OSEDUCONF-2023



ПРАКТИКУМ ПО ИЗУЧЕНИЮ СКРЫТОГО В АЛГОРИТМАХ ПАРАЛЛЕЛИЗМА И ЕГО РАЦИОНАЛЬНОГО ИСПОЛЬЗОВАНИЯ В ВЫЧИСЛЕНИЯХ

Ежегодная конференция Свободное программное обеспечение в Высшей Школе
Тема: Научные проекты, связанные с разработкой и использованием свободного программного обеспечения

Баканов Валерий Михайлович, РТУ МИРЭА / НИУ ВШЭ
915-053-5469, e881e@mail.ru, http://vbakanov.ru/left_1.htm


АКТУАЛЬНОСТЬ ПРОЕКТА:

- Развитие в России процессоров архитектуры, непосредственно ориентированной на параллелизацию вычислений (“первая ласточкой” можно считать процессоры семейства ЭЛЬБРУС архитектуры VLIW)
- Часто формальное и явно недостаточное для дальнейшего практического применения студентами преподавание кустá дисциплин ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ
- Недостаток “на местах” аппаратной части для освоения данных дисциплин

ПРЕДЛАГАЕМОЕ РЕШЕНИЕ (данный проект):

- Разработка набора программных компьютерных моделей для исследования явления параллелизма и его практического использования
- Свободная доступность разработок (как на уровне исходных кодов, так и исполняемых файлов) для всех заинтересованных лиц и учреждений
- Возможно полное покрытие всех сторон данной области знания (от формального выявления параллелизма в алгоритмах до его практического использования в вычислительных практиках)
- Разработка набора методических материалов, направленных на формальную и исследовательскую стороны данной области знания

ОРИЕНТИРОВАННОСТЬ ПРОЕКТА:

- В первую очередь на студентов – будущих разработчиков СИСТЕМНОГО ИНСТРУМЕНТАЛЬНОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ (конкретно – создание эффективных распараллеливающих блоков компиляторов / интерпретаторов)
- Обеспечение Научно-Исследовательской Работы (*Научных Семинаров*) студентов направлений, связанных с компьютерной обработкой данных

Минимальный список *наивных вопросов* по параллельным вычислениям, *количественные ответы* на которые необходимо получить для практического использования данной технологии

- Каково минимальное время *параллельного* выполнения *заданного (произвольного) алгоритма*? От каких параметров алгоритма зависит это время?
- При каком именно числе параллельных вычислителей обеспечивается этот *минимум времени выполнения*?
  - Истинно ли выражение: “чем параллельных вычислителей больше – тем алгоритм вы́полнится быстрее”?
    - Значит ли это, что при стремле́нии числа вычислителей к бесконечности время выполнения алгоритма может быть сколь угодно ма́лым?
- Как время выполнения алгоритма зависит от числа параллельных вычислителей?

- Более сложные вопросы, ответы на которые может быть эмпирически получен с применением методов моделирования параллельного выполнения алгоритмов:
  - Сохраняется ли присущий данному алгоритму закон вычислительной сложности при параллельном выполнении этого алгоритма?
    - Предложите наиболее простой и максимально показательный эксперимент, подтверждающий или опровергающий предложенную гипотезу по предыдущему вопросу.

Аппаратный и программный пути реализации выявления параллелизма в алгоритме

Подходы к выделению в программе параллельных участков и планирование их выполнения

Аппаратный путь *

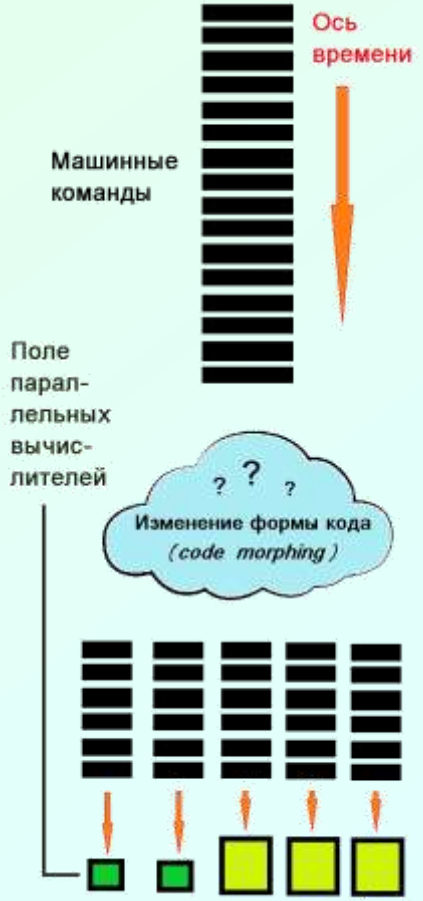
Достоинство: “обычный” компилятор

Недостатки: большие внутрипроцессорные накладные расходы на распараллеливание

Программный путь **

Достоинства: отсутствие внутрипроцессорных накладных расходов

Недостатки: сложный “smart” компилятор



* Пото́ковые вычислители (архитектура DATA-FLOW) - Джек Деннис, 1970. Опытные разработки: JUMBO (Великобритания), Manchester Dataflow Computer, Monsoon и Epsilon (США), CSRO (Австралия). Всеволод Бурцев (теория, эксперименты).

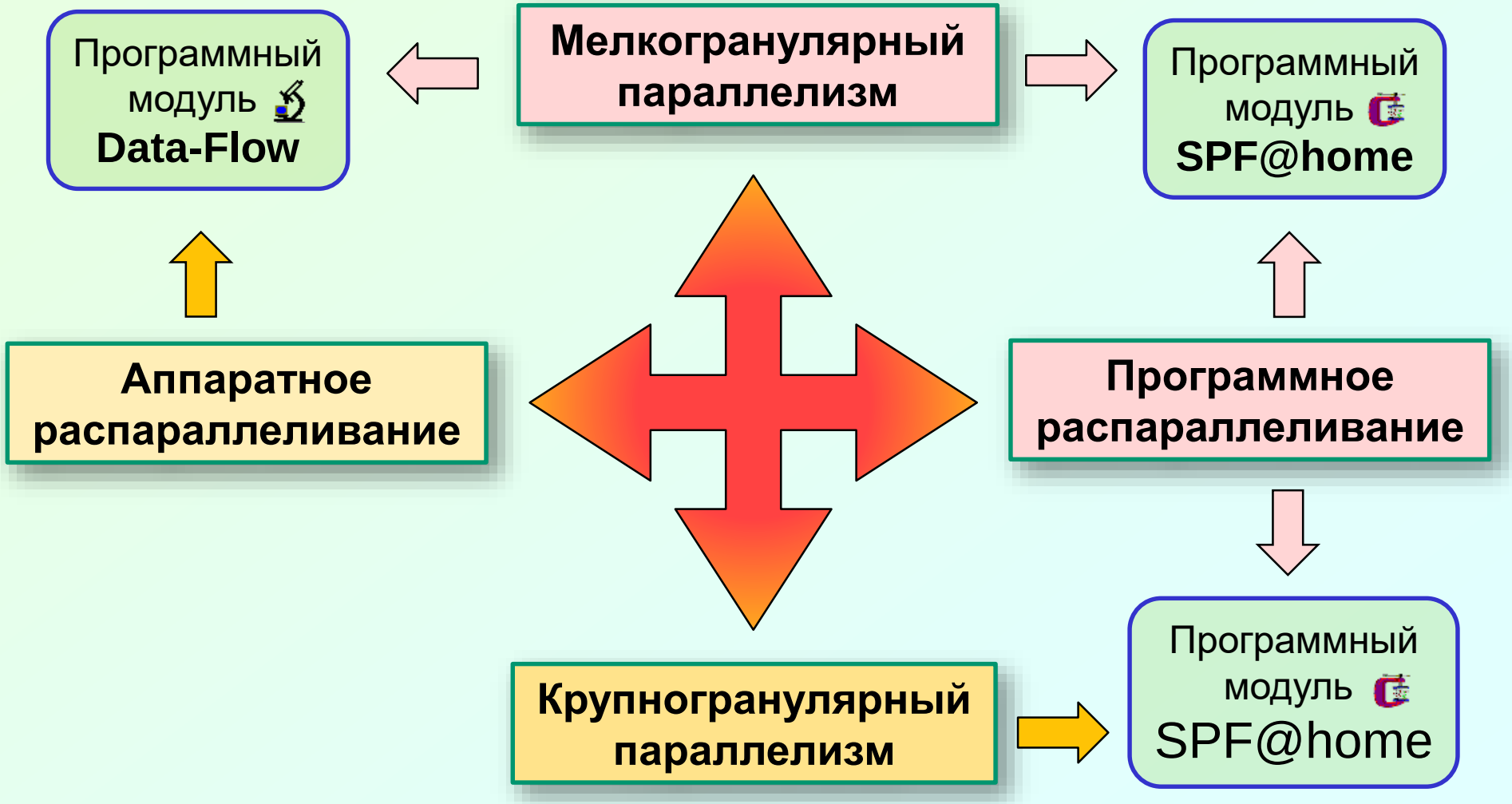
🛠 Одна из (многих) компьютерных моделей: здесь (инсталлятор, платформа Win’32, GUI) и здесь (описание).

💣 “Стихи” о DATA-FLOW здесь…

** Суперкомпьютеры “ЭЛЬБРУС” (Всеволод Бурцев, СССР, 70-е годы), ITANIUM (Intel, США, 90-е годы), Crusoe, Efficeon (Transmeta, США). Современные микропроцессоры “ЭЛЬБРУС” (ИНЭУМ, МЦСТ, Россия).

🛠 Одна из (многих) компьютерных моделей: здесь (инсталлятор, платформа Win’32, GUI) и здесь (описание).

Учёт многогранности рассматриваемой области знаний при условии минимизации сложности программного обеспечения для моделирования процессов



Ориентированность проекта на сущность **АЛГОРИТМ** *

📖 Грань **ВНУТРЕННИЙ (*скры́тый*) ПАРАЛЛЕЛИЗМ** – *относительно новая* исследуемая сторона алгоритма, изучающая наличие в алгоритме собственно параллелизма и его параметров (что необходимо для определения возможности его (параллелизма) использования при обработке данных на параллельных вычислительных системах.

CODE MORPHING (символическое изображение на рис. справа сверху) – преобразование кодовой последовательности из одного вида в другой. Одно из применений ***code morphing*** – преобразование кода из последовательного представления в параллельное с заданными параметрами.

