СОДЕРЖАНИЕ

1 Умножение матриц по Винограду

Наивный алгоритм перемножения матриц требует $O\left(n^3\right)$ операций. Алгоритм Штрассена улучшает асимптотику до $O\left(n^{2.807355}\right)$. Алгоритм Копперсмита-Винограда работает за $O\left(n^{2.375477}\right)$, но на практике не используется из-за очень большой скрытой константы. На лекциях рассказывался просто алгоритм Винограда с небольшим лайфхаком, который все равно работает за куб, но с меньшим количеством умножений.

1.1 Теоретическое обоснование

В обычном алгоритме перемножения матриц элемент считается как

$$c_{ij} = \begin{pmatrix} a_{i1} & a_{i2} & \dots & a_{iN} \end{pmatrix} \begin{pmatrix} b_{1i} \\ b_{2i} \\ \dots \\ b_{Nj} \end{pmatrix} = \sum_{k=1}^{N} a_{ik} b_{kj}, \tag{1.1}$$

То есть всего мы сделаем N^3 умножений и $N^3 - N^2$ сложений.

Однако можно заметить, что

$$c_{ij} = \sum_{k=1}^{N/2} \left(a_{i,2k-1} + b_{2k,j}\right)\left(a_{i,2k} + b_{2k-1,j}\right) - \sum_{k=1}^{N/2} a_{i,2k-1} a_{i,2k} - \sum_{k=1}^{N/2} b_{2k-1,j} b_{2k,j}. \tag{1.2}$$

Второе и третье слагаемое можно предпосчитать и потом много раз использовать. То есть всего умножений и сложений будет

$$\frac{1}{2} N^3 + N^2 \tag{1.3}$$

$$\frac{3}{2} N^3 + 2N^2 - 2N \tag{1.4}$$

соответственно. Это немного веселее, чем в обычном алгоритме, потому что сложение дешевле, чем умножение.

1.2 Алгоритм

```
// предпосчет второго слагаемого
```

```
for i = 1:N
  row(i) = A(i, 1) * A(i, 2)
  for k = 2:N/2
    row(i) = row(i) + A(i, 2*k-1) * A(i, 2*k)
  end
end

// предпосчет третьего слагаемого
for j = 1:N
  col(j) = B(1, j) * B(2, j)
  for k = 2:N/2
    col(j) = col(k) + B(2*k-1, j) * B(2*k, j)
  end
end

// само перемножение
for i = 1:N
  for j = 1:N
    C(i, j) = -row(i) - col(j)
    for k = 1:N/2
      C(i, j) = C(i, j) + (A(i, 2 * k - 1) + B(2 * k, j)) *
(A(i, 2 * k) + B(2 * k - 1, j))
    end
  end
end
// если размерность матрицы нечетная, то надо не забыть
последние столбец/строку
if N % 2 == 1 then
  for i = 1:N
    for j = 1:N
      C(i, j) = C(i, j) + A(i, N) * B(N, j)
    end
  end
end
```

1.3 Векторный алгоритм

```
for i = 1:N
  row(i) = A(i, 1:2:N) * A(I, 2:2:N)'
end
for j = 1:N
  col(j) = B(1:2:N, j)' * B(2:2:N, j)
end
for i = 1:N
  for j = 1:N
    C(i, j) = [A(i, 1:2:N) + B(2:2:N, j)'] * [A(i, 2:2:N)' +
  + B(1:2:N, j)] – row(i) – col(j)
  end
end
```

2 Умножение матриц по Штрассену

$C = AB, \quad A, B, C \in \mathbb{R}^{2^n \times 2^n}$

Если размер не степень двойки, то можно дополнить нулевыми строками/столбцами.

Разделим каждую из матриц на 4 одинаковых блока

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} \tag{1.5}$$

где $A_{ij}, B_{ij}, C_{ij} \in \mathbb{R}^{2^{n-1} \times 2^{n-1}}$

Тогда C выражается как

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned} \tag{1.6}$$

Как и в обычном методе, требуется 8 умножений. Однако можно перевыразить все по-другому, определив сначала новые элементы:

$$\begin{aligned} P_1 &= \left(A_{11} + A_{22}\right)\left(B_{11} + B_{22}\right) \\ P_2 &= \left(A_{21} + A_{22}\right)B_{11} \\ P_3 &= A_{11}\left(B_{12} + B_{22}\right) \\ P_4 &= A_{22}\left(B_{21} - B_{11}\right) \\ P_5 &= \left(A_{11} + A_{12}\right)B_{22} \\ P_6 &= \left(A_{21} - A_{11}\right)\left(B_{11} + B_{12}\right) \\ P_7 &= \left(A_{12} - A_{22}\right)\left(B_{21} + B_{22}\right) \end{aligned} \tag{1.7}$$

А потом выразив через них

$$
\begin{aligned}
C_{11} &= P_1 + P_4 - P_5 + P_7 \\
C_{12} &= P_3 + P_5 \\
C_{21} &= P_2 + P_4 \\
C_{22} &= P_1 - P_2 + P_3 + P_6
\end{aligned}
\tag{1.8}
$$

Заметим, что таким образом нам требуется 7 умножений, а не 8. За это одно умножение платим 14 сложений. Более того, такой подход позволяет рекурсивно считать подматрицы, что хорошо прогается. Обычно такой подход продолжают до размеров матриц около 512, а потом используют наивный алгоритм, потому что данный метод теряет свою эффективность из-за бОльшего числа сложений, чем обычный.

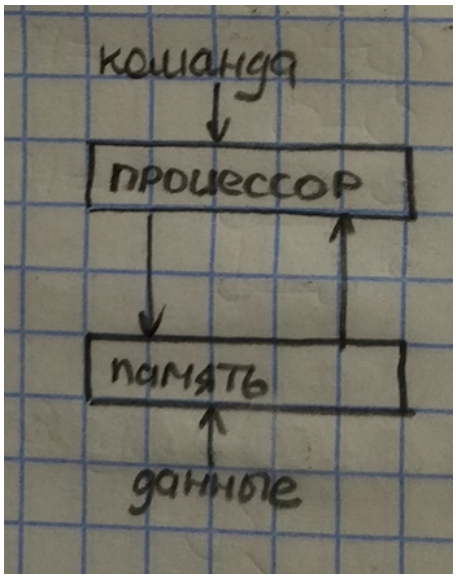
3 Классификация параллельных вычислительных систем по Флинну, модели представления параллельных алгоритмов

Вычисления **синхронные**, если вычислительный процесс определяется потоком команд.

Вычисления **асинхронные**, если вычислительный процесс определяется потоком данных.

	Один поток данных	Много потоков данных
Один поток команд	SISD Модель Неймана	SIMD Векторный процессор
Много потоков команд	MISD Конвейерный процессор	MIMD Многопроцессорная ЭВМ

3.1 SISD



Модель фон Неймана:

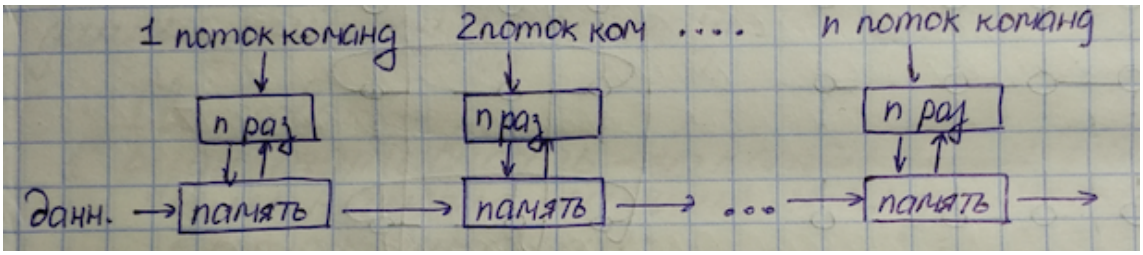
- Действия последовательны
- Данные обрабатываются последовательно

3.2 SIMD

Векторные ЭВМ

- В N раз быстрее
- Ограничение: необходимость векторизации численного метода

3.3 MISD

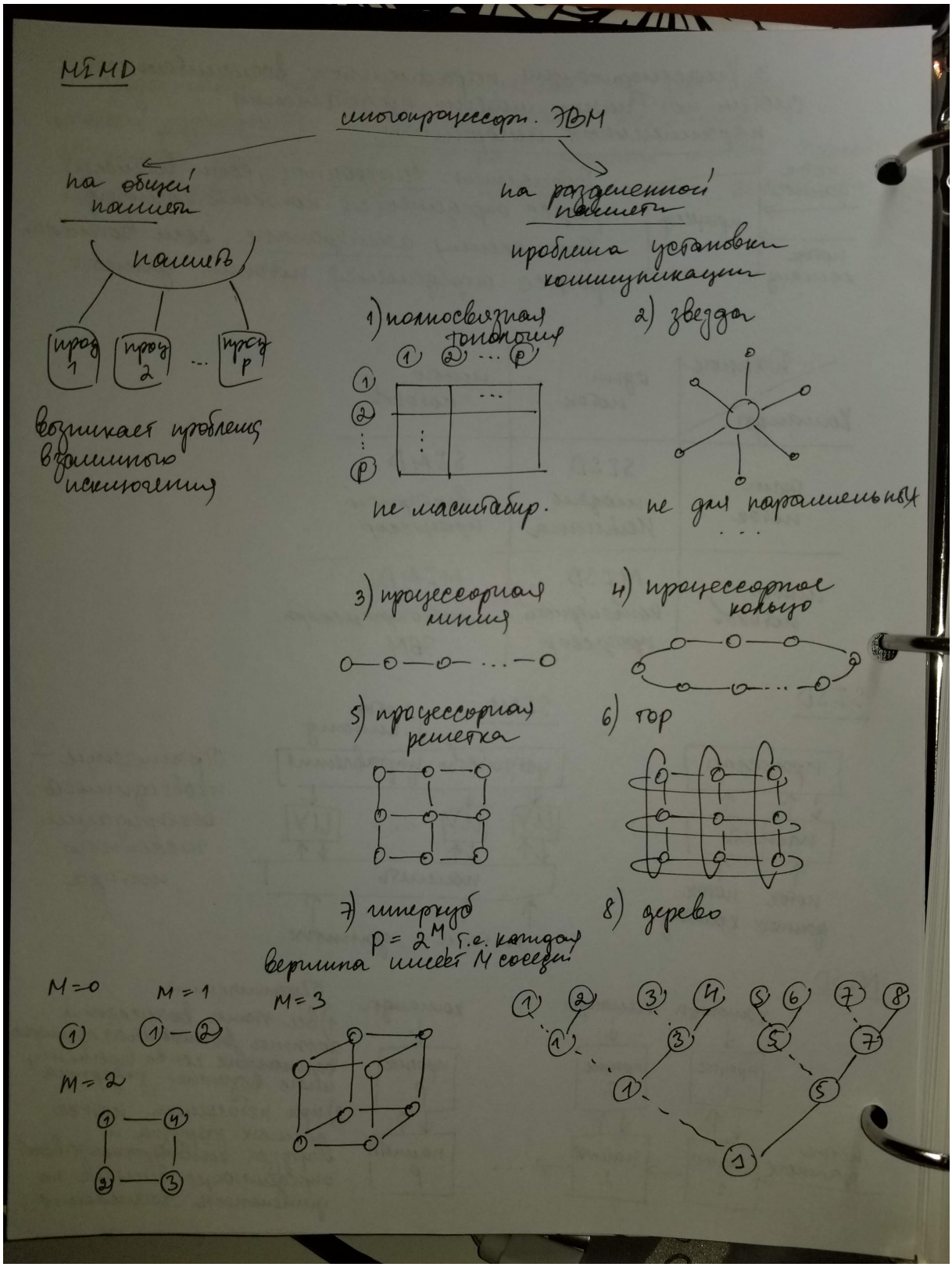


Конвейерный процессор. Конвейерный алгоритм предполагает разбиение задачи на подзадачи.

Ограничения:

- Все этапы вычисления должны выполняться примерно одинаковое количество времени, иначе возникнет задержка.
- При небольшом потоке данных загрузка и выгрузка конвейера оказывают определяющее влияние на длительность вычисления.

3.4 MIMD


MIMD
многопроцессор. ЭВМ
на общей памяти
на разделенной памяти
проблема установки коммуникации
память
проц 1
проц 2
проц p
возникает проблема взаимного исключения
1) полносвязная топология
2) звезда
не масштабир.
не для параллельных
3) процессорная линия
4) процессорное кольцо
5) процессорная решетка
6) тор
7) гиперкуб
p = 2^M т.е. каждая вершина имеет M соседей
8) дерево
M=0
M=1
M=2
M=3

3.5 Модели представления параллельных алгоритмов

Алгоритм (по Маркову) – явное предписание, задающее вычислительный процесс, идущий от варьируемых начальных данных к исходному результату.

3.6 Свойства алгоритмов:

- Однозначность
- Результативность
- Дискретность
- Массовость

3.7 Формы представления алгоритмов:

- Словесная
- Графическая
- Алгоритмическая

3.8 Модели представления

Модель задача/канал

Задача — последовательный алгоритм и данные, необходимые для его реализации (кружочек)

Канал — указывает на коммуникации между задачами (однонаправленная стрелочка)

Коммуникация организуется по принципу «первый зашел, первый вышел». Во время вычислений задачи могут возникать и упраздняться.

