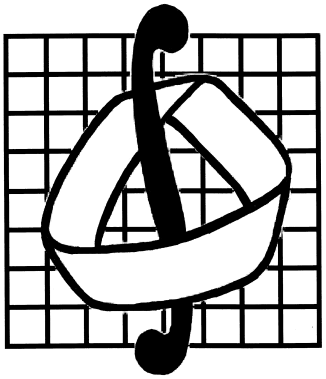
МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М. В. ЛОМОНОСОВ
Механико-математический факультет



Решение линейной системы уравнений методом Гаусса с выбором главного элемента по всей матрице. Блочный вариант алгоритма. Асимптотическая оценка сложности.Распараллеливание алгоритма.

Ковальков Максим,304 группа

Содержание

1. Постановка задачи **3**

2. Алгоритм **3**
2.1. Входные данные . . . 3
2.2. Неблочный случай . . . 3
2.3. Блочный вариант алгоритма . . . 5
2.4. Функции работы с блоками . . . 6
2.5. Точная оценка работы алгоритма . . . 7
2.5.1. Сложность поиска опорного элемента . . . 8
2.5.2. Сложность прямого хода алгоритма Гаусса . . . 8
2.5.3. Итоговая сложность . . . 8
2.6. Случаи $m = n$ и $m = 1$ . . . 8
2.6.1. m=1 . . . 8
2.6.2. m=n . . . 9

3. Распараллеливание алгоритма (с помощью multithread) **9**
3.1. Распараллеливание поиска опорного элемента . . . 9
3.2. Распараллеливание прямого хода Гаусса . . . 10

3.3. Оценка сложности и числа точек синхронизации 10
3.3.1. Сложность выбора лидирующего элемента 11
3.3.2. Сложность прямого хода алгоритма Гаусса 11
3.3.3. Итоговая сложность для параллельного алгоритма 11
3.3.4. Оценка числа точек синхронизации алгоритма 12

4. Расспараллеливание алгоритма (с помощью MPI) 12
4.1. Чтение и хранение матрицы в памяти 12
4.2. Расспараллеливание алгоритма 13
4.3. Сложность, число точек синхронизации, число и объем пересылок . . 14
4.3.1. Сложность выбора лидирующего элемента 14
4.3.2. Сложность прямого хода алгоритма Гаусса 15
4.3.3. Итоговая сложность для параллельного алгоритма 15
4.3.4. Оценка числа точек синхронизации алгоритма 15
4.3.5. Количество и объём пересылок данных 15

1. Постановка задачи

Дана система линейных уравнений, n неизвестных, нужно найти ее решение $\overrightarrow{x}=(x_1,\ldots,x_n)$.
Метод решения: Метод Гаусса с выбором по всей матрице.

2. Алгоритм

2.1. Входные данные

Сиситему из n линейных уравнений записываем в матричном виде:

$$A = \left(\begin{array}{ccc|c} a_{11} & \ldots & a_{1n} & b_1 \\ \vdots & \ddots & \vdots & \vdots \\ a_{n1} & \ldots & a_{nn} & b_n \end{array}\right)$$

$\exists$ Два способа получения этой матрицы:

1) Элементы матрицы задаются некотрой заранее известной формулой.

2) Матрица считывается из некоторого текстового файла.

Замечание: Во втором случае в первой строке входного файла будет содержаться размерность матрицы, Затем записана непосредственно сама матрица.

2.2. Неблочный случай

Найдем в матрице A наибольший по модулю элемент a_{ij} Поменяем местами i-ю строку с первой иj-й столбец с первым. Эти операции можно сделать последовательным обменом элементов. На данном этапе нам потребуется n^2 операций для выбора ведущего элемента и $2n$ операций для обмена местами строк и столбцов. Если $a_{11}<0$, то множим превую строку на -1. Поделим первую строку на $k=a_{11}$.Для этого поченно поделим каждый элемент первой строки: $a_{1i} = \frac{a_{1i}}{k}, i = \overline{1,n}; b_1 = \frac{b_1}{k}$

$$\tilde{A} = \left(\begin{array}{ccc|c} 1 & \ldots & \frac{a_{1n}}{k} & \frac{b_1}{k} \\ \vdots & \ddots & & \vdots \\ a_{n1} & \ldots & a_{nn} & b_n \end{array}\right)$$

Замечание: Менять столбцы просто так нельзя. Для этого нужно завести специальный массив индексов p, такой что изначально $p[i] = i$, а при каждой перестанове столбцов происходит соответсвующая перестановка элементов этого массива. Это же замечание переносится на блочный вариант алгоритма.

