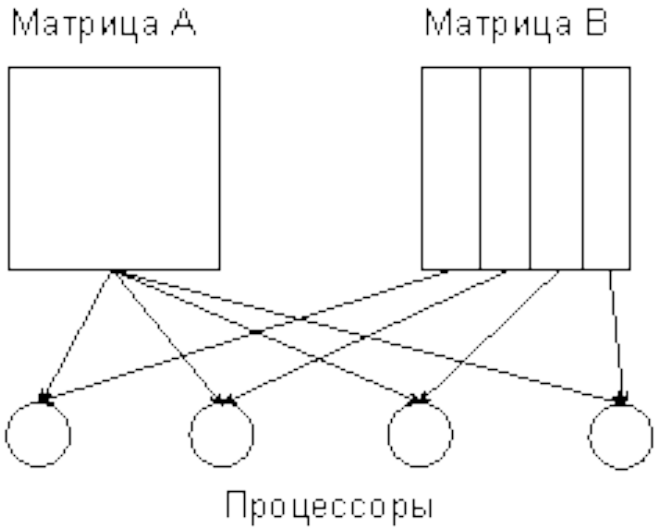
Вычислительная схема операции умножения строки матрицы на вектор.

Матричное умножение

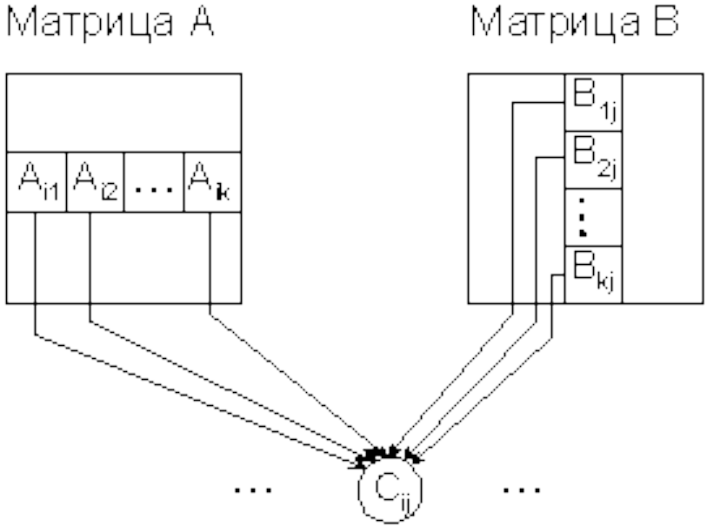
Задача умножения матрицы на матрицу $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}, 1 \le i, j \le n$

Количество операций скалярных умножений и сложений n^3.

Вычислительная схема матричного умножения при использовании макроопераций умножения матрицы A на столбец матрицы B



Информационный граф матричного умножения при блочном представлении матриц



Пересылка данных оказывается распределенной по времени, что позволяет совместить процессы передачи и обработки данных.

Иерархическая декомпозиционная методика построения параллельных методов – одна из основных и широко используемых.

Геометрический принцип распараллеливания – распределение между процессорами обрабатываемых данных с учетом близости их расположения в содержательных постановках задач.

Пузырьковая сортировка

Используется модификация метода *чет-нечетная перестановка* (*odd-even transposition*).

В алгоритм сортировки введены два разных правила выполнения итераций метода – в зависимости от четности или нечетности номера итерации сортировки для обработки выбираются элементы с четными или нечетными индексами соответственно, сравнение выделяемых значений всегда осуществляется с их правыми соседними элементами.