Модель параллелизма данных

Ориентирована на производство векторных и матричных вычислений. Ее формализм совпадает с языком Matlab.

Модель общей памяти

Предыдущая модель дополняется операторами записи и чтения общей памяти, а также механизмами решения проблемы взаимного исключения (дедлок?).

4 Модели вычислительных процессов

Вычислительный процесс – совокупность действий для выполнения программы.

Два **вычислительных процесса эквивалентны**, если при одинаковых начальных данных они приводят к одному результату.

Параллельный вычислительный процесс характеризуется одновременным исполнением нескольких действий.

Пример:

$L_1 : c = \log c$

$L_2 : d = \log b$

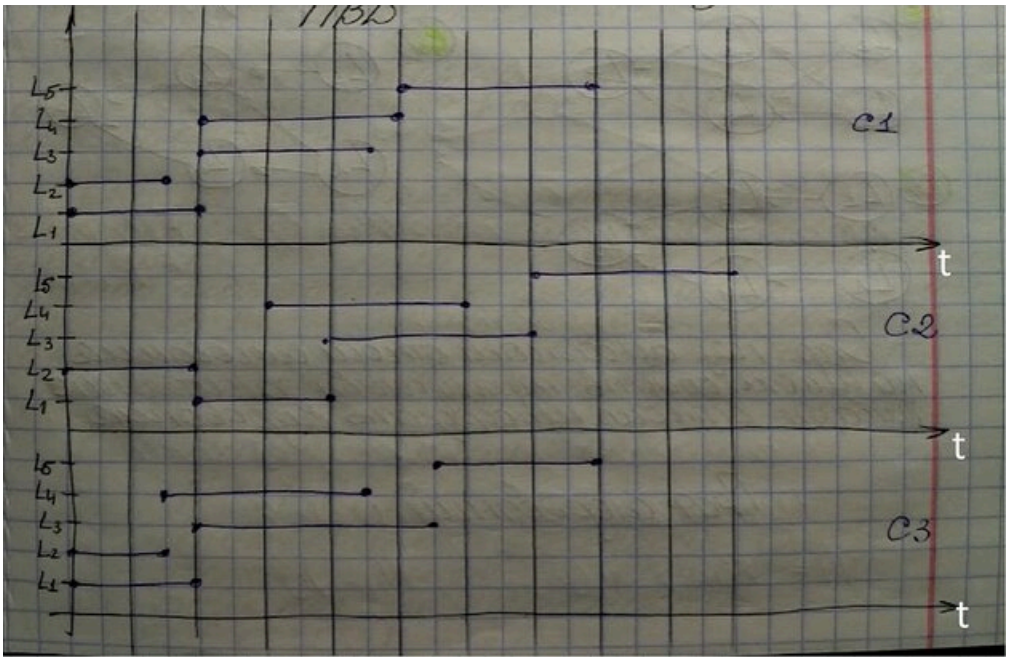
$L_3 : e = c + d$

$L_4 : f = d \times d$

$L_5 : g = e / f$

Эквивалентные вычислительные процессы, например: $L_1 L_2 L_4 L_3 L_5$, $L_2 L_1 L_3 L_4 L_5$, $L_2 L_1 L_4 L_3 L_5$.

4.1 Пространственно-временная диаграмма (ПВД)



ПВД – исчерпывающая информация о процессах, показывающая когда и сколько по времени выполнялась каждая команда.

4.2 Графическое представление вычислительных процессов

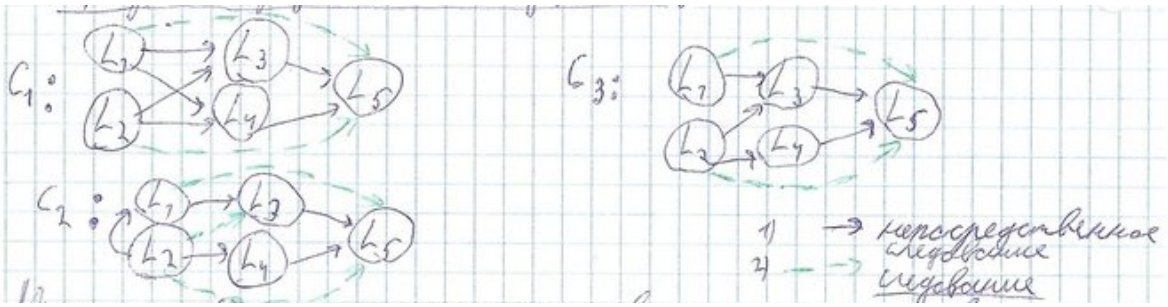
Поставим во взаимно однозначное соответствие любому оператору программы действие вычислительной системы по его исполнению, если таких действий несколько, примем их за одно.

На множестве действий зададим бинарное отношение непосредственного следования.

Действие d_2 **непосредственно следует** за $d_1 : (d_1, d_2)$, если d_2 начинается сразу после окончания d_1 и нет действия d_3, которое начинается после окончания d_1 и завершается перед началом d_2.

Введем бинарное отношение **следования** как транзитивное замыкание отношения непосредственного следования.

Граф непосредственного следования:



Сравнению подлежат исключительно эквивалентные процессы.

Назовем **процесс** C **не менее параллельным**, чем C', если для любой пары, связанной бинарным отношением следования (d_1, d_2) в C, верно, что в C' они также связаны бинарным отношением следования.

Из рисунка выше: C_3 не менее параллелен, чем C_1, C_2.

Максимально параллельный процесс – процесс, не менее параллельный, чем все эквивалентные ему.

Две **программы эквивалентны**, если эквивалентны любые два вычислительных процесса ими порожденные.

Данная **программа не менее параллельна**, чем некоторая эквивалентная ей, если множество процессов, порожденное данной

программой, содержит в качестве подмножества множество процессов, порожденных другой программой.

Максимально параллельная программа – программа, не менее параллельная, чем все эквивалентные ей.

5 Сети, сети Петри (определения, формализм, примеры)

5.1 Определение сети

Сетью назовем три объекта: $N = (P, T, F)$, где

P – непустое конечное множество мест,

T – непустое конечное множество переходов,

F – функция инцидентности:

$F : (P \times T \cup T \times P) \to \mathbb{N}_0$.

5.2 Свойства сети

- $P \cap T = \varnothing$
- $\forall_{p \in \mathrm{P}} \exists_{t \in T} : F(p,t) \neq 0 \vee F(t,p) \neq 0$
- $\forall_{t \in T} \exists_{p \in P} : F(t,p) \neq 0 \vee F(p,t) \neq 0$
- $\forall_{p_1, p_2 \in P} p_1^* = p_2^* \wedge p_1^{**} = p_2^{**} \Rightarrow p_1 = p_2$, где

  p^* – множество входных переходов $p : \{t : F(t,p) \neq 0\}$,

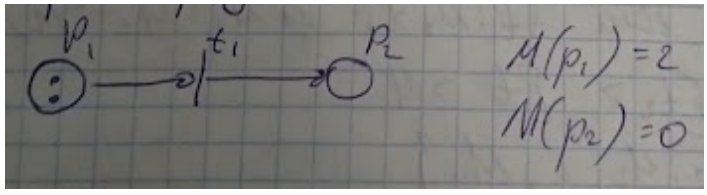
  p^{**} – множество выходных переходов $p : \{t : F(p,t) \neq 0\}$,

5.3 Сети Петри

$PN = (P, T, F, M)$, где

M – функция разметки $M : P \to \mathbb{N}_0$

Работа сети Петри определяется срабатыванием ее переходов и сменой разметки.



5.4 Условие срабатывания перехода

t может (но не должен) сработать, если $\forall_{p \in t^*} M(p) \geq F(p,t)$, где t^* – все места, из которых ведут стрелки в данном переходе (т.е. стрелок из t в p не больше, чем количество фишек в p).

Векторная функция $M \geq F^*$

$$\begin{pmatrix} M(p_1) \\ \vdots \\ M(p_n) \end{pmatrix} \geq \begin{pmatrix} F(p_1,t) \\ \vdots \\ F(p_n,t) \end{pmatrix}$$

5.5 Правило смены разметок

Если t сработал, тогда $\forall_{p \in P}$ верно:

$$M'(p) = M(p) - F(p,t) + F(t,p)$$

Закона сохранения фишек нет (в результате перехода может измениться сумма количества фишек)

$$M' = M - F^*(t) + F^{**}(t)$$

Работа сети Петри в целом определяется множеством сработанных переходов t и множеством достижимых разметок RH.

Введем отношение непосредственного следования на M:

$$M_1[t > M_2, \text{ если } \exists_t : M_1 \geq F^* \wedge M_2 = M_1 - F^*(t) + F^{**}(t)$$

$$M_1[t_1 > M_2[t_2 > \ldots[t_{\ldots} > M_{\ldots}$$

$$RH = \{ M_1, M_2, \ldots M_{\ldots} \}$$

6 Автоматическое распараллеливание ациклических фрагментов последовательных программ, построение стандартного графа и графа зависимостей

6.1 Построение стандартного графа

Очевидно, что любую последовательную программу можно записать с помощью лишь двух типов операторов: присваивание и условный переход.

По такой программе построим ориентированный граф, различая 2 типа его вершин: преобразователи ▢ (один или несколько операторов присваивания как один преобразователь) и распознаватель ◯ (один условный переход – один распознаватель).

Определяя функцию инцидентности, зададим на множестве операторов последовательной программы бинарное отношение непосредственного следования.

Таким образом из преобразователя выходит не более одной стрелки, а из распознавателя – не более двух соответствующих отношениям истинности и ложности проверяемого условия (1 и 0 соответственно).

Вершину, в которую не входит ни одна стрелка назовем входной, из которой не выходит ни одна стрелка – выходной.

Пример:

A: ввод(x, y)

B: l = x

C: h = y

D: v = c + y

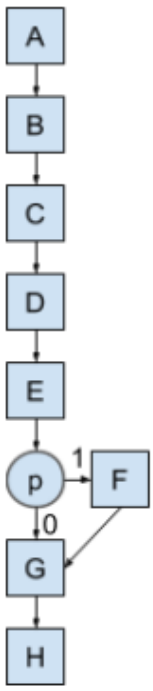
E: z = c + x

p: если x > y, то переходим на F, иначе – на G

F: h = x, l = y

G: печать (min(z, v), max(z, v))

H: печать (l, h)



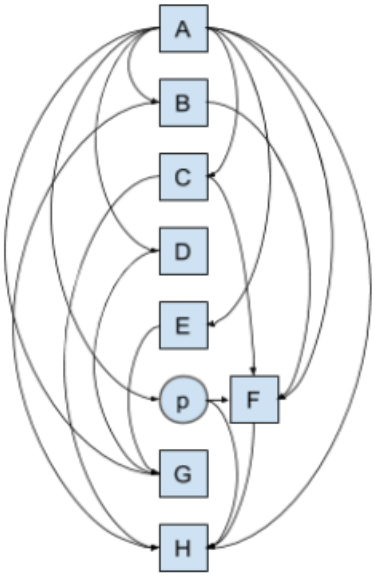
6.2 Построение графа зависимостей

Вершины этого графа совпадают с вершинами стандартного графа. А вот функция инцидентности строится на основе новых бинарных отношений и их транзитивного замыкания. Рассмотрим бинарные отношения информационной зависимости (на множестве вершин).

Будем говорить, что вершина B **информационно зависит** от вершины A, если они расположены на одном пути стандартного графа из входной вершины в выходную (причем A предшествует B) и хотя бы один оператор вершины B использует, но не изменяет значения переменной, которая формулирует хотя бы один оператор из вершины A (Out() – модифицированные переменные, In() – использованные без модификации. $Out(A) \cap In(B) \neq \varnothing$). Кроме того, на указанном пути между A и B отсутствует вершина C, хотя бы один оператор которой изменяет значение упомянутой переменной.

Будем говорить, что вершина B **логически зависит** от A, если A и B находятся на одном пути стандартного графа из входной вершины в выходную, причем A предшествует B и существует другой путь на стандартном графе из входной вершины в выходную, совпадающий с данным вплоть до вершины A, а далее от него отличающийся и не содержащий вершину B.

Будем говорить, что B **конкуренционно (параллельно) зависит** от A, если A и B находятся на одном пути стандартного графа из входной вершины в выходную, причем A предшествует B и есть хотя бы одна переменная, значение которой меняют операторы обеих вершин. Dep(A, B) – B зависит от A.



Далее строится транзитивное замыкание всех зависимостей без разбора их типов.

7 Автоматическое распараллеливание ациклических фрагментов последовательных программ, построение ЯПФ и параллельного алгоритма по стандартному графу и графу зависимостей. Формализм определения функции «с»

7.1 Построение ЯПФ (Ярусно-параллельной формы)

1. В 1 ярус помещаем вершины, ни от кого не зависящие
2. Пусть сформировано k ярусов. Если все вершины исчерпаны, ЯПФ построена. Иначе переходим к следующему шагу.
3. Построим k+1 ярус ЯПФ. Поместим в данный ярус вершины, зависящие исключительно от вершин предыдущИХ ярусОВ. Далее переходим на шаг 2.

При таком построении, все вершины одного яруса независимы. Следовательно, их операторы могут выполняться параллельно.