Далее из каждой строки вычтем первую строку с соответствующем коэффициентом: $a_{ij} = a_{ij} - a_{i1}a_{1,j}; b_i = b_i - a_{i1}b_1, i, j = \overline{2,n}$
На этом этапе получим матрицу:

$$\tilde{\tilde{A}} = \left(\begin{array}{ccc|c} 1 & \tilde{a_{12}} \dots & \tilde{a_{1n}} & \tilde{b_1} \\ 0 & \tilde{a_{22}} \dots & \tilde{a_{2n}} & \tilde{b_2} \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \tilde{a_{n2}} \dots & \tilde{a_{nn}} & \tilde{b_n} \end{array}\right)$$

Эту матрицу можно записать в виде:

$$\tilde{\tilde{A}} = \left(\begin{array}{c|ccc} 1 & \tilde{a_{12}} \dots & \tilde{a_{1n}} & \tilde{b_1} \\ \hline 0 & & & \\ \vdots & & A1 & \\ 0 & & & \end{array}\right)$$

В итоге получили матрицу $A1$, устроенную также, как матрица A. Проделываем с матрицей $A1$ те же действия, что и с A. Повторяя данную операции n раз, получаем матрицу:

$$A = \left(\begin{array}{ccc|c} 1 & a_{12} \dots & a_{1n} & b_1 \\ 0 & 1 \dots & a_{2n} & b_2 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & 0 \dots & 1 & b_n \end{array}\right)$$

Замечание:Сложность этой части алгоритма составляет $\sim 1.8O(n^3)$
Дальнейшим продолжением этого алгоритма является,так называемый, обратный ход Гаусса. Вычитаем из всех строк последню строку с подходящими коэффициентами: $b_i = b_i - a_{in} * b_n;\ a_{in} = 0, i = \overline{1, n-1}$, остальные коэффициенты остаются без изменений. После этих действий получаем матрицу:

$$\tilde{A} = \left(\begin{array}{ccc|c} 1 & a_{12} \dots & 0 & \tilde{b_1} \\ 0 & 1 \dots & 0 & \tilde{b_2} \\ \vdots & \ddots & \vdots & \vdots \\ 0 & 0 \dots & 1 & \tilde{b_n} \end{array}\right)$$

После этого вычетаем из первых n-2 строк полученной матрицы n-1 строку этой матрицы $b_i = b_i - a_{in-1}b_{n-1};\ a_{in-1} = 0,\ i = \overline{1, n-2}$, остальные коэффициенты- неизменны. Продолжаем данные операции по общей формуле для j-го шага: $b_i = b_i - a_{in-j+1}b_{n-j+1};\ a_{in-j+1} = 0,\ i = \overline{1, n-j}$
В итоге получаем единичную матрицу. Столбец свободных членов $\vec{b}$-решение данной системы.
Замечание:Сложность этой части алгоритма составляет $O(n^2)$

2.3. Блочный вариант алгоритма

Для начала введем матричную норму.

Определение:Пусть A-матрица $n \times n$, тогда число $||A|| = \max\limits_{i=1}^{n} \sum\limits_{j=1}^{n} |a_{ij}|$ называется нормой матрицы A.

Замечание:В блочном случае нам нужен еще один параметр m-размер блока.
Мы разбивае матрицуA на блоки $m \times m$. В этом случае матрица имеет вид:

$$\tilde{A} = \left(\begin{array}{cccc|c} A_{11}^{m\times m} & \dots & A_{1k-1}^{m\times m} & A_{1k}^{m\times p} & \tilde{B_1^m} \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ A_{k-1,1}^{m\times m} & \dots & A_{k-1,k-1}^{m\times m} & A_{k-1,k}^{m\times p} & \tilde{B_1^m} \\ A_{k1}^{p\times m} & \dots & A_{kk-1}^{p\times m} & A_{kk}^{p\times p} & \tilde{B_k^p} \end{array}\right)$$

где $k = \lceil \frac{n}{m} \rceil$, а p-остаток от деления n на m.
В этом варианте алгоритма нужно в качестве опорного элемента выбирать не наибольший по модулю элемент, а тот блок, обратная матрица которого имеет наименьшую норму. Из этого факта напрямую следует перый этап каждаго шага алгоритма-считаем норму обратной матрицы каждого блока и выбираем наименьший(выбор происходит только среди основных блоков размера $m \times m$). Сложность этого этапа составляет $O(k^2 \times m^3)$