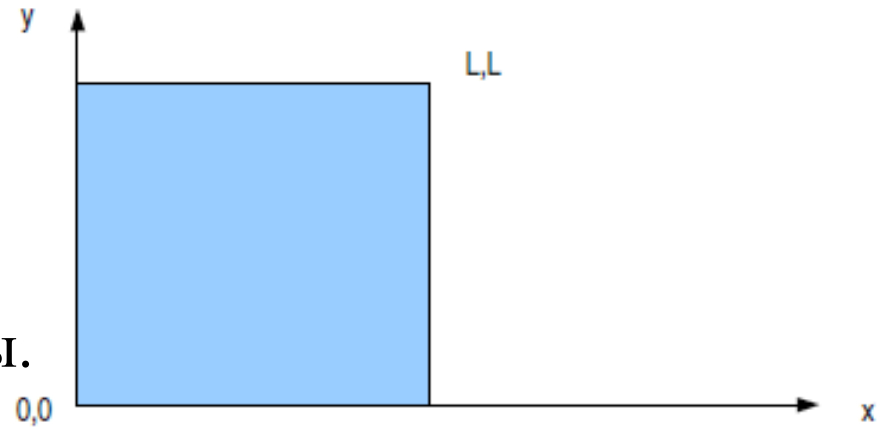
№ и тип итерации	Процессоры			
	1	2	3	4
Исходны еданные	2 3	3 8	5 6	1 4
1 нечет (1,2),(3.4)	2 3	3 8	5 6	1 4
	2 3	3 8	1 4	5 6
2 чет (2,3)	2 3	3 8	1 4	5 6
	2 3	1 3	4 8	5 6
3 нечет (1,2),(3.4)	2 3	1 3	4 8	5 6
	1 2	3 3	4 5	6 8
4 чет (2,3)	1 2	3 3	4 5	6 8
	1 2	3 3	4 5	6 8

Пример сортировки данных параллельным методом чет-нечетной перестановки
http://www.software.unn.ac.ru/ccam/files/HTML_Version/

Краевая задача для уравнения Лапласа

В двухмерном пространстве дифференциальное уравнение в частных производных уравнение Лапласа

$$\frac{\partial_2 U}{\partial x^2}+\frac{\partial_2 U}{\partial y^2}=0$$



Условия Дирихле или первые краевые условия

$$U(0,y)=U(L,y)=U(x,0)=U(x,L)=1$$

Для решения задачи используются численные методы.

Разностная схема

$$\omega_h=\{(x_i,y_i),\ x_i=ih,\ y_j=jh,\ i=0..\mathrm{N},\ j=0..\mathrm{n}\}$$

$$\frac{u_{i+1,j}-2\mathrm{u}_{i,j}+u_{i-1,j}}{h^2}+\frac{u_{i,j+1}-2\mathrm{u}_{i,j}+u_{i,j-1}}{h^2}=0,\quad i=\overline{1..\mathrm{N}-1},\quad j=\overline{1..\mathrm{N}-1}$$

$$u_{0,j}=1,\ j=0..\mathrm{N}$$

$$u_{N,j}=1,\ j=0..\mathrm{N}$$

$$u_{i,0}=1,\ i=0..\mathrm{N}$$

$$u_{i,N}=1,\ i=0..\mathrm{N}$$

Решение дифференциальных уравнений

При решении задачи методом конечных разностей на расчётной области строится сетка, выбирается разностная схема и для каждого узла сетки записывается разностное уравнение, производится учёт краевых условий (для краевых условий второго и третьего рода так же строится некоторая разностная схема).

Получается система линейных алгебраических уравнений, решая которую получают приближенные значения решения в узлах.

Итерационный метод Якоби – разновидность метода простой итерации для решения системы линейных алгебраических уравнений.

$$A\vec{x} = \vec{b}, \text{ где } A = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix}, \quad \vec{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$$

$$u_{i,j}^{k+1} = \frac{1}{4}\left(u_{i+1,j}^{k} + u_{i-1,j}^{k} + u_{i,j+1}^{k} + u_{i,j-1}^{k}\right)$$