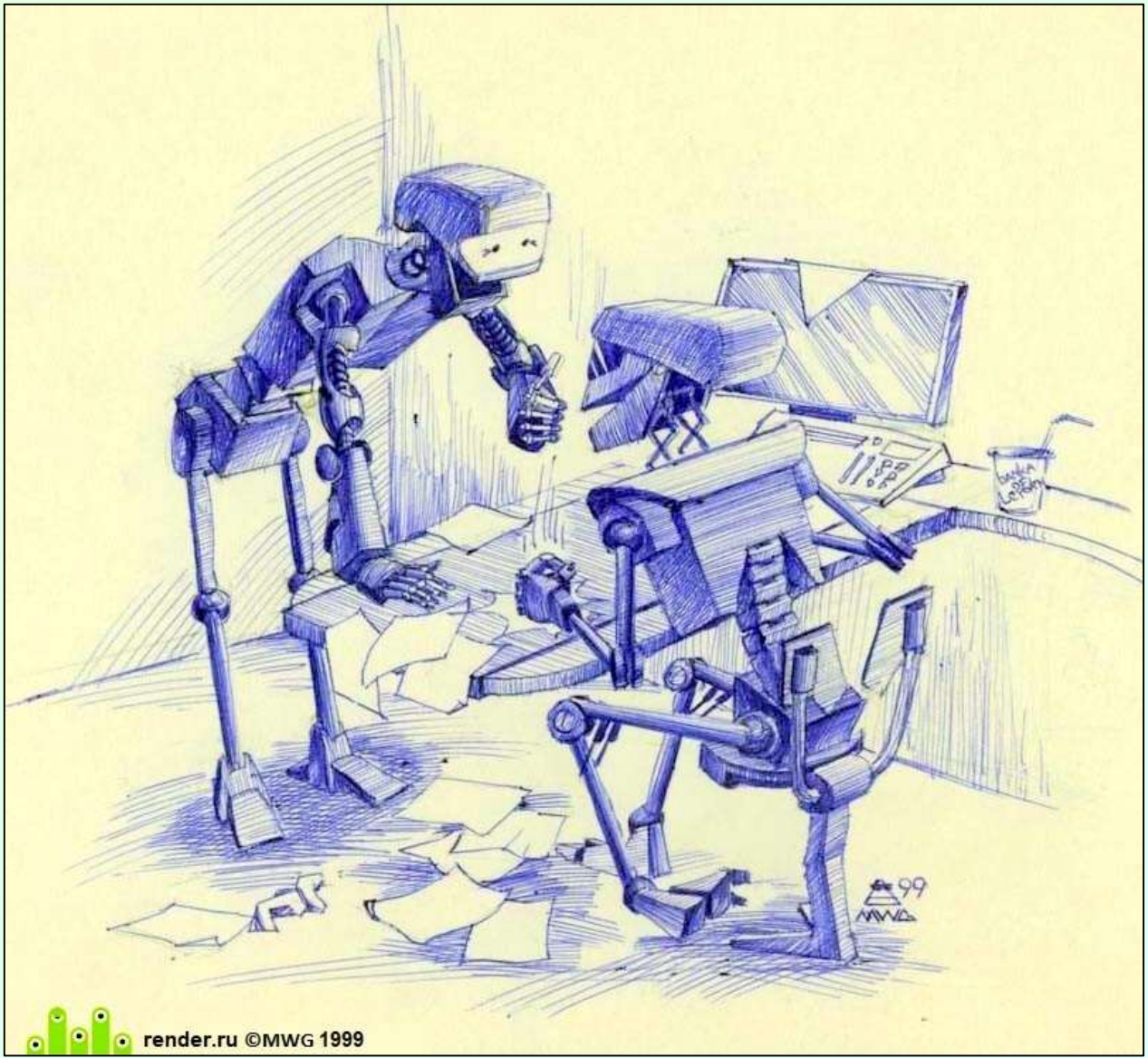
Собственно идея CODE MORPHING, пожалуй, впервые была реализована фирмой Transmeta Corp. при создании процессоров VLIW-архитектуры Crusoe (длина *сверхдлинного слова* 128 бит, 2000 год) и Efficeon (256 бит, 2004 год). В этом же ряду советско/российский проект ЭЛЬБРУС и ITANIUM (последний - совместная разработка Intel Corp. и Hewlet-Packard).

Анализ часто используемых алгоритмов представлен на WEB-ресурсе AlgoWiki http://algowiki-project.org/ru/. Полное название AlgoWiki - “Открытая энциклопедия свойств алгоритмов”; руководители: Воеводин Вл.В. (НИВЦ МГУ им. М.В.Ломоносова, РФ) и Джек Донгарра (разработчик теста LinPack - университет Теннеси, Knoxville, USA).

---

* О сложности и многоальтернативности понятия АЛГОРИТМ – здесь (и дополнительно)...

“Великие механи́кусы” Трурль и Клапау́ций о роли **АЛГОРИТМА** в жизни и творчестве



📖 – По правде, – бурчал Трурль, – надо бы как-то иначе скомбинировать… Впрочем, важнее всего алгоритм!

– Тоже мне открытие сделал! Известно, без алгоритма ни шагу ступить! Ну не́чего, надо экспериментировать!

”Семь путешествий Трурля и Клапауция. Путешествие второе, или Какую услугу оказали Трурль и Клапауций царю Жестокусу”. Станислав Лем, КИБЕРИáДА. 1964÷1979.

Общая схема создания параллельного приложения при использовании *программного пути* выявления и *рационального* использования параллелизма в алгоритме (программе)

Алгоритм (программа) выполнения заданных действий

Алгоритм (программа) могут быть заданы в *императивной* или *функциональной* формах

1. Выявление в алгоритме (программе) внутреннего (скрытого) параллелизма

2. Построение рационального плана (расписания) параллельного выполнения

3. Создание исполняемой программы согласно разработанному плану выполнения с применением выбранной технологии и языка программирования

Использование специальных методов и приёмов выявления параллелизма

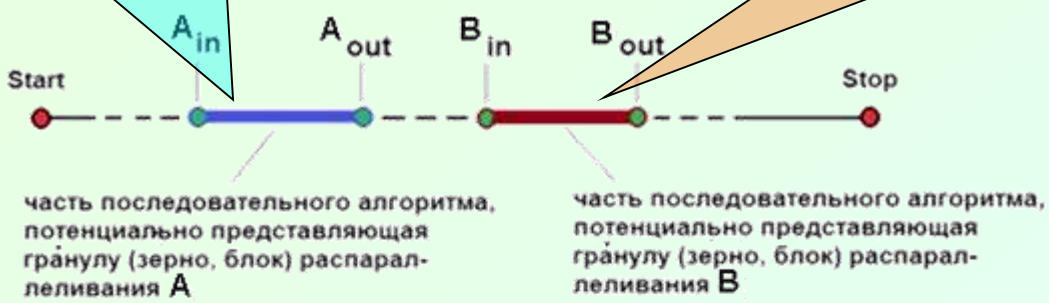
Преобразований параметров выявленного параллелизма с учётом параметров архитектуры параллельной вычислительной системы

В некоторых случаях этапы 1 и 2 совмещаются

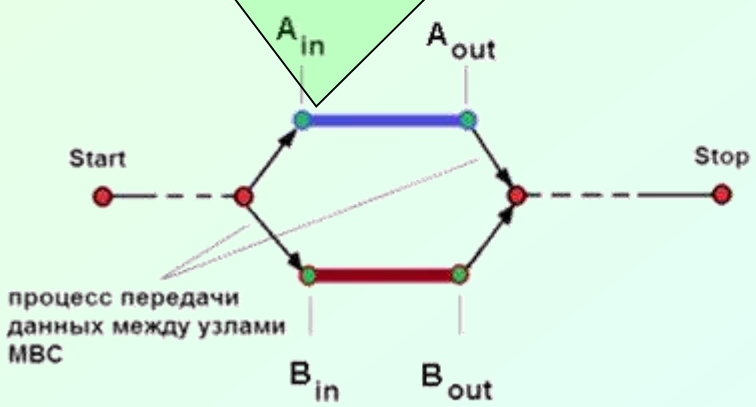
Каковы *наиболее общие подходы* для преобразования последовательного алгоритма в параллельный?

A_{in} – входные данные, A_{out} - выходные (вычисленные) данные последовательности команд A

B_{in} – входные данные, B_{out}^{j} - выходные (вычисленные) данные последовательности команд B



Если входные данные для блока (*зерна, гранулы*) j не зависят от выходных данных блока i, то эти блоки могут выполняться ***независимо*** (т.е. **ПАРАЛЛЕЛЬНО**)



Условием независимого (параллельного) выполнения отдельных частей (блоков, зёрен, гра́нул) алгоритма является их ***независимость по данным*** (если B_{in} зависит от A_{out}, то блок B не может выполняться параллельно с блоком A; в противоположном случае - может).

Проблема – такое разбие́ние (*декомпозиция*) на гранулы затруднено, ибо границы блоков априори неизвестны и подлежат определению, при этом распараллеливание каждого изначально-последовательного алгоритма возможно ***множеством способов***, а эффективность (в основном – время выполнения программы) каждого из способов может ***отличаться на порядки***.

Для решения конкретной задачи распараллеливания необходимо решить вопросы:

1) Каков размер *гранулы параллелизма* (от одной машинной команды до тысяч/миллионов)?
2) В каком конкретно месте последовательной программы находятся эти *гранулы*?

Общее число сочетаний таково́, что обычно приводит к *NP*-полной задаче…

Выявле́ния параллелизма в алгоритмах и использование параллелизма при вычислениях

Алгоритм является результатом разумной деятельности человечества и отражает в себе (в опосре́дованном виде, конечно) наиболее глуби́нные, фундаментальные законы развития Природы. Одно это является обоснованием необходимости исследования характеристик алгоритмов.

Валерий Баканов. Крым, Щёлкино / Казантип, август 2022.

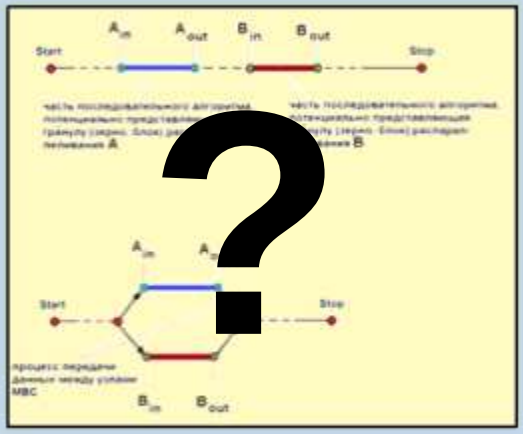
Алгоритм можно наглядно представить в виде “облака операторов” (см. рис. сверху слева ⬆). Такой подход максимально соответствует определению алгоритма как “*набора инструкций по преобразованию одних данных в другие за конечное число действий*”.

🛠 Имея “облако операторов”, остаётся ***реализовать механизм*** (*аппаратного* или *программного* уровня) выбора из него допусти́мых (или выбранных по определённому критерию) для исполнения на каждом из (свободных в данный момент) устройств в имеющемся поле параллельных вычислителей.

🕮 Заметим, что в реальности **порядок выполнения операторов** *может быть разным* (как при последовательном, так и при параллельном исполнении).

Именно этот механизм при́зван преодолеть пресловутый “семантический разрыв” между внутренним потенциалом параллелизма в алгоритме (программе) и параметрами конкретной параллельной вычислительной системы.