Замечание: Из-за того, что не каждая матрица имеет обратную следует естественное органичение на работу данного алгоритма. В отличии от неблочного варианта, он применим не во всех случаях.
Пусть на первом шаге лидирующий элемент-$A_{ij}^{m\times m}$, поменяем местами i -ю строку с первой и j-й столбец с первым. Затем умножим первую строку на $C = (A_{ij}^{m\times m})^{-1}$. Получим следующую матрицу:

$$\tilde{A} = \left(\begin{array}{cccc|c} E^{m\times m} & \dots & C * A_{1k-1}^{m\times m} & C * A_{1k}^{m\times p} & C * \tilde{B_1^m} \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ A_{k-1,1}^{m\times m} & \dots & A_{k-1,k-1}^{m\times m} & A_{k-1,k}^{m\times p} & \tilde{B_1^m} \\ A_{k1}^{p\times m} & \dots & A_{kk-1}^{p\times m} & A_{kk}^{p\times p} & \tilde{B_k^p} \end{array}\right)$$

Теперь обнулим первый столбец:
$A_{i,j}^{m\times m} = A_{i,j}^{m\times m} - A_{i,1}^{m\times m} * A_{1,j}^{m\times m}, i = 1, \dots, k-1, j = 1, \dots, k-1$
$A_{i,k}^{m\times p} = A_{i,k}^{m\times p} - A_{i,1}^{m\times m} * A_{1,k}^{m\times p}, i = 1, \dots, k-1$
$A_{k,i}^{p\times m} = A_{k,i}^{p\times m} - A_{k,1}^{p\times m} * A_{1,i}^{m\times p}, i = 1, \dots, k-1$
$A_{k,k}^{p\times p} = A_{k,k}^{p\times p} - A_{k,1}^{p\times m} * A_{1,k}^{m\times p}$
$B_i^{m\times 1} = B_i^{m\times 1} - A_{i,1}^{m\times m} * B_1^{m\times 1}, i = 1, \dots, k-1$
$B_k^{p\times 1} = B_k^{p\times 1} - A_{k,1}^{p\times m} * B_1^{m\times 1}$

Получили матрицу:

$$\tilde{\tilde{A}} = \left(\begin{array}{cccc|c} E^{m\times m} & \dots & \tilde{A^{m\times m}_{1k-1}} & \tilde{A^{m\times p}_{1k}} & B^m_1 \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & \dots & \tilde{A^{m\times m}_{k-1,k-1}} & \tilde{A^{m\times p}_{k-1,k}} & B^m_1 \\ 0 & \dots & \tilde{A^{p\times m}_{kk-1}} & \tilde{A^{p\times p}_{kk}} & B^p_k \end{array}\right)$$

Замечание:Сложность одного этого шага $O(m^3 \times k^2)$
Далее повторяем эти действия с нижней подматрицей k-2 раза. Получаем верхне-диагональную матрицу. Далее применяем обратный ход метода Гаусса, практически идентичный неблочному случаю. Как и тогда обратный ход дает значительно меньшую сложность.
Замечание:Итоговая сложность блочного алгоритма $\sim O(m^3 \times k^3) = O(n^3)$

2.4. Функции работы с блоками

Нужно определить две функции,для работы с вектором свободных членов:

```
void getvec(double*b,double*b1,int n,int m,int i1)
{
  int n1=m;
  if(i1==n/m){n1=n%m;}
  for(int l=0;l<n1;l++)
  {
    b1[l]=b[i1*m+l];
  }

}
void pushvec(double*b,double*b1,int n,int m,int i1)
{
 int n1=m;
 if(i1==n/m){n1=n%m;}
 for(int l=0;l<n1;l++)
 {
  b[i1*m+l]=b1[l];
 }
}
```

И две функции для работы с блоками матрицы:

```
void get(double*a,double*c,int n,int m,int i1,int j1)
{
 int n1=m;int m1=m;
```

```
 if(i1==n/m){n1=n%m;}
 if(j1==n/m){m1=n%m;}
 for(int i=0;i<n1;i++)
 {
  for(int j=0;j<m1;j++)
  {
   c[i*m1+j]=a[i1*n*m+j1*m+i*n+j];
  }
 }

}
void push(double*a,double*c,int n,int m,int i1,int j1)
{
 int n1=m;int m1=m;
 if(i1==n/m){n1=n%m;}
 if(j1==n/m){m1=n%m;}
 for(int i=0;i<n1;i++)
 {
  for(int j=0;j<m1;j++)
  {
   a[i1*n*m+j1*m+i*n+j]=c[i*m1+j];
  }
 }

}
```

