Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ

(часть 1)

Вл.В.Воеводин

Зав.кафедрой Суперкомпьютеров и квантовой информатики ВМК МГУ,
Директор НИВЦ МГУ,
чл.-корр. РАН, д.ф.-м.н., профессор

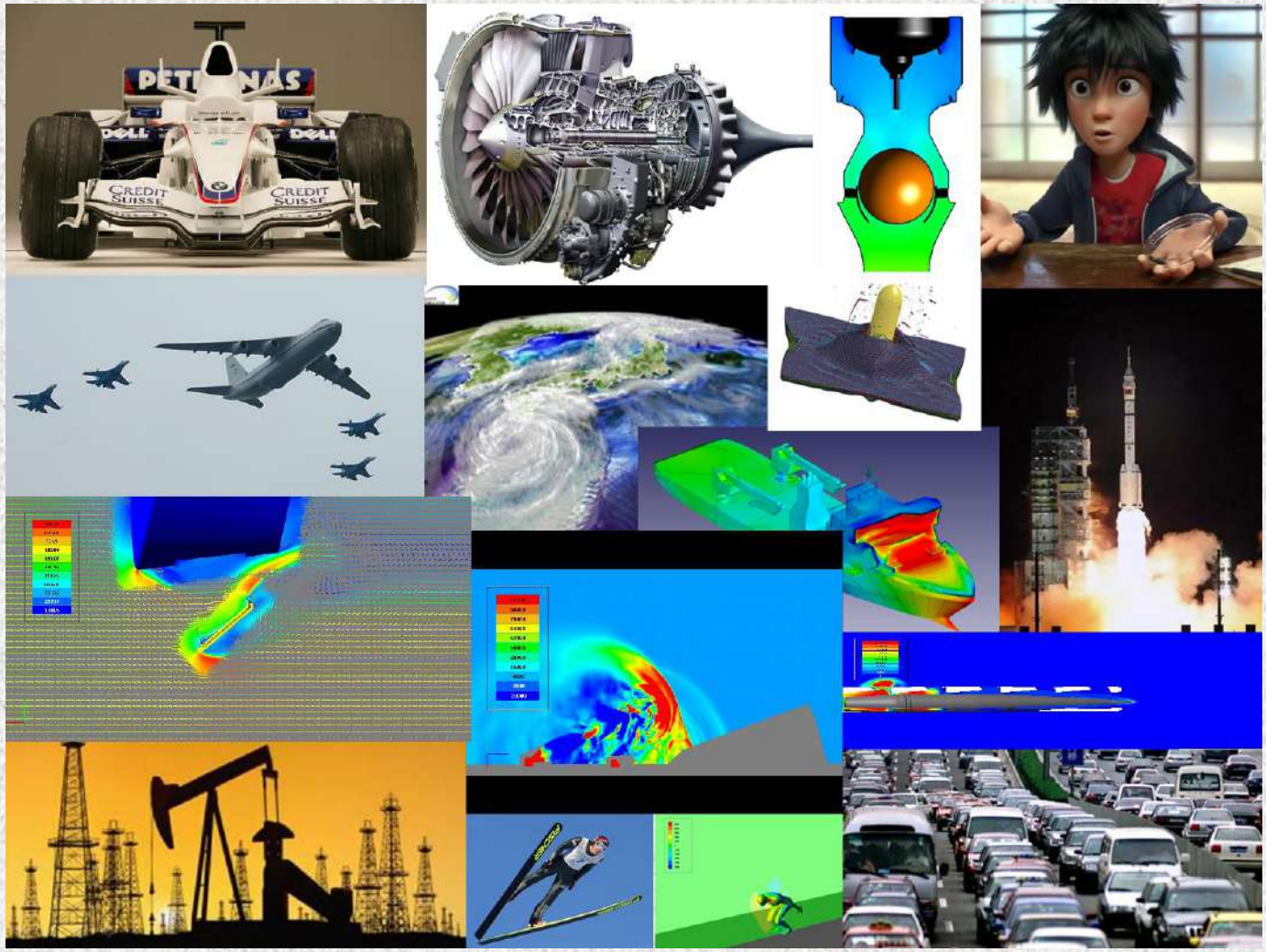
voevodin@parallel.ru

ВМК МГУ, 2020

Суперкомпьютеры – что это?

- *Суперкомпьютеры – это компьютеры, которые работают значительно быстрее остальной массы современных компьютеров;*
- *Seymour Cray: It's hard to define but you know when you see it;*
- *Суперкомпьютеры – это компьютеры, которые занимают большой зал;*
- *Суперкомпьютеры – это компьютеры, которые весят больше 1 тонны;*
- *Суперкомпьютеры – это компьютеры, которые стоят больше $1M ;*
- *Суперкомпьютеры – это компьютеры, которые сводят проблему вычислений к проблеме ввода/вывода;*
- *Суперкомпьютеры – это компьютеры, мощности которых немного не хватает для решения актуальных вычислительно сложных задач;*

Суперкомпьютерные технологии сегодня везде



Суперкомпьютерные технологии – это инструмент **обеспечения конкурентоспособности** науки, компаний, промышленности, экономики, страны.
Сделать быстрее, дешевле, качественнее, быстрее вывести на рынок…

Суперкомпьютерные технологии – ключевой элемент сквозных технологий цифровой экономики

Сквозные технологии:

- *математическое моделирование,*
- *искусственный интеллект,*
- *новые производственные технологии,*
- *хранение и анализ больших данных,*
- *технологии виртуальной и дополненной реальности,*
- *квантовые технологии,*
- *Интернет вещей,*
- *кибербезопасность,*
- *…*

Суперкомпьютерные технологии и большие данные

Лидеры рейтинга **Fortune500 Global** в 2019 году:

№	Компания	Страна	Отрасль	Выручка
1	Walmart		Розничная торговля	$514 млрд
2	Sinopec		Добыча и переработка нефти и газа	$415 млрд
3	Royal Dutch Shell		Добыча и переработка нефти и газа	$397 млрд
4	China National Petroleum		Добыча и переработка нефти и газа	$393 млрд
5	State Grid		Электроэнергетика	$387 млрд
6	Saudi Aramco		Добыча и переработка нефти и газа	$356 млрд
7	BP		Добыча и переработка нефти и газа	$304 млрд
8	Exxon Mobil		Добыча и переработка нефти и газа	$290 млрд
9	Volkswagen		Производство автомобилей	$278 млрд
10	Toyota Motor		Производство автомобилей	$273 млрд

Все эти компании в своей работе опираются на использование технологий больших данных и высокопроизводительные вычислительные системы.

Walmart, торговля:

- годовой оборот более 500 млрд.долл., прибыль около 10 млрд.долл.,
- 20 тыс. магазинов в 28 странах мира,
- темп поступления и обработки данных: 2.5 ПБайт/час (10^{15} Байт/час),
- принятие оперативных решений на основе совокупной обработки данных из 200 источников, включая: транзакционные данные, метеорологические данные, экономические данные, телекоммуникационные данные, данные социальных сетей, цены на бензин, данные о событиях в населенных пунктах, где находятся магазины и многие другие.

Суперкомпьютерные технологии и квантовые технологии

Связь суперкомпьютерных и квантовых технологий естественна, но особенно сильно прослеживается по двум направлениям:

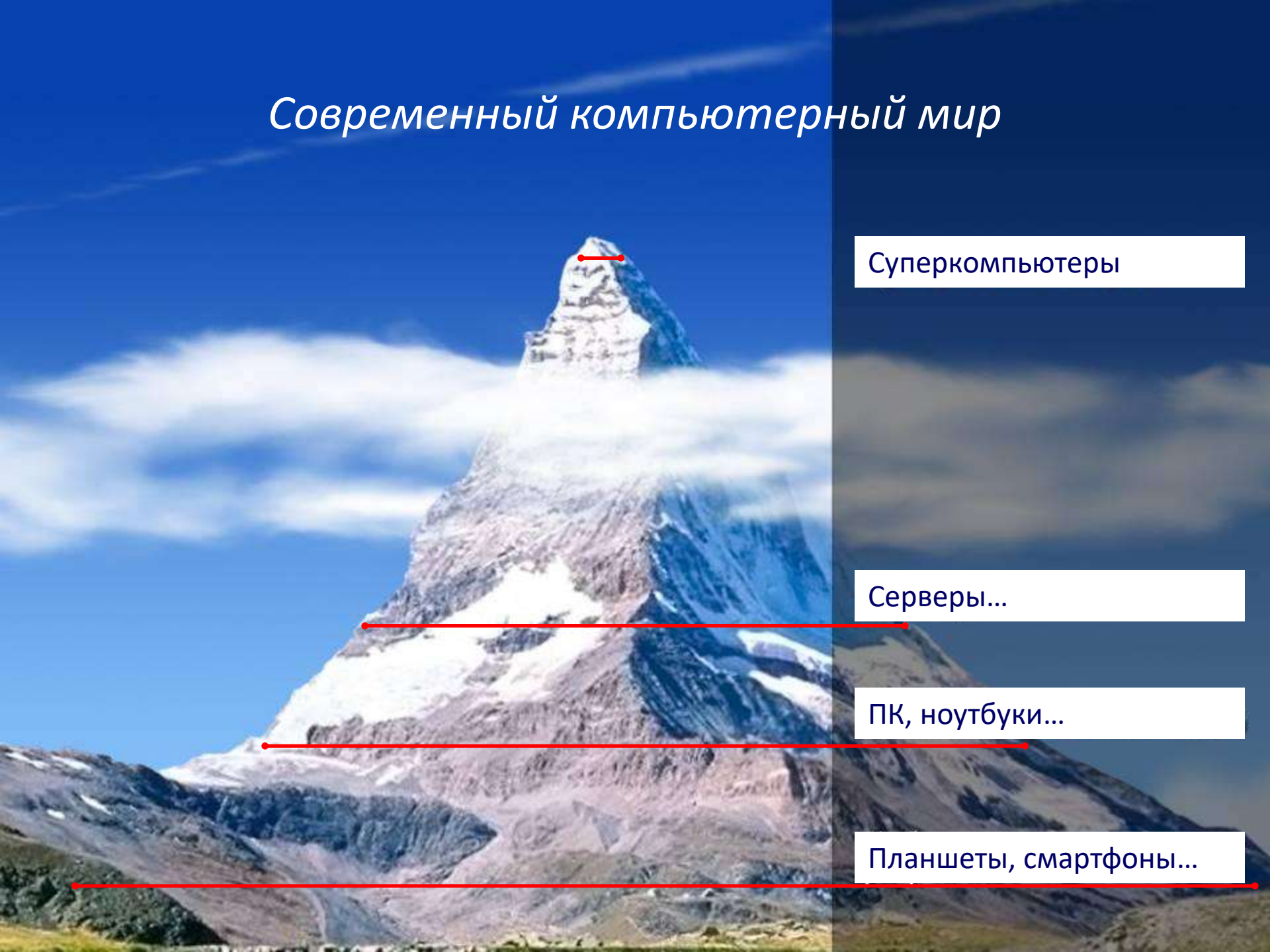
- Моделирование квантовых алгоритмов:
  - Понимание границы “квантового превосходства”, проверка новых схем коррекции ошибок, анализ влияния шумов.
  - Моделирование ведется с использованием лучших суперкомпьютеров мира (Cori - №14 в Top500, Edison Supercomputer, IBM BlueGene/Q) и собственных высокопроизводительных вычислительных систем компаний.
  - Активные работы ведут: Google, Alibaba, Huawei…

- Выбор параметров и управление квантовыми компьютерами:
  - успешные квантовые вычисления возможны только с надежной коррекцией ошибок, что увеличивает число необходимых физических кубитов на много порядков, что, в свою очередь, требует эффективного массивного управления (алгоритм Шора: для успешной работы необходимы 10^4-10^5 физических кубитов на каждый логический кубит);
  - для анализа работы реального физического квантового преобразования (гейта) необходимо проводить квантовую томографию (набирать статистику), однако сложность как по памяти, так и по скорости растет экспоненциально.

Суперкомпьютеры – что это?

… мобильные устройства –
персональные компьютеры – серверы –
суперкомпьютеры

Современный компьютерный мир



Персональный компьютер, 2020 г. – какой он?

Производительность: около 100 млрд. оп/с.

Память – 8-64 Гбайт

Диски – 500 Гбайт

Важные сокращения

Мега (Mega)	$– 10^6$	(миллион)
Гига (Giga)	$– 10^9$	(биллион / миллиард)
Тера (Tera)	$– 10^{12}$	(триллион)
Пета (Peta)	$– 10^{15}$	(квадриллион)

Флоп/с, Flop/s – Floating point operations per second

15 Tflop/s = $15 * 10^{12}$ арифметических операций в секунду над вещественными числами, представленными в форме с плавающей точкой.

Важные сокращения

Мега (Mega)	$– 10^6$	(миллион)
Гига (Giga)	$– 10^9$	(биллион / миллиард)
Тера (Tera)	$– 10^{12}$	(триллион)
Пета (Peta)	$– 10^{15}$	(квадриллион)

Флоп/с, Flop/s – Floating point operations per second

15 Tflop/s = $15 * 10^{12}$ арифметических операций в секунду над вещественными числами, представленными в форме с плавающей точкой.

Персональный компьютер, 2020 г. – какой он?

Производительность: около 100 Гфлоп/с.

Память – 8-64 Гбайт

Диски – 500 Гбайт

Важные сокращения

(взгляд в недалекое будущее)

Мега (Mega)	*– 10^6*	*(миллион)*
Гига (Giga)	*– 10^9*	*(биллион / миллиард)*
Тера (Tera)	*– 10^{12}*	*(триллион)*
Пета (Peta)	*– 10^{15}*	*(квадриллион)*
Экза (Exa)	*– 10^{18}*	*(квинтиллион)*

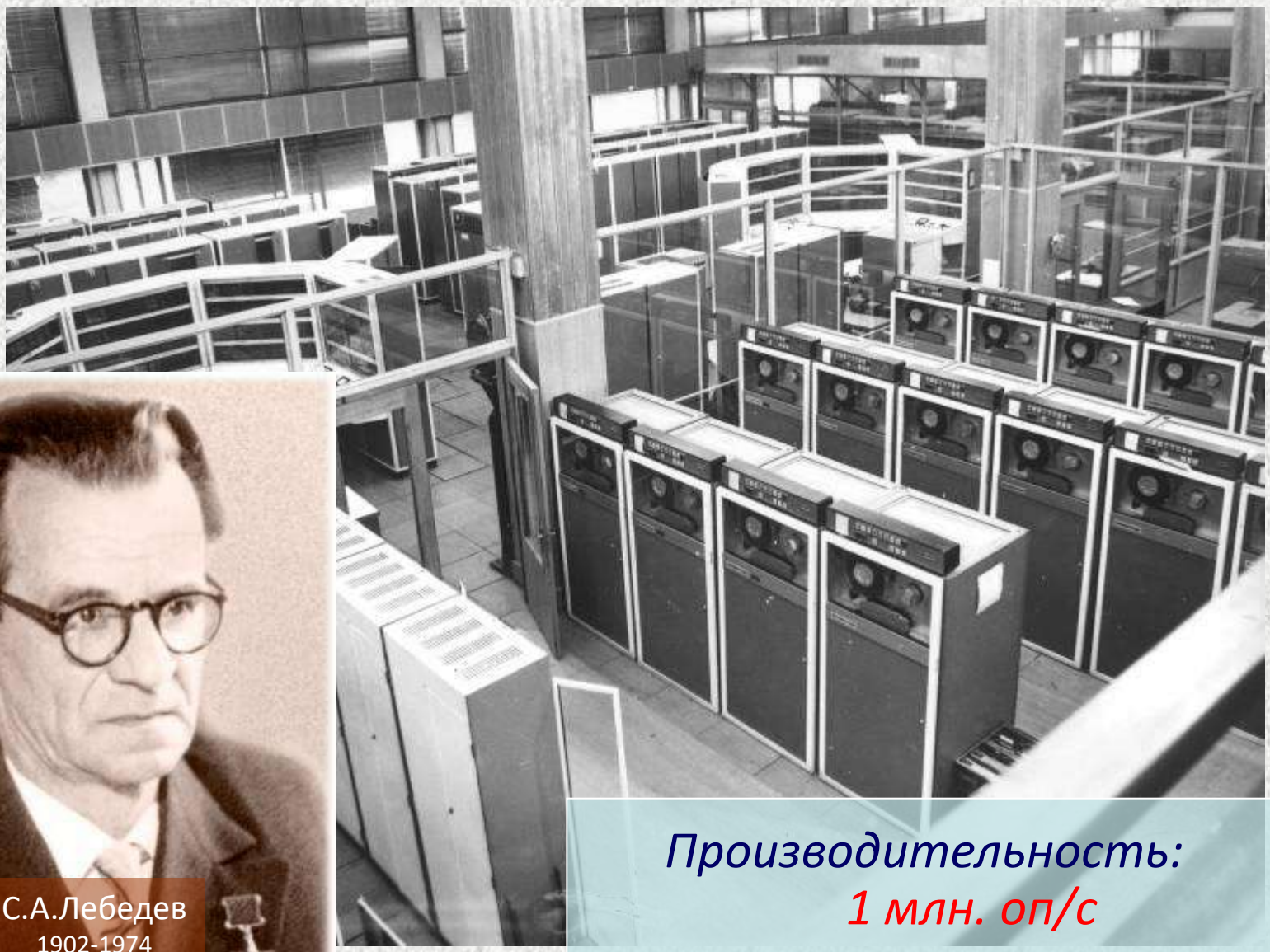
1 000 000 000 000 000 000 – ‘Экза’

Большие машины и суперкомпьютеры: какие они?

Вычислительный центр МГУ, 1956 г.

Вычислительный центр МГУ, 1959 г.

НИВЦ МГУ, ЭВМ “БЭСМ-6”, 1968 г.



С.А.Лебедев
1902-1974

Производительность:
1 млн. оп/с

Суперкомпьютер IBM “RoadRunner”, США

(#1 Top500 в 2008-2009 г.)

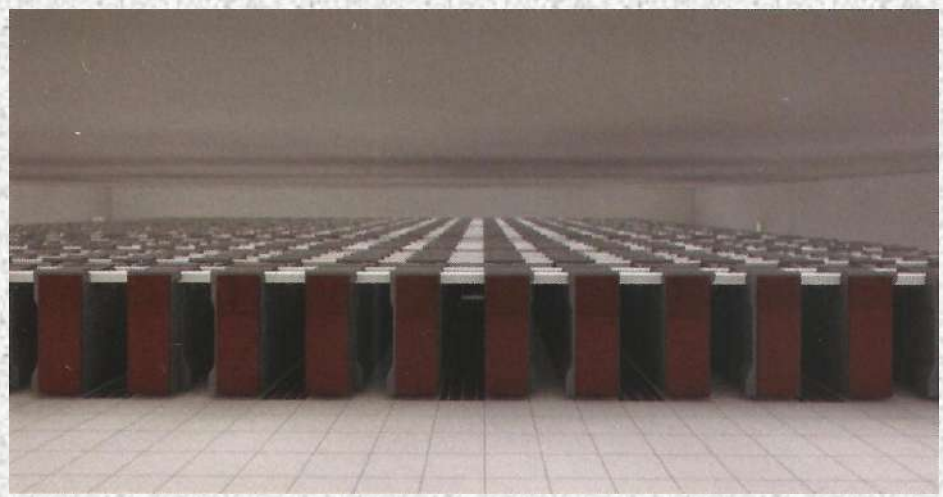
122 400 процессоров IBM Cell + AMD Opteron.

Производительность: 1 Pflop/s.

“K Computer”, Япония

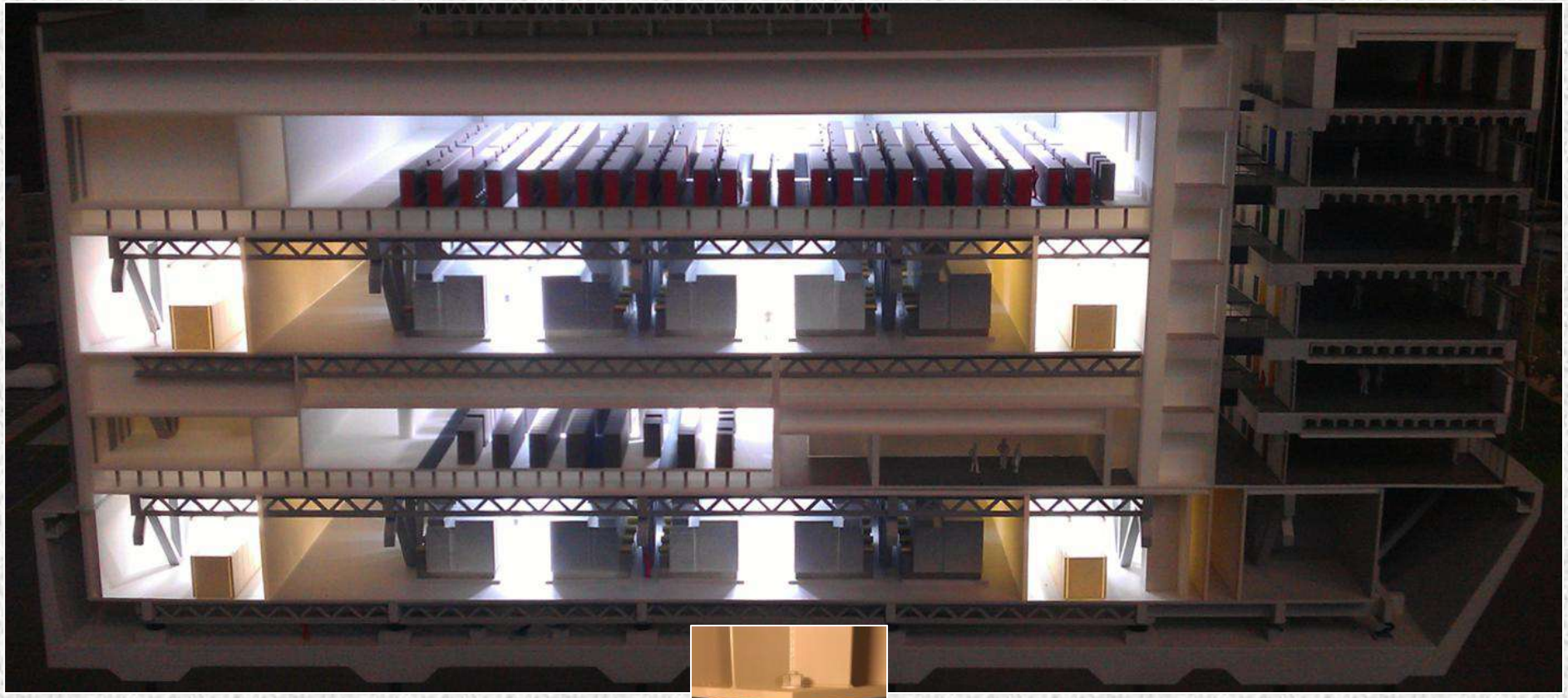
(#1 Top500 в 2011 г.)

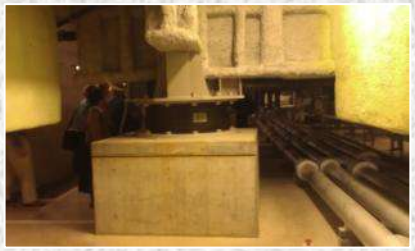
Производитель: Fujitsu, Япония
Пик (теория): 11.3 Pflop/s
Тест Linpack: 10.5 Pflop/s
Эффективность: 93%
Число стоек: 800+
Процессор: SPARC64 VIIIfx, 2.0 GHz
Число процессорных ядер: 705 024
Связь между процессорами: Tofu, 6D mesh/torus
Энергопотребление: 12.6 МВт

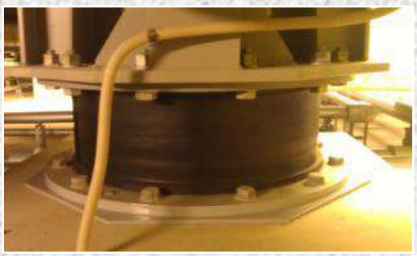


“K Computer”, Япония

(#1 Top500 в 2011 г.)







Суперкомпьютер “Sequoia”, IBM BlueGene/Q, США

(#1 Top500 в 2012 г., июнь)

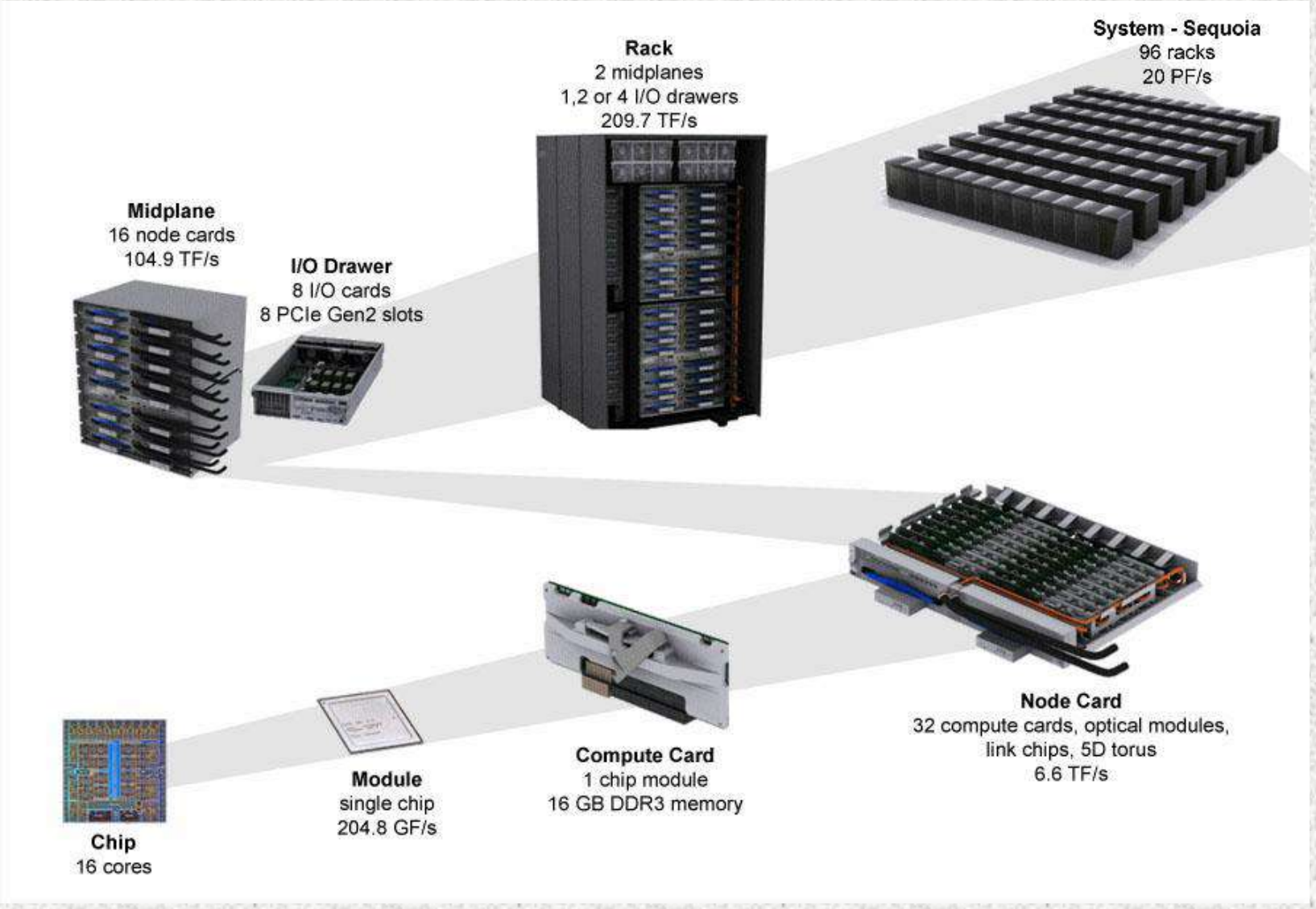
1 572 864 ядра,
IBM PowerPC A2 (64-bit),
1.6 GHz.

Связь между процессорами:
5-мерный тор

Производительность:
16.32 Pflop/s

Иерархичность архитектуры и параллелизма суперкомпьютеров

(IBM BlueGene/Q, Sequoia, 2012 г.)



Суперкомпьютер “Titan”, Cray XK7, США

(#1 Top500 в 2012 г., ноябрь)

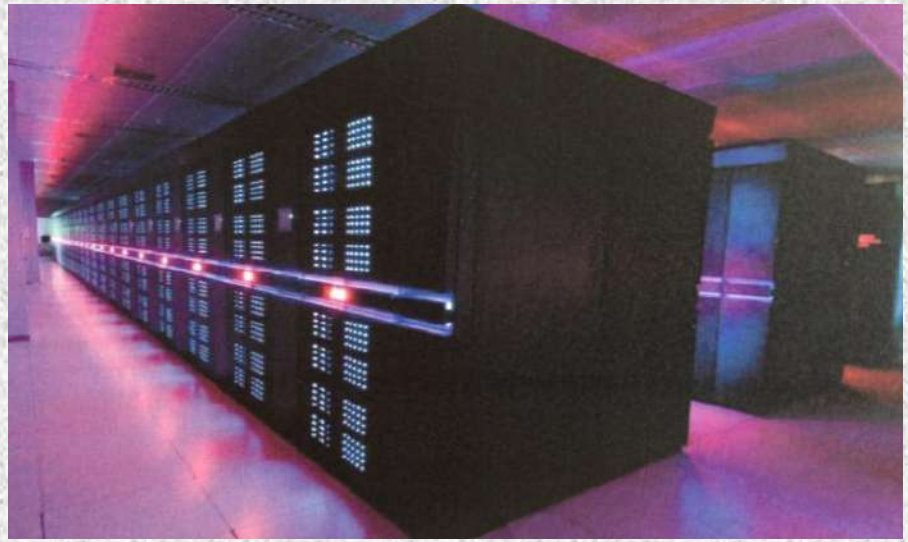
18 688 вычислительных узлов:

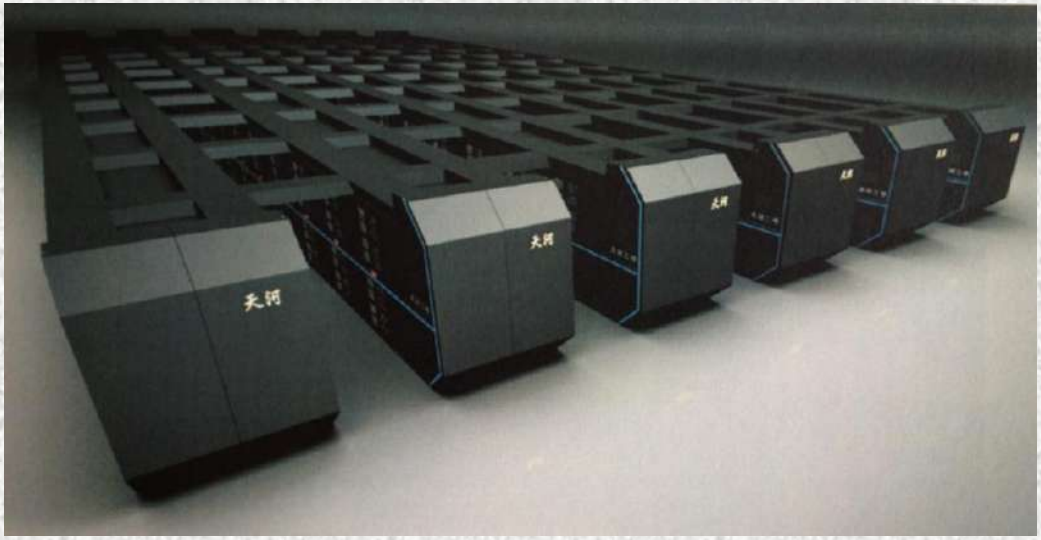
18 688 AMD Opteron, 16-core (299 008 ядер),
18 688 NVIDIA Tesla K20.

Производительность:
17.59 Pflop/s

Суперкомпьютер “Tianhe-2”, Китай

(#1 Top500 в 2013-2015 г.)





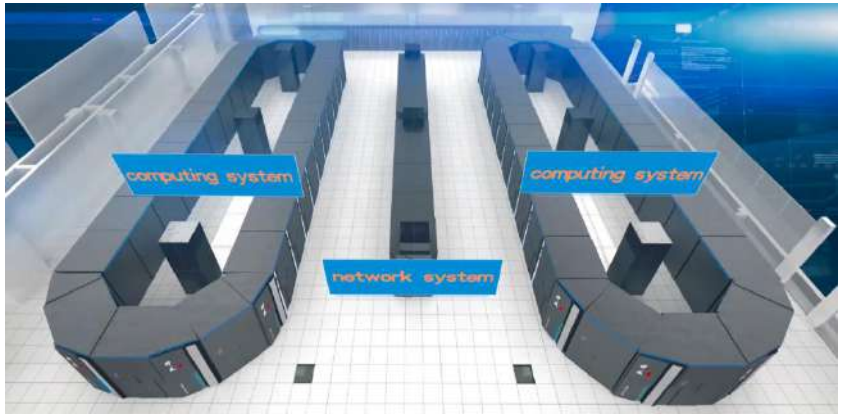
16 000 вычислительных узлов

32 000 Intel Xeon IvyBridge, 12-core
48 000 Intel Xeon Phi (Matrix-2000)

Всего: 3 120 000 ядер

Производительность:
Peak (теория): 54.9 Pflop/s
Тест Linpack: 33.86 Pflop/s
Эффективность: 62%

Суперкомпьютер Sunway TaihuLight, Китай
(#1 Top500 в 2016-2017 г.)



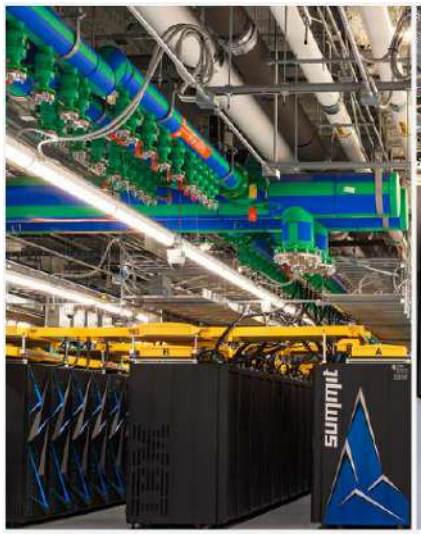
40 960 вычислительных узлов
40 960 CPUs (SW26010, 260 ядер)

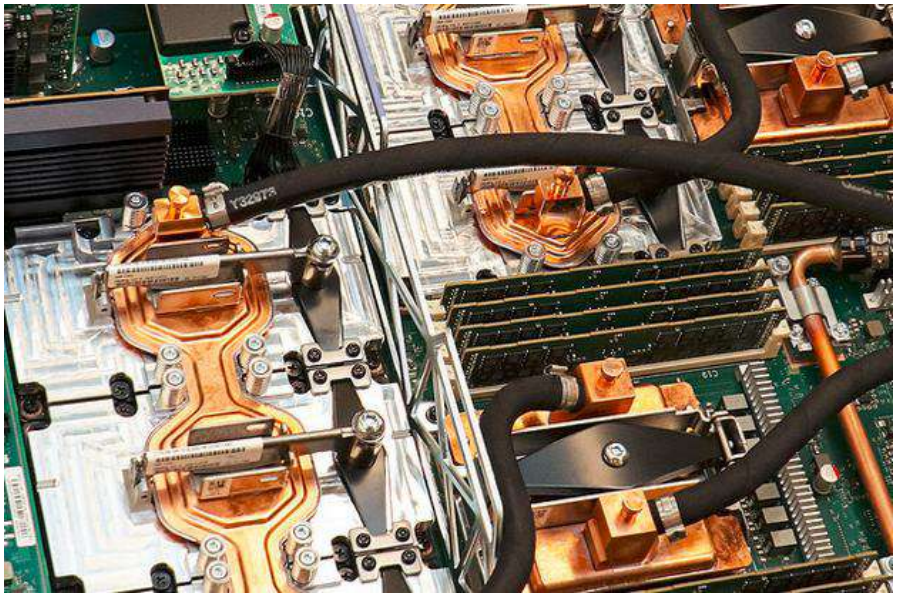
Всего: 10 649 600 процессорных ядер

Производительность:
Пик (теория): 125.4 Pflop/s
Тест Linpack: 93 Pflop/s (74%)

Оперативная память = 1.31 Пбайт
HDD = 20 ПБайт

Суперкомпьютер IBM Summit, США
(#1 Top500 в 2018-2019 г.)





4 608 вычислительных узлов,
в каждом узле:
2 x CPUs (IBM Power9, 22 ядра)
6 x GPU (NVIDIA Tesla V100)

Производительность:
Пик (теория): 200.8 Pflop/s
Тест Linpack: 148.6 Pflop/s (74%)

Оперативная память = 10 Пбайт
HDD = 250 ПБайт

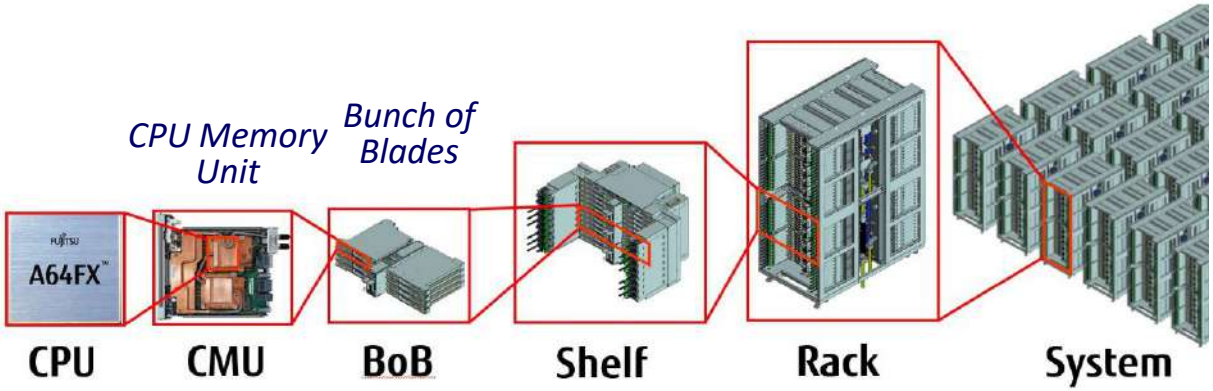
Суперкомпьютер Fujitsu Fugaku, Япония
(#1 Top500 в 2020 г.)

158 976 выч.узлов, 432 стойки

CPU: A64FX ARM v8.2-A, 48/52 cores
Векторные инструкции (SVE),
аппаратная поддержка барьеров…
Int: 1,2,4,8 байт; Float: 16,32,64 бит

Производительность:
Пик (теория): 513 Pflop/s
Тест Linpack: 415 Pflop/s (81%)
Оперативная память = 4.85 Пбайт
Энергопотребление = 28 МВт
Площадь = 1920 м^2

Суперкомпьютер Fujitsu Fugaku, Япония
(#1 Top500 в 2020 г.)



Иерархия в архитектуре суперкомпьютеров

Суперкомпьютеры и их характеристики

(производительность, память, диски, ленты)

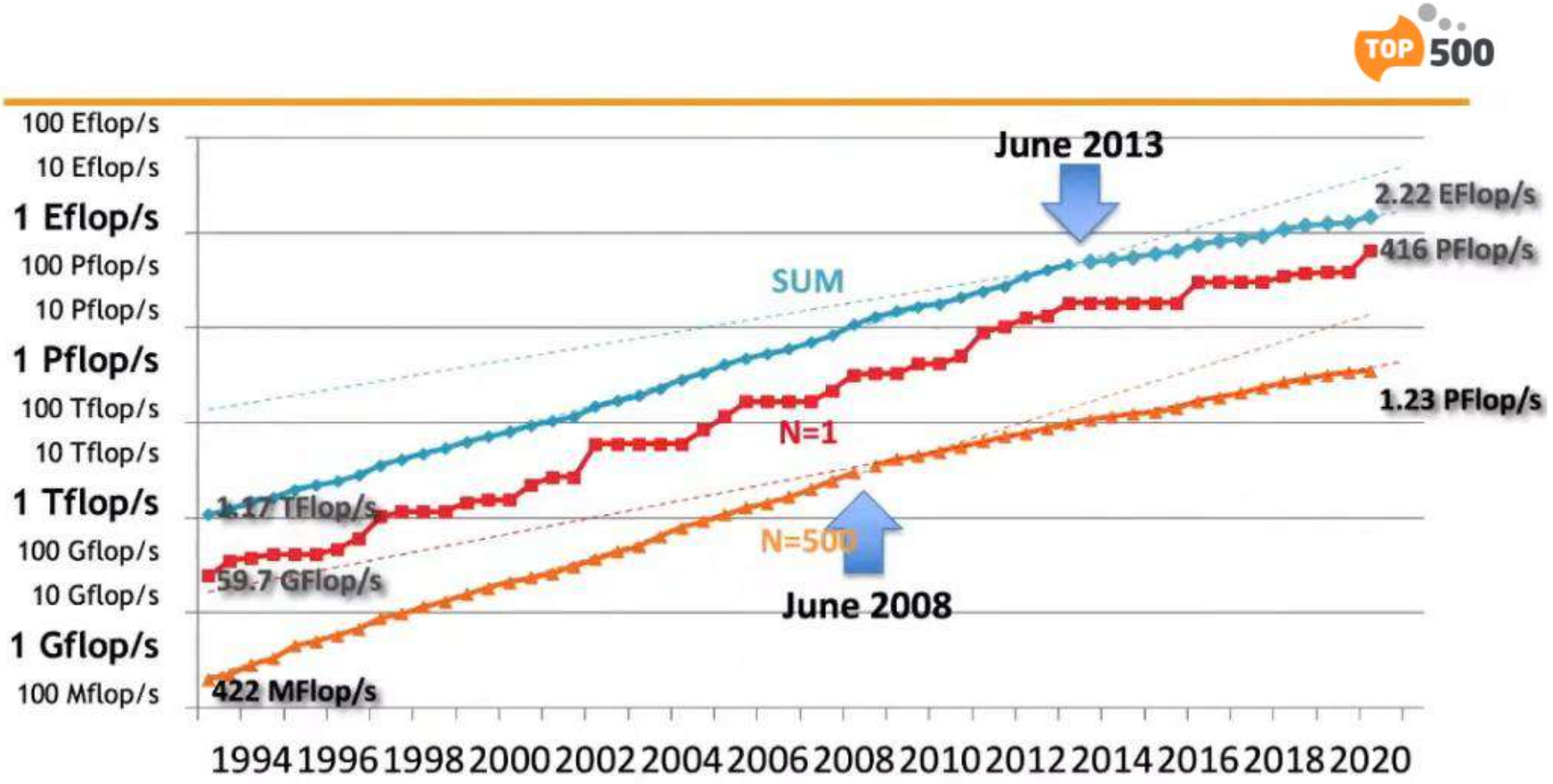
- *IBM Summit, 4608 вычислительных узлов:*
  2 × IBM Power9 (22 ядра), 4 GHz + 6 × NVIDIA Tesla V100
  148.6 Pflop/s, ОП = 10 PB, HDD = 250 PB
- *Sunway TaihuLight, 40960 вычислительных узлов:*
  SW26010 (260 cores), 1.45 GHz
  93 Pflop/s, ОП = 1.3 PB, HDD = 20 PB
- *Tianhe-2, 16 000 вычислительных узлов:*
  2 × Intel XeonE5-2692 (12cores), 2.93GHz + 3 × Matrix-2000,
  61.4 Pflop/s, ОП = 2.3 PB, HDD = 12 PB
- *Cray XK7, Titan, 18 688 вычислительных узлов:*
  AMD Opteron 16-core + NVIDIA Tesla K20
  17.59 Pflop/s, ОП = 710 TB, HDD = 10 PB
- *IBM BlueGene/Q, Sequoia, 1 572 864 ядра, IBM PowerPC A2, 1.6 GHz*
  16.32 Pflop/s, ОП = 1.57 PB, HDD = 55 PB, Tape = “virtually unlimited”

Top500 самых мощных суперкомпьютеров мира
(http://top500.org, июнь, 2020 г.)

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,299,072	415,530.0	513,854.7	28,335
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
5	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
6	**HPC5** - PowerEdge C4140, Xeon Gold 6252 24C 2.1GHz, NVIDIA Tesla V100, Mellanox HDR Infiniband, Dell EMC Eni S.p.A.	669,760	35,450.0	51,720.8	2,252

Рост производительности суперкомпьютеров

(закономерности отрасли)

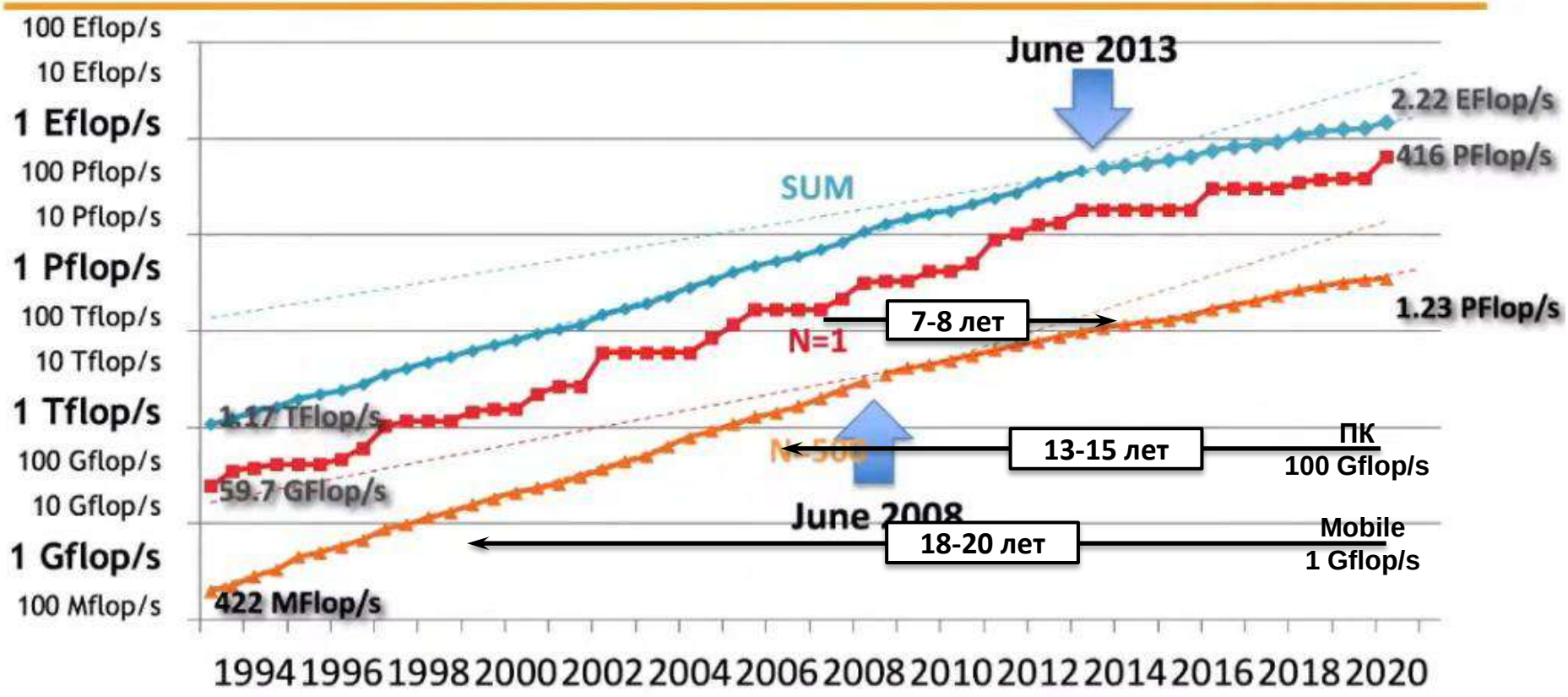


TOP500 SUPERCOMPUTER SITES

Courtesy of E.Strohmaier, LBNL, USA

Рост производительности суперкомпьютеров

(закономерности отрасли)



Top500 самых мощных суперкомпьютеров мира
(http://top500.org, июнь, 2020 г.)

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,299,072	415,530.0	513,854.7	28,335
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
5	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
6	**HPC5** - PowerEdge C4140, Xeon Gold 6252 24C 2.1GHz, NVIDIA Tesla V100, Mellanox HDR Infiniband, Dell EMC Eni S.p.A.	669,760	35,450.0	51,720.8	2,252

Gflops/Watt

Лидеры по энергоэффективности

(http://top500.org, Green500 list, июнь 2020)

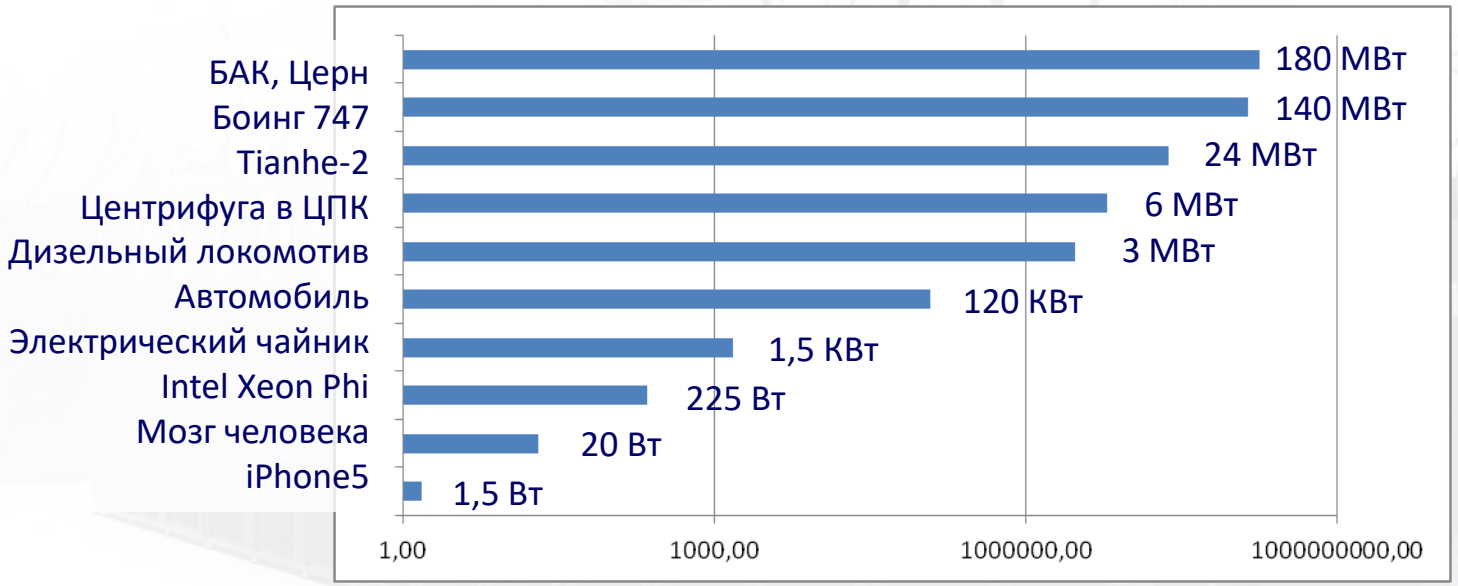
The GREEN 500

Rank	TOP500 Rank	System	Cores	Rmax (TFlop/s)	Power (kW)	Power Efficiency (GFlops/watts)
1	393	**MN-3** - MN-Core Server, Xeon 8260M 24C 2.4GHz, MN-Core, RoCEv2/MN-Core DirectConnect, Preferred Networks Preferred Networks Japan	2,080	1,621.1	77	21.108
2	7	**Selene** - DGX A100 SuperPOD, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	272,800	27,580.0	1,344	20.518
3	468	**NA-1** - ZettaScaler-2.2, Xeon D-1571 16C 1.3GHz, Infiniband EDR, PEZY-SC2 700Mhz, PEZY Computing / Exascaler Inc. PEZY Computing K.K. Japan	1,271,040	1,303.2	80	18.433
4	204	**A64FX prototype** - Fujitsu A64FX, Fujitsu A64FX 48C 2GHz, Tofu interconnect D, Fujitsu Fujitsu Numazu Plant Japan	36,864	1,999.5	118	16.876
5	26	**AiMOS** - IBM Power System AC922, IBM POWER9 20C 3.45GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM	130,000	8,339.0	512	16.285

TOP500 SUPERCOMPUTER SITES

[Gflops/Watt]

Энергопотребление вокруг нас…



Автомобиль - БАК (Большой Адронный Коллайдер) - Боинг 747 - Дизельный локомотив
Мозг человека - Центрифуга в ЦПК - Электрический чайник
Intel Xeon Phi - iPhone5 - Tianhe-2

Top50 самых мощных компьютеров СНГ

(top50.supercomputers.org)

Top50 – это совместный проект НИВЦ МГУ и МСЦ РАН

Top50 самых мощных суперкомпьютеров СНГ

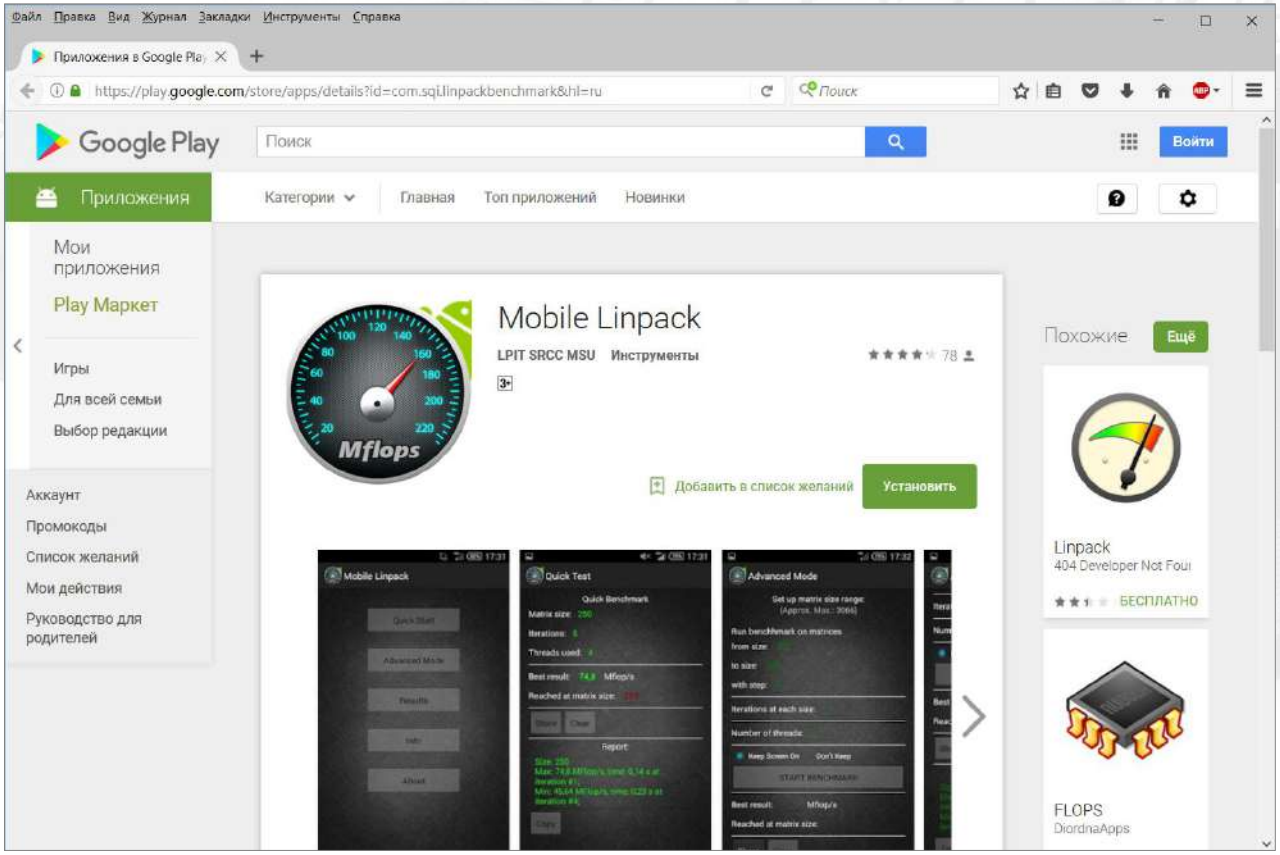
(top50.supercomputers.ru)

№	Название Место установки	Узлов Проц. Ускор.	Архитектура: кол-во узлов: конфигурация узла сеть: вычислительная / сервисная / транспортная	Rmax Rpeak (Тфлоп/с)	Разработчик *Область применения*
1 new	«Кристофари» SberCloud (ООО «Облачные технологии») , СберБанк, Москва	75 150 1200	75: NVIDIA DGX-2 CPU: 2x Intel Xeon Platinum 8168 24C 2.7GHz, 1536 GB RAM Acc: 16x NVIDIA Tesla V100 EDR Infiniband / 100 Gigabit Ethernet / 10 Gigabit Ethernet	6669.0 8789.76	**SberCloud (ООО «Облачные технологии») NVIDIA** *Облачный провайдер*
2 ▽	«Ломоносов-2» Московский государственный университет имени М.В.Ломоносова, Москва	1696 1696 1856	1536: CPU: 1x Intel Xeon E5-2697v3, 64 GB RAM Acc: 1x NVIDIA Tesla K40M 160: CPU: 1x Intel Xeon Gold 6126, 96 GB RAM Acc: 2x NVIDIA Tesla P100 FDR Infiniband / Gigabit Ethernet / FDR Infiniband	2478.0 4946.79	**Т-Платформы** *Наука и образование*
3 ▽	ФГБУ 'ГВЦ Росгидромета', Москва	976 1952 н/д	976: CPU: 2x Intel Xeon E5-2697v4, 128 GB RAM Aries / Aries + Gigabit Ethernet / Aries + Infiniband	1200.35 1293.0	**Т-Платформы** **Cray** *Исследования*
4 ▽	«Ломоносов» Московский государственный университет имени М.В.Ломоносова, Москва	6199 12406 2130	4160: CPU: 2x Intel Xeon X5570, 12 GB RAM 260: CPU: 2x Intel Xeon X5570, 24 GB RAM 640: CPU: 2x Intel Xeon X5670, 24 GB RAM 40: CPU: 2x Intel Xeon X5670, 48 GB RAM 30: CPU: 2x IBM PowerXCell 8i, 16 GB RAM 777: CPU: 2x Intel Xeon E5630, 12 GB RAM Acc: 2x NVIDIA Tesla X2070	901.9 1700.21	**Т-Платформы** *Исследования*

Сравните скорость своего мобильного устройства с суперкомпьютерами прошлого и настоящего

(http://linpack.hpc.msu.ru/)

Тест Linpack для мобильных устройств: Android 1.6 и выше, iOS 6.0 и выше.



Адрес для отзывов: ml@parallel.ru

Первый вычислительный кластер МГУ, 1999 г.

(12 узлов, 24 CPUs, Intel P-III/500MHz, сеть SCI)

Производительность:

12 Гфлоп/с

Суперкомпьютер МГУ “Ломоносов”

Суперкомпьютер МГУ “Ломоносов”

Суперкомпьютер МГУ “Ломоносов”

Всего в системе 10 т гликоля и 40 т воды

Суперкомпьютер МГУ “Ломоносов”

Суперкомпьютер МГУ “Ломоносов”

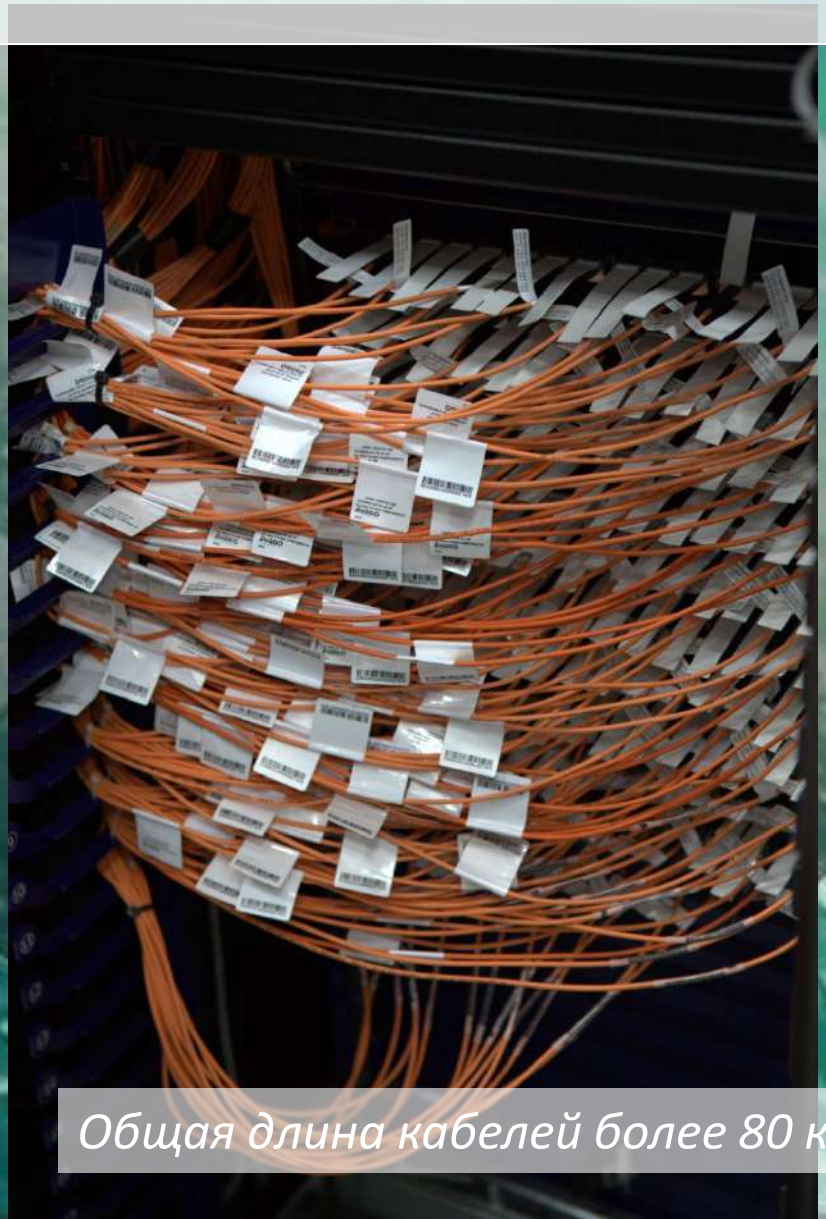
Суперкомпьютер МГУ “Ломоносов”

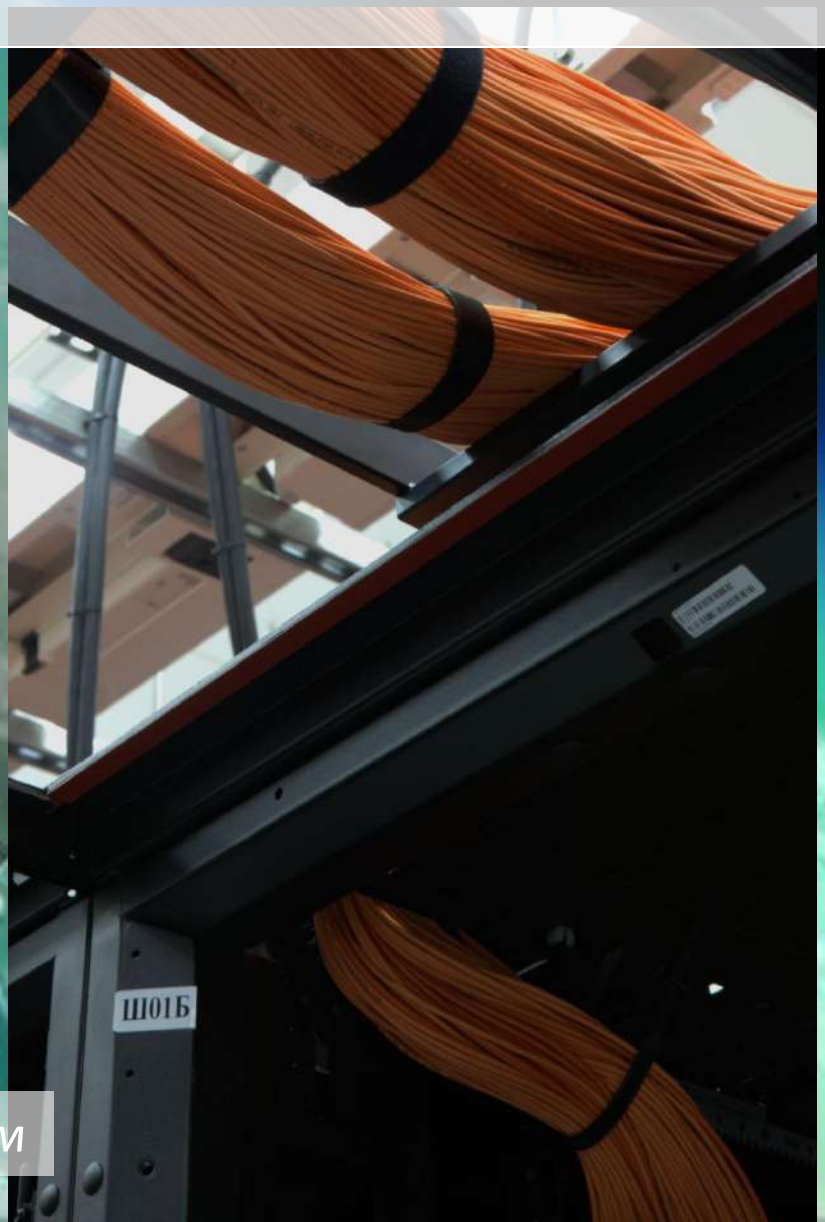
Суперкомпьютер МГУ “Ломоносов”

Суперкомпьютер МГУ “Ломоносов”

Вес оборудования в маш.зале – 57 т, СБЭ – 92т

Суперкомпьютер МГУ “Ломоносов”





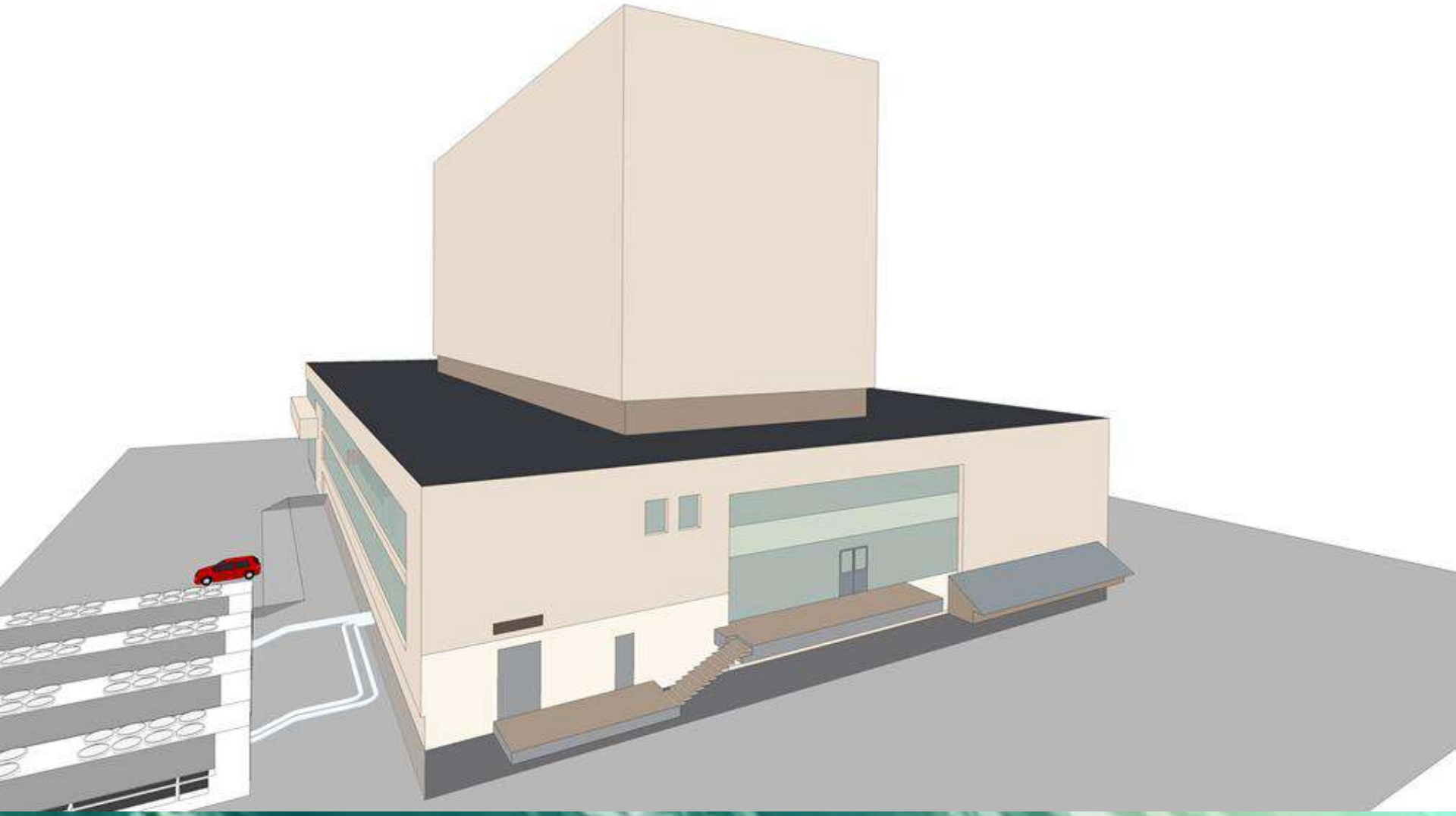
Общая длина кабелей более 80 км

Суперкомпьютер МГУ “Ломоносов”

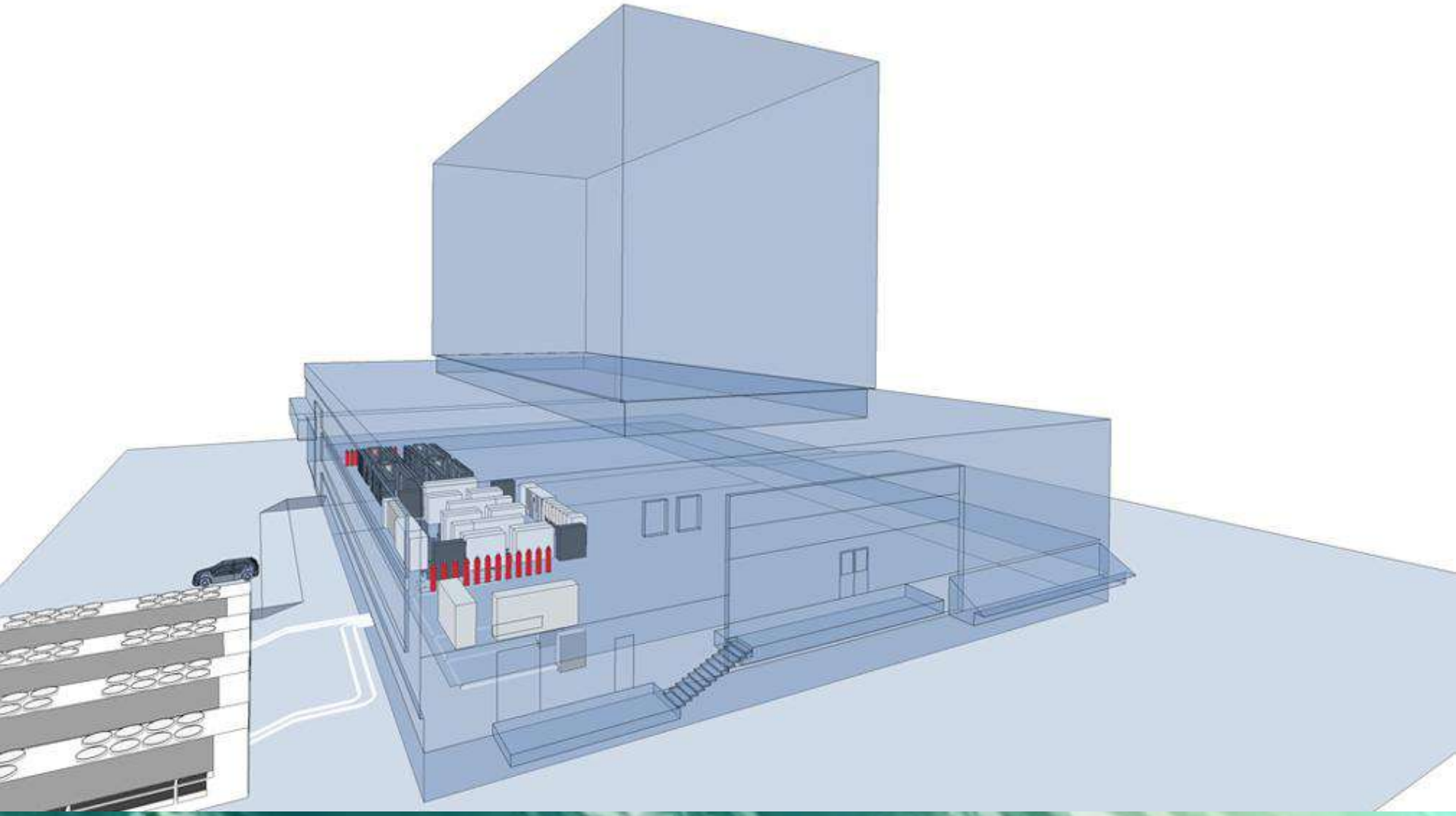
TOP500
This Site is ranked
NO. 12 in the TOP500 List
published 11/2009

Суперкомпьютер МГУ “Ломоносов”

Суперкомпьютер МГУ “Ломоносов”



Суперкомпьютер МГУ “Ломоносов”

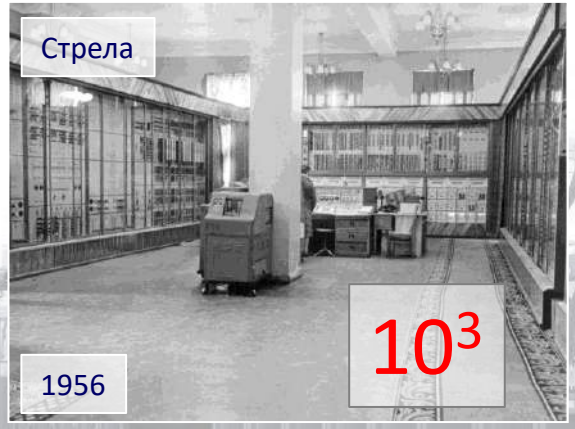


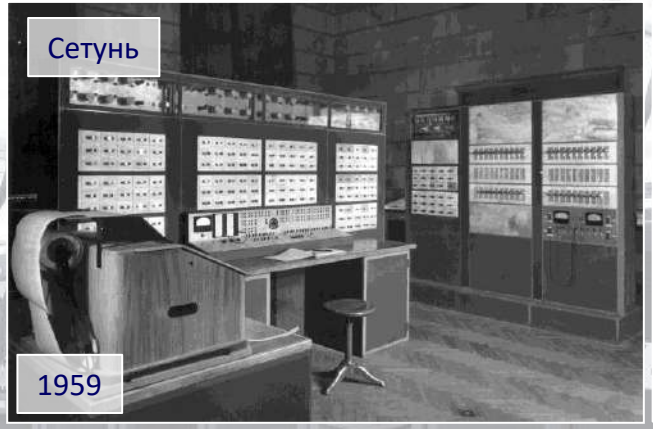
Суперкомпьютер МГУ “Ломоносов”

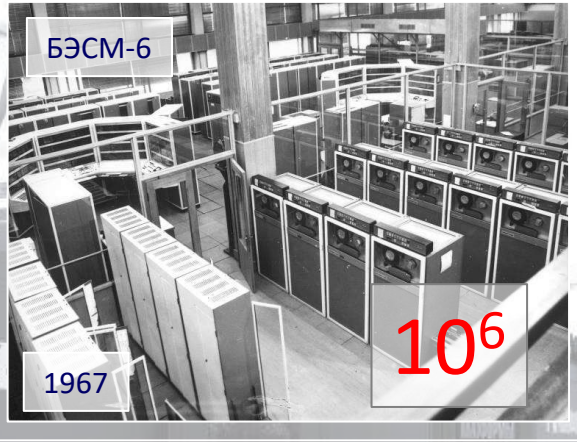
Суперкомпьютер МГУ “Ломоносов”

Компьютерный путь Московского университета

(с 1956 г. до наших дней)

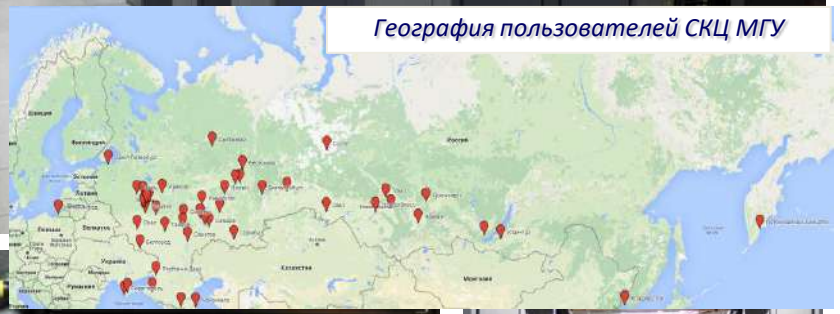






Суперкомпьютер МГУ “Ломоносов-2”

Суперкомпьютерный центр МГУ сегодня:
Пользователи: 2955
Проекты: 880



Факультеты / Институты МГУ: 21
Институты РАН: 95
Университеты России: 102

Города России: 50

1 стойка = 256 узлов: Intel Xeon (14c) + NVIDIA K40/P100= 515 Tflop/s
Суперкомпьютер “Ломоносов-2” = 4.9 Pflop/s

Компьютерный мир в сравнении

Производительность	Вес	Энергопотребление	Цена	
10^6	100 т	3 МВт	$30М	Суперкомпьютеры
10	10 кг	1 КВт	$10К	Серверы…
1	1.5 кг	90 Вт	$1К	ПК, ноутбуки…
0.05	0.1 кг	2 вт	$300	Планшеты, смартфоны…

Суперкомпьютеры... Зачем?