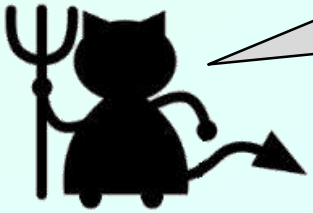
Формальные методы и приёмы выявления параллелизма в алгоритмах



Федотов И.Е. Параллельное программирование. Модели и приёмы. — М.: СОЛОН-Пресс, 2018. - 390 с. (ссылки)

Методы и приёмы выявления параллелизма в алгоритмах:

- Построение *я́русно-параллельной формы* (ЯПФ) информационного графа алгоритма (ИГА)
- Использование сетей конечных автоматов
- Применение сетей Пе́три
- Модель а́ктеров (*англ. actor "актёр - действующий субъект"*)



Механизм преодоления "семантического разрыва"

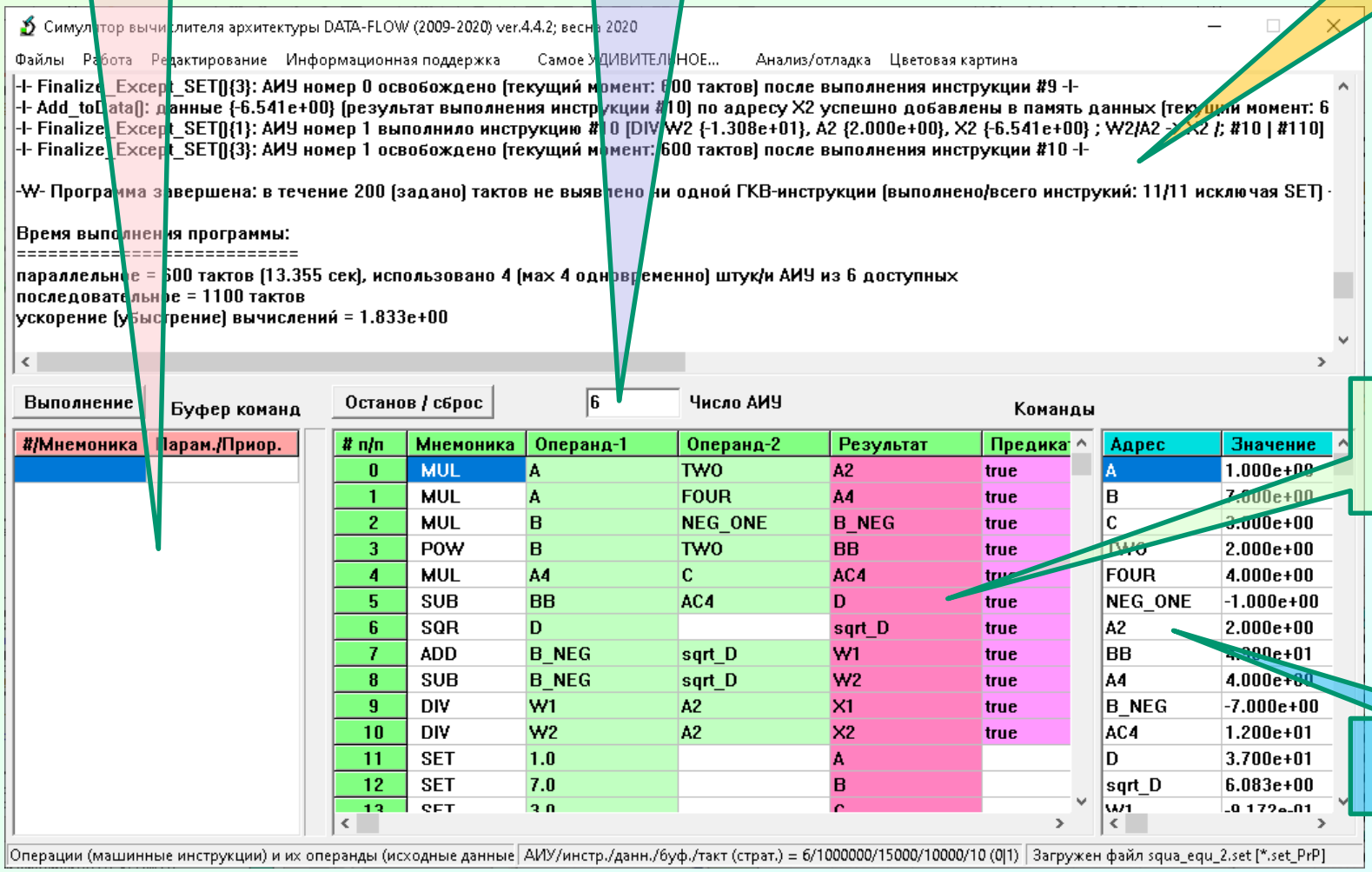
В данной работе используется в основном первый из описанных методов (построение и *целенаправленное изменение* ЯПФ).

Текущая (версия “зима 2022-2023 г.г.) реализация программы моделирования (*симуляции*) потóкового вычислителя DATA_FLOW

Окно содержимого буфера команд

Число параллельных вычислителей

Окно вывода протокола решения задачи



Окно памяти исполняемых инструкций

Окно памяти данных

Использование предикатов для условного выполнения операторов (программа *SQUA_EQU_2.PRED.SET*)


```
MUL A, TWO, A2, !false ; A2 ← 2 * A
MUL A, FOUR, A4 ; A4 ← 4 * A
MUL B, NEG_ONE, B_NEG ;  B_NEG ← NEG_ONE * B
POW B, TWO, BB ; BB ←  B^2
MUL A4, C, AC4 ; AC4 ← A4 * C
SUB BB, AC4, D ;  D[iskriminant] ← BB - AC4
SQR D, sqrt_D, IS_re ; ← sqrt(D) -> sqrt_D
ADD B_NEG, sqrt_D, W1, IS_re ; W1 ← B_NEG + D_SQRT
SUB B_NEG, sqrt_D, W2, IS_re ; W2 ← B_NEG - D_SQRT
DIV W1, A2, re_X1, IS_re ; re_X1 ← W1/A2
DIV W2, A2, re_X2, IS_re ; re_X2 ← W2/A2
MUL D, NEG_ONE, NEG_D, !IS_re ; NEG_D ← NEG_ONE x D
SQR NEG_D, sqrt_D, !IS_re ; sqrt_D ← sqrt(NEG_D)
DIV B_NEG, A2, re_X1, !IS_re ; 1-th root (real)
DIV sqrt_D, A2, im_X1, !IS_re ; 1-th root (img)
CPY re_X1, re_X2, !IS_re ; 2-th root (real)
DIV sqrt_D, A2, W, !IS_re ; temp for im_X2
MUL W, NEG_ONE, im_X2, !IS_re ; 2-th root (im)
SET 1, A ; A ← 1
SET 3, B ; B ← 7/3 (re / im)
SET 3, C ; C ← 3
SET 2,  TWO  ; TWO ← 2
SET 4,  FOUR ; FOUR ← 4
SET -1, NEG_ONE ; NEG_ONE ← (-1)
SET 0, ZERO ; ZERO ← 0
PGE D, ZERO, IS_re ; IS_re ← true if D>=0
```

Использована команда PGE, устанавливающая флаг предиката IS_re в зависимости от *соотношения значений переменных **D** и **ZERO***

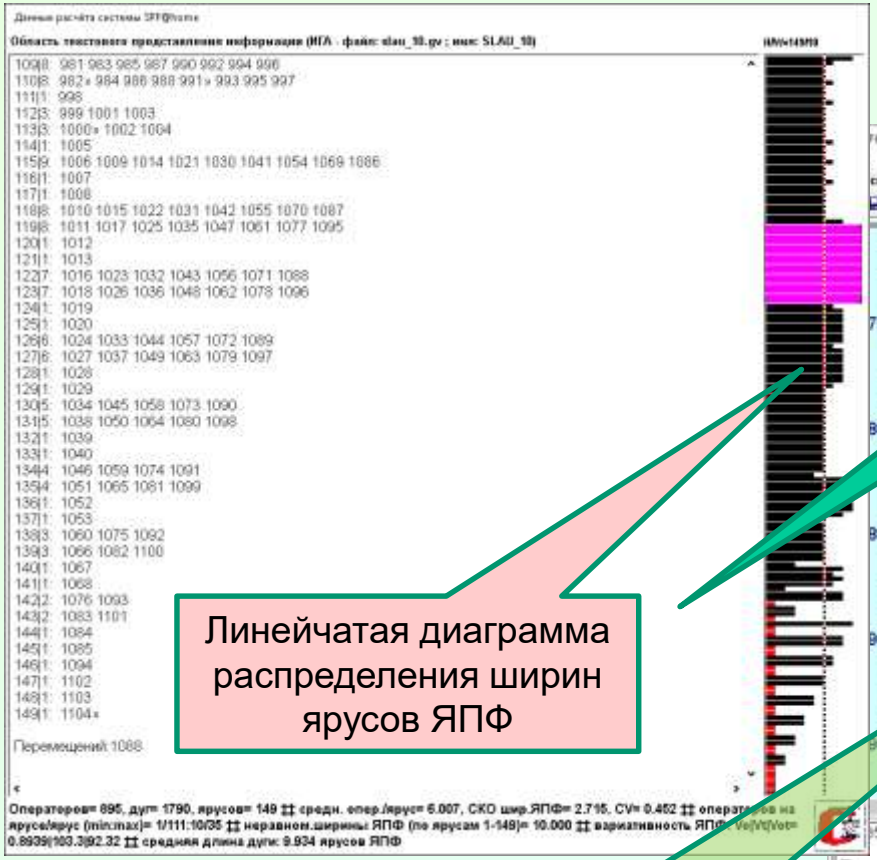
Пояснения. В качестве флага-предиката здесь используется **IS_re**. Командой **PGE D, ZERO, IS_re** флаг **IS_re** устанавливается в ‘true’ при условии D≥0 (здесь D – дискриминант полного квадратного уравнения) или в ‘false’ в противном случае; далее вычисления производятся в зависимости от значения флага **IS_re** (четвёртое поле команды)