Комментарий к функциям: Для работы с блоками матрицы нам необходимы 3 буфера обмена, в которых заносятся блоки матрицы, происходят операции умножения и сложения блоков, перезапись блоков матрицы. Первым этапом в каждой из функций является определение размеров соответсвующего блока, по входным параметрам i, j. Если оба параметра меньше n/m ,то блок имеет размер $m \times m$, если $i = n/m$ $j < n/m$,то $(n\%m) \times m$, если $j = n/m$ $i < n/m$,то $m \times (n\%m)$, иначе $(n\%m) \times (n\%m)$ блоков. Далее в соответствии с фонкцией мы либо заполняем буфер из массива, либо записываем значения буфера в массив.

Замечание:Для практической реализации нужно заменить 1-индексацию, использованную при математической работе с матрицами на 0-индексацию, принятую в нумерации массивов в C++

2.5. Точная оценка работы алгоритма

По ходу изложения блочного варианта алгоритма Гаусса было сделано несколько асимптотических оценок сложности частей алгоритма. В этом пункте соберем их воедино и дадим оценку сложности всего алгоритма.

2.5.1. Сложность поиска опорного элемента

Нахождение обратной матрицы размера $m \times m$ имеет сложность $\frac{8}{3} * m^3 + O(m^2)$
На i-м шаге алгоритма нужно найти максимум из $([\frac{n}{m}]-i+1)^2$ блоков размера $m \times m$. Нахождение нормы матрицы имеет квадратичную сложность и, как следствие, не оказывает влияние на значиние коэффициента амортизации главного члена асимптотической оценки сложности. Итого суммарная сложность этой части алгоритма составляет:

$$(\frac{8}{3}m^3 + O(m^2)) \sum_{i=1}^{[\frac{n}{m}]} ([\frac{n}{m}] - i + 1)^2 = \frac{8}{3}m^3 \times \frac{1}{3}[\frac{n}{m}]^3 + O(m \times n^2) = \frac{8}{9}n^3 + O(m \times n^2)$$

2.5.2. Сложность прямого хода алгоритма Гаусса

Обратная матрица находится $[\frac{n}{m}]$ раз, эти оперции имеют суммарную сложность $\frac{8}{3}[\frac{n}{m}]m^3 + O(n \times m)$. На i-м шаге алгоритма происходит $[\frac{n}{m}] - i$ операций умножения элементов строки на обратную и $[\frac{n}{m}]^2 - i$ операций перемножения элементов i-й строки на ведущие элементы нижних строк, также происходит вычитание блоков, но оно имеет квадртичную сложность. Итого, этот этап имеет сложность:

$$\frac{8}{3}[\frac{n}{m}]m^3 + 2m^3 \sum_{i=1}^{[\frac{n}{m}]-1} (([\frac{n}{m}] - i)^2 + [\frac{n}{m}] - i) + O(m \times n^2) = \frac{2n^3}{3} + O(m \times n^2)$$

Замечание:Очевидно,что сложность обратного хода алгоритма Гаусса войдет в $O(m \times n^2)$

2.5.3. Итоговая сложность

Суммируя сложности, полученные в двух прошлых пунктах, получаем

$$\boxed{\frac{14}{9}n^3 + O(m \times n^2)}$$

Замечание: Интересным фактом является то, что наиболее затратным этапом данного алгоритма будет по сути подготовительный этап: поиска матрицы с минимальной нормой обратной к ней матрицы.

2.6. Случаи $m = n$ и $m = 1$

2.6.1. m=1

Данный случай практически идентичен неблочному алгоритму, но будет работать несколько дольше, поскольку в ходе своего выполнения множество раз вызывает функцию поиска обратной и пермножения матриц.
Подставляя $m = 1$ в общую формулу, имеем сложность: $\frac{14}{9}n^3 + O(m \times n^2)$

2.6.2. m=n

Этот случай считается значительно более трудно, т.к. нужно учесть все члены оценки имеющие множители m^3. Если это аккуратно сделать, то получим $\sim \frac{145}{18}n^3 + O(n^2)$
Замечание: Блочный алгоритм в этом последнем случае существенно проигрывает своему линейному анологу.

3. Распараллеливание алгоритма (с помощью multithread)

В параллельном варианте алгоритма с командной строки подается еще один параметр p– количество потоков. При $p = 1$ алгоритм по сути является нераспараллеленым. При $p =$*количество ядер машины* достигается максимальная производительность параллельного алгоритма. Также предварительно скажем, что если после деления на блоки есть вырожденная строка, то ее всегда обрабатывает нулевой поток (основной, из которого созданы в се остальные).