Или

$$\begin{cases} a_{11}x_1 + \dots + a_{1n}x_n = b_1 \\ \dots \\ a_{n1}x_1 + \dots + a_{nn}x_n = b_n \end{cases}$$

$$i = \overline{1..N-1}, \quad j = \overline{1..N-1}, \quad k = \overline{0,1,..}$$

```cpp
#include <iostream>
#include <algorithm>
#include <math.h>
#include <omp.h>
#include <fstream>

using namespace std;

#define eps 0.001

int main() {
	int N = 200;
	int i,j;
	int iter=0;
	double ** grid = new double * [N+1];
	double ** newgrid = new double * [N+1];

	for (i=0;i<N+1;i++)
	{
		grid[i] = new double[N+1];
		newgrid[i] = new double [N+1];
	}
	for (i=0;i<N+1;i++)
	{
		grid[i][0]=1;
		grid[0][i]=1;
		grid[N][i]=1;
		grid[i][N]=1;
		newgrid[i][0]=1;
		newgrid[0][i]=1;
		newgrid[N][i]=1;
		newgrid[i][N]=1;
	}
	for (i=1;i<N;i++)
		for (j=1;j<N;j++)
			grid[i][j]=0;

	double maxdiff=0;
	do{
		for (i=1;i<N;i++)
			for (j=1;j<N;j++)
			{
				newgrid[i][j]= 0.25 * (grid[i-1][j]+
				grid[i+1][j]+grid[i][j-1]+grid[i][j+1]);
			}
		maxdiff=0;
		for (i=1;i<N;i++)
			for (j=1;j<N;j++)
			{
				maxdiff = max(maxdiff, fabs(newgrid[i][j]-grid[i][j]));
			}

		for (i=1;i<N;i++)
			for (j=1;j<N;j++)
			{
				grid[i][j]=newgrid[i][j];
			}
		iter++;
	}while (maxdiff > eps);

	cout<<"Iter:"<<iter<<endl;
	ofstream fout("out.txt");
	for (i=0;i<N+1;i++)
	{
		for (j=0;j<N+1;j++)
			fout<<grid[i][j]<<" ";
		fout<<endl;
	}

	fout.close();

	for (i=0;i<N+1;i++)
	{
		delete [] grid[i];
		delete [] newgrid[i];
	}

	delete grid;
	delete newgrid;

	return 0;
}
```

Реализация: последовательная версия

(Лекции Науменко С.А.)