Неужели есть настолько сложные задачи, что для их решения хорошего сервера не хватает ?

Неужели есть настолько важные задачи, которые оправдывают крайне высокую стоимость суперкомпьютеров ?

А далеко ли вычислительно сложные задачи?

Задача о числе счастливых билетиков :

```
count = 0;
for ( I1 = 0; I
  for ( I2 = 0;
    for ( I3 = 0
      for ( I4 =
        for ( I5 =
          for ( I6
            if(
              (
          }
```

Intel Core Duo 2.6 ГГц:

8 цифр – 0.1 с

10 цифр – 10 с

12 цифр – 1780 с

А далеко ли вычислительно сложные задачи?

Задача о числе счастливых билетиков :

- *Поможет ли оптимизация программы?*
- *Поможет ли использование суперкомпьютера?*

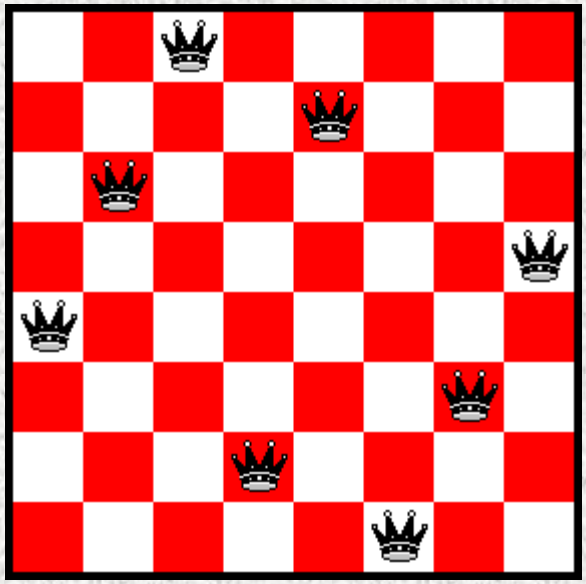
```
count = 0;
for ( I1 = 0; I1 < 10; ++i1)
  for ( I2 = 0; I2 < 10; ++i2)
    for ( I3 = 0; I3 < 10; ++i3)
      for ( I4 = 0; I4 < 10; ++i4)
        for ( I5 = 0; I5 < 10; ++i5)
          for ( I6 = 0; I6 < 10; ++i6) {
              if( i1+i2+i3 == i4+i5+i6 )
                  count = count+1;
}
```

Intel Core Duo 2.6 ГГц:
8 цифр – 0.1 с
10 цифр – 10 с
12 цифр – 1780 с

Вычислительная сложность – ~~10^n~~ операций !

А далеко ли вычислительно сложные задачи?

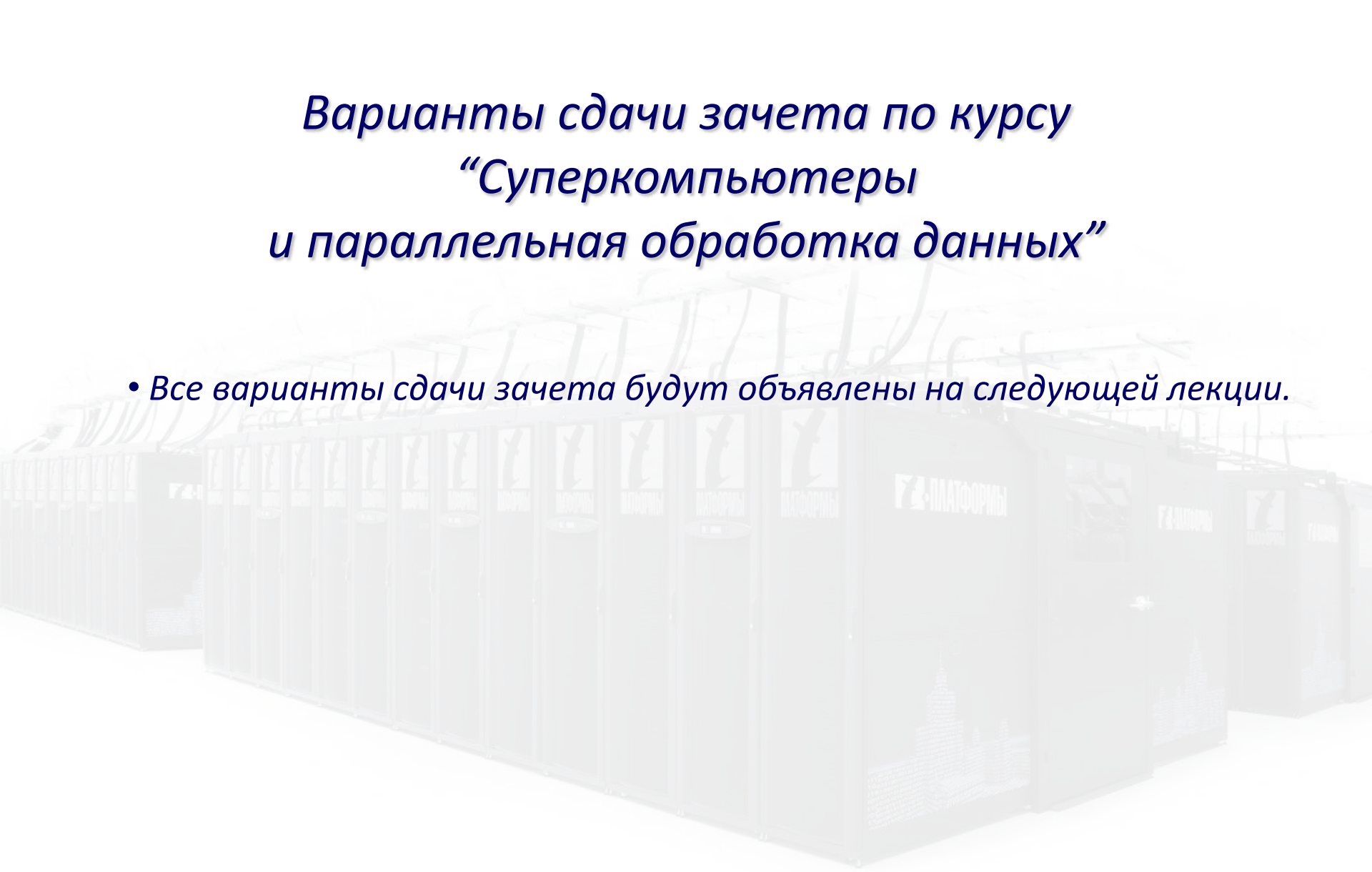
Как разместить 8 ферзей
*на шахматной доске размером 8*8 ?*



Как разместить N ферзей
*на шахматной доске размером K*K ?*

Варианты сдачи зачета по курсу “Суперкомпьютеры и параллельная обработка данных”

- *Все варианты сдачи зачета будут объявлены на следующей лекции.*



Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ

(часть 1)

Вл.В.Воеводин

Зав.кафедрой Суперкомпьютеров и квантовой информатики ВМК МГУ,
Директор НИВЦ МГУ,
чл.-корр. РАН, д.ф.-м.н., профессор

voevodin@parallel.ru

ВМК МГУ, 2020

Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ

(часть 2)

Вл.В.Воеводин

Зав.кафедрой Суперкомпьютеров и квантовой информатики ВМК МГУ,
Директор НИВЦ МГУ,
чл.-корр. РАН, д.ф.-м.н., профессор

voevodin@parallel.ru

ВМК МГУ, 2020

Важные сокращения

Мега (Mega)	$– 10^6$	(миллион)
Гига (Giga)	$– 10^9$	(биллион / миллиард)
Тера (Tera)	$– 10^{12}$	(триллион)
Пета (Peta)	$– 10^{15}$	(квадриллион)
Экза (Exa)	$– 10^{18}$	(квинтиллион)

Флоп/с, Flop/s – Floating point operations per second

15 Tflop/s = $15 * 10^{12}$ арифметических операций в секунду над вещественными числами, представленными в форме с плавающей точкой.

Суперкомпьютер Fujitsu Fugaku, Япония
(#1 Top500 в 2020 г.)

158 976 выч.узлов, 432 стойки

CPU: A64FX ARM v8.2-A, 48/52 cores
Векторные инструкции (SVE),
аппаратная поддержка барьеров…
Int: 1,2,4,8 байт; Float: 16,32,64 бит

Производительность:
Пик (теория): 513 Pflop/s
Тест Linpack: 415 Pflop/s (81%)
Оперативная память = 4.85 Пбайт
Энергопотребление = 28 МВт
Площадь = 1920 м^2

Суперкомпьютеры... Зачем?

Неужели есть настолько сложные задачи, что для их решения хорошего сервера не хватает ?

Неужели есть настолько важные задачи, которые оправдывают крайне высокую стоимость суперкомпьютеров ?

А далеко ли вычислительно сложные задачи?

Задача о числе счастливых билетиков :

```
count = 0;
for ( I1 = 0; I
  for ( I2 = 0;
    for ( I3 = 0
      for ( I4 =
        for ( I5 =
          for ( I6
            if(
              (
          }
```

Intel Core Duo 2.6 ГГц:

8 цифр – 0.1 с

10 цифр – 10 с

12 цифр – 1780 с

А далеко ли вычислительно сложные задачи?

Задача о числе счастливых билетиков :

- *Поможет ли оптимизация программы?*
- *Поможет ли использование суперкомпьютера?*

```
count = 0;
for ( I1 = 0; I1 < 10; ++i1)
  for ( I2 = 0; I2 < 10; ++i2)
    for ( I3 = 0; I3 < 10; ++i3)
      for ( I4 = 0; I4 < 10; ++i4)
        for ( I5 = 0; I5 < 10; ++i5)
          for ( I6 = 0; I6 < 10; ++i6) {
              if( i1+i2+i3 == i4+i5+i6 )
                  count = count+1;
}
```

Intel Core Duo 2.6 ГГц:

8 цифр – 0.1 с

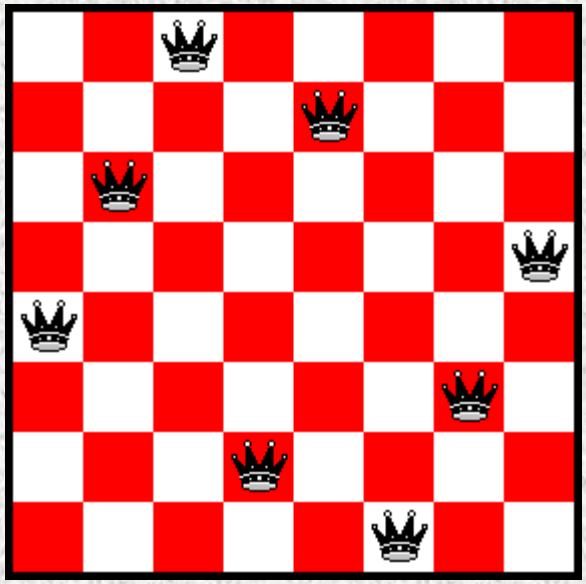
10 цифр – 10 с

12 цифр – 1780 с

Вычислительная сложность – ~~10^n~~ операций !

А далеко ли вычислительно сложные задачи?

Как разместить 8 ферзей
*на шахматной доске размером 8*8 ?*

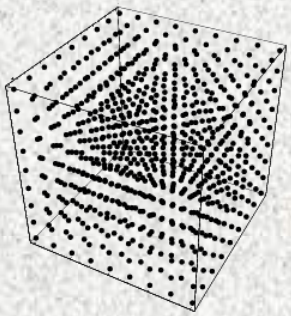


Как разместить N ферзей
*на шахматной доске размером K*K ?*

Сверхвысокая производительность - зачем?

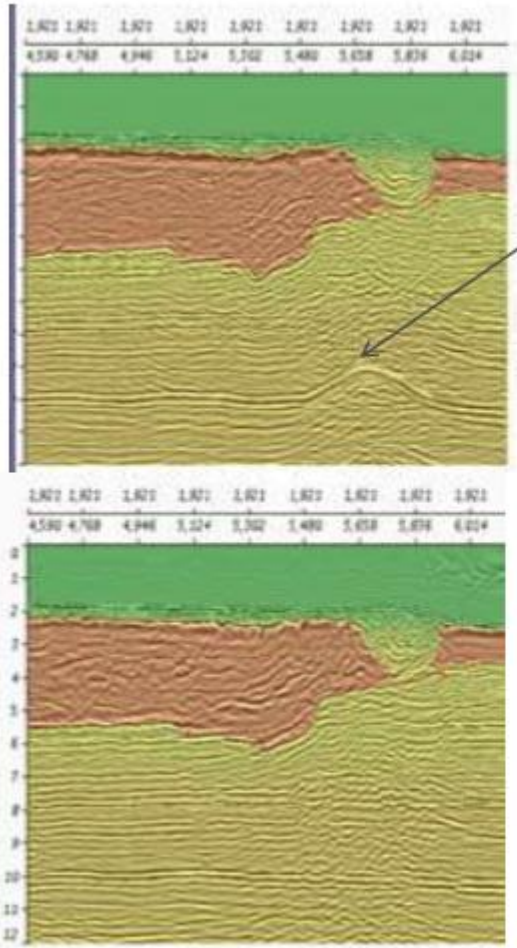
Моделирование нефтяных резервуаров:

- *нефтеносная область – 100*100*100 точек*
- *в каждой точке вычисляется от 5 до 20 функций (скорость, давление, концентрация, температура, …)*
- *200-1000 операций для вычисления каждой функции в каждой точке*
- *100-1000 шагов по времени*
- *Итого: 10^6 (точек сетки) * 10 (функций) * * 500 (операций) * 500 (шагов) = = 2500 млрд. операций*



Суперкомпьютеры для добычи нефти и газа

(Во ВСЕХ добывающих компаниях!)



Seismic profiles of a region of the Gulf of Mexico.

The top image, in 2003 , on **64 processors,**
At the bottom right-hand side , a structure shaped like a bowler hat, typical of a petroleum zone.

Based on this image, ready to install boring equipment on this site.

Fresh data analysis, on a **13 000 cores** supercomputer revealed that the structure was an artefact.

Thanks to HPC, 80 M$ saved

In the **mid-90's, only 40%** of deposits fulfilled their promises. Numerical simulations that analyse data obtained by seismic echography have radically changed the playing field. Armed with the new supercomputer … , the Total engineers are **now** hitting the bull's eye **in 60-70% of cases."**

Journal La Recherche, special HPC, July 2009,

Acknowledgements: H. Calandra, Ph. Ricoux (TOTAL)

TOTAL

Courtesy of P.Ricoux, Total, France

Суперкомпьютеры для добычи нефти и газа

(Во ВСЕХ добывающих компаниях!)

HPC wire

Since 1987 - Covering the Fastest Computers in the World and the People Who Run Them

- Home
- Technologies
- Sectors
- AI/ML/DL
- Exascale
- Specials
- Resource Library
- Events
- Job Bank
- About

Eni Takes the Lead in Industrial Supercomputing

By Tiffany Trader

January 23, 2018

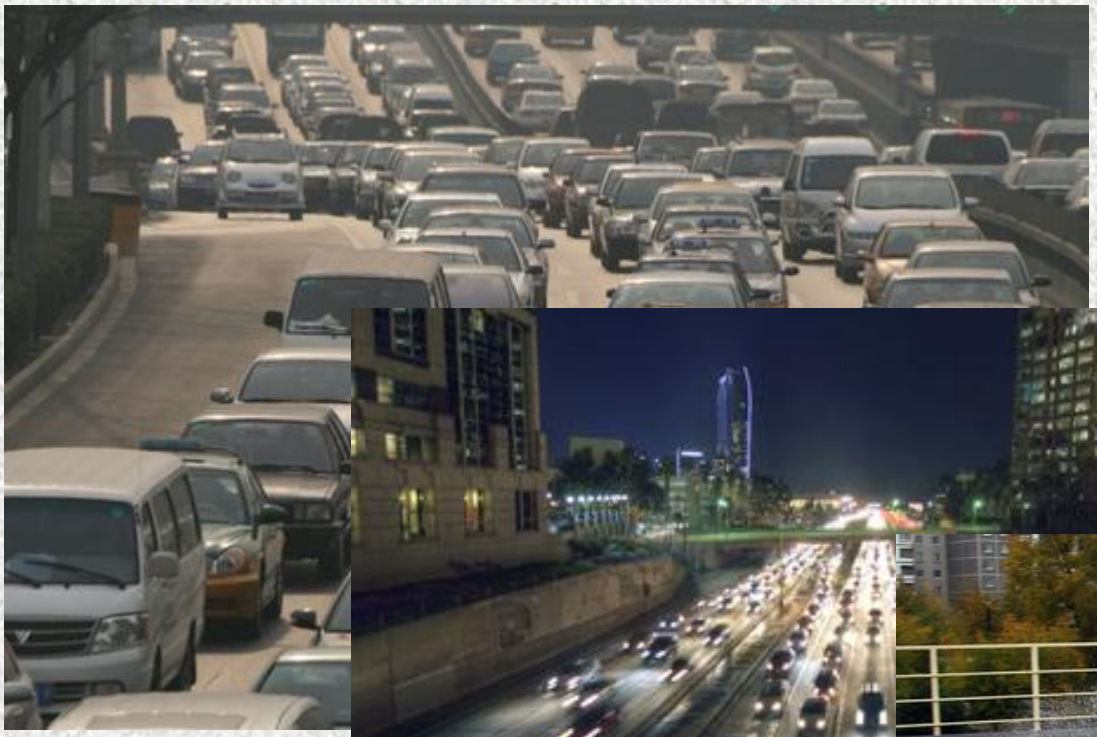
No sooner had one system, used by BP, been declared the most powerful supercomputer in the industrial sphere than it quickly has been displaced. Last Thursday (Jan. 18) it was Italian energy company Eni's turn to take the lead with the launch of its latest petawhopper. At 18.6 petaflops (peak) the new cluster, HPC4, becomes the world's most powerful commercial system (that we know of) and quadruples the company's computing capacity to an aggregate peak performance of 22.4 petaflops.

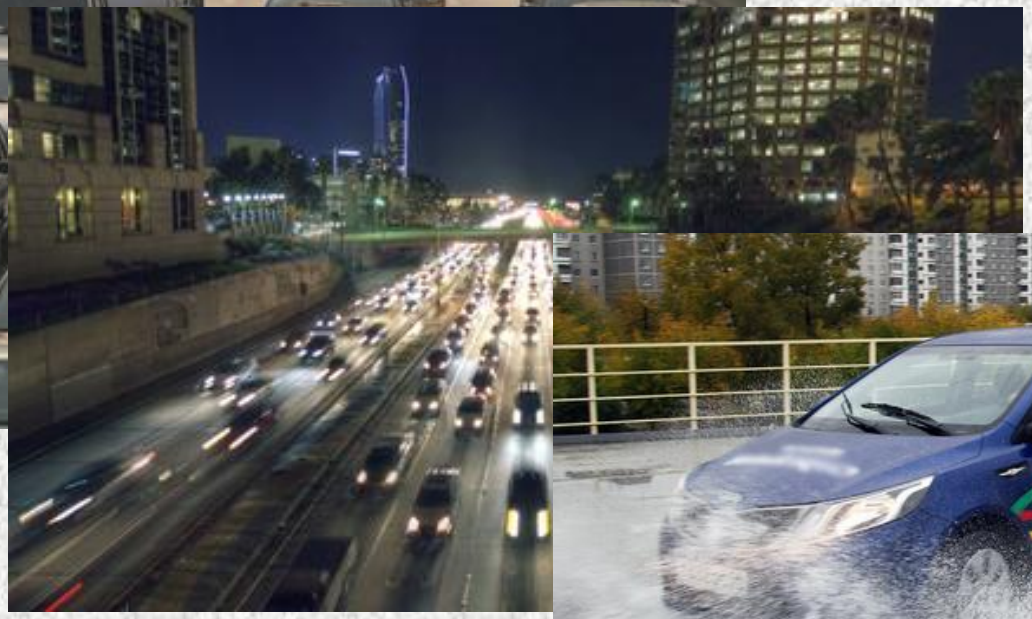
#13 в Top500
(июнь 2018)

#6(!) в Top500
(июнь 2020)

Автомобилестроение

(BMW, Audi, Ford, Chevrolet, Renault,… – ВСЕ!)

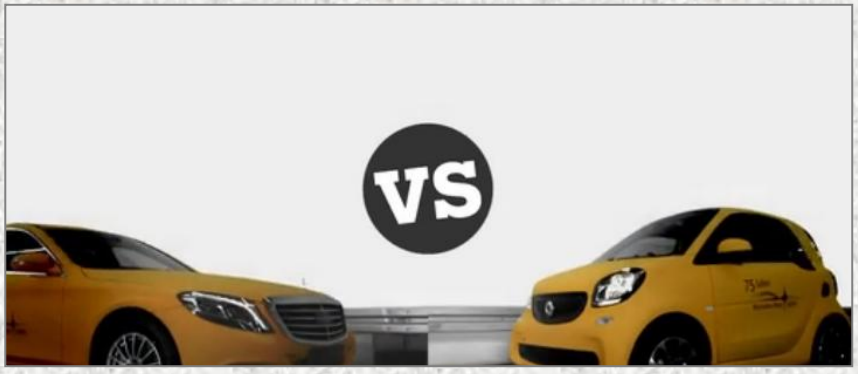




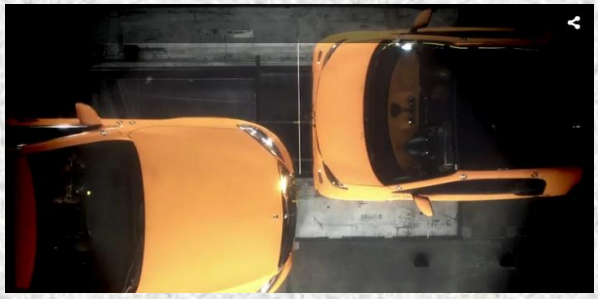
Автомобилестроение

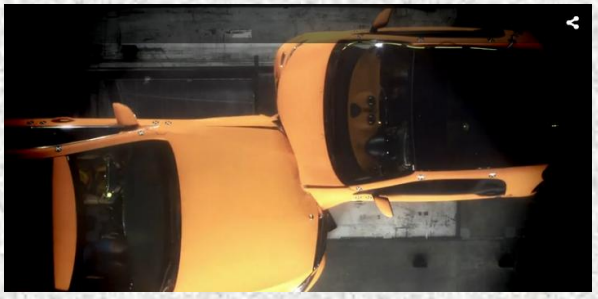
(BMW, Audi, Ford, Chevrolet, Renault,… – ВСЕ!)

Mercedes
E-class



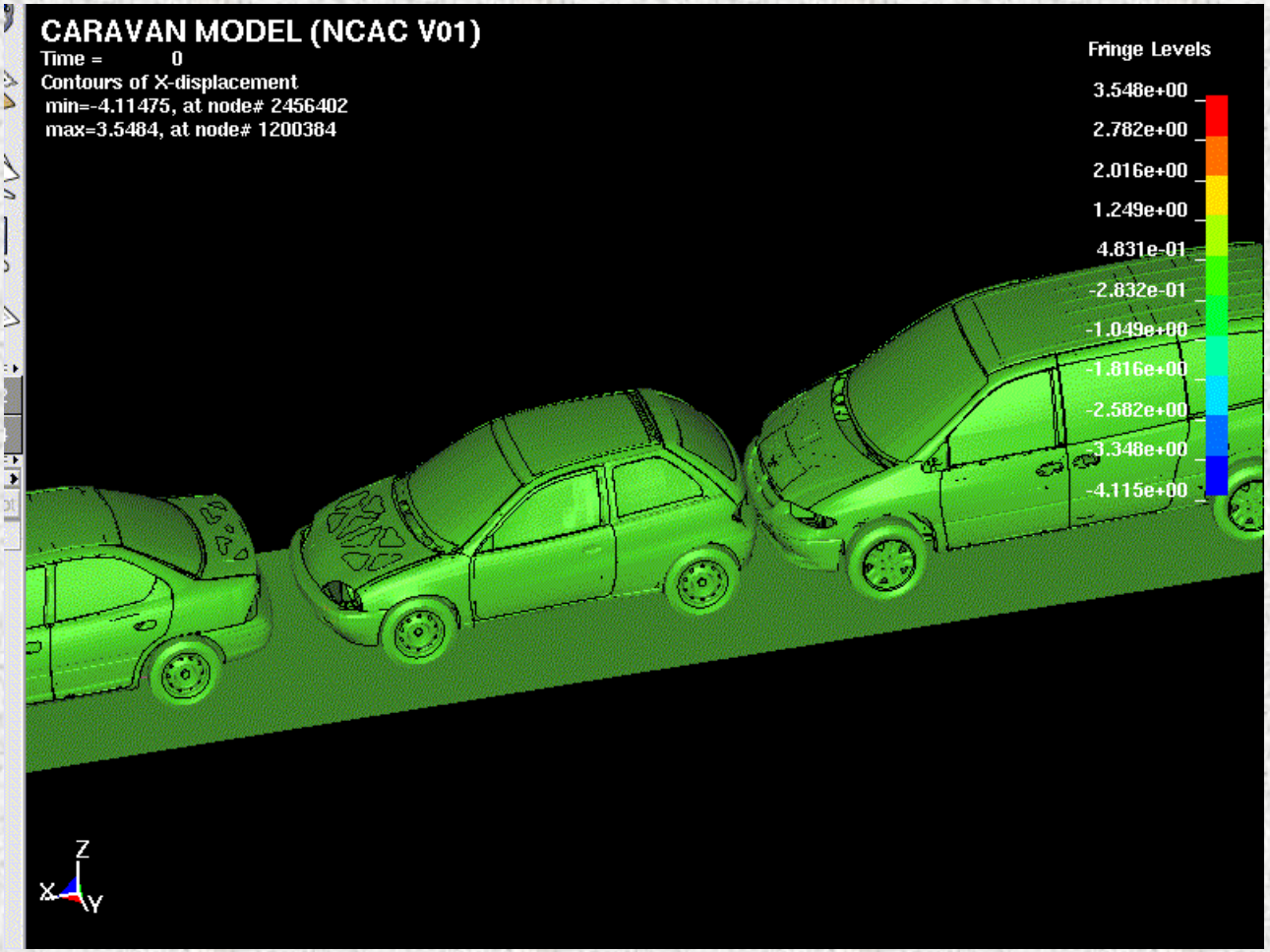
Smart





Автомобилестроение

(BMW, Audi, Ford, Chevrolet, Renault,… – ВСЕ!)

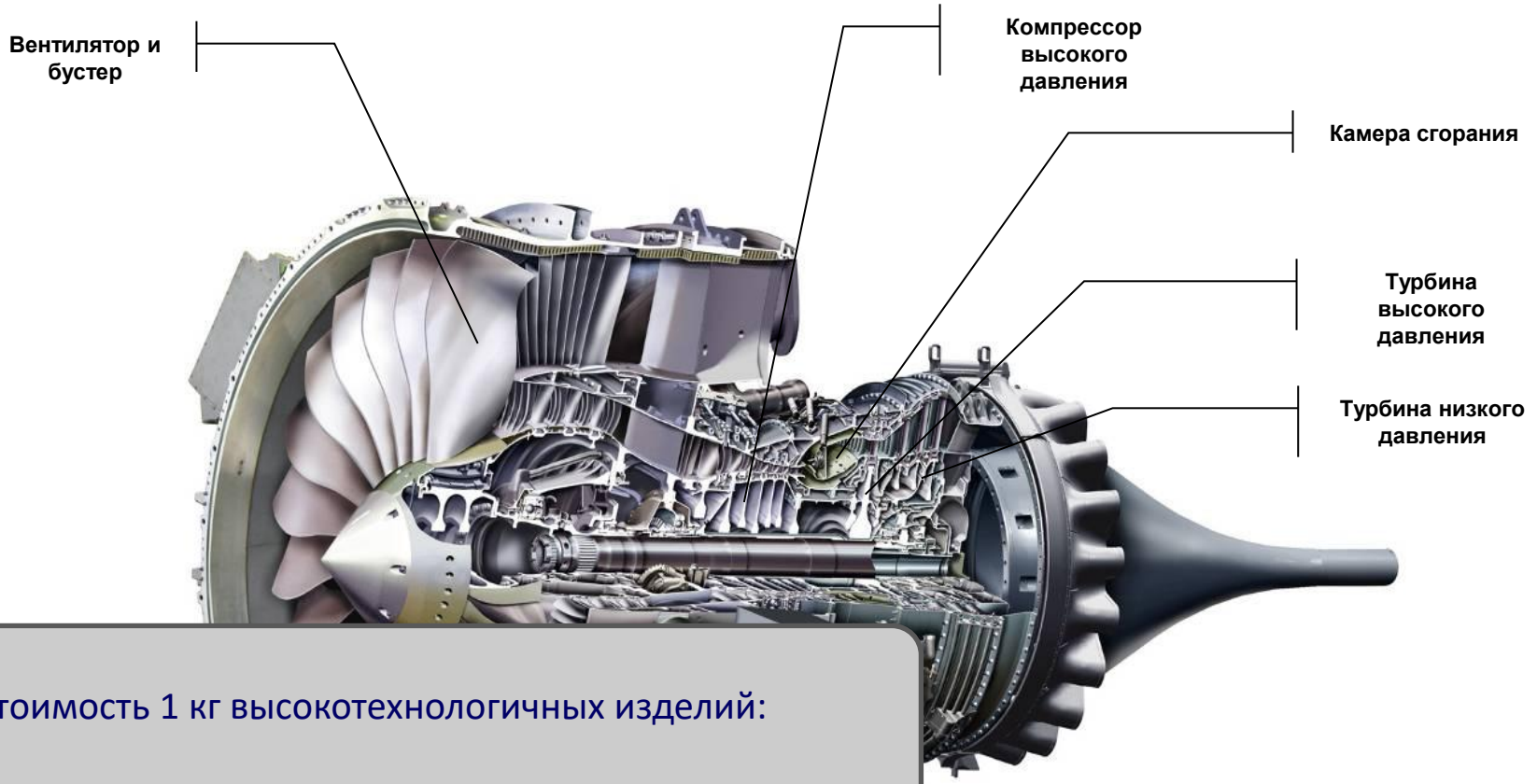


BMW, 2006 г. – 12 Tflop/s
Williams, 2007 г. – 8 Tflop/s
Renault, 2008 г. – 38 Tflop/s
BMW, 2009 г. – 50 Tflop/s

НПО “САТУРН”

(авиационные двигатели и газовые турбины)

Основные узлы ГТД



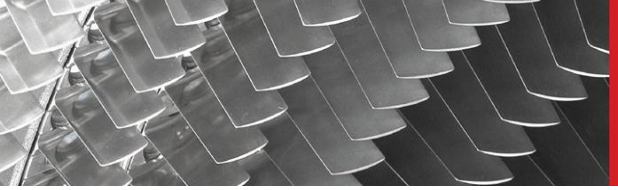
Стоимость 1 кг высокотехнологичных изделий:

Автомобиль: $25

Истребитель пятого поколения: $1000

Двигатель истребителя пятого поколения: $5000

Суперкомпьютер: $400-500



Объем расчетов

Стационарные режимы

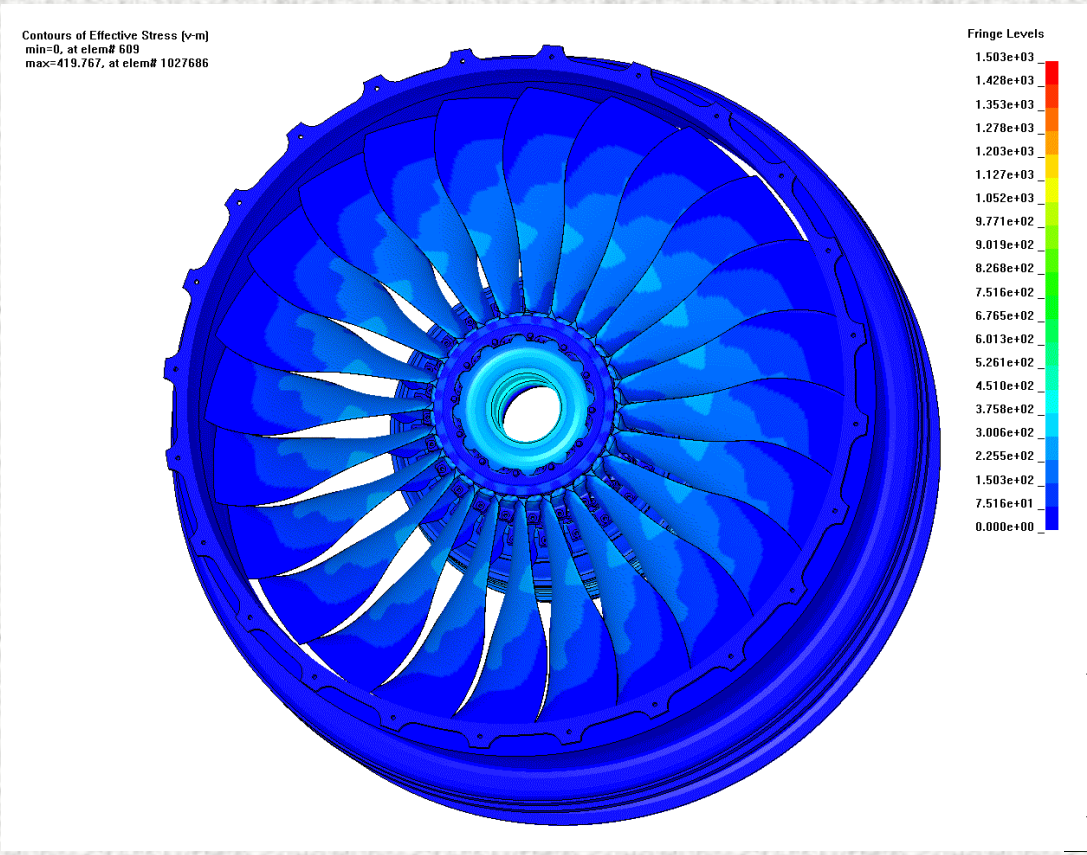
Узел	Размер задачи (млн. ячеек)	Кол-во операций с плавающей точкой для расчета узла
Вентилятор	10	$2,5 * 10^{16}$
Компрессор низкого давления	8	$1,5 * 10^{16}$
Компрессор высокого давления	10	$4,5 * 10^{16}$
Камера сгорания	20	$2,4 * 10^{17}$
Турбина высокого давления	8	$1,1 * 10^{17}$
Турбина низкого давления	8	$1,1 * 10^{16}$

10^{12} – Тера, 10^{15} – Пета, 10^{18} - Экза

Для нестационарных режимов объем расчетов увеличивается в ~ 10 раз.

НПО “САТУРН”

(авиационные двигатели и газовые турбины)



Расчет

Эксперимент

BENCHMARKS AUTRES INDUSTRIES

Wing prototypes: **77**

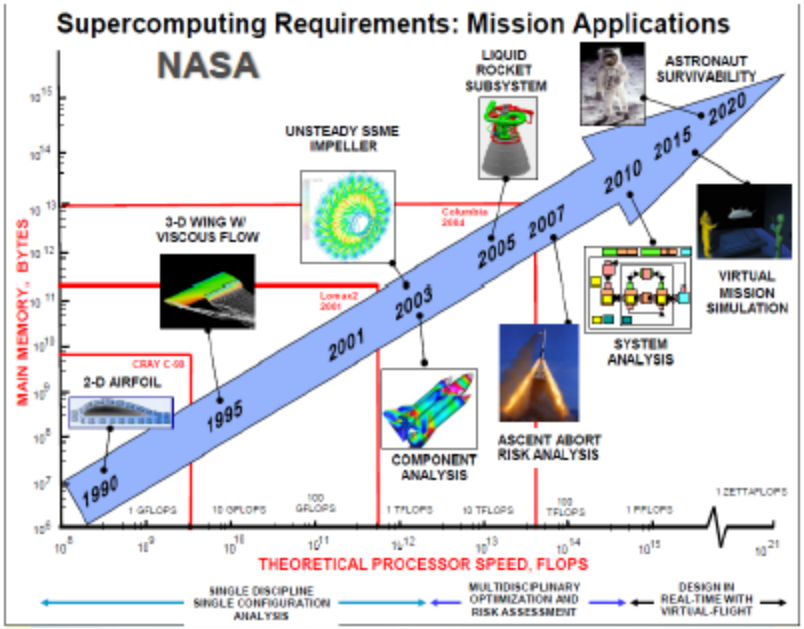
Boeing From 767 to 787

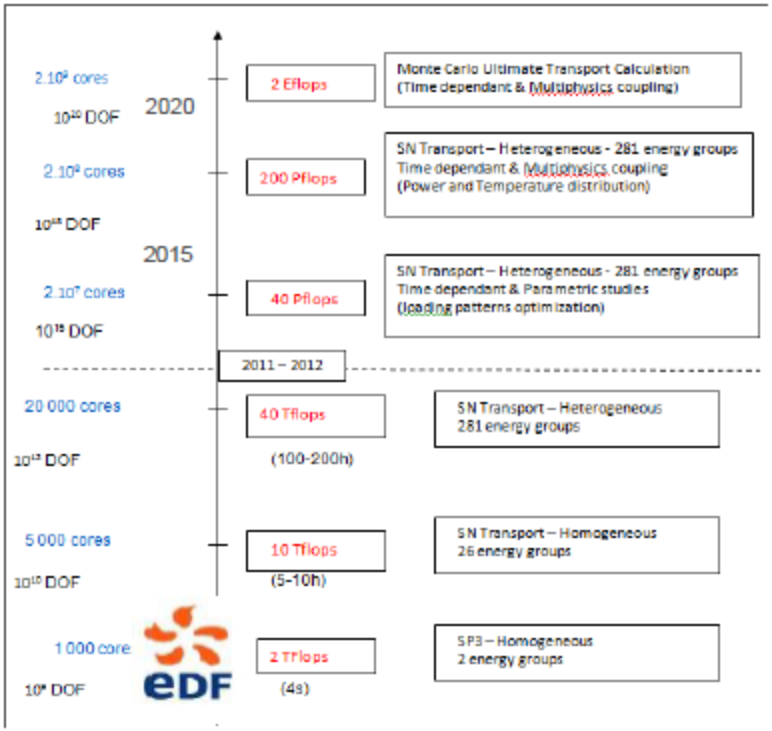
- **40%** less **wind-tunnel days**
- **25%** saving in aerodynamics **development time**
- **20%** saving on wind-tunnel **tests cost**

Thanks to HPC-enabled CFD runs, especially in high-speed regime, even providing better representation of aerodynamics phenomenon turned into better design choices.

But … Digital Aircraft will require at least 1 ZetaFlop !

Wing prototypes: **11** (2008)
→ **5** by new methods and HPC (2015)





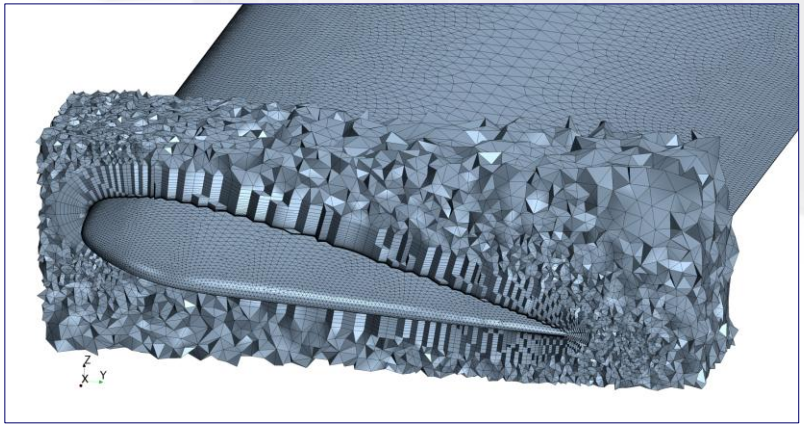
Courtesy of P.Ricoux, Total, France

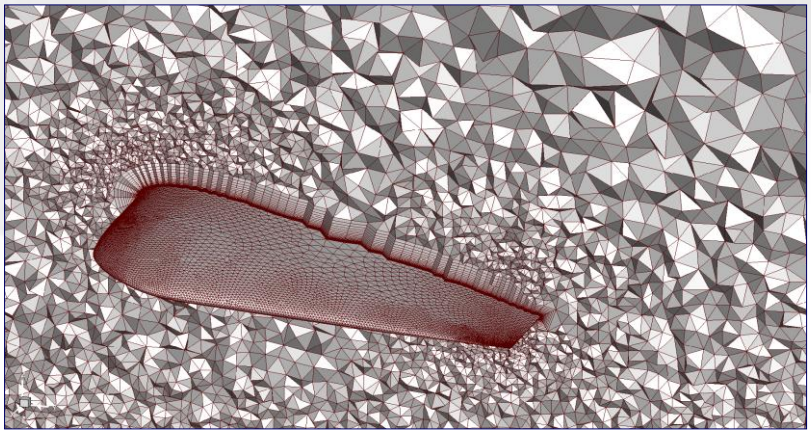
Задачи аэродинамики и аэроакустики

Моделирование простых элементов летательного аппарата: крыло без механизации, отдельной области механизации или шасси, несущего винта вертолета, отсека вооружения и т.д.:
Сетка: около 50 млн.ячеек,
Трудоемкость на ячейку: 10-30 тыс.операций,
Шагов по времени: около 250 тыс.,

Итого: $50*10^6*20*10^3*250*10^3 = 2.5*10^{16}$ операций.
Несколько режимов: разные углы атаки, разная скорость потока, итого – 10^{17} операций.

Расчет сложных конфигураций: крыло с механизацией, крыло с двигателем, несущий винт с учетом фюзеляжа вертолета и т.д.:
Минимальная сетка порядка 100-200 млн.ячеек, итого – 10^{18} операций.





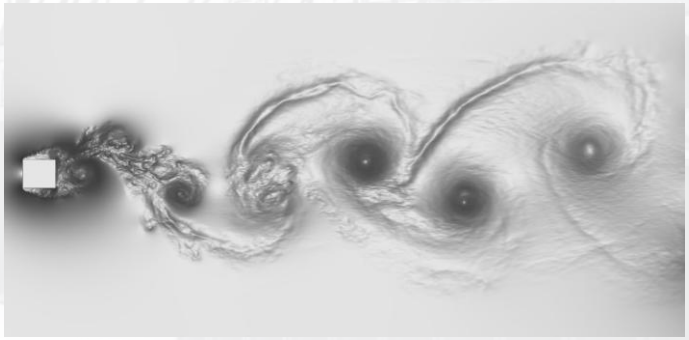
Моделирование течения вокруг лопасти несущего винта вертолета

Предоставлено сектором вычислительной аэроакустики ИПМ им М.В.Келдыша РАН.

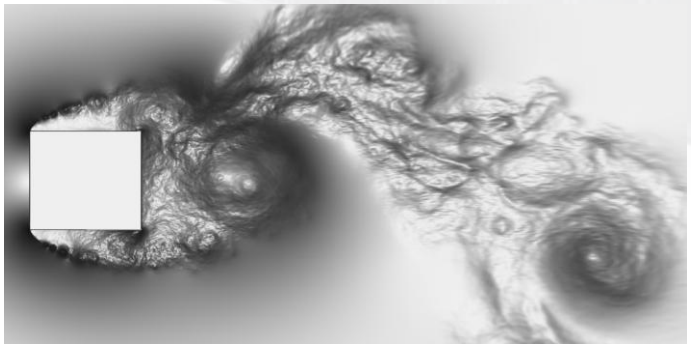
Расчет турбулентных течений методом прямого численного моделирования (DNS)

Течение вокруг бесконечного цилиндра квадратного сечения
Сетка 300 млн. ячеек, схема 4-го порядка
0.7 млн. шагов по времени
Стоимость расчета: около 10^{18} операций.

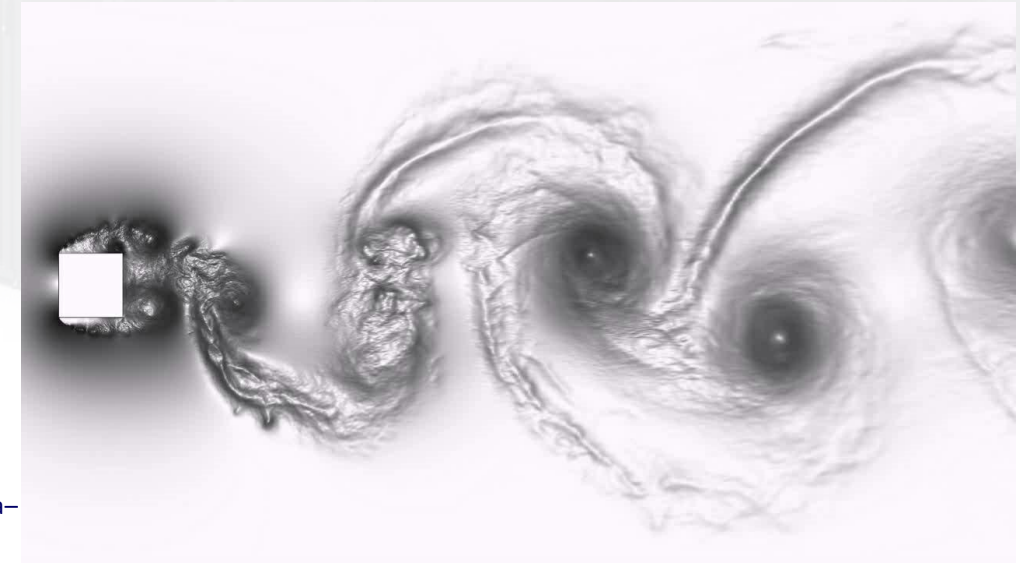
Ламинарное (на переднем плане) и турбулентное течение вокруг подлодки



Турбулентный след за цилиндром (дорожка Кармана)



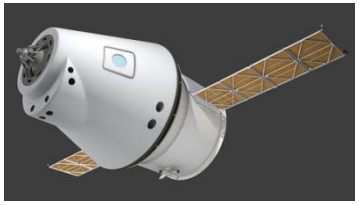
Увеличение в области препятствия. Развитие неустойчивости Кельвина–Гельмгольца (тонкие вихревые дорожки от передних углов цилиндра)



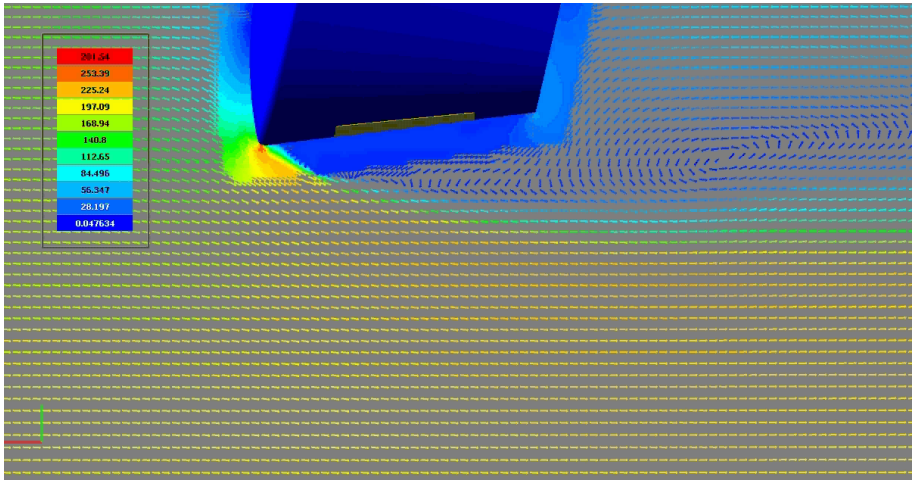
Предоставлено сектором вычислительной аэроакустики ИПМ им М.В.Келдыша РАН.

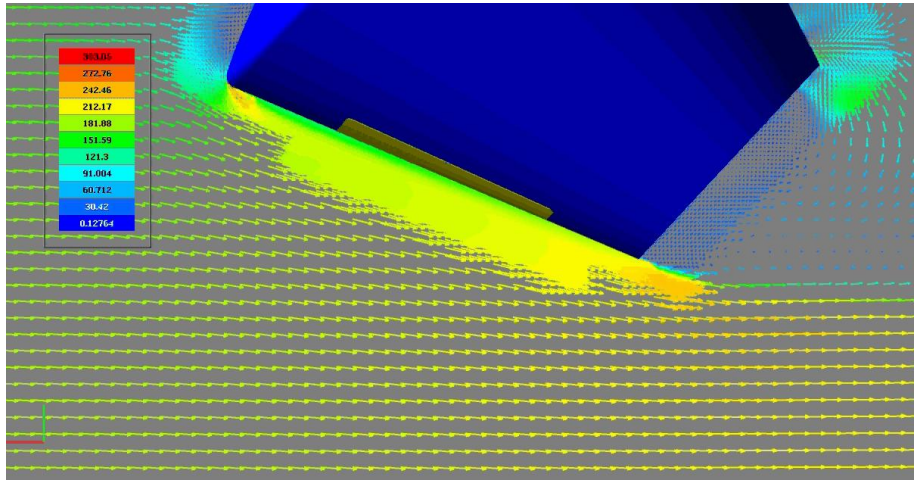
Использование суперкомпьютера “Ломоносов” в проектировании космической техники

Задача: найти минимальный импульс отстрела крышки парашютного отсека пиропатронами, при котором крышка не ударит по корпусу корабля при отстреле.



Эта задача не может быть отработана экспериментально.
Только численное моделирование!





Совместная работа МГУ и компании ТЕСИС для РКК “Энергия”

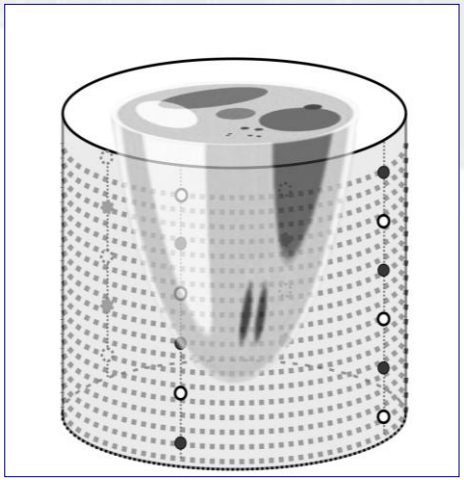
ЭНЕРГИЯ

3D ультразвуковая томография в медицине

Одно из основных приложений — диагностика онкологических заболеваний на ранних стадиях.

- *Сетка: 500*500*500 точек.*
- *Количество источников ультразвука: около 100.*
- *Количество положений приёмников: около 10000.*
- *Обратная задача восстановления внутренней структуры 3D объекта является нелинейной и решается с помощью итерационного процесса. Количество итераций — около 100.*
- *В каждой точке сетки необходимо выполнить порядка 100 операций.*
- *Количество шагов по времени — 1000.*

*Итого: $2.5*10^8$ точек сетки * 10^2 операций * 10^3 шагов * 10^2 источников * 10^2 итераций = $2.5*10^{17}$ операций.*



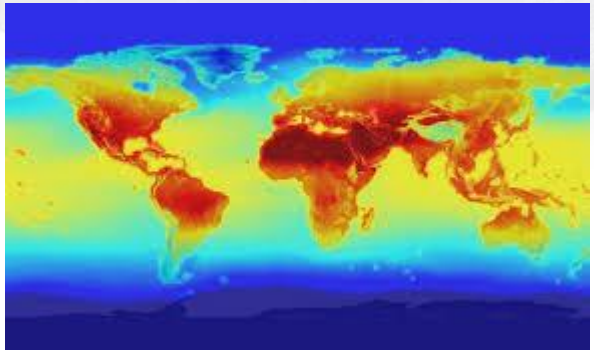
Современные модели климата

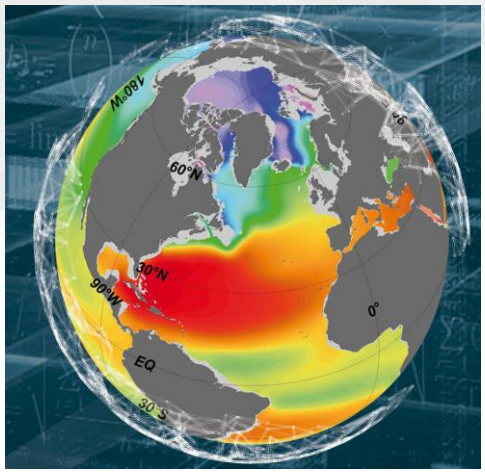
Воспроизведение субмезомасштабных вихрей в океане: вихри с размерами от нескольких сотен метров до нескольких километров.

- *Сетка: 2000*1000*50 точек.*
- *Несколько десятков прогностических переменных (три компоненты скорости, температура, соленость в океане, переменные описывающие взаимодействие химических веществ…).*
- *Порядка 100 операций на каждую переменную.*
- *Для расчета одного года — 10^5 шагов по времени.*
- *Климатические расчеты выполняются на сотни лет.*

*Итого: 10^8 точек сетки * 10 переменных * 10^2 операций * 10^5 шагов * * 10^2 лет расчета = 10^{18} операций.*

- *Близкие оценки для моделей атмосферной циркуляции.*



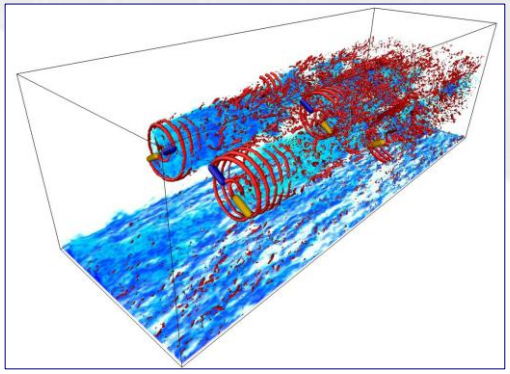


Моделирование параметров течения в ветропарке

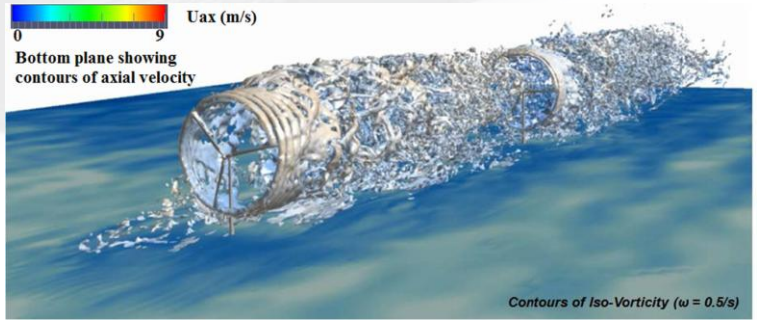
Цель: определение физических параметров течения, эффективного расположения ветроэнергетических установок (ВЭУ) и оценка генерируемой мощности ветропарка.

- *Габариты расчетной области ветропарка: 4 км * 4 км * 1 км.*
- *Общее число ВЭУ мощностью от 2 до 5 МВт: от 20 до 50 (высота 90м, диаметр лопастей около 126м).*
- *Сетка: 572*572*144 точек = $47*10^6$ (примерно 7м между точками).*
- *В каждой точке сетки вычисляется от 25 до 40 функций в зависимости от выбранной модели турбулентности.*
- *Порядка 100 операций на каждую функцию.*
- *Шагов по времени: 56000 (около 3 часов физического времени).*

*Итого: $47*10^6$ точек сетки * 25 функций * 10^2 операций * $56*10^3$ шагов = = $6.5*10^{15}$ операций.*



Поперечное расположение 2 ВЭУ



Продольное расположение 2 ВЭУ

GE Research Uses Summit Supercomputer for Groundbreaking Study on Wind Power

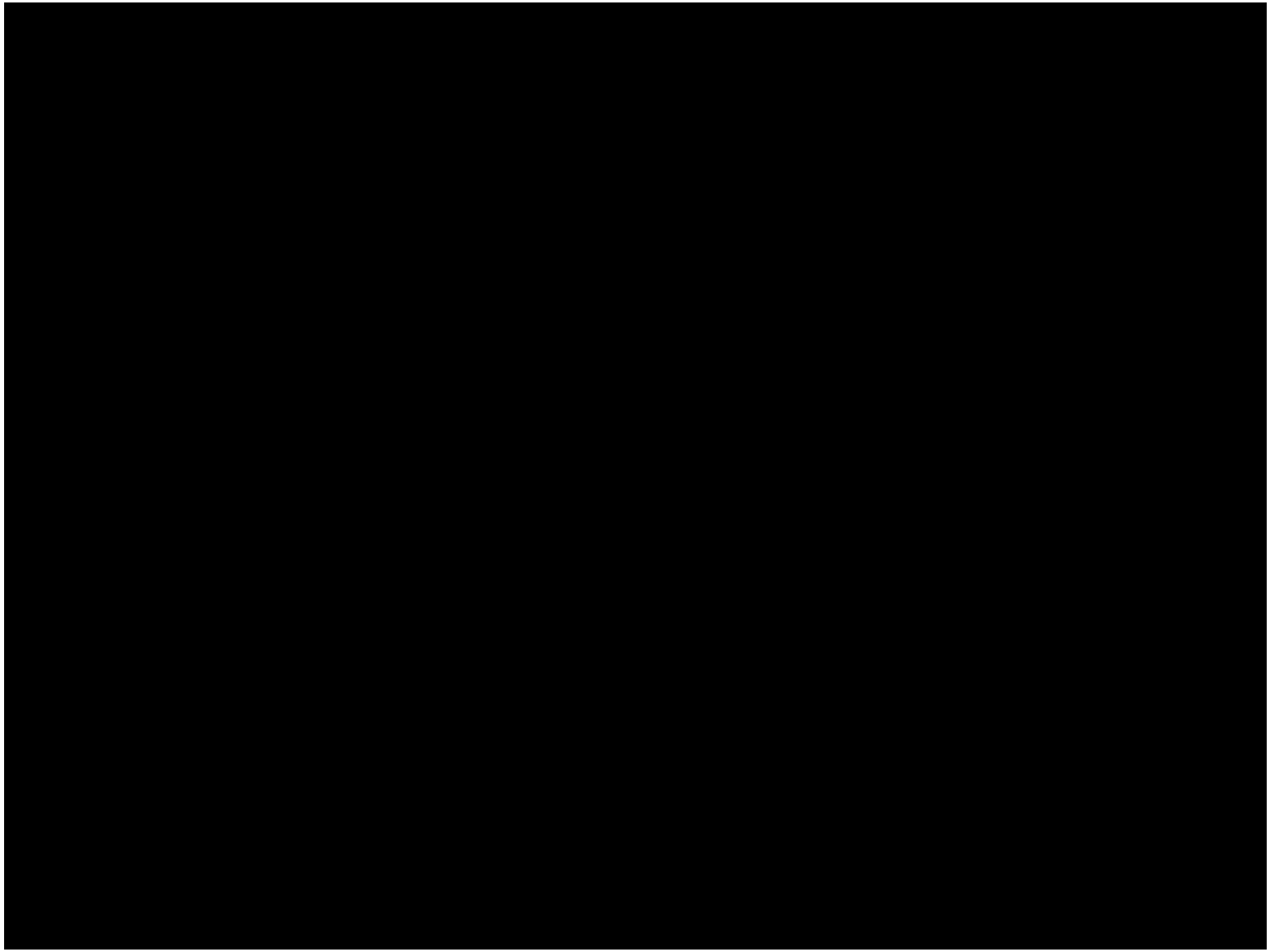
General Electric (GE) recently announced that the Department of Energy has granted GE access to the Summit supercomputer at the Oak Ridge Leadership Computing Facility for research aimed at improving efficiencies in offshore wind energy production.

This work is considered groundbreaking because its use of supercomputer-driven simulations will enable an investigation of wind power that would otherwise have been infeasible.

GE states that the knowledge acquired will strengthen DOE's leadership in exascale computing for clean power through collaboration with the National Renewable Energy Laboratory.

Источники: www.ge.com, www.exascaleproject.org,
August 5, 2020

Суперкомпьютеры в спорте

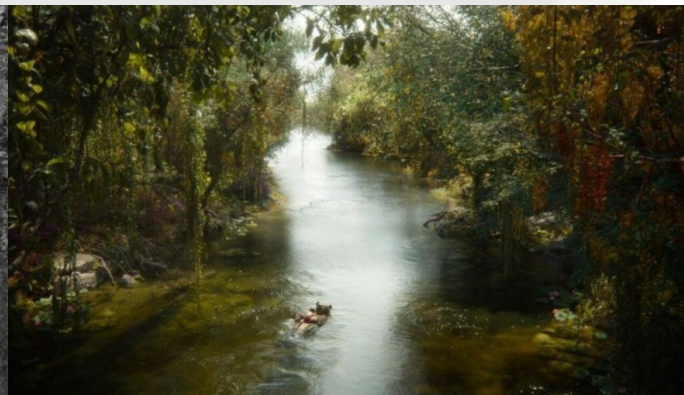


Суперкомпьютеры и анимация

(“Книга Джунглей”, 2016 год)

- 24 кадра в секунду,
- более 152 000 кадров в фильме,
- кадр обрабатывается одним процессором,
- в среднем 19 часов на рендеринг одного кадра,
- всего: 30 млн. процессорочасов
  (потребовалось бы 3400 лет работы одного процессора).

Бюджет: $175 млн. Сборы в мире: $966 млн.



Суперкомпьютеры в кинематографе

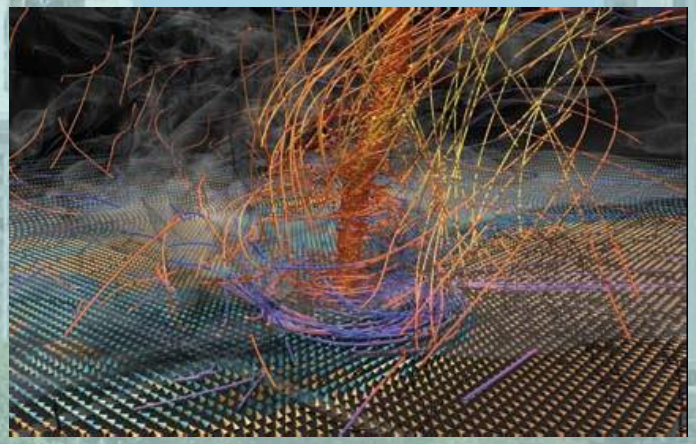
Использование суперкомпьютера МГУ “Ломоносов-2”: спецэффекты для фильма “Время первых”.