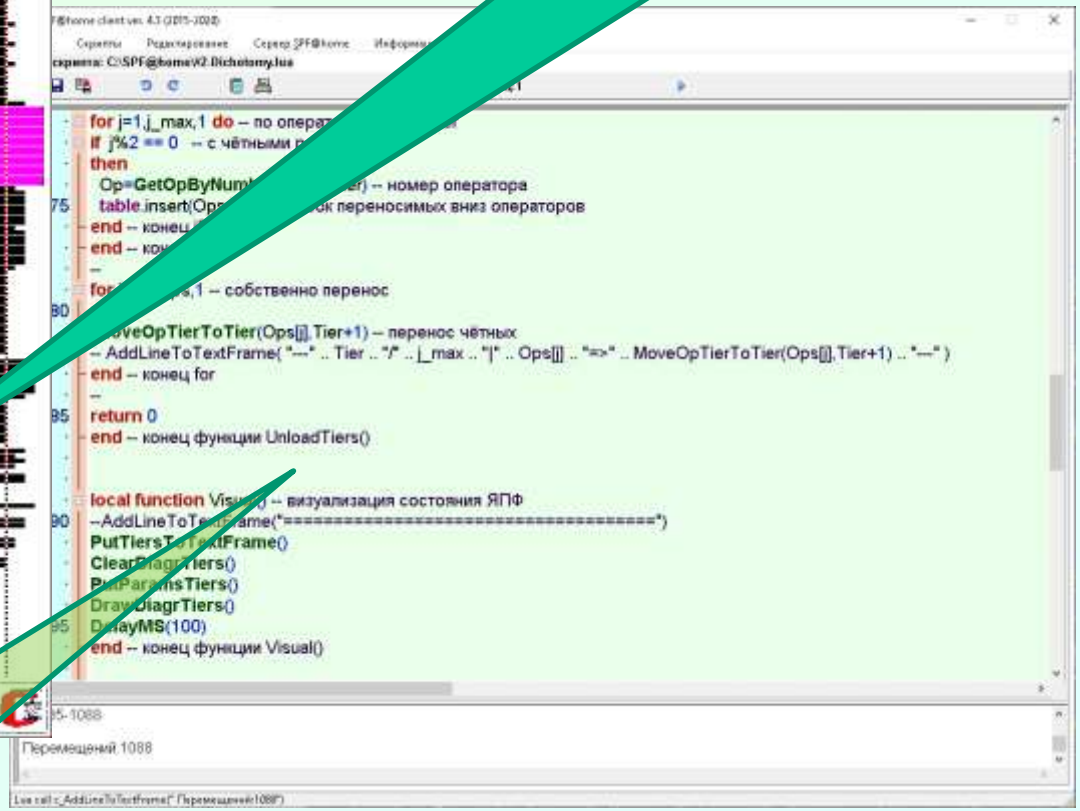
Список некоторых алгоритмов для исследования на наличия параллелизма и его параметров

Умножение квадратных матриц классическим способом (общее название m_matr_N.set, где *N* – порядок матриц)	m_matr_2.set, m_matr_3.set, m_matr_5.set, m_matr_7.set, m_matr_10.set
Умножение квадратной матрицы на вектор (общее название m_matr_vec_N.set, где *N* – порядок матриц)	m_matr_vec_2.set, m_matr_vec_3.set m_matr_vec_5.set, m_matr_vec_7.set, m_matr_vec_10.set
Решение систем линейных алгебраических выражение простым (безитерационным) методом Гаусса (общее название slau_N.set, где *N* – порядок системы)	slau_2.set, slau_3.set, slau_4.set, slau_5.set, slau_7.set, slau_10.set
Аппроксимация точек прямой методом наименьших квадратов (общее название mnk_N.set, где *N* – число точек при аппроксимации)	mnk-5.set, mnk-10.set, mnk-15.set, mnk-20.set
Аппроксимация точек параболой методом наименьших квадратов (общее название mnk-2_N.set, где *N* – число точек при аппроксимации)	mnk-2_5.set, mnk-2_10.set, mnk-2_15.set, mnk-2_20.set
Вычисление коэффициента парной корреляции (общее название korr_N.set, где *N* – число точек)	korr_5.set, korr_10.set, korr_15.set, korr_20.set
Вычисление первых 10 чисел Фибоначчи, “трибоначчи”, “квадроначчи”	fibonn_10.set, tribonn_10.set, quadronn_10.set
Вычисление полинома 10-го порядка (метод прямого возведения чисел в степень, метод возведения в степень последовательным умножением)	polinom_10-1.set, polinom_10-2.set
Сложение 32 чисел методом сдва́ивания	doubling_32.set
Нахождение максимума чисел в одномерном массиве из 8 чисел (метод последовательного сравнения пар чисел, алгоритм сдва́ивания)	max-1_mass-8.pred.set, max-2_mass-8.pred.set
Нахождение корней полного квадратного уравнения (только действительные корни, комплексные корни)	squa_equ_2.set, squa_equ_2.pred.set

Текущая реализация клиентской части проекта SPF@home

Дочернее окно выдачи результатов расчётов в текстовом и графическом видах





Линейчатая диаграмма распределения ширин ярусов ЯПФ

Главное окно разработки и отладки Lua-скриптов и управления их выполнением

Как постро́ить *Ярусно-Параллельную Форму* (ЯПФ) для заданного *Информационного Графа Алгоритма* (ИГА) ?

📖 Исходные данные: информация о ***вершинах графа*** + информация об ***объединяющих вершины ду́гах*** (с уточнением – эта дуга является *входя́щей* или *исходя́щей* относительно данной вершины графа). **Следует учесть, что общее число ярусов ЯПФ априори неизвестно** *(для теоретиков – определяется длиной критического пути в графе)* !..

🛠 Собственно алгоритм постро́ения ЯПФ (в данном случае ЯПФ строится по направлению вектора времени – от начала к концу выполнения программы; возможен и противоположный вариант) тако́в:

$1. Найти среди всех вершин такие, у которых ***нет входя́щих*** дуг. Это будет ярус номер #0 - ярус исходных данных алгоритма.

$2. Среди оставшихся вершин найти такие, которые по дугам (*информационным зависимостям*) зависят **ТОЛЬКО от вершин, размещённым на предыдущих ярусах**. Найденные вершины поместить на следующий после предыдущего ярус.

Действия по результатам выполнения $2:

Вариант 1). Повторять действия $2, пока соответствующие условиям $2 вершины графа найдутся.

Вариант 2). Если найденные вершины ***не имеют выходя́щих*** дуг, то поместить их на последний ярус ЯПФ (это ярус *вычисления выходных величин алгоритма*). Процесс построения **канонического вида** ЯПФ завершён.

Использование API-вызовов скриптового языка Lua для *целенаправленного изменения* ЯПФ как *плана параллельного выполнения алгоритма (программы)*

Приведён пример программы (*сценария, скриптá*) на языке Lua для получения ЯПФ из gv-файла, запоминания его во внутреннем представлении системы SPF@home, создания ЯПФ и запоминания его в Lua-массиве для дальнейшей обработки. Жирным зелёным цветом выделены API-вызовы системы SPF@home, красным – ключевые слова Lua, двойной дефис означает начало комментария до конца строки):

```
CreateTiersByEdges("EdgesData.gv") -- создать ЯПФ по файлу
-- EdgesData.gv -- с подтя́нутостью операторов “вверх”
-- CreateTiersByEdges_Bottom("EdgesData.gv") -- создать ЯПФ по
-- файлу EdgesData.gv -- с подтя́нутостью операторов “вниз”
--
OpsOnTiers={} -- создаём пустой 1D-массив OpsOnTiers
for iTier=1,GetCountTiers() do – цикл по ярусам ЯПФ
 OpsOnTiers[iTier]={} -- создаём iTier-тую строку 2D-массива OpsOnTiers
  for nOp=1,GetCountOpsOnTier(iTier) do -- по порядковым номерам
    -- операторов на ярусе iTier  по списку слева направо
   OpsOnTiers[iTier][nOp]=GetOpByNumbOnTier(nOp,iTier) – получить актуальный
    -- номер оператора nOp на ярусе iTier по списку слева направо
end end -- конец циклов for по iTier и for по nOp
```

Полученная ЯПФ как (“*сырóй*”) план параллельного выполнения алгоритма (*программы*) и возможность ***целенаправленного изменения*** ЯПФ

📖 При представле́нии информационного графа алгоритма (ИГА) в виде ЯПФ на каждом ярусе располагаются операторы, зависящие (по *операндам*) только от операторов (*результатов их выполнения*), находящихся на ярусах ***выше данного***. Вычислительная сложность создания ЯПФ $O(N^2)$, где N – общее число операторов (вершин графа).

🔔 ЯПФ фактически является *наивным* (т.е. *примитивным, нуждающимся в улучшении*) планом (расписанием) параллельного выполнения алгоритма (программы).

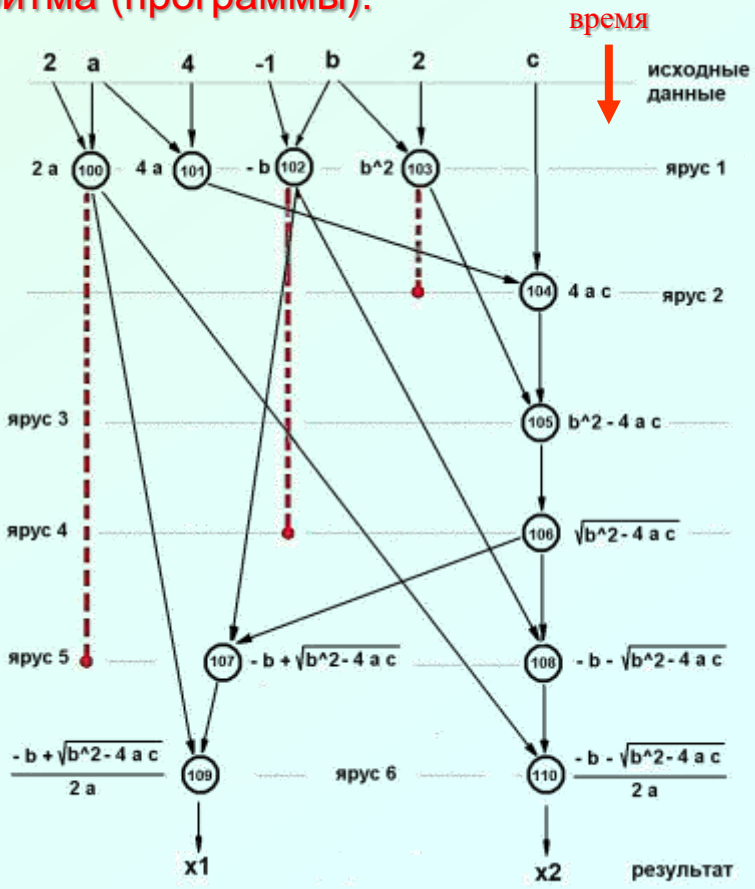
♦ Находящиеся на каждом ярусе операторы ***могут быть выполнены параллельно***, т. к. они информационно независимы.

♦ Насколько быстрее (по сравнению с *последовательным* вариантом) при этом выполнится данный алгоритм? Ответ ясен – в 11/6 ≈ 1,83 раза!