Пример (вроде как графа из пред вопроса):

Ярусы	Вершины
1	A
2	B, C, D, E, P
3	F, G
4	H

7.2 Построение параллельного алгоритма

Для начала построим по графам ЯПФ. Обозначим вершины на k-ом ярусе следующим образом: $A_{k,1},...,A_{k,n_k}$.

1. Полагаем, что речь идет о p процессорах и нужно указать μ-ому процессору вершины, операции которых он будет исполнять. Тогда вершины 1 яруса распределятся между процессорами так, что μ-ому процессору достанутся $A_{1,\mu},...,A_{1,\mu+p},...$ до исчерпания 1 яруса.

2. Пусть вершины k-ого яруса распределены между процессорами. Если ЯПФ исчерпана, алгоритм окончен. Иначе переходим к след шагу.
3. Распределим (k+1) ярус. μ-ому процессору достанутся следующие действия: $c(A_{k+1,\mu}); A_{k+1,\mu}; c(A_{k+1,\mu+p}); A_{k+1,\mu+p}; \ldots$????

7.3 Формализм определения (значения) функции «с»

(неформализм/человеческий язык: функция «эс не как доллар» удостоверяется, что все необходимые дела для выполнения какого-то действия A_{ij} сделаны.

Может быть три варика:

- дела сделаны, тогда все збс
- дела не сделаны и никогда не будут сделаны ((, тогда мы скипаем то действие, перед которой стоит наша цэ и начинаем готовиться к следующему (такое может быть, например, если какая-то другая вершина содержит иф, который решает, надо ли вообще выполнять наше действие)
- дела не сделаны, но пока ничего не понятно, может будут сделаны, а может нет. В любой непонятной ситуации ложись спать.

) теперь формально:

Пусть из B вершины в выходную идет n путей, различающихся до вершины A.

$\xi_1, \xi_2, \ldots, \xi_n$

$c_1, c_2, \ldots, c_n$ — истина / ложь / неопределено (см пред параграф че когда)

Тогда $c\left(A\right) = c_1 \vee c_2 \vee \ldots \vee c_n$, но как???

Пусть путь ξ_i состоит из вершин $A_1,\ldots,A_k$, выделим отсюда те, от которых зависит A: $A^1,\ldots,A^m$. Тогда

$$c_i = d^1 \cap d^2 \cap \ldots \cap d^m$$

Где d^l:

- $c\left(A^l\right)$, если А — преобразователь
- $c\left(A^l = 1\right)$, если А — распознаватель истины
- $c\left(A^l = 0\right)$, если А — распознаватель лжи

8 Распараллеливание циклических фрагментов программ (пространство итераций, ускорение, метод пирамид)

8.1 Определения

```
for i_1 = 1 : n_1
  for i_2 = 1 : n_2
    ...
      for i_k = 1 : n_k
        T(i_1, i_2, ..., i_k)
```

Поставим во взаимно однозначное соответствие любой операции конструкции, приведенной выше, вектор:

$$I = \left\{ (i_1, i_2, ..., i_k) : 1 \le i_q \le n_q : 1 \le q \le k \right\}$$

Введем понятие **пространства итераций** – множество всех возможных векторов.

Цель: отыскание покрытия пространства I.

$\left\{ I_q \right\}_{q=1}^{m}$, $I = \bigcup_{q=1}^{m} I_q$ (где n — количество подмножеств в покрытии)

Такое покрытие в большинстве случаев не единственно. Важная характеристика – в любом подмножестве I_q все вектора независимые (то есть соответствующие им T можно выполнить одновременно).

Каждое покрытие пространства I и соответствующий параллельный алгоритм характеризуются следующей величиной:

$S = \frac{\prod_{q=1}^{k} n_j}{m}$, которая является **ускорением**, где m – количество подмножеств в одном покрытии.

8.2 Метод пирамид

Шаг 1: На пространстве итераций выделим вектора, от которых не зависит ни один другой вектор данного пространства. Назовем их результирующими.

Шаг 2: Отнесем к каждой задаче вектора из пространства итераций, от которых зависит выделенный результирующий вектор.

Шаг 3: Упорядочим множества выделенных векторов в соответствии с бинарным отношением следования на пространстве итераций.

Характерные особенности метода пирамид:

1. Отсутствие метода коммуникаций
2. Платой за отсутствие коммуникаций является многократное дублирование вычислений

8.3 Пример

```
for i = 1 : r1
  for j = 1 : r2
    T(I, j)
```

Для данной циклической конструкции результирующими векторами являются вектора $p = \{(r1, j) : 1 \le j \le r2\}$, а вектора, от которых будет зависеть выбранный результирующий (помечен звездочкой на рисунке ниже), находятся в треугольнике между прямыми l и h. (Рисунок ниже в помощь).

Параллельный алгоритм будет состоять из $r2$ задач, k-ая задача ($k = 1 : r2$) в цикле по i производит следующие действия:

```
for j = max{1, k-(r1-i)tg(α)} : min{r2, k+(r1-i)tg(β)}
```

T(i, j),где $\tan(\alpha) = \frac{h_2}{h_1}$, $\tan(\beta) = \frac{l_2}{l_1}$,

α – острый угол между осью i и прямой l,

β – острый угол между осью i и прямой h.

Цикл по j пробегает все вектора с координатой i, лежащие внутри треугольника, ограниченного прямыми l и h.

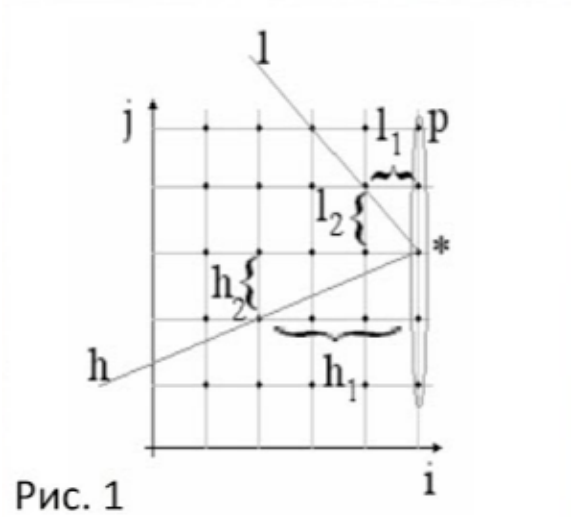


Рис. 1

9 Распараллеливание циклических фрагментов программ (пространство итераций, ускорение, метод параллелепипедов)

9.1 Пространство итераций и ускорение (также есть в пред вопросе)

В большинстве алгоритмов для численных методов отношение числа операций, производящихся в циклах, к общему числу операций, близко к единице. Следовательно, можно рассматривать задачу распараллеливания циклических участков программ как весьма важную. Рассмотрим участок последовательной программы, состоящий из k вложенных циклов:

```
for i_1 in range(n_1):
    for i_2 in range(n_2):
        ...
        for i_k in range(n_k):
            T(i_1, i_2, … ,i_k),
где T(i_1, i_2, … ,i_k) - Тело цикла
```

Каждой итерации этой циклической конструкции поставим в соответствие целочисленный вектор $(i_1, i_2, ..., i_k)$, $i_g = \overline{1, ng}$, $g = \overline{1, k}$

Пространство итераций $I = \{(i_1, i_2, ..., i_k) : 1 \le i_g \le n_g, 1 \le g \le k\}$ состоит из таких векторов. Очевидно, между пространством итераций и множеством итераций циклической конструкции можно установить взаимно однозначное соответствие.

Наша задача - найти совокупность множеств $\{I_q\}_{i=\overline{1,s}}$ являющуюся

$$I = \bigcup_{q=1}^{s} I_q$$

покрытием пространства итераций , таких, что вектора каждого множества I_q независимы друг от друга.

Для определения отношения зависимости на пространстве итераций "развернем" циклическую конструкцию в ациклический фрагмент:

T(1,1, … 1); T(1,1, … 2); … T(n_1, n_2, … ,n_k)

и построим для него граф зависимостей.

Если хотя бы две вершины такого графа, содержащие операторы из T' и T'', окажутся в отношении зависимости, то назовем зависимыми и соответствующие T' и T'' вектора из I.

Итерации, связанные с векторами одного множества I_q, можно выполнять независимо друг от друга (параллельно). Каждое покрытие пространства I и соответствующи параллельный алгоритм характеризуется величиной $\chi = \frac{\prod_{j=1}^{k} n_j}{s}$ – **ускорением**.

Следует искать покрытие, обеспечивающее максимальное ускорение параллельного алгоритма.

9.2 Метод параллелепипедов

(Данная страница для ознакомления, это писать нельзя)

В методе параллелепипедов пространство итераций разбивается (в теоретико-множественном смысле) на параллелепипеды. За счет этого, в отличии от предыдущих методов, метод параллелепипедов позволяет параллелить простые циклы. Например, для программы

for i:=1 to 15 do x(i):=x(i-4);

пространство итераций будет одномерным (рис. 1.13), каждый вектор этого пространства будет зависеть от векторов, находящихся слева по оси i на расстоянии, кратном 4.

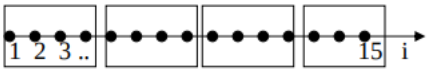


Рис. 1.13 Разбиение на параллелепипеды пространства итераций простого цикла.

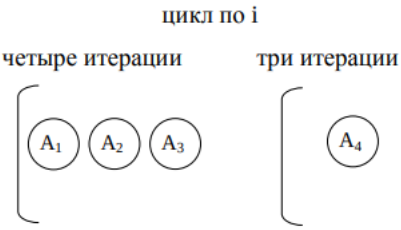


Рис.1.14. Модель параллельного алгоритма для параллелепипедов в случае простого цикла.

Между множеством векторов в параллелепипеде, содержащем наибольшее количество векторов, и задачами алгоритма установим биекцию Тогда задач будет 4 (рис. 1.14). Задаче под номером k (k=1,4) поручим следующие вычисления:

for i:=k to 15 step 4 do x(i):=x(i-4);

Первые три задачи произведут по 4 итерации цикла, четвертая 3. Алгоритм не нуждается в синхронизациях и передачах сообщений, так как операторы, содержащиеся в каждой задаче, соответствуют векторам, зависящим только от векторов данной задачи.

Алгоритм

Шаг 1: На пространстве итераций выделим покрытие, по которому характерное подмножество имеет вид параллелепипеда.

Шаг 2: На ребрах этого параллелепипеда выберем новые координатные оси, переобозначив в новой системе координат все вектора внутри данного параллелепипеда.

Шаг 3: Пусть вектор g имеет координаты $(g_1, g_2, \ldots, g_k)$ и длины ребер параллелепипеда есть $(p_1, p_2, \ldots, p_k)$, тогда параллельный алгоритм для M-той задачи имеет вид:

```
for i_1 = g_1 : p_1 : n_1
  for i_2 = g_2 : p_2 : n_2
    …
      for i_k = g_k : p_k : n_k
        +коммуникации
```

Особенности:

1. В теле цикла параметры цикла могут входить только в линейные выражения
2. Может случиться, что покрытие обеспечивает лучшее ускорение с большим объемом коммуникаций, чем другое с меньшим ускорением, тогда их сравнение по ускорению не окончательно.

10 Выражение произведения матриц через операции saxpy, gaxpy и модификацию внешним произведением

10.1 Операции с векторами (1 уровня)

- Сложение векторов: $z = x + y;\ x, y, z \in \mathbb{R}^{n\times 1}$
- Умножение вектора на скаляр: $z = \alpha x;\ \alpha \in \mathbb{R}$
- Скалярное произведение: $\alpha = x^T y$

10.2 Операции с матрицами (2 уровня)

- Saxpy: $z = \alpha x + y,\ \alpha \in \mathbb{R}$
  - Столбцовый: $z, x, y \in \mathbb{R}^{n\times 1}$
  - Строчный: $z, x, y \in \mathbb{R}^{1\times n}$
- Gaxpy:
  - Столбцовый: $z = Ax + y,\ A \in \mathbb{R}^{n\times n}$
  - Строчный: $z = xA + y,\ A \in \mathbb{R}^{n\times n}, x \in \mathbb{R}^{1\times n}$
- Модификация матрицы внешним произведением:

  $A = A + xy^T,\ A \in \mathbb{R}^{n\times m}, x \in \mathbb{R}^{n\times 1}, y \in \mathbb{R}^{m\times 1}$

Также стоит упомянуть про линейную комбинацию строк/столбцов.

10.3 Произведение матриц

Название	Векторная операция	Матричная операция	Способ доступа к памяти
ijk	Скалярное произведение	Строчный gaxpy	Смешанный
kij	Строчный saxpy	MM (внешнее произведение)	Строчный
jki	Столбцовый saxpy	Столбцовый gaxpy	Столбцовый

jik	Скалярное произведение	Столбцовый gaxpy	Смешанный
ikj	Строчный saxpy	Строчный gaxpy	Строчный
kji	Столбцовый saxpy	MM (внешнее произведение)	Столбцовый

Если хотите запомнить таблицу:

Индекс i соответствует столбцовому saxpy, если стоит последним, и строчному gaxpy, если стоит первым. Индекс j соответствует строчному saxpy, если последний, иначе – столбцовому gaxpy.Индекс k – скалярному произведению и MM.

