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 Коммерсмита-Винограда работает за $O\left(n^{2.375477}\right)$, но на практике не используется из-за очень большой скрытой константы. На лекциях рассказывался небольшой лайфхак, который все равно работает за куб, но немного по другому.

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

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

и

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

соответственно (конкретные чиселки зависят от того, что мы считаем сложением, лучше в коде везде раскидать комментарии, сколько где производится операций). Это немного веселее, чем в обычном алгоритме, потому что сложение дешевле, чем умножение. Приведем сам код

```
// 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
```

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

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

Заметим, что таким образом нам требуется 7 умножений, а не 8. Более того, такой подход позволяет рекурсивно считать подматрицы, что хорошо прогается. Обычно такой подход продолжают до размеров матриц около 32-128, а потом используют наивный алгоритм, потому что данный метод теряет свою эффективность из-за бОльшего числа сложений, чем обычный.

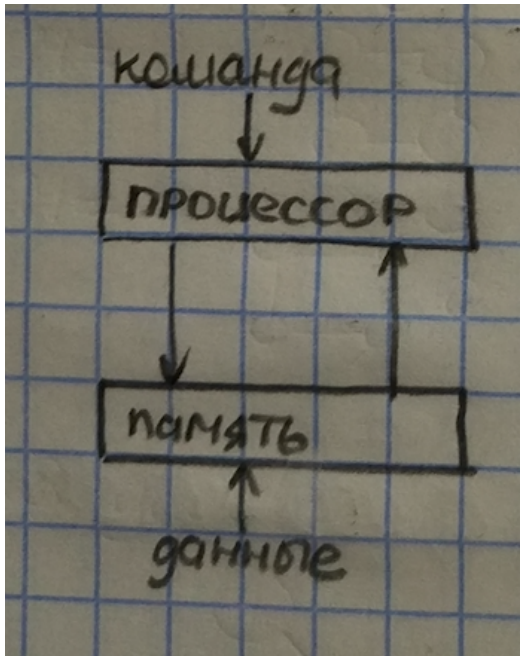
3 Классификация параллельных вычислительных систем по Флинну, модели представления параллельных алгоритмов

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:

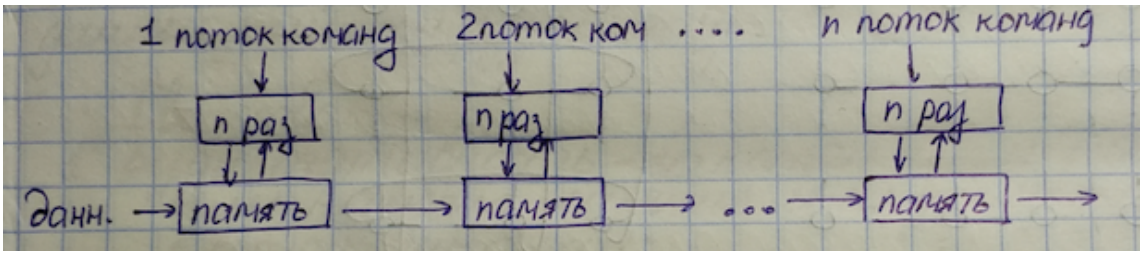
- Действия последовательны
- Данные обрабатываются последовательно