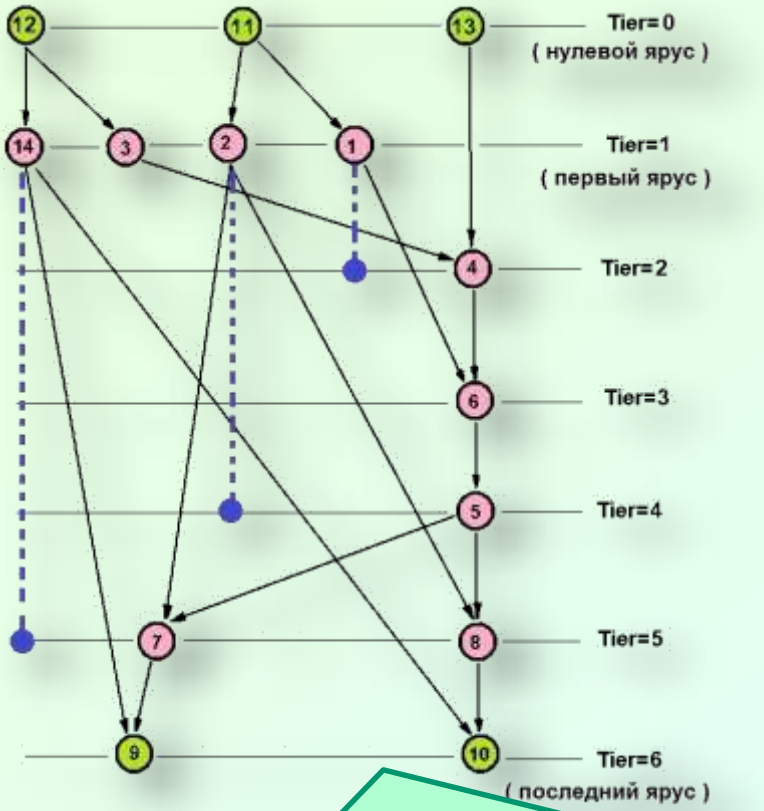
♦ **Быстрее выполнить не получится, ибо число ярусов ≡ *длине критического пути* в ИГА…**

♦ Что видим? На 1-м ярусе необходимо задействовать 4 параллельных вычислительных узла, на 5 и 6 – 2 узла, на 2,3,4 ярусах – по одному. Такая неравномерность использования ресурсов – плохо..!

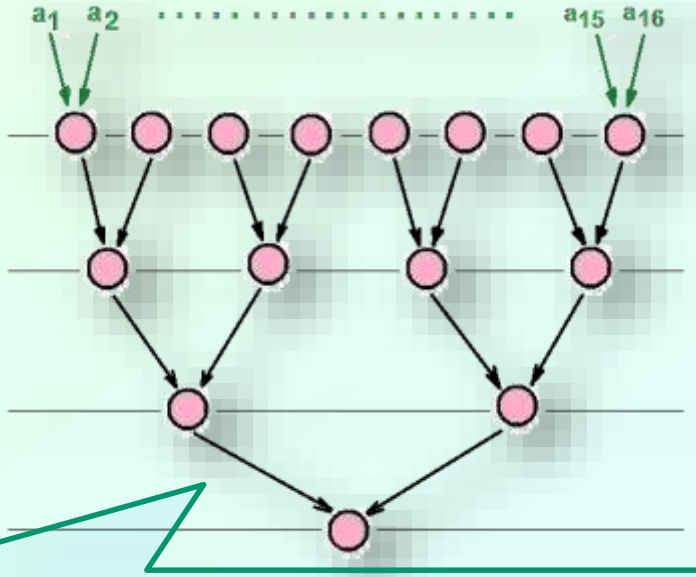
♦ **Красным пунктиром показано допусти́мое расположение операторов по ярусам ЯПФ.** При этом для выполнения данного алгоритма достаточно всего 2-х параллельно работающих вычислительных узлов, причём при этом время параллельного выполнения остаётся неизменным (число ярусов ЯПФ не увеличивается).



Вариативность размещения операторов по ярусам в ЯПФ информационного графа алгоритма



ось вре́мени выполнения алгоритма



🛠 Показан граф операции вычисления ***методом сдва́ивания*** (справедлив для ассоциативных операций). Важно, что возможности перемещения операторов между ярусами нет (***вариативность нулевая***).

🛠 На рис. показан пример Информационного Графа Алгоритма (ИГА) в Ярусно-Параллельной Форме (ЯПФ), причём исходные данные подаются на входы 12, 11, 13, выходами являются 9, 10. На самом деле это граф программы решения полного квадратного уравнения в вещественных числах (впервые предложен индийским математиком Брахмагупта, 598-670 г.г. н.э.).

🛠 Здесь же синими линиями показан возможный диапазон перемещений операторов с яруса на ярус (***вариативность***). Нетрудно видеть, что благодаря “разгру́зке” 1-го яруса от операторов 14 и 2 данный алгоритм можно выполнить на двух *(вместо 4-х… **SIC !!!**)* параллельно работающих вычислителях !

Какие задачи построения *рациональных* планов выполнения программ на заданном поле параллельных вычислителей можно решать с помощью ПРАКТИКУМА ? *(1)*

1. За основу при постро́ения Плана Параллельного Выполнения (ППВ) целевого алгоритма (программы) берём ранее сформированную ЯПФ информационного графа этого алгоритма (*при этом считаем, что последовательно выполняются группы операторов, расположенные на ярусах ЯПФ начиная с начального и до конечного*). Такой ППВ назовём сыры́м (наи́вным) – т.е. нуждающимся в определённом *целенаправленном улучшении*.

2. Параметры ”сырого” ППВ алгоритма обычно плохо ложатся на имеющуюся архитектуру параллельной вычислительной системы (срабатывает пресловутый ***семантический разры́в*** между описанием алгоритма и архитектурных принципов исполняющей вычислительной системы).

Важной проблемой параллельного выполнения алгоритмов (программ) является вопрос **наиболее полного использования имеющихся вычислительных ресурсов** (фактически всех из присутствующих в наличии вычислителей ПВС (***параллельной вычислительной системы***). Напр., практически всегда ширина ярусов ЯПФ значительно прквышает числа параллельных вычислителей данной ПВС. С другой стороны, при слишком малой ширине ярусов ЯПФ часть вычислителей будет простаивать (распараллеливающий компилятор/ интерпретатор будет вынужден фиктивно нагружа́ть эти вычислители вставкой команд *NOP* (No OPeration, “*операция-пусты́шка*”); в результате суммарная производительность ПВС снижается.

Какие задачи построения ***рациональных*** планов выполнения программ на заданном поле параллельных вычислителей можно решать с помощью ПРАКТИКУМА ? *(2)*

3. Логически разумный вариант при таком подходе – *целенаправленно* изменить ППВ (фактически “подогна́ть” форму ЯПФ под имеющее ППВ). В данном ПРАКТИКУМЕ это совершается путём целенаправленного изменения (при сохранении информационных связей между операторами) ЯПФ под управлением сценариев на скриптовом языке Lua.

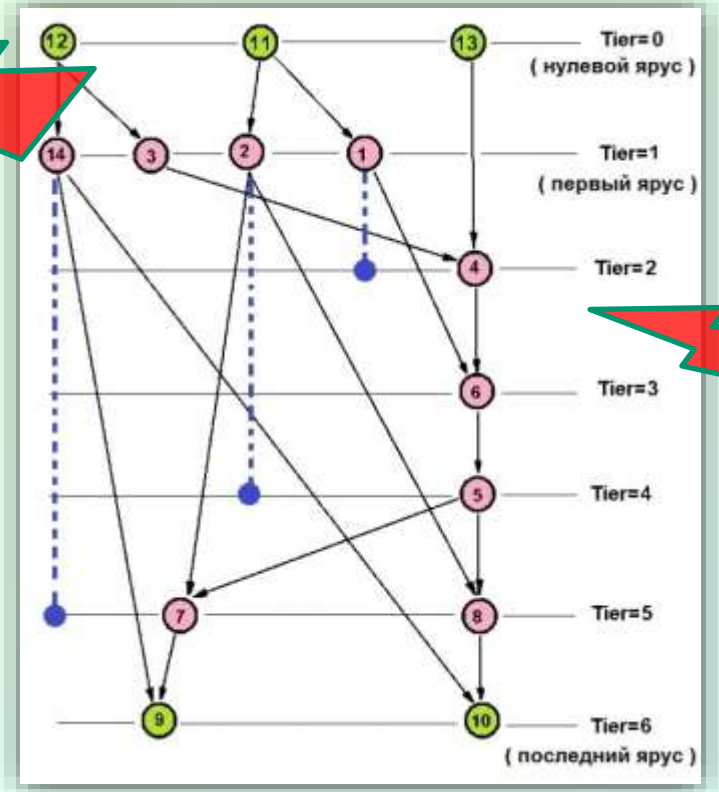
3а. Однако задача составления расписания – типично *NP*–полная задача, точных алгоритмов решения которых неизвестны. Поэтому выбран вариант эвристического (*созидательного, творческого,* часто *интуитивного*) подхода к разработке сценариев целенаправленного реформирования ЯПФ.

4. Важным требованием является ***минимизация времени отработки алгоритма*** собственно преобразования ЯПФ (возможно быстрая компиляция приложения).

5. Ещё одним дополнительным требованием может служить, например, ***минимизация объёма вре́менной локальной памяти вычислителя*** во время выполнения целевого алгоритма (согласно требования максимума быстродействия - это внутренние регистры процессора - ибо они *весьма дорогосто́ящи*).

Этапы реализации процедуры целенаправленного изменения ЯПФ как плана параллельного выполнения исследуемого алгоритма

Получить (с помощью *информационных функций*) данные о ЯПФ графа



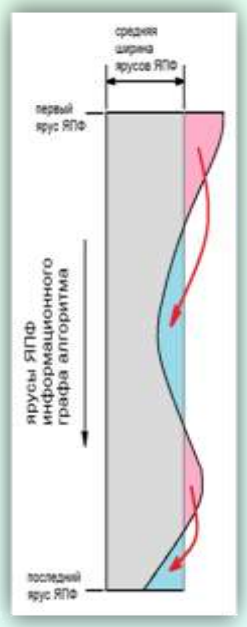
Выполнить скрипт на Lua (*акционные функции*), реализующий выбранную стратегию преобразования ЯПФ

Распозна́ть ситуацию и выбрать наиболее эффективную стратегию преобразования ЯПФ для этой ситуации

Примеры реализации стратегий *целенаправленного изменения* ЯПФ

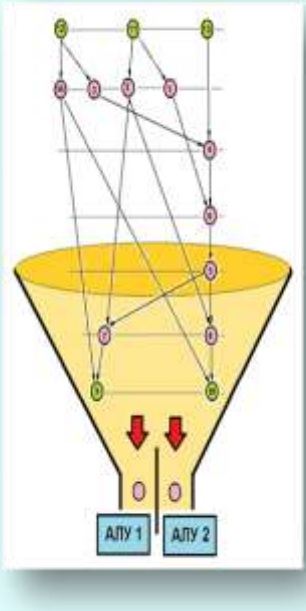
1. Задание – разработать план (расписание) параллельного выполнения программы с максимально равномерным распределением операторов по ярусам без увеличения высоты ЯПФ (оптимизация по критерию максимального использования ресурсов параллельного вычислителя при условии не увеличения времени выполнения программы).