3.1. Распараллеливание поиска опорного элемента

Замечание:Поиск опорного элемента-это начало i-го шага прямого хода Гаусса. Для удобства описание расспараллеливания прямого хода разбито на два пункта.

Как можно видеть из оценок сложности, данных выше, большую часть вычислительной сложности алгоритма дает этап поиска опорных элементов при прямом ходе блочного алгоритма Гаусса. Кажется естественным, что именно этот этап должен быть распараллелен в первую очередь.
Итак, рассмотрим i-й шаг алгоритма $i = \overline{0, k-1}$, где $k = [\frac{n}{m}]$. t-й поток обрабатывает строки блоков с номерами j такими, что $j \equiv t \pmod p$. Каждый поток производит в строках блоков с соответсвующими номерами следующие операции: обращает блоки этой строки, вычисляет норму обратных матриц, выбирает минимальный блок, запоминая при этом его норму и координаты $i1, j1$ данного блока. После этого основной поток производит поиск лидера по всей матрице, среди лидеров потоков, сравнивая их нормы. Этот лидер имеет координаты $i2, j2$. Заметим, что для выполнения этого шага нужно дождаться завершения всех потоков. Таким образом получае k точек синхронизации. Далее идет обмен строк и столбцов матрицы, который происходит таким образом, что $(i2, j2)$-й элемент займет место (i, i). Этап обмена строк и столбцов распараллеливается сходным образом, причем при перестановке строк должны происходить соотвествющие перестановки в массиве индексов. Однако из-за трудностей в работе с памятью лучше оставить этот этап алгоритма в главном потоке. Этот обмен дает в общем случае $2k$ точек синхронизации.

3.2. Распараллеливание прямого хода Гаусса

Из прежних оценок сложности, мы знаем, что прямой ход Гаусса также является одним из наиболее вычислительно тяжелых участков алгоритма. Сначала в родительском потоке умножим i-ю строку на $C = (A_{i,i}^{m\times m})^{-1}$
Этот этап аналогичен соответсвующему этапу для последовательного алгоритма. Например, при $i = 0$ полчим матрицу следующего вида:

$$\tilde{A} = \left(\begin{array}{cccc|c} E^{m\times m} & \dots & C * A_{0k-2}^{m\times m} & C * A_{0k-1}^{m\times p} & C * \tilde{B_0^m} \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ A_{k-2,0}^{m\times m} & \dots & A_{k-2,k-2}^{m\times m} & A_{k-2,k-1}^{m\times p} & \tilde{B_0^m} \\ A_{k-1,0}^{p\times m} & \dots & A_{k-1,k-2}^{p\times m} & A_{k-1,k-1}^{p\times p} & B_{k-1}^{\tilde{p}} \end{array}\right)$$

Вообще говоря, данный этап можно тоже распараллелить, передавая каждому потоку матрицу C. Распараллеливание производится аналогично: t-й поток работает со столбцами, имеющими номер, кратный t.
Следющим шагом алгоритма вычтем из каждой строки,с номером, превосходящим i, i-ю строку, умноженную на i-й элемент данной строки. Этот шаг - основной с точки зрения вычислительной сложности, его мы и распределяем между потоками, каждый из которых вычитает из строк, за которые отвечает, i-ю. Заметим, что обращение разных потоков к одной памяти может привести к различным коллизиям. Поэтом сделаем $p-1$ копию i-й строки и передадим их потокам. Ожидание копирования дает нам k точек синхронизации. Приведем формулы подсчета новых блоков в t-м потоке:

$$A_{i1,j}^{m\times m} = A_{i1,j}^{m\times m} - A_{i1,i}^{m\times m} * A_{i,j}^{m\times m}$$
$$A_{i1,k}^{m\times p} = A_{i1,k}^{m\times p} - A_{i1,i}^{m\times m} * A_{i,k}^{m\times p}$$
$$B_{i1}^{m\times 1} = B_{i1}^{m\times 1} - A_{i1,i}^{m\times m} * B_{i}^{m\times 1}$$

Где i- шаг алгоритма, а $i1 \equiv t \pmod p$.
Отметим, что мы не можем перейти к следующему шагу алгоритма, пока все потоки не закончат работу, таким образом получаем еще k точек синхронизации.
После окончания прямого хода алгоритма Гаусса, в основном потоке обработаем "нецелую"строку, если таковая имеется. Затем идет обратный ход Гаусса. Его можно оставить в последовательном варианте, т.к. это вычислительно-легкий этап алгоритма, он не дает основной вклад в асимптотическую сложность.