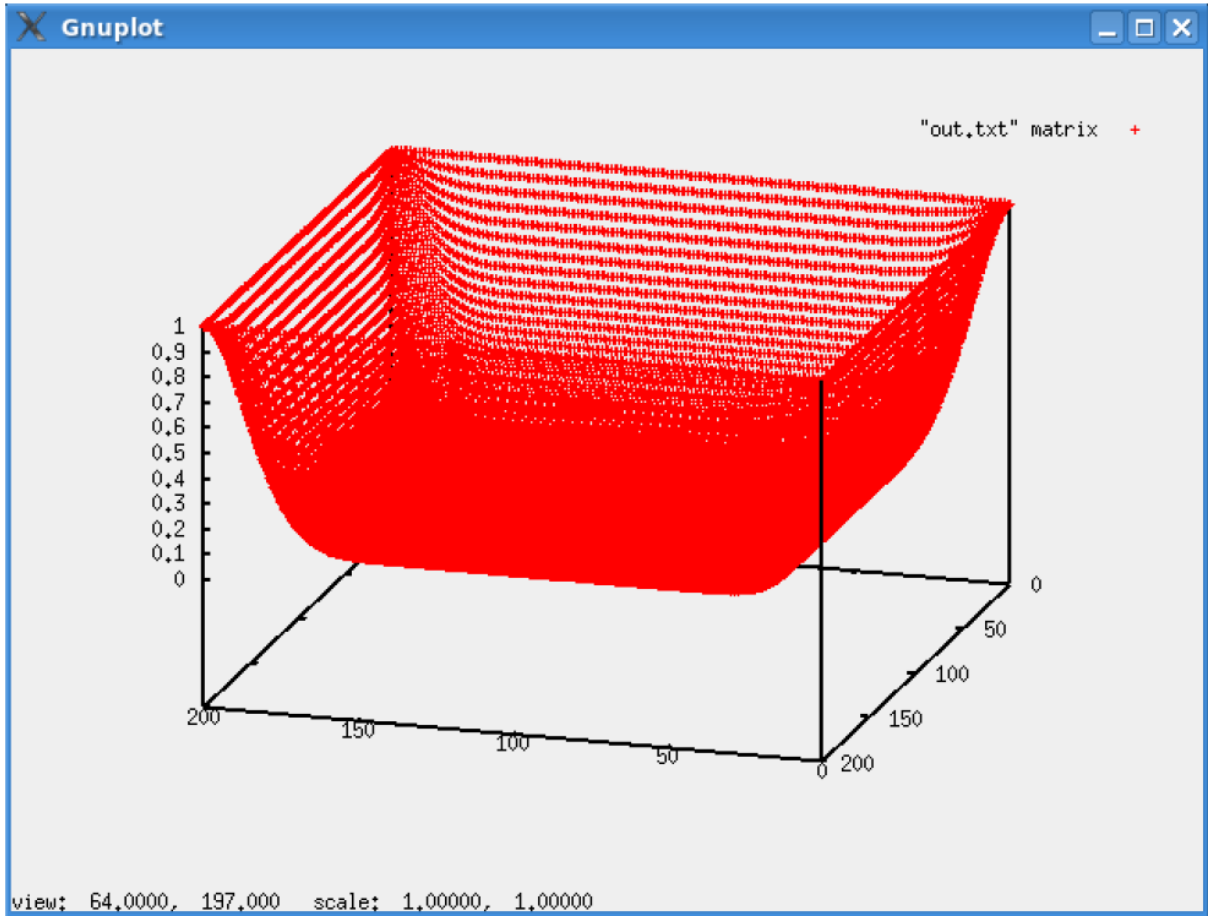
Реализация: параллельная версия

```
#include <math.h>
#include <omp.h>
#include <fstream>
using namespace std;
#define eps 0.001
int main()
{
    double start = omp_get_wtime();
    int i,j; int iter = 0;  int N = 200;    double maxdiff=0;
    double ** grid = new double * [N+1];    double ** newgrid = new double * [N+1];
    for (i=0;i<N+1;i++)
        {
            grid[i] = new double[N+1];  newgrid[i] = new double [N+1];
        }
#pragma omp parallel shared(grid,newgrid,maxdiff) private (i,j)
{
#pragma omp for
    for (i=0;i<N+1;i++)
        {
            grid[i][0]=1;    grid[0][i]=1;    grid[N][i]=1;    grid[i][N]=1;
            newgrid[i][0]=1; newgrid[0][i]=1; newgrid[N][i]=1; newgrid[i][N]=1;
        }
#pragma omp for //collapse(2)
    for (i=1;i<N;i++)
        for (j=1;j<N;j++)
            grid[i][j]=0;
}
```

```
do
{
#pragma omp parallel shared(grid,newgrid,maxdiff) private (i,j)
    {
#pragma omp for //collapse(2)
    for (i=1;i<N;i++)
        for (j=1;j<N;j++)
            newgrid[i][j]= 0.25 * (grid[i-1][j]+grid[i+1][j]+grid[i][j-1]+grid[i][j+1]);
#pragma omp master
        {
            maxdiff=0;
            for (i=1;i<N;i++)
                for (j=1;j<N;j++)
                    maxdiff = max(maxdiff, fabs(newgrid[i][j]-grid[i][j]));
        }
    }
#pragma omp barrier
#pragma omp for //collapse(2)
    for (i=1;i<N;i++)
        for (j=1;j<N;j++)
            {
                grid[i][j]=newgrid[i][j];
            }
    iter++;
}
    while (maxdiff > eps);
        for (i=0;i<N+1;i++)
            {
                delete [] grid[i]; delete [] newgrid[i];
            }
delete grid; delete newgrid; double end = omp_get_wtime();
cout<<"Time:"<<end-start<<endl;
return 0;
}
```

Построение графика

```
gnuplot
gnuplot> splot "out.txt" matrix
```



Сравнение работы программ на процессоре Centrino Duo T2050

Размер области	200	400	1000
Последовательная	0,77	3,28	20,1
OpenMP	0,73	2,97	18,6