Представим ЯПФ в вертикальном направлении как поверхность земли. Поверхность не гладкая – имеются возвышенности и впадины. Наша цель – сгла́дить поверхность (по возможности привести ширину всех ярусов к среднеарифметической величине). ***Метафора*** – отва́л бульдозера сгреба́ет землю с холмов во впадины…



2. Задание – разработать план (расписание) параллельного выполнения программы на заданном числе вычислителей при возможном возрастании высоты ЯПФ (оптимизация по использованию заданного количества вычислительных ресурсов при возможном увеличении времени выполнения программы).

Нам надо “сжать” ЯПФ по ширине, получив величину не более заданной. ***Метафора*** - предста́вим ЯПФ в виде “куска" продукта, поступающего в мясорубку (размер её выходного конуса – число параллельных вычислителей). На выходе – иско́мое… Вариант с раска́ткой теста на доске также годится!..



Общее описание алгоритма действий (*шаблон*) по целенаправленному преобразованию ЯПФ

# #		Действие	Примечание	Используемые LUA-вызовы
1		Вычислить среднее арифметическое ширин ярусов W_0 по всей ЯПФ	Требуется однократный цикл по ярусам ЯПФ (сначала получить общее число ярусов); результат W_0 округлить до целого сверху	GetCountTiers() + GetCountOpsOnTiers() для суммирования в цикле или вызов CalcParamsTiers(1)
2		Цикл по всем ярусам ЯПФ (сначала получить число операторов W_i на данном ярусе)	Цикл A (последний ярус рассматривать бессмысленно)	GetCountOpsOnTiers()
3		Если $W_i \geqslant W_0$, идти к следующему ярусу	Ширина текущего яруса меньше среднего арифметического – отсюда переносить операторы бессмысленно	
3		Цикл по всем операторам данного яруса	Цикл B (сначала получить число операторов на данном ярусе)	GetCountOpsOnTiers()
4		Получить *реальный* номер оператора *по номеру его положения* на текущем ярусе ЯПФ	Реальный номер оператора неизвестен, пока не укажешь его положение в ЯПФ	GetOpByNumbOnTier()
5		Для текущего оператора получить два числа (диапазон его допустимого положения по ярусам ЯПФ) T_{min} и T_{max}. При условии $T_{min}=T_{max}$ перейти к следующему оператору на данном ярусе	Если $T_{min}=T_{max}$, то *вариативность* (возможность оператора к перемещению по ярусам ЯПФ) положения оператора нулевая и перемещать его нельзя в принципе	GetMinTierMaybeOp(), GetMaxTierMaybeOp()
6		Цикл по ярусам ЯПФ, начиная с текущего+1 до последнего	Цикл C	
7		**Если найдётся ярус $T_{min} \leqslant T \leqslant T_{max}$ с числом операторов $W_i < W_0$, то это *формальный претендент* для переноса сюда рассматриваемого оператора (осуществить / не осуществлять собственно перенос)**	Исследователь самостоятельно определяет, осуществлять ли перенос по *формальным признакам* или использовать какие-то более сложные эвристические правила реконфигурации ЯПФ	MoveOpTierToTier(), GetOpsMoves(), PutTiersToTextFrame()
8		Конец циклов A,B,C	Конец всех циклов	

🛠 Слева приведена (*одна из…*) общих схем разработки Lua-сценариев для целенаправленного преобразования ЯПФ.

🔔 Согласно данной схеме Lua-программа содержит гнездо циклов глубиной 3 (показаны цветом в левой стороне таблицы), крайний справа столбец содержит подсказку в виде API-вызовов Lua, с помощью которых можно обеспечить требуемый функционал (см. руководство http://vbakanov.ru/spf@home/content/API_User.pdf).

➔ Полезно придерживаться общей тактики БУЛЬДОЗЕР, однако:

- при решении задачи с ***сохранением высоты ЯПФ*** границу между “холмами” и “впадинами” следует считать равной среднеарифметическому значению ширин ярусов по всей ЯПФ (естественно, округлённой до целого вверх),
- при постановке *задачи с **возможным возрастанием высоты ЯПФ*** – заданному значению ширины ЯПФ (т.е. числу параллельных вычислителей).

Максимально творческая часть при предложенном подходе – определение ***правил выбора*** переноси́мых с яруса на ярус операторов (из общего числа операторов на данном ярусе ЯПФ, являющихся потенциальными кандидатами для списка переносимых) – эта часть выделена красным цветом на схеме слева).

Конкретизация **функций цели** при оптимизации планов параллельного выполнения алгоритмов (программ)

1. Параметр: **ПЛОТНОСТЬ КОДА** плана параллельного выполнения алгоритма (программы).

🛠 Количественное выражение оптимизируемой величины: степень неравномерности распределения ширин отдельных ярусов всей ЯПФ (коэффициент вариации *CV*)

🛠 Формула для вычисления: $CV = \sqrt{\frac{\sum(W_i - \overline{W})^2}{H-1}} / \overline{W}$, где *CV* – коэффициент вариации, W_i – ширина *i*-того яруса ЯПФ (суммирование идёт по *i*), *H* – высота (число ярусов) ЯПФ, $\overline{W}$ – среднеарифметическое ширин всех ярусов данной ЯПФ

1. ***Цель*** - получение плана параллельного выполнения при ***максимально полном использовании ресурсов вычислительной системы*** (числа параллельных вычислителей)

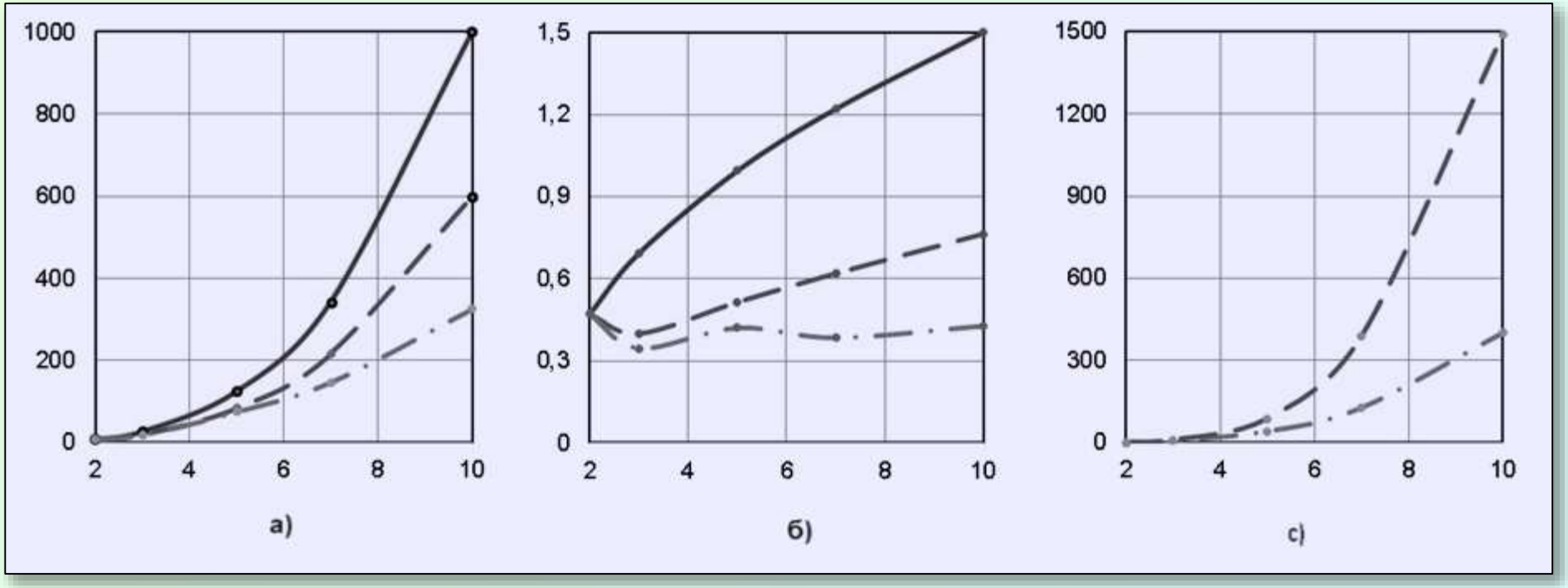
2. Параметр: **ВЫЧИСЛИТЕЛЬНАЯ СЛОЖНОСТЬ** получения плана параллельного выполнения алгоритма (программы)

🛠 Количественное выражение оптимизируемой величины: число ***элементарных действий***, выполненных при целенаправленной реорганизации ЯПФ при получении плана параллельного выполнения

🛠 За элементарное действие в данном случае предложено принимать ***перестановку оператора*** с яруса на другой ярус ЯПФ (аналогично перестановке элементов при сортировке одномерных массивов)

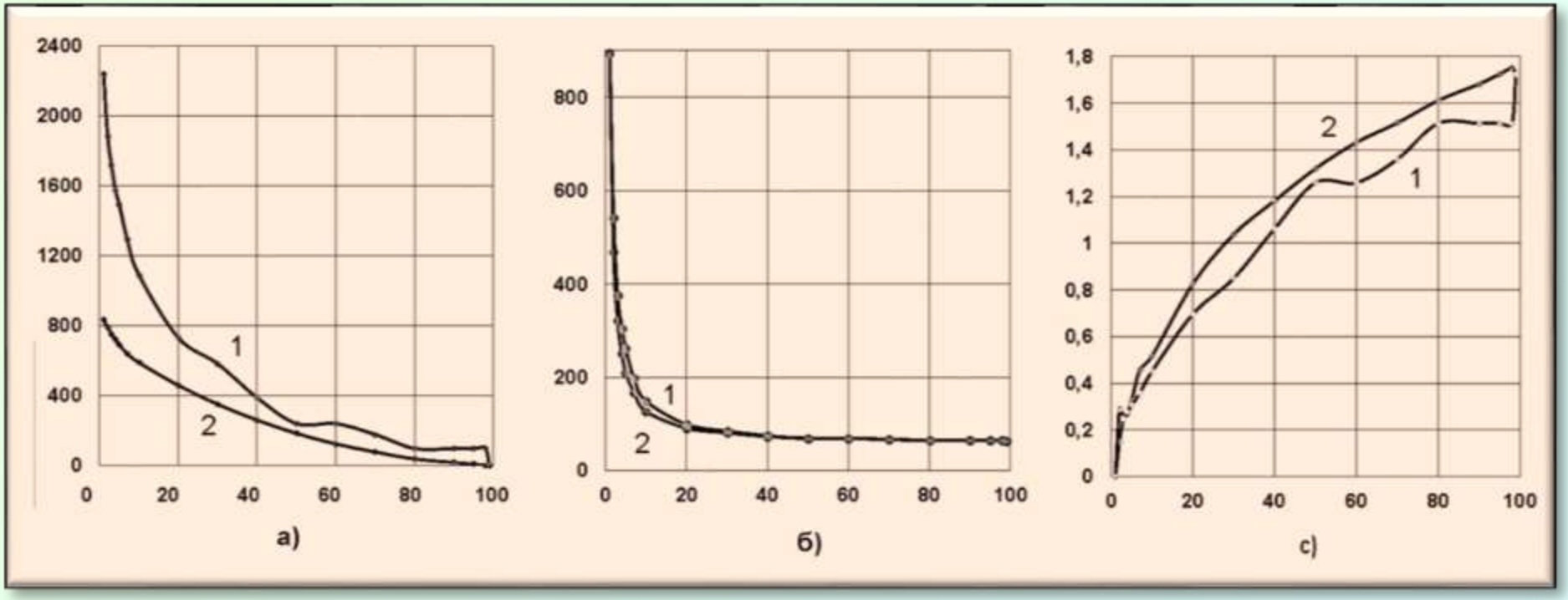
2. ***Цель*** **–** получить план параллельного выполнения (с заданными условиями-ограничениями) за ***минимальное число элементарных действий*** (фактически – за минимальное время)