Если не хотите запоминать, то алгоритм и операции можно написать и без нее. По индексам фигачим обычные форики с обычным перемножением матриц. Например, рассмотрим jik:

```
1 for j = 1 : n
2   for i = 1 : n
3     for k = 1 : n
4       C(i, j) = C(i, j) + A(i, k) * B(k, j)
```

Из произведения на 4 строчке делаем векторную операцию (сворачиваем последний форик):

```
1 for j = 1 : n
2   for i = 1 : n
3     C(i, j) = C(i, j) + A(i, 1 : n) * B(1 : n, j)
```

Замечаем, что мы перемножаем два вектора и получаем скаляр. Следовательно, это скалярное произведение. Свернем еще один форик, чтобы понять, какая здесь будет матричная операция:

```
1 for j = 1 : n
2     C(1:n, j) = C(1:n, j) + A(1:n, 1:n) * B(1:n, j)
```

Замечаем, что мы умножаем матрицу на вектор. Немного сынкинг и вспоминаем, что это, кажется gaxpy. А именно, столбцовый (мы ж на столбец ее умножаем, а не на строку). Изи пизи.

Видим, что здесь и к строкам обращаемся, и к столбцам, так что доступ к памяти будет смешанный.

11 Векторные алгоритмы gaxpy для симметричных и ленточных матриц

Такие матрицы выгодно хранить по диагоналям: $A_{\text{diag}}\left(\overset{0}{\leftrightarrow}\overset{1}{\leftrightarrow}\ldots\right)$.

Выразим для них операцию gaxpy: $z = y + Ax$.

Введем векторную операцию взятия одной диагонали: $D\left(A,k\right) = \text{diag}\left(0,\ldots,0,\overset{k}{\leftrightarrow},0,\ldots,0\right)$

Например:

$$A = \begin{pmatrix} 5 & 3 & 1 \\ 3 & 2 & 7 \\ 1 & 7 & 8 \end{pmatrix} =$$

$$= \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 3 & 0 & 0 \\ 0 & 7 & 0 \end{pmatrix} + \ldots + \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \tag{1.9}$$

$$= D\left(A,-2\right) + D\left(A,-1\right) + \ldots + D\left(A,2\right)$$

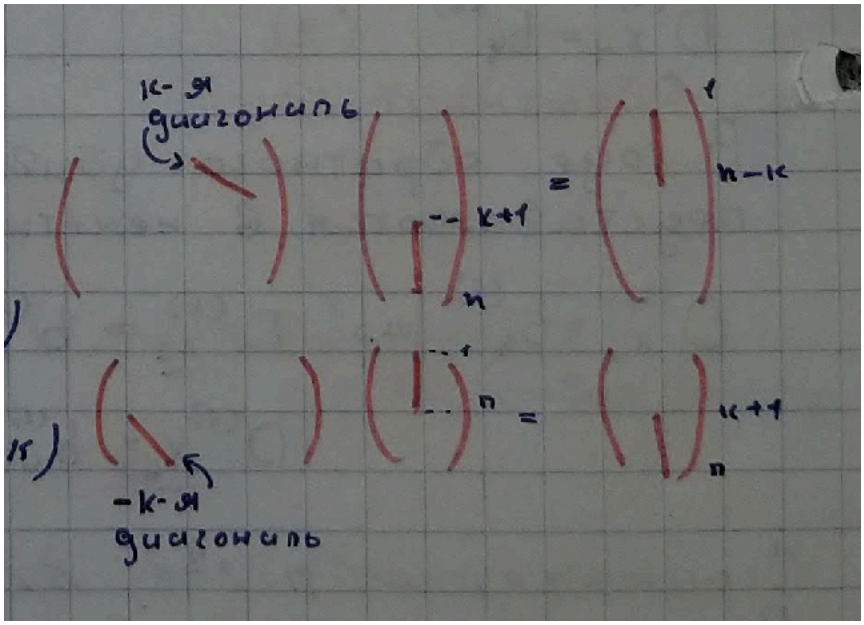
Отсюда

$$A = \sum_{k=-(n-1)}^{n-1} D\left(A,k\right) = D\left(A,0\right) + \sum_{k=1}^{n-1}\left(D\left(A,k\right) + D\left(A,-k\right)\right) \tag{1.10}$$

И домножим на вектор x

$$Ax = D\left(A,0\right)x + \sum_{k=1}^{n-1}\left(D\left(A,k\right)x + D\left(A,-k\right)x\right) \tag{1.11}$$

Будем последовательно перебирать диагонали, и каждую покомпонентно умножать на вектор



А потом получившиеся куски искомого вектора сложим.

Тут можно очень сочно проебаться с индексами, поэтому еще раз все распишем

```
D(A,  k) .* x(k+1:n) = y(1:n-k)
D(A, -k) .* x(1:n-k) = y(k+1:n)
```

Приведем код всего этого добра (A_diag это одномерный массив, где все элементы лежат друг за другом по диагоналям)

```
// нулевая диагональ
y = A_diag(1:n) .* x
// остальные
for k = 1:n-1
  // найдем индексы нашей диагонали в одномерном массиве
  t = k * n – k * (k - 1) / 2
  y(k+1:n) = y(k+1:n) + A_diag(t+1:t+n-k) .* x(1:n-k)
  y(1:n-k) = y(1:n-k) + A_diag(t+1:t+n-k) .* x(k+1:n)
end
```

12 Метод циклической редукции

(блочная циклическая редукция из голуба)

Пусть есть система с матрицей специального вида, где

$$\begin{pmatrix} D & F & & \cdots & 0 \\ F & D & F & & \vdots \\ & F & D & F & \\ \vdots & & \ddots & \ddots & \ddots \\ 0 & \cdots & & F & D \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ \vdots \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ \vdots \\ \vdots \\ b_n \end{pmatrix} \tag{2.1}$$

Также $DF = FD, \quad n = 2^k - 1, k \in \mathbb{N}$.

В общем случае в результате одной итерации циклической редукции мы перейдем от $2^k - 1$ уравнений относительно переменных $D^{(p)}, F^{(p)}, b^{(p)}$ к новой системе из $2^{k-1} - 1$ уравнений относительно $D^{(p+1)}, F^{(p+1)}, b^{(p+1)}$ (дальше в коде будет написано как именно) Будем так делать, пока у нас не останется одно уравнение относительно самого блока. Решим стандартным методом и начнем раскручивать всё назад.

Вычислим саму редукцию

for $p = 1 : k - 1$
  $D^{(p)} = 2[F^{(p-1)}]^2 - [D^{(p-1)}]^2$
  $F^{(p)} = [F^{(p-1)}]^2$
  $r = 2^p$
  for $j = 1 : 2^{k-p} - 1$
    $b_{jr}^{(p)} = F^{(p-1)}(b_{jr-r/2}^{(p-1)} + b_{jr+r/2}^{(p-1)}) - D^{(p-1)} b_{jr}^{(p-1)}$
  end
end

А потом найдем иксы обратным ходом

```
Решить D^(k-1) x_(2^k-1) = b_1^(k-1) относительно x_(2^k-1).
for p = k - 2 : -1 : 0
    r = 2^p
    for j = 1 : 2^(k-p-1)
        if j = 1
            c = b^(p)_((2j-1)r) - F^(p) x_(2jr)
        elseif j = 2^(k-p+1)
            c = b^(p)_((2j-1)r) - F^(p) x_((2j-2)r)
        else
            c = b^(p)_((2j-1)r) - F^(p) (x_(2jr) + x(2j - 2)r)
        end
        Решить D^(p) x_((2j-1)r) = c относительно x_((2j-1)r).
    end
end
```

(тут в коде где elseif должно быть 2^{k-p-1})

Плюсы: алгоритм масштабируем, максимальное количество одновременно задействованных процессоров — k.

Минусы: число коммуникаций — $O(\log n)$.

Наилучшим алгоритмом с независимым от размерности числом коммуникаций и масштабируемостью является метод диагонализации области.

Пример из голуба для n=7

подстановки.

Для понимания общей процедуры достаточно рассмотреть случай $n = 7$:

$$\begin{aligned}
b_1 &= Dx_1 + Fx_2, \\
b_2 &= Fx_1 + Dx_2 + Fx_3, \\
b_3 &= \quad Fx_2 + Dx_3 + Fx_4, \\
b_4 &= \quad\quad Fx_3 + Dx_4 + Fx_5, \\
b_5 &= \quad\quad\quad Fx_4 + Dx_5 + Fx_6, \\
b_6 &= \quad\quad\quad\quad Fx_5 + Dx_6 + Fx_7, \\
b_7 &= \quad\quad\quad\quad\quad Fx_6 + Dx_7.
\end{aligned} \tag{4.5.14}$$

Для $i = 2$, 4 и 6 мы умножим уравнения $i-1$, i и $i+1$ на матрицы F, $-D$ и F соответственно, и, складывая результат, получим

$$\begin{aligned}
(2F^2 - D^2)x_2 + \quad F^2 x_4 \quad\quad\quad &= F(b_1 + b_3) - Db_2, \\
F^2 x_2 + (2F^2 - D^2)x_4 + F^2 x_6 \quad &= F(b_3 + b_5) - Db_4, \\
F^2 x_4 + (2F^2 - D^2)x_6 &= F(b_5 + b_7) - Db_6.
\end{aligned}$$

Таким образом, следуя этой тактике, мы удалим все x с нечетными индексами и у нас останется уменьшенная блочная трехдиагональная система вида

$$\begin{aligned}
D^{(1)}x_2 + F^{(1)}x_4 \quad\quad &= b_2^{(1)}, \\
F^{(1)}x_2 + D^{(1)}x_4 + F^{(1)}x_6 &= b_4^{(1)}, \\
F^{(1)}x_4 + D^{(1)}x_6 &= b_6^{(1)}.
\end{aligned}$$

где матрицы $D^{(1)} = 2F^2 - D^2$ и $F^{(1)} = F^2$ являются перестановочными. Применяя такую же стратегию исключения, как и выше, мы умножим эти три уравнения соответственно на $F^{(1)}$, $-D^{(1)}$ и $F^{(1)}$. Если эти преобразованные уравнения сложить вместе, мы получим простое уравнение

$$(2[F^{(1)}]^2 - D^{(1)\,2})x_4 = F^{(1)}(b_2^{(1)} + b_6^{(1)}) - D^{(1)}b_4^{(1)},$$

которое запишем в виде

$$D^{(2)}x_4 = b^{(2)}.$$

Это полная циклическая редукция. Мы решим эту (маленькую) систему $q \times q$ относительно x_4. Векторы x_2 и x_6 находятся потом решением систем

$$D^{(1)}x_2 = b_2^{(1)} - F^{(1)}x_4, \quad D^{(1)}x_6 = b_6^{(1)} - F^{(1)}x_4.$$

В заключение мы используем первое, третье, пятое и седьмое уравнения из (4.5.14) для вычисления x_1, x_3, x_5 и x_7 соответственно.

13 Систолический алгоритм gaxpy на процессорном кольце с декомпозицией матрицы на блочные строки

Gaxpy:

$z = Ax + y,$

Где $z \in R^{n\times 1}, x \in R^{n\times 1}, y \in R^{n\times 1}, A \in R^{n\times n}$.

Матрица хранится по блочным строкам. Пусть количество процессоров равно **p**. Тогда gaxpy можно записать:

$$\begin{pmatrix} z_1 \\ z_2 \\ \dots \\ z_p \end{pmatrix} = \begin{pmatrix} \overline{A_{11}\ A_{12} \dots A_{1p}} \\ \overline{A_{21}\ A_{22} \dots A_{2p}} \\ \overline{\dots} \\ \underline{A_{p1}\ A_{p2} \dots A_{pp}} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_p \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_p \end{pmatrix}$$

Процессор с номером μ ($1 \le \mu \le p$) Вычисляет блок z_μ.

$$z_\mu = \sum_{k=1}^{p} A_{\mu k} x_k + y_\mu$$

$\mu -$ ю блочную строку матрицы A можно хранить в памяти $\mu -$ го процессора. Тогда элементы матрицы A не передаются по сети. Аналогично можно поступить с $\mu -$ м блоком y.

Алгоритм.

Инициализация $\{\mu -$ номер процессора $; p -$ количество процессоров $; r = \frac{n}{p}; row = (\mu - 1)r + 1 : \mu r; y_{loc} = y(row); A_{loc} = A(row, :); left, right; x_{loc} = x(row)\}$

for t = 1 : p
  send($x_{loc}, right$)
  recv($x_{loc}, left$)
  $\tau = \mu - t$
  if $\tau \le 0$
    $\tau = \tau + p$
  endif
  $y_{loc} = y_{loc} + A_{loc}(:, (\tau - 1)r + 1 : \tau r) x_{loc}$
endfor

14 Систолический алгоритм gaxpy на процессорном кольце с декомпозицией матрицы на блочные столбцы

Gaxpy:

$$z = Ax + b$$

Систолический алгоритм характеризуется строгой последовательностью. Все действия производятся в цикле для синхронизации ???