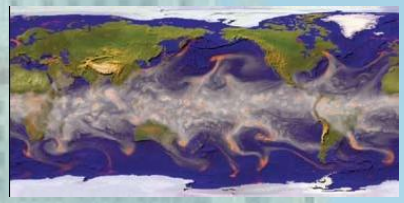
Выступление компании Whirlpool на научной конференции

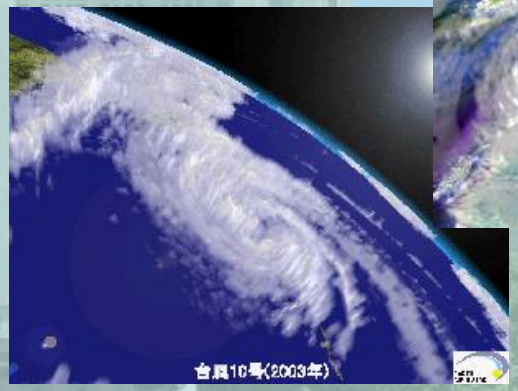
Суперкомпьютерные технологии и наука

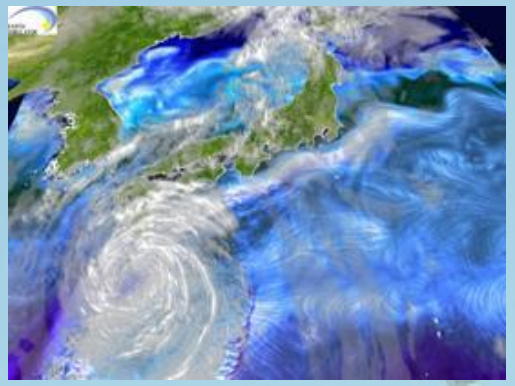
Наука:

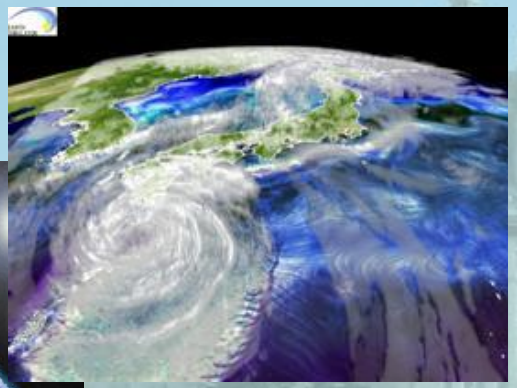
- *теоретическая*
- *экспериментальная*
- *вычислительная*







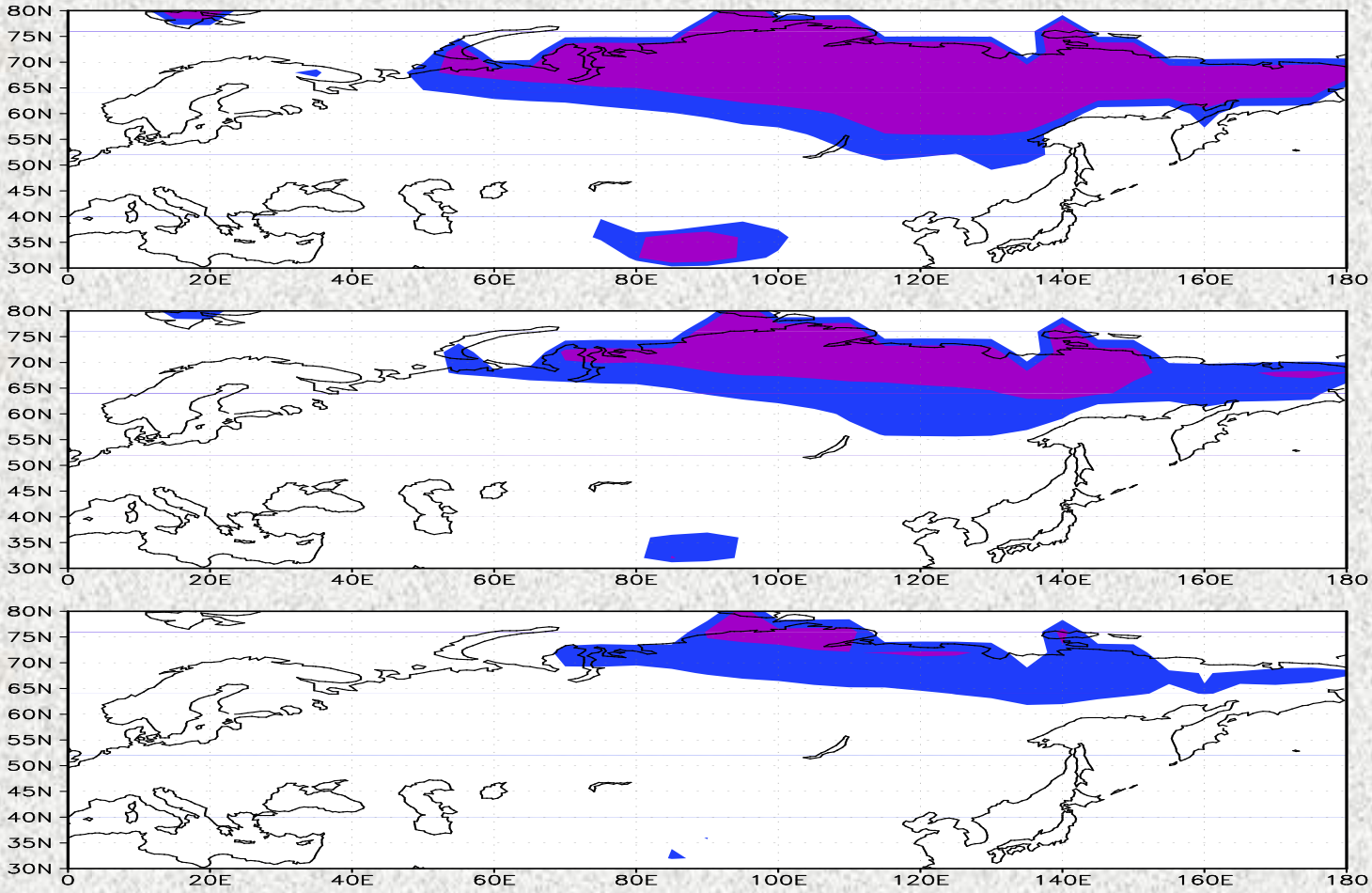




От моделирования к полному воспроизведению …

PARALLEL.RU

Суперкомпьютеры – инструмент моделирования климатических изменений



Верхняя картинка – распределение вечной мерзлоты на территории России.

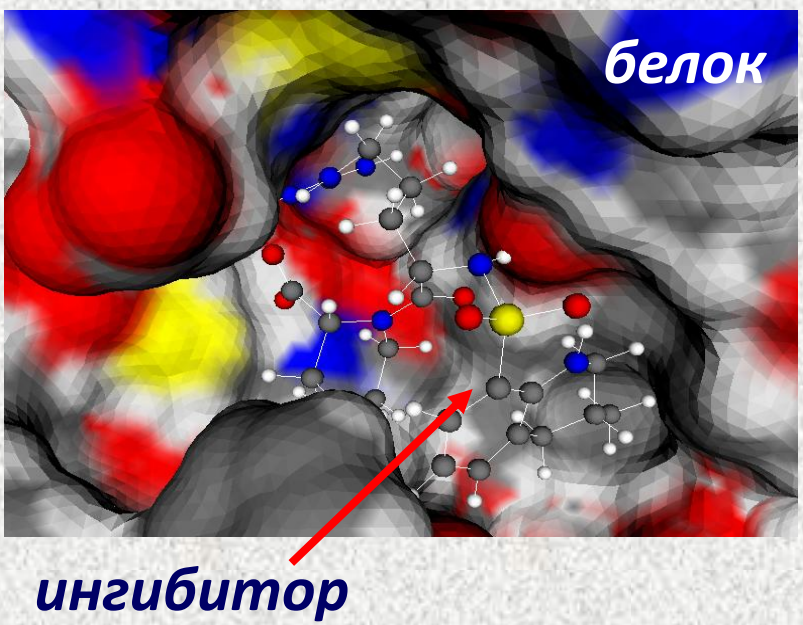
Средняя и нижняя картинки – прогноз распределения вечной мерзлоты на будущее при двух сценариях воздействия человека на климат.

Компьютерный дизайн лекарств

Разработка нового лекарства требует 10-15 лет и до $500 млн.

<u>Действие лекарства:</u>
блокировка ингибитором активного центра белка, пораженного болезнью.

Поиск ингибиторов – это ключевой этап разработки.



***Суперкомпьютеры** и **грид-технологии** позволяют ускорить разработку лекарств в несколько раз и удешевить в сотни раз.*

Теоретико-числовые задачи, ориентированные на криптографические приложения

Факторизация больших целых чисел *требует решения огромных разреженных систем линейных уравнений над конечными полями и* ***компьютеров с петафлопсной производительностью.***

Факторизация: N = p q, где p и q – простые числа.

RSA232=1230186684530117755130494958384962720772853569595334792197322452151726400507263657518745202199786469389956474942774063845925192557326303453731548268507917026122142913461670429214311602221240479274737794080665351419597459856902143413

p=3347807169895689878604416984821269081770479498371376856891243138898288379387800228761471165253174308773781446799948 9

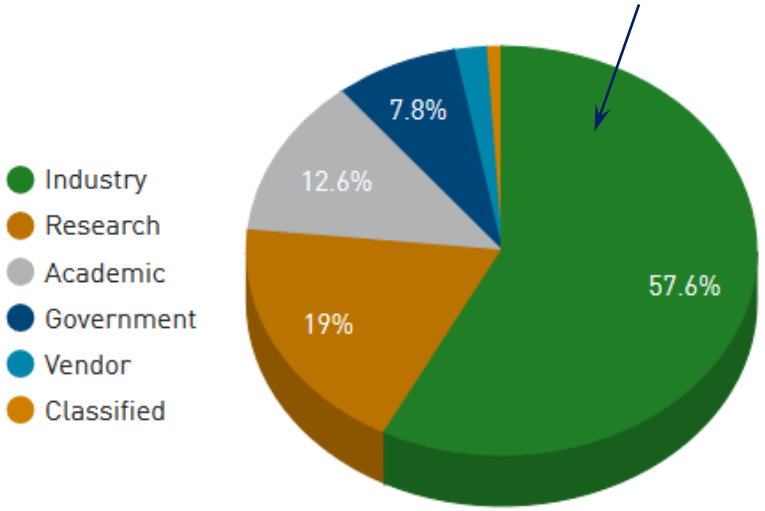
q=3674604366679959042824463379962795263227915816434308764267603228381573966651127923337341714339681027009279873630891 7

Этап просеивания: 2 года на 800 компьютерах.

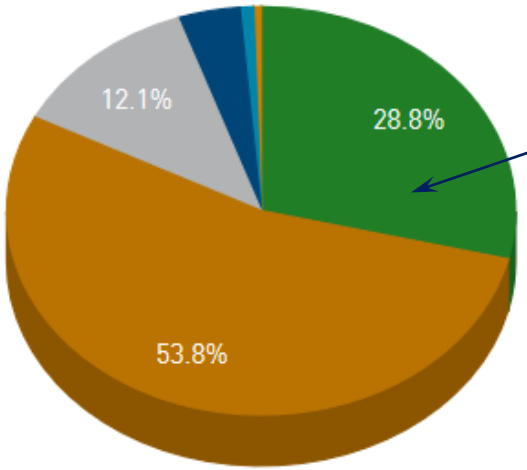
Размерность системы уравнений: $1.9 * 10^9$. Вычислительная сложность: 10^{20} операций.

Top500 самых мощных суперкомпьютеров мира. Распределение по областям применения
(июнь, 2020)

Свидетельство целесообразности использования суперкомпьютерных технологий для развития экономики государств: более половины (56,8%) самых мощных суперкомпьютеров мира установлены в промышленности.



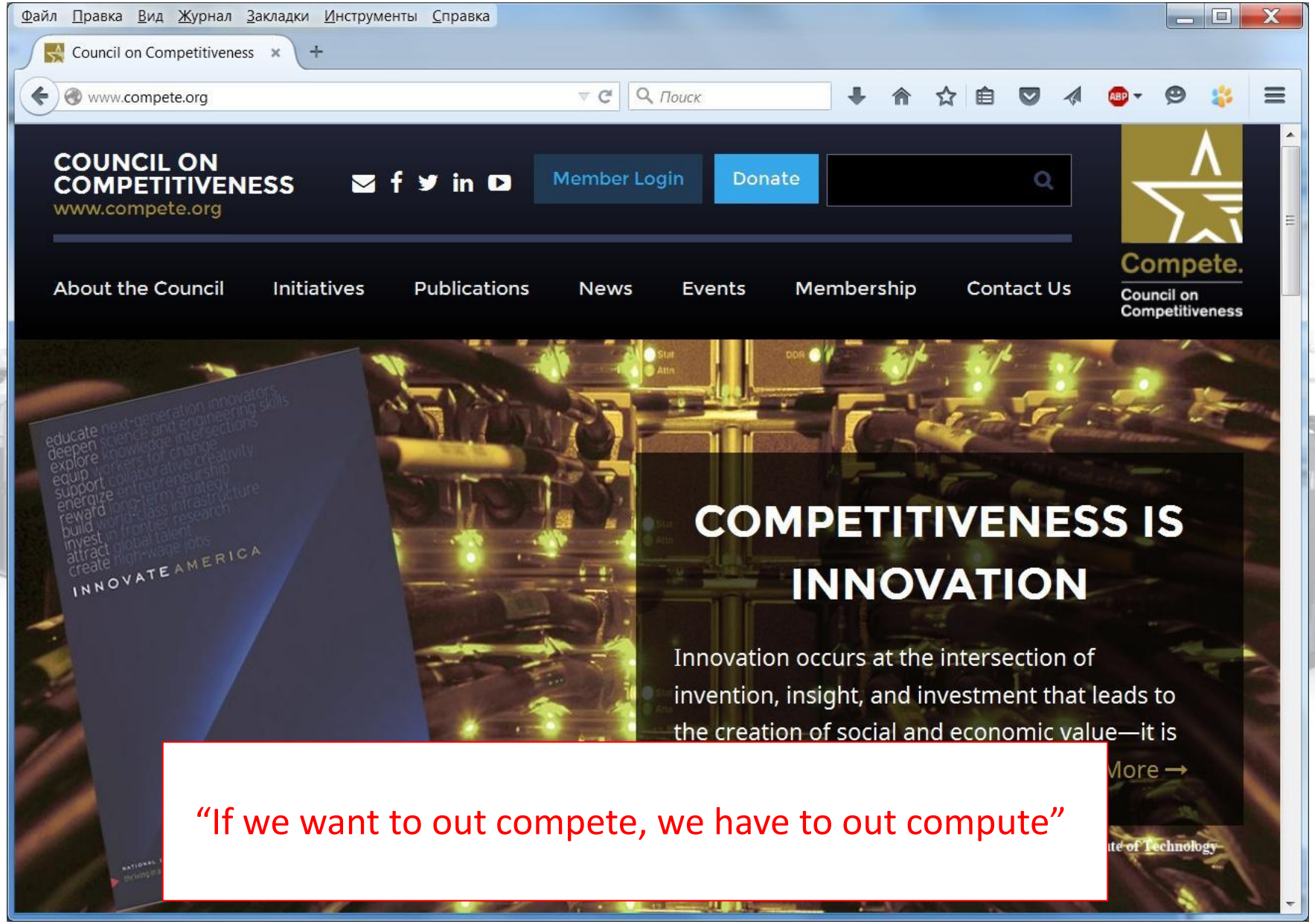
По числу суперкомпьютеров в списке Top500



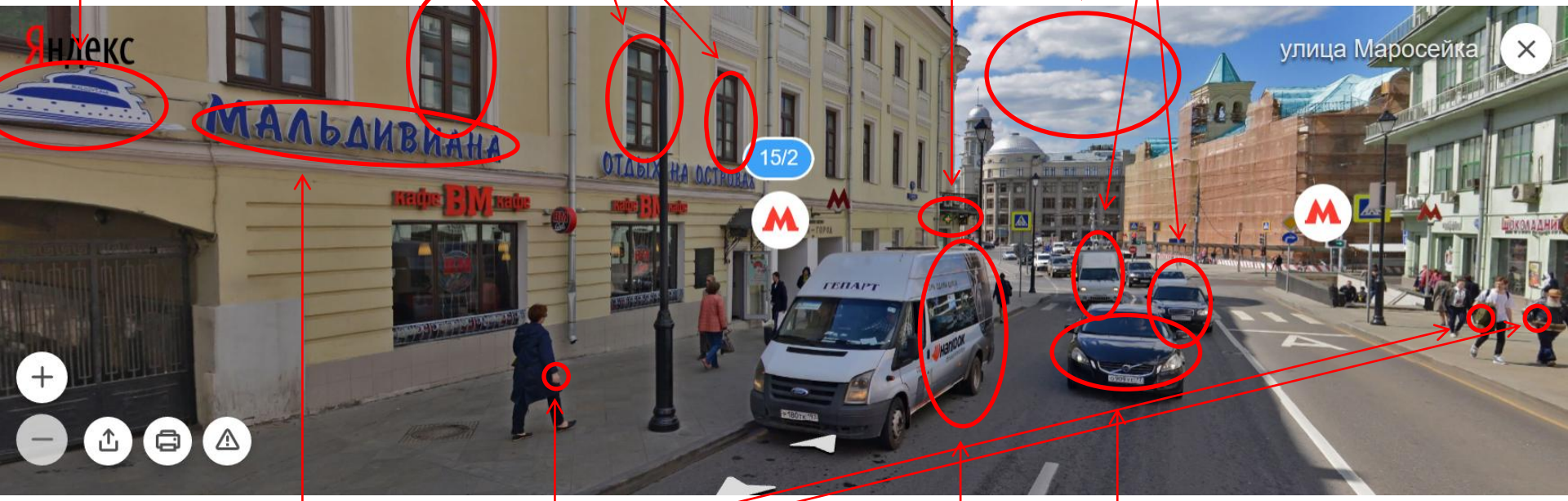
По суммарной производительности суперкомпьютеров

Свидетельство экономической обоснованности использования суперкомпьютерных технологий в промышленности.

TOP 500 The List. - http://Top500.org


Файл Правка Вид Журнал Закладки Инструменты Справка
Council on Competitiveness
www.compete.org
Поиск
COUNCIL ON COMPETITIVENESS
www.compete.org
Member Login
Donate
About the Council
Initiatives
Publications
News
Events
Membership
Contact Us
Compete.
Council on Competitiveness
INNOVATE AMERICA
COMPETITIVENESS IS INNOVATION
Innovation occurs at the intersection of invention, insight, and investment that leads to the creation of social and economic value—it is
More
“If we want to out compete, we have to out compute”

Случайный взгляд на улицы Москвы



Судостроение

Газ

Погода, климат

Аптека

Нефть

Авиация

Мобильные телефоны

Автомобиль

Шины

Суперкомпьютеры... Зачем?

И в промышленности, и в науке есть большое число задач, для решения которых необходимы именно суперкомпьютеры.

Почему суперкомпьютеры имеют столь высокую производительность ?

Прогресс в элементной базе ?
Архитектура суперкомпьютеров ?

Увеличение производительности компьютеров: за счет чего?

EDSAC, 1949 год

такт: $2*10^{-6}$ *с*

произв.: 10^{2} *оп/с*

Увеличение производительности компьютеров: за счет чего?

	EDSAC, 1949 год	*Cray Titan, #1, 2012*
такт:	$2*10^{-6}$ *с*	$4.5*10^{-10}$ *с (2.2 GHz)*
произв.:	10^{2} *оп/с*	$1.7*10^{16}$ *оп/с*

Время такта = 1/(тактовая частота)

Увеличение производительности компьютеров: за счет чего?

	EDSAC, 1949 год	изменение	Cray Titan, #1, 2012
такт:	$2*10^{-6}$ с	$\approx 4.4*10^{3}$	$4.5*10^{-10}$ с (2.2 GHz)
произв.:	10^{2} оп/с	$\approx 1.7*10^{14}$	$1.7*10^{16}$ оп/с

Время такта = 1/(тактовая частота)

Увеличение производительности компьютеров: за счет чего?

	EDSAC, 1949 год	*изменение*	*Fugaku, #1, 2020*
такт:	$2*10^{-6}$ *с*	$\approx 4.4*10^{3}$	$4.5*10^{-10}$ *с (2.2 GHz)*
произв.:	10^{2} *оп/с*	$\approx 4.1*10^{15}$ ↑	$4.1*10^{17}$ *оп/с*

Время такта = 1/(тактовая частота)

Почему суперкомпьютеры имеют столь высокую производительность ?

Два вывода.

1. Безусловно, без развития элементной базы не было бы такого прогресса в развитии компьютеров.

2. Но основной вклад в увеличении производительности компьютеров – это развитие архитектуры, и прежде всего, за счет глубокого внедрения идей параллелизма.

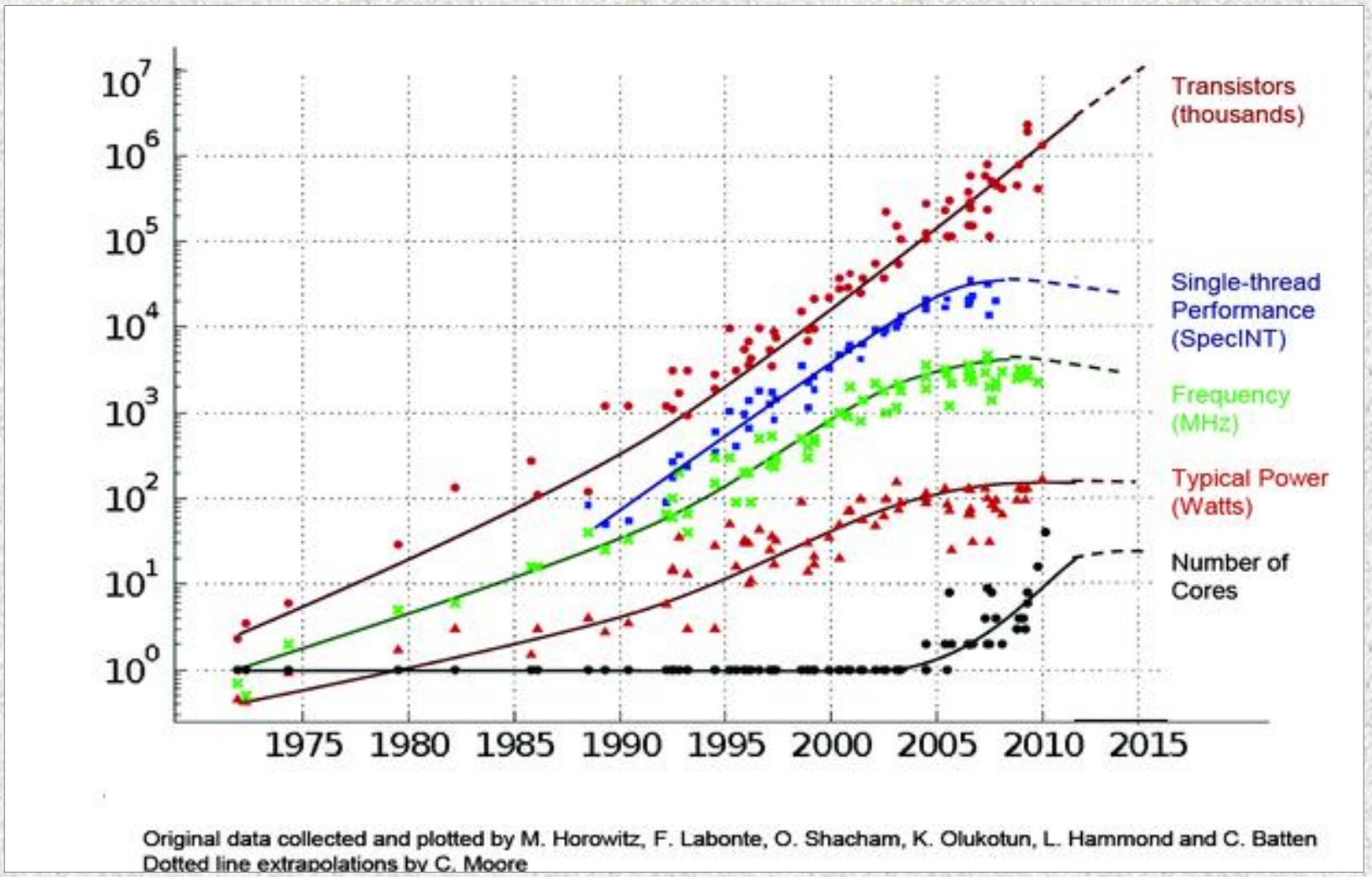
Годы, флопсы и степень параллелизма

(когда и как был достигнут очередной ‘X’flops)

10^6	Mflops	1964 г.	10 MHz	1 CPUs
10^9	Gflops	1985 г.	125 MHz	8 CPUs
10^{12}	Tflops	1997 г.	200 MHz	9152 CPUs
10^{15}	Pflops	2008 г.	3,2 GHz	122400 Cores
10^{18}	Eflops	2020-2022		$10^8 – 10^9$

Изменение параметров микропроцессоров

(остановка роста тактовой частоты)

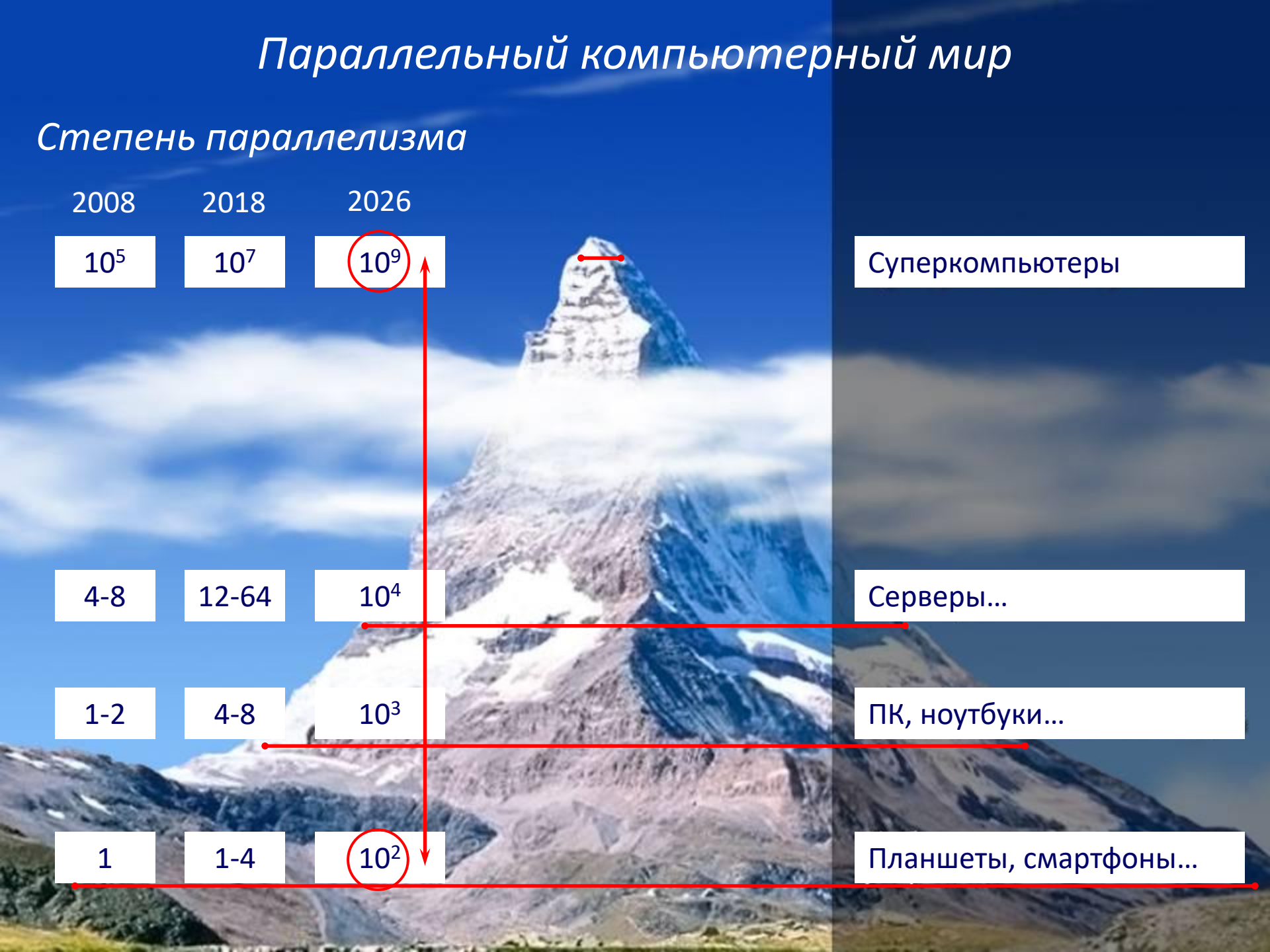


Сегодня основной резерв для роста производительности процессоров – это увеличение числа ядер.

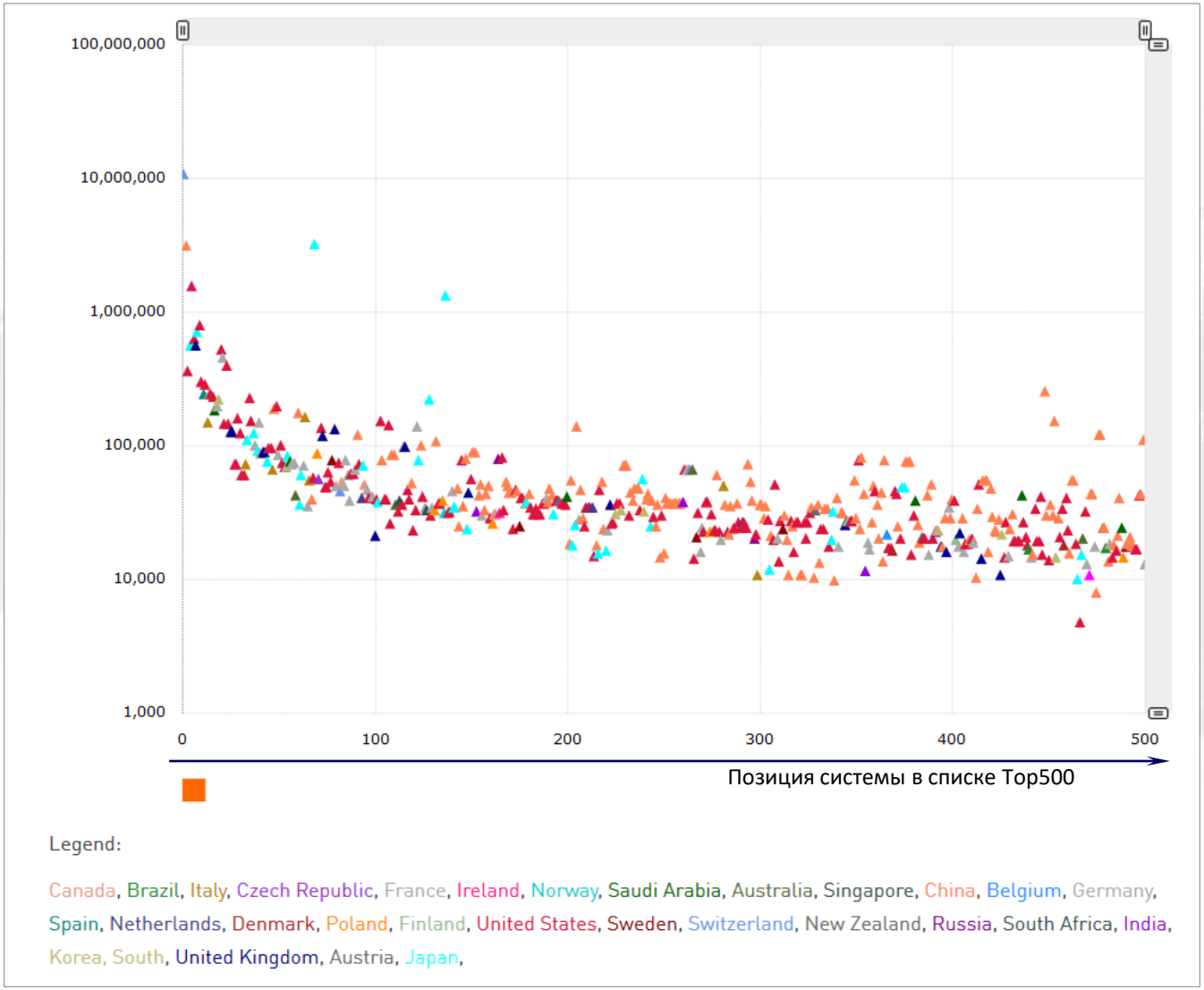
Параллельный компьютерный мир

Степень параллелизма

2008	2018	2026	
10^5	10^7	10^9	Суперкомпьютеры
4-8	12-64	10^4	Серверы…
1-2	4-8	10^3	ПК, ноутбуки…
1	1-4	10^2	Планшеты, смартфоны…

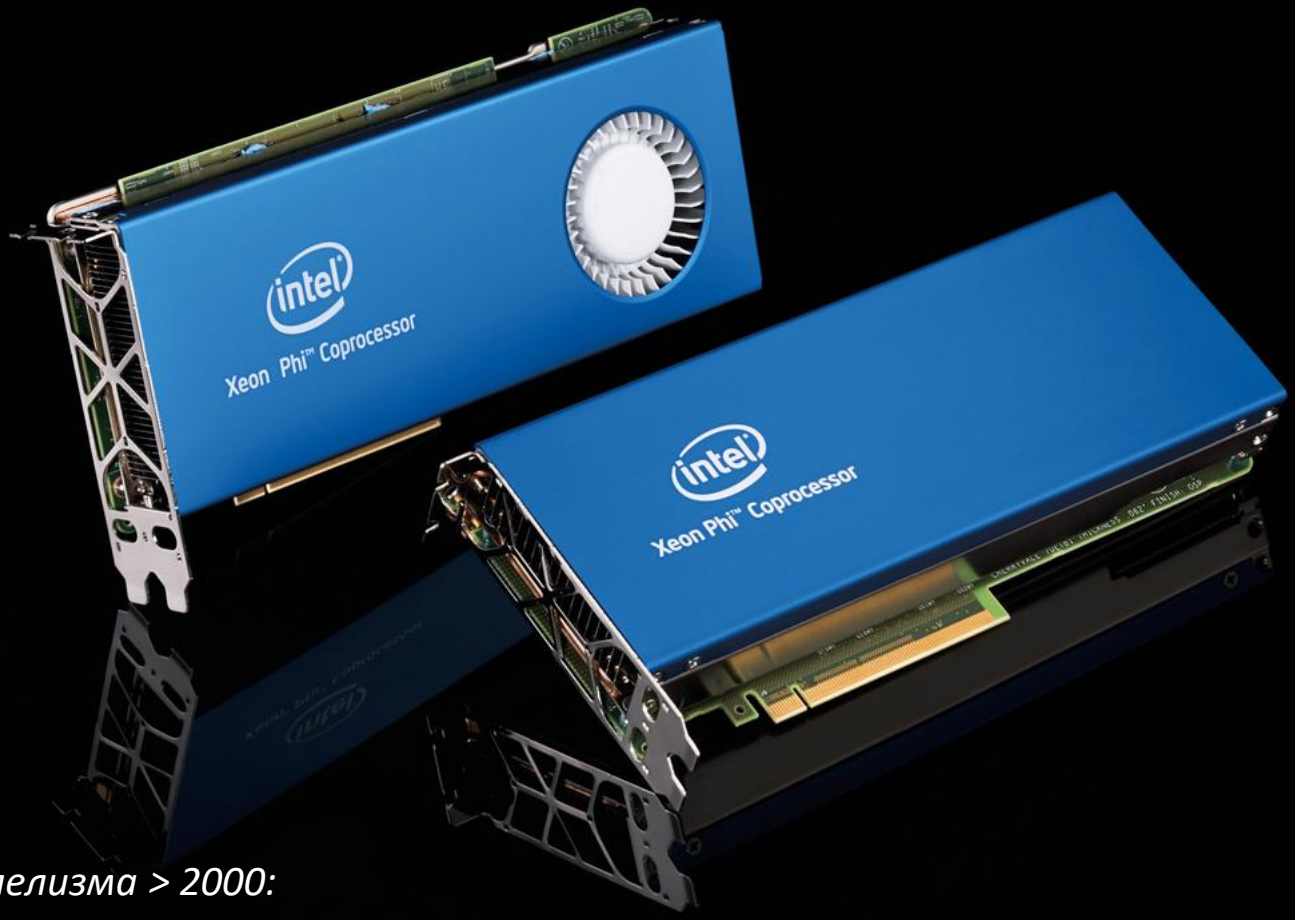


Число ядер в системах списка Top500



Доступный параллелизм: Intel Phi

(Knights Landing)



Степень параллелизма > 2000:

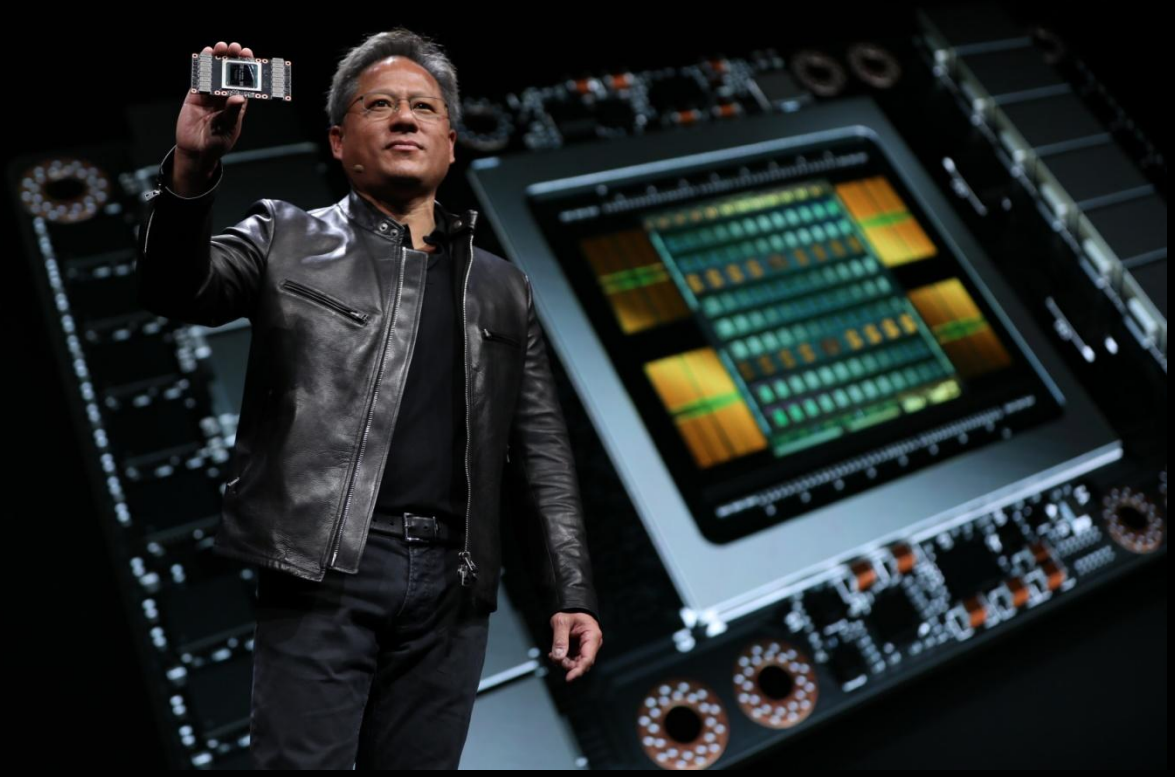
- 72 ядра,

- 4 нити на ядро,

*- AVX-512 инструкции (8 операций * 64 разряда)*

Доступный параллелизм: NVIDIA GPU

(Tesla V100, Volta)



- 5120 ядер,
- 7.5 Тфлопс на двойной точности,
- 15 Тфлопс на одинарной точности,
- 120 Тфлопс (tensor flops, FP16)

Варианты сдачи зачета (экзамена) по курсу “Суперкомпьютеры и параллельная обработка данных”

Пять вариантов описаны на странице: Parallel.ru/vmk20

- *Зачёт (экзамен) в форме беседы с преподавателем.*
- *Зачёт в форме электронного тестирования.*
- *Зачёт за выполнение практического задания.*
- *Конкурс изображений "Параллелизм вокруг нас".*
- *Конкурс выступлений в формате "Научный стенд-ап".*

Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ

(часть 2)

Вл.В.Воеводин

Зав.кафедрой Суперкомпьютеров и квантовой информатики ВМК МГУ,
Директор НИВЦ МГУ,
чл.-корр. РАН, д.ф.-м.н., профессор

voevodin@parallel.ru

ВМК МГУ, 2020

Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ

(часть 3)

Вл.В.Воеводин

Зав.кафедрой Суперкомпьютеров и квантовой информатики ВМК МГУ,
Директор НИВЦ МГУ,
чл.-корр. РАН, д.ф.-м.н., профессор

voevodin@parallel.ru

ВМК МГУ, 2020

Важные сокращения

Мега (Mega)	$– 10^6$	(миллион)
Гига (Giga)	$– 10^9$	(биллион / миллиард)
Тера (Tera)	$– 10^{12}$	(триллион)
Пета (Peta)	$– 10^{15}$	(квадриллион)
Экза (Exa)	$– 10^{18}$	(квинтиллион)

Флоп/с, Flop/s – Floating point operations per second

15 Tflop/s = $15 * 10^{12}$ арифметических операций в секунду над вещественными числами, представленными в форме с плавающей точкой.

Суперкомпьютер Fujitsu Fugaku, Япония
(#1 Top500 в 2020 г.)

158 976 выч.узлов, 432 стойки

CPU: A64FX ARM v8.2-A, 48/52 cores
Векторные инструкции (SVE),
аппаратная поддержка барьеров…
Int: 1,2,4,8 байт; Float: 16,32,64 бит

Производительность:
Пик (теория): 513 Pflop/s
Тест Linpack: 415 Pflop/s (81%)
Оперативная память = 4.85 Пбайт
Энергопотребление = 28 МВт
Площадь = 1920 м^2

Суперкомпьютеры... Зачем?

Неужели есть настолько сложные задачи, что для их решения хорошего сервера не хватает ?

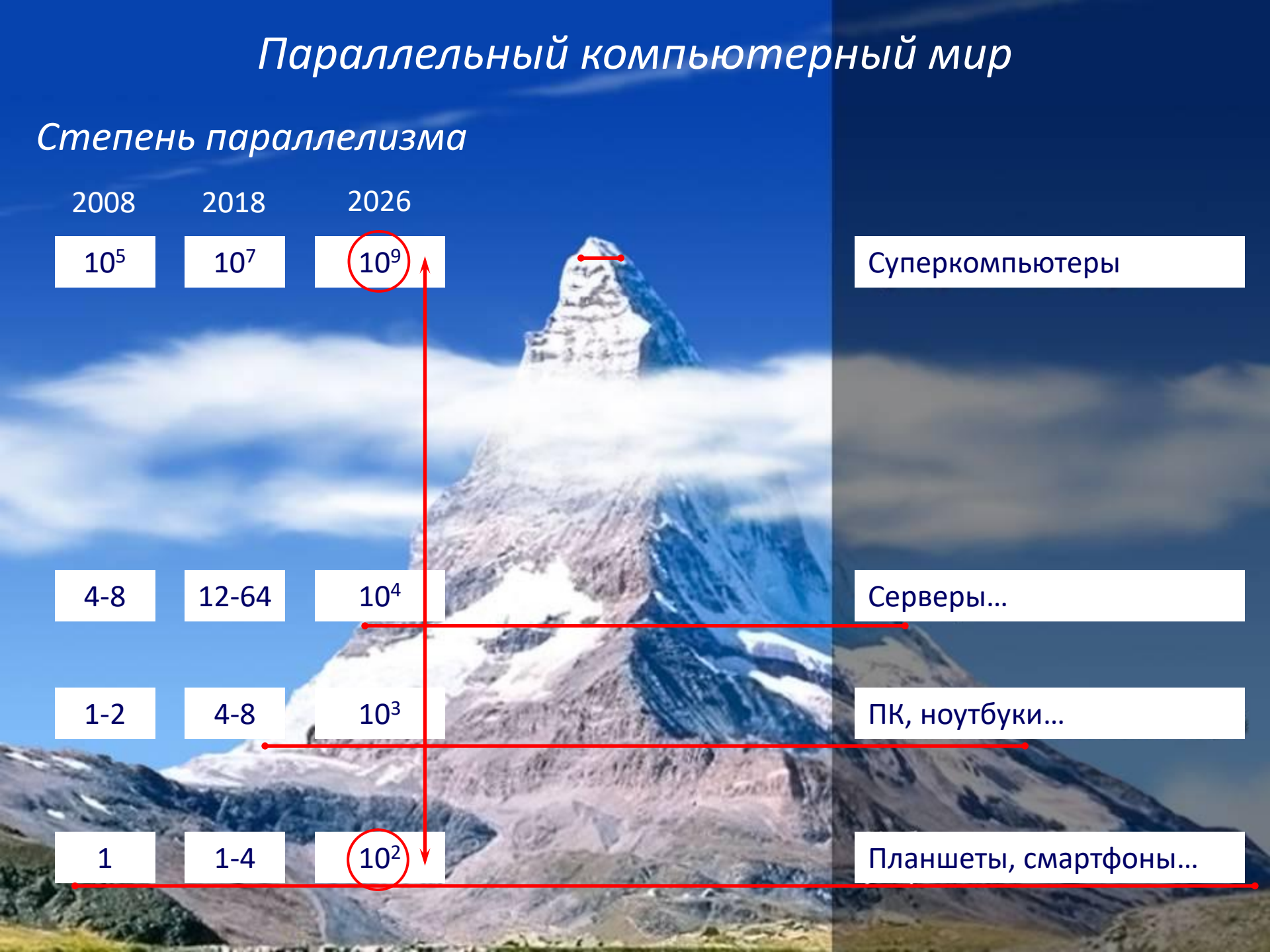
Неужели есть настолько важные задачи, которые оправдывают крайне высокую стоимость суперкомпьютеров ?

PARALLEL.RU

Параллельный компьютерный мир

Степень параллелизма

2008	2018	2026	
10^5	10^7	10^9	Суперкомпьютеры
4-8	12-64	10^4	Серверы…
1-2	4-8	10^3	ПК, ноутбуки…
1	1-4	10^2	Планшеты, смартфоны…

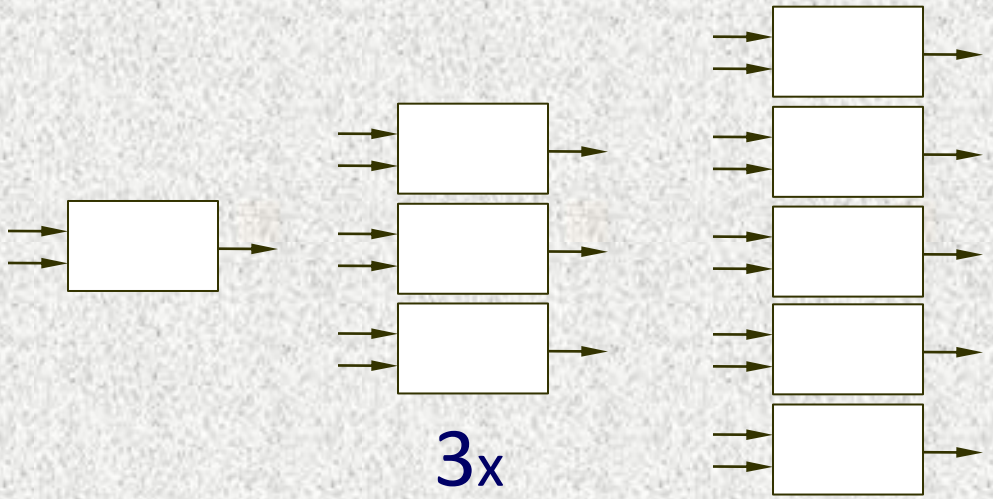


Два вида параллельной обработки данных

- *Параллелизм*
- *Конвейерность*

Параллелизм в архитектуре компьютеров

- *Параллелизм*



- *Конвейерность*

Длина конвейера = число ступеней конвейера.

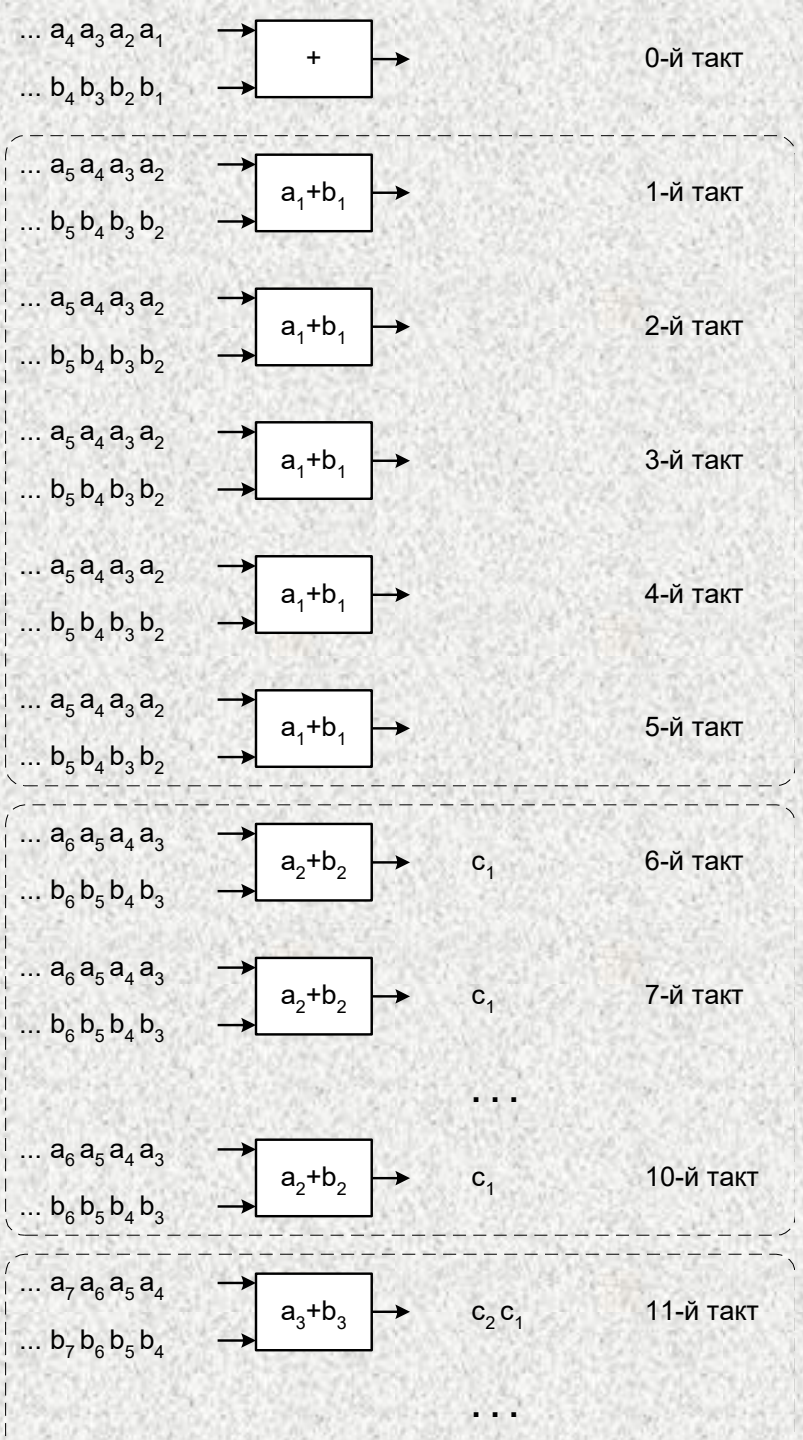
Последовательная обработка данных

Суммирование векторов C = A + B с помощью 1 последовательного устройства.

Устройство выполняет 1 операцию за 5 тактов.

Векторы A и B содержат по 100 элементов.

Время выполнения данной операции: 500 тактов.



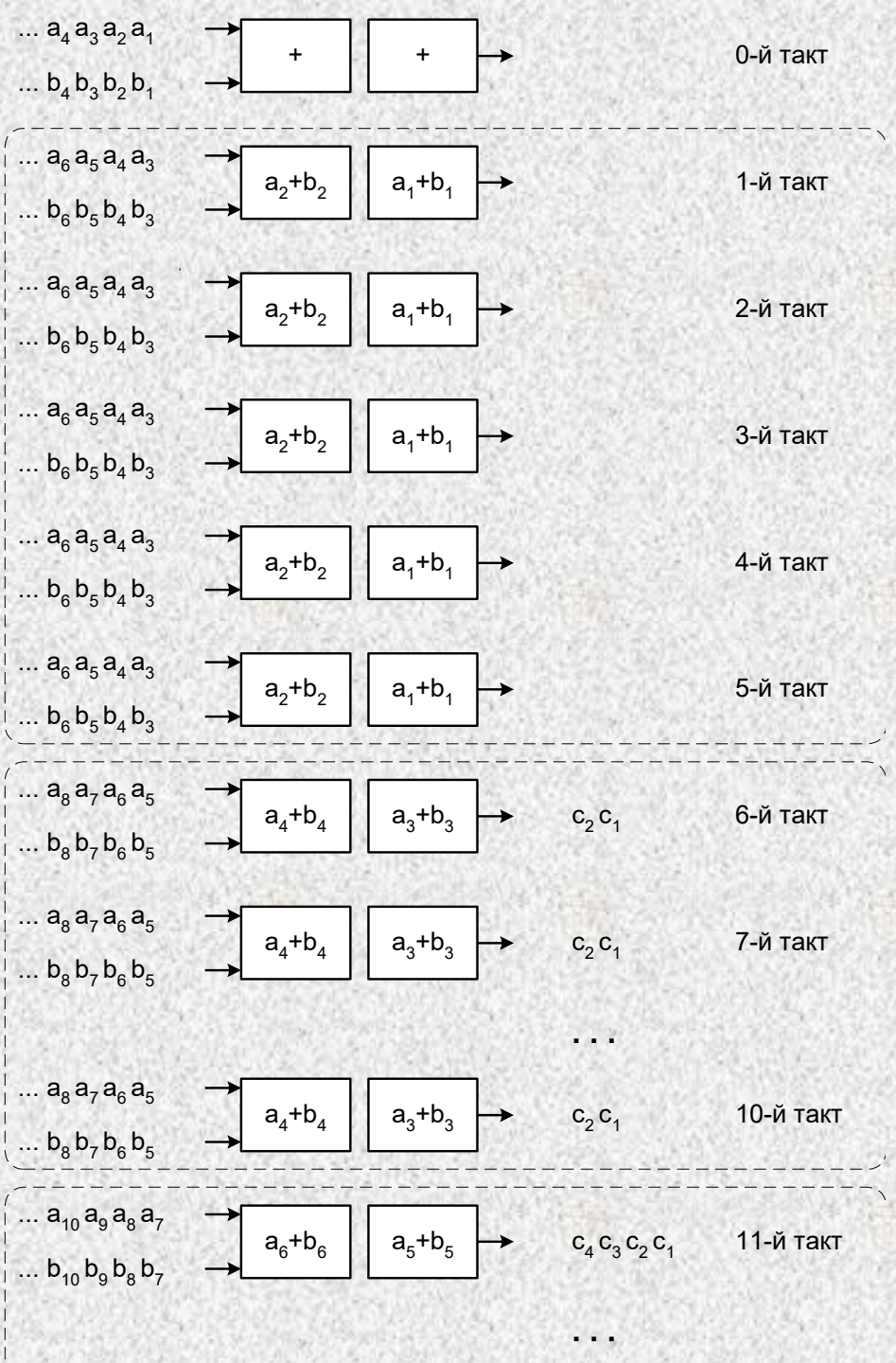
Параллельная обработка данных

Суммирование векторов C = A + B с помощью 2 одинаковых последовательных устройств.

Каждое устройство выполняет по 1 операции за 5 тактов.

Векторы A и B содержат по 100 элементов.

Время выполнения данной операции: 250 тактов.



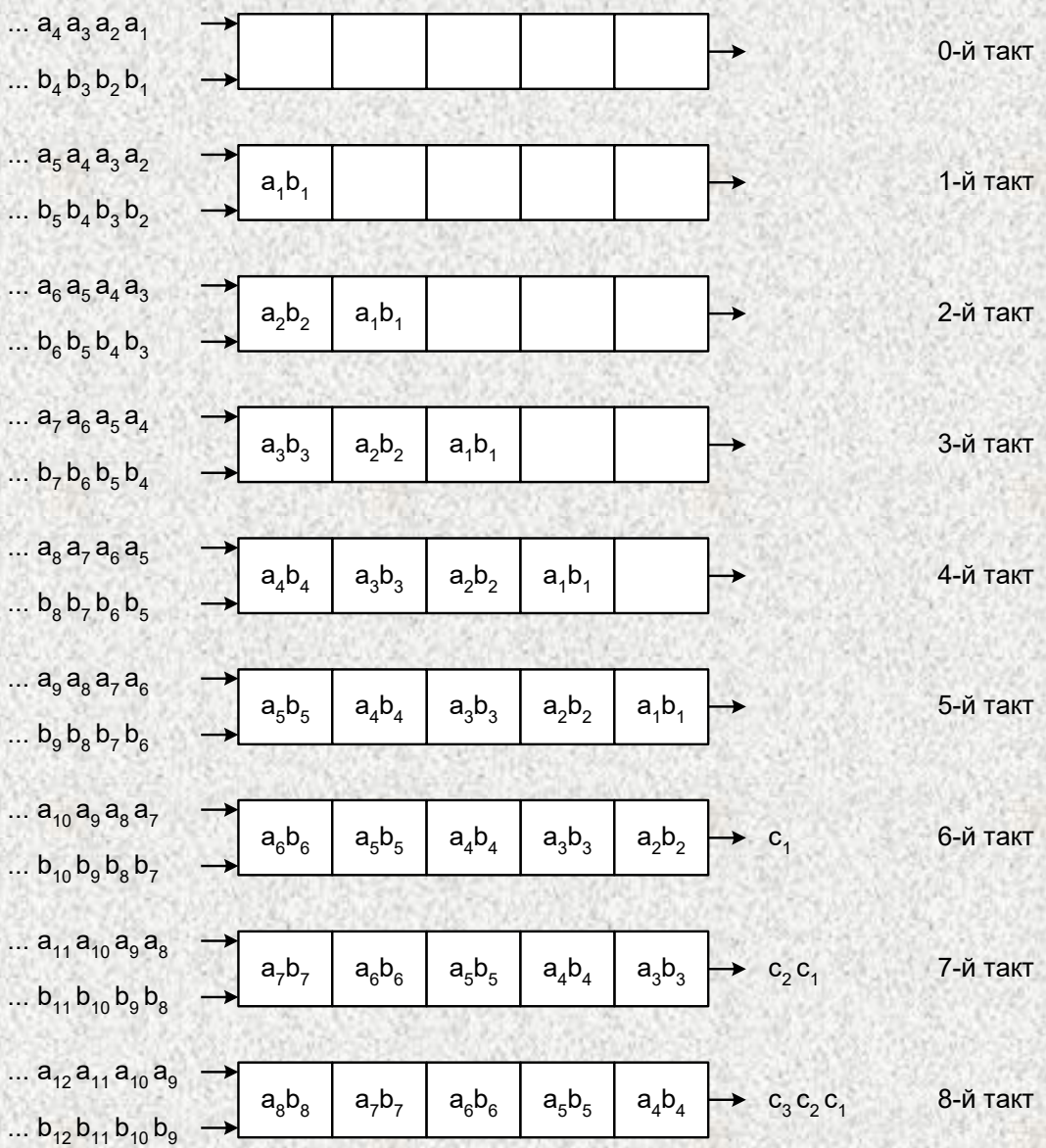
Конвейерная обработка данных

Суммирование векторов
C = A + B с помощью конвейерного устройства.

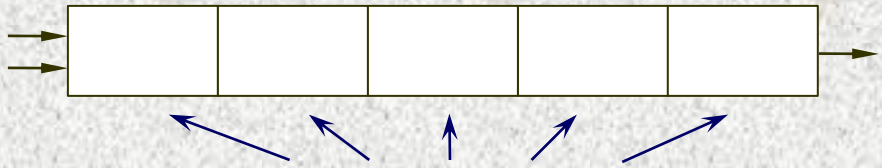
Конвейерное устройство состоит из 5 ступеней.
Каждая из 5 ступеней срабатывает за 1 такт.

Векторы A и B содержат по 100 элементов.

Время выполнения данной операции: 104 такта.



Конвейерная обработка данных
(основные понятия)



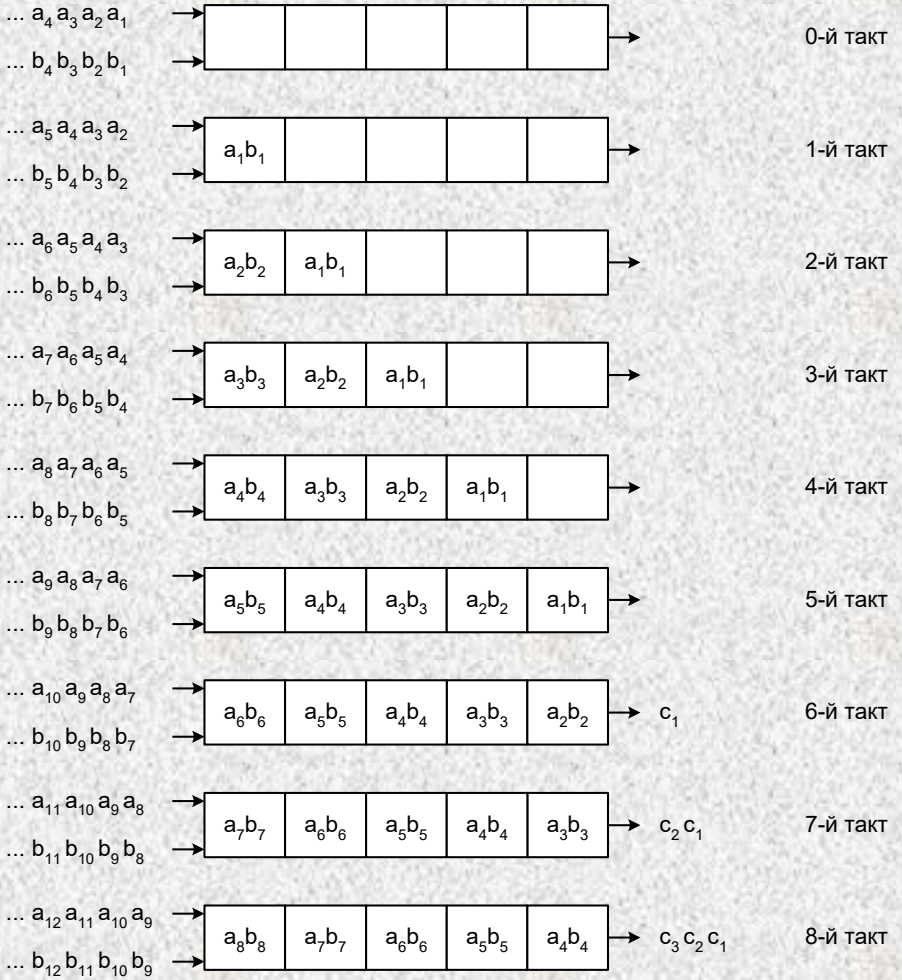
L - длина конвейера (число ступеней конвейера).
Каждая ступень срабатывает за 1 такт.
Темп выдачи результатов – каждый такт.

Время обработки всего набора из N входных данных:

$$T = L + (N-1)$$

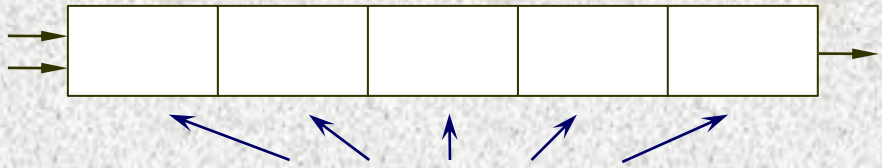
Время (число тактов) для заполнения всего конвейера (одновременно будет обработан первый входной аргумент).

Время (число тактов) для обработки оставшегося набора из N-1 входных аргументов.



Конвейерная обработка данных

(вопрос…)

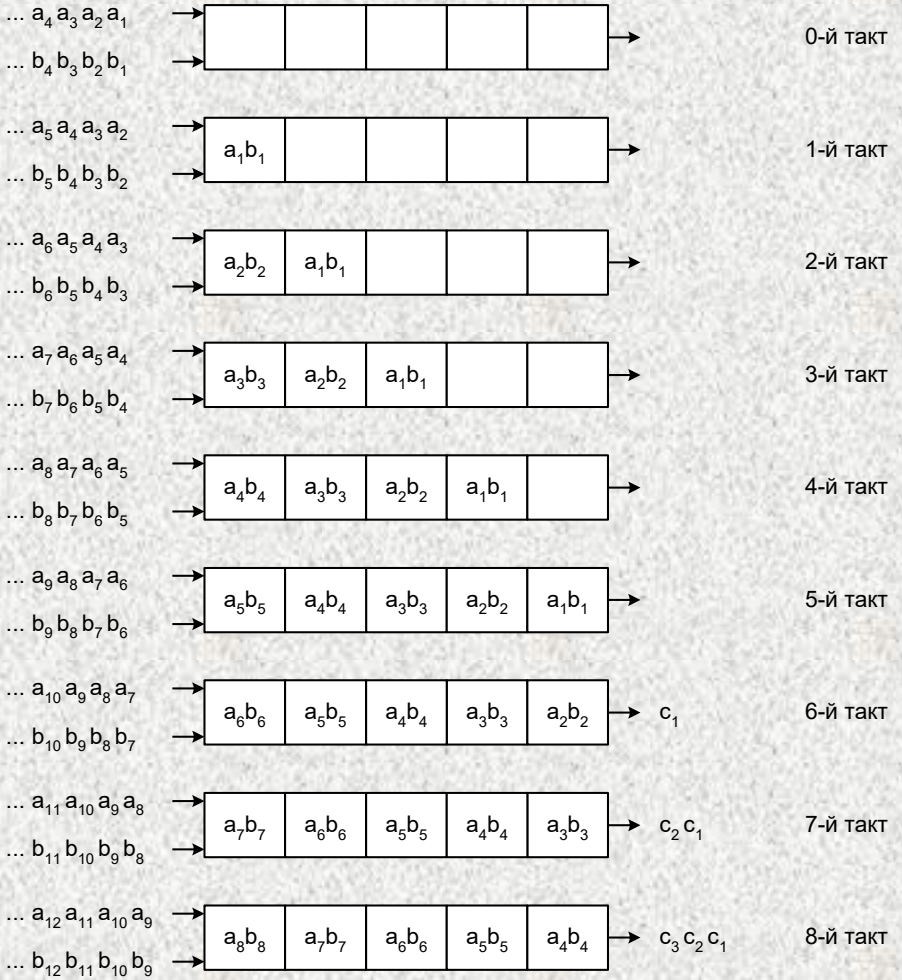


L - длина конвейера (число ступеней конвейера).

Каждая ступень срабатывает за T_i тактов.

С какой частотой будут выдаваться результаты в устоявшемся режиме?

$$\max_i(T_i)$$



Конвейерная обработка данных

(зацепление устройств – это часто на практике!)

L_1 - длина конвейера сложения.
L_2 - длина конвейера умножения.
Каждая ступень срабатывает за 1 такт.

Нужно выполнить операцию

$$A_i = B_i + C_i * d,\ i=1..N$$

Зацепление функциональных устройств – это такой режим обработки, при котором выход одного функционального устройства подаётся на вход другого.

- Время обработки в режиме без зацепления: $T = L_2 + (N-1) + L_1 + (N-1)$
- Время обработки в режиме с зацеплением: $T = L_2 + L_1 + (N-1)$

Иерархия памяти

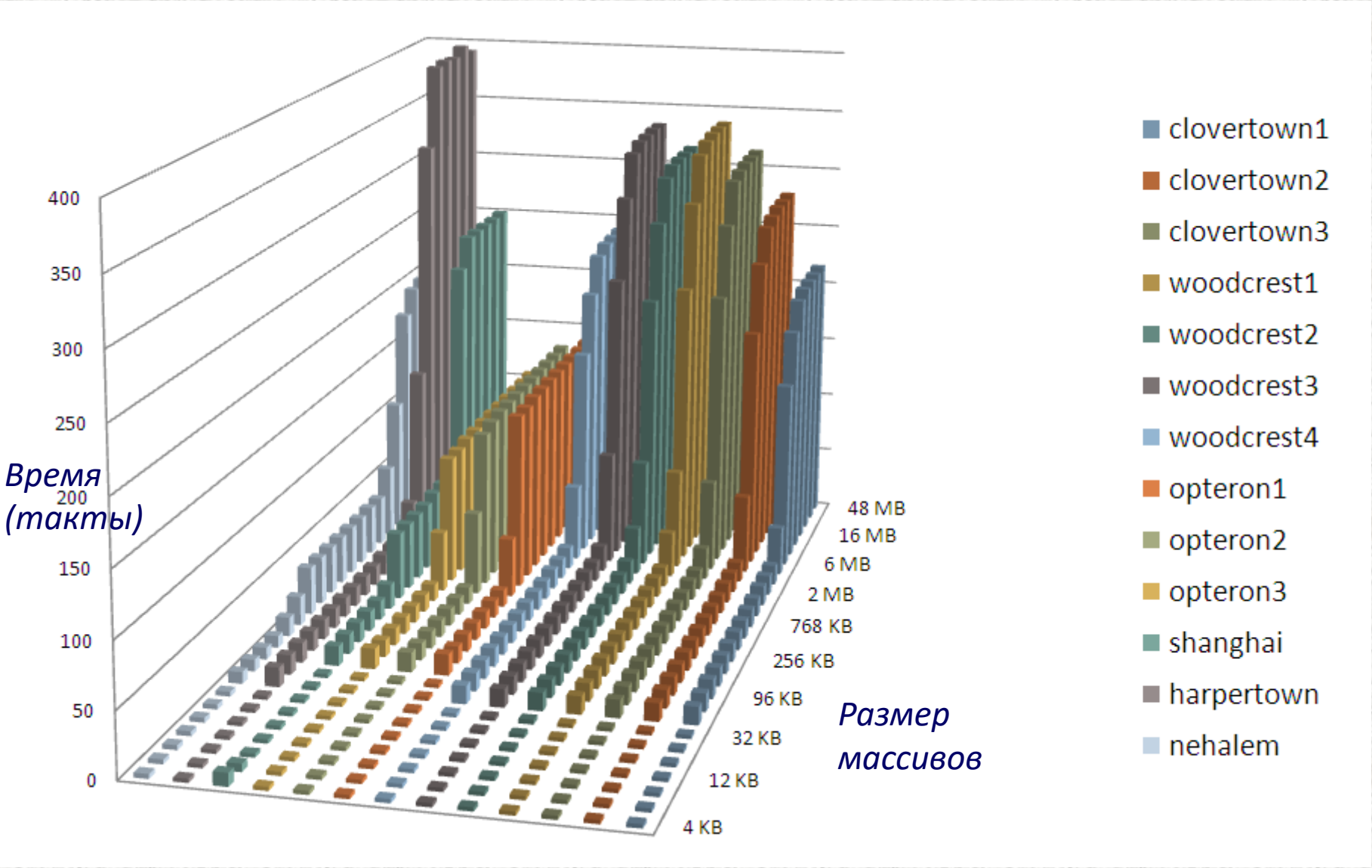
- регистры
- кэш-память 1-го уровня
- кэш-память 2-го уровня
- оперативная память
- дисковые устройства
- ленточные накопители
- …

Очень важные характеристики программ:

- *Локальность вычислений*
- *Локальность использования данных*

PARALLEL.RU

Время чтения для различных уровней иерархии памяти



Свойства работы программ с памятью

Ключевое понятие – локальность: насколько близко расположены данные/команды друг относительно друга.