Пример исследования: сравнение эффективности сценариев построения плана выполнения программы при минимальном времени её выполнения (условие невозрастания высоты исходной ЯПФ)



- Исследуемый алгоритм – умножение квадратных матриц порядков 2-10 (ось абсцисс) прямым (безитерационным) методом Гаусса
- На рисунках: а) – ширина ЯПФ, б) - коэффициент вариации (CV) ширин ярусов ЯПФ, с) - вычислительная сложность сценария (в единицах перемещения операторов между ярусами ЯПФ)
- Сплошные (красные) линии – исходная ЯПФ, пунктир (синяя) и штрих-пунктир (зелёная) – результат применения Lua-сценариев *01_bulldozer* и *02_bulldozer*

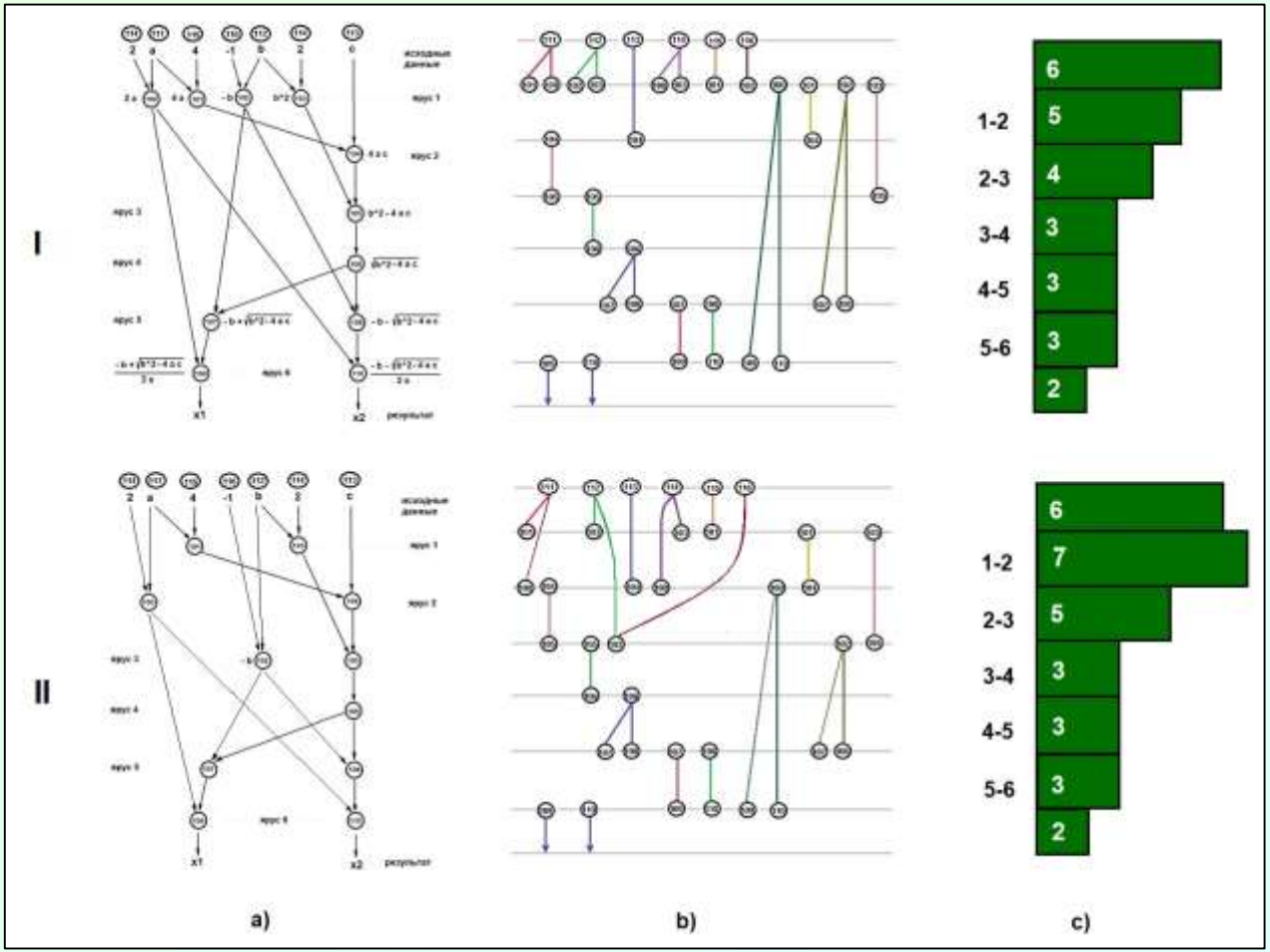
Пример исследования: сравнение эффективности сценариев построения плана выполнения программы на заданном числе параллельных вычислителей



- Исследуемый алгоритм - решение систем линейных алгебраических уравнений (СЛАУ) 10-го порядка прямым (безитерационным) методом Гаусса
- На рисунках: а) – вычислительная сложность сценария (в единицах перемещения операторов между ярусами ЯПФ), б) - высота ЯПФ, с) - коэффициент вариации (*CV*) ширин ярусов ЯПФ (*CV*) в функции заданного количества параллельных вычислителей
- Преобразования ЯПФ проводились согласно Lua-сценариям *01_Strategy* и *02_Strategy* (кривые 1 и 2 соответственно)

“Время жизни промежуточных данных” и проблема оптимизации использования регистров общего назначения (*РОН*) для хранения и передачи данных

В процессе работы алгоритма при срабатывании операторов создаются новые данные, которые затем используются иными операторами в качестве операндов. Эти данные надо где-то (обычно в регистрах процессора – РОН) временно хранить. РОН всегда не хватает и поэтому стоит з***адача оптимизации использования РОН для временного хранения данных***…



На рис. слева в ***столбце a)*** в верхнем ряду “сырая” ЯПФ в “верхнем” варианте, в нижнем ряду – *частично* модифицированная ЯПФ (изменено расположение операторов 100 и 102). ***Столбец в)*** – график времени “*жизни промежуточных данных*”, ***столбец с)*** – изменение количества этих данных по мере выполнения операторов на ярусах ЯПФ

В результате модификации ЯПФ ширина ея сократилась с 4 до 2 (коэффициент вариации *CV* снизился с 0,64 до 0,22), но количество временных данных увеличилось (см. светлые цифры на элементах линейчатой диаграммы в столбце с).

InterNet - ресурсы для проведения исследований с использованием рассматриваемого **ПРАКТИКУМА**:

- **Математическая (компьютерная) модель вычислителя пото́ковой архитектуры для исследования произвольных алгоритмов на наличие внутреннего (*скрытого*) параллелизма и параметров его практического использования (Data-Flow) :**
  - **Описание здесь: http://vbakanov.ru/dataflow/**
  - **Инсталляция здесь: http://vbakanov.ru/dataflow/content/install_df.exe**
- **Программная система для исследования и выбора рациональных методов построения планов (*расписаний*) выполнения программ на заданном поле параллельных вычислителей (SPF@home) :**
  - **Описание здесь: http://vbakanov.ru/spf@home/**
  - **Инсталляция здесь: http://vbakanov.ru/spf@home/content/install_spf.exe**
- **О вычислительном кластере кафедры КБ-5 МИРЭА :**
  - **http://vbakanov.ru/hist_clu/clusters.htm**
- **Тематические стать на Habr'е :**
  - **https://habr.com/ru/post/530078/**
  - **https://habr.com/ru/post/534722/**
  - **https://habr.com/ru/post/535926/**
  - **https://habr.com/ru/post/540122/**
  - **https://habr.com/ru/post/545498/**
  - **https://habr.com/ru/post/551688/**
  - **https://habr.com/ru/post/688196/**
- **"Стихотворные" произведения о ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЯХ (*"околонаучный стёб"*) :**
  - **Размышления о необходимости параллельных вычислений**
  - **О по́льзе пото́ковых (DATA-FLOW) вычислительных архитектур**

Приложение

История процессоров архитектуры VLIW (*Very Large Instruction Word*, процессоры со сверхдлинным машинным словом)

От суперкомпьютера **ЭЛЬБРУС-2** для военного применения в СССР через процессоры **Crusoe** и **Efficeon** (*Transmeta*, *США*) до **Itanium** (*Intel, Hewlett-Packard, США*) к микропроцессорам **ЭЛЬБРУС** (МЦСТ, Россия)

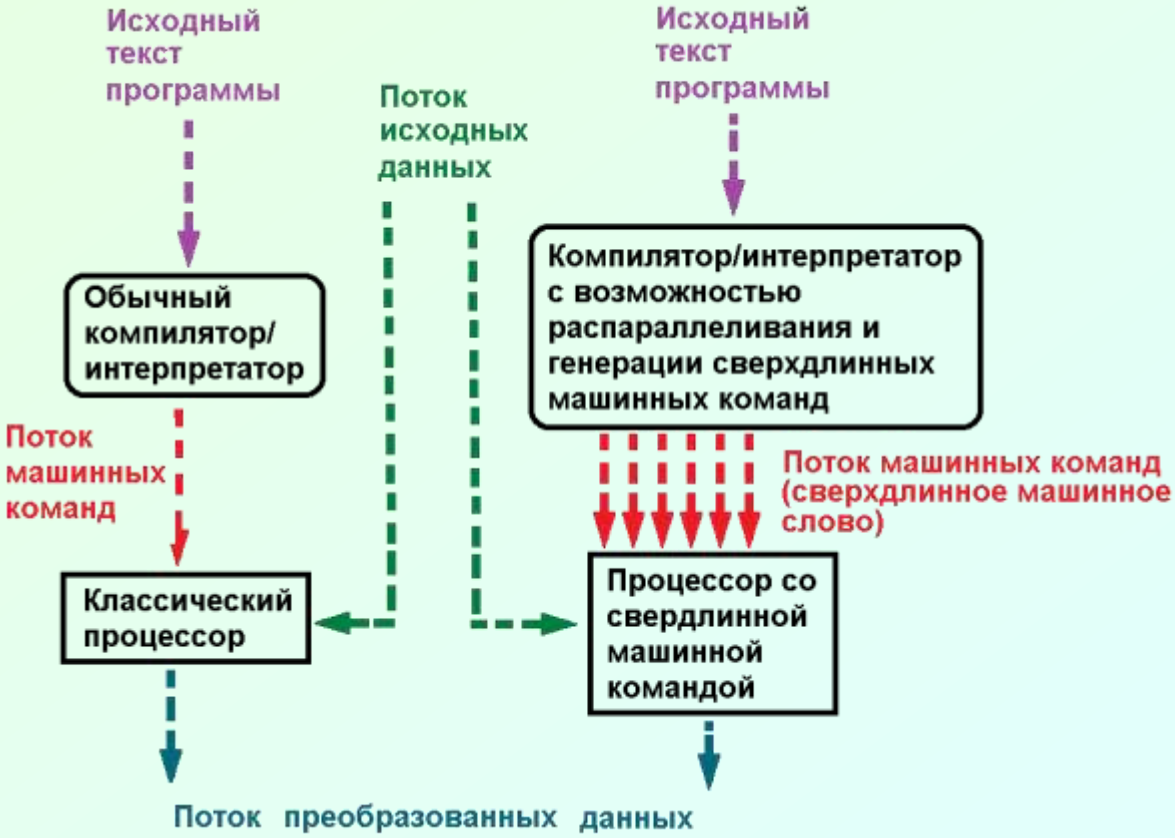
Пример параллельных вычислителей – отечественные VLIW-процессоры ЭЛЬБРУС

П.2

📖 Процессорная архитектура VLIW (*Very Large Instruction Word*), на которой основаны отечественные процессоры серии ЭЛЬБРУС (ИНЭУМ, МЦСТ, Россия) * показана схеме внизу справа ⬇ .