Оптимизация алгоритма

```
for (i=1;i<N;i++)
        for (j=1;j<N;j++)
                newgrid[i][j]= 0.25 * (grid[i-1][j]+grid[i+1][j]+grid[i][j-1]+grid[i][j+1]);

for (i=1;i<N;i++)
        for (j=1;j<N;j++)
                grid[i][j]= 0.25 * (newgrid[i-1][j]+newgrid[i+1][j]+newgrid[i][j-1]+newgrid[i][j+1]);

maxdiff=0;
for (i=1;i<N;i++)
        for (j=1;j<N;j++)
                maxdiff = max(maxdiff, fabs(newgrid[i][j]-grid[i][j]));

for (i=1;i<N;i++)
        for (j=1;j<N;j++)
                grid[i][j]=newgrid[i][j];
```

раскрутка цикла +
избавиться от присвоений

Размер области	200	1000
Без оптимизации	0,78	20,6
С оптимизацией	0,48	12,6

Решение СЛАУ методом Якоби

Результаты вычислительных экспериментов http://www.hpcc.unn.ru/?dir=1052

Эксперименты проводились на следующих компьютерах:
1) Процессор Intel(R) Core(TM) i7-2630QM CPU @ 2.00GHz, ОС Windows 8 64-bit.
2) Процессор AMD Phenom(tm) II X4 945, ОС Windows 7 64-bit.
3) Процессор AMD Turion(tm) II P540 Dual-Core Processor 2.40GHz, ОС Windows 7 64-bit.
Каждое значение, представленное в данной таблице, является средним временем на основе 25 экспериментов.

Размер матрицы	8 процессов				4 процесса				2 процесса			
	Последовательный алгоритм (мс)	Параллельный алгоритм (мс)	Теор. время паралл. алгоритма (мс)	Ускорение	Последовательный алгоритм (мс)	Параллельный алгоритм (мс)	Теор. время паралл. алгоритма (мс)	Ускорение	Последовательный алгоритм (мс)	Параллельный алгоритм (мс)	Теор. время паралл. алгоритма (мс)	Ускорение
3000*3000	77	13	10	5,92	90	26	23	3,46	127	98	64	1,29
6000*6000	294	46	37	4,22	350	99	88	3,89	474	370	237	1,28
9000*9000	631	90	79	4,01	801	269	200	2,98	1189	953	595	1,24
12000*12000	1109	159	139	6,97	1439	565	360	2,55	2020	1789	1010	1,12
15000*15000	1721	254	215	6,78	2451	920	613	2,66	2912	2644	1456	1,1

Еще решение дифференциальных уравнений в частных производных

Численное решение задачи Дирихле для уравнения Пуассона – задача нахождения функции u = u (x, y), удовлетворяющей в области определения D уравнению

$$\begin{cases} \dfrac{\delta^2 u}{\delta x^2} + \dfrac{\delta^2 u}{\delta y^2} = f(x,y), & (x,y) \in D, \\ u(x,y) = g(x,y), & (x,y) \in D^0, \end{cases}$$

и принимающей значения g (x, y) на границе D^0 области D (f и g – функции, задаваемые при постановке задачи).

Для простоты в качестве области задания D использован единичный квадрат

$$D = \{(x,y) \in D : 0 \le x,y \le 1\}.$$

Метод конечных разностей (*метод сеток*) – один из распространенных подходов численного решения дифференциальных уравнений.

Область решения представляется в виде дискретного (как правило, равномерного) набора (*сетки*) точек (*узлов*).