- *Локальность данных,*
- *Локальность команд.*

- *Пространственная локальность данных. Отражает среднее расстояние по памяти между запрашиваемыми данными.*

- *Временн**А**я локальность данных. Показывает среднюю частоту обращений по одному адресу в память за время исполнения программы.*

Суперкомпьютеры –
это колоссальная мощь.
Но есть ли здесь проблемы ?

Увы, да, эффективная организация параллельных вычислений –
это очень не просто…

Производительность компьютера и время решения задачи

Вскапываем огород, 10×10 м – все хорошо: если нужно ускориться, то зовем 5, 10, 50 помощников…

Копаем яму, 1×1×1 м – как ускорить процесс? Еще 10 или 50 помощников сильно время не уменьшат…

Копаем фундамент: разметку может делать только бригадир, остальное – рабочие.

- *1 час разметка, остальное - 10 часов (1 рабочий) = 11 часов*
- *1 час разметка, остальное - 1 час (10 рабочих) = 2 часа*
- *1 час разметка, остальное - 6 мин. (100 рабочих) = 1 час 6 мин.*

…бригадир становится “узким местом”.

Характеристики параллельных программ

(основные обозначения)

f – *доля последовательных операций* ($0 \le f \le 1$),

p – *число процессоров (ядер, вычислительных узлов),*

T_1 – *время работы программы на одном процессоре,*

T_p – *время работы программы на системе из* p *процессоров,*

$S = \frac{T_1}{T_p}$ – *это ускорение работы программы при переходе с одного процессора на систему из* p *процессоров.*

Закон Амдала

(формулируется просто, но влияет сильно)

p – число процессоров (ядер, вычислительных узлов),

T_1 – время работы программы на одном процессоре,

T_p – время работы программы на системе из p процессоров,

$S = \frac{T_1}{T_p}$ – это ускорение работы программы при переходе с одного процессора на систему из p процессоров (во сколько раз программа начинает работать быстрее),

f – доля последовательных операций в исходной программе ($0 \le f \le 1$).

Закон Амдала.

Ускорение работы программы при переходе с одного процессора на систему из p процессоров можно оценить следующим образом:

$$S \le \frac{1}{f + (1 - f) / p}$$

На практике: $S < p$; идеальный случай: $S = p$ – *линейное ускорение работы программы*.

Закон Амдала. Следствие

$$S \approx \frac{1}{f}$$ *(при большом числе процессоров)*

На практике. Если доля последовательных операций в некоторой программе равна 0.1, значит вне зависимости от числа используемых процессоров ускорение не превысит 10.

Закон Амдала. Следствие

В теории. Для того чтобы ускорить программу в q раз, необходимо ускорить не менее, чем в q раз не менее, чем (1-1/q)-ю часть программы.

На практике. Нужно ускорить работу программы в 100 раз. Значит необходимо ускорить не менее, чем в 100 раз не менее, чем 99% этой программы.

Производительность компьютера

(теория и практика)

• *Пиковая производительность компьютера,* R_{peak}

Определяется по архитектуре компьютера в предположении, что все устройства работают с полной загрузкой, без остановок, необходимые аргументы всегда готовы, а передача данных выполняется мгновенно. Это теоретически максимальная производительность компьютера.

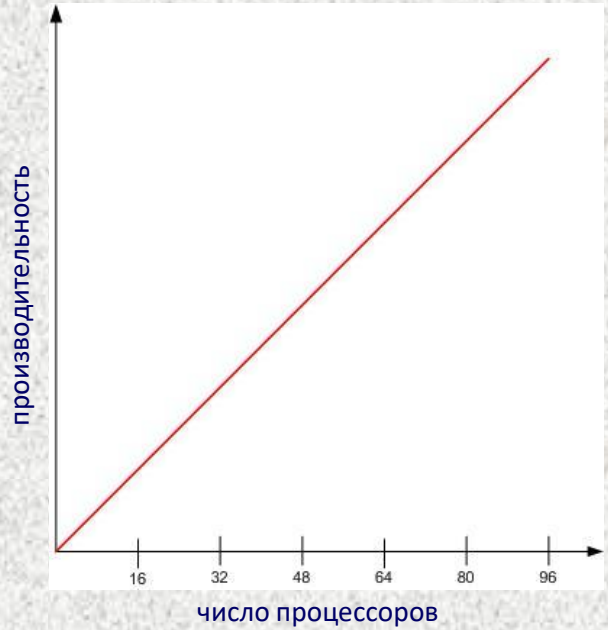
• *Реальная производительность компьютера,* R_{max}

Индивидуальна для каждой программы и/или каждого компьютера. Определяется по параметрам работы программы на компьютере:

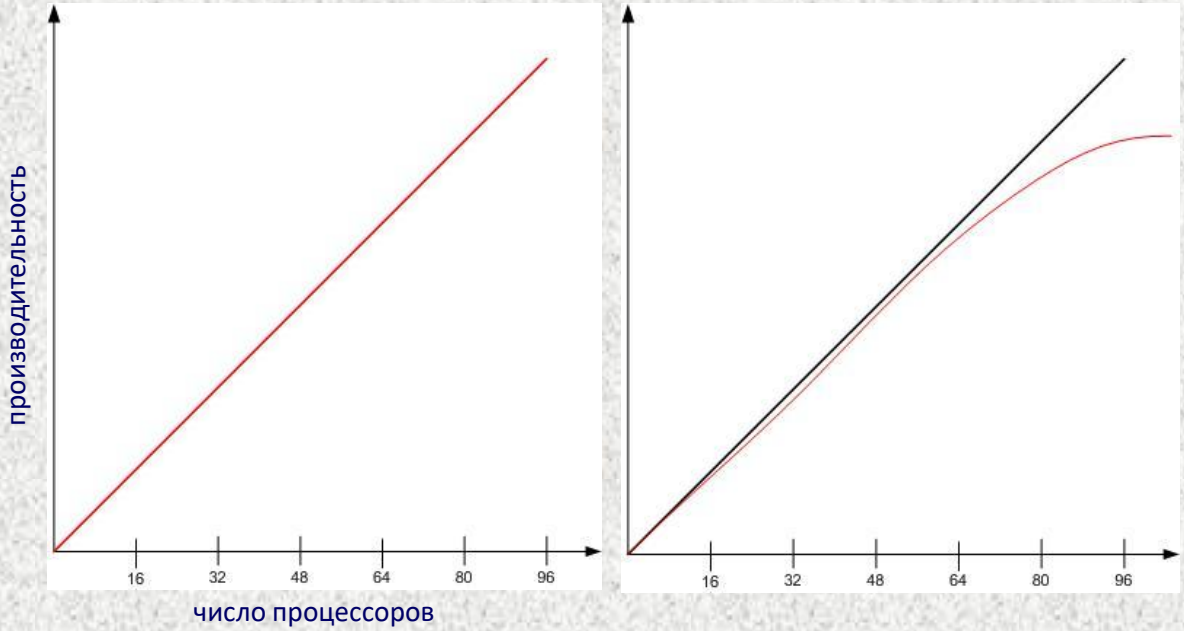
$$R_{max} = \frac{\text{число операций в программе}}{\text{время выполнения программы на компьютере}}$$

Пиковая производительность недостижима на практике $\Rightarrow$ $R_{max} < R_{peak}$

Рост производительности параллельных вычислительных систем
(в теории)

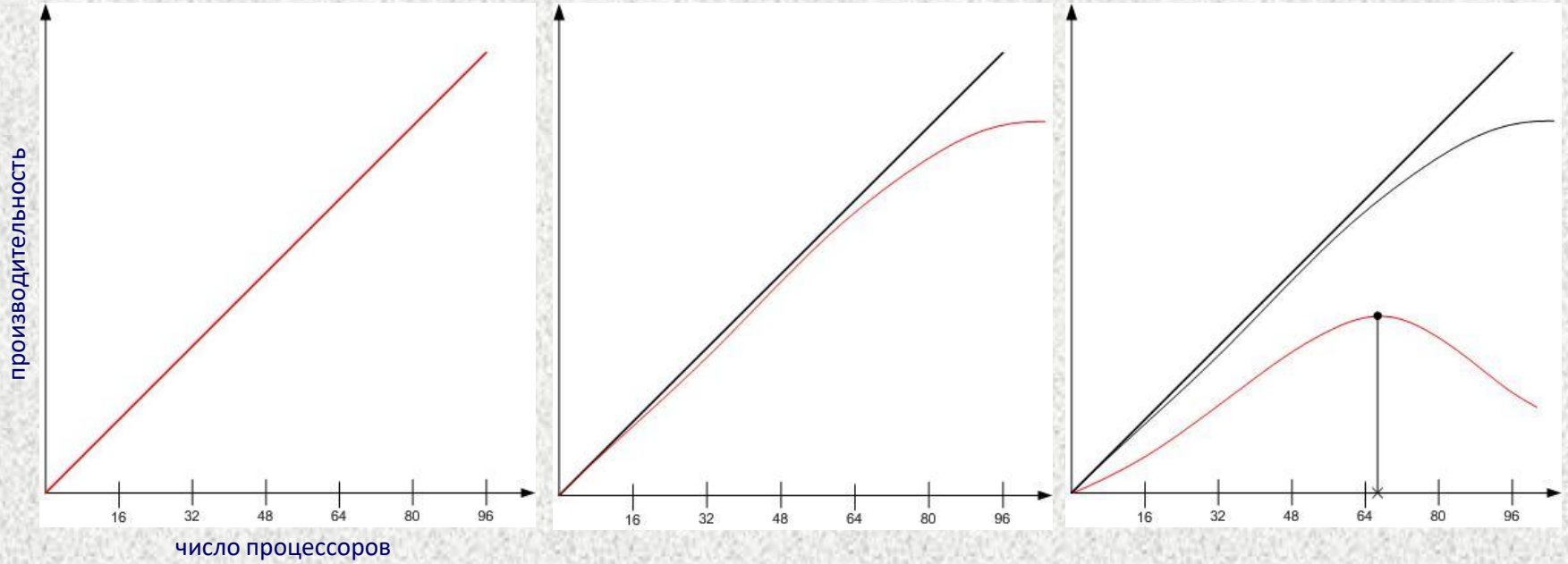


Рост производительности параллельных вычислительных систем
(в теории и на практике)



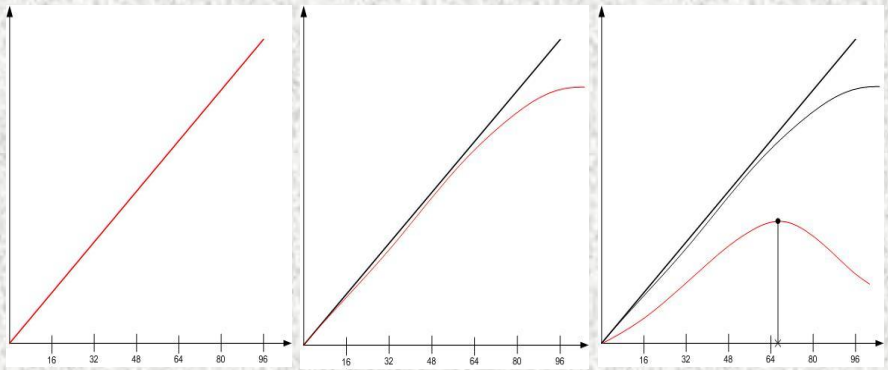
Рост производительности параллельных вычислительных систем

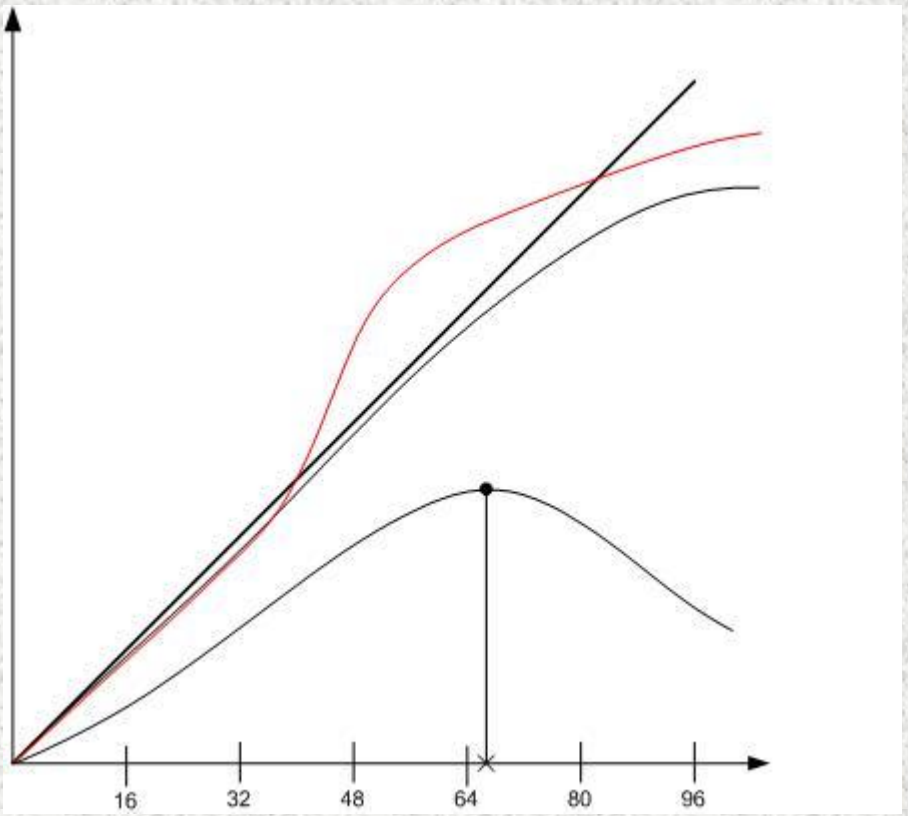
(в теории и на практике)



Рост производительности параллельных вычислительных систем

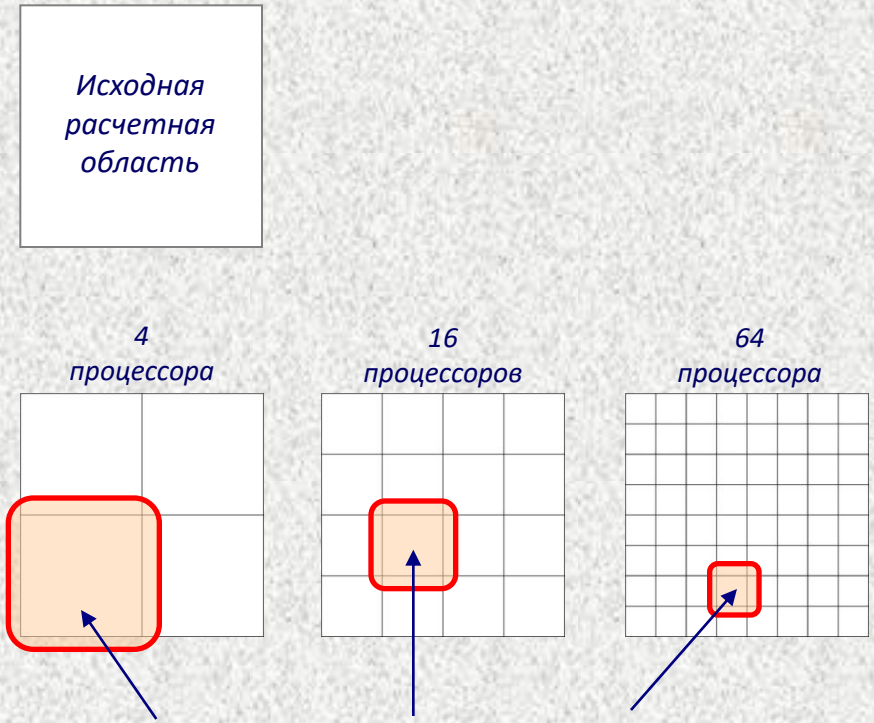
(в теории и на практике)



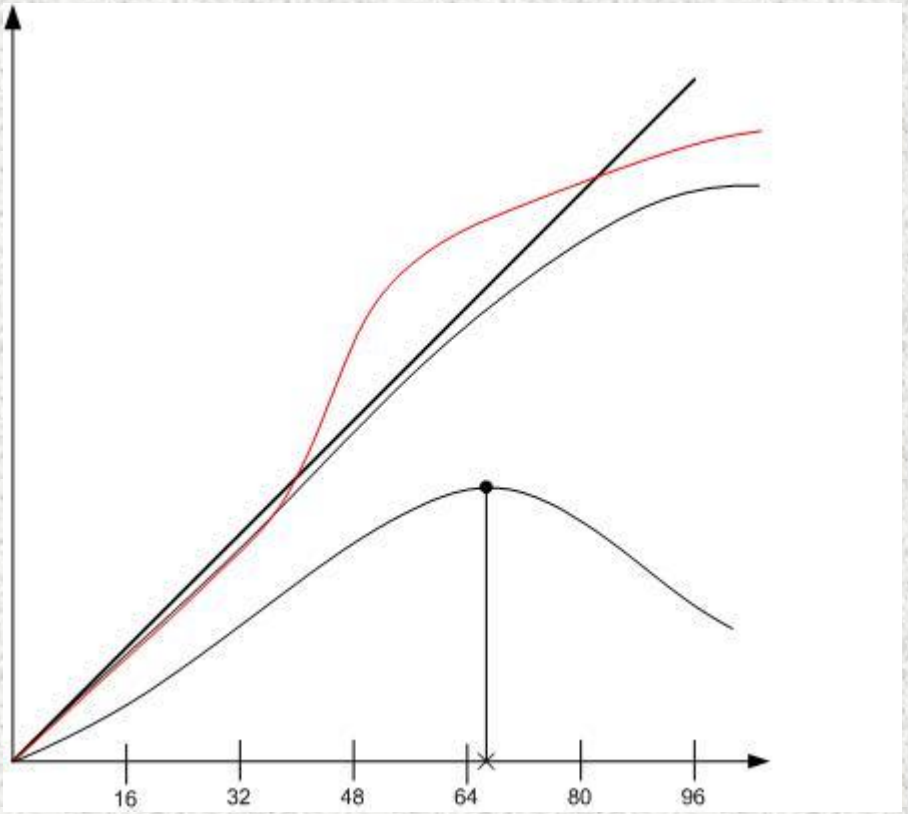


Рост производительности параллельных вычислительных систем

(в теории и на практике, суперлинейное ускорение)



Часть расчетной области, которая приходится на каждый процессор



S > p – суперлинейное ускорение работы программы, эффект возникает из-за кэш-памяти.

Эффективность работы компьютера (программы)

(практика)

- *Пиковая производительность компьютера, R_{peak}*
- *Реальная производительность компьютера, R_{max}*

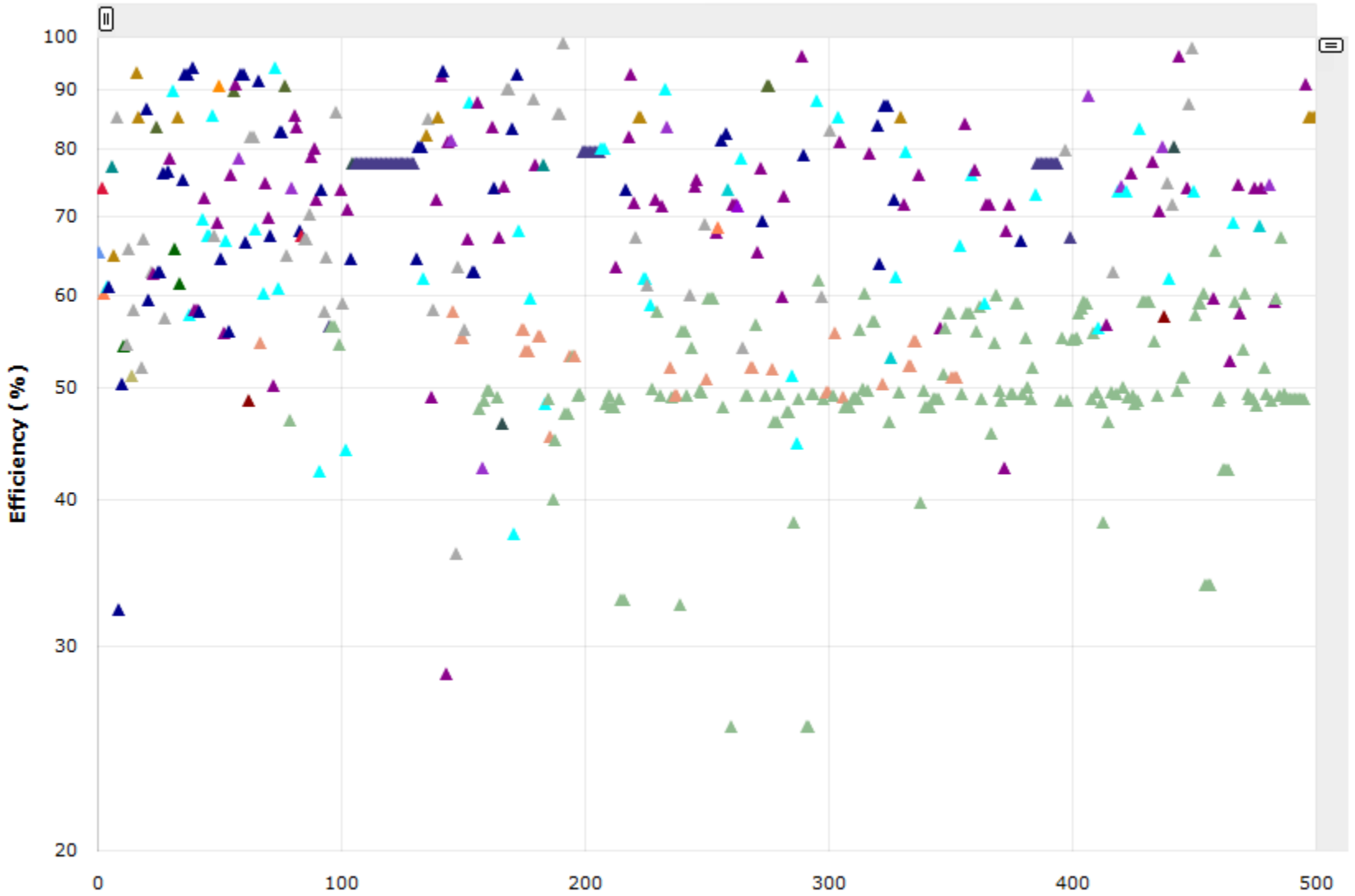
$$\textit{Эффективность} = \frac{R_{max}}{R_{peak}}$$

- эффективность реализации программы;
- эффективность работы компьютера (КПД);

Эффективность показывает, в какой степени (насколько полно) используются вычислительные возможности компьютера.

На тестах и простых примерах: $R_{max} < R_{peak}$; В реальности: $R_{max} << R_{peak}$.

Top500, Linpack, Эффективность



Параллельные вычисления: важны все детали…

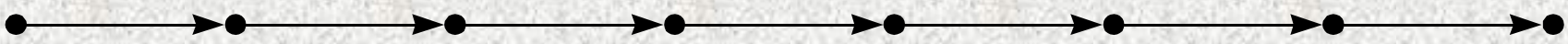
Вопрос.
Трое мужчин могут покрасить три забора за три часа.
За сколько времени покрасит один человек один забор?

Ответ:

Возможен любой промежуток времени, как сколь угодно маленький, так и сколь угодно большой…

Суммирование элементов массива.
Параллелизма нет…

```
s = 0.0;
for ( i = 0; i < n; ++i )
    s = s + A[ i ];
```

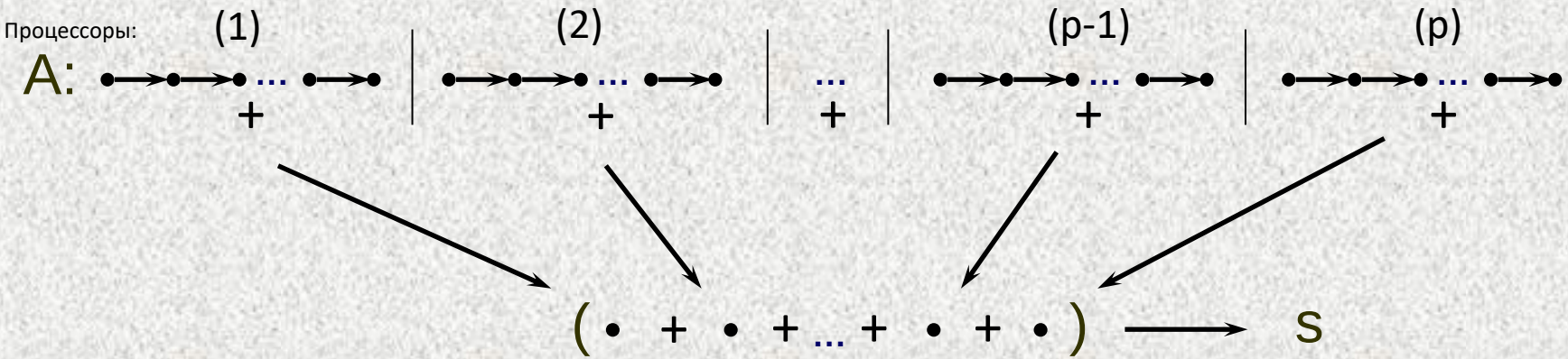


Трагедия?...

Не совсем – замена алгоритма может помочь…

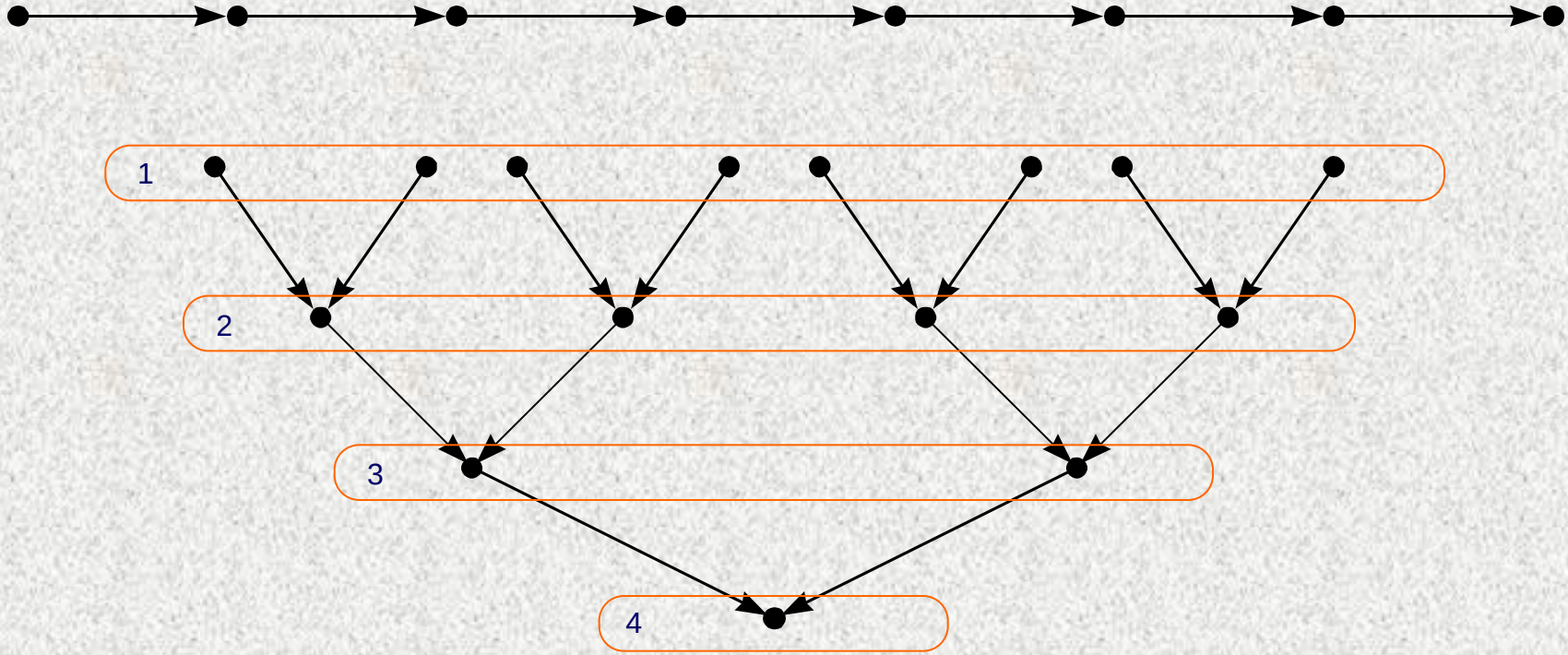
Суммирование элементов массива.
Замена алгоритма

```
s = 0.0;
for ( i = 0; i < n; ++i )
    s = s + A[ i ];
```



Суммирование элементов массива.

Замена алгоритма



Простые, но фундаментальные вопросы параллельных вычислений

Как решить задачу быстрее?

Параллелизм, конвейерность…

Как решить задачу намного быстрее?

Как организовать работу нескольких человек/ядер/процессоров…?

эффективную

миллионов

Теория / Практика…

Компьютеры с различной архитектурой…

Синхронность / Асинх-ть / Модели…

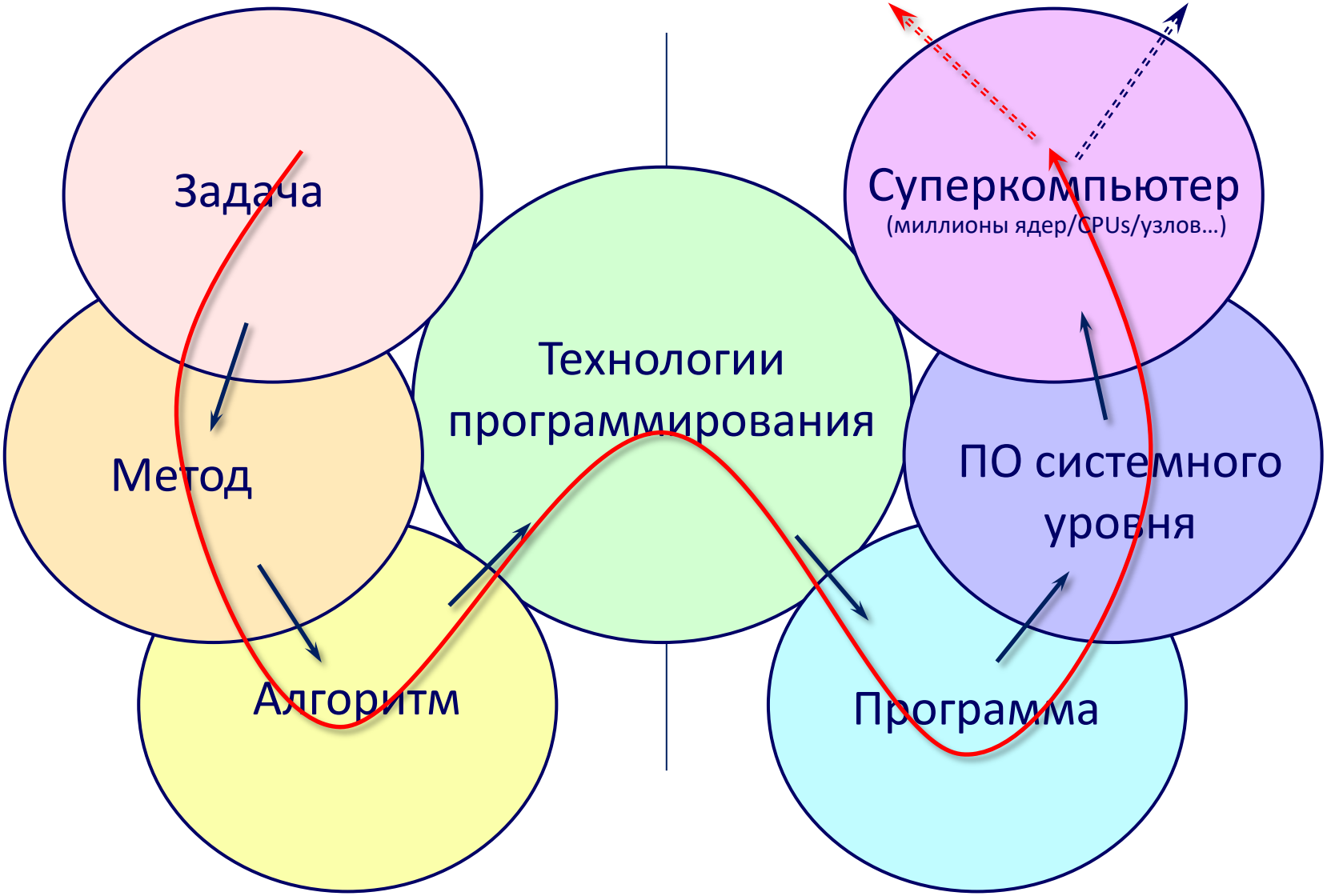
Каковы основные этапы в решении задач на параллельных вычислительных системах ?

Решение задач и вычислительные системы

(Реальность)
Реальная
производительность

Огромный
разрыв !

(Ожидания)
Пиковая
производительность



Алгоритмическая сторона

Компьютерная сторона

Особенность современных компьютеров – это высокая степень параллельности.

Решение задачи на параллельной вычислительной системе будет эффективным только в том случае, если на всем пути от Задачи до Компьютера нет ни одного “узкого места”!

Центральная проблема:
отображение программ и алгоритмов на архитектуру параллельных вычислительных систем (co-design, кодизайн)

Варианты сдачи зачета (экзамена) по курсу “Суперкомпьютеры и параллельная обработка данных”

Пять вариантов описаны на странице: Parallel.ru/vmk20

- *Зачёт (экзамен) в форме беседы с преподавателем.*
- *Зачёт в форме электронного тестирования.*
- *Зачёт за выполнение практического задания.*
- *Конкурс изображений "Параллелизм вокруг нас".*
- *Конкурс выступлений в формате "Научный стенд-ап".*

Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ

(часть 3)

Вл.В.Воеводин

Зав.кафедрой Суперкомпьютеров и квантовой информатики ВМК МГУ,
Директор НИВЦ МГУ,
чл.-корр. РАН, д.ф.-м.н., профессор

voevodin@parallel.ru

ВМК МГУ, 2020

Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ (лекция 4)

А.С.Антонов

Вед. н.с. НИВЦ МГУ, к.ф.-м.н.

asa@parallel.ru

ВМК МГУ, 2020

Основные показатели эффективности и масштабируемости параллельных программ

Основные показатели эффективности и масштабируемости параллельных программ

Основные обозначения:

p — число процессоров (процессорных ядер).

T_1 — время работы программы на одном процессоре.

T_p — время работы программы на p процессорах.

Основные показатели эффективности и масштабируемости параллельных программ

Ускорение (*speedup*) $S = T_1/T_p$, где T_p — время исполнения распараллеленной программы на p процессорах, T_1 — время исполнения исходной программы.

В идеальном случае (отсутствие накладных расходов на организацию параллелизма) получаем $S = p$ – *линейное ускорение*.

Суперлинейное ускорение $S > p$.

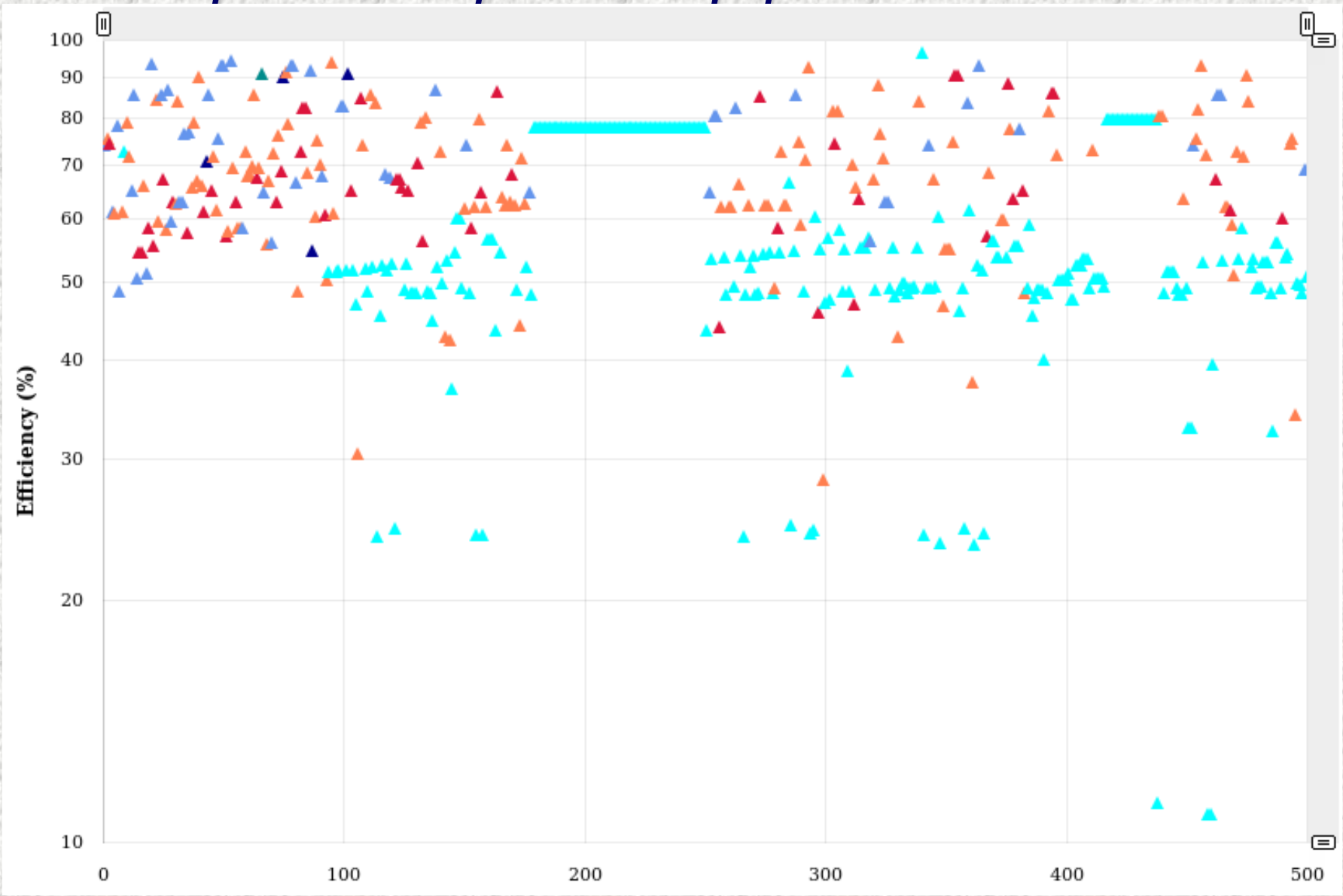
Основные показатели эффективности и масштабируемости параллельных программ

Эффективность реализации программы R_{max}/R_{peak} определяется как отношение реальной производительности R_{max} к пиковой производительности R_{peak}.

Поскольку пиковая производительность недостижима на практике, эффективность реализации программы всегда меньше единицы.

Чем ближе этот показатель к единице, тем лучше для пользователя, поскольку говорит о том, что более эффективно задействованы ресурсы компьютера.

Top500, Linpack, Эффективность



Основные показатели эффективности и масштабируемости параллельных программ

Эффективность распараллеливания E = S/p определяет среднюю долю времени выполнения параллельного алгоритма, в течение которого процессоры реально используются для решения задачи.

Оценка качества распараллеливания предполагает получение наилучших (максимальных) значений ускорения и эффективности распараллеливания.

Получение большого ускорения за счёт большого числа процессоров зачастую приводит к снижению эффективности распараллеливания.

Основные показатели эффективности и масштабируемости параллельных программ

Стоимость (cost) вычислений

$C = pT_p$.

$T_0 = pT_p - T_1$ — суммарные *накладные расходы (total overhead)*. При увеличении p значение, как правило, возрастает.

$T_p = (T_1 + T_0)/p$

$S = T_1/T_p = pT_1/(T_1 + T_0)$

Основные показатели эффективности и масштабируемости параллельных программ

$E = S/p = T_1/(T_1+T_0)=1/(1+T_0/T_1)$

Если T_1 фиксировано, то при увеличении числа процессоров p эффективность распараллеливания E, как правило, уменьшается за счёт роста накладных расходов T_0.

Если фиксировано число процессоров p, то эффективность распараллеливания E можно увеличить, увеличивая сложность решаемой задачи T_1.

Основные показатели эффективности и масштабируемости параллельных программ

Масштабируемость (*scalability*) — способность системы увеличивать свою производительность при добавлении ресурсов (обычно аппаратных).

Система называется *масштабируемой*, если она способна увеличивать производительность пропорционально дополнительным ресурсам.

Основные показатели эффективности и масштабируемости параллельных программ

Масштабируемость можно оценить через отношение прироста производительности системы к приросту используемых ей ресурсов.

Чем ближе это отношение к единице, тем масштабируемость лучше.

Основные показатели эффективности и масштабируемости параллельных программ

Масштабируемость:

- Компьютера или его компонент (например, коммуникационной сети).
- Алгоритмов безотносительно к компьютеру.
- Параллельных программ относительно данного компьютера.

Основные показатели эффективности и масштабируемости параллельных программ

Масштабируемость компьютера:

Вертикальная масштабируемость (*масштабируемость вглубь, scale up*) – возможность замены платформы, в которой функционирует система, на новую, обладающую большей производительностью.

Горизонтальная масштабируемость (*масштабируемость вширь, scale out*) – возможность увеличения производительности системы за счет добавления дополнительных программных или аппаратных средств.

Основные показатели эффективности и масштабируемости параллельных программ

Вычислительная сложность задачи

W – количество основных вычислительных шагов лучшего последовательного алгоритма, необходимых для решения задачи на одном процессоре.

W является некоторой функцией от размера входных данных.

Если для простоты предположить, что каждый основной вычислительный шаг выполняется за единицу времени, то получим $W = T_1$.

Основные показатели эффективности и масштабируемости параллельных программ

Примеры:

Для сложения N чисел:

$W = N-1$

Для скалярного произведения векторов:

$W = 2N-1$

Для перемножения матриц:

$W = N^2(2N-1)$

Основные показатели эффективности и масштабируемости параллельных программ

Масштабируемость параллельной программы определяется относительно конкретного компьютера и показывает, как изменяются динамические характеристики данной программы при использовании бóльших вычислительных ресурсов.

Основные показатели эффективности и масштабируемости параллельных программ

Масштабируемость параллельных программ:

Сильная масштабируемость (*strong scaling*) — зависимость производительности R от количества процессоров p при фиксированной вычислительной сложности задачи (W = const).

Основные показатели эффективности и масштабируемости параллельных программ

Масштабируемость параллельных программ:

Масштабируемость вширь (wide scaling) – зависимость производительности R от вычислительной сложности задачи W при фиксированном числе процессоров (p = const).

Основные показатели эффективности и масштабируемости параллельных программ

Масштабируемость параллельных программ:

Слабая масштабируемость (*weak scaling*) — зависимость производительности R от количества процессоров p при фиксированной вычислительной сложности задачи в пересчёте на один процессор (W/p = const).

Основные показатели эффективности и масштабируемости параллельных программ

Нужна метрика масштабируемости.

Для многих задач при увеличении вычислительной сложности задачи W (а, следовательно, и времени T_1) эффективность распараллеливания E растёт.

Если при одновременном увеличении числа процессоров p и вычислительной сложности задачи W эффективность распараллеливания E остаётся прежней, данную задачу на данном компьютере можно считать *масштабируемой*.

Основные показатели эффективности и масштабируемости параллельных программ

Функция изоэффективности **(***isoefficiency function***)**:

Определим, в какой степени должна увеличиваться вычислительная сложность задачи W в зависимости от числа процессоров p, чтобы эффективность распараллеливания E оставалась постоянной. Эта характеристика будет показывать масштабируемость конкретной вычислительной системы.

Чем меньше необходимая степень роста W для поддержания нужного уровня эффективности распараллеливания, тем более масштабируемой является система.

Основные показатели эффективности и масштабируемости параллельных программ

Функция изоэффективности (*isoefficiency function*):

$E = 1/(1+T_0/T_1) = 1/(1+T_0/W)$.

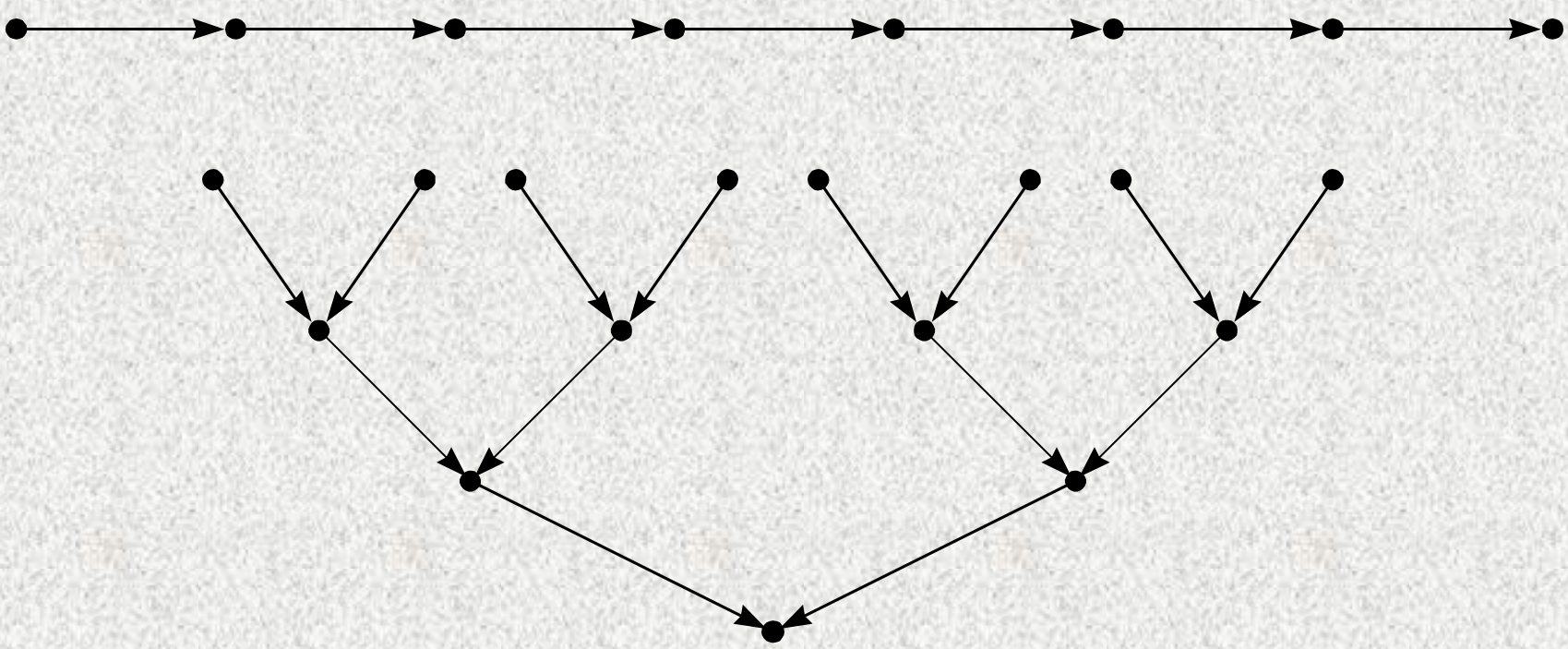
$W = E/(1-E)*T_0 = K*T_0$, где $K = E/(1-E)$

Получившаяся зависимость размера задачи, необходимой для достижения заданной эффективности распараллеливания, от числа процессоров – *функция изоэффективности (isoefficiency function)*.

Основные показатели эффективности и масштабируемости параллельных программ

Пример:

Суммирование методом сдваивания (каскадная схема).



Основные показатели эффективности и масштабируемости параллельных программ

Пример:

Пусть для сложения n чисел используется p процессоров

$W = T_1 \approx n$

$T_p \approx n/p + 2\log_2 p$

$S = T_1/T_p = n/(n/p+2\log_2 p) = p/(1+2p\log_2 p/n)$

$E = S/p = 1/(1+2p\log_2 p/n)$

Основные показатели эффективности и масштабируемости параллельных программ

Пример:

$C = pT_p = p(n/p+2\log_2 p) = n+2p\log_2 p$

$T_0 = pT_p - T_1 = 2p\log_2 p$

$W = KT_0 = 2Kp\log_2 p = \Theta(p\log_2 p)$

При увеличении числа процессоров от p до p' для поддержания постоянной эффективности распараллеливания E необходимо увеличить размер задачи в $(p'\log_2 p')/(p\log_2 p)$ раз.

Основные показатели эффективности и масштабируемости параллельных программ

Пример:

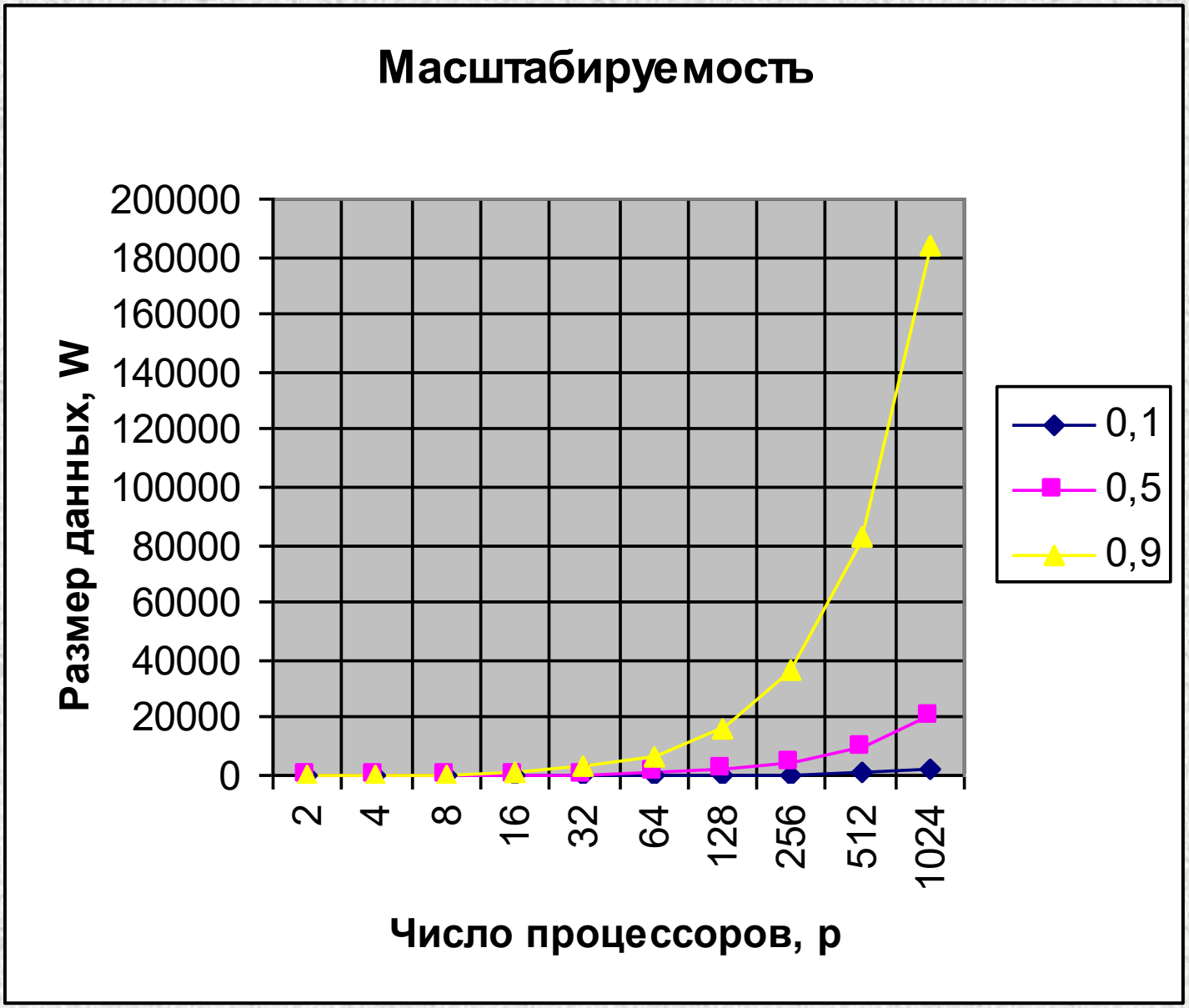
Пусть E = 0.5, тогда K = E/(1-E) = 1.

Пусть p = 16, тогда $W = 2Kp\log_2 p = 128$.

Пусть p = 64, тогда $W = 2Kp\log_2 p = 768$.

Пусть p = 1024, тогда $W = 2Kp\log_2 p = 20480$.

Основные показатели эффективности и масштабируемости параллельных программ



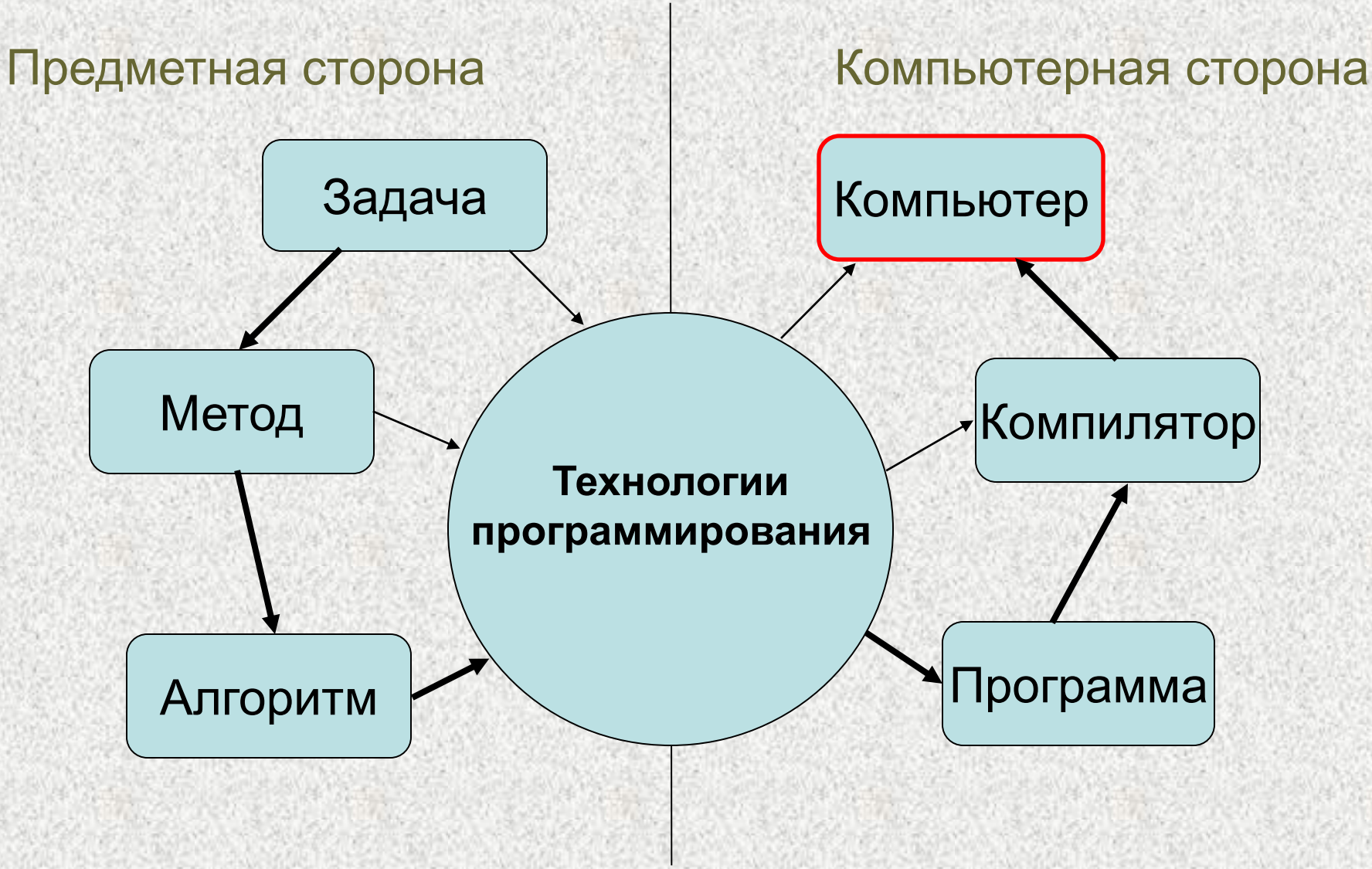
Основные показатели эффективности и масштабируемости параллельных программ

Основные помехи масштабируемости параллельных программ:

- Закон Амдала (последовательные части программы).
- Накладные расходы на коммуникации (латентность, пропускная способность).
- Неравномерность загрузки (load balancing) процессоров.
- Предел декомпозиции данных.

Архитектура параллельных вычислительных систем

Решение задачи на компьютере



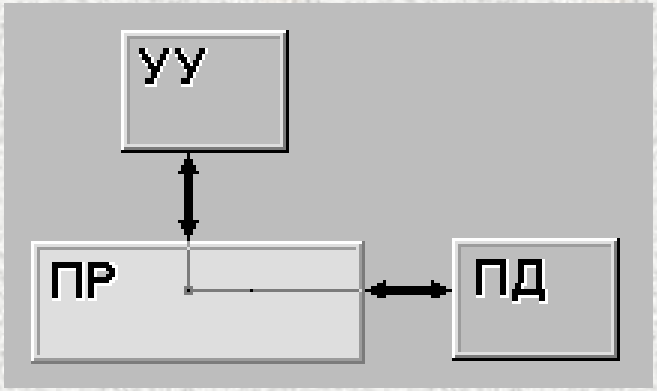
Классификация Флинна

По-видимому, самой ранней и наиболее известной является классификация архитектур вычислительных систем, предложенная в 1966 году М.Флинном.

Классификация базируется на понятии потока, под которым понимается последовательность элементов, команд или данных, обрабатываемая процессором. На основе числа потоков команд и потоков данных Флинн выделяет четыре класса архитектур: SISD,MISD,SIMD,MIMD.

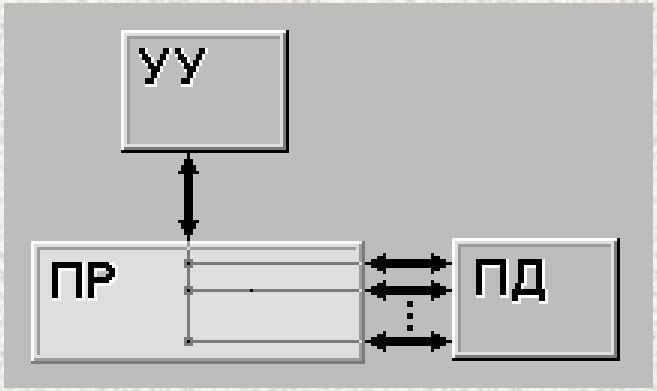
Классификация Флинна

SISD (single instruction stream / single data stream) - одиночный поток команд и одиночный поток данных. К этому классу относятся, прежде всего, классические последовательные машины, или иначе, машины фон-неймановского типа. В таких машинах есть только один поток команд, все команды обрабатываются последовательно друг за другом и каждая команда инициирует одну операцию с одним потоком данных. Для увеличения скорости обработки команд и скорости выполнения арифметических операций может применяться конвейерная обработка.



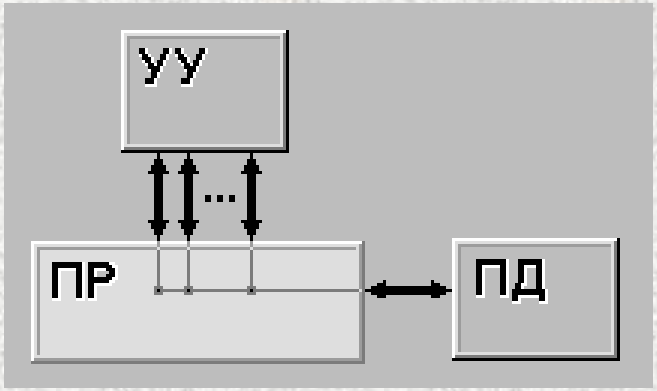
Классификация Флинна

SIMD (single instruction stream / multiple data stream) - одиночный поток команд и множественный поток данных. В архитектурах подобного рода сохраняется один поток команд, включающий, в отличие от предыдущего класса, векторные команды. Это позволяет выполнять одну арифметическую операцию сразу над многими данными - элементами вектора. Способ выполнения векторных операций не оговаривается, поэтому обработка элементов вектора может производится либо процессорной матрицей, либо с помощью конвейера.



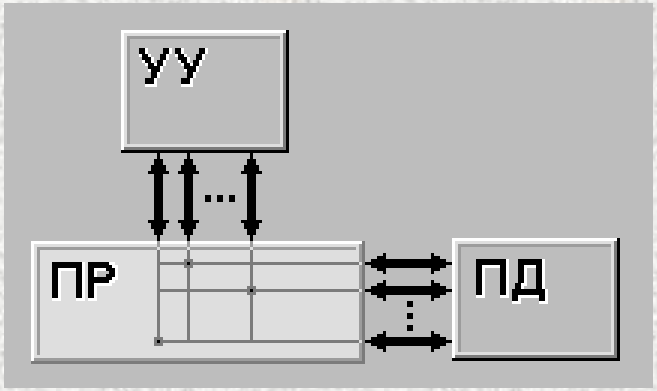
Классификация Флинна

MISD (multiple instruction stream / single data stream) - множественный поток команд и одиночный поток данных. Определение подразумевает наличие в архитектуре многих процессоров, обрабатывающих один и тот же поток данных. Данный класс пуст.

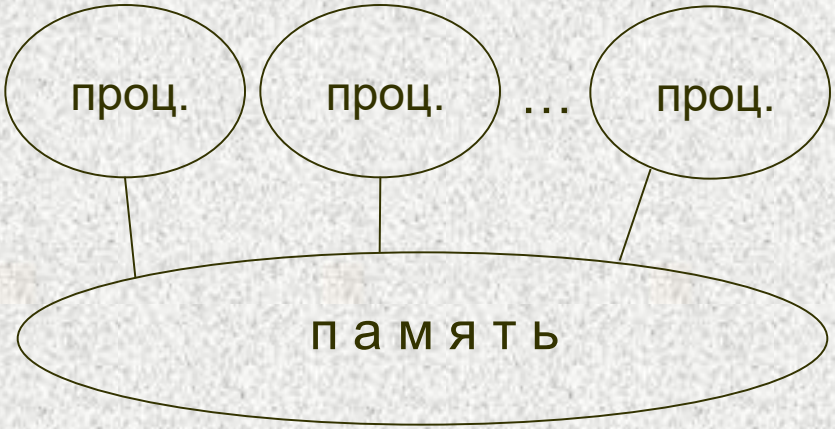


Классификация Флинна

MIMD (multiple instruction stream / multiple data stream) - множественный поток команд и множественный поток данных. Этот класс предполагает, что в вычислительной системе есть несколько устройств обработки команд, объединенных в единый комплекс и работающих каждое со своим потоком команд и данных.



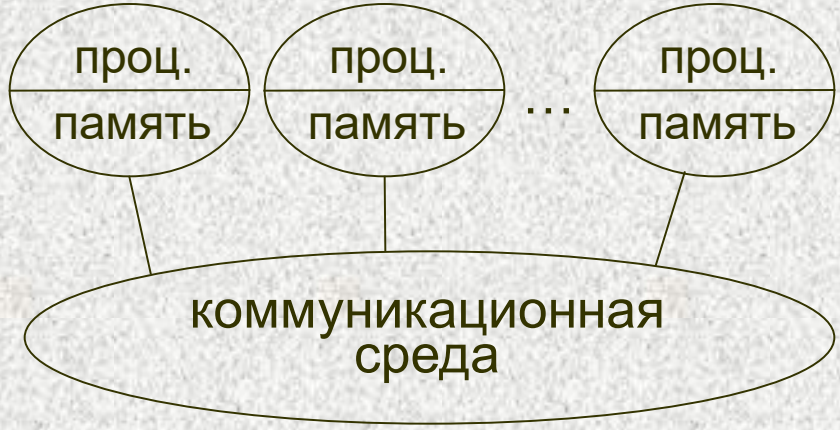
Компьютеры с общей памятью



SMP-компьютеры, два варианта расшифровки:
Shared Memory Processors
Symmetric MultiProcessor

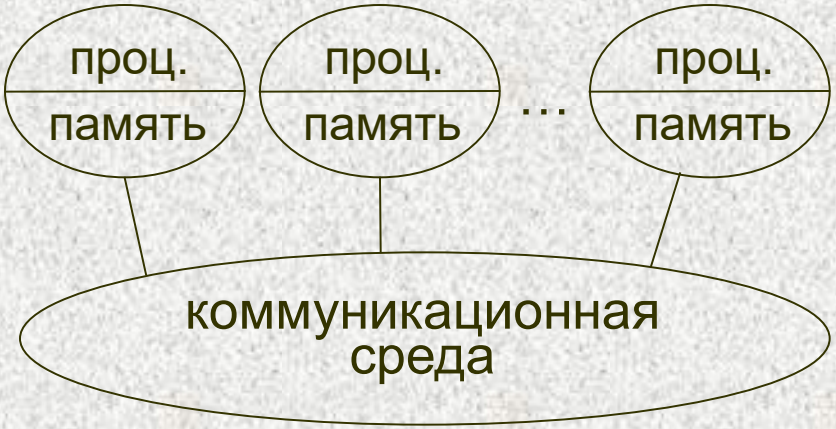
В SMP-компьютерах все, кроме процессоров, в одном экземпляре: образ операционной системы, память, подсистема ввода-вывода…

Компьютеры с распределенной памятью



Компьютеры с распределенной памятью состоят из вычислительных узлов, каждый из которых является полноценным компьютером со своей памятью, ОС, устройствами ввода-вывода и т.п., взаимодействующих друг с другом через коммуникационную среду.

Компьютеры с общей и распределенной памятью



\+ *относительная простота параллельного программирования,*
– *сложность увеличения числа процессоров (роста производительности)*

– *сложность параллельного программирования,*
\+ *относительная простота увеличения числа процессоров (роста производительности)*

Высокопроизводительные компьютеры и две задачи параллельных вычислений

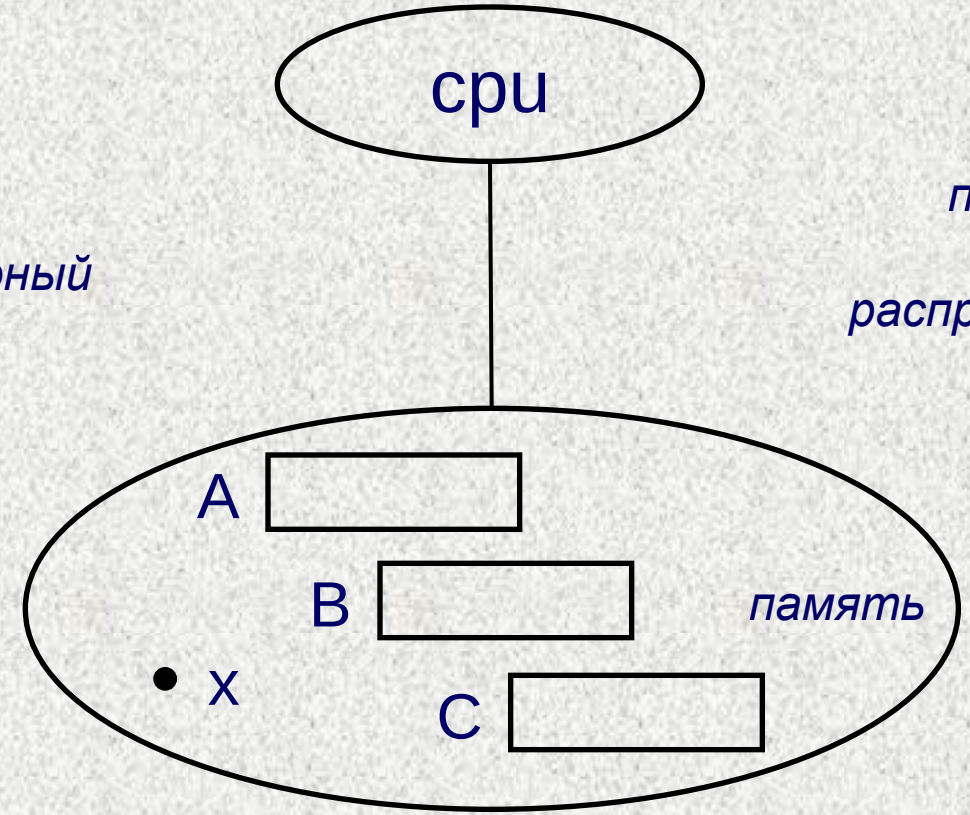
Две задачи параллельных вычислений:

- *построение вычислительных систем с максимальной производительностью:*
  - *это компьютеры с распределенной памятью*

- *эффективное программирование параллельных вычислительных систем:*
  - *это компьютеры с общей памятью.*

Почему компьютеры с распределенной памятью программировать сложнее, чем компьютеры с общей памятью?