В начале цикла выполняют отправки:

```
for t = 1 : T
  send(...)
  recv(...)
  ...
end
```

Пусть A распределена по блочным столбцам:

$$\begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_p \end{pmatrix} = \left(\begin{array}{c|c|c|c} & & & \\ A_1 & A_2 & \cdots & A_p \\ & & & \end{array} \right) \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_p \end{pmatrix}$$

В основе алгоритма линейная комбинация блочных столбцов:

$$z = \sum_{k=1}^{p} A_k x_k + y$$

Пусть

$$\omega_\mu = A_\mu x_\mu$$

$$z_\mu = \sum_{k=1}^{p} \omega_k(col) + y_\mu = \sum_{k=1}^{p} \omega_k((\mu-1)r+1:\mu r) + y_\mu$$

Инициализация:

$$n, p; r = \frac{n}{p}; \mu;$$

$$col = (\mu-1)r+1:\mu r;$$

$left, right$;

$A_{loc} = A(:, col),\ x_{loc} = x(col),\ y_{loc} = y(col).$

Алгоритм:

```
w = A_loc*x_loc
for t = 1 : p
  send(w, right)
  recv(w, left)
  y_loc = y_loc + w(col)
end
```

Итого: p пересылок.

Можно уменьшить число коммуникаций на 1, сделав форик $t = 1 : p - 1$ и добавив еще одну строчку перед ним:

```
w = A_loc*x_loc
y_loc = y_loc + w(col)
for t = 1 : p - 1
  send(w, right)
  recv(w, left)
  y_loc = y_loc + w(col)
end
```

15 Столбцово-ориентированные алгоритмы умножения матриц на кольце

Хотим посчитать

$$\begin{gathered} D = C + AB \\ A, B, C, D \in \mathbb{R}^{n \times n} \end{gathered} \tag{2.2}$$

параллельно, то есть есть p процессоров, а ранг текущего — μ. На каждый процессор без ограничения общности выделим $r = \frac{n}{p}$ столбцов.

При этом всем, матрица у нас будет храниться по блочным столбцам, то есть

$$\begin{gathered} \left(\begin{array}{c|c|c|c} D_1 & D_2 & \dots & D_p \end{array} \right) = \left(\begin{array}{c|c|c|c} C_1 & C_2 & \dots & C_p \end{array} \right) + \\ + \left(\begin{array}{c|c|c|c} A_1 & A_2 & \dots & A_p \end{array} \right) \left(\begin{array}{c|c|c|c} B_1 & B_2 & \dots & B_p \end{array} \right) \end{gathered} \tag{2.3}$$

Каждый процессор будет в итоге считать свой блочный столбец, то есть

$$D_\mu = C_\mu + AB_\mu = C_\mu + \sum_{k=1}^{p} A_k B_{k\mu} \tag{2.4}$$

Видно, что каждому процессору понадобится каждый блочный столбец A_k. Ими они и будут обмениваться этаким хороводиком. Приведем код алгоритма:

```
// инициализация
  // индексы нашего столбца
  col = (mu - 1) * r + 1 : mu * r
  // блочные столбцы, которыми мы будем пользоваться
  // результат положим в матрицу C
  A_loc = A(:, col)
  B_loc = B(:, col)
  C_loc = C(:, col)
  // индексы соседей
  left = (mu - 2 + p) % p + 1
  right = mu % p + 1

// вычисления
```

```
for t = 1 : p
  send(A_loc, right)
  recv(A_loc, left)
  // индекс блочного столбца, который сейчас лежит в A_loc
  tau = (mu - t - 1 + p) % p + 1
  // соответсвующие ему строчные индексы
  row_tau = (tau - 1) * r + 1 : tau * r
  // изменяем ответ
  C_loc = C_loc + A_loc * B_loc(row_tau, :)
end
```

16 Строчно-ориентированные алгоритмы умножения матриц на кольце

(очень похоже на предыдущий вопрос)

Хотим посчитать

$$D = C + AB$$
$$A, B, C, D \in \mathbb{R}^{n \times n} \tag{2.5}$$

параллельно, то есть есть p процессоров, а ранг текущего — μ. На каждый процессор без ограничения общности выделим $r = \frac{n}{p}$ строк.

При этом всем, матрица у нас будет храниться по блочным строкам, то есть

$$\left(\begin{array}{c} D_1 \\ \hline D_2 \\ \hline \vdots \\ \hline D_p \end{array}\right) = \left(\begin{array}{c} C_1 \\ \hline C_2 \\ \hline \vdots \\ \hline C_p \end{array}\right) + \left(\begin{array}{c} A_1 \\ \hline A_2 \\ \hline \vdots \\ \hline A_p \end{array}\right) \left(\begin{array}{c} B_1 \\ \hline B_2 \\ \hline \vdots \\ \hline B_p \end{array}\right) \tag{2.6}$$

Каждый процессор будет в итоге считать свою блочную строку, то есть

$$D_\mu = C_\mu + A_\mu B = C_\mu + \sum_{k=1}^{p} A_{k\mu} B_k \tag{2.7}$$

Видно, что каждому процессору понадобится каждая блочная строка B_k. Ими они и будут обмениваться этаким хороводиком. Приведем код алгоритма:

```
// инициализация
  // индексы нашего столбца
  row = (mu - 1) * r + 1 : mu * r
  // блочные строки, которыми мы будем пользоваться
  // результат положим в матрицу C
  A_loc = A(row, :)
  B_loc = B(row, :)
  C_loc = C(row, :)
  // индексы соседей
  left = (mu - 2 + p) % p + 1
```

```
  right = mu % p + 1

// вычисления
  for t = 1 : p
    send(B_loc, right)
    recv(B_loc, left)
    // индекс блочной строки, который сейчас лежит в A_loc
    tau = (mu - t - 1 + p) % p + 1
    // соответсвующие ей столбцовые индексы
    col_tau = (tau - 1) * r + 1 : tau * r
    // изменяем ответ
    C_loc = C_loc + A_loc(:, col_tau) * B_loc
  end
```

17 Систолический алгоритм умножения матриц на торе

Основан на блочном скалярном произведении.

$a = \alpha N + \beta$, где все числа $\in \mathbb{N}_0$, $0 \le \beta < N$.

Определим модуль a по N:

$$|a|_N = \begin{cases} \beta, \beta \neq 0 \\ \text{N, else} \end{cases}$$

Инициализация:

$p, N = \sqrt{p}, n, r = \frac{n}{N}, (\mu, \lambda)$ – номер процессора;

$C_{loc} = C_{\mu,\lambda} = C((\mu-1)r+1 : \mu r, (\lambda-1)r+1 : \lambda r)$

$A_{loc} = A_{\mu, <\lambda+\mu-1>_N}$

Для процессоров: $\left(\begin{array}{c|c|c} 1,1 & 1,2 & 1,3 \\ \hline 2,1 & 2,2 & 2,3 \\ \hline 3,1 & 3,2 & 3,3 \end{array}\right)$ хранятся: $\left(\begin{array}{c|c|c} A_{11} & A_{12} & A_{13} \\ \hline A_{22} & A_{23} & A_{13} \\ \hline A_{33} & A_{31} & A_{32} \end{array}\right)$

Таким образом, в μ-ой строке происходит циклический сдвиг на $\mu - 1$ влево.

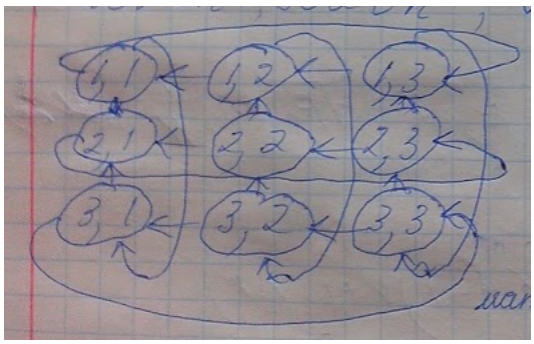
$B_{loc} = B_{<\mu+\lambda-1>_N, \lambda}$

То есть для B сдвиг по столбцам на $\lambda - 1$ вверх:

$\left(\begin{array}{c|c|c} B_{11} & B_{22} & B_{33} \\ \hline B_{21} & B_{32} & B_{13} \\ \hline B_{31} & B_{12} & B_{23} \end{array}\right)$

north, south, west, east. (Инициализация закончилась).

Посылаем блоки матрицы A с востока на запад, а матрицу B с юга на север.



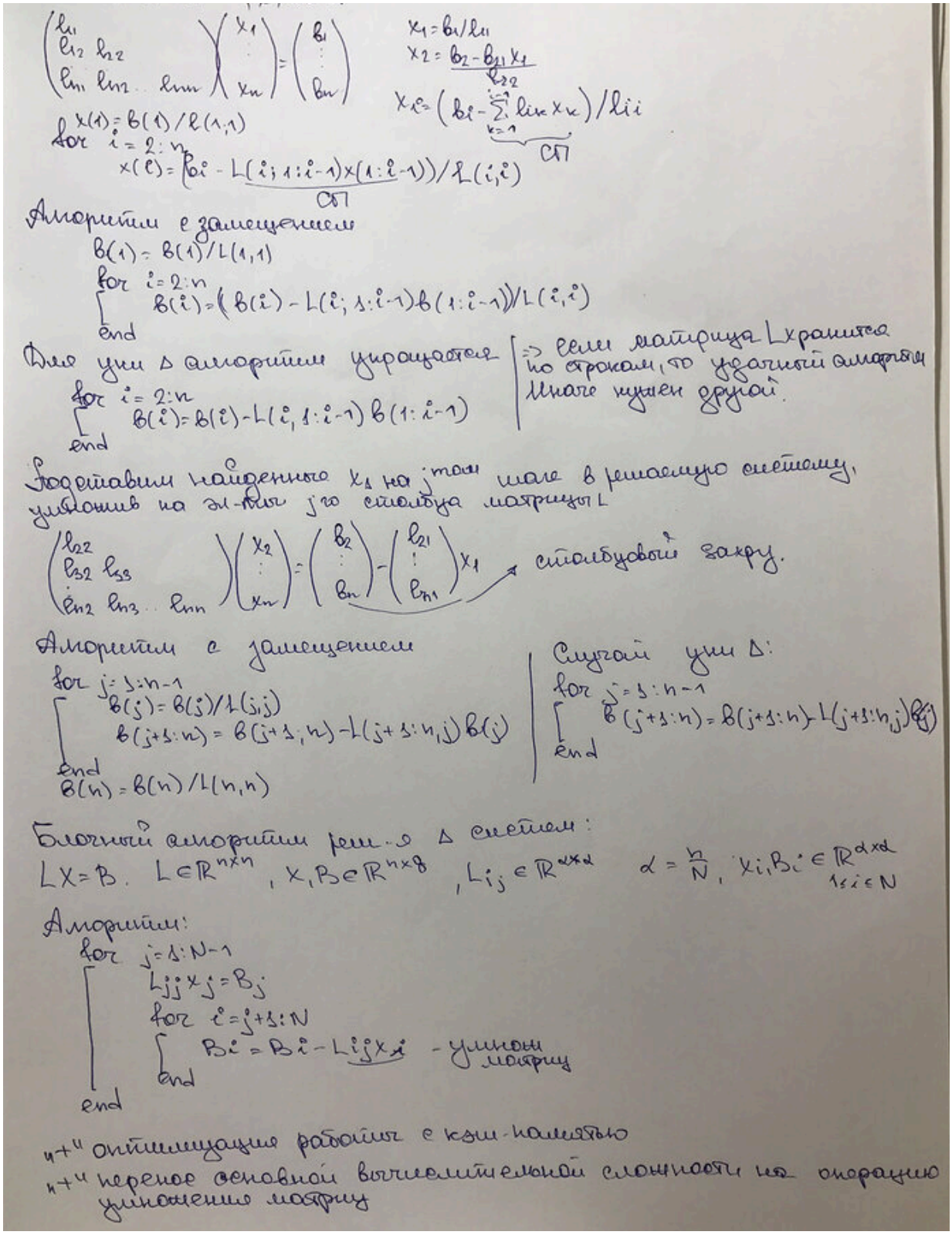
Алгоритм:

```
for t = 1 : √p
  send(A_loc, west)
  send(B_loc, north)
  recv(A_loc, east)
  recv(B_loc, south)
  C_loc = C_loc + A_loc * B_loc
end
```

Сравнивая данный алгоритм с предыдущим (алгоритмом на кольце), можно отметить значительно меньший объем коммуникаций при $N \to \infty$. Ранее приходилось передавать всю матрицу A (или B), в то время как в данном алгоритме мы пересылаем только строку A и столбец B.