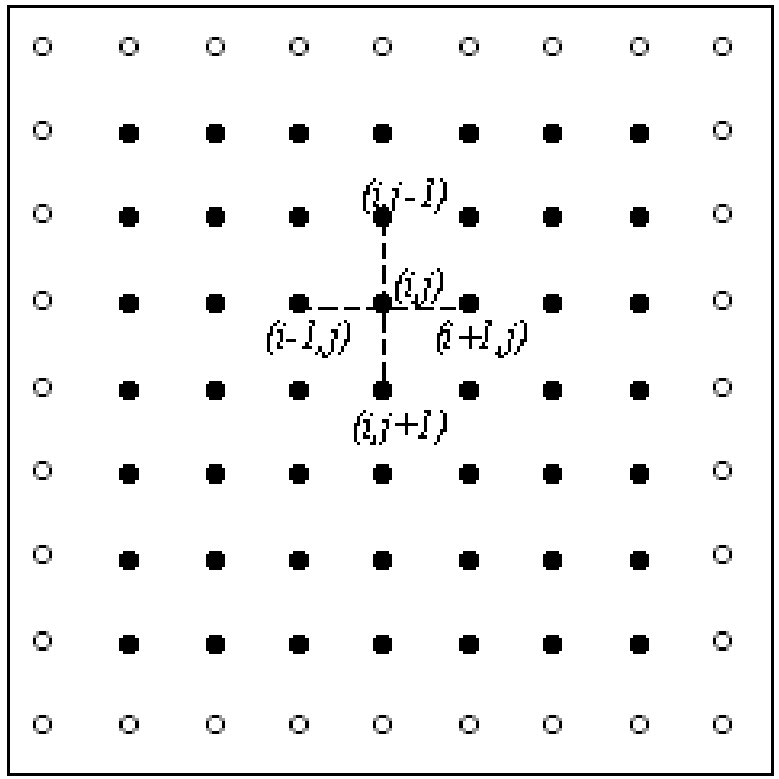
http://www.hpcc.unn.ru/files/HTML_Version/part6.html

Последовательные методы решения задачи Дирихле

Для вычисления значений производных, уравнение Пуассона представлено в *конечно-разностной форме*

$$\frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{ij}}{h^2} = f_{ij}$$

Результат разностного уравнения служит основой для построения различных *итерационных схем* решения задачи Дирихле, в которых в начале вычислений формируется некоторое приближение для значений u_{ij} , а затем эти значения последовательно уточняются в соответствии с приведенным соотношением.



Прямоугольная сетка в области D

Последовательный алгоритм

```
// Алгоритм 6.1
 do {
   dmax = 0; // максимальное изменение значений u
   for ( i=1; i<N+1; i++ )
     for ( j=1; j<N+1; j++ ) {
       temp = u[i][j];
       u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
                       u[i][j-1]+u[i][j+1]-h*h*f[i][j]);
       dm = fabs(temp-u[i][j]);
       if ( dmax < dm ) dmax = dm;
     }
 } while ( dmax > eps );
```

Общее количество итераций метода Гаусса-Зейделя составило 210 при точности решения eps = 0,1 и N = 100. В качестве начального приближения использовались значения, сгенерированные датчиком случайных чисел.

Использование OpenMP

```
// Алгоритм 6.2
omp_lock_t dmax_lock;
omp_init_lock (dmax_lock);
do {
  dmax = 0; // максимальное изменение значений u
  #pragma omp parallel for shared(u,n,dmax) private(i,temp,d)
  for ( i=1; i<N+1; i++ ) {
    #pragma omp parallel for shared(u,n,dmax) private(j,temp,d)
     for ( j=1; j<N+1; j++ ) {
       temp = u[i][j];
       u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
       u[i][j-1]+u[i][j+1]-h*h*f[i][j]);
       d = fabs(temp-u[i][j]);
      omp_set_lock(dmax_lock);
       if ( dmax < d ) dmax = d;
      omp_unset_lock(dmax_lock);
    } // конец вложенной параллельной области
  } // конец внешней параллельной области
} while ( dmax > eps );
```

http://www.hpcc.unn.ru/files/HTML_Version/part6.html

Вариант параллельного алгоритма для метода сеток, если разрешен произвольный порядок пересчета значений u_{ij}

Проблема синхронизации параллельных вычислений

Наличие общих данных обеспечивает возможность взаимодействия потоков.