3.3. Оценка сложности и числа точек синхронизации

Предварительно скажем, что помимо главного члена в оценке асимптотической сложности параллельного алгоритма n^3 большое значение играют также слагаемые

$n^2 * m$ и nm^2. Это происходит из-за того, что мы распараллеливаем не все этапы алгоритма. Поэтому все оценки будем проводить с точностью до $o(n^3+n^2m+nm^2)$.

3.3.1. Сложность выбора лидирующего элемента

Нахождение обратной матрицы размера $m \times m$ имеет сложность $\frac{8}{3} * m^3 + o(m^2)$ На i-м шаге алгоритма нужно найти максимум из $([\frac{n}{m}]-i)^2$ блоков размера $m \times m$. Нахождение нормы матрицы имеет квадратичную сложность и, как следствие, не оказывает влияние на значиние коэффициента при главном члене. асимптотической оценки сложности. Каждый поток работает с $\frac{1}{p} \times [\frac{n}{m}]$ строками. Тогда сложность этого этапа составляет:

$$\frac{8m^3}{3p} \times [\frac{n}{m}] \sum_{i=0}^{[\frac{n}{m}]-1} ([\frac{n}{m}]-i)^2 = \frac{8n^3}{9p} + \frac{4n^2m}{3p} + \frac{4nm^2}{9p} + o(n^3+n^2m+nm^2)$$

3.3.2. Сложность прямого хода алгоритма Гаусса

Нормирование строки на i-м шаге алгоритма обойдется нам в $m^3 \times [\frac{n}{m}] - i$ операций. Эта операция не распараллелена. Тогда по всему алгоритму получаем

$$m^3[\frac{n}{m}] \sum_{i=1}^{[\frac{n}{m}]-1} ([\frac{n}{m}]-i) = \frac{n^2m-nm^2}{2} + o(n^3+n^2m+nm^2)$$

Далее идет распараллеленная часть алгоритма. Сложность поиска обратного к нальному элементу строки:

$$\frac{8m^3}{3p} \sum_{i=1}^{[\frac{n}{m}]} ([\frac{n}{m}]-i) = \frac{4n^2m}{3p}$$

На i-м шаге алгоритма в каждом потоке происходит $\frac{1}{p}([\frac{n}{m}]-i)^2$ операций перемножения элементов i-й строки на ведущие элементы нижних строк, также происходит вычитание блоков, но оно войдет в $o(n^3+n^2m+nm^2)$. Итого, этот этап имеет сложность:

$$\frac{2m^3}{p} \sum_{i=1}^{[\frac{n}{m}]-1} ([\frac{n}{m}]-i)^2 + o(n^3+n^2m+nm^2) = \frac{2n^3}{3p} - \frac{n^2m}{p} + \frac{nm^2}{3p} + o(n^3+n^2m+nm^2)$$

3.3.3. Итоговая сложность для параллельного алгоритма

Аккуратно сложив оценки каждого шага получим:

$$\boxed{\frac{14}{9p}n^3 + (\frac{7}{3p}+\frac{1}{2})n^2m + (\frac{7}{9p}+\frac{13}{6})nm^2 + o(n^3+n^2m+nm^2)}$$

Стоит заметить, что коэффициенты при n^2m и nm^2 поделились на p, таким образом чистого ускорения параллельного алгоритма в p раз добиться не удалось.

3.3.4. Оценка числа точек синхронизации алгоритма

Поговорим про число точек синхронизации. Собрав оценки, данные по ходу повествования в двух предыдущих пунктах, полчаем $5k$ точек синхронизации.

4. Расспараллеливание алгоритма (с помощью MPI)

Основным отличием MPI- программы является разделение памяти между потоками, в то время, как в multithread-программе потоки работают с общей памятью. Опишим необходимые изменения в реализации, учитывающие эту особенность.

4.1. Чтение и хранение матрицы в памяти

Выделяем в каждом потоке память в размере $8m \times n \times ([\frac{[\frac{n}{m}]}{p}] + 1)$ байт. Помимо этого в нулевом потоке выделим еще $n \times m$ байт. Это нужно из-за того, что в памяти нулевого потока мы собираемся хранить последний неполноразмерный столбец, если он существует, и столбец свободных членов, кроме того свободная память нулевого потока понадобится во время процесса чтения матрицы. Как эта память будет заполняться? Мы хотим, чтобы память t-го потока была заполнена данными из столбцов исходной матрицы, номера которых, j удовлетворяют условию $j \equiv t \pmod p$

Опишем процедуру считывания матрицы.На i-м шаге алгоритма считываем i-ю блочную строку матрицы в дополнительную память нулевого потока, рассылаем блоки строки соответсвующим потокам, затем переходим к следующему шагу алгоритма.