18 Векторные и блочные алгоритмы решения треугольных СЛАУ

$Lx = b,\ L \in \mathbb{R}^{n\times n},\ x, b \in \mathbb{R}^{n\times 1}$



Блочный алгоритм реш-я Δ систем:
$LX = B.\ L \in \mathbb{R}^{n\times n},\ X, B \in \mathbb{R}^{n\times s},\ L_{ij} \in \mathbb{R}^{\alpha\times\alpha},\ \alpha = \frac{n}{N},\ x_i, B_i \in \mathbb{R}^{\alpha\times\alpha},\ 1 \le i \le N$

n+4 оптимизация работы с кэш-памятью
n+4 перенос основной вычислительной сложности на операцию умножения матриц

19 Векторные алгоритмы LU-разложения на основе операции модификации матрицы внешним произведением

$$\begin{pmatrix} 1 & 0 \\ -\frac{x_2}{x_1} & 1 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}=\begin{pmatrix} x_1 \\ 0 \end{pmatrix}$$

Первая матрица является матрицей преобразования. Ее 21 элемент является множителем Гаусса.

$$\begin{pmatrix} 1 & & & & & 0 & \\ & 1 & & & & 0 & \\ & & \ddots & & & \vdots & \\ & & & 1 & & 0 & \\ & & & & -\tau_{k+1}^{(k)} & & \\ & & & & \vdots & \ddots & \\ & & & & -\tau_n^{(k)} & & 1 \end{pmatrix}\begin{pmatrix} x_1 \\ \vdots \\ \\ x_k \\ x_{k+1} \\ \vdots \\ x_n \end{pmatrix}=\begin{pmatrix} x_1 \\ \vdots \\ \\ x_k \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

$$\tau^{(k)}=\begin{pmatrix} 0 \\ \vdots \\ 0 \\ \tau_{k+1}^{(k)} \\ \vdots \\ \tau_n^{(k)} \end{pmatrix},\ \tau_i^{(k)}=\frac{x_i}{x_k},\ i>k$$

$M_k = I - \tau^{(k)} e_k^T$, где e_k – нулевой вектор с единицей на k-ой позиции.

$$M_k A = A - \tau^{(k)} e_k^T A = A - \tau^{(k)} A(k,:)$$

$$\left(A\right)-\left(\begin{array}{c} 0 \\ \hline \dots \end{array}\right)\ \Big\}k$$

$$\tau_i^{(k)}=\frac{A(i,k)}{A(k,k)},\quad i>k$$

$$\tau^k A(k,:) = \begin{pmatrix} \dfrac{0}{A(k+1,k)} \\ \dots \\ A(n,k) \end{pmatrix} \Big\}k$$

$$M_k A = \begin{pmatrix} \dfrac{A(1:k,:)}{0} \\ \vdots \\ 0 \end{pmatrix} \Big\}k$$ (образуются нули в k-ом столбце)

$$M_1 A = \begin{pmatrix} & & \\ 0 & & \\ \vdots & & \\ 0 & & \end{pmatrix} \quad M_2 M_1 A = \begin{pmatrix} & & \\ 0 & & \\ \vdots & \ddots & \\ 0 & 0 & \end{pmatrix}$$

$$M_3 M_2 M_1 A = \begin{pmatrix} & & \\ 0 & & \\ \vdots & \ddots & \\ 0 & 0 & 0 \end{pmatrix} = U$$

Алгоритм нахождения U:

```
for k = 1 : n − 1
  g(k + 1 : n) = A(k + 1 : n, k) / A(k, k)
  A(k + 1 : n, k + 1 : n) = A(k + 1 : n, k + 1 : n) −
    − g(k + 1 : n)A(k, k + 1 : n)
end
```

$$A = M_1^{-1} M_2^{-1} \dots M_{n-1}^{-1} U \Rightarrow L = M_1^{-1} \dots M_{n-1}^{-1}$$

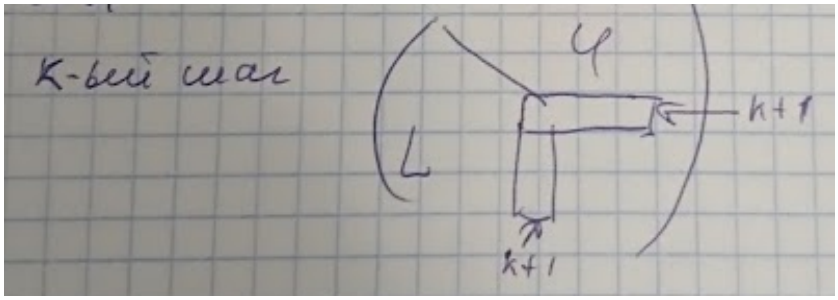
$$M_k^{-1} = I + \tau^{(k)} e_k^T$$

$$M_1^{-1} \dots M_{n-1}^{-1} = I + \sum_{k=1}^{n-1} \tau^{(k)} e_k^T = \begin{pmatrix} 1 & & & 0 \\ \tau_2^{(1)} & 1 & & \\ \vdots & & \ddots & \\ \tau_n^{(1)} & \dots & \tau_n^{(n-1)} & 1 \end{pmatrix}$$

На k-ой итерации находим k+1 строку U и записываем ее в верхний треугольник A.

Алгоритм LU:

```
for k = 1 : n − 1
  A(k + 1 : n, k) = A(k + 1 : n, k) / A(k, k)
  A(k + 1 : n, k + 1 : n) = A(k + 1 : n, k + 1 : n) −
     − A(k + 1 : n, k)A(k, k + 1 : n)
end
```



20 Gaxpy варианты LU-разложения

$A \in R^{n \times n}$, главные подматрицы A(1:k, 1:k) невырождены для k = 1:n-1. A = LU, где L – нижняя унитреугольная матрица, а U – верхняя треугольная. Обе матрицы будем хранить в матрице A.

Положим, что за j-1 шаг найдены j-1 столбцы матриц L и U. На j шаге ищем j столбцы L и U. Выразим j столбец матрицы A:

A(:, j) = LU(:, j)

Выразим верхнюю и нижнюю часть столбца отдельно.

1. A(1:j-1, j) = L(1:j-1, :)U(:, j)
   Так как L(1:j-1, j:n) = 0
   A(1:j-1, j) = L(1:j-1, 1:j-1)**U(1:j-1, j)**,
   где неизвестное выделено жирным шрифтом и красным цветом (все остальное мы знаем). Значит изи пизи находим, решая СЛАУ
2. A(j:n, j) = L(j:n, :)U(:, j)
   U(j+1:n, j) = 0
   A(j:n, j) = L(j:n, 1:j)U(1:j, j) =

$$= \sum_{k=1}^{j} L(j:n, k)U(k, j) =$$

$$= \sum_{k=1}^{j-1} L(j:n, k)U(k, j) + L(j:n, j)U(j, j) =$$

= L(j:n, 1:j-1)U(1:j-1, j) + **L(j:n, j)U(j, j)**

   Будем выражать неизвестное через некий вектор v(j:n):
   v(j:n) = A(j:n, j) - L(j:n, 1:j-1)U(1:j-1, j)
   L(j, j)U(j, j) = v(j) =>
   U(j, j) = v(j), L(j+1:n, j) = v(j+1:n)/v(j, j)

Запишем наконец-то алгоритм.

1. Скалярный вариант

```
for j = 1 : n
  for k = 1 : j-1
    for i = k+1 : j-1
      A(i, j) = A(i, j) - A(i, k)A(k, j)
    end
  end
  for k = 1 : j-1
    for i = j : n
      A(i, j) = A(i, j) - A(i, k)A(k, j)
    end
  end
  for i = j+1 : n
    A(i, j) = A(i, j) / A(j, j)
  end
end
```

2. Векторный вариант

```
for j = 1 : n
  for k = 1 : j-1
    A(k+1 : j-1, j) = A(k+1 : j-1, j) - A(k+1 : j-1, k)A(k, j)
  end
  for k = 1 : j-1
    A(j : n, j) = A(j : n, j) - A(j : n, k)A(k, j)
  end
  A(j+1 : n, j) = A(j+1 : n, j) / A(j, j)
end
```

P.S. Алгоритмы взяты из Голуба и чутка редактированы. Все индексы соответствуют индексам Голуба.

Матричный вариант

```
for j = 1 : n
  for k = 1 : j-1
    A(k+1 : j-1, j) = A(k+1 : j-1, j) - A(k+1 : j-1, k)A(k, j)
  end
  A(j:n, j) = A(j:n, j) – A(j:n, 1:j-1) A(1:j-1, j)
  A(j+1 : n, j) = A(j+1 : n, j) / A(j, j)
end
```

Здесь записан jki вариант (Аналогично jik, просто поменяв внутренние форики местами). Есть еще так же ijk, ikj.

21 Блочные алгоритмы LU-разложения

Блочное LU-разложение.

LU-разложение — представление матрицы A в виде произведения двух матриц, $A = LU$, где L — нижняя унитреугольная матрица, U — верхняя треугольная матрица.

Блочное LU-разложение матрицы A выглядит следующим образом:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & I_{n-r} \end{pmatrix} \begin{pmatrix} I_r & 0 \\ 0 & \tilde{A} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & I_{n-r} \end{pmatrix};$$

$$A_{11}, L_{11}, I_r, U_{11} \in R^{n\text{x}n},\ A_{21}, L_{21} \in R^{(n-r)\text{x}n};$$

$$A_{12}, U_{12} \in R^{r\text{x}(n-r)},\ A_{22}, \tilde{A}, I_{n-r} \in R^{(n-r)\text{x}(n-r)},$$

Где r – блочный параметр.

Произведём операцию умножения для трёх матриц, которые выше. Получим:

$$A = \begin{pmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & L_{21}U_{12} + \tilde{A} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix};$$

Значит можно записать следующим образом:

1) $A_{11} = L_{11}U_{11}$;
2) $A_{12} = L_{11}U_{12}$;
3) $A_{21} = L_{21}U_{11}$;
4) A_{22}=$L_{21}U_{12} + \tilde{A}$.

Применяем рекурсивно алгоритм:

1) Блоки L_{11} и U_{11} найдем, применив стандартный метод Гаусса для разложения блока A_{11}.
2) Затем найдём блоки U_{12} и L_{21} решив треугольные системы с несколькими правыми частями.
3) После этого находим редуцированную матрицу $\tilde{A} = A_{22}$ - $L_{21}U_{12}$.

Ещё алгоритмы, но подробно как они работают хз. Взято из Голуба страницы 99- 100:

Алгоритм 3.2.5 (Блочная версия *LU*-разложения с внешним произведением). Пусть $A \in \mathbf{R}^{n \times n}$ и главные подматрицы $A(1:k, 1:k)$ невырождены для $k = 1:n-1$. Предположим, что r удовлетворяет условию $1 \leqslant r \leqslant n$. Следующий алгоритм вычисляет разложение $A = LU$ посредством r-ранговой модификации. Причем элементы $A(i, j)$ замещаются на элементы $L(i, j)$ при $i > j$ и элементы $A(i, j)$ замещаются на $U(i, j)$ при $j \geqslant i$.

$\lambda = 1$
while $\lambda \leqslant n$
  $\mu = \min(n, \lambda + r - 1)$
  Используется алгоритм 3.2.3 для замещения $A(\lambda:\mu, \lambda:\mu) = \tilde{L}\tilde{U}$ на $\tilde{L}$ и $\tilde{U}$.
  Решаем: $\tilde{L}Z = A(\lambda:\mu, \mu+1:n)$. Замещаем: $A(\lambda:\mu, \mu+1, n) \leftarrow Z$
  Решаем: $W\tilde{U} = A(\mu+1:n, \lambda:\mu)$. Замещаем: $A(\mu+1:n, \lambda:\mu) \leftarrow W$
  $A(\mu+1:n, \mu+1:n) = A(\mu+1:n, \mu+1:n) - WZ$
  $\lambda = \mu + 1$
end

Алгоритм требует $2n^3/3$ флопов.

Алгоритм 3.2.6 (Блочная gaxpy-версия *LU*-разложения). Пусть матрица $A \in \mathbf{R}^{n \times n}$ имеет ненулевые главные миноры $A(1:k, 1:k)$ для $k = 1:n-1$. И пусть целочисленный параметр r удовлетворяет соотношению $1 \leqslant r \leqslant n$. Тогда следующий алгоритм вычисляет разложение $A = LU$ посредством r-ранговой модификации. На выходе $A(i, j)$ замещаются на $L(i, j)$ для $i > j$ и $A(i, j)$ замещаются на $U(i, j)$ для $j \geqslant i$.

$\lambda = 1$
while $\lambda \leqslant n$
  $\mu = \min(n, \lambda + r - 1)$
  $r_1 = \mu - \lambda + 1$
  Выполняем r_1 шагов алгоритма 3.2.4 для $A(\lambda:n, \lambda:n)$
  Замещаем $A(\lambda:\mu, \mu+1:n)$ в соответствии с решением
    $A(\lambda:\mu, \lambda:\mu)Z = A(\lambda:\mu, \mu+1:n)$
  $A(\mu+1:n, \mu+1:n) = A(\mu+1:n, \mu+1:n)$
    $- A(\mu+1:n, \lambda:\mu)A(\lambda:\mu, \mu+1:n)$
  $\lambda = \mu + 1$
end

Алгоритм требует $2n^3/3$ флопов.

Алгоритмы 3.2.3 и 3.2.4 (они используются в этих алгоритмах):

Алгоритм 3.2.3 (Исключение Гаусса с внешним произведением). Предположим, что матрица $A \in \mathbf{R}^{n \times n}$ обладает таким свойством, что подматрицы $A(1:k, 1:k)$ невырождены для $k = 1:n-1$. Данный алгоритм вычисляет разложение $M_{n-1} \ldots M_1 A = U$, где матрица U является верхней треугольной, а каждая матрица M_k – это матрица преобразования Гаусса. Матрица U хранится в верхнем треугольнике матрицы A. Множители, задающие матрицы M_k, запоминаются в элементах $A(k+1:n, k)$, т. е. $A(k+1:n, k) = -M_k(k+1:n, k)$.

for $k = 1:n-1$
  $t =$ **gauss**$(A(k:n, k))$
  $A(k+1:n, k) = t$
  $A(k:n, k+1:n) =$ **gauss.app**$(A(k:n, k+1:n, t))$
end

Этот алгоритм требует $(2/3)n^3$ флопов и является классической формулировкой метода *исключения Гаусса*. Заметим, что каждое прохождение через k-цикл реализует внешнее произведение.