Программирование на общей и распределенной памяти

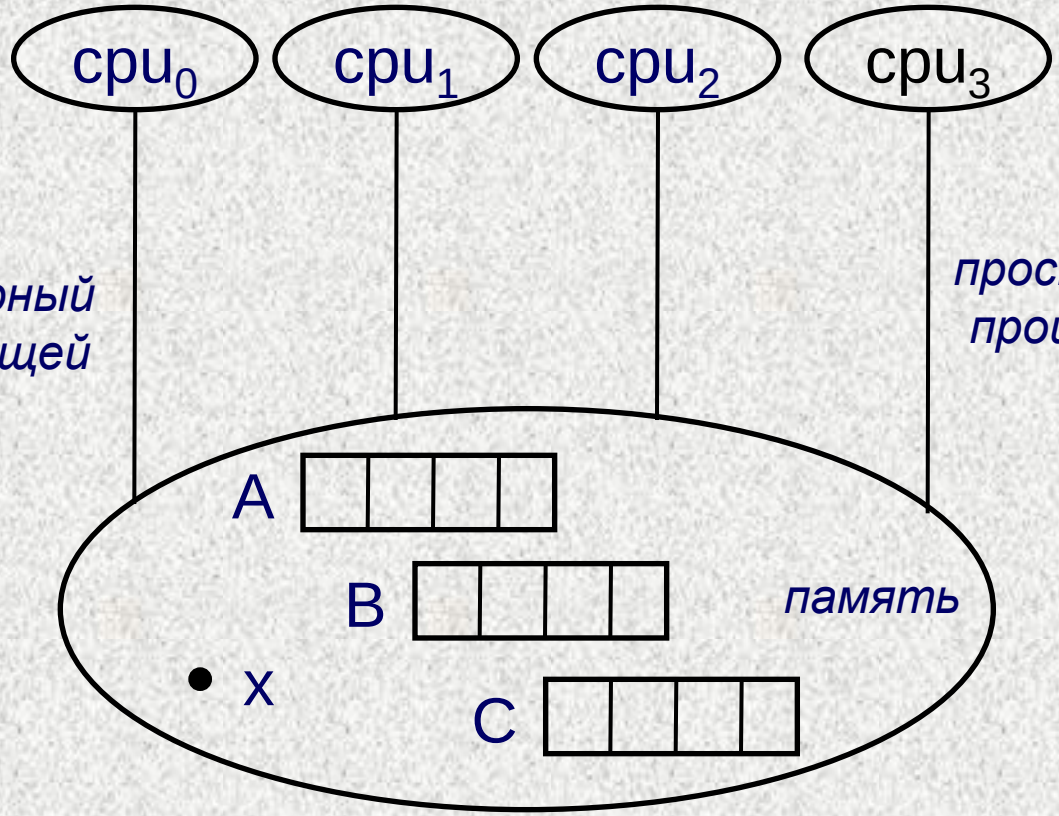


Однопроцессорный компьютер

Одно адресное пространство, один процессор, с распределением работы проблем нет.

$$A_i = B_i + C_i * x, \quad i = 1, \ldots, n$$

Программирование на общей и распределенной памяти

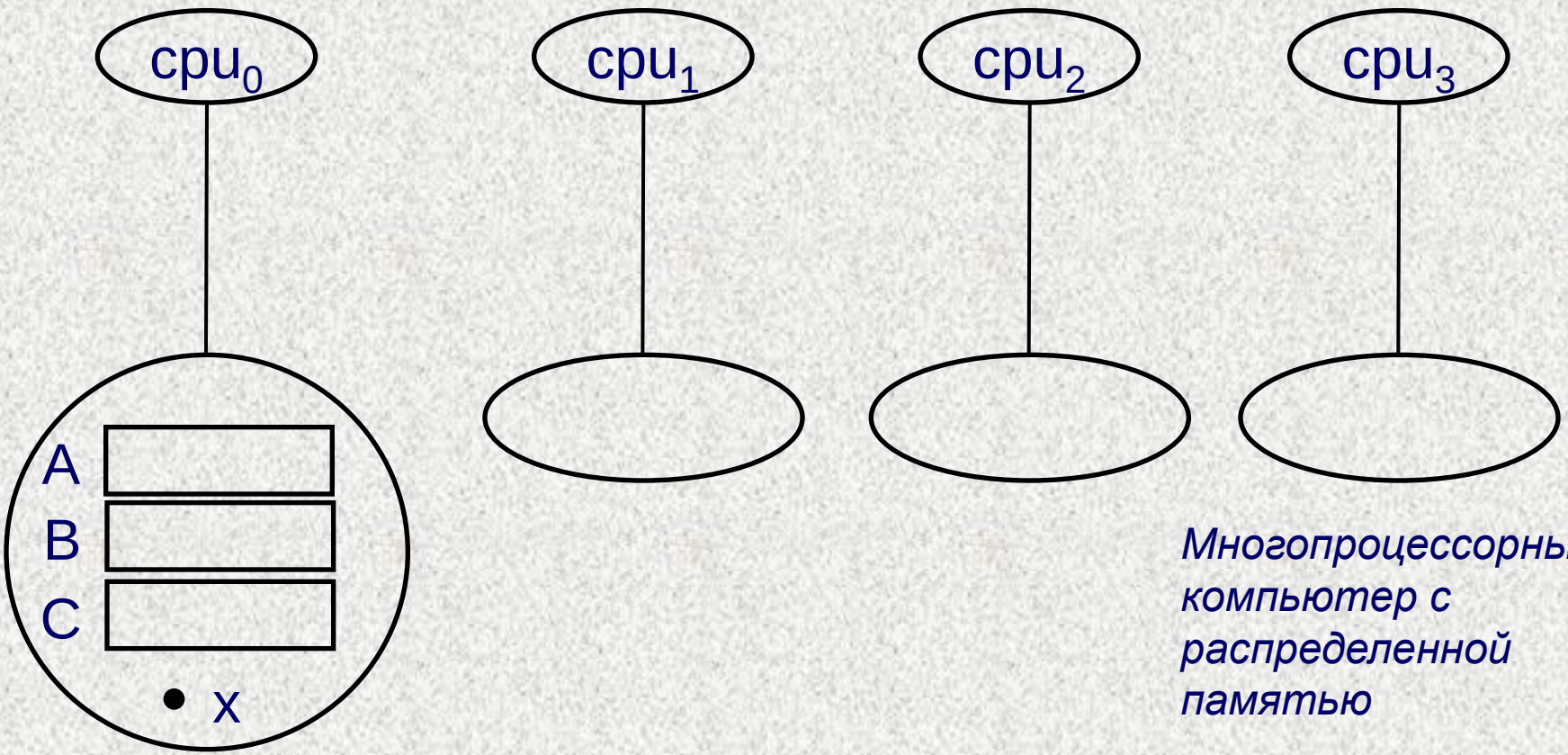


Многопроцессорный компьютер с общей памятью

Единое адресное пространство, много процессоров, каждый выполняет свою часть работы.

$$A_i = B_i + C_i * x, \quad i = 1, \ldots, n$$

Программирование на общей и распределенной памяти

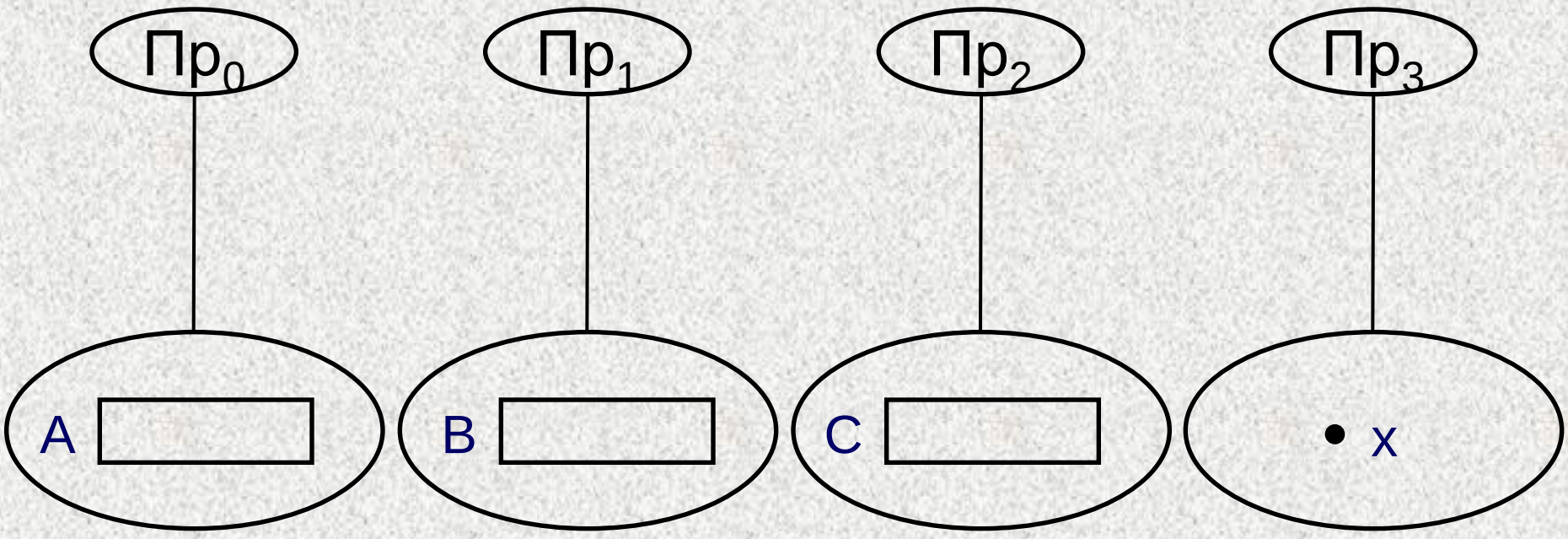


Многопроцессорный компьютер с распределенной памятью

Различные адресные пространства, плохое распределение данных, низкая локальность данных, много пересылок, низкая производительность.

$$A_i = B_i + C_i * x, \quad i = 1, \ldots, n$$

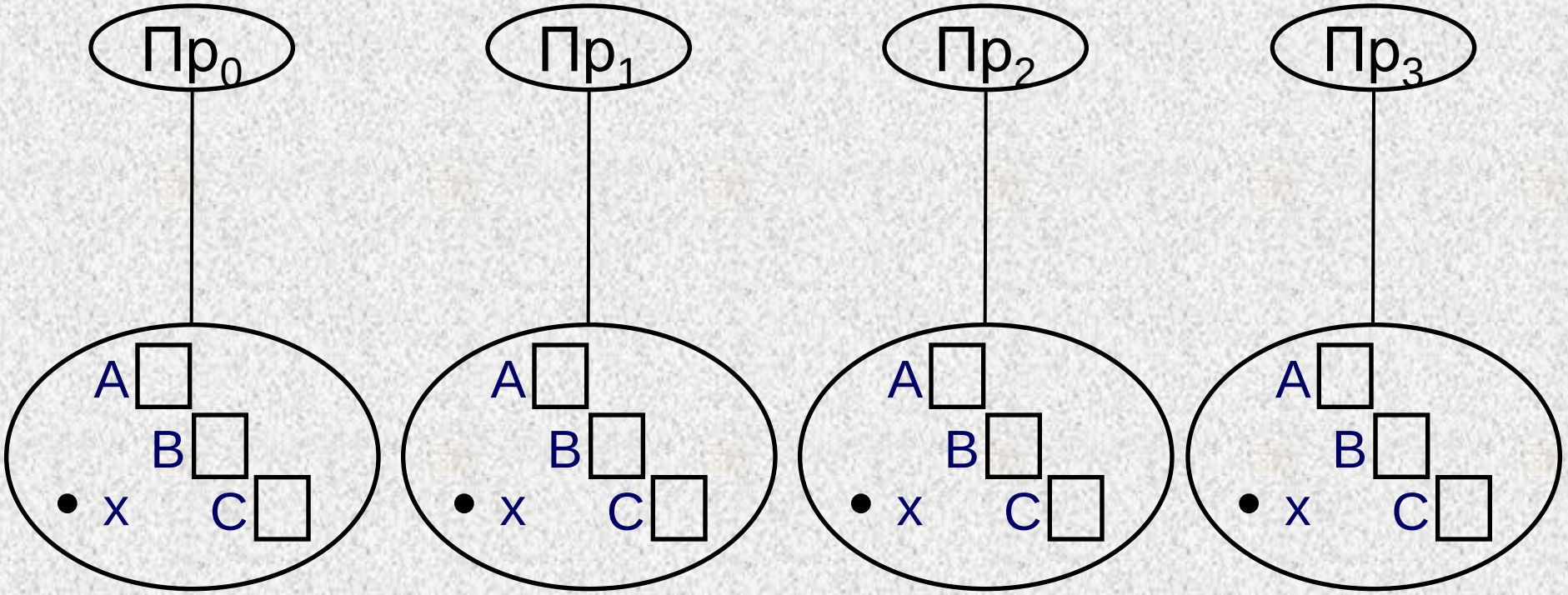
Программирование на общей и распределенной памяти



Различные адресные пространства, плохое распределение данных, низкая локальность данных, много пересылок, низкая производительность.

$$A_i = B_i + C_i * x, \quad i = 1, \ldots, n$$

Программирование на общей и распределенной памяти



Различные адресные пространства, хорошее распределение данных, высокая локальность данных, мало пересылок, высокая производительность.

$$A_i = B_i + C_i * x, \quad i = 1, \ldots, n$$

Программирование на общей и распределенной памяти

Последовательный алгоритм:

```
N=1000;
for(i=0; i<N; ++i)
   A[i]=B[i]+C[i]*x;
for(i=0; i<N; ++i)
   A[i]=(A[i]+C[i])*B[N-i-1];
```

На последовательном компьютере все данные (массивы A, B, C и скаляр x) лежат в общей памяти, различий в классах переменных нет.

Программирование на общей и распределенной памяти

Оба цикла являются параллельными, их итерации могут быть распределены между процессорами. Распределение вычислений обязательно должно быть согласовано с распределением данных!

На параллельном компьютере с общей памятью распределение данных не требуется (память общая), нужно распределить операции (итерации циклов). Возможных вариантов распределений очень много.

Программирование на общей и распределенной памяти

Первый вариант (блочное распределение итераций):

```
N=1000;
n_pr=4;
size=N/n_pr;
ibegin=proc_id*size;
iend=ibegin+size;
for(i=ibegin; i<iend; ++i)
   A[i]=B[i]+C[i]*x;
for(i=ibegin; i<iend; ++i)
   A[i]=(A[i]+C[i])*B[N-i-1];
```

Возникает понятие классов переменных. В данном случае переменные proc_id, i, ibegin, iend должны быть локальными (своя копия на каждом процессоре), а переменные A, B, C, x должны быть распределёнными (одна копия данных для всех процессоров).

Программирование на общей и распределенной памяти

Другой вариант (циклическое распределение итераций):

```
N=1000;
n_pr=4;
for(i=proc_id; i<N; i+=n_pr)
   A[i]=B[i]+C[i]*x;
for(i=proc_id; i<N; i+=n_pr)
   A[i]=(A[i]+C[i])*B[N-i-1];
```

Переменные proc_id, i должны быть локальными (своя копия на каждом процессоре), а переменные A, B, C, x должны быть распределёнными (одна копия данных для всех процессоров).

Программирование на общей и распределенной памяти

На параллельном компьютере с распределённой памятью помимо распределения операций требуется также распределение данных по локальным модулям памяти разных процессоров и организация обменов данными.

```
N=1000;
n_pr=4;
N1=N/n_pr;
<обмен 1> // рассылка частей массивов A, B, C и скаляра x
for(i=0; i<N1; ++i)
   A[i]=B[i]+C[i]*x;
<обмен 2> // обмен частями B: процессы 0<->3 и 1<->2
for(i=0; i<N1; ++i)
   A[i]=(A[i]+C[i])*B[N-i-1];
<обмен 3> // пересылка результирующего массива A
```

Программирование на общей и распределенной памяти

Для эффективного программирования компьютеров С общей памятью необходимо:

1.Найти в программе ресурс параллелизма.
2.Распределить операции по исполнительным устройствам.
3.Разграничить доступ к данным в общей памяти.

Программирование на общей и распределенной памяти

Для эффективного программирования компьютеров с распределенной памятью необходимо:

1. *Найти в программе ресурс параллелизма.*
2. *Распределить данные по модулям памяти вычислительных узлов.*
3. *Распределить операции по исполнительным устройствам.*
4. *Согласовать распределение данных с параллелизмом вычислений.*
5. *Организовать необходимые пересылки данных.*

Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ (лекция 4)

А.С.Антонов

Вед. н.с. НИВЦ МГУ, к.ф.-м.н.

asa@parallel.ru

ВМК МГУ, 2020

Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ (лекция 5)

А.С.Антонов

Вед. н.с. НИВЦ МГУ, к.ф.-м.н.

asa@parallel.ru

ВМК МГУ, 2020

Компьютеры с общей памятью

Архитектуры SMP и NUMA

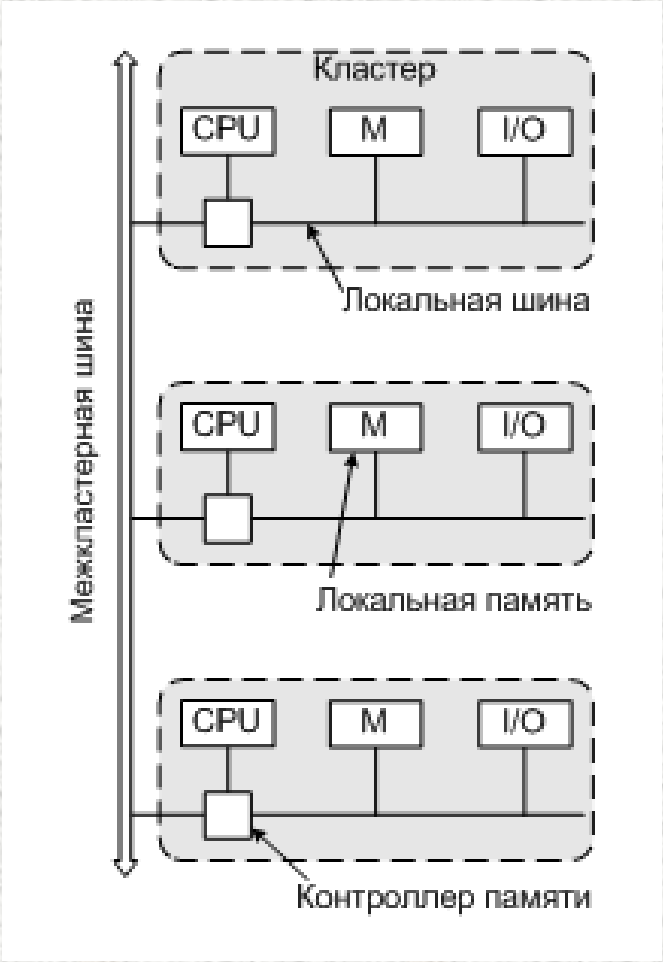
Иногда говорят, что классические SMP-компьютеры обладают архитектурой UMA (Uniform Memory Access), обеспечивая одинаковый доступ любого процессора к любому модулю памяти.

Как сохранить единое адресное пространство и одновременно сохранить возможность для увеличения числа процессоров?

…например, отказаться от однородности доступа к памяти, перейти от архитектуры SMP к архитектуре NUMA:

Non Uniform Memory Access.

Архитектура NUMA
(компьютер Cm*)



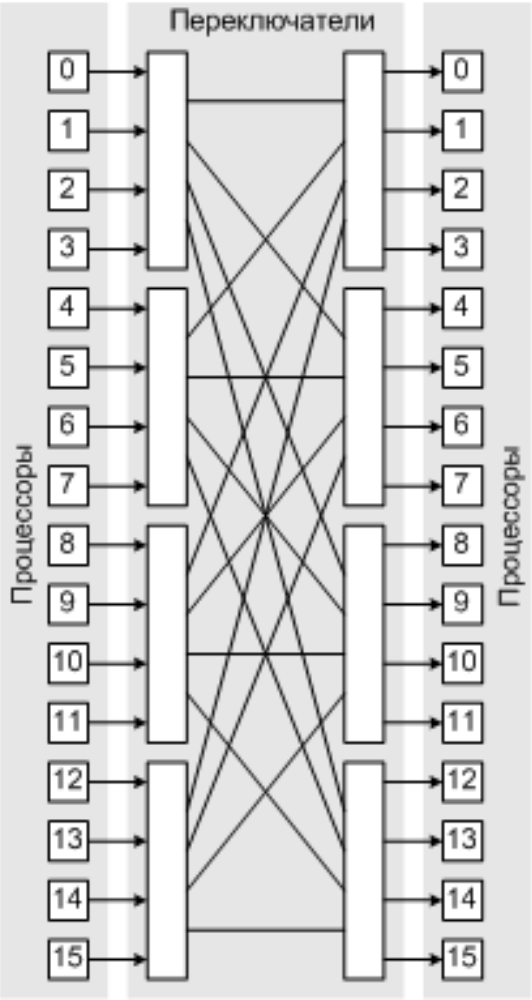
Конец 70-х годов 20-го века.

Контроллер памяти по старшим разрядам адреса определяет, в каком модуле хранятся нужные данные.

Запрос выставляется либо на локальную шину, либо на межкластерную шину.

Межкластерная шина – узкое место данной архитектуры.

Архитектура NUMA
(компьютер BBN Butterfly)



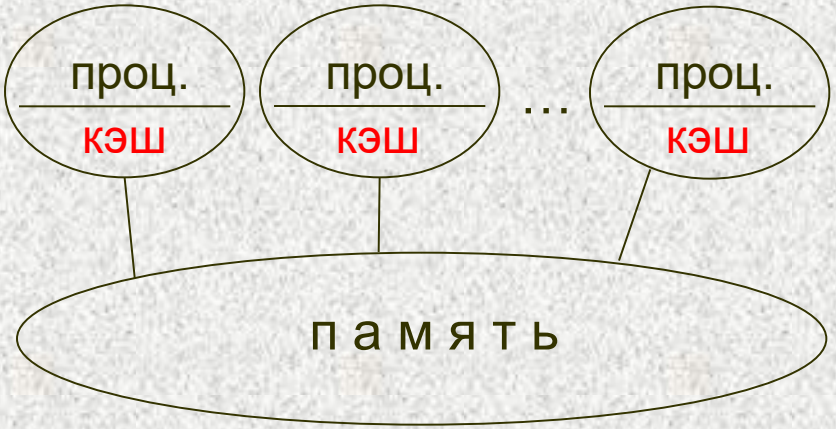
Максимальная конфигурация – 512 процессоров.

Связь процессоров с удалёнными блоками данных – через коммутатор Butterfly.

Локальные ссылки – около 2 мкс, удалённые – около 6 мкс.

Архитектура NUMA и ccNUMA

(все бы хорошо, да кэш-память появилась…)



В процессорах первых NUMA-компьютеров кэш-памяти не было.

Если процессор P1 сохранил значение x в ячейке q, а затем процессор P2 читает значение ячейки q, что он получит? Если x не вытеснено из кэш-памяти в оперативную память, то получит старое значение.

Проблема согласования содержимого кэш-памяти (cache coherence problem, проблема когерентности кэшей)

Архитектура NUMA и ccNUMA

(все бы хорошо, да кэш-память появилась…)

Специальная модификация NUMA-архитектуры:

ccNUMA:
cache coherent Non Uniform Memory Access.

Проблема когерентности кэшей решается на аппаратном уровне.

Важный вопрос – насколько неоднородна архитектура NUMA?

Если обращение к памяти другого узла требует на 5-10% больше времени, чем обращение к своей памяти, то к такой системе в большинстве случаев можно относиться, как к UMA (SMP).

Но чаще всего разница составляет 200-700%, и это существенно влияет на производительность.

Компьютеры с общей памятью

(Hewlett-Packard Superdome)

Выпускается с 2000 г. Крайне успешный – в 2001 году в Top500 занимал 147 позиций.

В стандартной комплектации объединяет от 2 до 64 процессоров с возможностью расширения системы.

Архитектура ccNUMA.

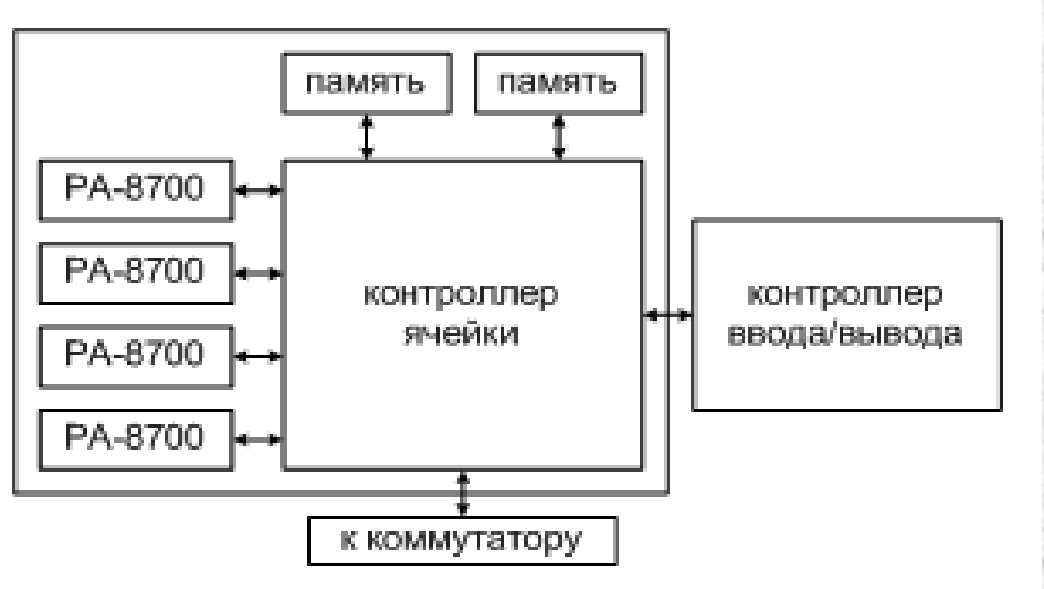
До 256 Гбайт оперативной памяти.

Используемые процессоры:
- *PA-8600, PA-8700, PA-8900*
- *Intel IA/64: Itanium, Itanium 2*

Компьютеры с общей памятью

(Hewlett-Packard Superdome)

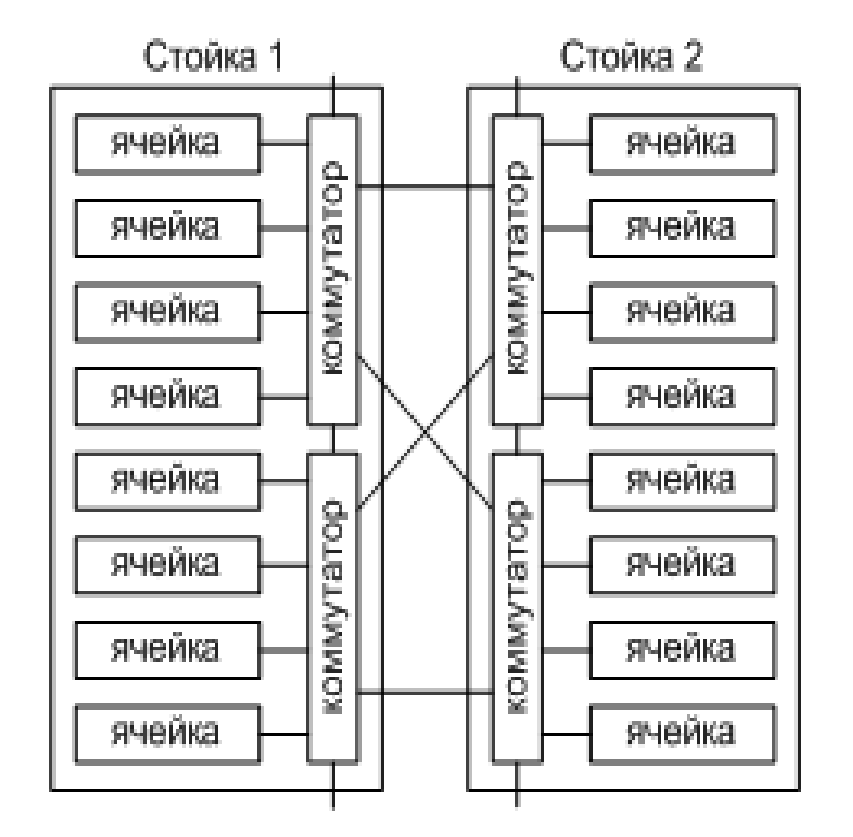
Вычислительные ячейки (cells):



До 4 процессоров.
До 16 Гбайт оперативной памяти, разделённой на 2 банка.
Контроллер ячейки (24 миллиона транзисторов) – интерфейсные функции, когерентность кэшей.
Канал процессор-контроллер – 2 Гбайт/сек.
Канал банк памяти-контроллер – 2 Гбайт/сек.
Канал контроллер-внешний коммутатор – 8 Гбайт/сек.

Компьютеры с общей памятью
(Hewlett-Packard Superdome)

Базовая конфигурация:



2 стойки.

В каждой стойке – 2 восьмипортовых коммутатора.

Все порты коммутаторов работают со скоростью 8 Гбайт/сек.

К каждому коммутатору подключено 4 ячейки.

3 других порта для связи с коммутаторами.

8-ой порт – для формирования многоузловой конфигурации.

Компьютеры с общей памятью

(Hewlett-Packard Superdome)

Виды задержек при обращении процессора к памяти, являющихся своего рода платой за высокую масштабируемость системы в целом:

- *процессор и память располагаются в одной ячейке – в этом случае задержка минимальна;*
- *процессор и память располагаются в разных ячейках, но обе эти ячейки подсоединены к одному и тому же коммутатору;*
- *процессор и память располагаются в разных ячейках, причём обе эти ячейки подсоединены к разным коммутаторам – в этом случае запрос должен пройти через два коммутатора, и задержки будут максимальными.*

В HP Superdome средняя задержка при переходе от 4 до 64 процессоров возрастает только в 1.6 раза.

Компьютеры с общей памятью

(Hewlett-Packard Superdome)

Компьютер HP Superdome может быть классическим единым компьютером с общей памятью. Однако его можно сконфигурировать и таким образом, что он будет являться совокупностью независимых разделов (nPartitions), работающих под различными операционными системами, в частности, под HP UX, Linux и Windows. Также предусмотрены организация эффективной работы с большим числом внешних устройств, возможности горячей замены всех основных компонент аппаратуры, резервирование, мониторинг базовых параметров.

Процессор PA-8700 имеет тактовую частоту 750 МГц, может выполнять 4 операции за такт, что даёт пиковую производительность 3 Гфлопс. Следовательно, пиковая производительность базовой 64-процессорной конфигурации – 192 Гфлопс.

Причины снижения производительности компьютеров с общей памятью

1. *Закон Амдала.*
2. *ccNUMA (неоднородность доступа к памяти).*
3. *ccNUMA (необходимость согласования кэш-памяти).*
4. *Конфликты при обращении в память.*
5. *Сбалансированность вычислительной нагрузки процессоров.*
6. *Производительность отдельных процессоров.*
7. *…*

Причин много, все они в той или иной мере проявляются в каждой программе. Их нужно ясно себе представлять, чтобы выбрать правильный способ решения.

Закон Амдала

$$S \le \frac{1}{\alpha + \frac{(1-\alpha)}{p}} \qquad S \approx \frac{1}{\alpha}$$

где:

α – доля последовательных операций,

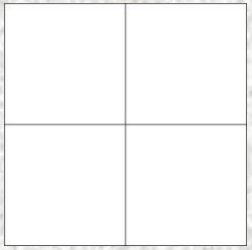
p – число процессоров в системе.

Если в программе 20% всех операций должны выполняться строго последовательно, то ускорения больше 5 получить нельзя вне зависимости от числа использованных процессоров.

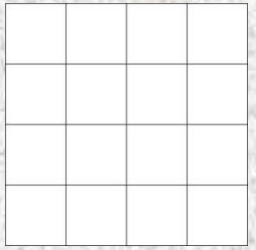
При использовании моделей параллельных программ с общей памятью возникают дополнительные участки последовательного кода, связанные с синхронизацией доступа к общим данным, например, критические секции. Реально эти фрагменты будут последовательными участками кода.

Сбалансированность вычислительной нагрузки процессоров

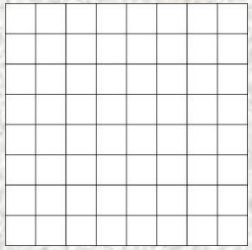
4 процессора

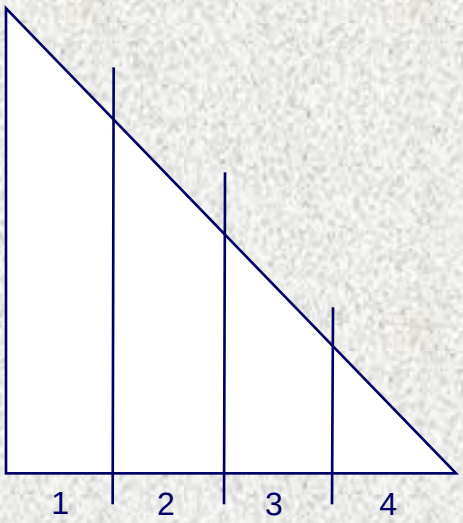


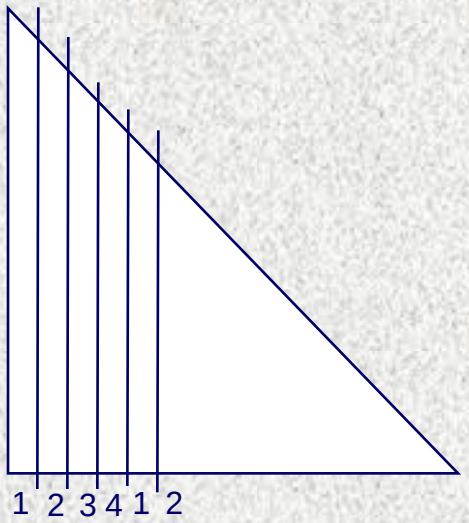
16 процессоров

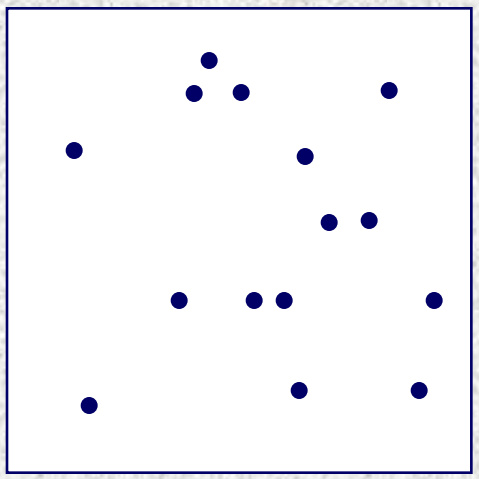


64 процессора









В случае систем с общей памятью ситуация упрощается тем, что практически всегда системы являются однородными, поэтому о сложной стратегии распределения работы речь, как правило, не идёт.

Компьютеры с распределенной памятью

Компьютеры с распределенной памятью

- *Используемые процессоры,*
- *Коммуникационная сеть.*

Примеры:

- *Intel Paragon: Intel i860, двумерная прямоугольная решетка.*
- *IBM SP1/SP2: IBM Power, высокоскоростной коммутатор каждый-с-каждым.*
- *«Ломоносов»: Intel Xeon + NVIDIA Tesla,QDR Infiniband, топология «толстое дерево»*

Компьютеры с распределенной памятью

Топология - конфигурация графа сети. Определяет взаимное расположение узлов и каналов связи.

В архитектуру сети помимо топологии входят также алгоритмы маршрутизации (routing) и управления потоками (flow control).

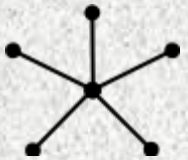
Компьютеры с распределенной памятью

Примеры коммуникационных топологий:

- *линейка – для системы из p узлов нужно p-1 соединение, средняя длина пути между двумя узлами равна p/3;*

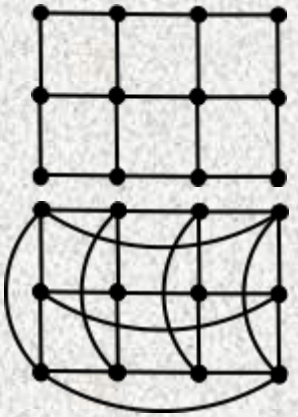
- *кольцо – для построения системы из p узлов нужно p соединений, средняя длина пути между двумя узлами равна p/6; увеличивается отказоустойчивость за счёт того, что передача сообщений может идти по двум направлениям;*

- *звезда – общение только через центральный узел, p-1 соединение, длина пути между концевыми узлами равна 2; большая нагрузка на центральный узел;*

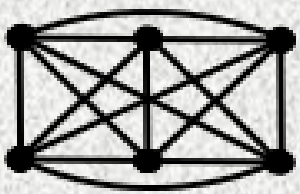


Компьютеры с распределенной памятью

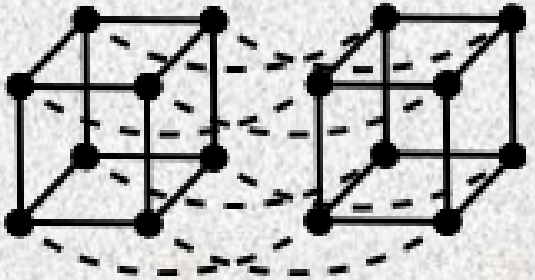
Примеры коммуникационных топологий:



- *двумерная решётка, двумерный тор, … многомерные аналоги;*



- *полносвязная топология – каждый узел имеет связь с каждым другим узлом; для соединения p узлов требуется p(p-1)/2 связей;*

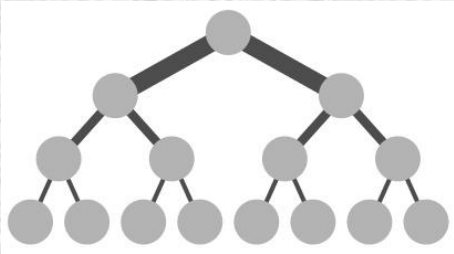


- *двоичный гиперкуб – в n-мерном пространстве поместим $p=2^n$ узлов системы в вершины единичного n-мерного куба; каждый узел соединим с соседом вдоль каждого из n измерений – всего $\log_2 p$ соединений; максимальное расстояние между вершинами n-мерного гиперкуба равно n.*

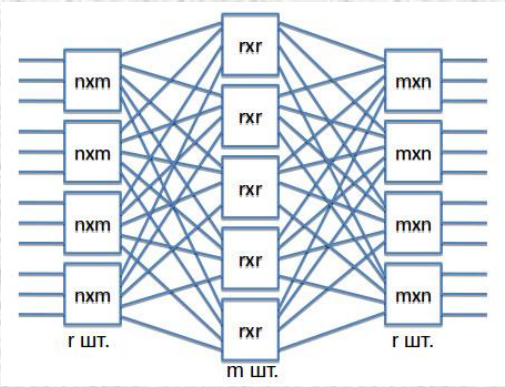
Компьютеры с распределенной памятью

Примеры коммуникационных топологий:

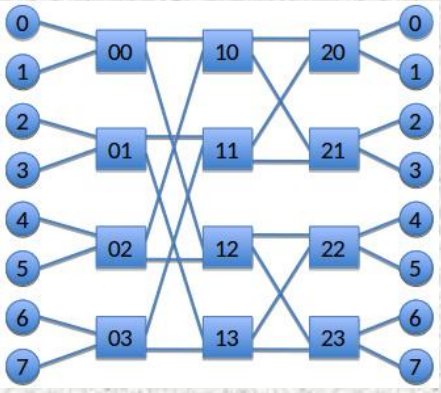
- *дерево, толстое дерево;*



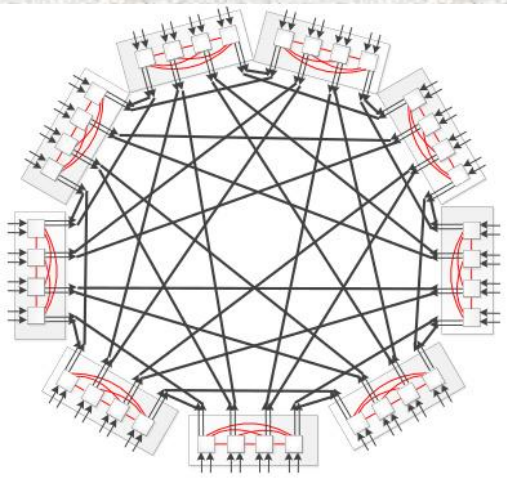
- *сети Клоза;*



- *бабочки (k-ary n-flies), flattened butterfly;*



- *dragonfly.*



Компьютеры с распределенной памятью

Параметры коммуникационных сетей:

- *Длина критического пути (диаметр) – минимальное количество элементарных связей, которые нужно пройти для коммутации двух самых удаленных процессоров.*
- *Связность – количество элементарных связей, которые нужно удалить, чтобы схема распалась на две несвязанные части.*
- *Сложность – общее количество необходимых элементарных связей.*

Компьютеры с распределенной памятью

Схема коммутации	Длина критического пути	Связность	Сложность
линейка	p-1	1	p-1
кольцо	$\lfloor p/2 \rfloor$	2	p
звезда	2	1	p-1
двумерная решётка	$2(\sqrt{p}-1)$	2	$2(p-\sqrt{p})$
двумерный тор	$2\lfloor \sqrt{p}/2 \rfloor$	4	2p
полносвязная топология	1	p-1	p(p-1)/2
двоичный гиперкуб	$\log_2 p$	$\log_2 p$	$p\log_2 p/2$

Компьютеры с распределенной памятью

Другие параметры коммуникационных сетей:

- *Количество портов маршрутизаторов (radix): low-radix/high-radix.*
- *Масштабируемость.*
- *Расширяемость.*
- *Простота эксплуатации.*
- *Устойчивость к отказам.*
- *Многообразие путей.*
- *…*

Компьютеры с распределенной памятью

(семейство Cray XT)

Семейство: T3D, T3E, XT3, XT4, XT5, XT6, XE6
Cray T3D – начало 90-х годов
Cray XT5 – на ноябрь 2010 г. второе место в Top500

Используемые процессоры:
T3D/T3E – DEC ALPHA,
XT3/XT4/XT5/XT6/XE6 – AMD Opteron,
XE6 + NVIDIA Tesla

Топология коммуникационной сети: трёхмерный тор.

Компьютеры с распределенной памятью

(семейство Cray XT)

Узлы суперкомпьютера:

- ***пользовательские**: компиляция, командные файлы, однопроцессорные задачи;*
- ***операционной системы**: выполнение многих системных сервисных функций ОС, в частности, работа с файловой системой;*
- ***вычислительные**: выполнение программ пользователя в монопольном режиме.*

Реальные конфигурации Cray T3E: 24/16/576 или 7/5/260 (пользовательские узлы/узлы ОС/вычислительные узлы)

Вычислительные узлы:
процессор, локальная память, …
+ сетевой интерфейс (он же – контроллер прямого асинхронного доступа в память).
Любой процессор через свой сетевой интерфейс может обращаться к памяти любого другого процессора, не прерывая его работы.

Компьютеры с распределенной памятью

(семейство Cray XT)

Коммуникационная сеть

Трёхмерный тор (в узлах тора сетевые маршрутизаторы):

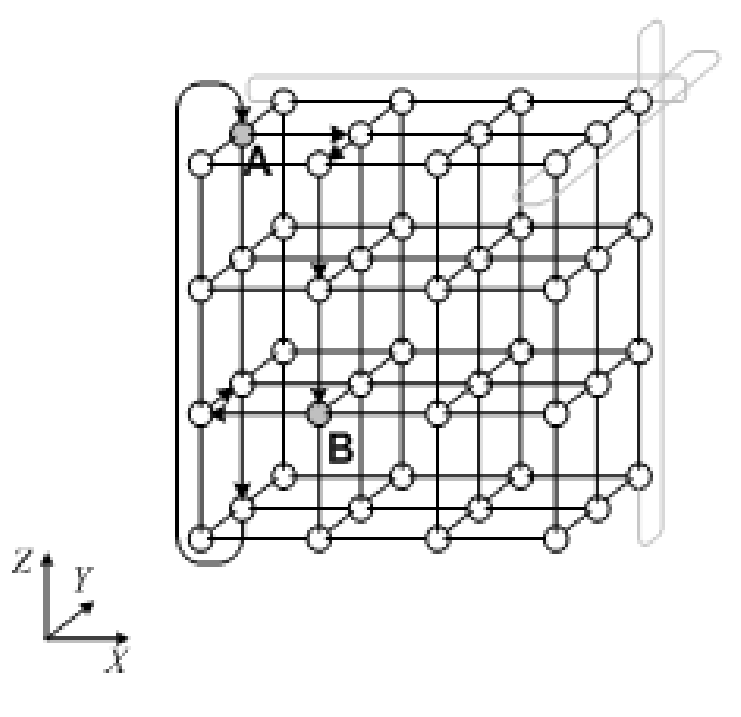
- *отказоустойчивость, возможность выбора альтернативного маршрута для обхода поврежденных связей;*
- *сокращение расстояния между узлами.*

У каждого узла всегда шесть непосредственных соседей.

Между узлами – два однонаправленных канала передачи данных, что допускает одновременный обмен данными в противоположных направлениях.

Компьютеры с распределенной памятью
(семейство Cray XT)

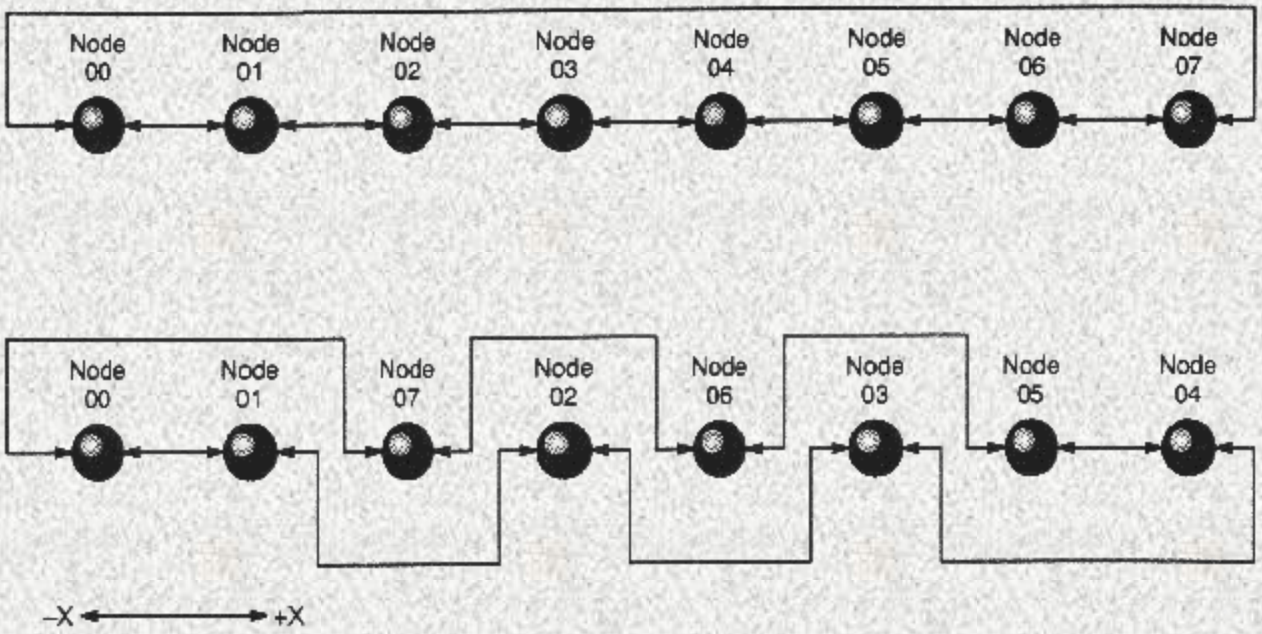
Коммуникационная сеть



Маршрутизация между вершинами A и B:
Сначала смещение по измерению X до тех пор, пока координата очередного транзитного узла и вершины B по измерению X не станут равными. Затем аналогичная процедура по Y, а в конце по Z.
Смещение может быть как положительным, так и отрицательным, минимизируя число перемещений по сети и обходя повреждённые связи.
Параллельный транзит данных по каждому из измерений X, Y и Z может выполняться одновременно.

Компьютеры с распределенной памятью
(семейство Cray XT)

Коммуникационная сеть

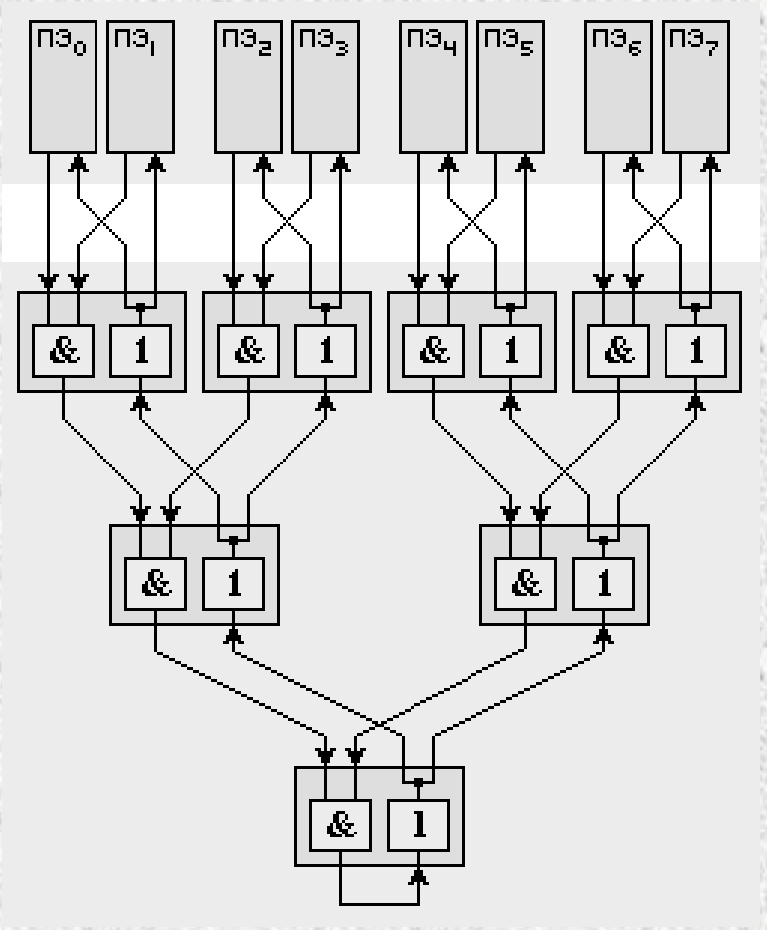


Все узлы в коммуникационной сети в размерностях X и Z расположены с чередованием, что позволяет минимизировать длину максимального физического соединения между узлами.

Компьютеры с распределенной памятью

(семейство Cray XT)

Аппаратная поддержка барьерной синхронизации:



Барьер - точка в программе, при достижении которой каждый процесс должен ждать, пока остальные процессы также не дойдут до барьера. После этого все процессы могут продолжать работу дальше.

Несколько входных и выходных регистров синхронизации. Каждый разряд этих регистров соединён со своей независимой цепью реализации барьера.
Двоичное дерево на основе двух типов устройств: логическое умножение (&) и дублирование входа на два своих выхода (1).

Кластеры, кластеры, кластеры…

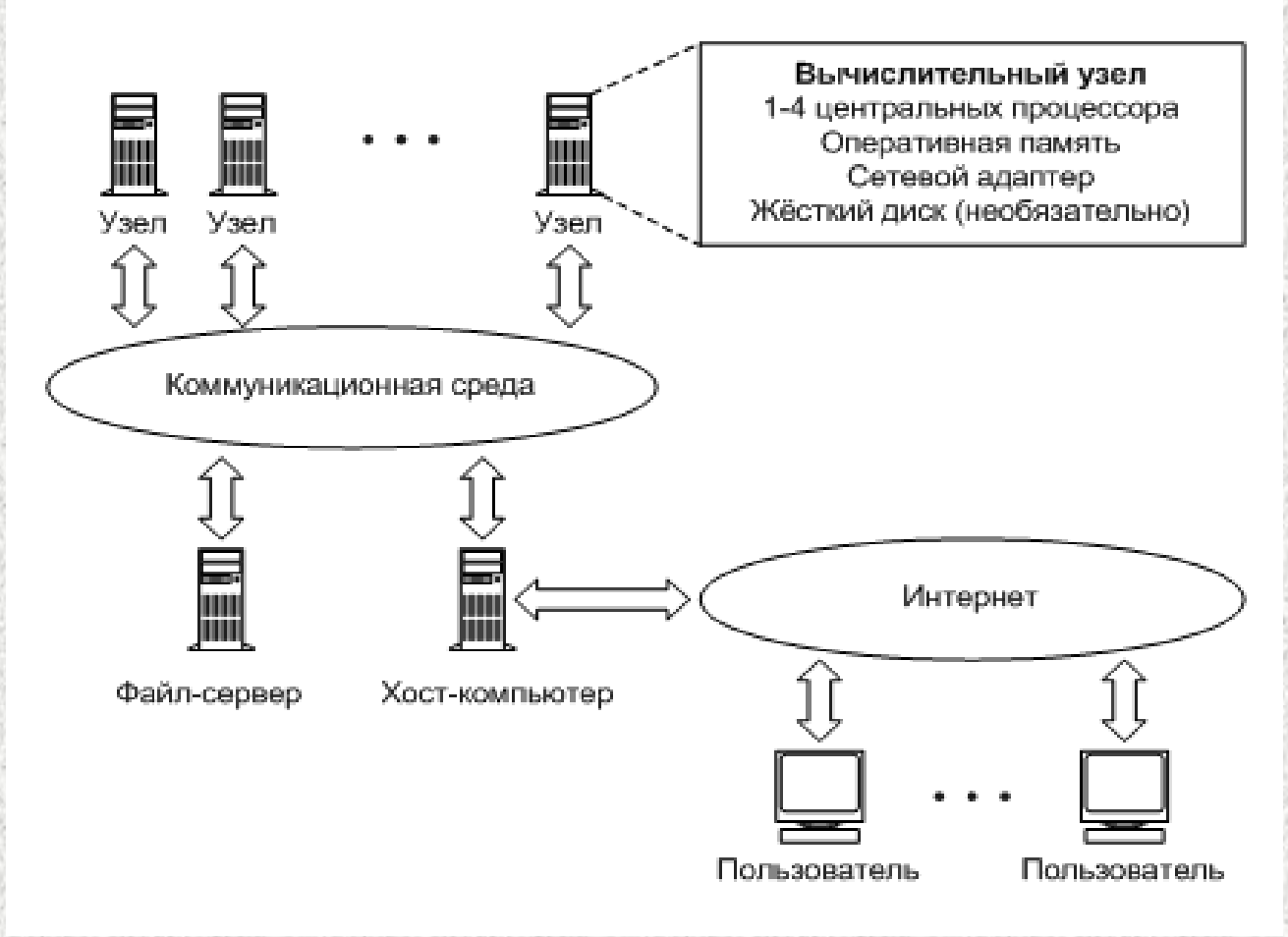
Вычислительный кластер - это совокупность компьютеров, объединённых в рамках некоторой коммуникационной сети для решения одной задачи.

- Доступные процессорные платформы
- Стандартные сетевые технологии
- Свободное ПО: Linux, MPI, GNU...
- Невысокая стоимость

474 компьютера из Top500 мира – кластеры

(по данным 55-ой редакции списка Top500, июнь 2020 г.)

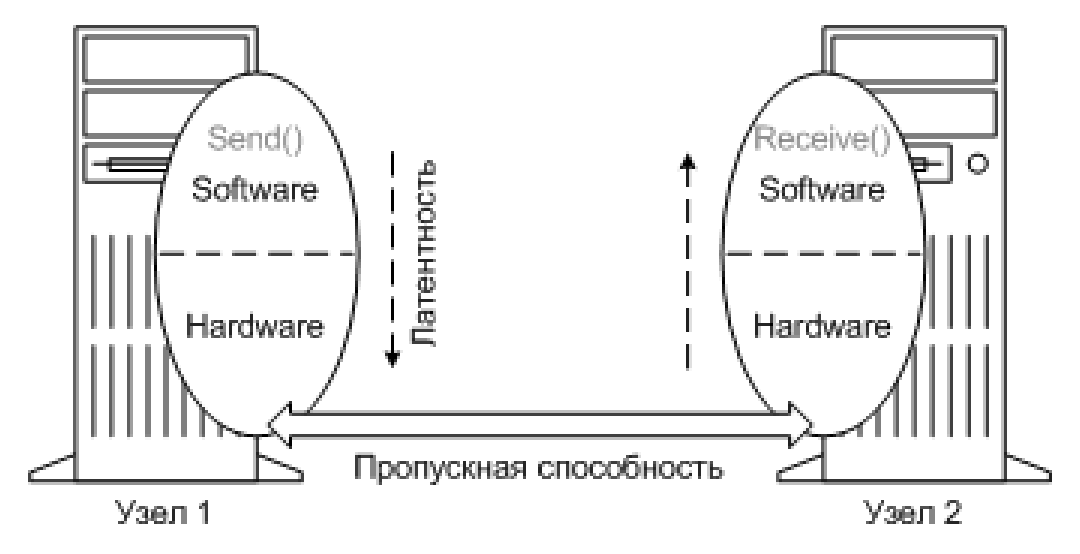
Компьютеры с распределенной памятью
(вычислительные кластеры)



Возможность формирования архитектуры кластера с учетом характерных особенностей решаемых задач

Компьютеры с распределенной памятью

(вычислительные кластеры)



Латентность и пропускная способность сети – основные параметры коммуникационной сети кластеров.

***Латентность** - время начальной задержки при посылке сообщений.*

***Пропускная способность** сети определяется скоростью передачи информации по каналам связи и измеряется объёмом передаваемой информации в единицу времени.*

Компьютеры с распределенной памятью

(вычислительные кластеры)

Время на передачу сообщения по коммуникационной сети вычисляется по следующей формуле:

$t_N = t_0 + N/S$,

Где t_0 - латентность, N - объём передаваемых данных, S - пропускная способность сети.

Если в программе много маленьких сообщений, то сильно скажется латентность.

Если сообщения передаются большими порциями, то важна высокая пропускная способность каналов связи.

Наличие латентности определяет и тот факт, что максимальная скорость коммуникаций не может быть достигнута на сообщениях с небольшой длиной.

Компьютеры с распределенной памятью

(вычислительные кластеры)

На практике пользователям не столько важны заявляемые производителем пиковые характеристики, сколько реальные показатели, достигаемые на уровне приложений, например, из программ, использующих технологию MPI.

Измерение латентности на практике:

- *процесс p_1 замеряет время t_1;*
- *процесс p_1 посылает сообщение длины 0 процессу p_2;*
- *процесс p_2 принимает сообщение от процесса p_1 и сразу посылает сообщение длины 0 обратно процессу p_1;*
- *процесс p_1 принимает сообщение от процесса p_2;*
- *процесс p_1 замеряет время t_2;*
- *под латентностью понимают величину $t_0 = (t_2 - t_1)/2$.*

Компьютеры с распределенной памятью

(вычислительные кластеры)

Пропускная способность и латентность сильно зависят от алгоритмов маршрутизации и топологии коммуникационной сети.

Важна также стоимость коммуникационной сети, которая складывается из:

- *Количества и стоимости маршрутизаторов.*
- *Длины и типа кабелей.*

Что снижает производительность компьютеров с распределенной памятью?

1. *Закон Амдала*
2. *Латентность передачи по сети*
3. *Пропускная способность каналов передачи данных*
4. *Особенности использования SMP-узлов*
5. *Балансировка вычислительной нагрузки*
6. *Возможность асинхронного счета и передачи данных*
7. *Особенности топологии коммуникационной сети*
8. *Производительность отдельных процессоров*
9. *...*

Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ (лекция 5)

А.С.Антонов

Вед. н.с. НИВЦ МГУ, к.ф.-м.н.

asa@parallel.ru

ВМК МГУ, 2020

Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ (лекция 6)

А.С.Антонов

Вед. н.с. НИВЦ МГУ, к.ф.-м.н.

asa@parallel.ru

ВМК МГУ, 2020

Суперкомпьютерный комплекс МГУ имени М.В.Ломоносова

«Ломоносов-2»

5.5 Pflops

«Ломоносов»

1.7 Pflops

«Чебышев»

60 Tflops

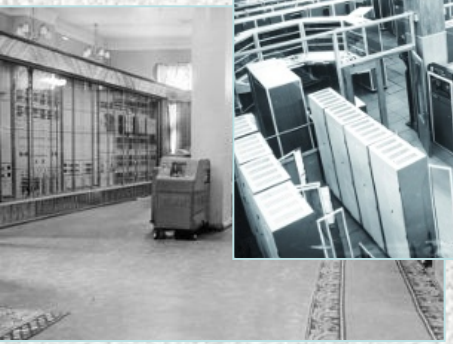
SCI

12 Gflops

БЭСМ-6

1 млн. оп/сек.

«Стрела»



2000 оп/сек.