Итак, после процедуры считывания матрицы в памяти t-го потока хранится матрица:

$$A = \begin{pmatrix} A_{0,t}^{m\times m} & A_{0,t+p}^{m\times m} & \dots & A_{0,w_t}^{m\times m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{k,t}^{ost\times m} & A_{k,t+p}^{ost\times m} & \dots & A_{k,w_t}^{ost\times m} \end{pmatrix}$$

Где $ost = n - [\frac{n}{m}] * m$, $w_t = \max\limits_{\substack{j \equiv t \pmod p \\ j < k}} j$

Замечание:В записи этой матрицы мы использовали координаты исходной блочной матрицы (глобальные координаты), однако сама матрица хранится в памяти потока в локальных координатах. Во избежание путаницы напишем функции перехода из одной системы координат в другую. Пусть (i, j)-глобальные координаты, а (i', j')-локальные координаты для t-го потока. Тогда координаты связаны следующим образом:

$$\begin{cases} i = i' \\ j = t + j' \times p \end{cases}$$

Для удобной работы нужно изменить функции **get(...)** и **p**ush(...), предварительно

перейдя из глобальной системы в локальную (ведь каждый поток имеет доступ только к своей памяти). Тогда получим:

```
void get(double*a,double*c,int n,int m,int i1,int j1,int p,int t)
{
 j1=(j1-t)/p;
 int n1=m;int m1=m;
 if(i1==n/m){n1=n%m;}
 if(j1==n/m){m1=n%m;}
 for(int i=0;i<n1;i++)
 {
  for(int j=0;j<m1;j++)
  {
   c[i*m1+j]=a[i1*n*m+j1*m+i*n+j];
  }
 }

}
void push(double*a,double*c,int n,int m,int i1,int j1,int p,int t)
{
 j1=(j1-t)/p;
 int n1=m;int m1=m;
 if(i1==n/m){n1=n%m;}
 if(j1==n/m){m1=n%m;}
 for(int i=0;i<n1;i++)
 {
  for(int j=0;j<m1;j++)
  {
   a[i1*n*m+j1*m+i*n+j]=c[i*m1+j];
  }
 }

}
```

4.2. Расспараллеливание алгоритма

Рассматриваем q-й шаг алгоритма.
Каждый поток ищет в своей локальной матрице блок с наименьшей нормой обратного. Ждем, пока все потоки выполнят работу по поиску своего лидера, для этого применяем функцию `MPI_Allreduce(...)`. Из лидеров потоков выбираем опорный блок всей матрицы. Посылаем всем потокам номер строки i_0, в которой находится опорный, далее каждый поток меняет местами строки q и i_0 в своей локальной матрице, вновь применяем `MPI_Allreduce(...)` . Затем пересылаем столбец j_0, в котором находится опорный элемент всем потокам с помощью `MPI_Bcast(...)`. В

потоке, который хранит q-й столбец посылаем его потоку с j_0 и меняем эти столбцы местами. После этого используем `MPI_Allreduce(...)`.
Следующим этапом алгоритма является нормирование строки q. Эта часть алгоритма расспараллеливается (в отличии от прошлого варианта распараллеливания, где потоки работали со строками матрицы). Например при $q = 0$ полчаем матрицу:

$$\tilde{A} = \left(\begin{array}{cccc|c} E^{m\times m} & \dots & C * A^{m\times m}_{0k-2} & C * A^{m\times p}_{0k-1} & C * \tilde{B^m_0} \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ A^{m\times m}_{k-2,0} & \dots & A^{m\times m}_{k-2,k-2} & A^{m\times p}_{k-2,k-1} & \tilde{B^m_0} \\ A^{ost\times m}_{k-1,0} & \dots & A^{ost\times m}_{k-1,k-2} & A^{ost\times p}_{k-1,k-1} & \tilde{B^o_{k-1}st} \end{array}\right)$$

Нормировка ведущей строки также требует точки синхронизации. Нормировка проводится по следующей формуле:

$$A^{m\times m}_{q,j} = (B^{m\times m}_q)^{-1} * A^{m\times m}_{q,j}$$

Осталось вычесть из всех строк с номером, превосходящим q, q-ю строку, умноженный на нужный элемент буфера. Приведем формулы пересчета матрицы A, входе прямого хода алгоритма Гаусса

$$A^{m\times m}_{i,j} = A^{m\times m}_{i,j} - B^{m\times m}_i * A^{m\times m}_{q,j}$$
$$A^{ost\times m}_{k,j} = A^{ost\times m}_{k,j} - B^{ost\times m}_k * A^{m\times m}_{q,j}$$