Разделяемые переменные могут рассматриваться как *общие ресурсы потоков* и, как результат, их использование должно выполняться с соблюдением *правил взаимоисключения* (изменение каким-либо потоком значений общих переменных должно приводить к блокировке доступа к модифицируемым данным для всех остальных потоков).

Разделяемым ресурсом является величина **dmax**, доступ потоков к которой регулируется служебной переменной (*семафором*) **dmax_lock** и функциями **omp_set_lock** (разрешение или блокировка доступа) и **omp_unset_lock** (снятие запрета на доступ).

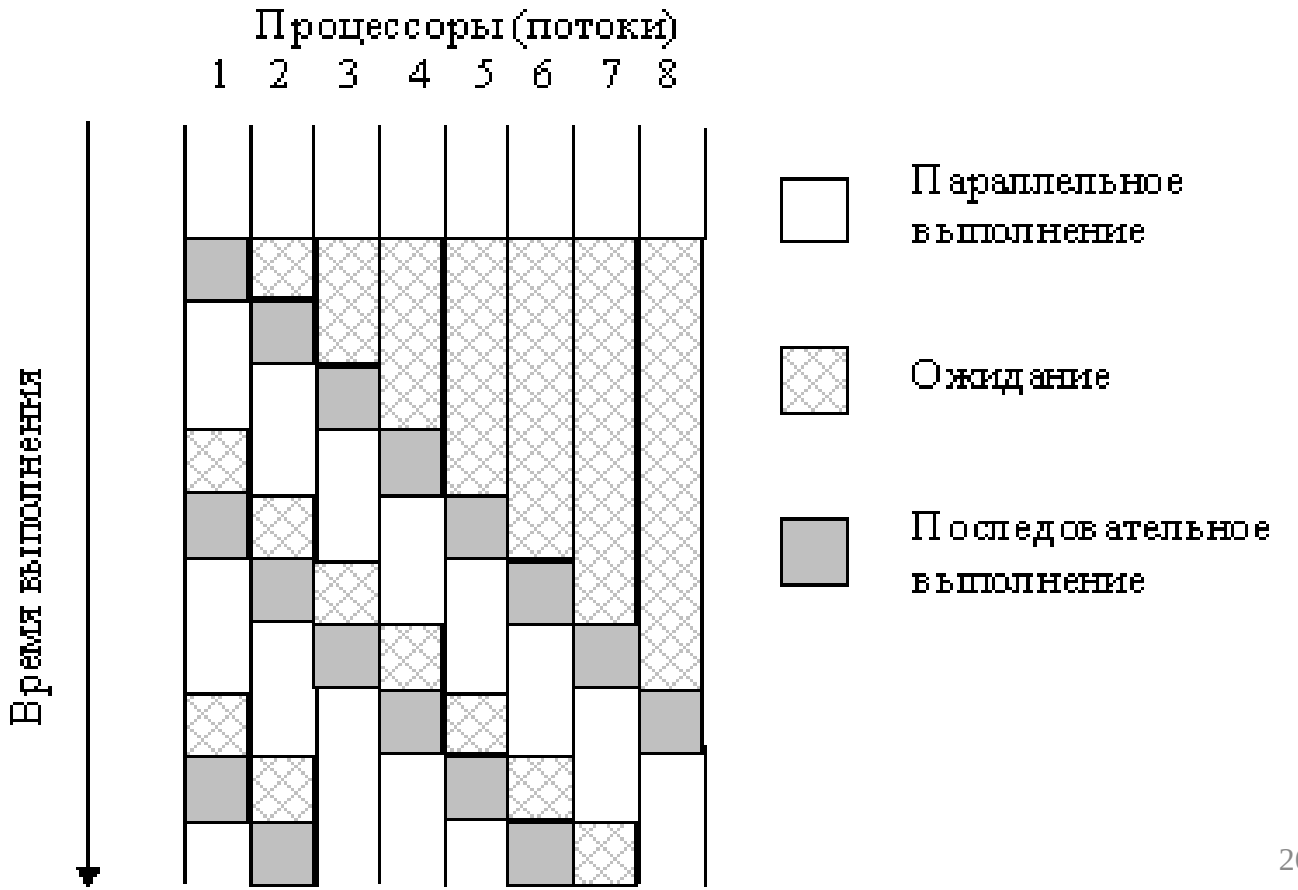
Подобная организация программы гарантирует единственность доступа потоков для изменения разделяемых данных.