1956 1968 2000 2007 2009 2014

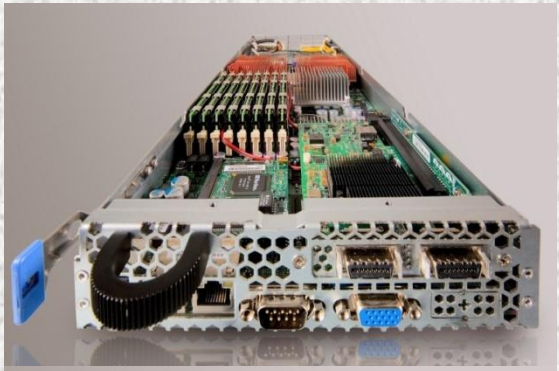
Суперкомпьютерный комплекс МГУ имени М.В.Ломоносова

- Суперкомпьютер «Ломоносов-2», 5.5 PFlop/s
- Суперкомпьютер «Ломоносов», 1.7 PFlop/s
- Суперкомпьютер «Чебышёв», 60 Tflop/s
- Суперкомпьютер Polus, 56 TFlop/s
- Суперкомпьютер IBM Blue Gene/P, 27 TFlop/s

Суперкомпьютер СКИФ МГУ «Чебышёв»

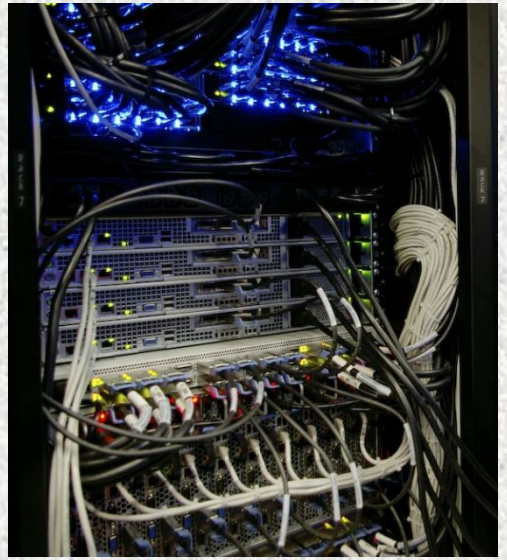
- 1 место в 8-ой редакции списка Топ50 наиболее мощных компьютеров СНГ (март 2008 года)
- 36 место в 31-ой редакции списка Top500 наиболее мощных компьютеров мира (июнь 2008 года)
- Пиковая производительность: 60 TFlop/s
- Производительность на Linpack: 47.32 TFlop/s (79% пиковой), матрица 740000x740000
- 625 вычислительных узлов, 1250 процессоров, 5000 процессорных ядер
- 42 стойки: 14 вычислительных, 28 инфраструктурных
- Помещение 98 $м^2$
- Общий вес оборудования: более 30 тонн
- Энергопотребление вычислительной части 330 КВт, всего комплекса в пике до 720 КВт
- Система бесперебойного электропитания
- 10 минут автономной работы
- Система охлаждения
- Звукоизоляция
- Система автоматического газового пожаротушения

Суперкомпьютер СКИФ МГУ «Чебышёв»



Вычислительные узлы:

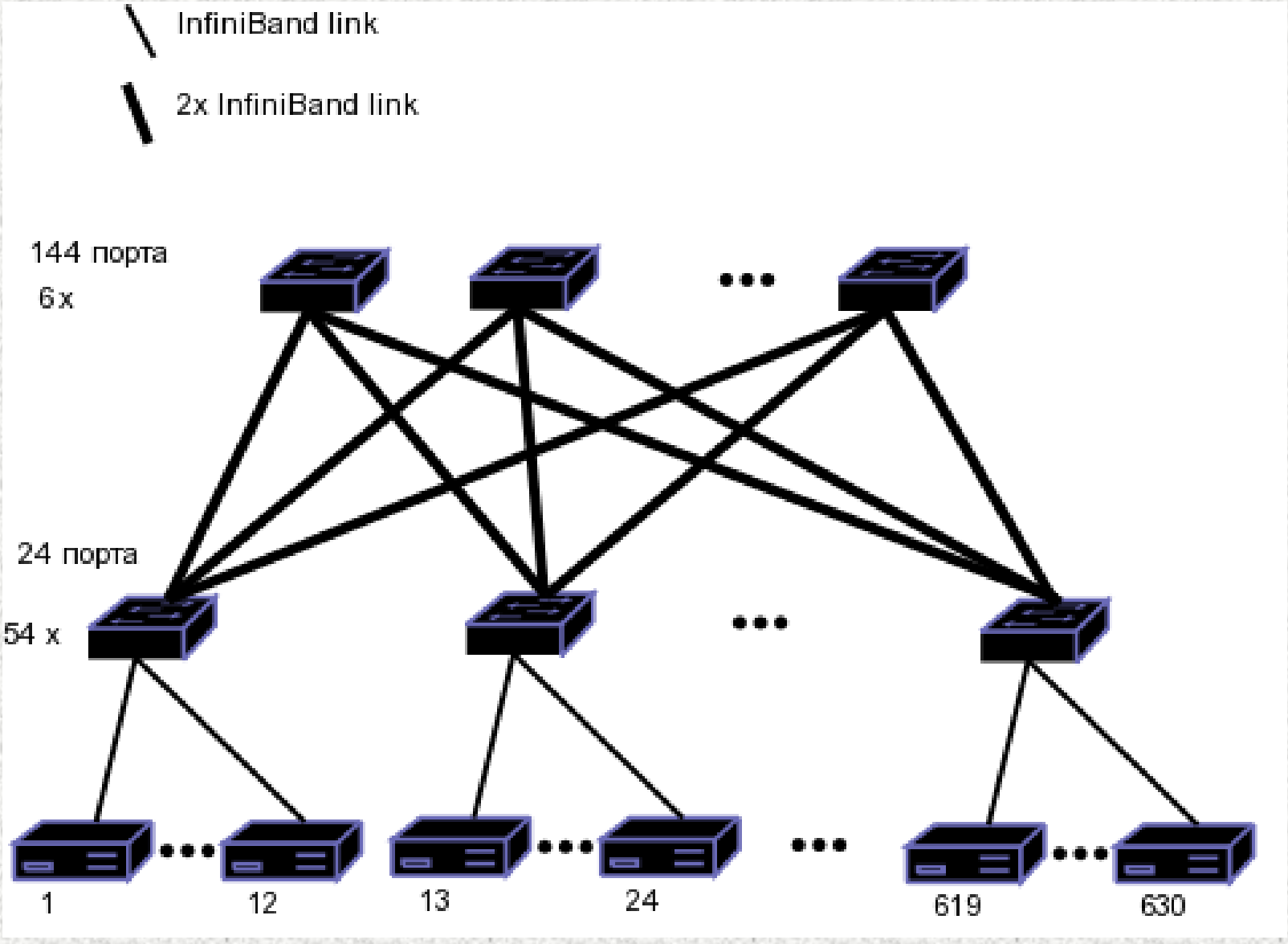
- Процессоры:
  - ✓1250 Intel E5472 3.0 ГГц Harpertown
- Блэйд-шасси T-Blade («Т-Платформы»)
  - ✓Форм-фактор 5 U
  - ✓До 10 вычислительных узлов
- Оперативная память:
  - ✓529 x 8 ГБ, бездисковые
  - ✓64 x 8 ГБ, 160 ГБ HDD
  - ✓32 x 16 ГБ, 160 ГБ HDD
  - ✓8 x 32 ГБ, 160 ГБ HDD



Коммуникационная сеть:

- DDR InfiniBand
  - ✓Mellanox MT25418 NIC
  - ✓FatTree
  - ✓SilverStorm 9120 – базовые коммутаторы
  - ✓Flextronix F-X430046 – листовые коммутаторы
- Характеристики
  - ✓1.3 – 1.95 μs латентность
  - ✓1.7 ГБ/с пропускная способность

Схема построения Fat Tree в суперкомпьютере СКИФ МГУ «Чебышёв»



Суперкомпьютер СКИФ МГУ «Чебышёв»

Вспомогательные сети:

•Gigabit Ethernet: коммутаторы Force10 C300 и Force10 S2410

•Управляющая сеть ServNet + IPMI

Хранилище данных:

•60 ТБ распределённое отказоустойчивое сетевое хранилище T-Platforms ReadyStorage ActiveScale Cluster

•15 ТБ локальных дисков на узлах

•Ленточное хранилище Quantum Scalar i500

Система охлаждения

•8 кондиционеров APC InfraStruXure ACR502, уровень резервирования N+2

•Холодильные машины Liebert-Hiross SLH 023, одновременно работают 2 из 3

Горячий коридор:

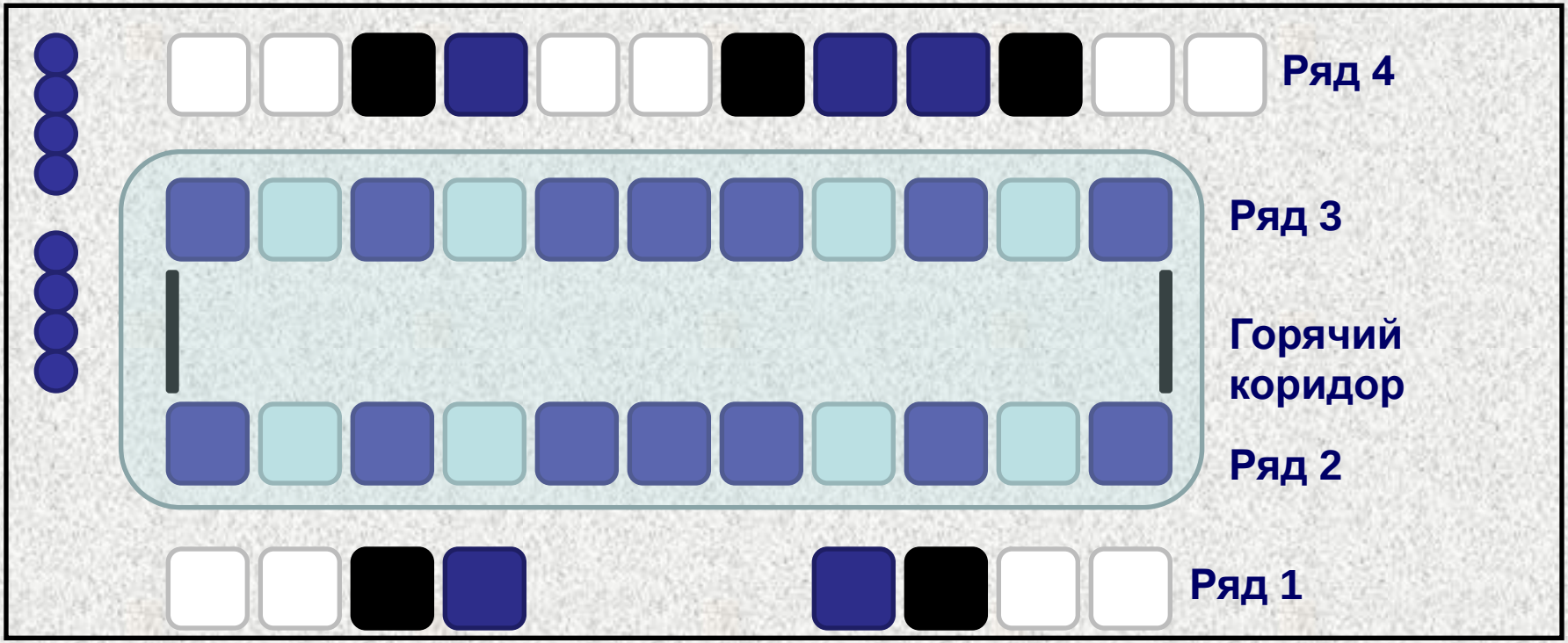
•Меньший объём охлаждаемой части помещения

•Более тесная компоновка

•Встречные воздушные потоки

Суперкомпьютер СКИФ МГУ «Чебышёв»

Компоновка системы:



Системы охлаждения

Вычислительные узлы, система хранения, коммуникационная сеть

Батареи

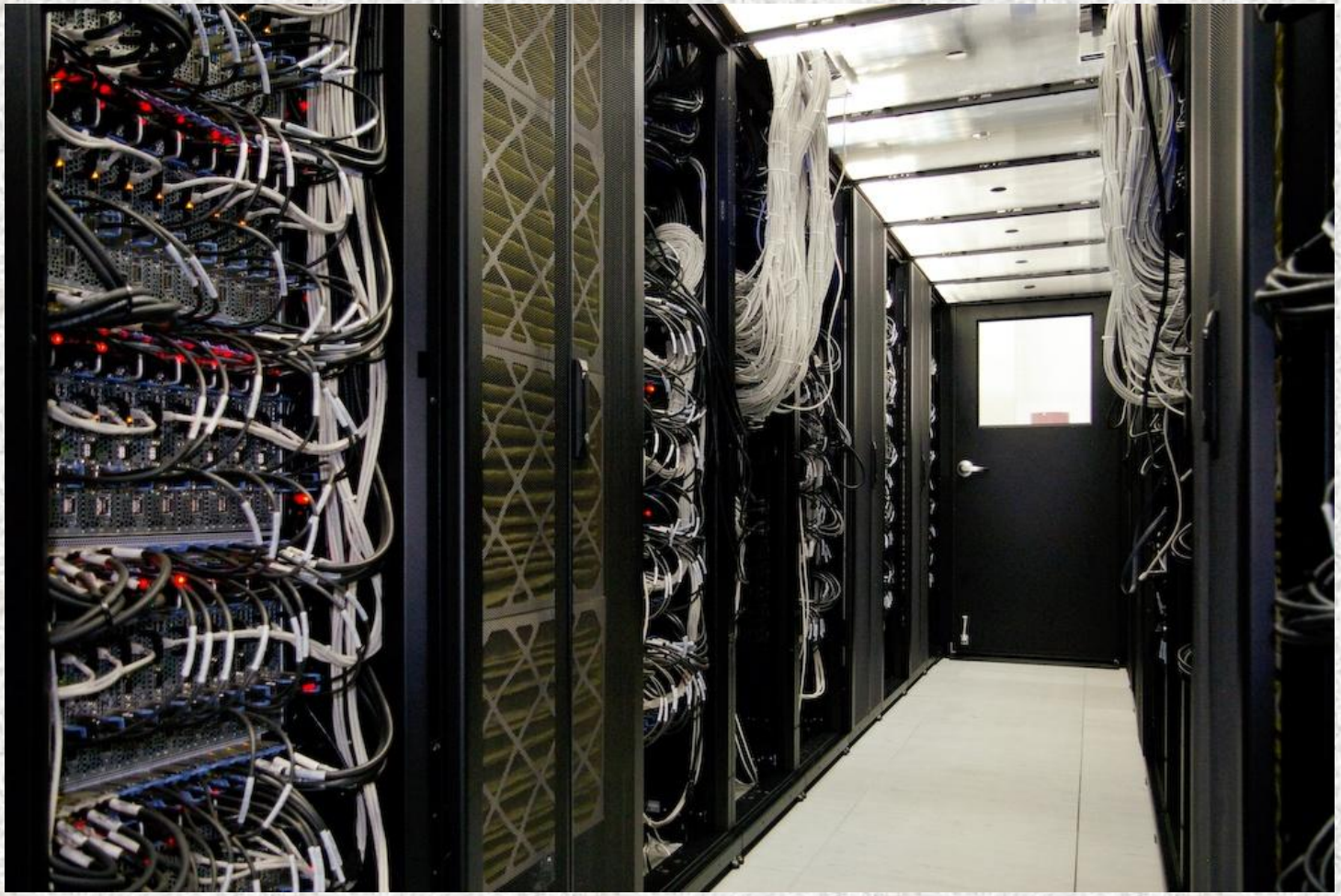
UPS

PDU

Газовые баллоны

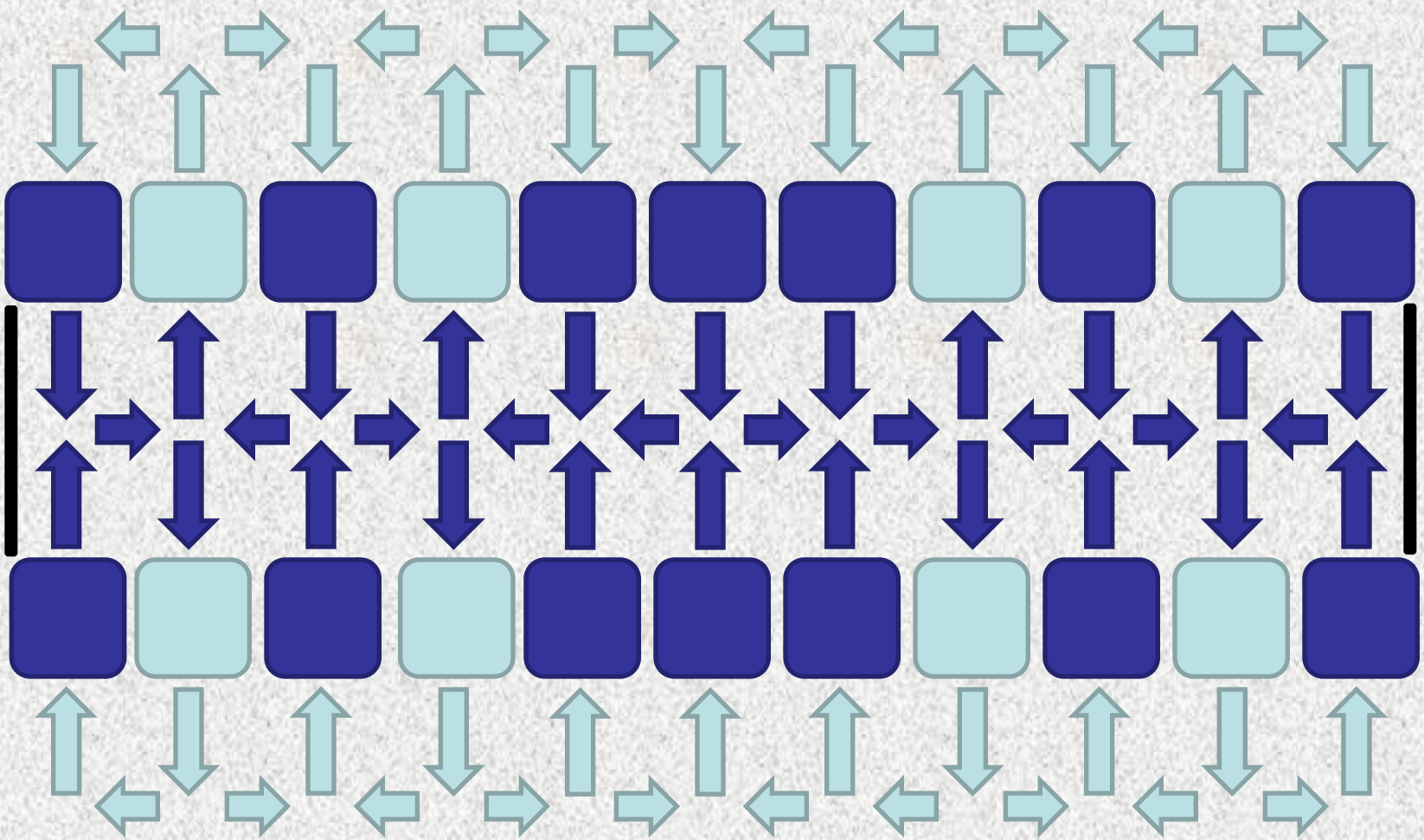
Суперкомпьютер СКИФ МГУ «Чебышёв»

Горячий коридор:



Суперкомпьютер СКИФ МГУ «Чебышёв»

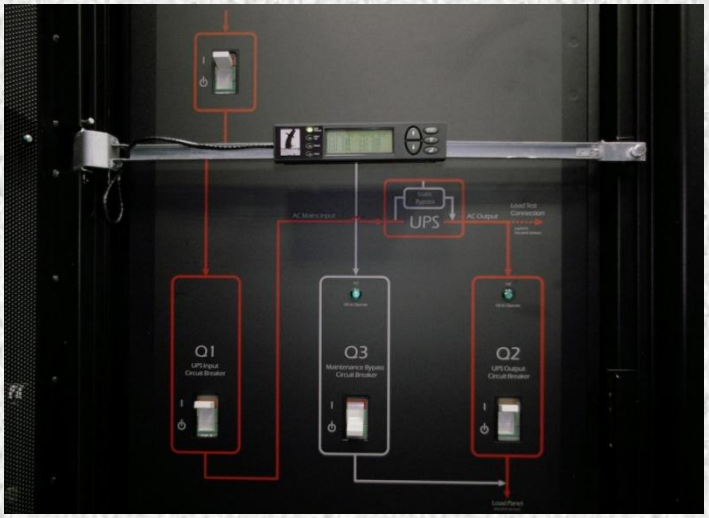
Горячий коридор:



Суперкомпьютер СКИФ МГУ «Чебышёв»

Система пожаротушения:
•Возможность ручного отключения всего комплекса
•Инертный газ
•3 месяца тестирования на ложные срабатывания
•При входе в помещение автоматическая система отключается

Электрическое оборудование
•1-ый и 4-ый ряды стоек
•PDU: APC AP9565
•UPS: APC Symmetra PX
•Мониторинг: ISX Manager
•Уровень резервирования N+1

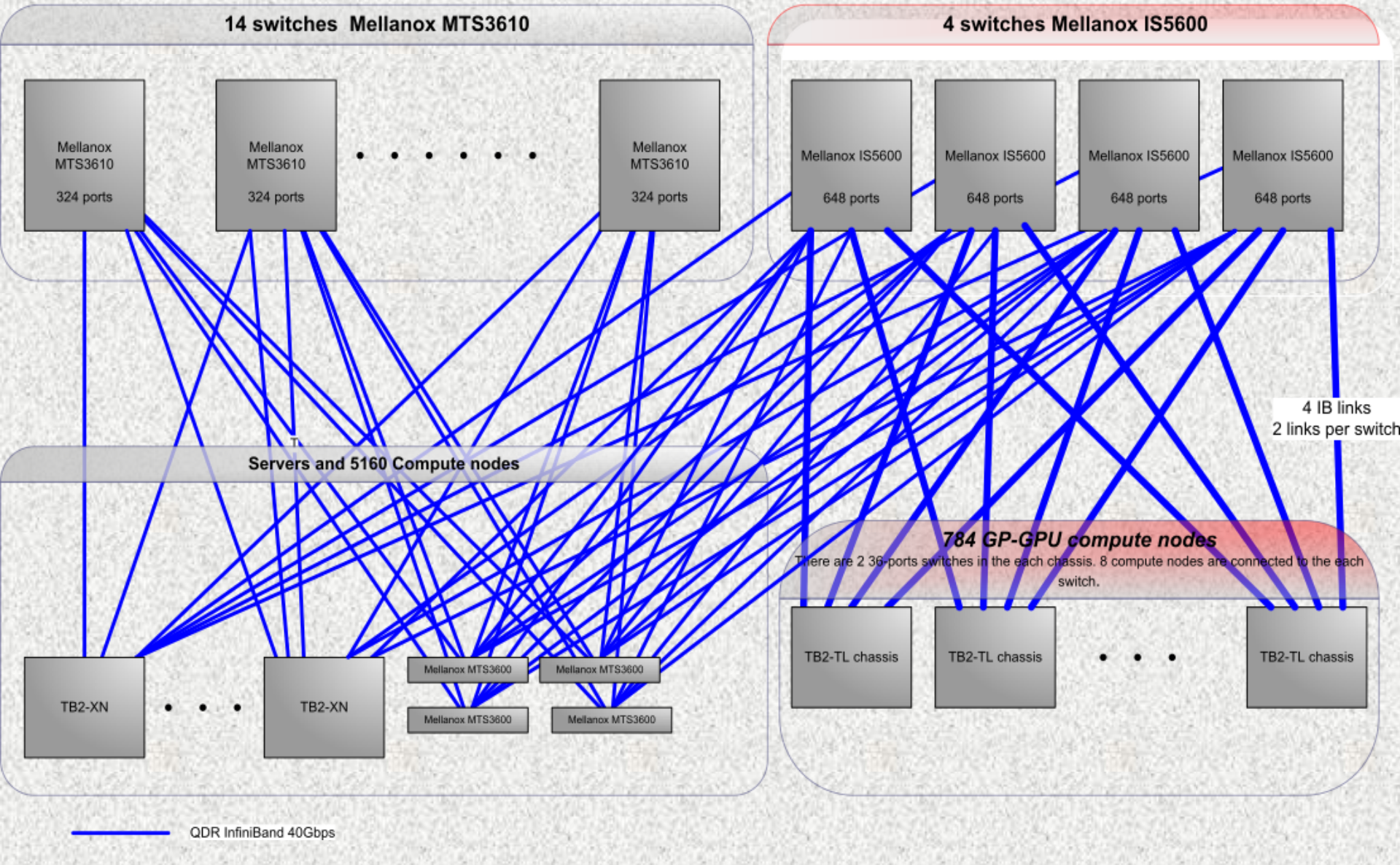


Суперкомпьютер «Ломоносов»

Пиковая производительность	1700.21 TFlop/s
Производительность (Linpack)	901.90 TFlop/s
Эффективность	53%
Вычислительных узлов (Intel)	5 104
Вычислительных узлов (ГПУ)	1 065
Вычислительных узлов (PowerXCell)	30
Процессоры Intel Xeon 5570, 5670	12 346
NVIDIA Tesla X2070	2 130
Число процессорных ядер (x86)	52 168
Число процессорных ядер (ГПУ)	954 240
Оперативная память	92 ТБайт
Коммуникационная сеть	QDR Infiniband / 10 GE
Система хранения данных	1.75 ПБайт, Lustre, NFS, …
Операционная система	Clusrtx T-Platforms Edition
Занимаемая площадь (вычислитель)	252 м^2
Энергопотребление (вычислитель)	2.8 МВт

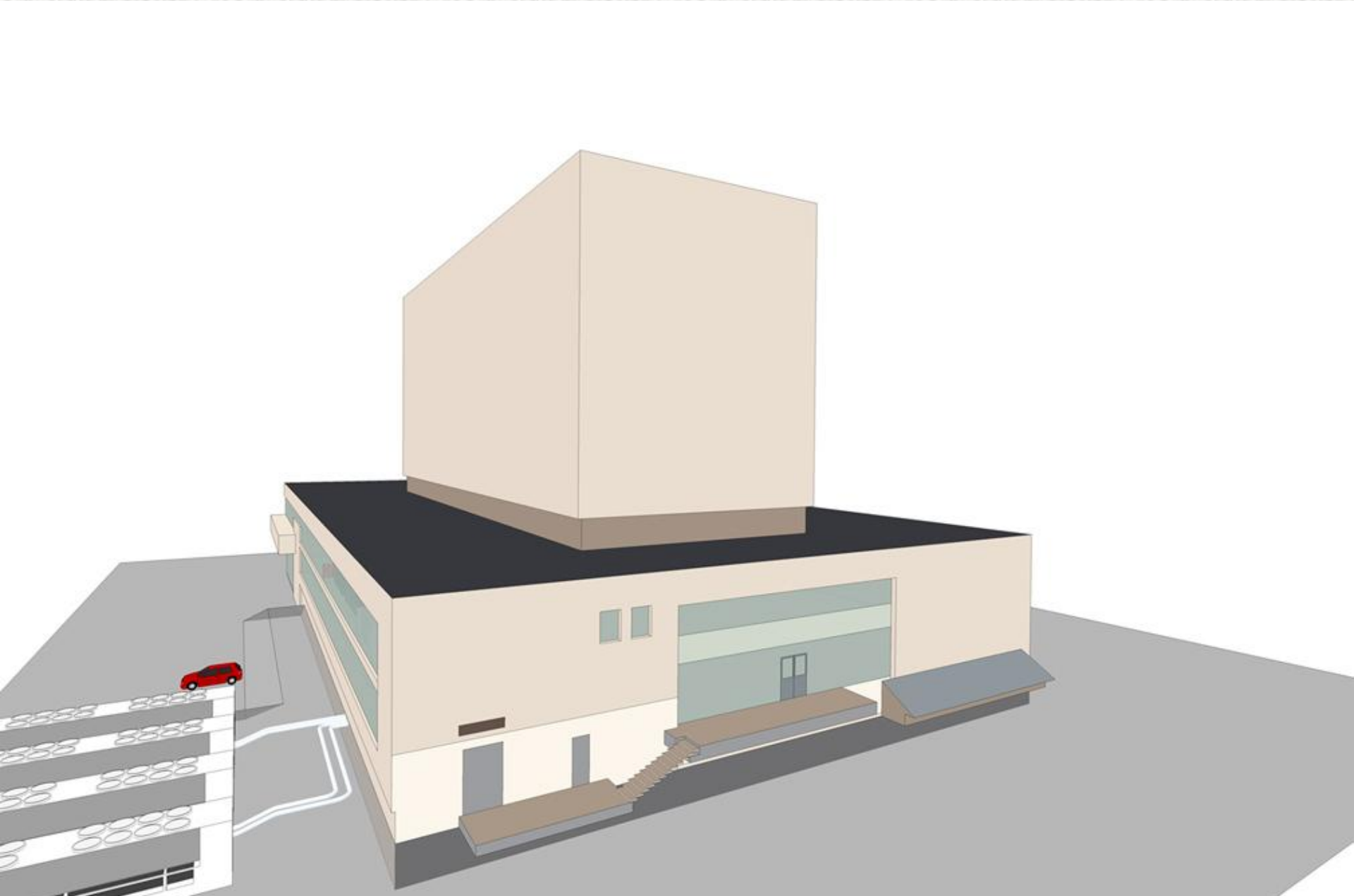
Суперкомпьютер «Ломоносов»

Схема построения Fat Tree в суперкомпьютере «Ломоносов»

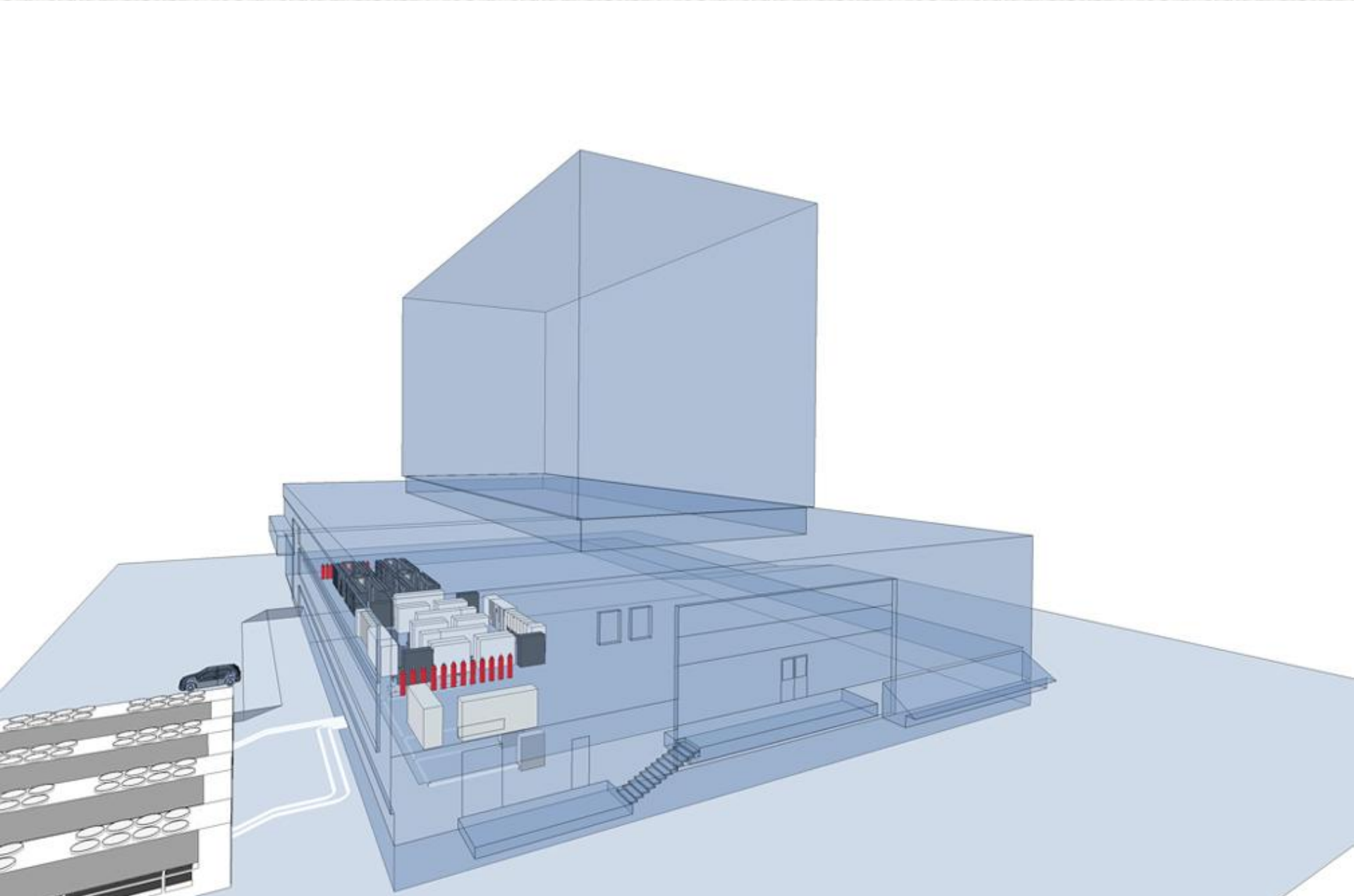


Суперкомпьютер «Ломоносов»

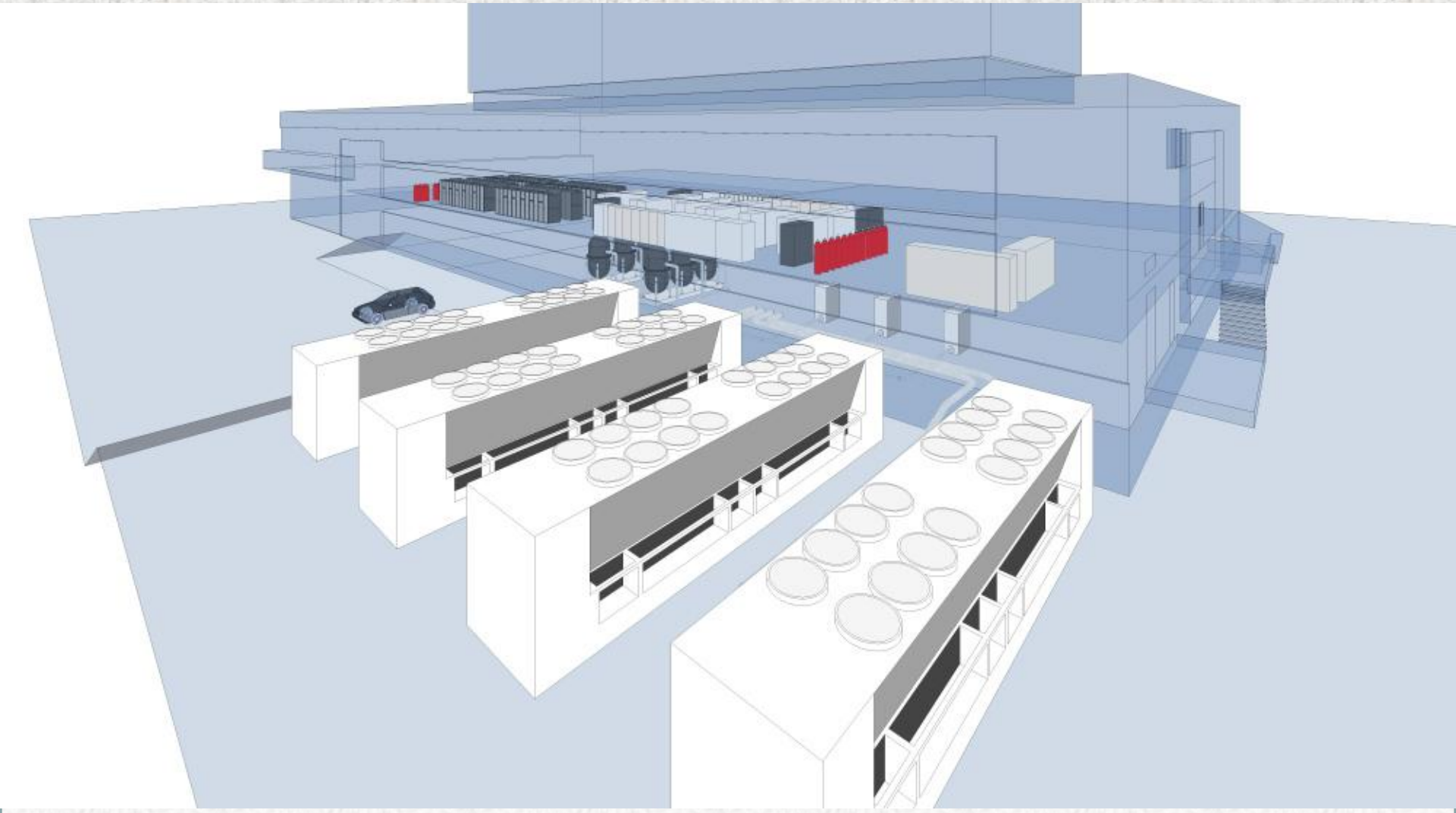
Суперкомпьютер «Ломоносов»



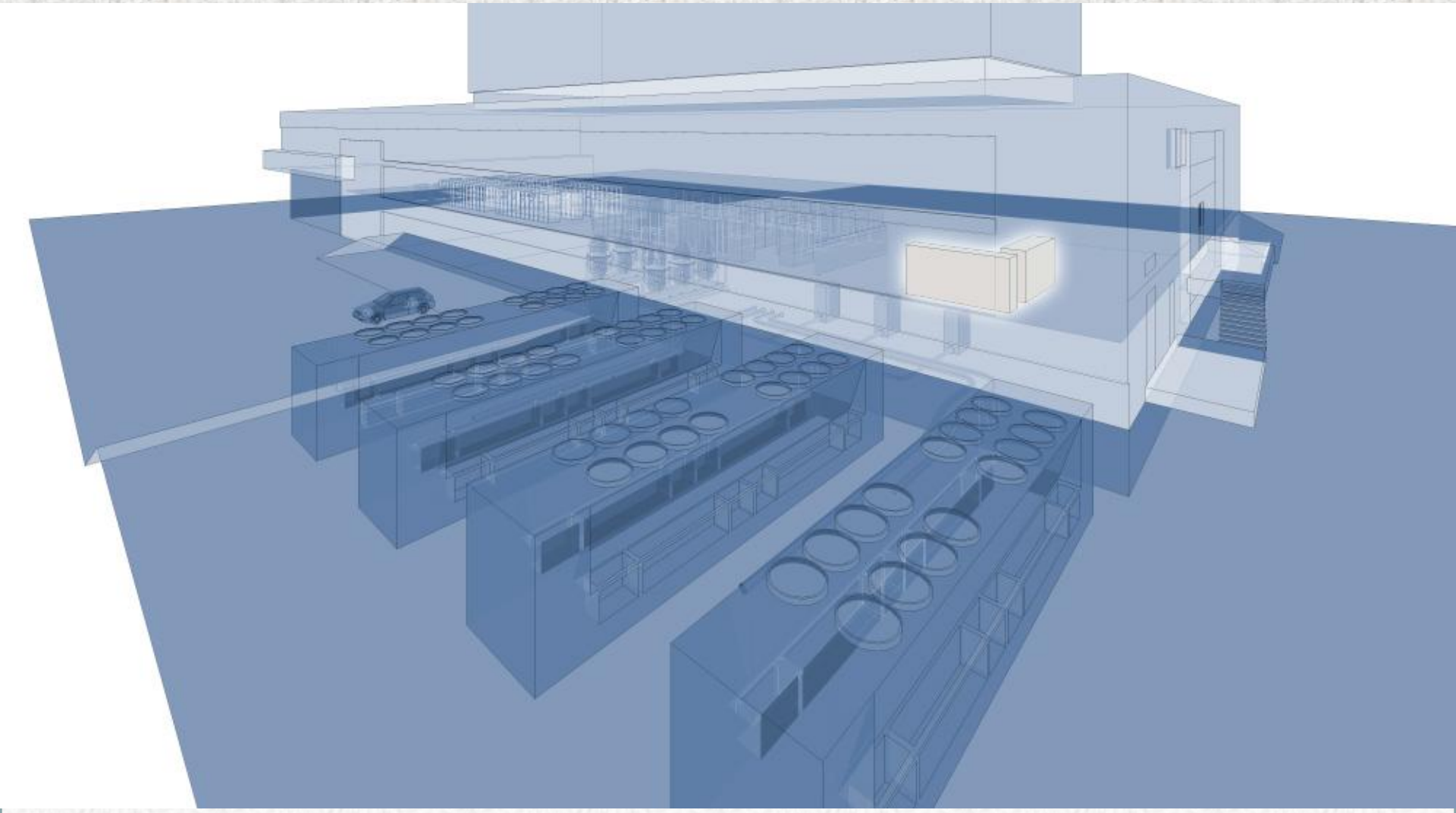
Суперкомпьютер «Ломоносов»



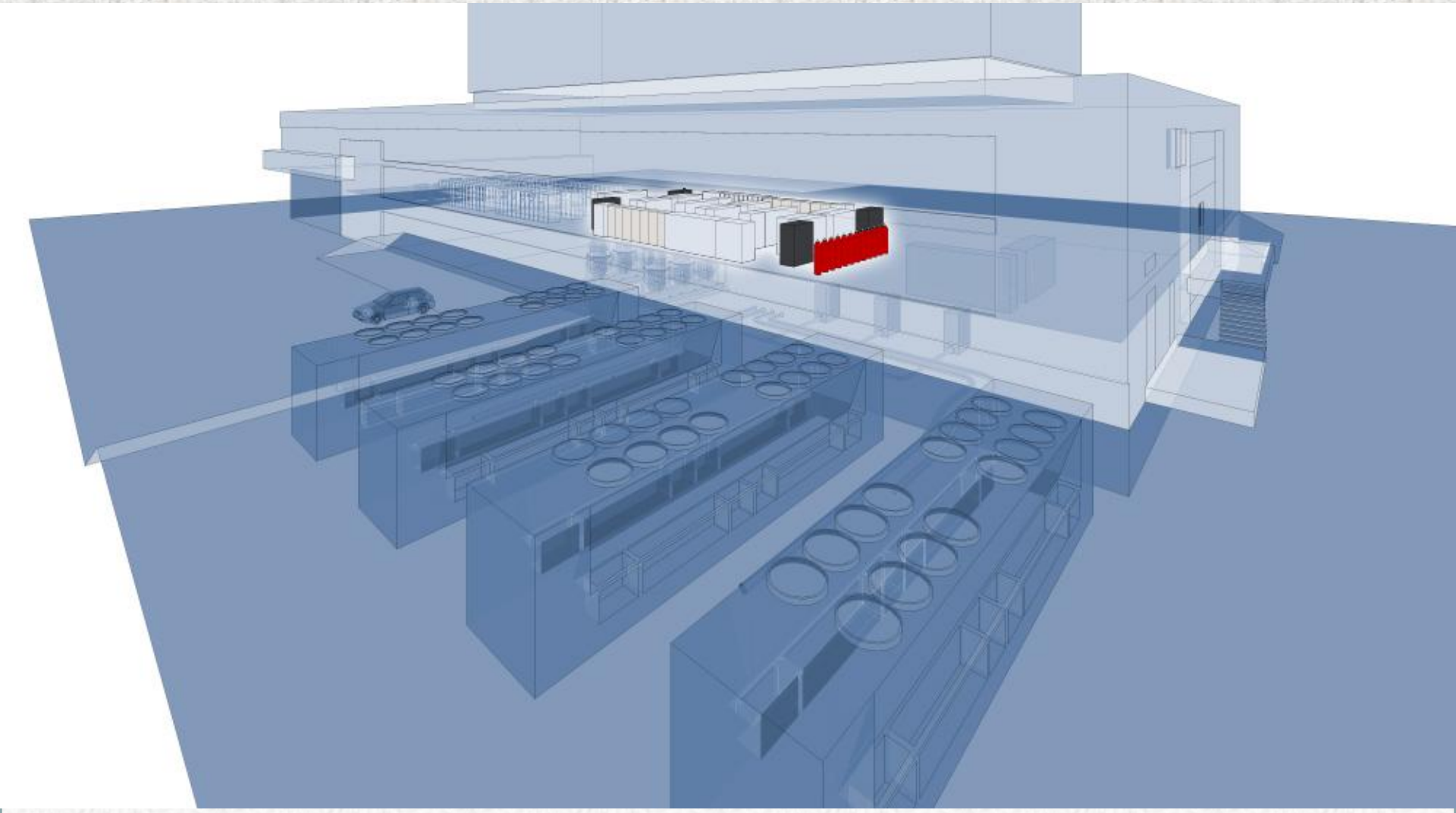
Суперкомпьютер «Ломоносов»



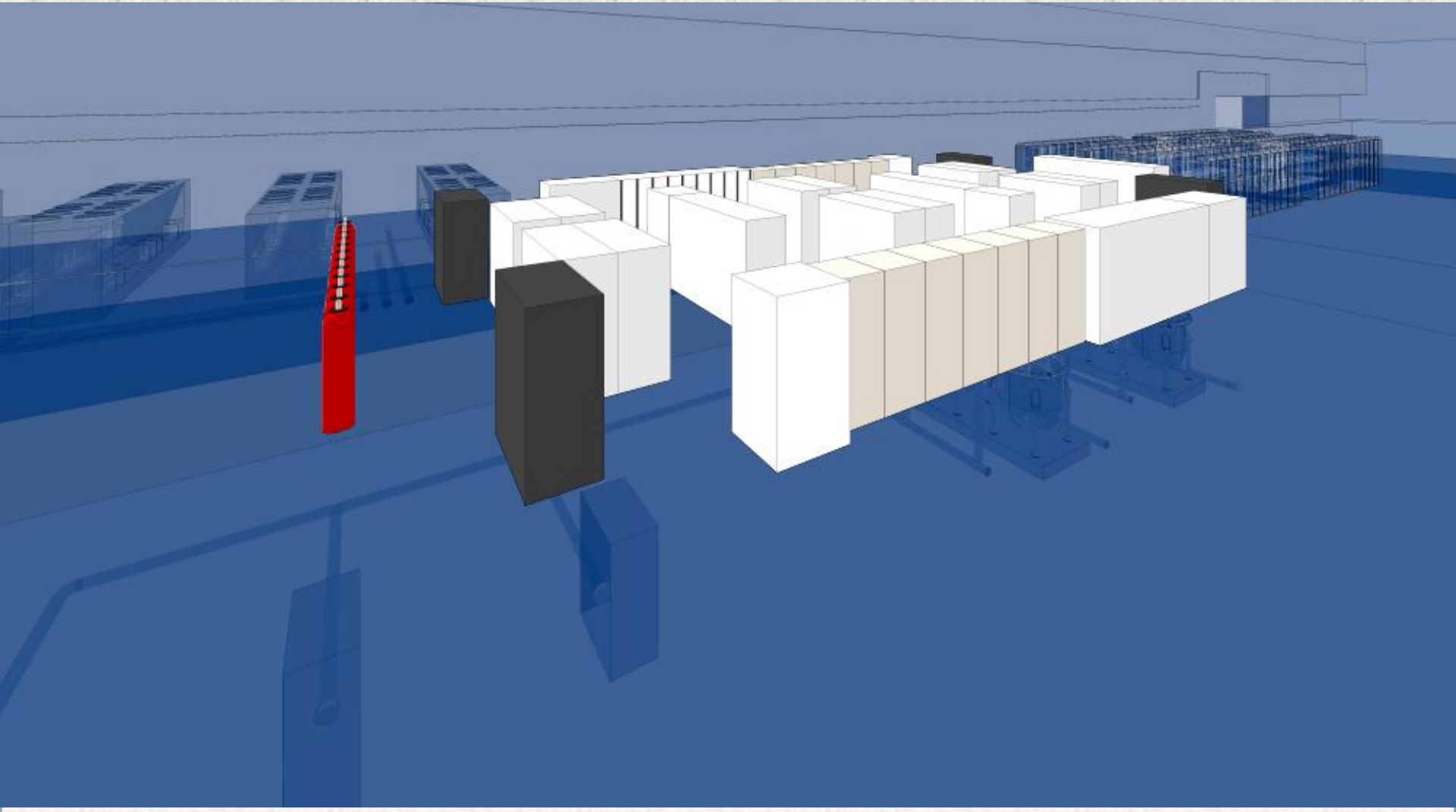
Суперкомпьютер «Ломоносов»



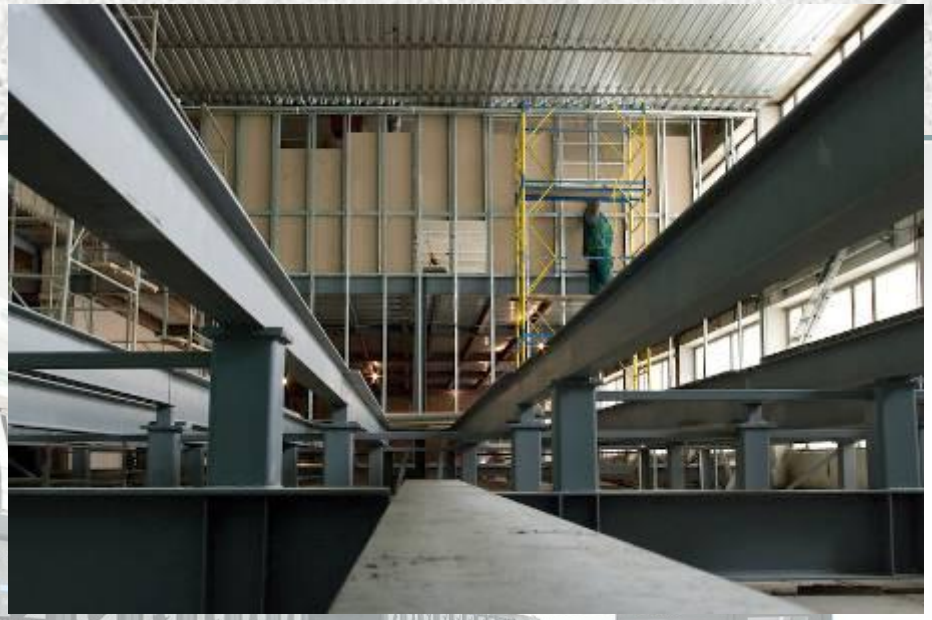
Суперкомпьютер «Ломоносов»



Суперкомпьютер «Ломоносов»



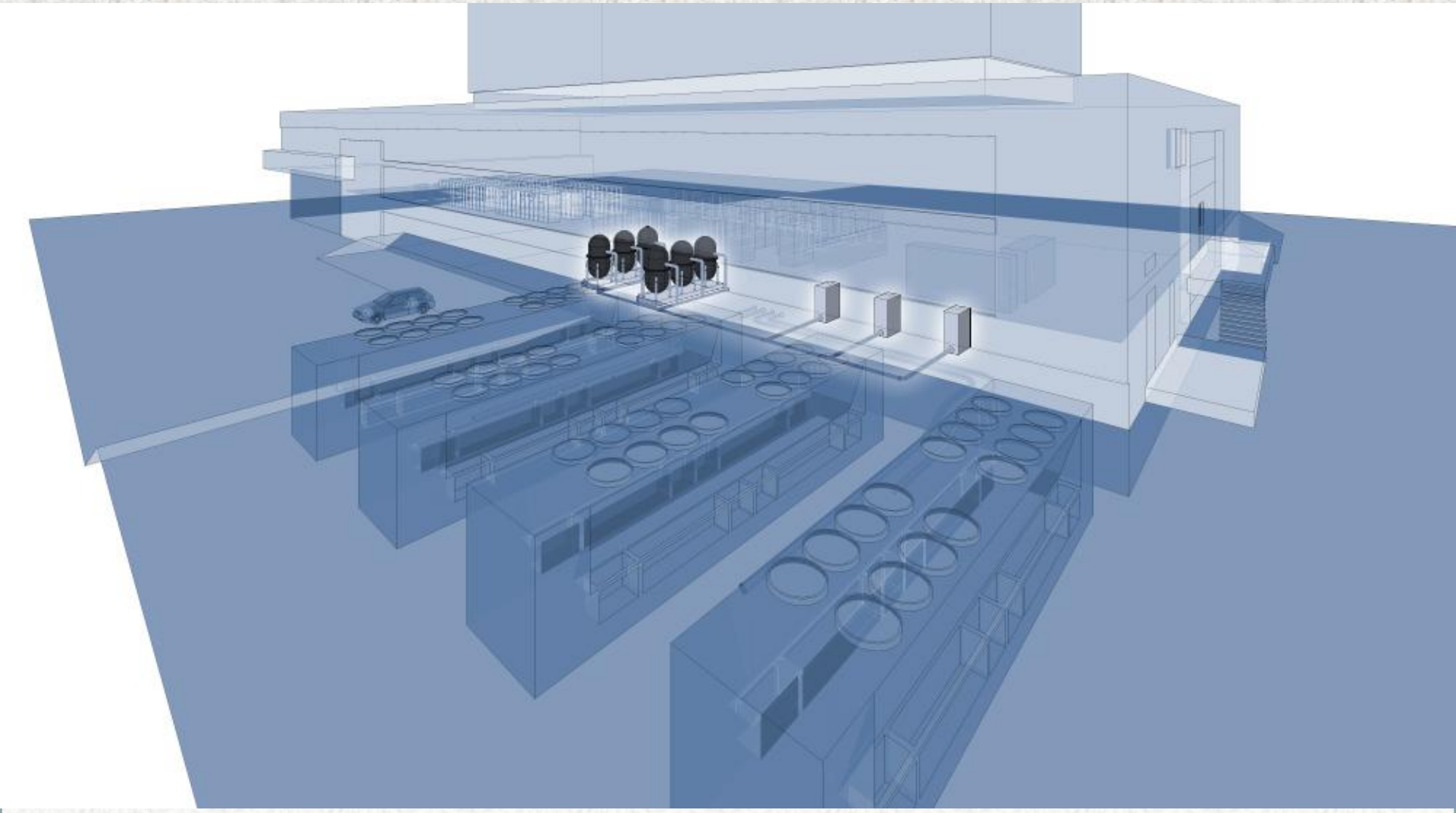
Суперкомпьютер «Ломоносов»



Суперкомпьютер «Ломоносов»

Суперкомпьютер «Ломоносов»

Суперкомпьютер «Ломоносов»



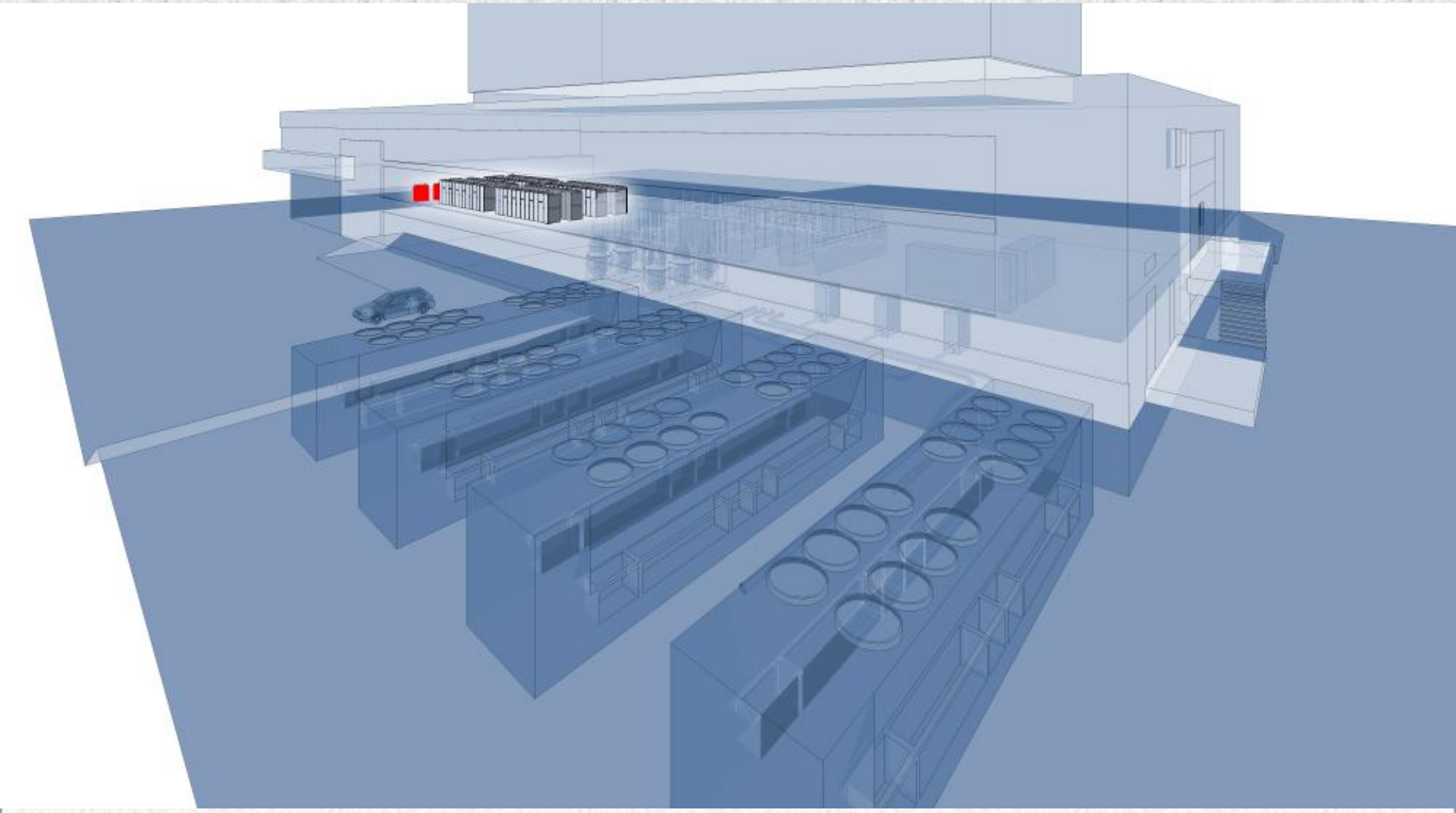
Суперкомпьютер «Ломоносов»

Суперкомпьютер «Ломоносов»

Суперкомпьютер «Ломоносов»

Суперкомпьютер «Ломоносов»

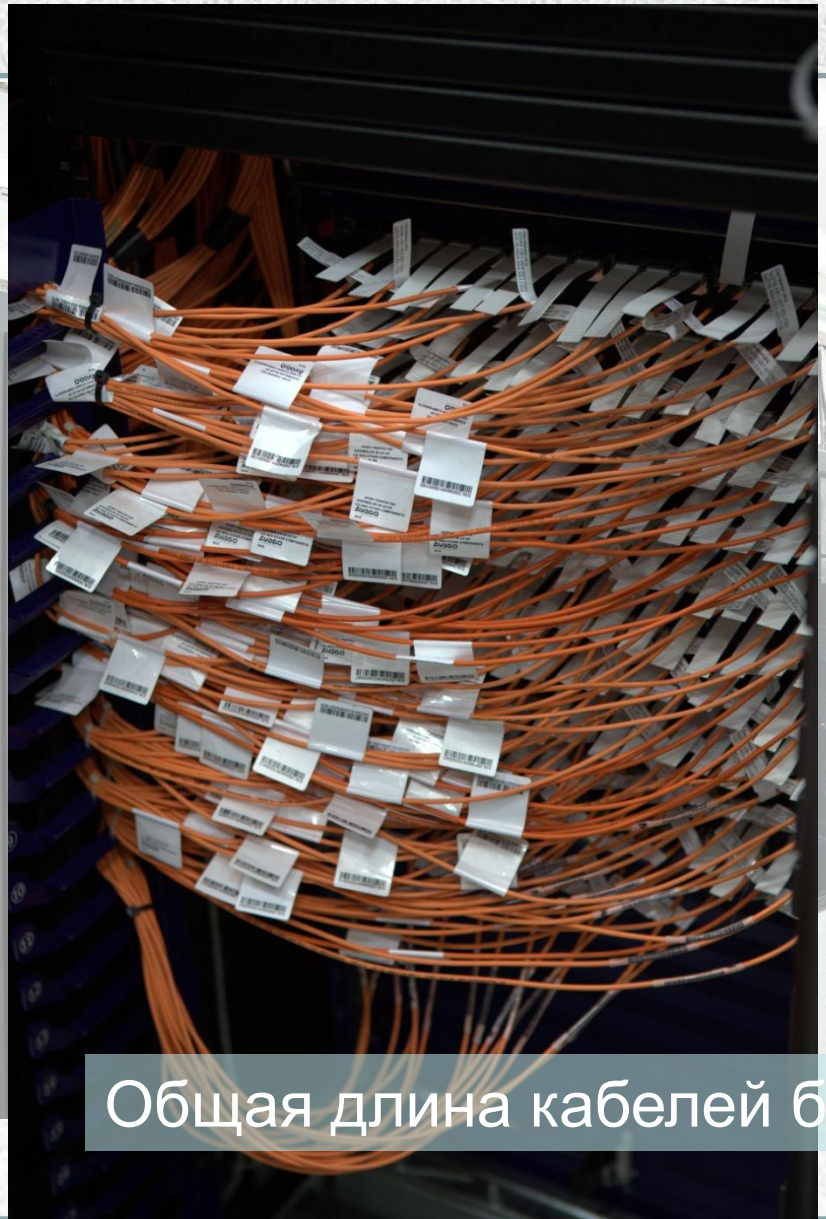
Суперкомпьютер «Ломоносов»

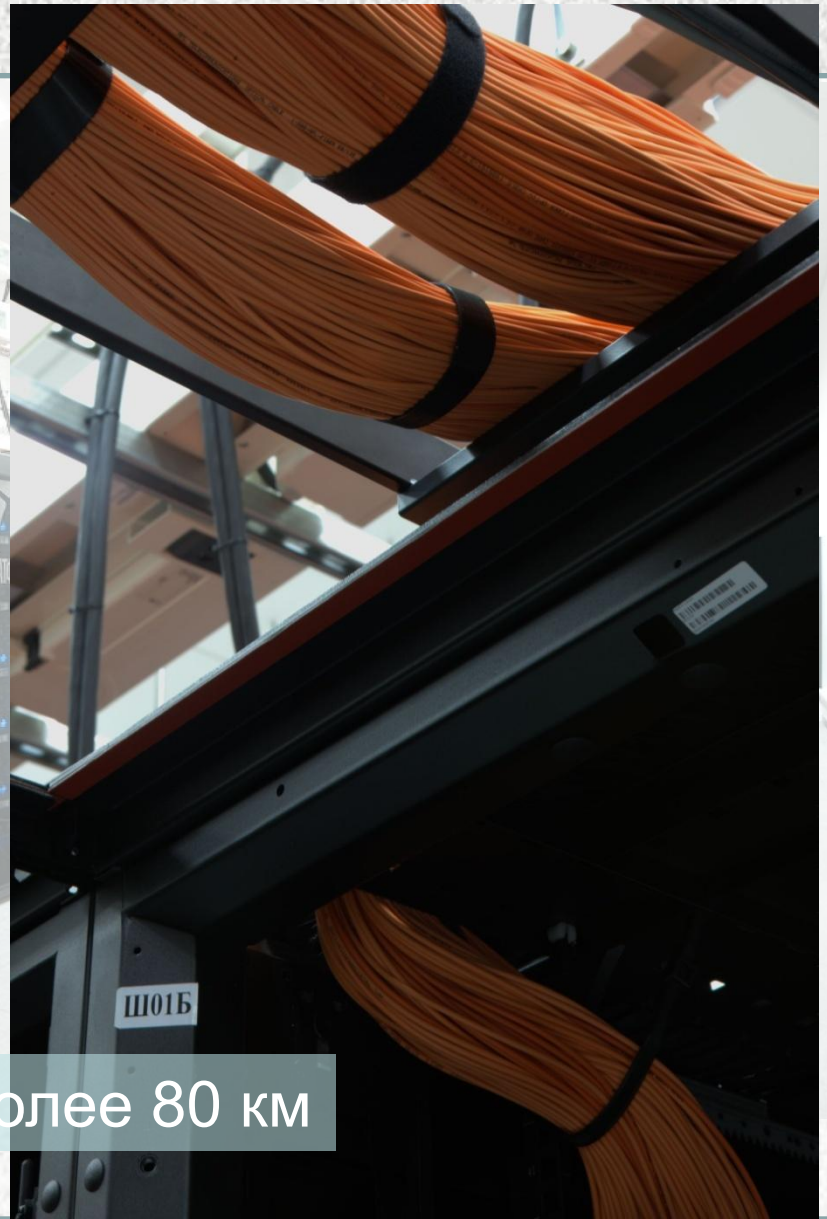


Суперкомпьютер «Ломоносов»

Вес оборудования машзала – 57 т, СБЭ – 92 т

Суперкомпьютер «Ломоносов»

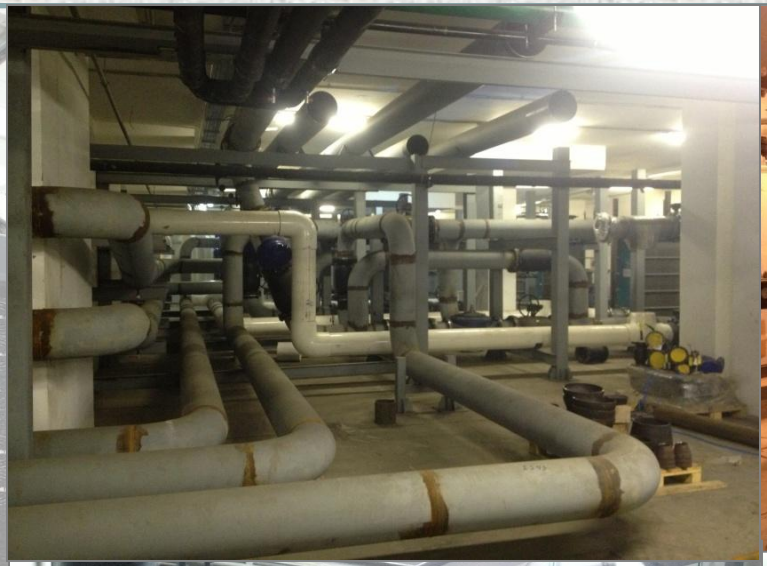




Общая длина кабелей более 80 км

Суперкомпьютер «Ломоносов»

Суперкомпьютер «Ломоносов-2» (инфраструктура)



Суперкомпьютер «Ломоносов-2»

1 стойка = 256 узлов:
Intel Xeon + NVIDIA K40 + NVIDIA P100 + NVIDIA V100
7 стоек = 5.5 PFlop/s

Суперкомпьютер «Ломоносов-2»

В.В.Путин во время визита в МГУ 30 октября 2014 года

(фотографии с сайта kremlin.ru)

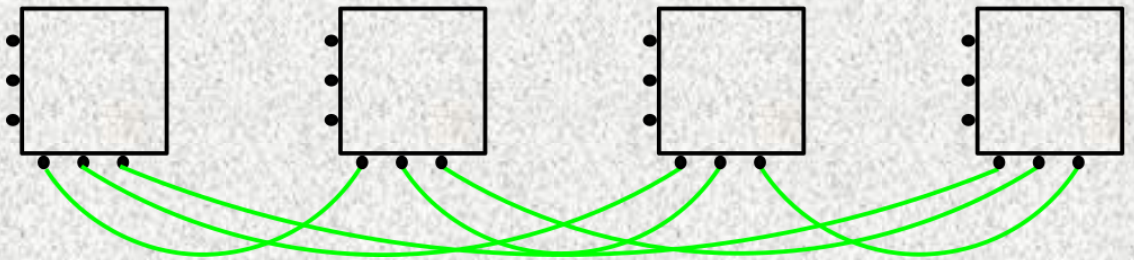
Суперкомпьютер «Ломоносов-2»

Пиковая производительность	5 505 TFlop/s
Производительность (Linpack)	2 478 TFlop/s
Эффективность	50%
Вычислительных узлов	1 722
Центральное процессоры	Intel Haswell-EP E5-2697v3, Intel Xeon Gold 6126, Intel Xeon Gold 6142, Intel Xeon Gold 6240
Ускорители	Nvidia Tesla K40M, Nvidia P100, Nvidia V100
Оперативная память	114 ТБайт
Коммуникационная сеть	FDR Infiniband / 10 GE
Система хранения данных	1392 ТБайт, Lustre, Panasas
Операционная система	CentOS 7

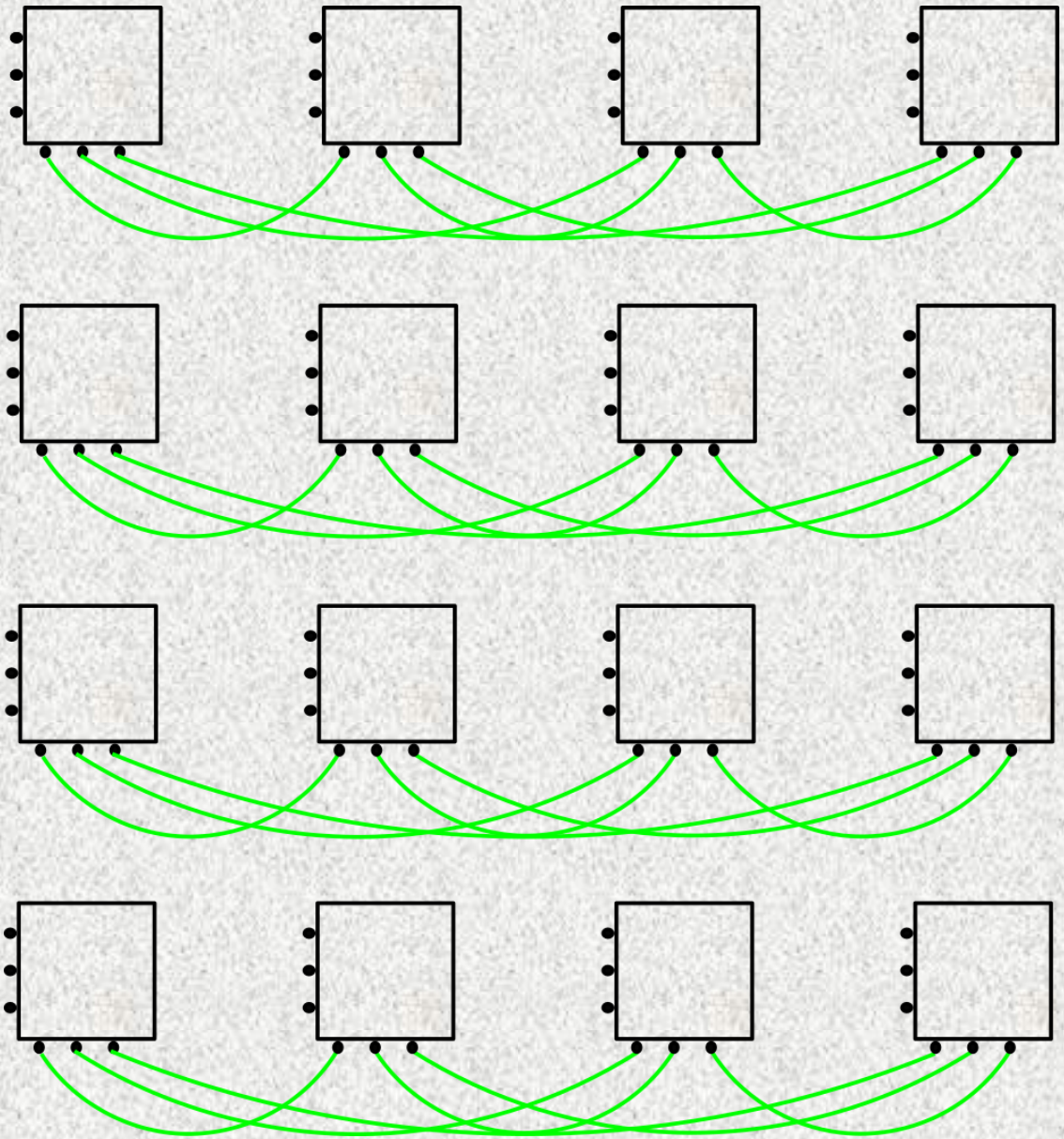
Вычислительная сеть суперкомпьютера «Ломоносов-2»

Тип связи	FDR InfiniBand (56 Gbit/s)
Топология	Flattened Butterfly
Число портов коммутатора	36
Размерность	8x8x8x4
Относительная ширина связей	1x1x1x2
Число вычислительных узлов, подключенных к одному коммутатору	8
Максимальное количество узлов	16 384

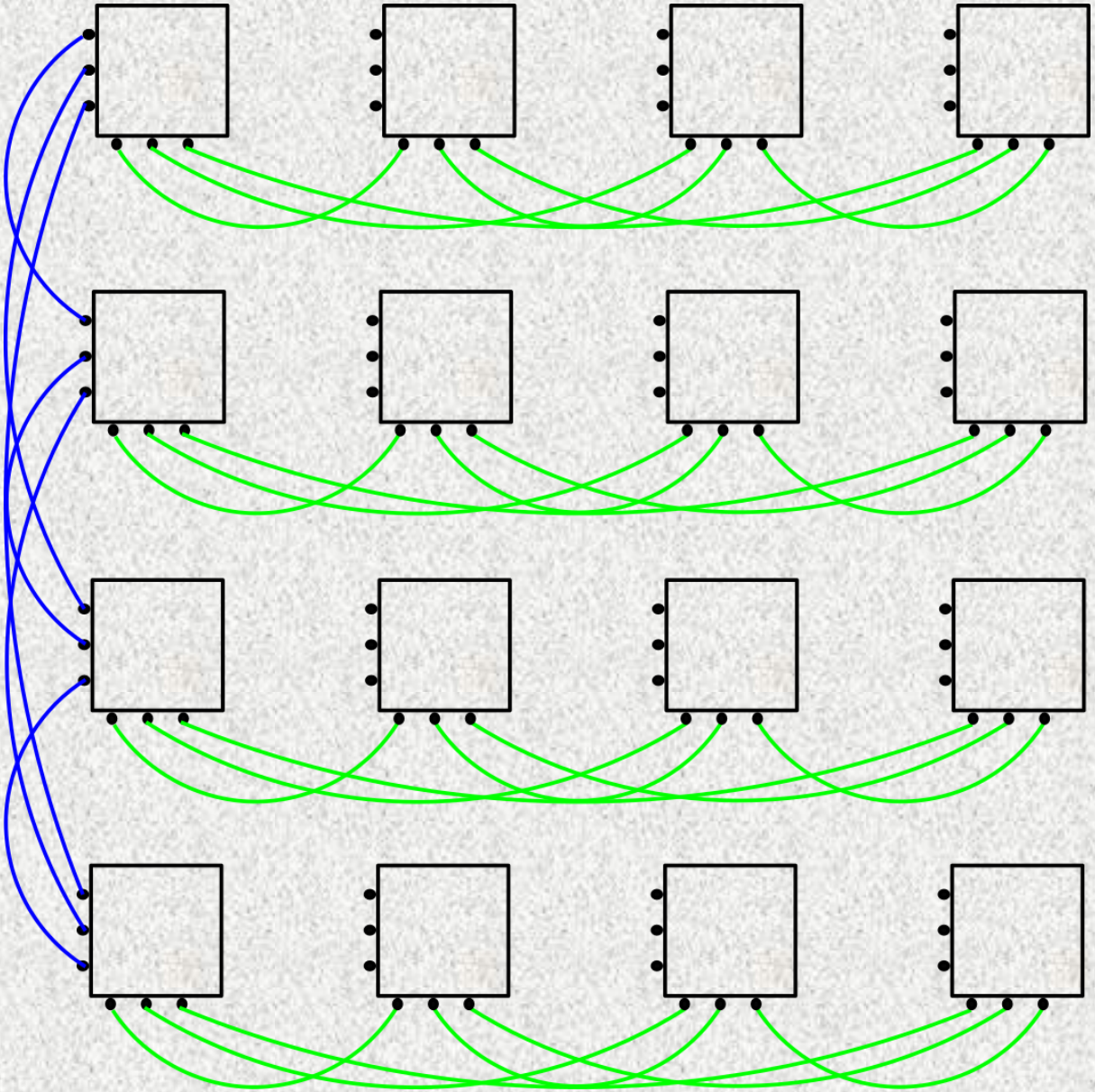
Топология Flattened Butterfly 1-D



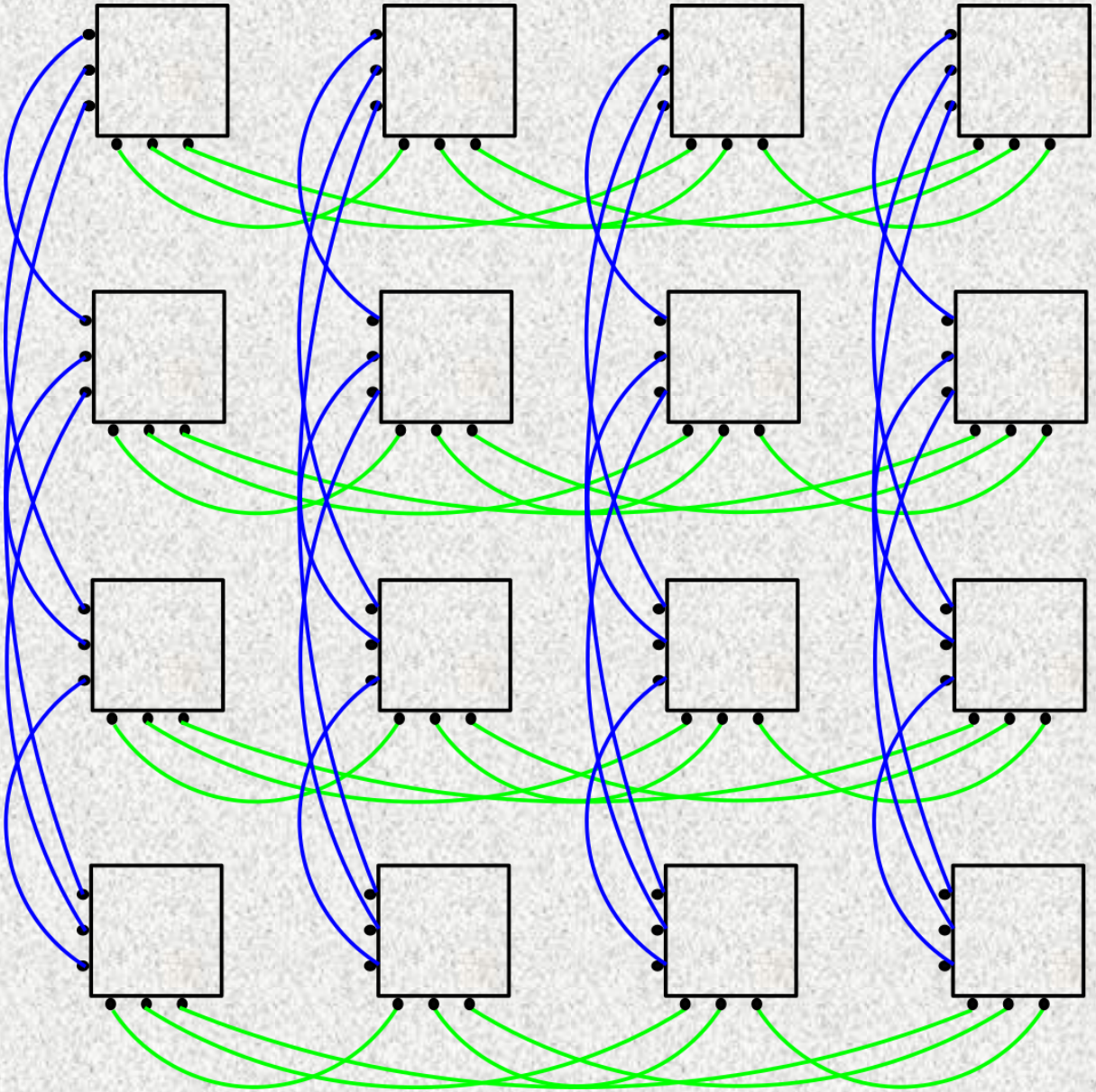
Топология Flattened Butterfly 2-D: добавляем узлы