Аналогичные преобразования нужно провести также со столбцом свободных членов.
Замечание:Пересылки в обратном ходе Гаусса ограничиваются пересылками столбца свободных членов. Сам алгоритм можно оставить последовательным: когда работаем с i-м столбцом, то соответсвующий поток(в которм этот столбец лежит) работает, а остальные простаивают. Можно применить и параллельную реализацию, но это вычислительно нетрудоекий этап, поэтому оставление его последовательным-допустимо.

4.3. Сложность, число точек синхронизации, число и объем пересылок

4.3.1. Сложность выбора лидирующего элемента

Нахождение обратной матрицы размера $m \times m$ имеет сложность $\frac{8}{3} * m^3 + o(m^2)$
На i-м шаге алгоритма нужно найти максимум из $([\frac{n}{m}] - i)^2$ блоков размера $m \times m$. Нахождение нормы матрицы имеет квадратичную сложность и, как следствие, не оказывает влияние на значиние коэффициента при главном члене асимптотической оценки сложности. Каждый поток работает с $\frac{1}{p} \times [\frac{n}{m}]$ строками. Тогда сложность

этого этапа составляет:

$$\frac{8m^3}{3p} \times [\frac{n}{m}] \sum_{i=0}^{[\frac{n}{m}]-1} ([\frac{n}{m}] - i)^2 = \frac{8n^3}{9p} + \frac{4n^2m}{3p} + \frac{4nm^2}{9p} + o(n^3 + n^2m + nm^2)$$

4.3.2. Сложность прямого хода алгоритма Гаусса

Нормирование строки на i-м шаге алгоритма обойдется нам в $m^3 \times [\frac{n}{m}] - i$ операций. Эта операция распараллелена. Тогда по всему алгоритму получаем

$$\frac{1}{p} m^3 [\frac{n}{m}] \sum_{i=1}^{[\frac{n}{m}]-1} ([\frac{n}{m}] - i) = \frac{n^2m - nm^2}{2p} + o(n^3 + n^2m + nm^2)$$

Сложность поиска обратного к начальному элементу строки:

$$\frac{8m^3}{3p} \sum_{i=1}^{[\frac{n}{m}]} ([\frac{n}{m}] - i) = \frac{4n^2m}{3p}$$

На i-м шаге алгоритма в каждом потоке происходит $\frac{1}{p}([\frac{n}{m}] - i)^2$ операций перемножения элементов i-й строки на ведущие элементы нижних строк, также происходит вычитание блоков, но оно войдет в $o(n^3 + n^2m + nm^2)$. Итого, этот этап имеет сложность:

$$\frac{2m^3}{p} \sum_{i=1}^{[\frac{n}{m}]-1} ([\frac{n}{m}] - i)^2 + o(n^3 + n^2m + nm^2) = \frac{2n^3}{3p} - \frac{n^2m}{p} + \frac{nm^2}{3p} + o(n^3 + n^2m + nm^2)$$

4.3.3. Итоговая сложность для параллельного алгоритма

Аккуратно сложив оценки каждого шага получим:

$$\boxed{\frac{14}{9p}n^3 + \frac{15}{6p}n^2m + \frac{5}{18p}nm^2 + o(n^3 + n^2m + nm^2)}$$

4.3.4. Оценка числа точек синхронизации алгоритма

Поговорим про число точек синхронизации. Собрав оценки, данные по ходу повествования в двух предыдущих пунктах, полчаем $5k$ точек синхронизации.

4.3.5. Количество и объём пересылок данных

Применение функций Allreduce и Bcast на каждом шаге алгоритма для каждого потока дает $2[\frac{n}{m}]$ пересылок„ объем которых равен $k*(n*m+1) = n^2 + [\frac{n}{m}]$. Помимо этого имеются пересылк при перестановке столбцов, но от них можно избавиться с помощью массива индексов столбцов (который и так уже есть и пременяется для персчета индексов).

Список литературы

[1] К. Ю. Богачев: Практикум на ЭВМ.Методы решения линейных систем и нахождения значений. Издательство Механико-Математического факультета МГУ. 1998.

[2] К. Ю. Богачев: Основы параллельного программирования. Бином. 2015.