Для критических секций между обращениями к функциям **omp_set_lock** и **omp_unset_lock** обеспечивается взаимоисключение.

Результат алгоритма 6.2

Программа работает медленно и ускорение вычислений от использования нескольких процессоров не существенно. Основная причина –
высокая *синхронизация* параллельных участков программы.

Пример возможной схемы выполнения параллельных потоков при наличии синхронизации (взаимоисключения).
Эффект вырождения параллелизма из-за интенсивной синхронизации параллельных участков программы называют *сериализацией* (*serialization*).



Еще раз OpenMP

Для достижения эффективности параллельных вычислений необходимо уменьшить моменты синхронизации.

Можно ограничиться распараллеливанием только одного внешнего цикла **for**.

Для снижения количества возможных блокировок для оценки максимальной погрешности использована многоуровневая схема расчета:

- параллельно выполняемый поток первоначально формирует локальную оценку погрешности **dm** только для своих обрабатываемых данных (одной или нескольких строк сетки),
- затем при завершении вычислений поток сравнивает свою оценку **dm** с общей оценкой погрешности **dmax**.

```
// Алгоритм 6.3
omp_lock_t dmax_lock;
omp_init_lock(dmax_lock);
do {
  dmax = 0; // максимальное изменение значений u
#pragma omp parallel for shared(u,n,dmax)\
                         private(i,temp,d,dm)
  for ( i=1; i<N+1; i++ ) {
    dm = 0;
    for ( j=1; j<N+1; j++ ) {
      temp = u[i][j];
      u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
                      u[i][j-1]+u[i][j+1]-h*h*f[i][j]);
      d = fabs(temp-u[i][j])
      if ( dm < d ) dm = d;
    }
    omp_set_lock(dmax_lock);
      if ( dmax < dm ) dmax = dm;
    omp_unset_lock(dmax_lock);
    }
  } // конец параллельной области
} while ( dmax > eps );
```

Результаты вычислительных экспериментов

Размер сетки	Последовательный метод Гаусса-Зейделя (алгоритм 6.1)		Параллельный алгоритм 6.2			Параллельный алгоритм 6.3		
	k	*t*	*k*	*t*	*S*	*k*	*t*	*S*
100	210	0,06	210	1,97	0,03	210	0,03	2,03
200	273	0,34	273	11,22	0,03	273	0,14	2,43
300	305	0,88	305	29,09	0,03	305	0,36	2,43
400	318	3,78	318	54,20	0,07	318	0,64	5,90
500	343	6,00	343	85,84	0,07	343	1,06	5,64
600	336	8,81	336	126,38	0,07	336	1,50	5,88
700	344	12,11	344	178,30	0,07	344	2,42	5,00
800	343	16,41	343	234,70	0,07	343	8,08	2,03
900	358	20,61	358	295,03	0,07	358	11,03	1,87
1000	351	25,59	351	366,16	0,07	351	13,69	1,87
2000	367	106,75	367	1585,84	0,07	367	56,63	1,89
3000	370	243,00	370	3598,53	0,07	370	128,66	1,89

k – количество итераций, t – время в сек, S – ускорение.

Использован четырехпроцессорный сервер кластера Нижегородского университета с процессорами Pentium III, 700 Mhz, 512 RAM

Проблемы и решения

- Возможность неоднозначности вычислений в параллельных программах (состязание потоков).
- Проблема взаимоблокировки.
- Исключение неоднозначности вычислений (единственность решения).
- Волновые схемы параллельных вычислений (волны вспомогательной диагональной сетки).
- Балансировка вычислительной нагрузки процессоров.

http://www.hpcc.unn.ru/files/HTML_Version/part6.html