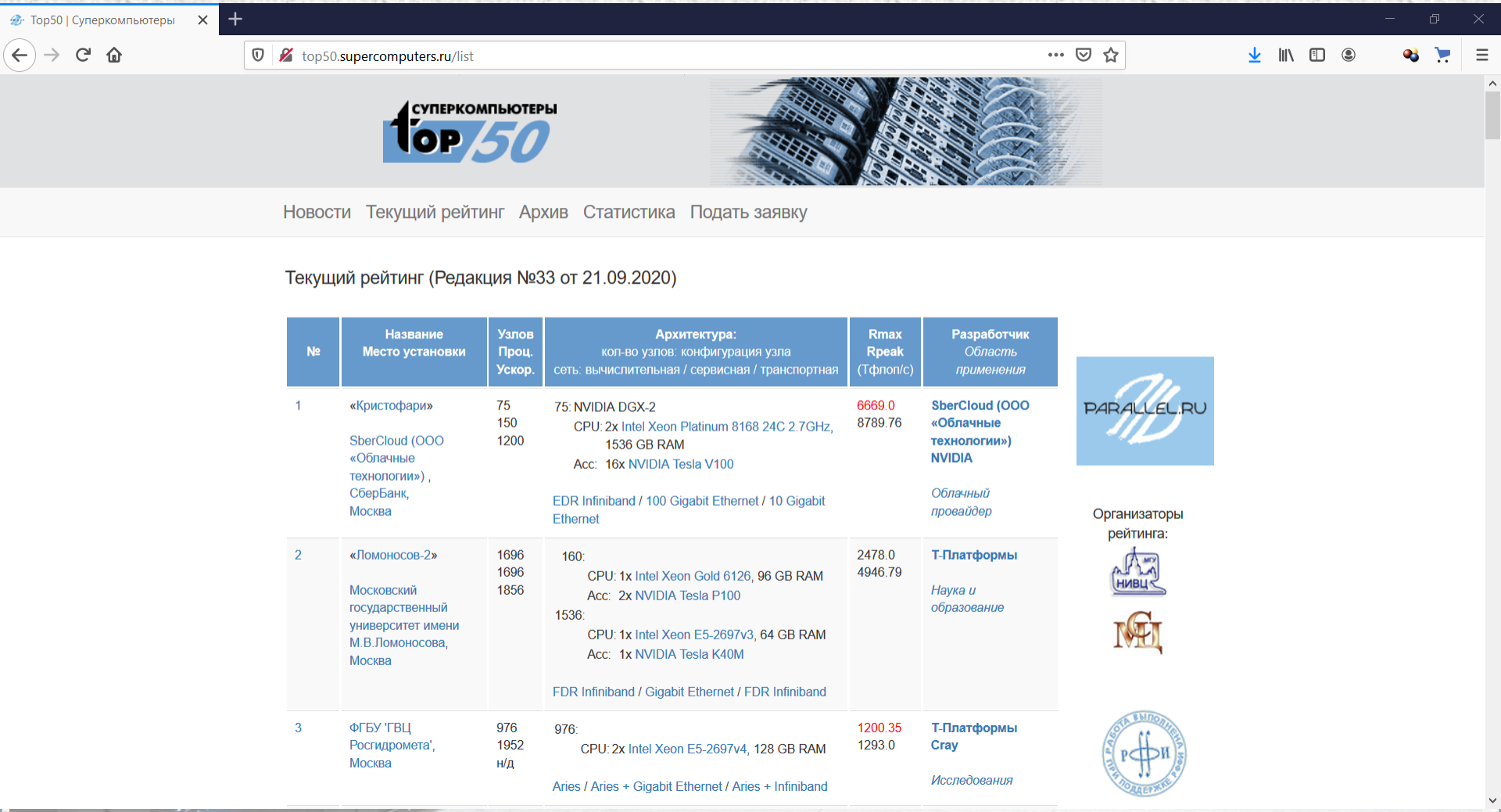
Топология Flattened Butterfly 2-D: связи по одной координате



Топология Flattened Butterfly 2-D: все связи



Суперкомпьютеры МГУ в списке Top50 самых мощных суперкомпьютеров СНГ



«Ломоносов-2» - 2 место

http://top50.supercomputers.ru

Суперкомпьютеры «Ломоносов» и «Ломоносов-2» в списке Top500 самых мощных суперкомпьютеров мира

«Ломоносов»

Дата	Производительность	Место
Ноябрь 2009 г.	0.4 PFlop/s	12
Ноябрь 2018 г.	1.7 PFlop/s	485
Июнь 2020 г.	1.7 PFlop/s	-

«Ломоносов-2»

Дата	Производительность	Место
Ноябрь 2014 г.	2.6 PFlop/s	22
Июнь 2020 г.	4.9 PFlop/s	130

«Нужно бежать со всех ног, чтобы только оставаться на месте, а чтобы куда-то попасть, надо бежать как минимум вдвое быстрее!»
(Льюис Кэрролл, «Алиса в зазеркалье»)

http://top500.org

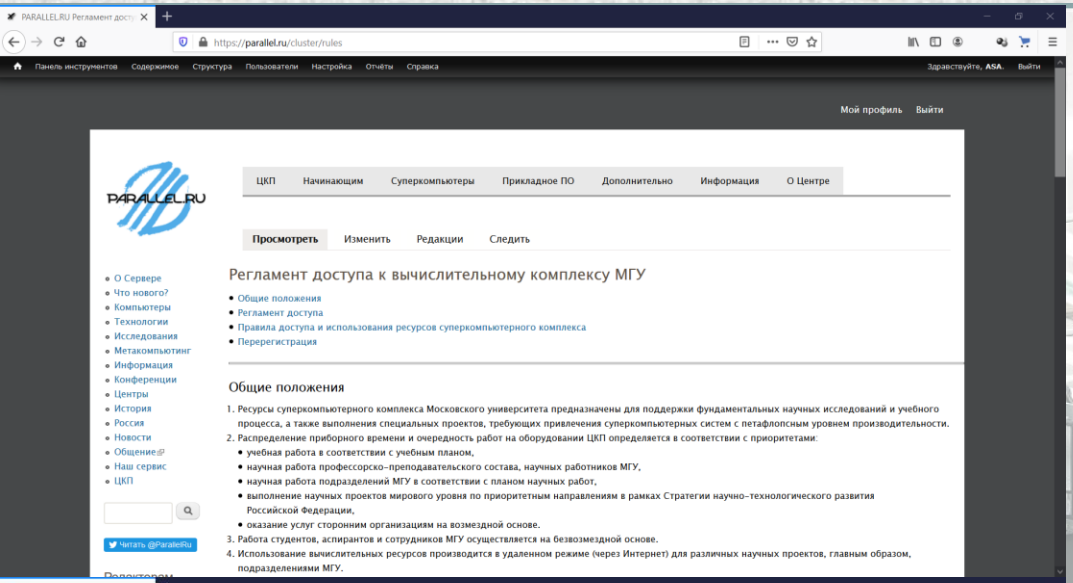
Суперкомпьютер «Ломоносов-2» - основной вычислительный комплекс России

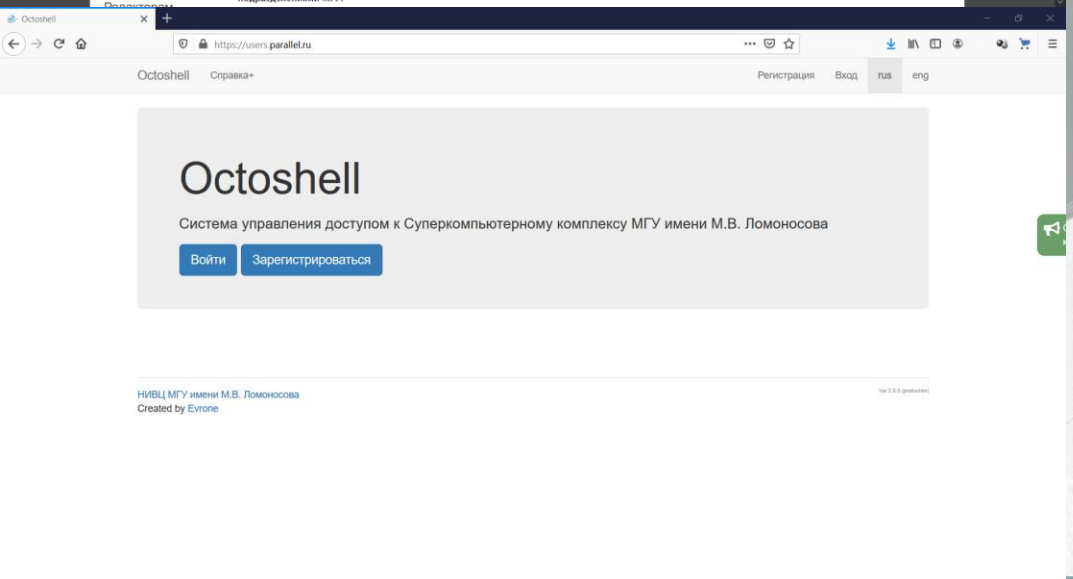
Распределение приложений пользователей по областям науки

- ✓ ***880 проектов***
- ✓ ***2955 пользователей***

- ✓ ***24 факультета и института МГУ***
- ✓ ***102 университета России***
- ✓ ***95 институтов РАН***

Доступ к системам Суперкомпьютерного комплекса МГУ имени М.В.Ломоносова





РОССИЙСКАЯ АКАДЕМИЯ НАУК

ИНСТИТУТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ
(ИВМ РАН)

Ул. Губкина, д.8, Москва 119333
Тел.: 938-17-69, 938-37-86. Факс 938-18-21
E-mail: director@inm.ras.ru, http://www.inm.ras.ru
ОКПО 04742320, ОГРН 1027700542363
ИНН/КПП 7736060033/773601001

06.10.2008 № 10256/225
на № _____ от _____

Ректору
Московского государственного
университета
имени М.В. Ломоносова
академику В.А. Садовничему

Глубокоуважаемый Виктор Антонович!

Просим Вас рассмотреть вопрос о предоставлении доступа на суперкомпьютерный комплекс НИВЦ МГУ сотрудникам Института вычислительной математики РАН

Володину Евгению Михайловичу
Глазунову Андрею Васильевичу

Доступ необходим для решения вычислительных задач, связанных с моделированием климата и его изменений, а также задач по моделированию турбулентности в пограничном слое атмосферы.

Гарантируем использование предоставленных ресурсов только для указанных задач и полное соблюдение правил работы на суперкомпьютерном комплексе НИВЦ МГУ, опубликованных по адресу http://parallel.ru/cluster/rules/.

Директор ИВМ РАН академик Дымников В.П.
6 октября 2008г.

Суперкомпьютер «Ломоносов»

Виртуальная экскурсия по суперкомпьютеру «Ломоносов»:
http://lom1.virtual-hpc.parallel.ru

Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ (лекция 6)

А.С.Антонов

Вед. н.с. НИВЦ МГУ, к.ф.-м.н.

asa@parallel.ru

ВМК МГУ, 2020

Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ (лекция 7)

А.С.Антонов

Вед. н.с. НИВЦ МГУ, к.ф.-м.н.

asa@parallel.ru

ВМК МГУ, 2020

Векторно-конвейерные компьютеры

Векторно-конвейерные компьютеры

Функциональное устройство:

- *скалярное;*
- *конвейерное (операция делится на несколько микроопераций, количество микроопераций определяет число ступеней конвейера).*

Команда:

- *скалярная (все аргументы - скаляры);*
- *векторная (например, сложить все элементы массива x с числом b).*

Компьютер:

- *скалярный;*
- *векторный;*
- *конвейерный.*

Векторно-конвейерные компьютеры

(векторизация, векторы данных, независимые операции)

Процесс поиска подходящих фрагментов в программе и их замена векторными командами - ***векторизация программы****.*

Пример векторизуемого фрагмента:
for(i=0; i<n; i++) c[i] = a[i]+b[i];

Компилятор сгенерирует последовательность векторных команд:

- *загрузка векторов a и b из памяти в векторные регистры,*
- *векторная операция сложения,*
- *запись содержимого векторного регистра в память.*

Если весь фрагмент программы удалось заменить векторными командами, то говорят о его ***полной векторизации****. В противном случае мы имеем дело с частичной векторизацией или невозможностью векторизации фрагмента вовсе.*

Истоки векторности: *линейная организация памяти, использование массивов.*

Векторно-конвейерные компьютеры

(векторизация, векторы данных, независимые операции)

Условия векторизации:

- *наличие векторов-аргументов;*
- *над всеми элементами векторов должны выполняться одинаковые, независимые операции, для которых существуют аналогичные векторные команды в системе команд компьютера.*

Вектор *- упорядоченный набор однотипных данных, все элементы которого размещены в памяти компьютера с одинаковым смещением друг относительно друга.*

Простейшие примеры векторов:

- *одномерные массивы;*
- *строки и столбцы матриц;*
- *диагональ квадратной матрицы;*
- *многомерный массив целиком.*

Не вектор - поддиагональная часть двумерной матрицы.

Векторно-конвейерные компьютеры

(векторизация, векторы данных, независимые операции)

В векторизуемом фрагменте могут использоваться и простые переменные.

for(i=0; i<n; i++) b[i] = a[i]+s;

*Основные кандидаты для векторизации - **самые внутренние циклы** всех циклических конструкций программы.*

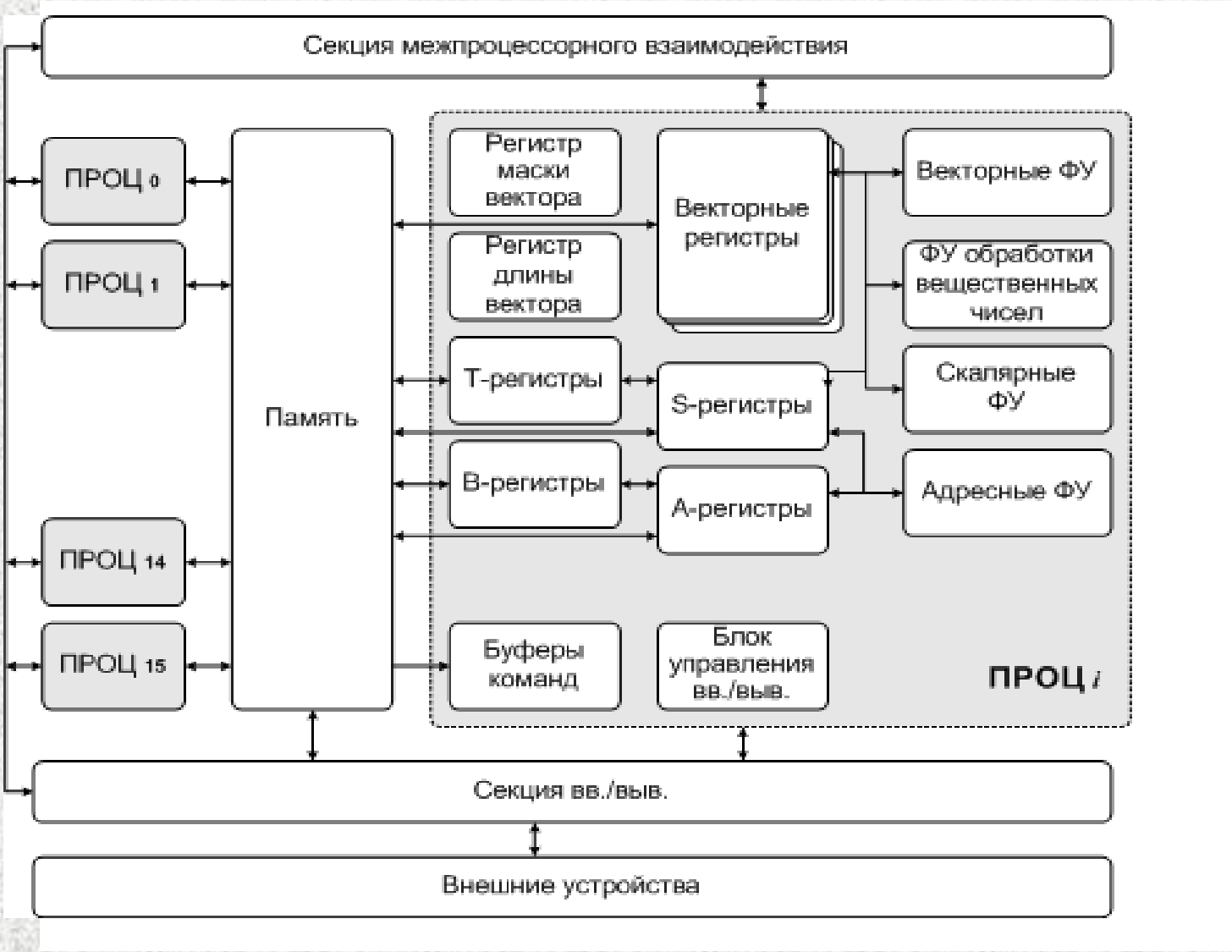
for(i=0; i<n; i++) a[i] = a[i-1]+b[i];

*Необходимо **отсутствие информационной зависимости** между операциями!*

for(i=0; i<n; i++) a[i] = f(a[i], b[i]);

Неизвестно, какая операция соответствует вызову функции f, и, следовательно, на какие векторные команды можно заменить данный фрагмент.

Векторно-конвейерные компьютеры



Общая схема суперкомпьютера CRAY C90 (1991 г.)

Функциональные устройства

- *Все функциональные устройства конвейерные.*

- *Число ступеней различно, однако каждая ступень каждого устройства всегда срабатывает за один такт.*

- *Все функциональные устройства независимы и могут работать одновременно друг с другом.*

- *Функциональные устройства данного компьютера делятся на четыре группы:*
  - ✓*адресные – операции над 32-разрядными целыми числами;*
  - ✓*скалярные – операции над 64-разрядными целыми числами;*
  - ✓*для векторных команд – операции над целочисленными векторами;*
  - ✓*функциональные устройства для выполнения вещественной арифметики – операции над 64-разрядными числами в форме с плавающей запятой (как в векторном, так и в скалярном режиме).*

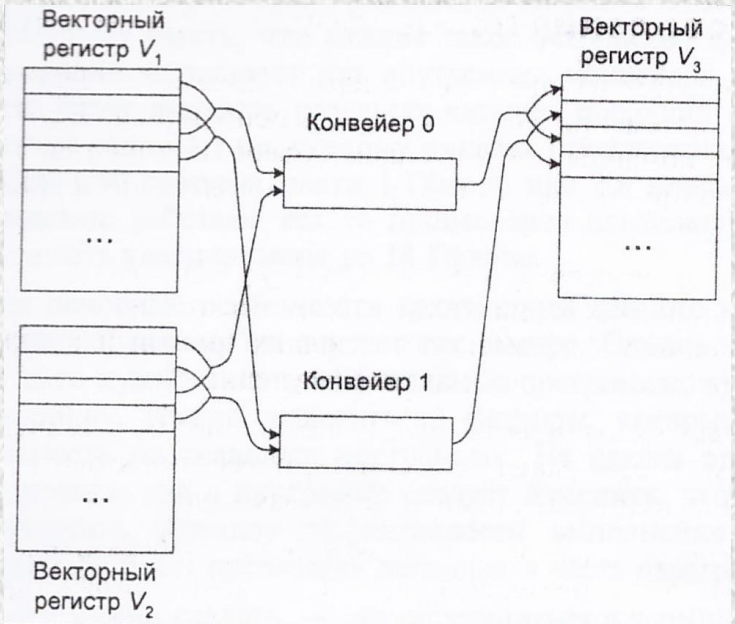
Время разгона конвейера и секционирование векторных команд

```
for(i = 0; i < n; i++)
    a[i] = b[i]*s+c[i];
```

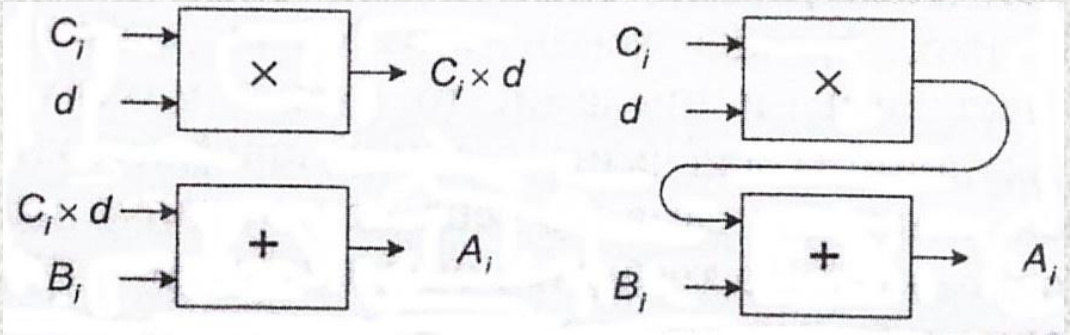
Длина вектора	Производительность (Mflops)
1	7.0
4	27.6
32	181.9
128	433.7
129	364.3
256	548.0
257	491.0
8192	802.0

Производительность CRAY C90 на операции $a_i = b_i*s+c_i$.

Дублирование и зацепление векторных устройств



Дублирование векторных устройств и устройств для вещественной арифметики



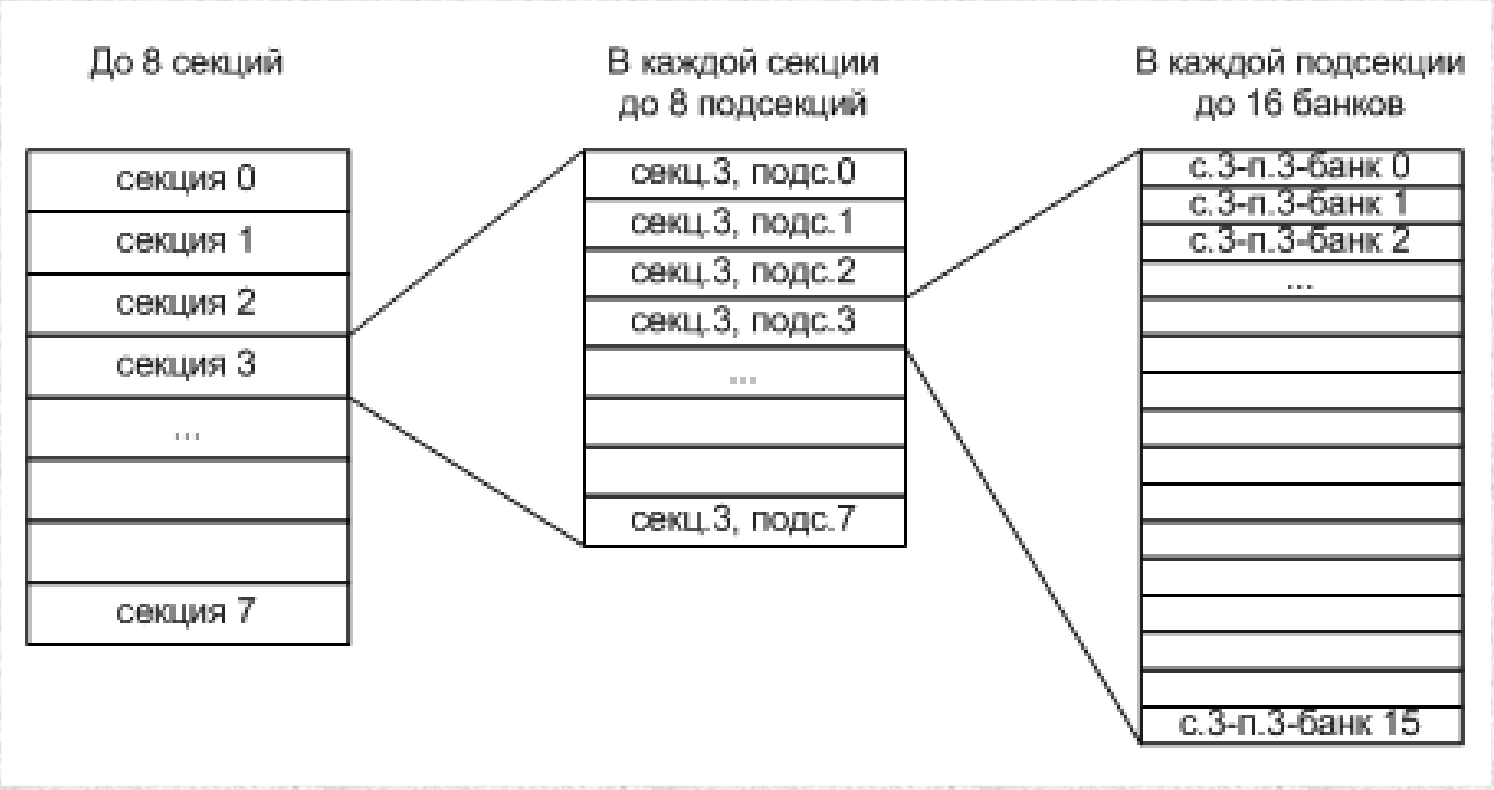
Зацепление векторных операций:

x - l_1 ступеней
+ - l_2 ступеней

- *Без зацепления: (l_1+n–1) + (l_2+n–1) = l_1+l_2+2*n–2 тактов*

- *С зацеплением: l_1+l_2+n–1 тактов*

Конфликты в памяти



- *При одновременном обращении к одной и той же секции возникает конфликт, который разрешается за 1 такт. В этом случае один из запросов продолжает обрабатываться, а другой просто блокируется на один такт.*
- *Если происходит одновременное обращение к одной и той же подсекции одной секции, то время на разрешение конфликта уже может достигать 6 тактов.*

Конфликты в памяти

Последовательные адреса идут с чередованием :

адрес 0 — в 0-й секции, 0-й подсекции, 0-м банке;
адрес 1 — в 1-й секции, 0-й подсекции, 0-м банке;
адрес 2 — в 2-й секции, 0-й подсекции, 0-м банке;
...
адрес 8 — в 0-й секции, 1-й подсекции, 0-м банке;
адрес 9 — в 1-й секции, 1-й подсекции, 0-м банке;
...
адрес 63 — в 7-й секции, 7-й подсекции, 0-м банке;
адрес 64 — в 0-й секции, 0-й подсекции, 1-м банке;
адрес 65 — в 1-й секции, 0-й подсекции, 1-м банке;
...

Максимальное число конфликтов будет происходить при постоянном обращении к одной и той же подсекции одной и той же секции. Это заведомо произойдет, например, при выполнении процессором векторной операции чтения данных, расположенных с шагом, кратным 64.

Конфликты в памяти

- *Операции чтения/записи последовательно расположенных данных проходят без возникновения конфликтов. В частности, все одномерные массивы будут обрабатываться именно так.*

- *При выборке данных с любым нечетным шагом конфликтов также не возникнет.*

- *Чем б**о**льшая степень двойки присутствует в качестве сомножителя в шаге выборки данных, тем больше времени требуется на разрешение возникающих конфликтов.*

Влияние конфликтов в памяти на производительность

```
for(i = 0; i < n*k; i +=k)
  a[i] = b[i]+c[i]*d;
```

Шаг по памяти	Производительность (Mflops)
1	705.2
2	444.6
4	274.6
8	142.8
16	84.5
32	44.3
64	22.7
128	22.6

Производительность CRAY C90 на операции $a_i = b_i + c_i * d$, n=1000

Влияние конфликтов в памяти на производительность
(многообразие проявления конфликтов)

```
Do i = 1, n
   Do j = 1, n
      Do k = 1, n
         X(i,j,k) = X(i,j,k) + P(k,i)* Y(k,j)
```

X(N1,N2,N3)

Фортран: хранение массивов “по столбцам”:

X(i,j,k) ≈ X(i+1,j,k) расстояние 1,

X(i,j,k) ≈ X(i,j+1,k) расстояние N1,

X(i,j,k) ≈ X(i,j,k+1) расстояние N1*N2.

Реальное описание массива: X(40,40,N)

X(i,j,k) X(i,j,k+1) расстояние 40*40 = 1600 = 64*25.

Решение? Описание массива X(41,41,N)

Каналы процессор-память

Три канала передачи данных, два из которых могут работать на чтение из памяти, а третий на запись.

Длина вектора	Производительность (Mflops)
10	57.0
100	278.3
1000	435.3
12801	445.0

Производительность CRAY C90 на операции

$a_i = b_i * c_i + d_i$

Операции чтения/записи в векторные регистры, ограниченное число векторных регистров

Раскрутка цикла:

```
for(j = 1; j <= 120; j++)
    for(i = 1; i <= n; i++)
        d[i] = d[i]+s*p[i][j-1]+t*p[i][j];
```

*120*3=360 операций чтения*
120 операций записи

```
for(j = 1; j <= 120; j+=2)
    for(i = 1; i <= n; i++)
        d[i] = d[i]+s*p[i][j-1]+t*p[i][j]+
                   s*p[i][j]+t*p[i][j+1];
```

*60*4=240 операций чтения*
60 операций записи

Глубина раскрутки	Производительность (Mflops)
1	612.9
2	731.6
3	780.7
4	807.7

Зависимость производительности CRAY C90 от глубины раскрутки

Несбалансированное использование устройств, отсутствие операции деления

Длина вектора	Производительность (Mflops)			
	$a_i=b_i+c_i$	$a_i=b_i/c_i$	$a_i=s/b_i+t$	$a_i=s/b_i\times t$
10	35.5	24.8	49.7	46.1
100	202.9	88.4	197.4	166.5
1000	343.8	117.2	283.8	215.9

Производительность CRAY C90 на различных операциях

Пользователь: почему?

$A_{ijk} = A_{i-1jk} + B_{jk} + B_{jk}$, i=1,40; j=1,40; k=1,1000

Cray C90, пиковая производительность 960 Mflop/s

```
do k = 1, 1000
   do j = 1, 40
      do i = 1, 40
         A(i,j,k) = A(i-1,j,k)+B(j,k)+B(j,k)
```

Производительность: **20** Mflop/s на Cray C90

Пользователь: почему?

$A_{ijk} = A_{i-1jk} + B_{jk} + B_{jk}$, i=1,40; j=1,40; k=1,1000

Cray C90, пиковая производительность 960 Mflop/s

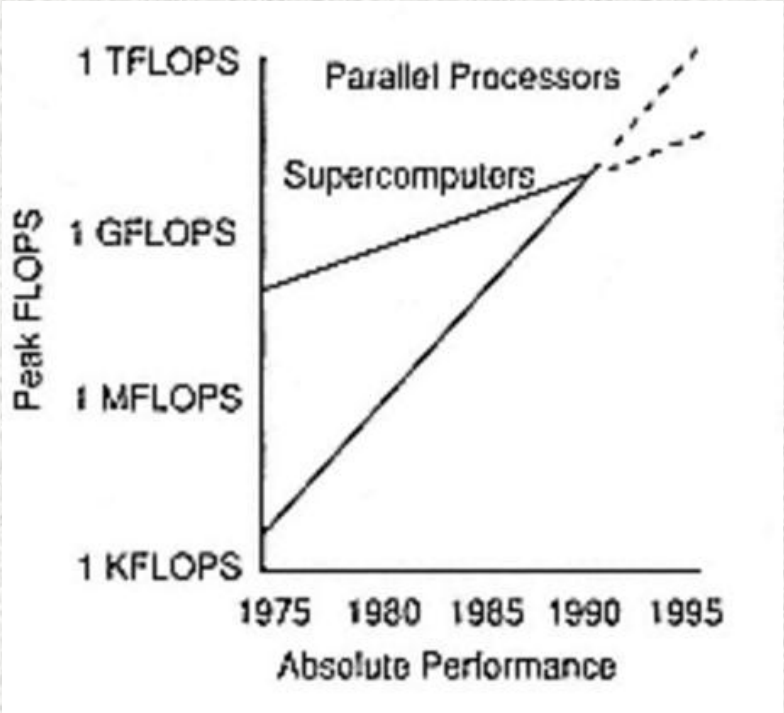
```
do i = 1, 40, 2
   do j = 1, 40
      do k = 1, 1000
         A(i,j,k) = A(i-1,j,k)+2*B(j,k)
         A(i+1,j,k) = A(i,j,k)+2*B(j,k)
```

Производительность: **700** Mflop/s на Cray C90

Векторно-конвейерные компьютеры

- Предсказывалось, что векторные компьютеры устареют к 1990 году.
- Основная причина – быстрый прогресс в производительности микропроцессоров. Векторные инструкции в микропроцессорах не использовались.

- Что ещё привело к такому положению дел?
  I. Сокращение государственных расходов США на суперкомпьютеры (после 1989 года).
  II. Распространение микропроцессоров Intel и AMD -> большие объёмы продаж -> много средств на исследование и разработку.



Векторно-конвейерные компьютеры

- Возвращение векторности по причине эволюции микропроцессоров и появления в них векторных регистров.
- MMX Instruction set (1997) – 8 64-битных регистров, только операции с целыми числами.
- SSE (Streaming SIMD Extension, 1999) – 8 128-битных регистров, операции с плавающей точкой. Получила развитие в виде SSE2, SSE3 и.т.д. (добавление новых инструкций).
- AVX (2008) – 16 256-битных регистров, *инструкции с 3 операндами*:

a = a + b -> c = a + b.

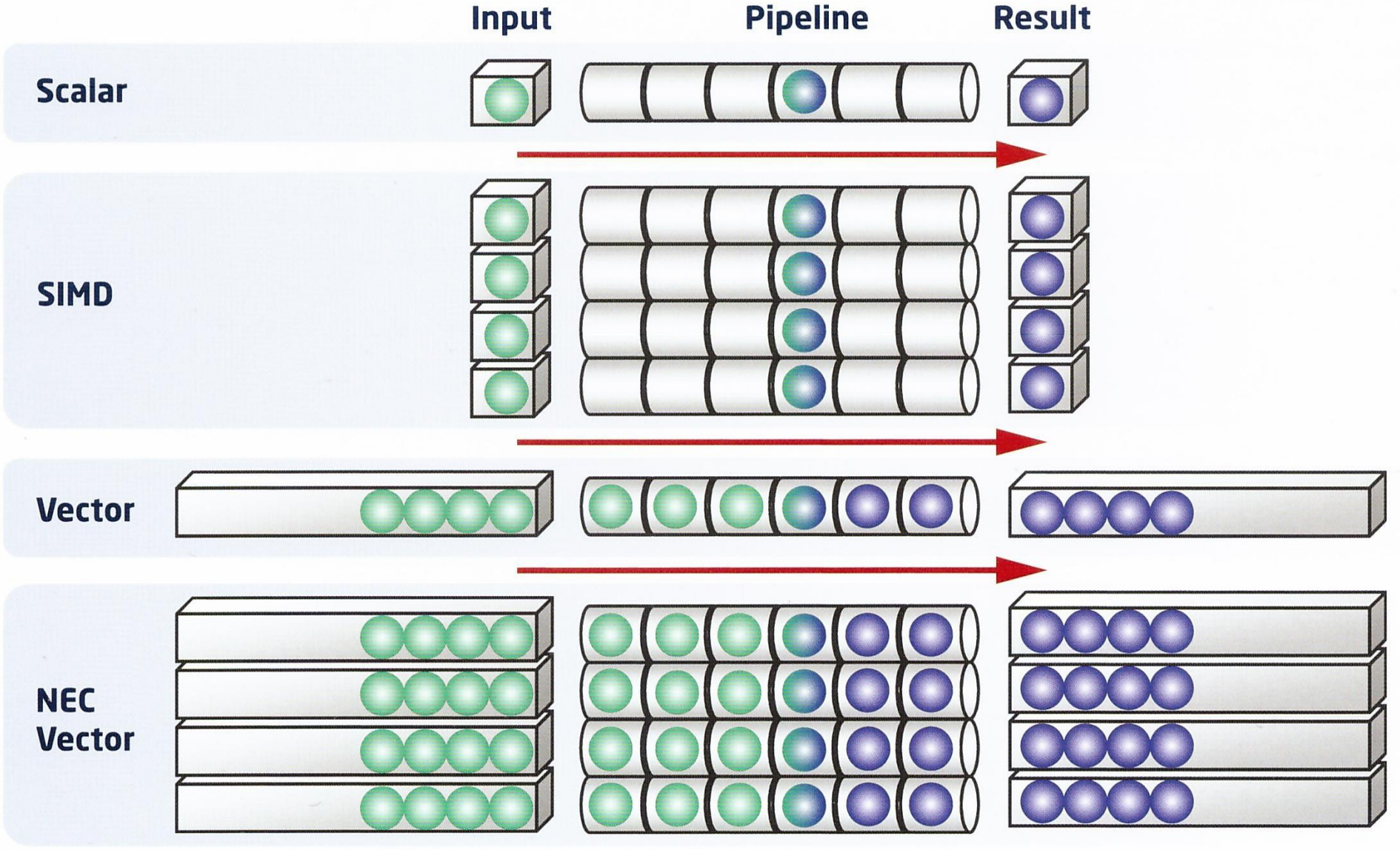
- AVX2 (2013) – имеются специальные инструкции для целых чисел. Добавлены инструции для операций вида a = b + c * d.
- AVX-512 – 32 512-битных регистра. Поддерживается в Xeon Phi KNC, KNL и Intel Skylake-X.

Векторно-конвейерные компьютеры

- AltiVec / Velocity Engine / VMX – в своё время прямой конкурент SSE.
  I. Регистры можно загружать и сохранять только в память.
  II. Больший (по сравнению с SSE) набор допустимых типов данных и операций.
  III. 32 128-битных регистра, инструкции с тремя операндами.
- ARM SVE – расширение для набора инструкций A64.
  I. Длина вектора от 128 до 2048 бит, кратна 128. Предпочитаемой длины вектора нет, всё зависит от конкретного «железа».
  II. Приложения подстраиваются под имеющийся объём.
  III. Не нужно ничего перекомпилировать / переписывать.
  IV. Ориентация на борьбу с проблемами автовекторизации (современные компиляторы часто не в состоянии провести векторизацию из-за зависимостей внутри вектора).
  V. Предназначен для решения научных задач с помощью HPC.

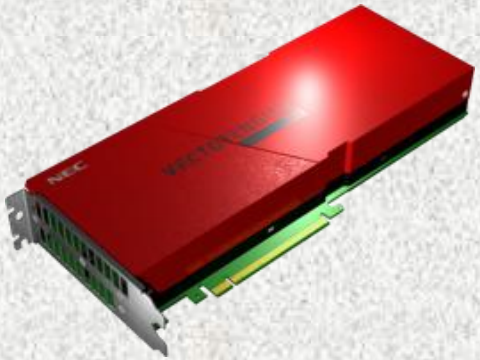
NEC SX-Aurora TSUBASA

NEC SX-Aurora TSUBASA

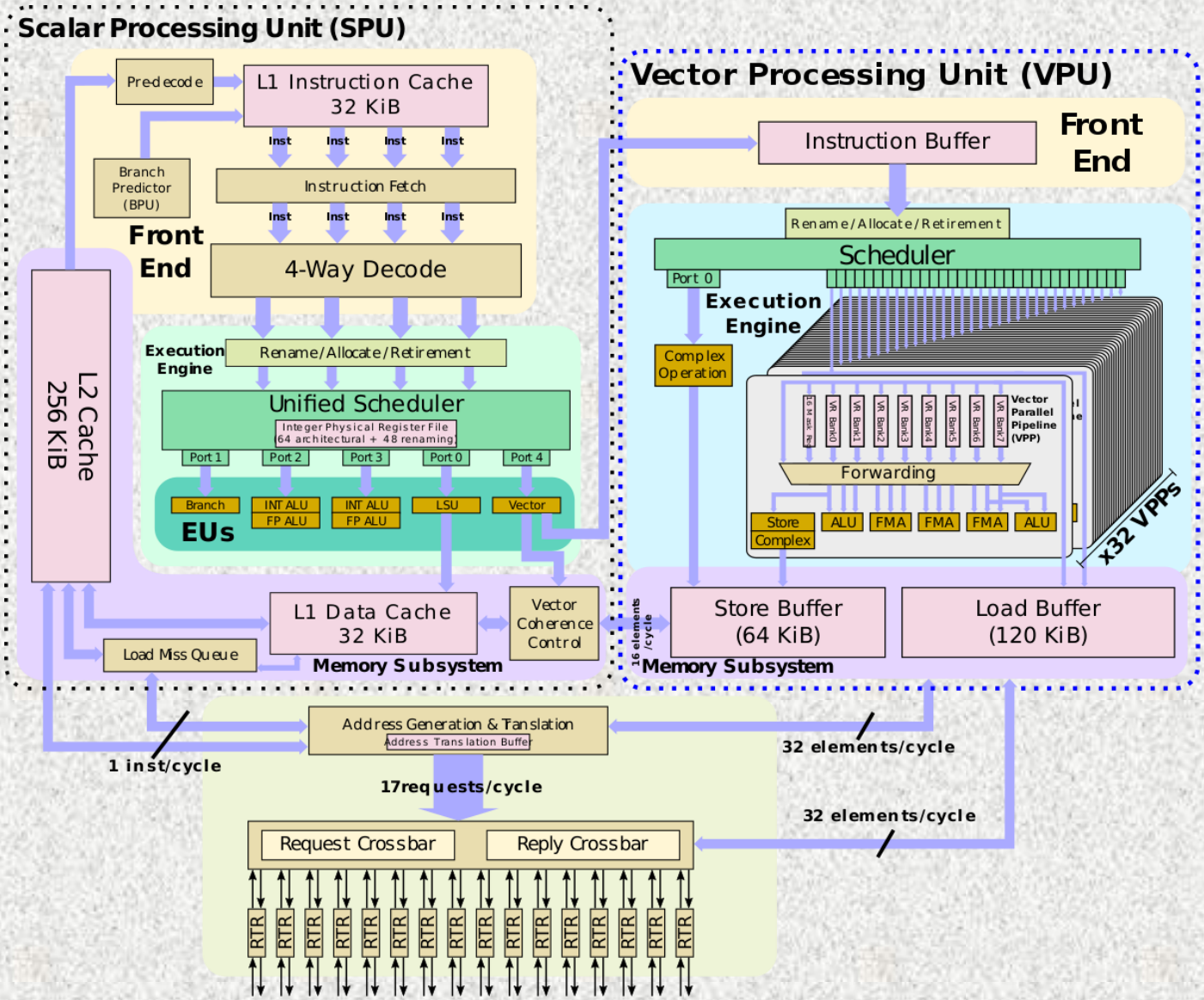


Совмещение векторных и SIMD команд

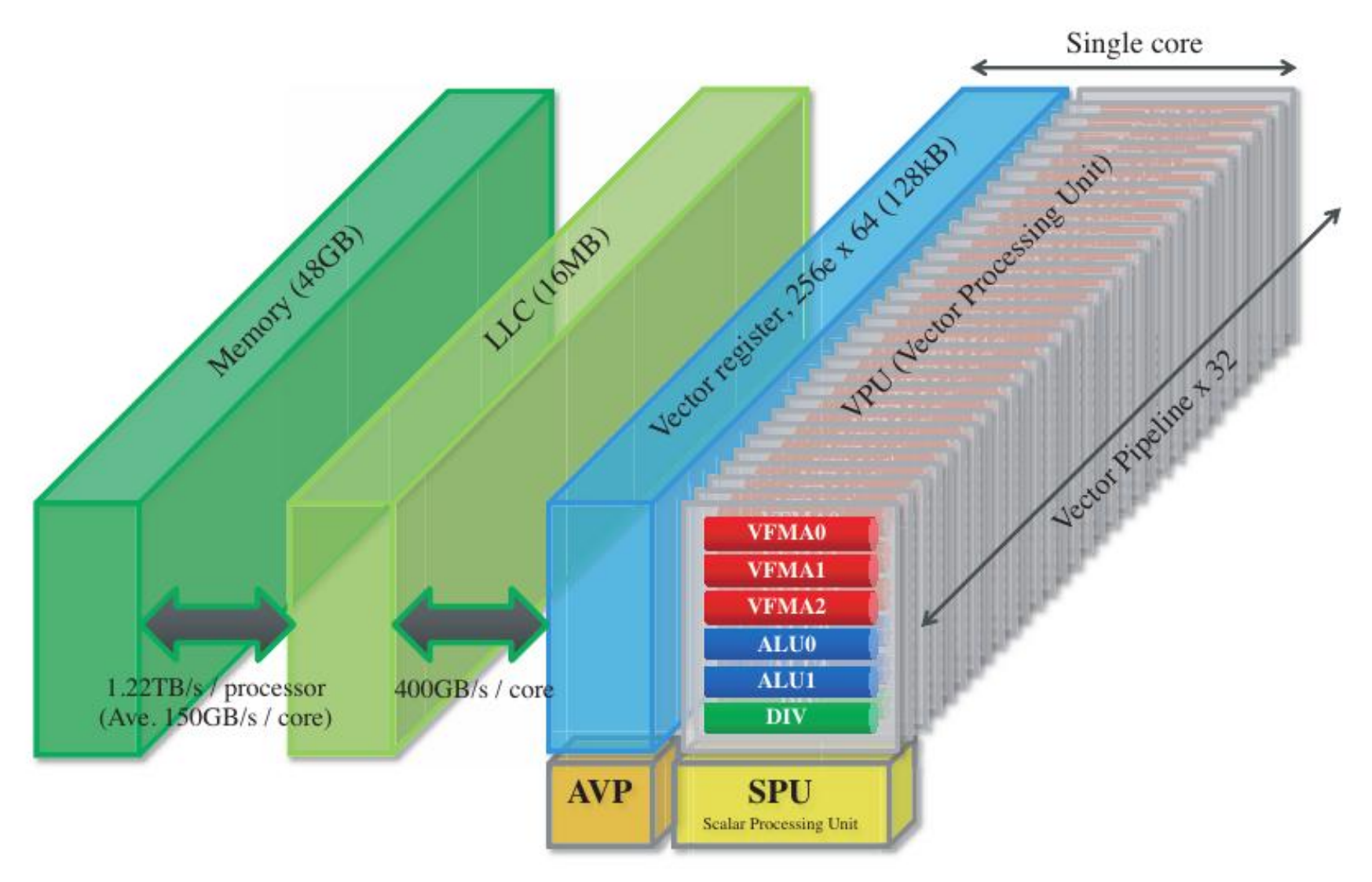
NEC SX-Aurora TSUBASA



Vector Engine (VE) в виде карты PCIe включает в себя 8 ядер, состоящих из скалярного и векторного блоков

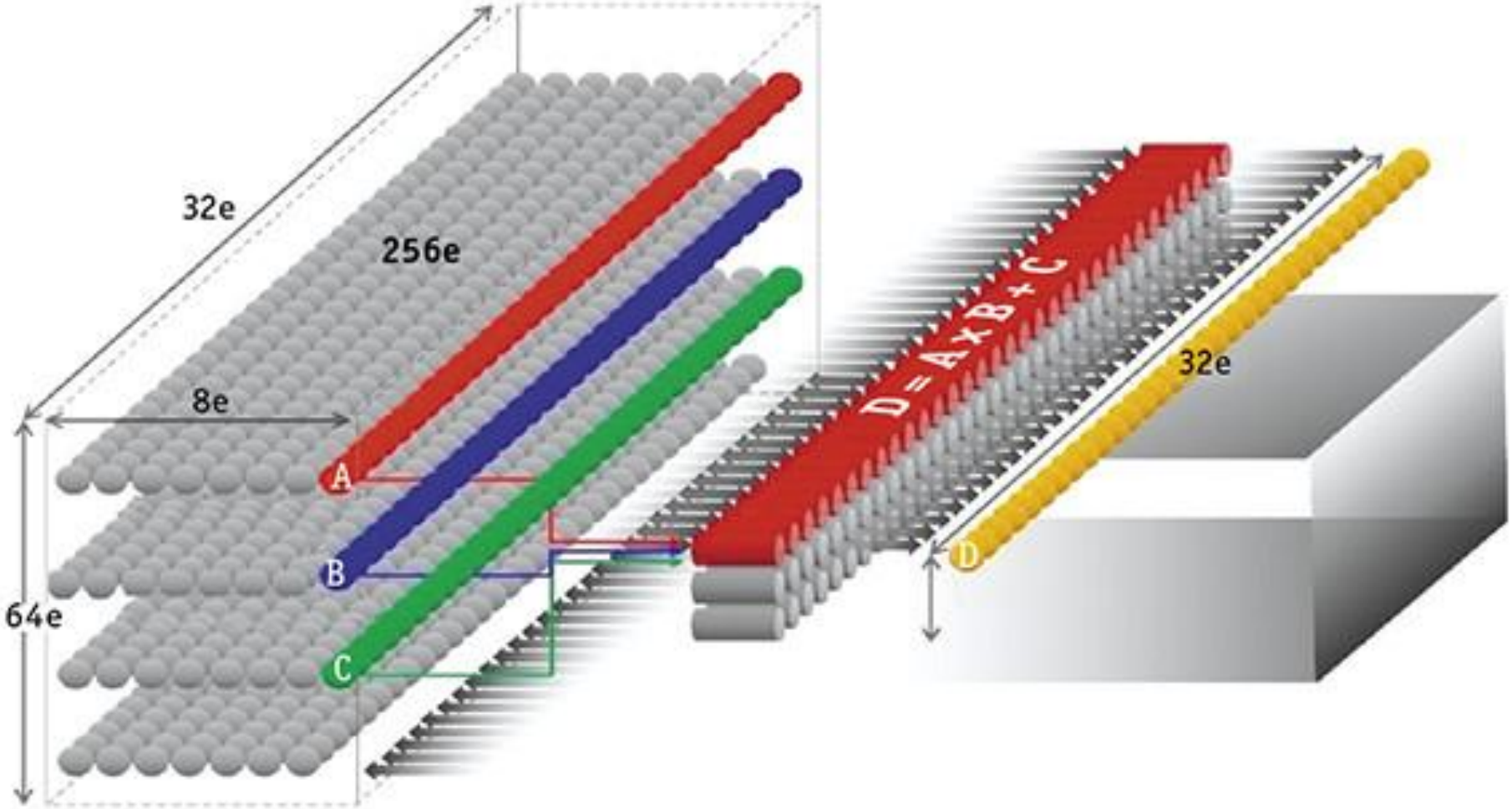


NEC SX-Aurora TSUBASA



В каждом ядре содержится по 64 векторных регистра длиной по 256 64-разрядных элементов. Суммарная ёмкость регистров в составляет 128 Кбайт.

NEC SX-Aurora TSUBASA



3 устройства FMA (Fused Multiply Add) выполняют операцию a * b + c над векторными регистрами – каждый такт 32 результата. Суммарно 32x3x2=192 Flop/такт. При тактовой частоте 1.6 ГГц получаем 307.2 GFlop/s на ядро или 2.4576 TFlop/s на процессор.

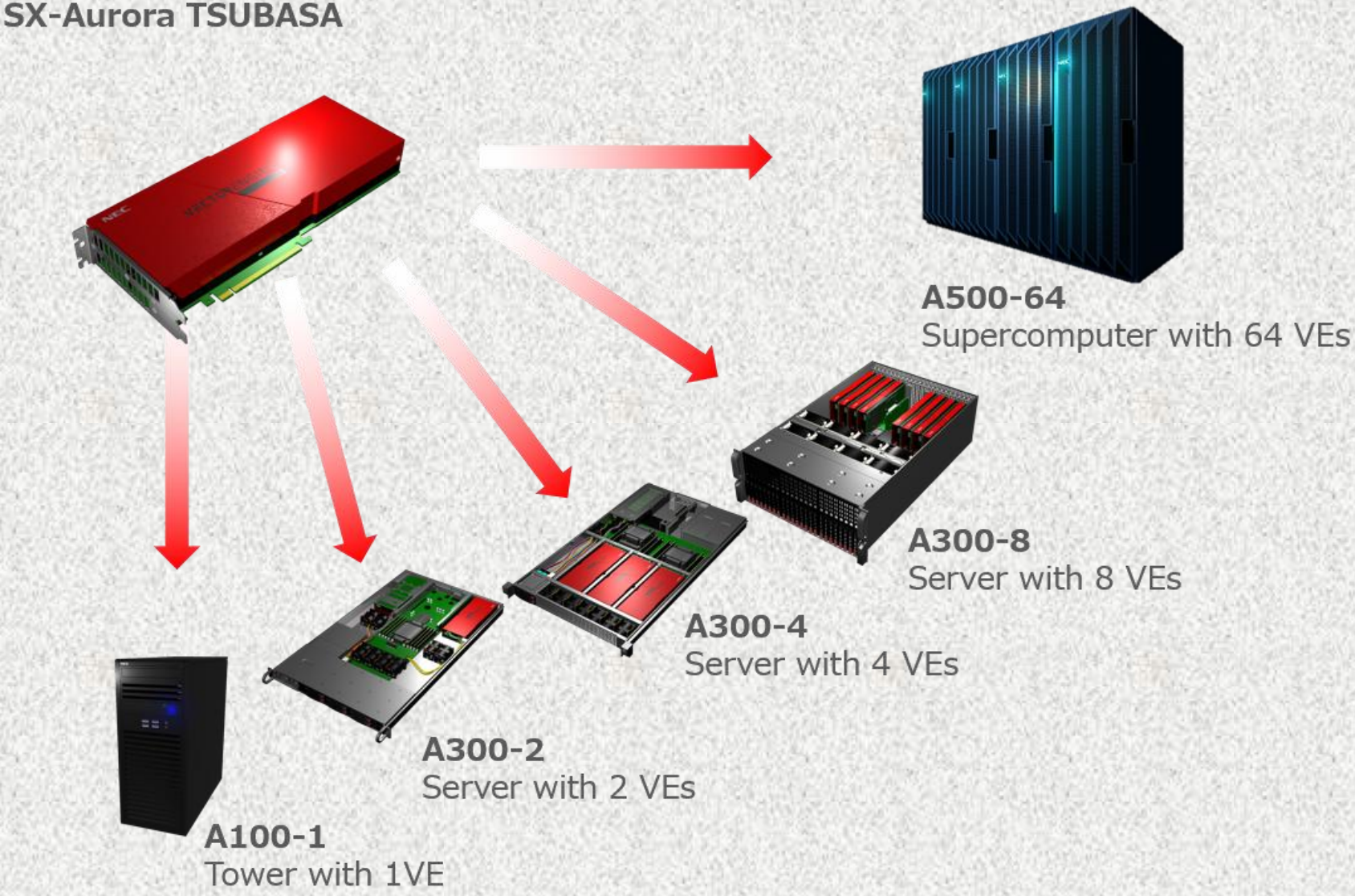
NEC SX-Aurora TSUBASA

В отличие от ускорителей, программа целиком выполняется на VE. Vector Host (VH) используется для компиляции приложений и реализации фонкций ОС, таких как выделение ресурсов, взаимодействие с файловой системой и т.д.

В рамках одного узла разные VE могут связываться друг с другом через PCIe. Большие параллельные системы, созданные с помощью SX-Aurora, используют в качестве коммуникационной сети Infiniband.

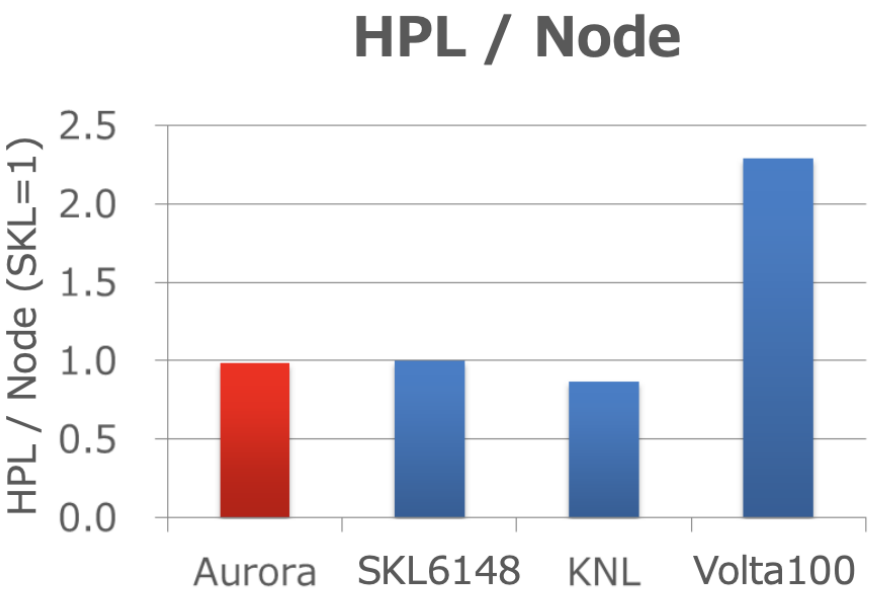
Операционная система VE называется VEOS и полностью выгружена в хост-систему, векторный хост (VH).

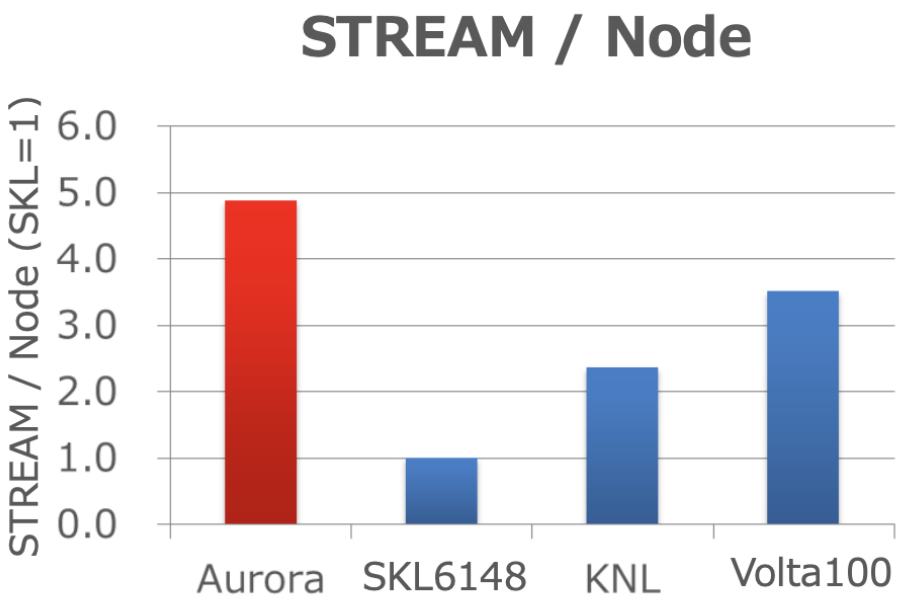
NEC SX-Aurora TSUBASA



Диапазон платформ TSUBASA от рабочих станций A100 до стоечных суперкомпьютеров A500 с пиковой производительностью 157.3 TFlop/s.

NEC SX-Aurora TSUBASA

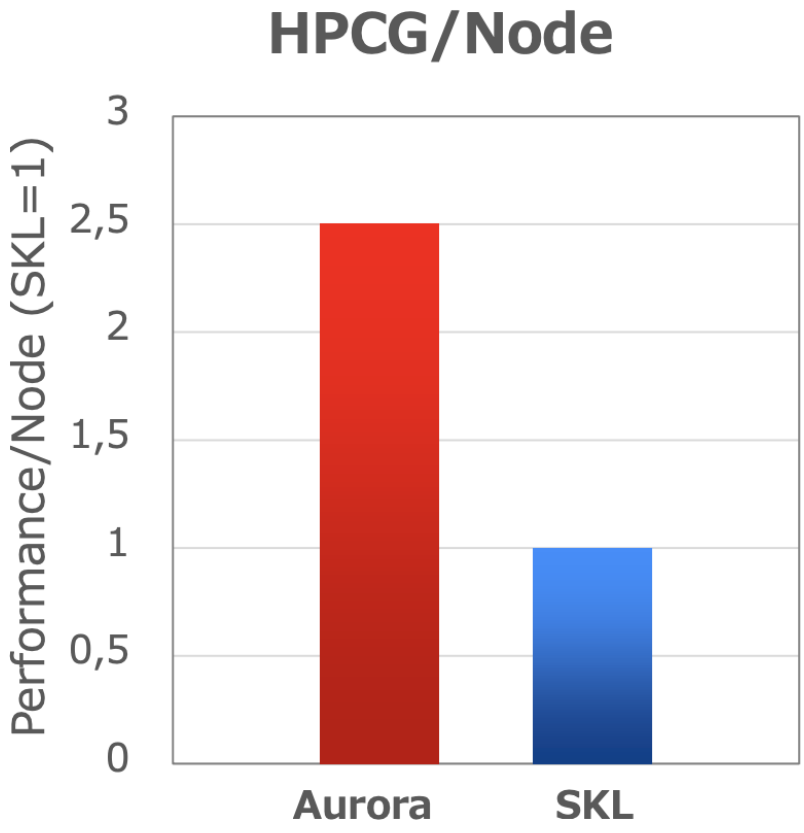


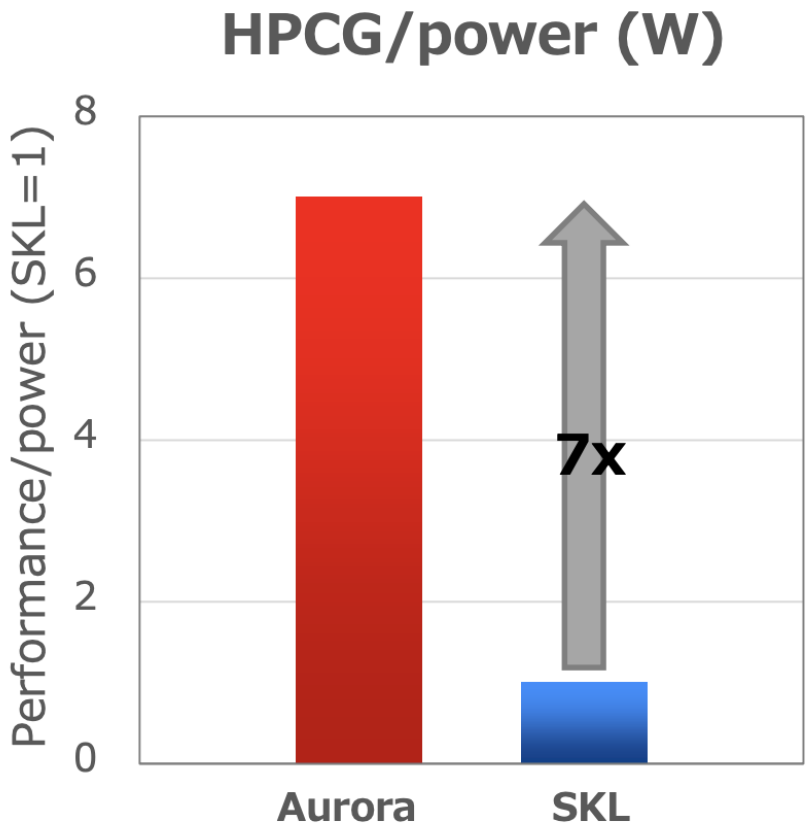


- Aurora is Vector Engine Type 10-B (1.4GHz, 8core)
- SKL is Intel Skylake 6148 Xeon x2/node
- KNL is Intel Knight Landing x1/node
- V100 is NVIDIA Tesla V100 x1/node

Сравнение производительности VE с Xeon Skylake (двухпроцессорная система с Xeon Gold 6148), Intel KNL и Nvidia Tesla V100.

NEC SX-Aurora TSUBASA





- Aurora is Vector Engine Type 10-B (1.4GHz, 8core)
- SKL is Intel Skylake 6148 Xeon x2/node

Сравнение производительности VE с Xeon Skylake (двухпроцессорная система с Xeon Gold 6148).

Векторно-конвейерные компьютеры

Основные особенности архитектуры, дающие заметный вклад в ускорение выполнения программ:

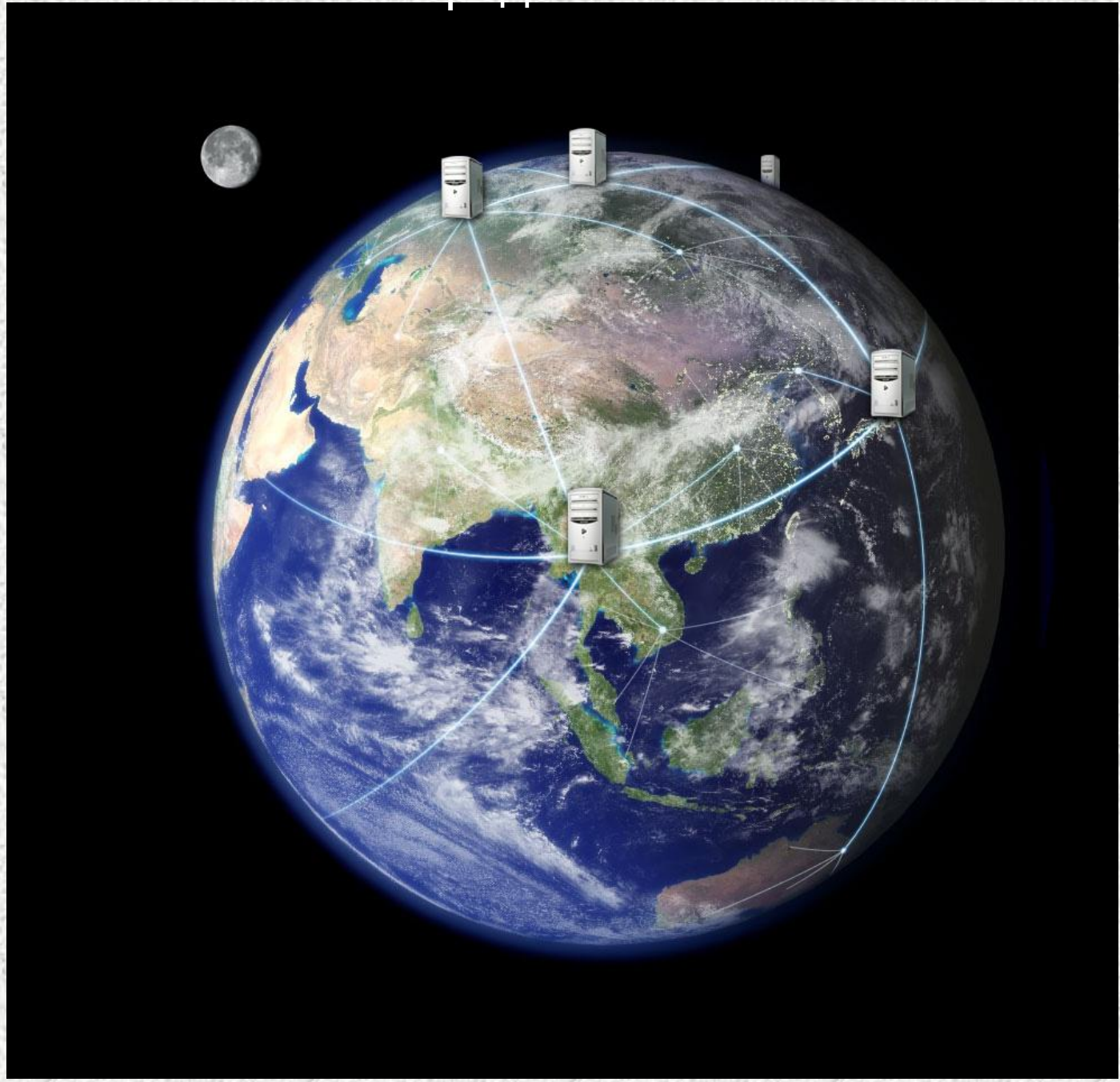
1. *Конвейеризация выполнения команд.*
2. *Независимость функциональных устройств.*
3. *Векторная обработка.*
4. *Дублирование конвейеров векторных устройств и устройств для вещественной арифметики.*
5. *Зацепление функциональных устройств.*
6. *Многопроцессорная обработка.*

Что снижает производительность векторно-конвейерных компьютеров?