Алгоритм 3.2.4. (Gaxpy-версия исключения Гаусса). Пусть матрица $A \in \mathbf{R}^{n \times n}$ и главные подматрицы $A(1:k,\ 1:k)$ невырожденны для $k = 1:n-1$. Данный алгоритм вычисляет разложение $A = LU$, где L – нижняя унитреугольная матрица, а U – верхняя треугольная. Если $i > j$, то элементы $A(i, j)$ содержат элементы $L(i, j)$. Если $i \leqslant j$, то элементы $A(i, j)$ содержат элементы $U(i, j)$.

```
for j = 1:n
    {Решить L(1:j − 1, 1:j − 1) U(1:j − 1, j) = A(1:j − 1, j)}
    for k = 1:j − 1
        for i = k + 1:j − 1
            A(i, j) = A(i, j) − A(i, k) A(k, j)
        end
    end
    {v(j:n) = A(j:n, j) − L(j:n, 1:j − 1) U(1:j − 1, j)}
    for k = 1:j − 1
        for i = j:n
            A(i, j) = A(i, j) − A(i, k) A(k, j)
        end
```

```
    end
    {L(j + 1:n, j) = v(j + 1:n)/v(j)}
    A(j + 1:n, j) = A(j + 1:n, j)/A(j, j)
end
```

Этот алгоритм требует $2n^3/3$ флопов.

22 Векторные алгоритмы разложения Холецкого на основе операции модификации матрицы внешним произведением

4.2.5. Метод Холецкого с внешним произведением

Альтернативная процедура Холецкого, основанная на модификации с внешним произведением, может быть получена из следующего разбиения:

$$A = \begin{bmatrix} \alpha & v^T \\ v & B \end{bmatrix} = \begin{bmatrix} \beta & 0 \\ v/\beta & I_{n-1} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & B - vv^T/\alpha \end{bmatrix} \begin{bmatrix} \beta & v^T/\beta \\ 0 & I_{n-1} \end{bmatrix}. \tag{4.2.6}$$

Здесь $\beta = \sqrt{\alpha}$ и мы знаем, что $\alpha > 0$, потому что A положительно определена. Заметим, что $B - vv^T/\alpha$ тоже положительно определена, потому что она является главной подматрицей матрицы X^TAX, где

$$X = \begin{bmatrix} 1 & -v^T/\alpha \\ 0 & I_{n-1} \end{bmatrix}.$$

Таким образом, если мы имеем разложение Холецкого $G_1G_1^T = B - vv^T/\alpha$, то из формулы (4.2.6) следует, что $A = GG^T$, где

$$G = \begin{bmatrix} \beta & 0 \\ v/\beta & G_1 \end{bmatrix}.$$

Далее наша цель – многократно выполнить разбиение (4.2.6) до предельно маленьких подматриц, почти так же, как в *kji*-алгоритме исключения Гаусса. Если мы замещаем нижнюю треугольную часть матрицы A на матрицу G, то получается

Алгоритм 4.2.2 (Метод Холецкого: версия с внешним произведением). Дана симметричная положительно определенная матрица $A \in \mathbf{R}^{n \times n}$; следующий алгоритм вычисляет нижнюю треугольную матрицу $G \in \mathbf{R}^{n \times n}$, такую, что $A = GG^T$. Для всех $i \geqslant j$ $G(i, j)$ замещает $A(i, j)$.

```
for k = 1:n
    A(k, k) = √A(k, k)
    A(k + 1:n, k) = A(k + 1:n, k)/A(k, k)
    for j = k + 1:n
        A(j:n, j) = A(j:n, j) − A(j:n, k) A(j, k)
    end
end
```

Алгоритм требует $n^3/3$ флопов. Заметим, что j-цикл вычисляет нижнюю треугольную часть модификации с внешним произведением

$$A(k + 1:n, k + 1:n) = A(k + 1:n, k + 1:n) - A(k + 1:n, k)\, A(k + 1:n, k)^T.$$

Возвращаясь к нашим рассуждениям в § 1.4.8 о сравнении gaxpy-версии и модификации с внешним произведением, можно легко показать, что алгоритм 4.2.1 осуществляет в два раза меньший векторный обмен, чем алгоритм 4.2.2.

23 Gaxpy варианты разложения Холецкого

4.2.4. Gaxpy-версия метода Холецкого

Сначала мы получим реализацию метода Холецкого, богатую операциями типа gaxpy. Если мы сравним j-е столбцы в уравнении $A = GG^T$, то мы получим

$$A(:, j) = \sum_{k=1}^{j} G(j, k)\, G(:, k).$$

Отсюда следует, что

$$G(j, j)\, G(:, j) = A(:, j) - \sum_{k=1}^{j-1} G(j, k)\, G(:, k) \equiv v. \tag{4.2.5}$$

Если мы знаем $(j-1)$ первых столбцов матрицы G, то может быть вычислен вектор v. Отсюда через сравнение компонент в формуле (4.2.5) следует, что $G(j{:}n, j) = v(j{:}n)/\sqrt{v(j)}$. Это и есть скалярная операция gaxpy, и поэтому мы получаем следующий метод, основанный на gaxpy-операции для вычисления разложения Холецкого:

for $j = 1{:}n$
  $v(j{:}n) = A(j{:}n, j)$
  for $k = 1{:}j-1$
    $v(j{:}n) = v(j{:}n) - G(j, k)\, G(j{:}n, k)$
  end
  $G(j{:}n, j) = v(j{:}n)/\sqrt{v(j)}$
end

Вычисления можно организовать так, что G замещает нижние треугольные матрицы A.

Алгоритм 4.2.1 (Метод Холецкого: Gaxpy-версия). При данной симметричной положительно определенной матрице $A \in \mathbf{R}^{n\times n}$ следующий алгоритм вычисляет нижнюю треугольную матрицу $G \in \mathbf{R}^{n\times n}$, такую, что $A = GG^T$. Для всех $i \geqslant j$ $G(i, j)$ замещает $A(i, j)$.

for $j = 1{:}n$
  if $j > 1$
    $A(j{:}n, j) = A(j{:}n, j) - A(j{:}n, 1{:}j-1)\, A(j, 1{:}j-1)^T$
  end
  $A(j{:}n, j) = A(j{:}n, j)/\sqrt{A(j, j)}$
end

Алгоритм требует $n^3/3$ флопов.

Пример 4.2.1. Матрица

$$\begin{bmatrix} 2 & -2 \\ -2 & 5 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \sqrt{2} & 0 \\ -\sqrt{2} & \sqrt{3} \end{bmatrix} \begin{bmatrix} \sqrt{2} & -\sqrt{2} \\ 0 & \sqrt{3} \end{bmatrix}$$

является положительно определенной.

24 Блочные алгоритмы разложения Холецкого

1. На скалярном произведении

Пусть $A \in \mathbf{R}^{n \times n}$ – симметричная положительно определенная матрица. Будем рассматривать матрицу $A = (A_{ij})$ и ее множители Холецкого $G = G_{ij}$ как блочные $N \times N$-матрицы с квадратными диагональными блоками. Приравнивая (i, j)-е блоки из уравнения $A = GG^T$ при $i \geqslant j$, мы получаем, что $A_{ij} = \sum_{k=1}^{j} G_{ik} G_{jk}^T$. Обозначив

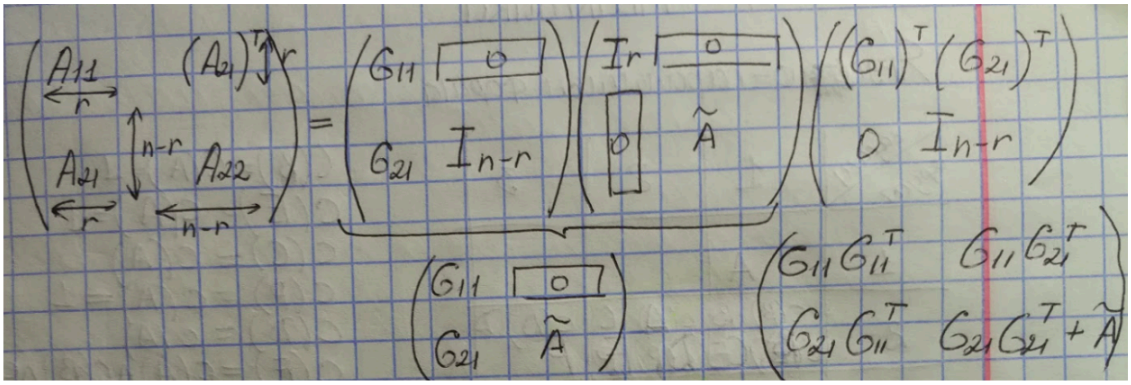
$$S = A_{ij} - \sum_{k=1}^{j-1} G_{ik} G_{jk}^T,$$

мы видим, что $G_{jj} G_{jj}^T = S$, если $i = j$, и $G_{ij} G_{jj}^T = S$, если $i > j$. При правильном упорядочивании эти уравнения могут быть использованы для вычисления всех матриц G_{ij}.

Алгоритм 4.2.3 (Метод Холецкого: блочная версия со скалярным произведением). Дана симметричная положительно определенная матрица $A \in \mathbf{R}^{n \times n}$; следующий алгоритм вычисляет нижнюю треугольную матрицу $G \in \mathbf{R}^{n \times n}$, такую, что $A = GG^T$. Нижний треугольник матрицы A замещается на нижнюю треугольную часть матрицы G. Матрица A рассматривается как блочная $N \times N$-матрица с квадратными диагональными блоками.

for $j = 1 : N$
  for $i = j : N$
    $S = A_{ij} - \sum_{k=1}^{j-1} G_{ik} G_{jk}^T$
    if $i = j$
      Вычислить разложение Холецкого $S = G_{jj} G_{jj}^T$.
    else
      Решить $G_{ij} G_{jj}^T = S$ относительно G_{ij}.
    end
    Выполнить замещение A_{ij} на G_{ij}.
  end
end

2. Головашкинский



Шаг 1: ищем разложение Холецкого блока A_{11} и находим G_{11}: $A_{11} = G_{11}G_{11}^T$

Шаг 2: решаем систему $G_{21}G_{11}^T = A_{21}$

Шаг 3: $\widetilde{A} = A_{22} - G_{21}G_{21}^T$, принимаем $A = \widetilde{A}$ и возвращаемся к шагу 1.

3. Gaxpy

Другая блочная процедура может быть получена из gaxpy-алгоритма Холецкого. Через r шагов алгоритма 4.2.1 нам известны матрицы $G_{11} \in \mathbf{R}^{r \times r}$ и $G_{21} \in \mathbf{R}^{(n-r) \times r}$ в разложении

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} G_{11} & 0 \\ G_{21} & I_{n-r} \end{bmatrix} \begin{bmatrix} I_r & 0 \\ 0 & \tilde{A} \end{bmatrix} \begin{bmatrix} G_{11} & 0 \\ G_{21} & I_{n-r} \end{bmatrix}^T.$$

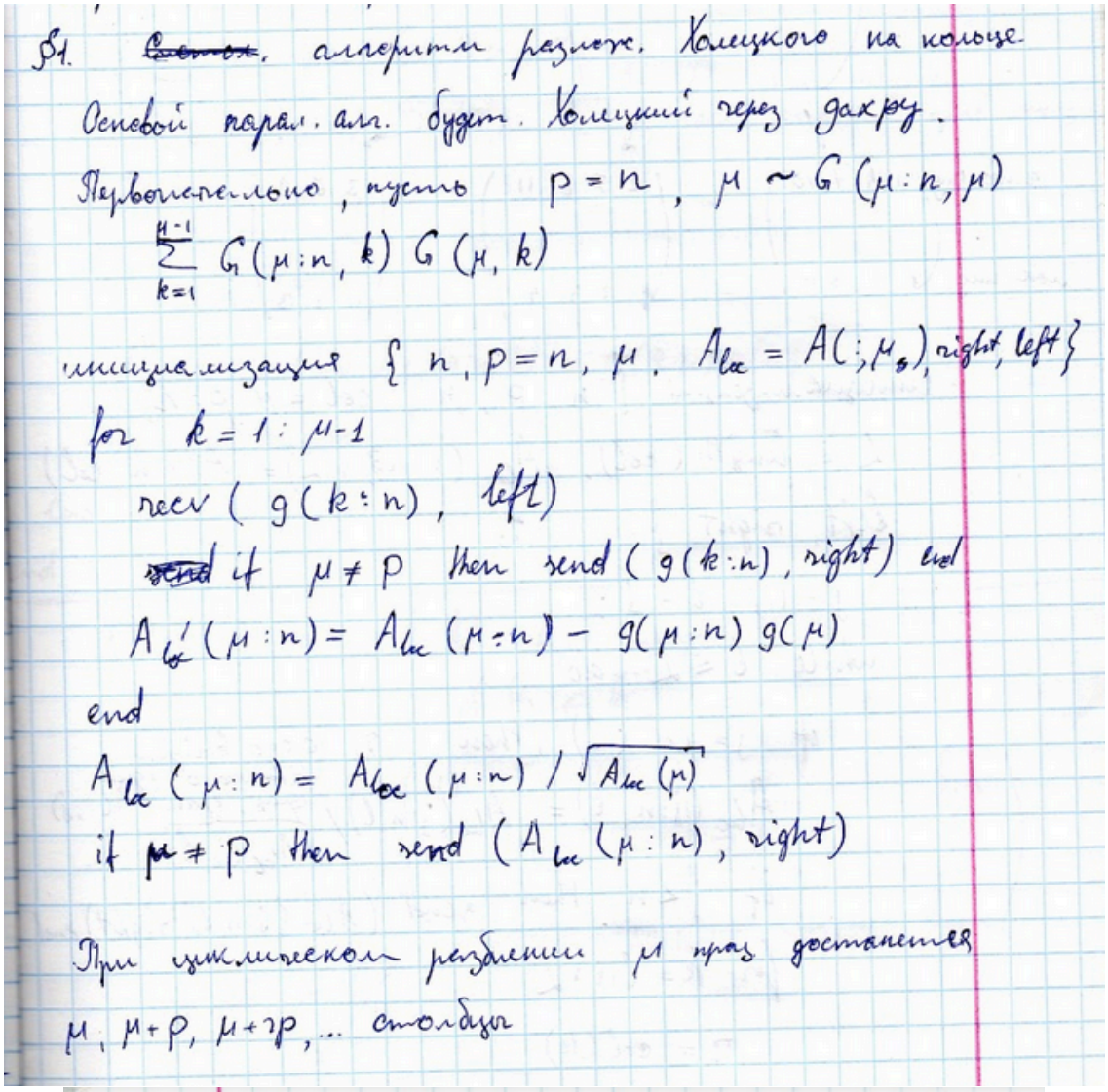
После r шагов gaxpy-версии Холецкого мы преобразуем уже не матрицу A, а редуцированную матрицу $\tilde{A} = A_{22} - G_{21}G_{21}^T$, которую *получили* явно с учетом симметрии. Продолжая этот подход, мы получим блочный алгоритм Холецкого, k-й шаг которого содержит r шагов gaxpy-версии Холецкого для матрицы порядка $n - (k-1)r$, следовательно, вычисления 3-го уровня имеют порядок $n - kr$. Вклад операций 3-го уровня аппроксимируется соотношением $1 - 3/(2N)$, если $n \approx rN$.