3.2 SIMD

Векторные ЭВМ

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

3.3 MISD

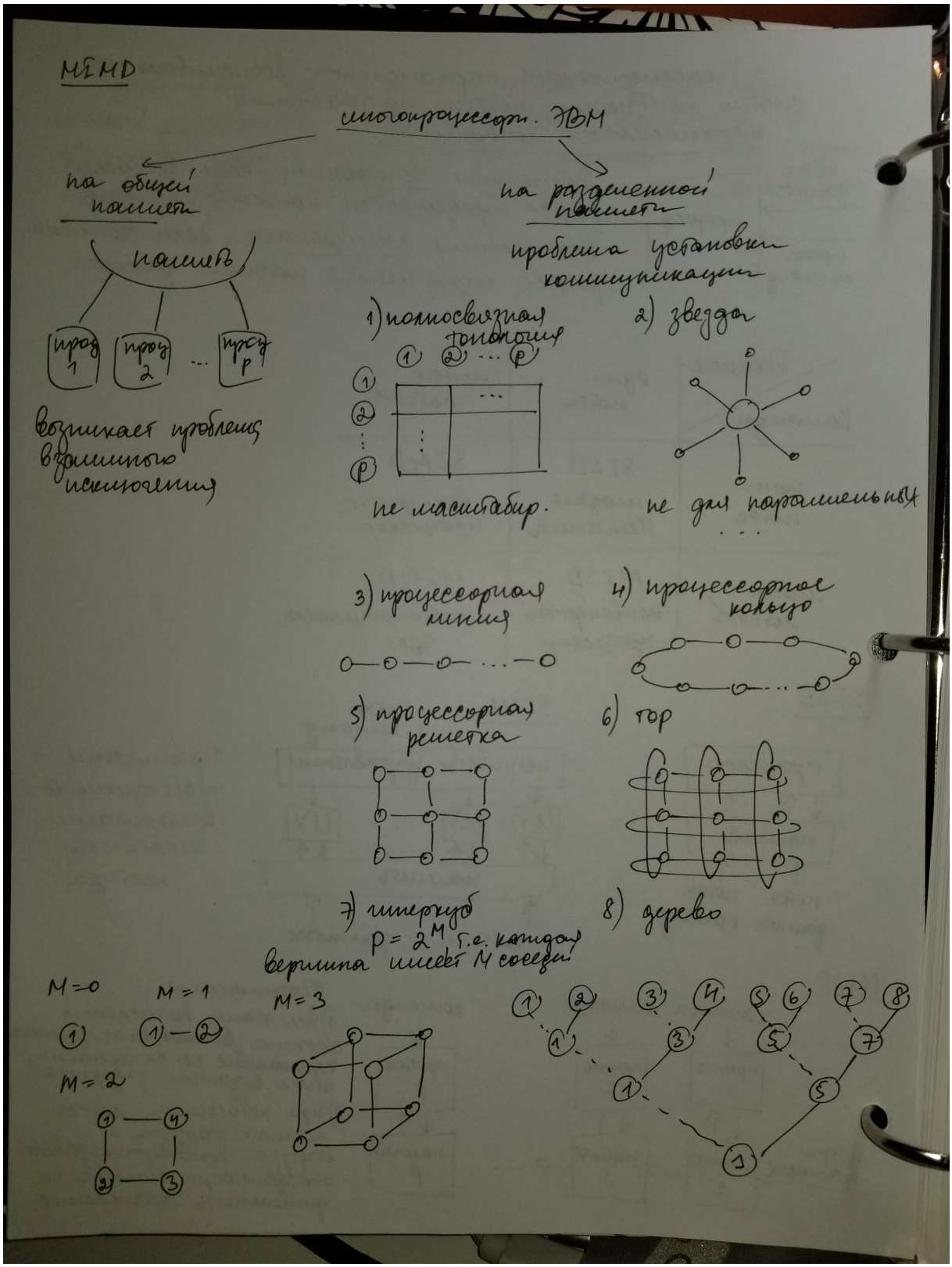


Конвейерный процессор. Конвейерный алгоритм предполагает разбиение задачи на подзадачи.

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
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

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:

$L_1 : c = \log c$

$L_2 : d = \log b$

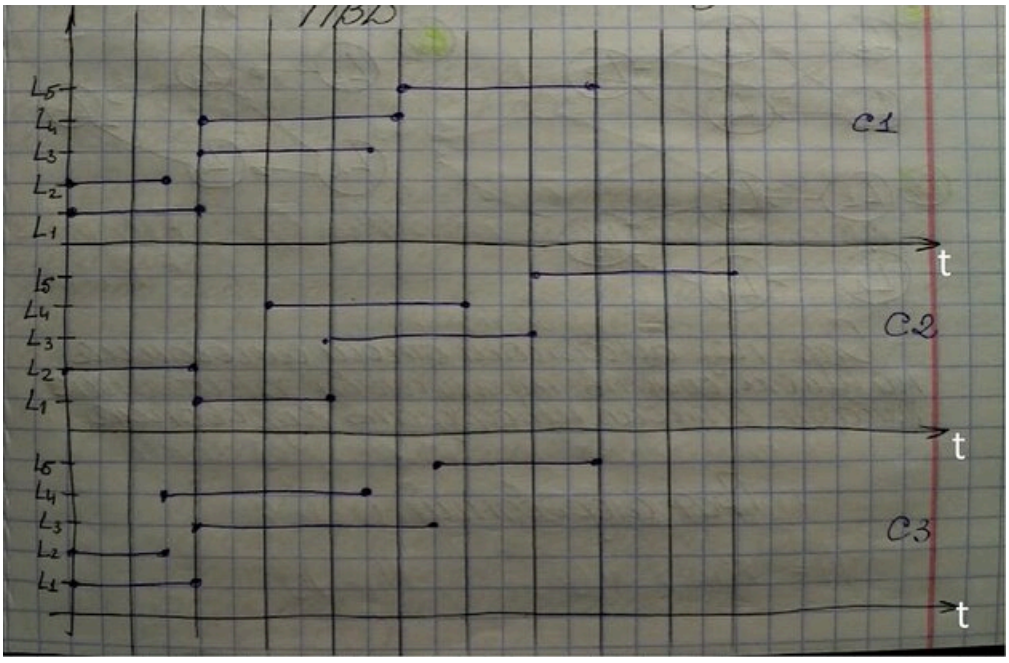
$L_3 : e = c + d$

$L_4 : f = d \times d$

$L_5 : g = e / f$

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

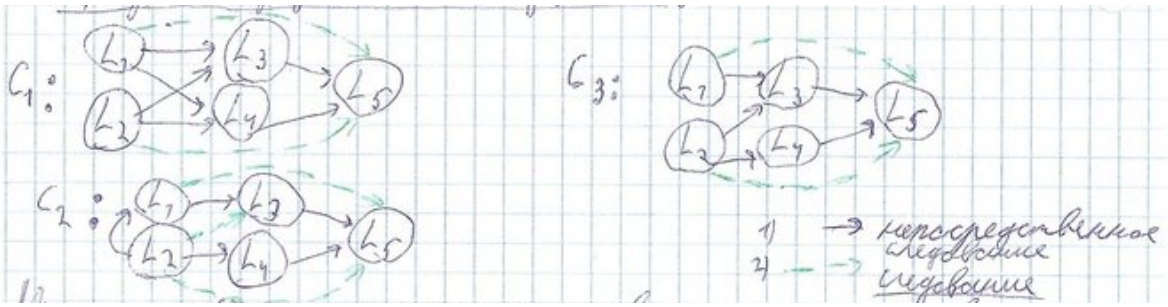
Поставим во взаимно однозначное соответствие любому оператору программы действие вычислительной системы по его исполнению, если таких действий несколько, примем их за одно.

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

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

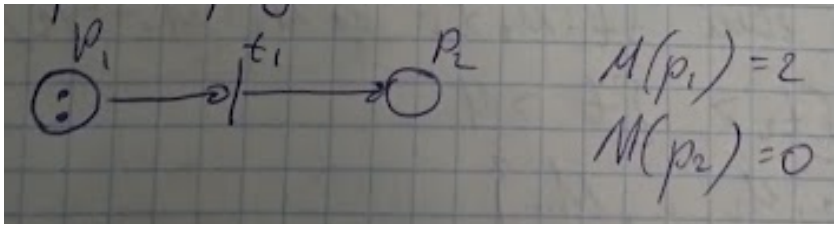
  p^* – множество входных переходов $p : \{t : F(t,p) \neq 0\}$,

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

5.3 Сети Петри

$PN = (P, T, F, M)$, где

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



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.

Векторная функция $M \geq F^*$

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

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 программ 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

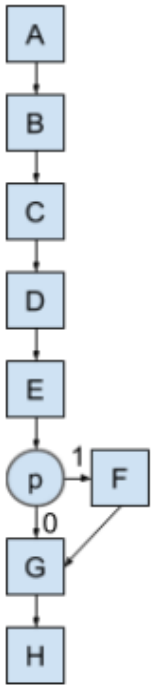
E: z = c + x

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

F: h = x, l = y

G: распечатка A(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).

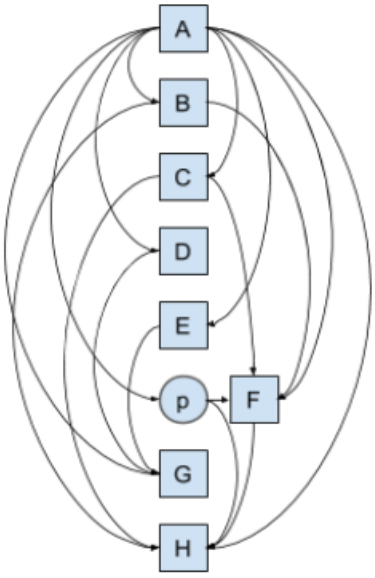
Определение. Будем говорить, что вершина B информационно зависит от вершины A, если они расположены на одном пути стандартного графа из входной вершины в выходную (причем A предшествует B) и хотя бы один оператор вершины B использует, но не изменяет значения переменной, которая формулирует хотя бы один оператор из вершины A.

$Out(A) \cap In(B) =/ \varnothing$

Кроме того, на указанном пути между A и B отсутствует вершина C, хотя бы один оператор которой изменяет значение упомянутой переменной.

Определение. 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.

Определение. 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»

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.

$$\left\{ I_q \right\}_{q=1}^{n}, \; I = \bigcup_{q=1}^{n} I_q$$

Такое покрытие в большинстве случаев не единственно. Важная характеристика – в любом подмножестве I_q все вектора независимые.

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

$S = \dfrac{\prod_{q=1}^{k} n_j}{n}$, которая является ускорением, где n – 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.

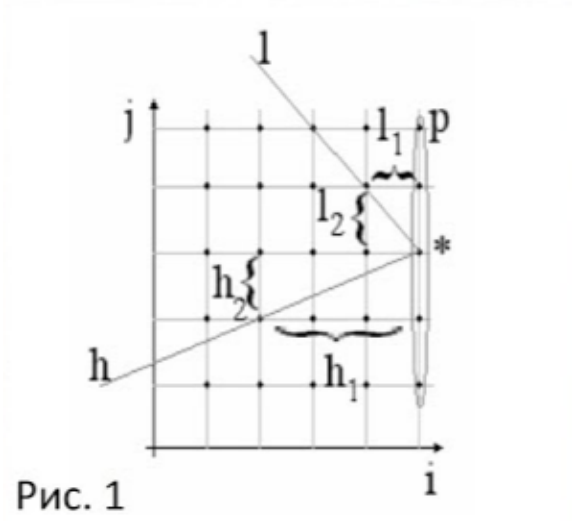


Рис. 1

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

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

11 Векторные алгоритмы gaxpy для симметричных и ленточных матриц

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

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

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

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

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

18 Векторные и блочные алгоритмы решения треугольных СЛАУ

19

20