🛠 VLIW предполагает подачу на вход процессора не по одной команде последовательно (как в классической фон Неймановской архитектуре), сразу несколько машинных команд, упакованных в сверхдлинное машинное слово (*bandl*, связка). В составе связки находятся команды, не имеющие информационных зависимостей между собой (и поэтому мóгущие выполняться параллельно).

🖋 При такой схеме процессор получается простым (т.к. процессом распараллеливания он сам “не занимается” - значит, имеет малое тепловыделение и дёшев в изготовлении).

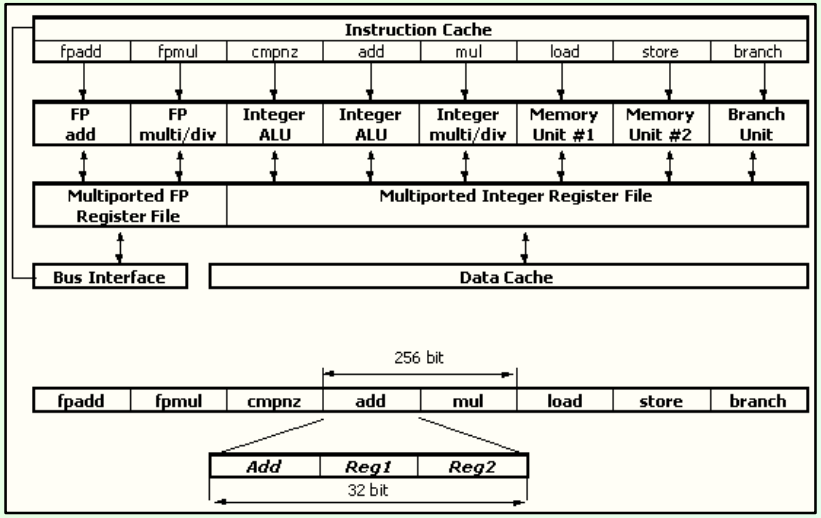


Выявление параллелизма в алгоритме и составление плана выполнения параллельной программы возлагается на “умный” (*smart*) компилятор (интерпретатор). Такая идеологема соответствует подходам EPIC (*Explicitly Parallel Instruction Computing*, набор инструкций с явным параллелизмом) и ILP (*Instruction-Level Parallelism*, параллелизм уровня машинных инструкций).

* ”Из этой же серии” процессоры ITANIUM (Intel Corp,, Hewlet Packard, США, 90-е годы), Crusoe, Efficeon (Transmeta Corp., США).

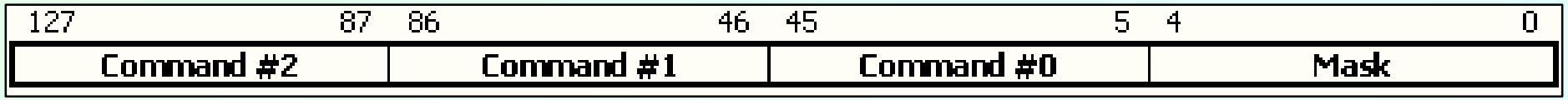
Классика VLIW – процессоры Itaniun (разработка фирм Intel + Hewlett-Packard)

П.3



←←← Изображение с ресурса https://www.ixbt.com/cpu/vliw.shtml. Общая схема VLIW-процессора с размером машинного слова в 8 процессорных команд. Каждая команда выполняет традиционную трёхоперандную RISC-подобную инструкцию типа *"код_операции регистр_приемника, регистр_ источни-ка"* и может непосредственно управлять специфическим функциональным блоком при минимальном *аппаратном* декодировании.

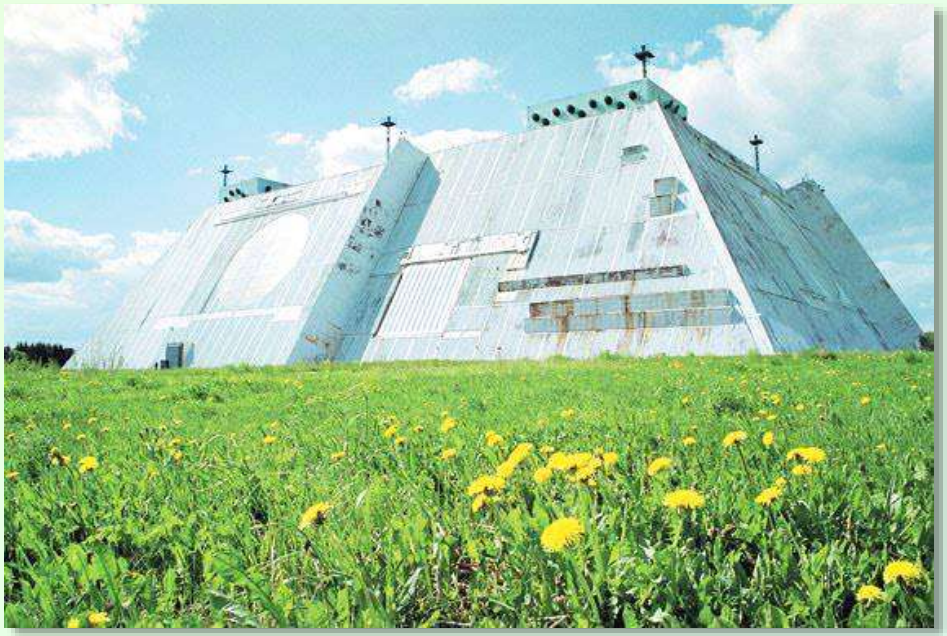
↓↓↓ На рис. ниже приведена схема сверхдлинного слова Intel-процессора IA-64 (серия *Itanium*). В данном случае связка (*bundle*) имеет длину 128 бит и включает в себя 3 поля (*слóты*) для команд длиной 41 бит каждое и 5-разрядный слот шаблона (*mask*). Предполагается, что команды связки могут выполняться параллельно разными функциональными устройствами. *В поле маски указываются возможные взаимозависимости, препятствующие параллельному выполнению команд одной связки.* Также маской задаются так называемые *останóвки*, определяющие слот, после начала выполнения команд которого инструкции последующих полей должны ждать завершения.



Р Проблемой VLIW-вычислителей является *качество планирования* каждой связки сверхдлинного слова, при этом связка часто получается неполной (недостаточна **плотность кода**, компилятор вставляет на пустые слоты команды NOP). Системный модуль SPF@home как раз и предназначен для получения рациональных планов выполнения параллельных программ (в частности, с максимальной ***плотностью кода***). **Работайте в этом направлении и "хватайте удачу за хвост"..!**

Процессоры архитектуры ЭЛЬБРУС *(история)*

П.4



🛠 Центральный элемент системы ПРО А-135 – радиолокационная станция кругового обзора “Дон-2Н” (Пушкинский район, 50 км к северу от Москвы, сдана в опытную эксплуатацию в 1989 г.). Первоначально управление станцией осуществляется вычислительным комплексом производительностью до миллиарда операций в секунду, построенном на основе четырёх суперкомпьютеров “Эльбрус-2”, генеральный конструктор - В.С.Бурцев (1927÷2005). “Эльбрус-2” эксплуатировался в ядерных исследовательских центрах в Челябинске-70 и в Арзамасе-16, в ЦУП’е (в комплексе с БЭСМ-6),

📖 За создании серии “Эльбрус’ов” многие разработчики были награждены орденами и медалями СССН – Б.А.Бабаян, В.В.Бардиж, В.С.Бурцев.

🗠 В работе над системой “Эльбрус-3” (первый VLIW-процессор в СССР) участвовал В.М.Пентковский (1946÷2012, автор языка ЭЛЬ-76, по легенде именно его имя было присвоено микропроцессору ПЕНТИУМ во время работы в Intel). “Эльбрус-3” не был запущен в серию, но его архитектура явилась основой создания современных микропроцессоров ЭЛЬБРУС. Чуть позднее в США начались работы по VLIW-архитектуре фирм Transmeta (процессоры Crusoe и Efficeon, оба используют технологию мóрфинга двоичного кода) и Intel (процессор Itanium).

Современные микропроцессоры ЭЛЬБРУС

П.5

Микропроцессоры ЭЛЬБРУС (E2K) на основе архитектуры ЭЛЬБРУС-3 (архитектура VLIW – *Very Large Instruction Word*) и готовые решения на их базе сегодня продвигает на рынке компания МЦСТ (см. также тут и тут). На сегодняшний день компьютеры компании МЦСТ в основном предназначены для - военных ведомств России, стран СНГ и БРИК; индустрии гражданского производства; РЛС гражданского назначения (наземного, морского и воздушного транспорта). Также возможно применение в сфере бизнеса и гражданских лиц, которым необходимы особо надежные и защищённые компьютеры. Компьютеры компании МЦСЕ обладают различным конструкторским исполнением, разным классом защиты в зависимости от требований. Все они обладают поддержкой или возможностью работы с GPS и ГЛОНАСС в зависимости от потребностей покупателя устройства.

По данным на 2021 г. микропроцессоры ЭЛЬБРУС (архитектура VLIW / EPIC) поддерживают выполнение до 23 инструкций за такт, имеют до 8 ядер, тактовую частоту 1’500 MHz, 3,5 млрд. транзисторов, выпускаются по технологии 28 нм, площадь кристалла 350 мм², производятся на заводе TSMC (*Taiwan Semiconductor Manufacturing Comp.*).