Алгоритм 4.2.1 (Метод Холецкого: Gaxpy-версия). При данной симметричной положительно определенной матрице $A \in \mathbf{R}^{n \times n}$ следующий алгоритм вычисляет нижнюю треугольную матрицу $G \in \mathbf{R}^{n \times n}$, такую, что $A = GG^T$. Для всех $i \geqslant j$ $G(i, j)$ замещает $A(i, j)$.

```
for j = 1:n
    if j > 1
        A(j:n, j) = A(j:n, j) − A(j:n, 1:j − 1)A(j, 1:j − 1)^T
    end
    A(j:n, j) = A(j:n, j)/√A(j, j)
end
```

25 Алгоритм LU-разложения на процессорном кольце

26 Алгоритм разложения Холецкого на процессорном кольце



§1. ~~Систем~~, алгоритм разлож. Холецкого на кольце.

Основой парал. алг. будет Холецкий через дакру.

Первоначально, пусть $p = n$, $\mu \sim G(\mu : n, \mu)$

$$\sum_{k=1}^{\mu-1} G_1(\mu : n, k)\, G(\mu, k)$$

инициализация { n, $p = n$, μ, $A_{loc} = A(:, \mu_0)$, right, left }

for $k = 1 : \mu - 1$

recv ($g(k : n)$, left)

~~send~~ if $\mu \neq p$ then send ($g(k : n)$, right) end

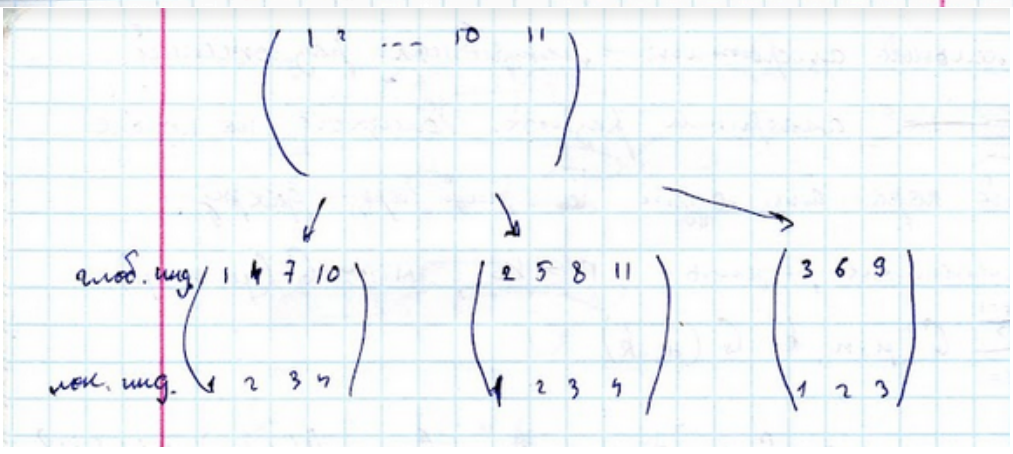
$A'_{loc}(\mu : n) = A_{loc}(\mu : n) - g(\mu : n)\, g(\mu)$

end

$A_{loc}(\mu : n) = A_{loc}(\mu : n) / \sqrt{A_{loc}(\mu)}$

if $\mu \neq p$ then send ($A_{loc}(\mu : n)$, right)

При циклическом разбиении μ пред достается $\mu, \mu + p, \mu + 2p, \ldots$ столбцы



```
синхронизация { n, p, M, col = M:p:n,
  L = length(col), A_loc(1:n, 1:L) = A(1:n, col)
  left, right }
  j = 1; l = 1;
while l ≤ L do
  if j = col(l) then  % сюда войдет только один проц. хран. A(:,j)
    A_loc(j:n, l) = A_loc(j:n,l) / A_loc(j, l)
    if j < n then send (A_loc(j:n, l), rightA) end
    for k = l+1 : L
      r = col(k)
      A_loc(r:n, k) = A_loc(r:n, k) - A_loc(r:n, l) · A_loc(r, l)
    end
    j = j+1; l = l+1;
  else
    recv (g(j:n), left)
    if (right ≠ α) & (β > j) then  % α - номер проц. хранящего j столбец
      send (g(j:n), right)         % β - наим. индекс
    end                            % неэл. столбца прав. соседа
    for k = l : L
      r = col(k);
      A_loc(r:n, k) = A_loc(r:n, k) - g(r:n) g(a)
    end.
    j = j + 1
  end
end
```

27 Алгоритм разложения Холецкого на процессорной решетке

За основу данного алгоритма взят алгоритм разложения Холецкого через модификацию матрицы внешним произведением. Для упрощения не будем учитывать симметричность. Тогда алгоритм можно записать:

for k = 1:n
A(k,k) = sqrt(A(k,k)) // (1)
A(k+1:n , k) = A(k+1:n , k) / A(k,k) // (2)
A(k , k+1:n) = A(k , k+1:n) / A(k,k) // (3)
*A(k+1:n , k+1:n) = A(k+1:n , k+1:n) – A(k+1:n , k)*A(k , k+1:n) // (4)*
endfor

Положим, что количество процессоров равно количеству элементов матрицы. Тогда каждый процессор выполняет помимо передачи и приема данных одно из следующих действий:

Извлечь корень (Если обрабатывается элемент на главной диагонали, в алгоритме выше действие (1))

Произвести деление(Если обрабатывается элемент не на главной диагонали, в алгоритме выше это действия (2), (3))

Модификация матрицы (Действие (4))

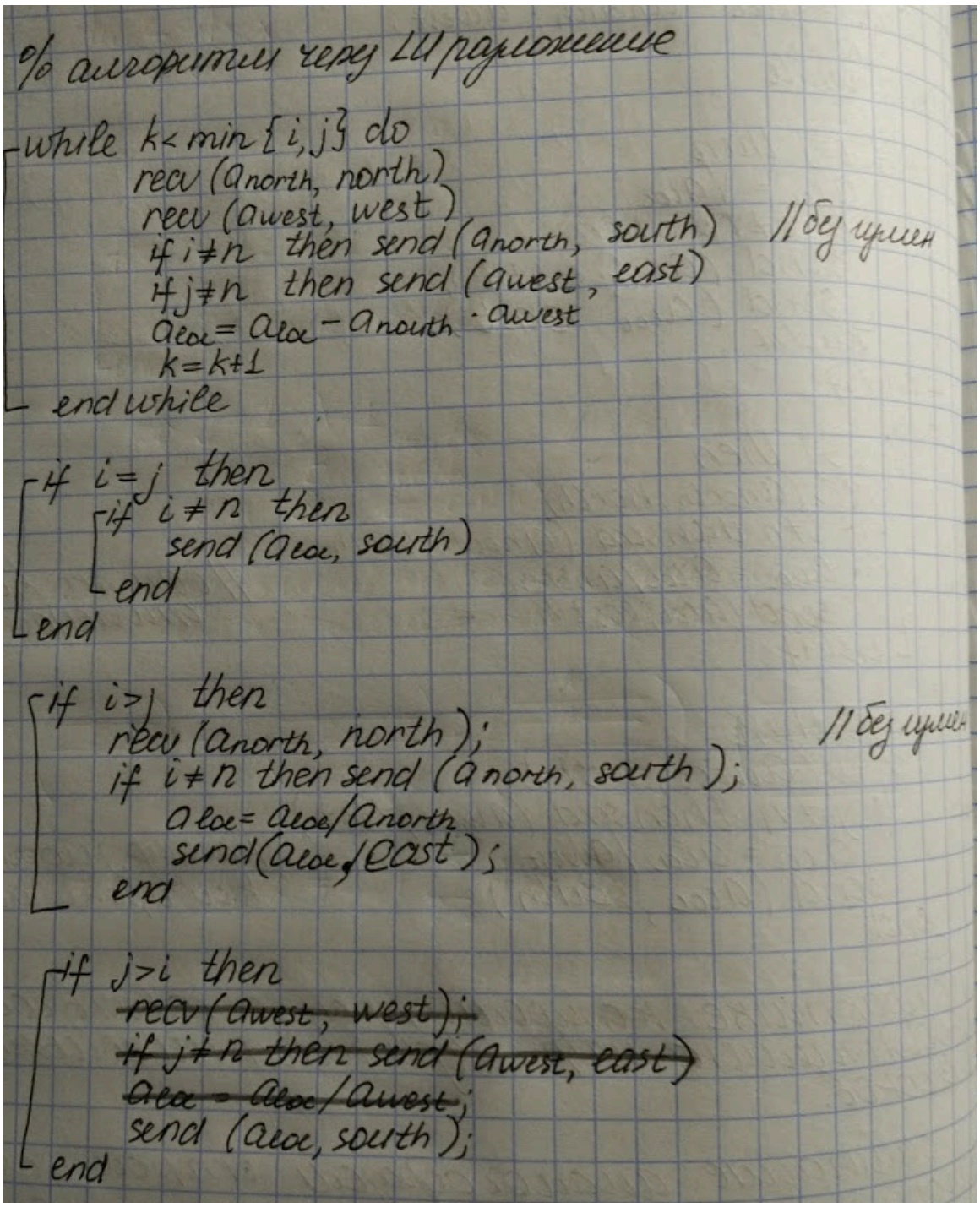
$\begin{pmatrix} 0\ 0\ 0\ 0 \dots 0 \\ 0\ 1\ 1\ 1 \dots 1 \\ 0\ 1\ 2\ 2 \dots 2 \\ \dots \\ 0\ 1\ 2\ 3 \dots n \end{pmatrix}$ – количество модификаций, которое сделает процессор перед выполнением деления или извлечения корня.

Алгоритм.

Инициализация $\{(i\,,j) - \text{номер процессора}\,; p - \text{количество процессоров}\,;$
$p = n^2\,; north\,, south\,, west\,, east - \text{соседи}\,; a_{loc} = A(i,j)\}$
k = 1
while k < min(i,j)
recv(a_{north} , north) ; recv(a_{west} , west)
if i < n
send(a_{north} , south)
endif
if j < n
send(a_{west} , east)
endif

$a_{loc} = a_{loc} - a_{north}a_{west}$
$k = k + 1$
endwhile
if $i == j$
$a_{loc} = sqrt(a_{loc})$
if $i < n$
send(a_{loc}, *south*) ; *send*(a_{loc}, *east*)
endif
elseif $i < j$
recv(a_{west}, *west*)
if $j < n$
send(a_{west}, *east*)
endif
$a_{loc} = a_{loc}/a_{west}$
send(a_{loc}, *south*)
elseif $i > j$
recv(a_{north}, *north*)
if $i < n$
send(a_{north}, *south*)
endif
$a_{loc} = a_{loc}/a_{north}$
send(a_{loc}, *east*)
endif

28 Алгоритм LU-разложения на процессорной решетке



```
% алгоритм через LU разложение

while k < min{i,j} do
      recv(a_north, north)
      recv(a_west, west)
      if i≠n then send(a_north, south)   // без циклов
      if j≠n then send(a_west, east)
      a_ea = a_ea - a_north · a_west
      k = k+1
end while

if i=j then
   if i≠n then
      send(a_ea, south)
   end
end

if i>j then
   recv(a_north, north);                 // без циклов
   if i≠n then send(a_north, south);
      a_ea = a_ea/a_north
      send(a_ea, east);
   end

if j>i then
   recv(a_west, west);
   if j≠n then send(a_west, east)
   a_ea = a_ea/a_west;
   send(a_ea, south);
end
```