1. *Закон Амдала*
2. *Время разгона конвейера*
3. *Секционирование векторных команд*
4. *Конфликты в памяти*
5. *Каналы процессор-память*
6. *Операции чтения/записи в векторные регистры*
7. *Ограниченное число векторных регистров*
8. *Несбалансированное использование устройств*
9. *Отсутствие операции деления*
10. *Перезагрузка буферов команд*
11. *...*

Распределенные вычислительные среды

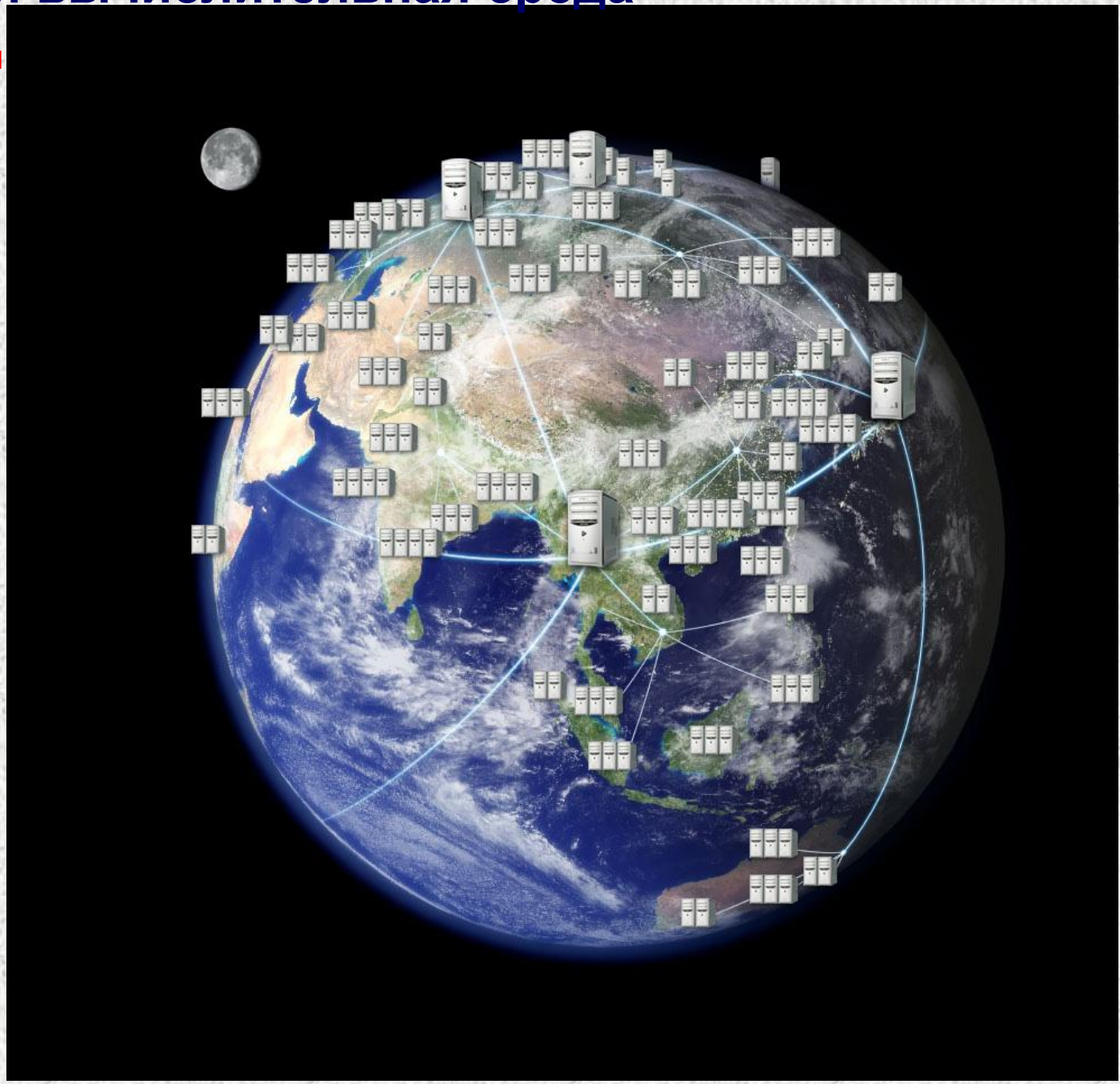
Распределенная вычислительная среда



Распределенные вычислительные среды, в чем отличия от классических компьютеров с распределенной памятью?

Распределенная вычислительная среда

Колоссальные ресурсы

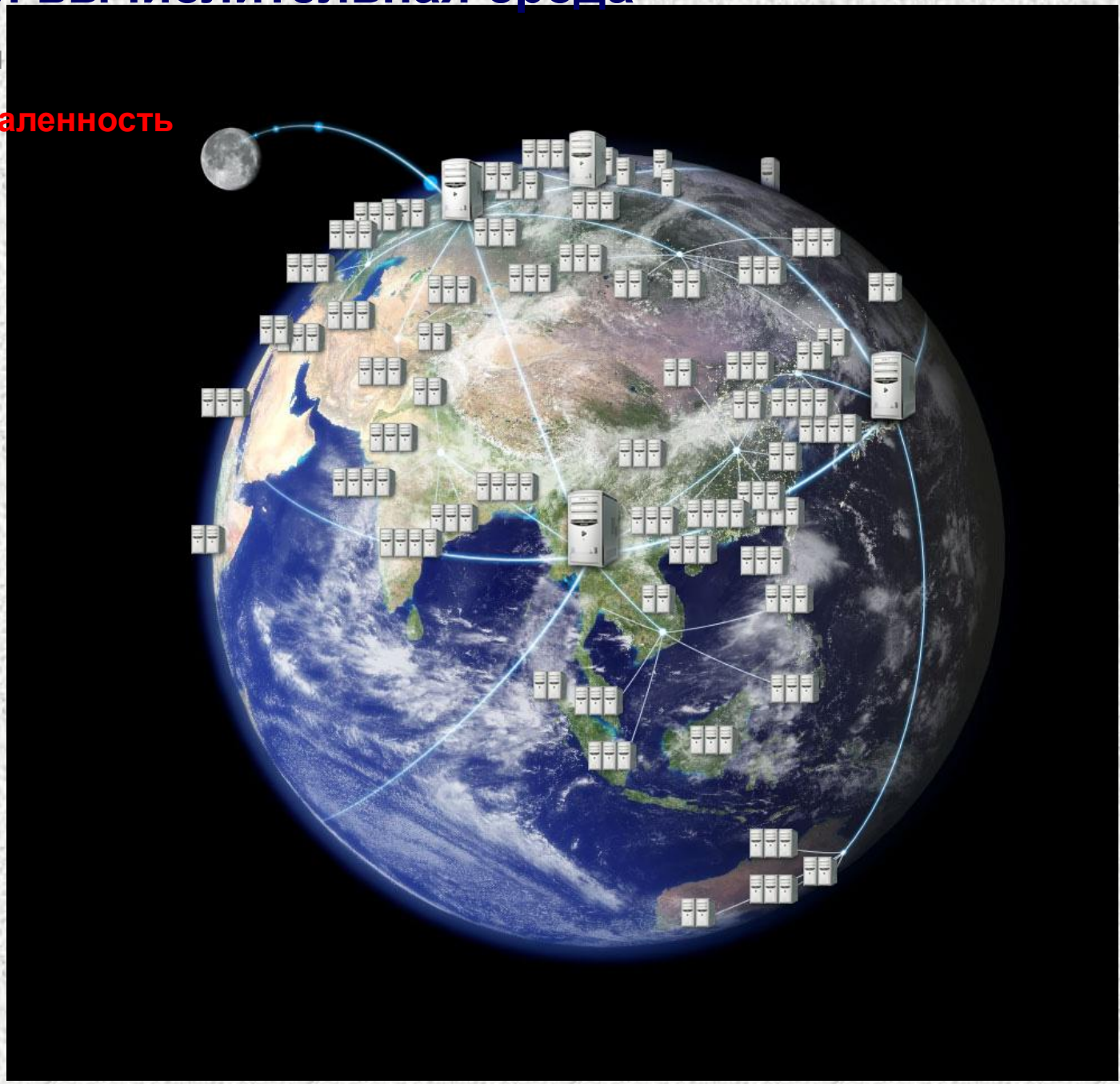


PARALLEL.RU

Распределенная вычислительная среда

Колоссальные ресурсы

Распределенность и удаленность

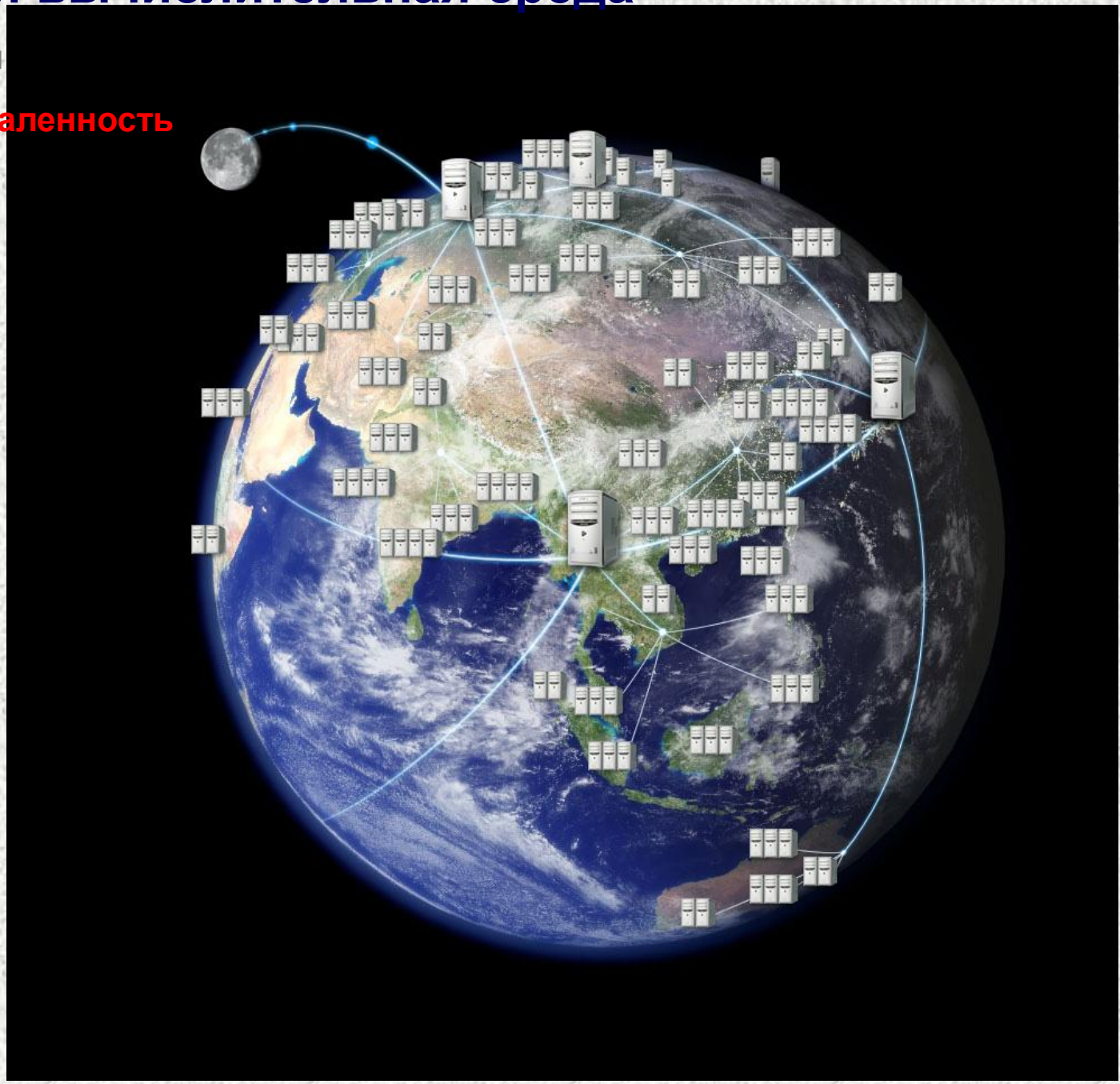


PARALLEL.RU

Распределенная вычислительная среда

Колоссальные ресурсы

Распределенность и удаленность

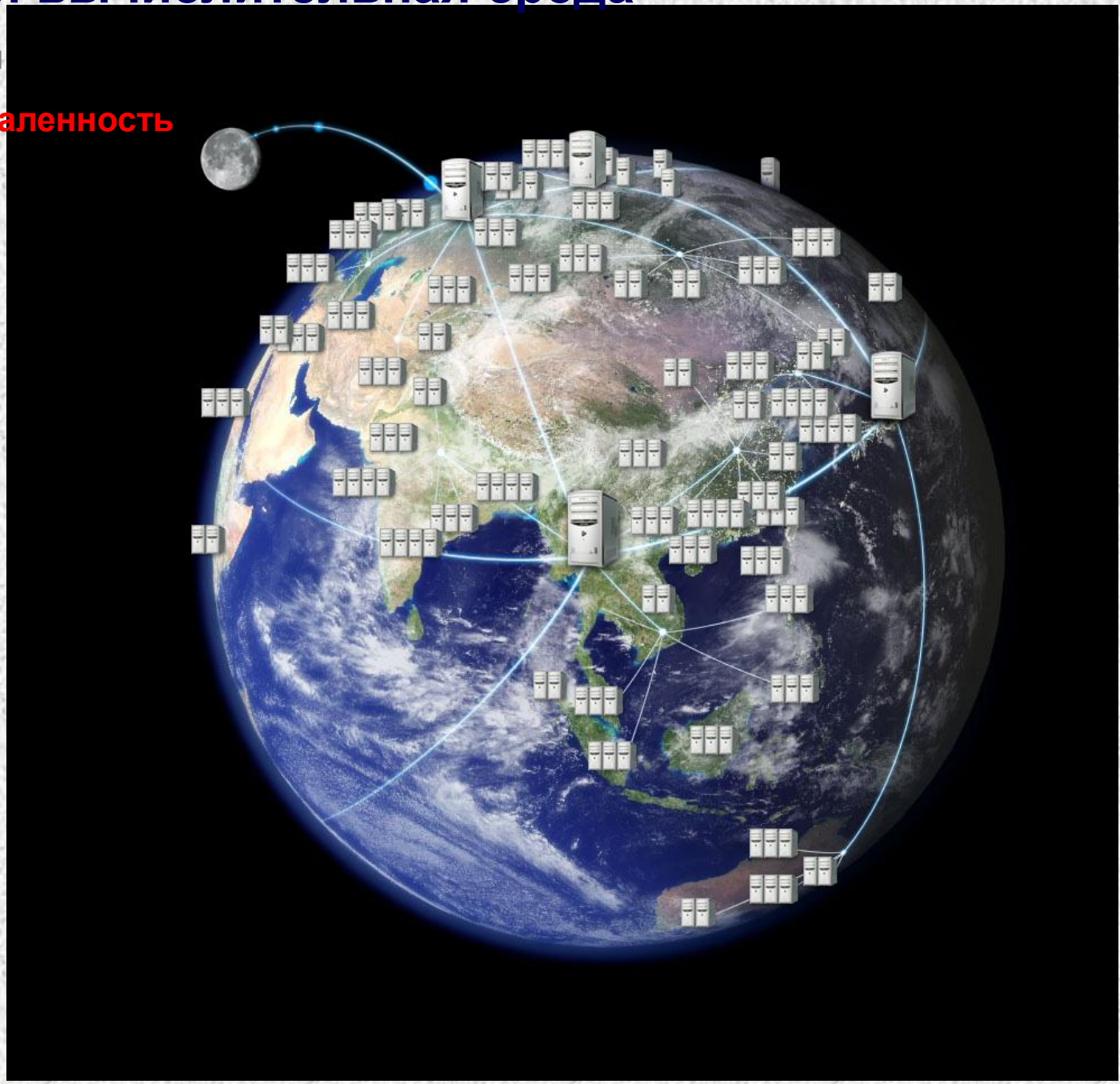


PARALLEL.RU

Распределенная вычислительная среда

Колоссальные ресурсы

Распределенность и удаленность

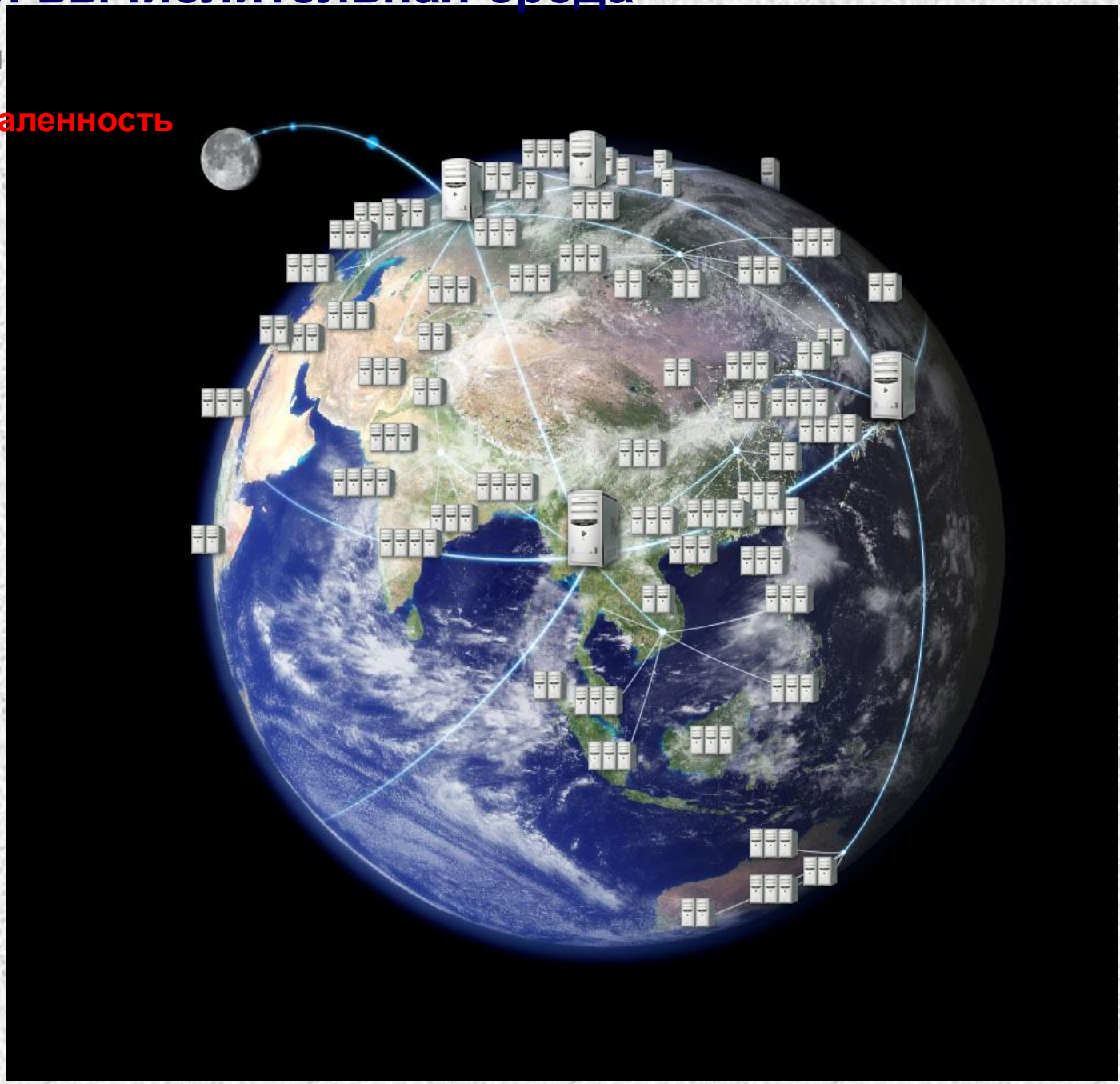


PARALLEL.RU

Распределенная вычислительная среда

Колоссальные ресурсы

Распределенность и удаленность



PARALLEL.RU

Распределенная вычислительная среда

Колоссальные ресурсы

Распределенность и удаленность

Динамичность конфигурации

PARALLEL.RU

Распределенная вычислительная среда

Колоссальные ресурсы

Распределенность и удаленность

Динамичность конфигурации

PARALLEL.RU

Распределенная вычислительная среда

Колоссальные ресурсы

Распределенность и удаленность

Динамичность конфигурации

PARALLEL.RU

Распределенная вычислительная среда

Колоссальные ресурсы

Распределенность и удаленность

Динамичность конфигурации

PARALLEL.RU

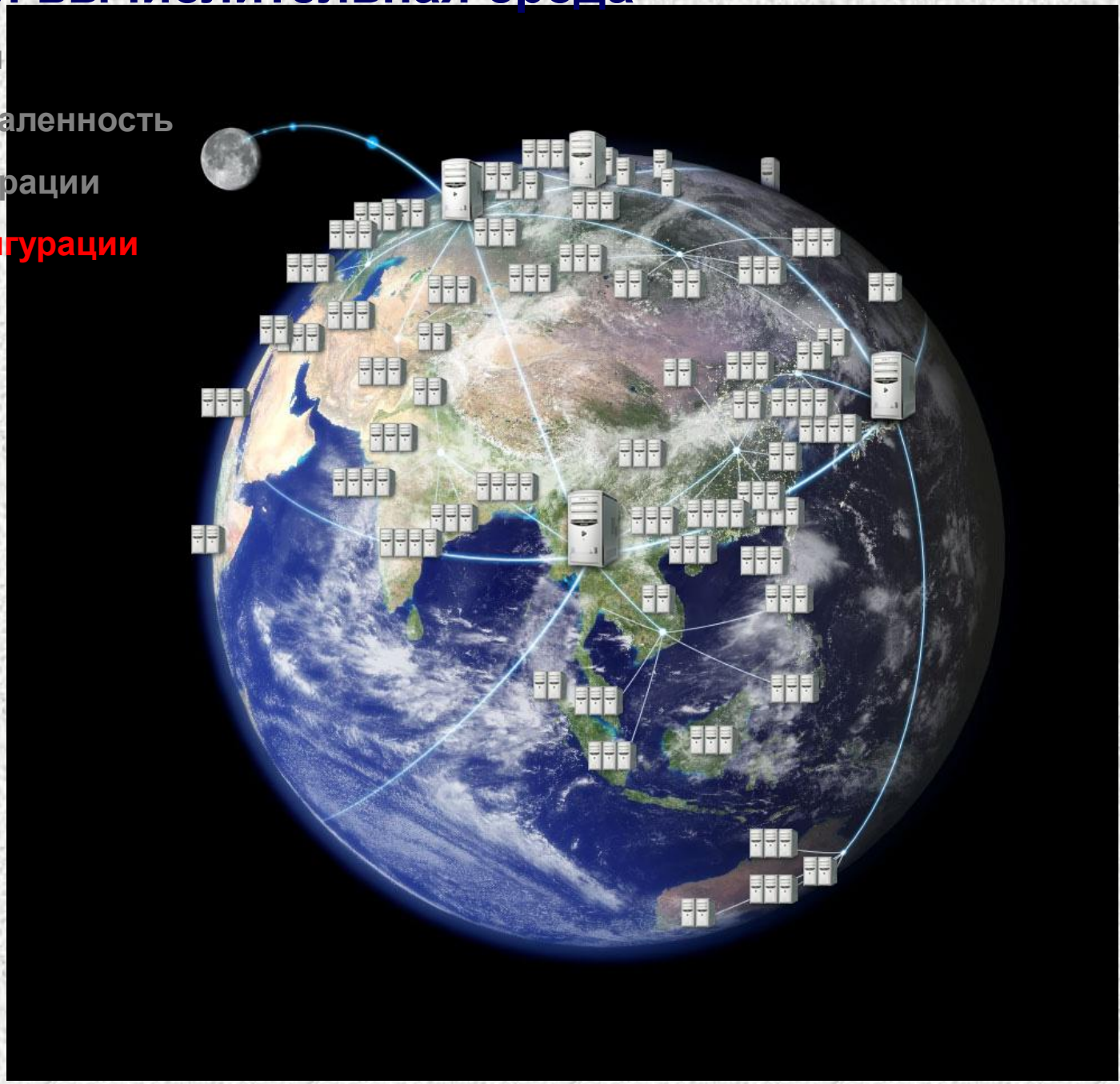
Распределенная вычислительная среда

Колоссальные ресурсы

Распределенность и удаленность

Динамичность конфигурации

Неоднородность конфигурации



PARALLEL.RU

Распределенная вычислительная среда

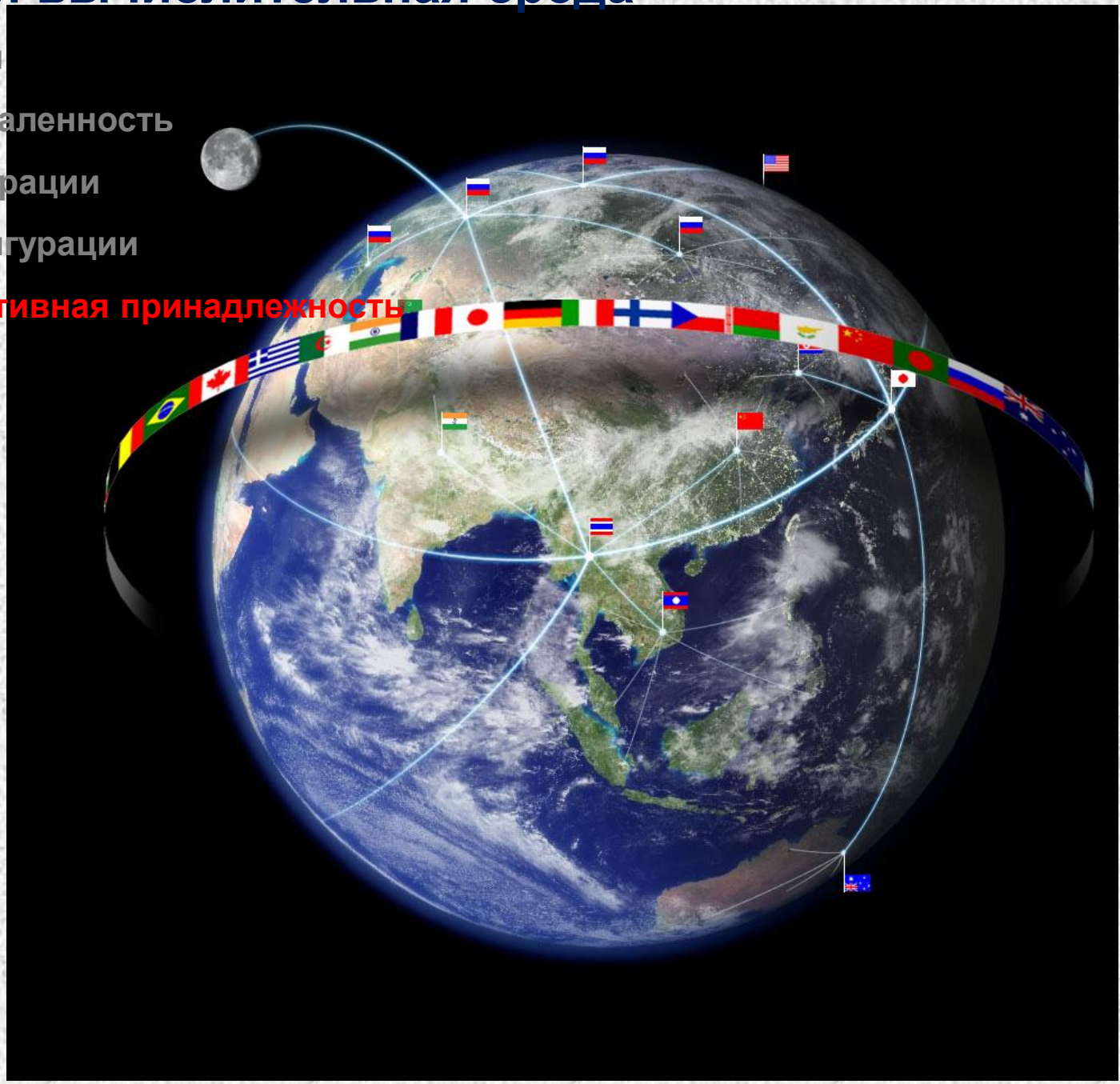
Колоссальные ресурсы

Распределенность и удаленность

Динамичность конфигурации

Неоднородность конфигурации

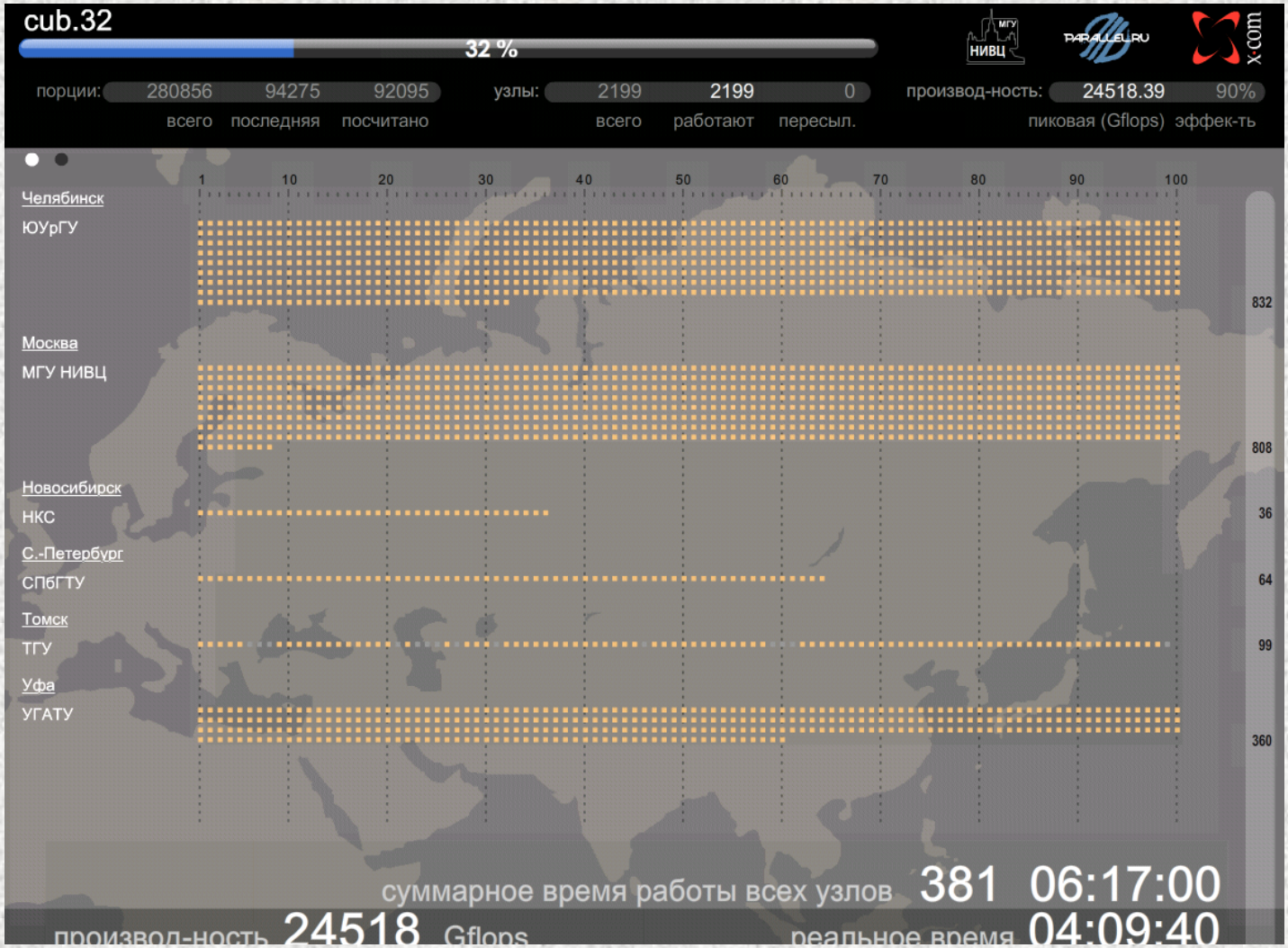
Различная административная принадлежность



PARALLEL.RU

Система метакомпьютинга X-COM

(http://x-com.parallel.ru)



x·com

Проектирование новых лекарств

НИВЦ МГУ, Гематологический центр РАМН

Время расчета: 31 дек. 2005 – 11 янв. 2006,

Процессорное время:42774 часа = 1782 дня =

*> **4.5 года** работы одного процессора*

Общее число работавших процессоров: 273,

Максимально процессоров одновременно: 244,

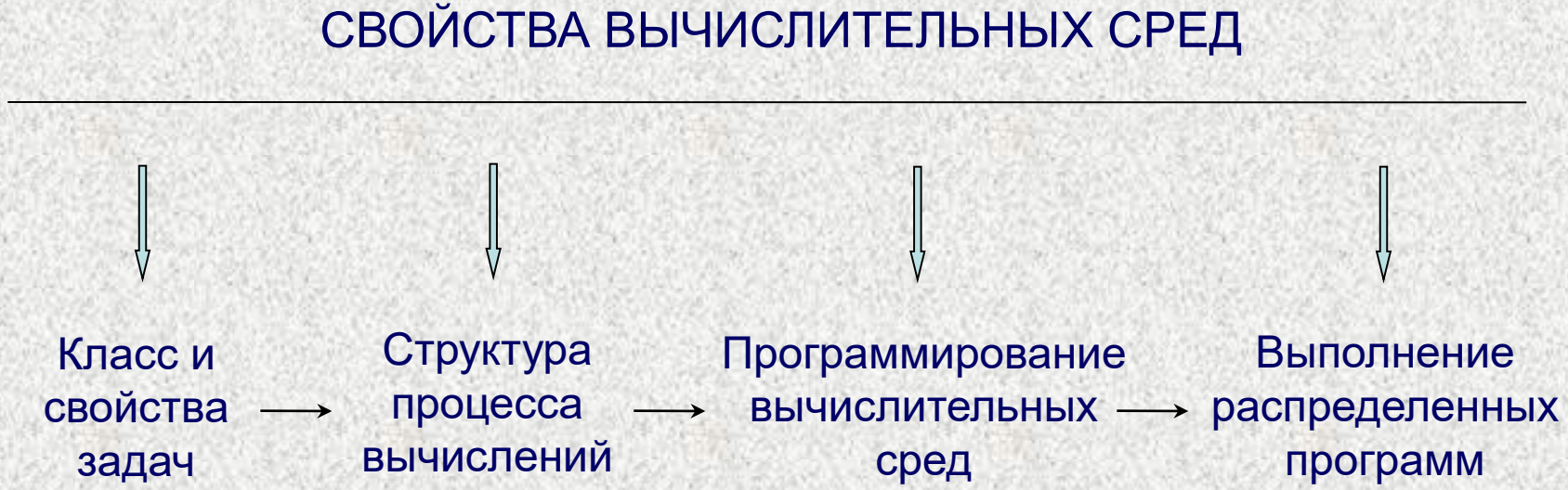
Пиковая производительность: >1 Тфлопс

Вычислительная среда:

кластеры МГУ + кластер ЮУрГУ (г.Челябинск) + учебный класс НИВЦ МГУ + компьютеры в лабораториях

x·com

Использование вычислительных сред



Смежное направление - ***проведение вычислений по требованию (on-demand computing).*** *Для многих организаций экономически невыгодно держать у себя высокопроизводительные компьютеры, поскольку потребности в проведении больших расчетов возникают лишь эпизодически, а стоимость приобретения и сопровождения такого оборудования велика.*

Другой популярный сейчас термин ***облачные вычисления (cloud computing)*** *- это технология распределённой обработки данных, в которой компьютерные ресурсы и мощности предоставляются пользователю как Интернет-сервис. Облачная обработка данных – парадигма, в рамках которой информация постоянно хранится на серверах в Интернет и временно кэшируется на клиентской стороне, например, на персональных компьютерах, игровых приставках, ноутбуках, смартфонах и т.д.*

Многопоточность *(**мультитредовость**, **multithreading**) — свойство платформы или приложения, состоящее в том, что процесс, порождённый в операционной системе, может состоять из нескольких потоков, выполняющихся «параллельно», то есть без предписанного порядка во времени. При выполнении некоторых задач такое разделение может достичь более эффективного использования ресурсов вычислительной машины.*

Гипертрейдинг *(**гиперпоточность**, **hyper-threading**) – технология, разработанная компанией Intel для процессоров на микроархитектуре NetBurst. Реализует идею «одновременной мультипоточности». После включения гипертрейдинга один физический процессор (одно физическое ядро) определяется операционной системой как два отдельных процессора (два логических ядра). При определённых рабочих нагрузках позволяет увеличить производительность процессора. Суть технологии: передача полезной работы бездействующим исполнительным устройствам.*

Параллелизм на уровне машинных команд

Параллелизм на уровне машинных команд

Параллелизм на уровне машинных команд *- способность процессора исполнять несколько независимых машинных команд одновременно в рамках одного программного потока (треда).*

Выгоды параллелизма на уровне машинных команд: отсутствие у пользователя необходимости в специальном параллельном программировании; переносимость.

Параллелизм на уровне машинных команд

Суперскалярная архитектура *- архитектура, использующая несколько дешифраторов команд, которые могут нагружать работой множество исполнительных блоков. Если в процессе работы команды, обрабатываемые конвейером, не противоречат друг другу, и одна не зависит от результата другой, то такое устройство может распараллелить выполнение команд.*

Суперскалярность не предполагает, что программа в терминах машинных команд будет включать в себя какую-либо информацию о содержащемся в ней параллелизме. ***Задача обнаружения параллелизма в машинном коде возлагается на аппаратуру,*** *она же и строит соответствующую последовательность исполнения команд. В этом смысле код для суперскалярных процессоров не отражает точно ни природу аппаратного обеспечения, на котором он будет реализован, ни точного временного порядка, в котором будут выполняться команды.*

Параллелизм на уровне машинных команд

VLIW (Very long instruction word, «очень длинное командное слово») *- архитектура процессоров с несколькими вычислительными устройствами. Характеризуется тем, что одна инструкция процессора содержит несколько операций, которые должны выполняться параллельно. Команда VLIW-процессора состоит из набора полей, каждое из которых отвечает за свою операцию. Если какая-то часть процессора на данном этапе не востребована, то соответствующее поле команды не задействуется.*

В процессорах VLIW задача распределения решается во время компиляции и в инструкциях явно указано, какое вычислительное устройство должно выполнять какую команду. ***Компилятор сам выявляет параллелизм в программе*** *и явно сообщает аппаратуре, какие операции не зависят друг от друга. Код для VLIW-процессоров содержит точный план того, как процессор будет выполнять программу: когда будет выполнена каждая операция, какие функциональные устройства будут работать, какие регистры какие операнды будут содержать и т.д.*

Параллелизм на уровне машинных команд

EPIC (Explicitly Parallel Instruction Computing) *- вычисления с явным (заданным) параллелизмом на уровне команд. В этой технологии компилятор явным образом говорит процессору, какие команды можно исполнить параллельно, а какие зависят от других команд.*

В архитектуре EPIC используется концепция длинного командного слова и добавляются следующие черты:

- ***Спекулятивная загрузка данных*** *- вынесение команд загрузки данных далеко вперёд инструкций, использующих эти данные. Это позволяет лучше использовать иерархию памяти. Если произведена запись, затрагивающая загружаемое содержимое, то производится вызов кода восстановления, перезагружающего значение.*
- ***Предикатное выполнение команд*** *- операции выполняются условно, в зависимости от параметра, имеющего булево значение - предиката, связанного с базовым блоком, содержащим операцию. Это позволяет: избежать излишних инструкций ветвления, если количество команд в ветвях условного оператора невелико; уменьшить нагрузку на устройство предсказания переходов.*

Высокопроизводительные компьютерные системы

Какие принципы построения?

Высокопроизводительные компьютерные системы
(процессорная основа)

GP-GPU … Векторные Классические

FPGAs Cell … Многоядерные

Эффективность (% от пика),
Производительность на ватт,
Производительность на объем,

…

Простота освоения,
Переносимость программ,
Скорость программирования,

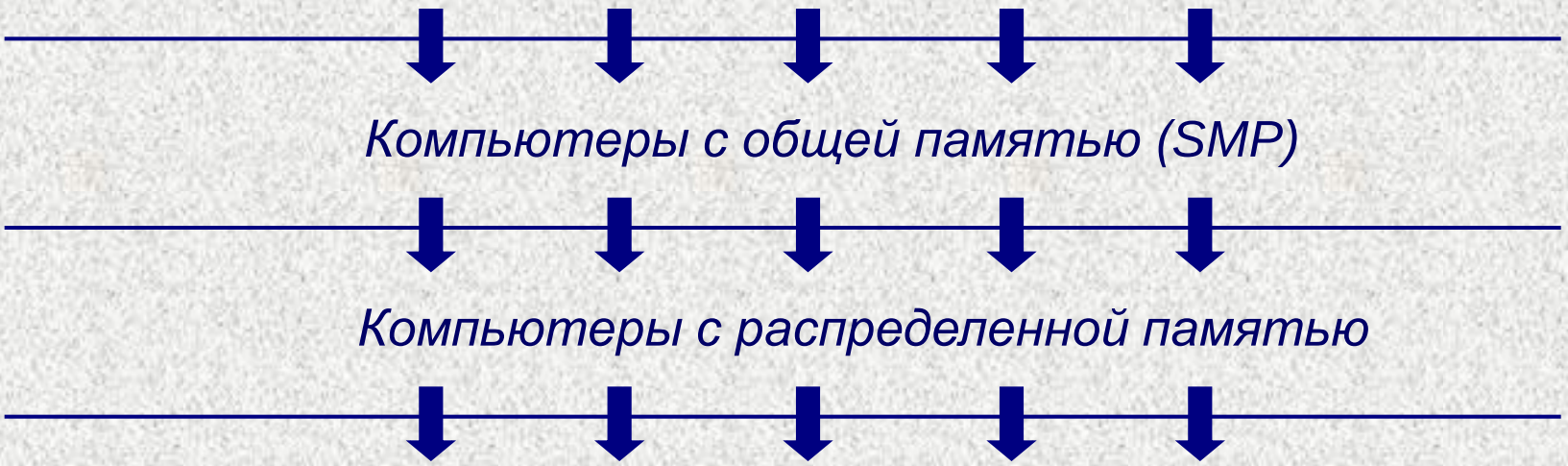
А.В.Каляев
1922 - 2004

В.С.Бурцев
1927 - 2005

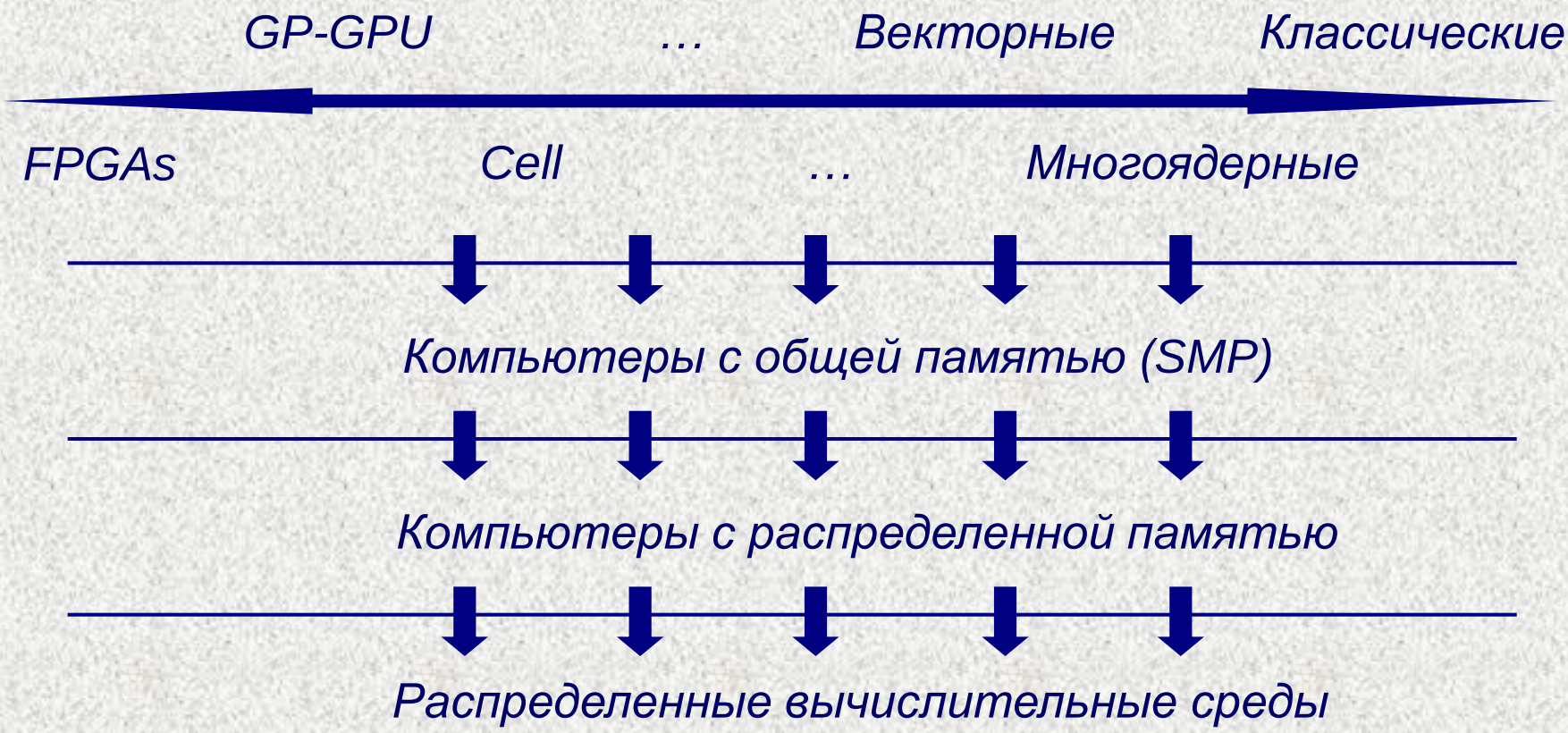
В.А.Мельников
1928 - 1993

Высокопроизводительные компьютерные системы
(процессорная основа)

Процессорная основа компьютеров



Высокопроизводительные компьютерные системы
(процессорная основа)



Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ (лекция 7)

А.С.Антонов

Вед. н.с. НИВЦ МГУ, к.ф.-м.н.

asa@parallel.ru

ВМК МГУ, 2020

Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ (лекция 8)

А.С.Антонов

Вед. н.с. НИВЦ МГУ, к.ф.-м.н.

asa@parallel.ru

ВМК МГУ, 2020

Методы оценки производительности

Производительность?

- *Вычислительное ядро: A = B+C*
- *Производительность: ‘ X ’ Mflops на Cray T90*

(например)

- *Пользователь: Я хочу больше, чем ‘X’...*

Производительность?

- *Вычислительное ядро: A = B+C*1*
- *Производительность: ‘2*X ’ Mflops на Cray T90 :))*

  А время осталось неизменным!!! :((

- *Пользователь: Я хочу больше, чем ‘X’...*
- *System Guru: Хочешь больше Mflops… Нет проблем...*

Методы оценки производительности

Какой компьютер выбрать?

В идеале было бы компьютеру однозначно сопоставить некое число.

***Пиковая производительность**: вычисляется просто и однозначно, но нет связи с реальной задачей пользователя. Даёт нижнюю оценку времени выполнения программы.*

Полезнее для пользователя оценка эффективности программно-аппаратной среды на некоторых задачах или наборе задач.

***Синтетические (или искусственные) тесты** не имеют отношения к реальным приложениям; предназначены для создания стрессовой нагрузки на отдельные подсистемы компьютера.*

***Реальные тесты** выполняют реальные задачи над реальными данными.*

Методы оценки производительности

Основные требования к тестам производительности:

- *Непротиворечивость и понятность результатов.*
- *Легкость в использовании.*
- *Масштабируемость.*
- *Переносимость.*
- *Репрезентативность.*
- *Доступность теста и его исходного кода.*
- *Воспроизводимость.*

Методы оценки производительности

Наиболее известные тесты (бенчмарки):

- *Linpack*
- *STREAM*
- *Ливерморские циклы*
- *Perfect Club Benchmarks*
- *SPEC*
- *HINT*
- *NAS Parallel Benchmarks (NPB)*
- *HPC Challenge*
- *Graph500*
- *HPCG*

Методы оценки производительности

Тест Linpack

- *Создан Джеком Донгарроой (Jack Dongarra) и его коллегами в 1979 году.*
- *Используется для формирования списков Top500 и Топ50.*
- *Решение больших систем линейных алгебраических уравнений с плотной квадратной матрицей методом LU-разложения.*
- *Число операций с плавающей точкой оценивается по формуле $2n^3/3 + 2n^2$, где n – линейный размер матрицы.*
- *Сначала Linpack 100×100, запрет изменений текста.*
- *Далее - Linpack 1000×1000, стандартная головная часть программы.*
- *Сейчас – произвольный (максимально возможный) размер матрицы, возможность вносить любые изменения в текст.*

Методы оценки производительности

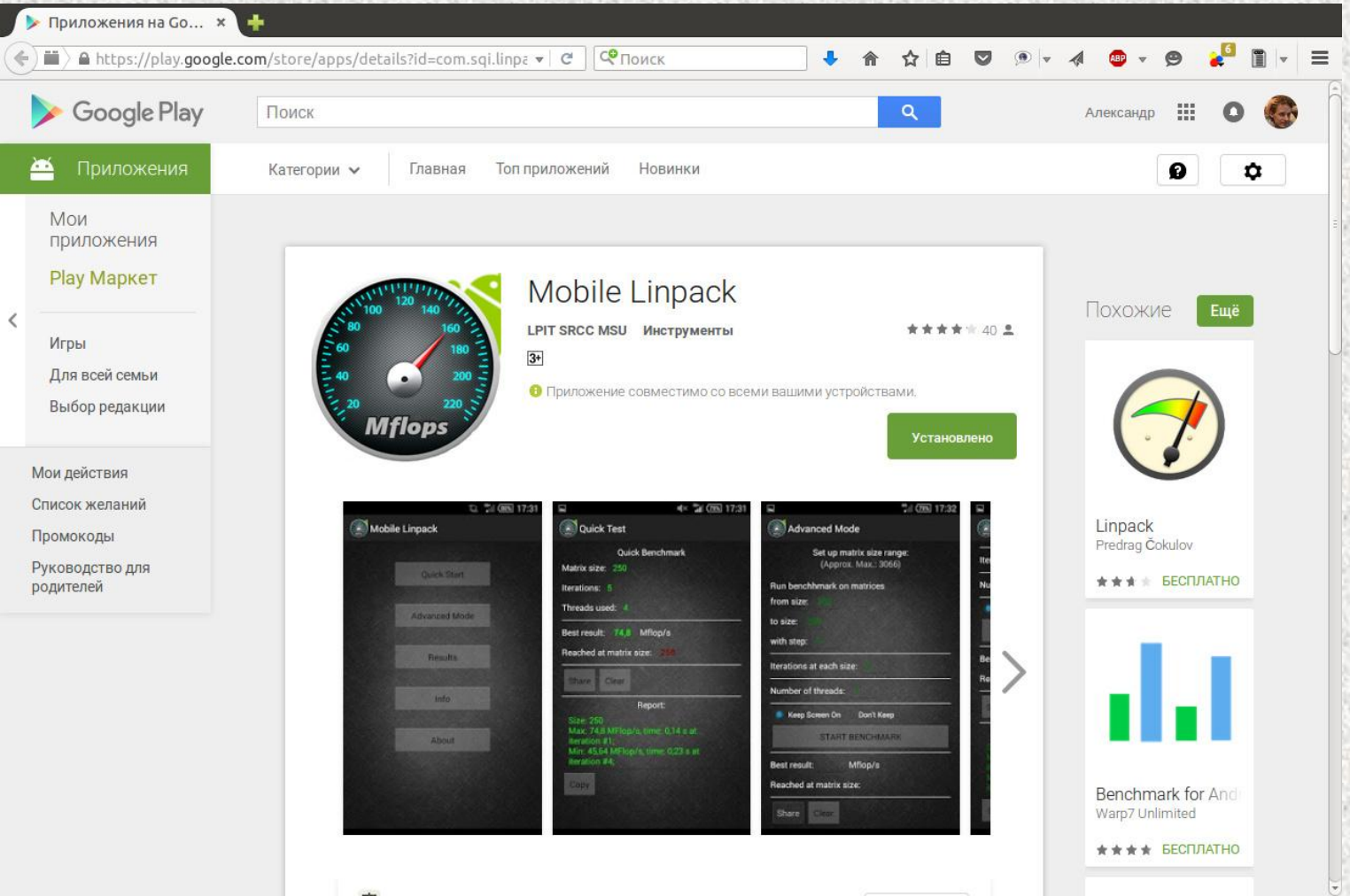
High Performance Linpack (HPL)

- *http://www.netlib.org/benchmark/hpl/*
- *Наиболее популярная реализация теста Linpack на языке Си.*
- *Обмены между процессорами выполняются через процедуры MPI.*
- *Вычисления на каждом процессоре основываются на низкоуровневой библиотеке базовых функций линейной алгебры BLAS (Basic Linear Algebra Subprograms) или VSIPL(Vector Signal Image Processing Library).*
- *Варианты BLAS:*
  - *ACML (AMD Core Math Library) для процессоров AMD Athlon и Opteron.*
  - *Intel MKL (Intel Math Kernel Library) для процессоров Intel.*
  - *cuBLAS в составе NVIDIA CUDA SDK для видеокарт и ускорителей.*
  - *ESSL (Engineering and Scientific Subroutine Library) для процессоров PowerPC.*
  - *ATLAS (Automatically Tuned Linear Algebra Software), реализация интерфейса BLAS с открытым исходным кодом.*
  - *uBLAS, часть библиотеки Boost.*

Методы оценки производительности

http://linpack.hpc.msu.ru/

Реализация теста Linpack для мобильных устройств.
Версии для Android 1.6 и выше и для iOS 6.0 и выше.



Адрес для отзывов: ml@parallel.ru

Методы оценки производительности

STREAM (Sustainable Memory Bandwidth in High Performance Computers)

http://www.cs.virginia.edu/stream/

- *Синтетический тест, оценивающий скорость работы с памятью с простой арифметикой и без.*
- *STREAM. Производительность на операциях:*
  - *a(i)=b(i);*
  - *a(i)=q*b(i);*
  - *a(i)=b(i)+c(i);*
  - *a(i)=b(i)+q*c(i);*
  - *замер скорости передачи данных.*
- *STREAM2. Производительность на операциях:*
  - *a(i)=q;*
  - *a(i)=b(i);*
  - *a(i)=a(i)+q*b(i);*
  - *sum = sum + a(i);*

Методы оценки производительности

STREAM (Sustainable Memory Bandwidth in High Performance Computers)

- *Размер массивов задаётся достаточным, чтобы выйти за размеры кэш-памяти. Отношение пиковой производительности к скорости передачи данных почти всегда больше 1.Чем отношение больше, тем больше несбалансированность компьютера.*
- *Параллельные версии с использованием OpenMP и MPI.*

Простота и переносимость тестов вызывает недоверие, появляются более сложные тесты, что вызывает проблемы с переносимостью и т.д. Поэтому создаются наборы тестов.

Методы оценки производительности

NAS Parallel Benchmarks

- *http://www.nas.nasa.gov/Software/NPB/*
- *Набор тестов производительности, разработанных в NASA Advanced Supercomputing (NAS) Division (ранее NASA Numerical Aerodynamic Simulation Program).*
- *Существуют последовательная реализация, параллельные реализации с использованием MPI, OpenMP, MPI+OpenMP , вариант на JAVA, версия для Grid на базе Globus.*
- *Последняя на данный момент версия NPB 3.4.*
- *Размер задачи – классы:*
  - *S, W (для тестовых прогонов);*
  - *A, B, C (каждый в среднем в 4 раза больше предыдущего);*
  - *D, E, F (каждый в среднем в 16 раз больше предыдущего)*

Методы оценки производительности

NAS Parallel Benchmarks

- *5 вычислительных ядер:*
  - *IS (Integer Sort);*
  - *EP (Embarrassingly Parallel);*
  - *CG (Conjugate Gradient);*
  - *MG (MultiGrid);*
  - *FT (Fast Fourier Transform).*
- *3 модельных приложения:*
  - *BT (Block Tri-diagonal solver);*
  - *SP (Scalar Penta-diagonal solver);*
  - *LU (Lower-Upper Gauss-Seidel solver).*
- *3 дополнительных теста:*
  - *UA (Unstructured Adaptive mesh);*
  - *DC (Data Cube);*
  - *DT (Data Traffic).*

Методы оценки производительности

HPC Challenge

- *http://www.hpcchallenge.org/*
- *Включает в себя 7 тестов:*
  - *HPL (Linpack);*
  - *DGEMM (вычисляет производительность перемножения матриц);*
  - *STREAM;*
  - *PTRANS (транспонирование матрицы);*
  - *RandomAccess (вычисляет скорость случайных обращений к памяти);*
  - *FFTE (реализация одномерного дискретного преобразования Фурье);*
  - *Communication bandwidth and latency (скорость передачи данных и латентность).*

Методы оценки производительности

Graph500

- *http://www.graph500.org/*
- *Анонсирован в 2010 году, составляется свой список наиболее производительных компьютеров.*
- *На данный момент последняя версия теста 3.0.0.*
- *Включает последовательный вариант, варианты на OpenMP, MPI и вариант для Cray XMT.*
- *Реализует поиск в ширину (BFS) и поиск кратчайшего пути от одной вершины (SSSP) в большом ненаправленном графе.*
- *Используется для измерения пропускной способности сетевой системы суперкомпьютеров.*
- *Производительность алгоритмов измеряется количеством пройденных дуг графа в секунду (Traversed Edges Per Second, TEPS). Используются обозначения ME/s и GE/s – миллионы и миллиарды пройденных дуг в секунду соответственно.*

Методы оценки производительности

HPCG

- *http://hpcg-benchmark.org/*
- *High Performance Conjugate Gradient – решение системы линейных алгебраических уравнений с разрежённой квадратной положительно определённой симметричной матрицей.*
- *Оценивает не только скорость самих вычислений, но и нерегулярных обращений в память.*
- *Разработчики: Jack Dongarra, Michael Heroux, Piotr Luszczek*
- *Анонсирован в 2013 году*
- *На данный момент последняя версия теста 3.1.*

Методы оценки производительности

Необходимость комплексного тестирования программно-аппаратной среды

- *базовый уровень ПО (ОС, компилятор, системы программирования);*
- *базовый уровень аппаратуры (элементарные операции, иерархия памяти);*
- *уровень операций ввода/вывода;*
- *базовый коммуникационный уровень;*
- *коммуникационный уровень приложений;*
- *уровень модельных приложений;*
- *уровень реальных приложений.*

Технологии параллельного программирования

Технологии параллельного программирования

Модели параллельного программирования:

• ***SPMD*** *(**Single Program, Multiple Data**, единая программа, множество данных) – на всех процессорах выполняется одна и та же программа над своим множеством входных данных.*

• ***Master-Workers*** *или **Master-Slave** (мастер-рабочие) – один из процессов (называемый главным, master) получает входные данные задачи, разделяет их на части и передаёт другим процессам (рабочим, workers) для обработки, а затем получает результаты и проводит финальную обработку.*

• ***Message passing*** *(модель передачи сообщений). Программа порождает несколько задач с уникальными идентификаторами. Взаимодействие осуществляется посредством отправки и приема сообщений. Новые задачи могут создаваться во время выполнения параллельной программы, несколько задач могут выполняться на одном процессоре.*

Технологии параллельного программирования

Модели параллельного программирования:

• ***Data parallel*** *(модель параллелизма данных). Одна операция применяется к множеству элементов структуры данных. Программа содержит последовательность таких операций. Распределяемыми данными обычно являются массивы. Как правило, языки программирования, поддерживающие данную модель, допускают операции над массивами, позволяют использовать в выражениях целые массивы, вырезки из массивов. Каждый процессор отвечает за то подмножество элементов массива, которое расположено в его локальной памяти.*

•***Shared memory*** *(модель общей памяти). Задачи обращаются к общей памяти, имея общее адресное пространство. Требуется разграничение доступа к общим данным.*

Технологии параллельного программирования

Критерии выбора

Различных технологий много. На чём основываться при выборе наиболее подходящей?

- *Возможность создания эффективных программ.*
- *Возможность быстрого создания параллельных программ (продуктивность программирования).*
- *Переносимость программ, гарантии сохранения эффектевности.*
- *Стоимость.*

Технологии параллельного программирования

При программировании систем с общей и распределённой памятью нужно учитывать разные факторы.

Компьютеры с общей памятью:

- *параллелизм вычислений;*
- *синхронизация доступа к общим данным.*

Компьютеры с распределённой памятью:

- *параллелизм вычислений;*
- *распределение данных;*
- *согласование параллелизма вычислений и распределения данных;*
- *организация пересылок данных.*

Технологии параллельного программирования

0. Распараллеливающий компилятор

Идея состоит в том, что компилятор достаточно интеллектуальный для того, чтобы мог сам извлекать параллелизм программ и отображать на архитектуру целевого параллельного компьютера.

Технологии параллельного программирования

1. Спецкомментарии

• *В чистом виде - компиляторы CRAY (CDIR$ NODEPCHK).*

• ***OpenMP*** *– стандарт для программирования на масштабируемых SMP-системах в модели общей памяти. Кроме спецкомментариев есть и языковые конструкции (вызов функций).*

• ***OpenACC*** *– стандарт, описывающий набор директив для написания гетерогенных программ, задействующих как центральный, так и графический процессор. Последняя версия стандарта – OpenACC 2.7. https://www.openacc.org*

Технологии параллельного программирования

2. Расширения существующих языков программирования

HPF (High Performance Fortran)*: ориентирован на создание переносимых программ. Отображение массив - массив-шаблон - виртуальный процессорный массив - физические процессоры. Есть и спецкомментарии, и языковые конструкции, например, оператор FORALL для параллельных циклов. Необходимые коммуникации и синхронизации реализуются компилятором. Сложность конструкций HPF оказалась непреодолимым препятствием для создания по-настоящему эффективных компиляторов. Есть система HPF Adaptor, переводящая программу с HPF на Fortran с MPI.*

Технологии параллельного программирования

2. Расширения существующих языков программирования

mpC *(ИСП РАН): язык программирования, основанный на языке C, предоставляющий средства создания параллельных программ для компьютеров с распределенной памятью. Ориентирован на неоднородные системы. Пользователь может задать топологию сети, распределение данных и вычислений и необходимые пересылки данных. Посылка сообщений организована с использованием интерфейса MPI. Информация, извлечённая из описания параллельного алгоритма, вместе с данными о реальной производительности процессоров и коммуникационных каналов, помогает системе программирования mpC найти эффективный способ отображения процессов mpC-программы на компьютеры сети.*

Технологии параллельного программирования

2. Расширения существующих языков программирования

*Языки **Fortran-DVM** и **C-DVM** (ИПМ РАН).*

DVM-система состоит из 5 основных компонентов: компиляторы, система поддержки выполнения параллельных программ, отладчик параллельных программ (сравнение промежуточных результатов), анализатор производительности, предсказатель производительности (предиктор).

- *высокоуровневая модель программирования;*
- *спецкомментарии, один вариант программы для параллельного и последовательного выполнения;*
- *основная работа по распределению вычислений и данных выполняется динамически системой поддержки;*
- *DVM-препроцессор переводит программу в Си (Фортран), расширенный функциями системы поддержки;*
- *для организации межпроцессного взаимодействия может использоваться одна из коммуникационных технологий (MPI, PVM, Router), что обеспечивает хорошую переносимость;*
- *директивы DVM: распределение данных, распределение вычислений, спецификация удалённых данных, редукционные операции.*

Технологии параллельного программирования

2. Расширения существующих языков программирования

***CAF (Co-Array Fortran)**:*

• небольшое расширение языка Fortran, достаточное для разработки эффективных параллельных программ;

• модель SPMD;

• понятие co-array (распределённого массива), компоненты которого распределены по процессам; доступ к распределённым компонентам осуществляется по правилам работы с обычными массивами.

***UPC (Unified Parallel C)**:*

• расширение языка программирования Си;

• модель SPMD;

• модель PGAS (Partitioned Global Address Space, глобальное разделённое адресное пространство) - адресуемая глобальная память в виде логических разделов, причем каждый из разделов локален для каждого из процессоров;

• синхронизация и обеспечение консистентности памяти.

•https://upc-lang.org/

Технологии параллельного программирования

2. Расширения существующих языков программирования

CUDA *(https://developer.nvidia.com) и* ***OpenCL*** *(http://opencl.org/) - расширения Си-подобных языков для программирования графических ускорителей.*

Linda *- параллельный язык программирования. Программа рассматривается как совокупность процессов, которые могут обмениваться данным через пространство кортежей. Используется совместно с другими языками высокого уровня как средство общения параллельных процессов.*

Технологии параллельного программирования

3. Специальные языки программирования

Occam *- язык параллельного программирования, ориентированный в первую очередь на написание программ для транспьютерных систем. Позволяет описать любую ЯПФ с помощью инструкций par, seq, отмечающих участки параллельных и последовательных процессов.*

НОРМА *(ИПМ РАН):*

- *декларативный язык, предназначенный для описания решения вычислительных задач сеточными методами;*
- *описание задач в нотации, близкой к исходной постановке проблемы математиком;*
- *не содержит традиционные конструкции языков программирования, фиксирующие порядок вычисления и/или иным образом «скрывающие/ограничивающие» параллелизм;*
- *язык с однократным присваиванием, нет конструкций вида X=X+1;*
- *распараллеливанием занимается компилятор, результат получается на Фортран+MPI, Фортран+PVM и др.*

Технологии параллельного программирования

3. Специальные языки программирования

Colamo *(НИИ МВС ЮФУ) - язык структурно-процедурного программирования высокого уровня реконфигурируемых вычислительных систем (на основе FPGA).*

Другие языки параллельного программирования: Chapel, X10, Fortress…

Технологии параллельного программирования

4. Библиотеки и интерфейсы, поддерживающие взаимодействие параллельных процессов

MPI (Message Passing Interface) *- хорошо стандартизованный механизм для построения программ по модели обмена сообщениями. Существуют стандартные «привязки» MPI к языкам C и Fortran. Существуют бесплатные и коммерческие реализации почти для всех суперкомпьютерных платформ, а также для сетей рабочих станций UNIX и Windows. В настоящее время MPI - наиболее широко используемый и динамично развивающийся интерфейс из своего класса.*

PVM (Parallel Virtual Machine) *- общедоступная библиотека, предоставляющая возможности управления процессами с помощью механизма передачи сообщений.*

Shmem *реализует схему работы над общей памятью с помощью операций Put/Get. Shmem включена в стандарт MPI 2.0 в качестве раздела «односторонние коммуникации».*

Технологии параллельного программирования

5. Средства и технологии для поддержки метакомпьютерных и распределённых вычислений

Globus, gLite, UNICORE, Condor, BOINC, X-Com, Map/Reduce…

Технологии параллельного программирования

6. Параллельные предметные библиотеки

PBLAS *- параллельные версии базовых процедур линейной алгебры (BLAS), уровней 1, 2, 3. Библиотека разработана в рамках проекта ScaLAPACK.*
ScaLAPACK *включает подмножество процедур LAPACK, переработанных для использования на параллельных компьютерах, включая: решение систем линейных уравнений, обращение матриц, ортогональные преобразования, поиск собственных значений и др.*
ATLAS - *оптимизированная библиотека, включающая реализацию процедур BLAS и часть функциональности LAPACK.*
MKL *- оптимизированная реализация BLAS от Intel.*
FFTW, DFFTPack *- быстрое преобразование Фурье.*
PETSc *- набор процедур и структур данных для параллельного решения научных задач с моделями, описываемыми в виде дифференциальных уравнений с частными производными.*
…

Технологии параллельного программирования

7. Специализированные пакеты и программные комплексы

FlowVision, ANSYS *для инженерный расчётов;*
GAMESS, GAUSSIAN, PRIRODA, GROMACS, CHARMM *для квантово-химических расчётов;*
OpenFOAM *для вычислительной гидродинамики;*
…

Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ (лекция 8)

А.С.Антонов

Вед. н.с. НИВЦ МГУ, к.ф.-м.н.

asa@parallel.ru

ВМК МГУ, 2020

Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ

(лекция 9)

А.С.Антонов

Вед. н.с. НИВЦ МГУ, к.ф.-м.н.

asa@parallel.ru

ВМК МГУ, 2020

Технология программирования OpenMP

OpenMP

OpenMP – технология параллельного программирования для компьютеров с общей памятью. Стандарт 5.0 принят в ноябре 2018 года.

Один вариант программы для параллельного и последовательного выполнения.

Любой процесс состоит из нескольких *нитей управления*, которые имеют общее адресное пространство, но разные потоки команд и раздельные стеки.

OpenMP

Макрос **_OPENMP** определён в формате **yyyymm**, где **yyyy** и **mm** – цифры года и месяца, когда был принят поддерживаемый стандарт OpenMP.

Условная компиляция:

```
#include <stdio.h>
int main(){
#ifdef _OPENMP
   printf("OpenMP is supported!\n");
#endif
}
```

OpenMP

Распараллеливание в OpenMP: вставка в текст программы специальных директив, а также вызов вспомогательных функций.

SPMD-модель (Single Program Multiple Data) параллельного программирования: для всех параллельных нитей используется один и тот же код.

OpenMP

Программа начинается с *последовательной области* – работает одна нить, при входе в *параллельную область* порождается ещё несколько нитей, между которыми распределяются части кода.

По завершении параллельной области все нити, кроме одной (нити-мастера), завершаются.

Любое количество параллельных и последовательных областей. Параллельные области могут быть вложенными друг в друга.

OpenMP

В OpenMP переменные в параллельных областях программы разделяются на два основных класса:

`shared` (*общие*; все нити видят одну и ту же переменную);

`private` (*локальные*, приватные; каждая нить видит свой экземпляр данной переменной).

По умолчанию, все переменные, порождённые вне параллельной области, при входе в неё остаются общими. Исключение составляют переменные, являющиеся счётчиками итераций в цикле. Переменные, порождённые внутри параллельной области, по умолчанию являются локальными.

OpenMP

Директивы OpenMP в языке Си задаются указаниями препроцессору, начинающимися с **`#pragma omp`**

`#pragma omp` ***`directive-name`***
`[опция[[,] опция]...]`

Объектом действия большинства директив является один оператор или блок, перед которым расположена директива в исходном тексте программы.

OpenMP

Чтобы задействовать *функции библиотеки OpenMP периода выполнения* (исполняющей среды), в программу нужно включить заголовочный файл **`omp.h`** (для программ на языке Фортран – файл **`omp_lib.h`** или модуль **`omp_lib`**).

Нужно задать количество нитей, выполняющих параллельные области программы, определив значение переменной среды **`OMP_NUM_THREADS`**:

```
export OMP_NUM_THREADS=n
```

OpenMP

Функции для работы с системным таймером:

```
double omp_get_wtime(void);
```

Возвращает астрономическое время в секундах, прошедшее с некоторого момента в прошлом.

```
double omp_get_wtick(void);
```

Возвращает в вызвавшей нити разрешение таймера в секундах.

OpenMP

Параллельные и последовательные области:

```
#pragma omp parallel [опция[[,] опция]...]
```

Порождаются новые **OMP_NUM_THREADS-1** нитей, каждая нить получает свой уникальный номер, причём порождающая нить получает номер **0** и становится основной нитью группы («мастером»). При выходе из параллельной области производится неявная синхронизация и уничтожаются все нити, кроме породившей.

OpenMP

`if(условие)` – выполнение параллельной области по условию;

`num_threads (целочисленное выражение)` – явное задание количества нитей, которые будут выполнять параллельную область;

`default(shared|none)` – всем переменным в параллельной области, которым явно не назначен класс, будет назначен класс **`shared`**; **`none`** – всем переменным класс назначается явно;

`private(список)` – переменные, для которых порождается локальная копия в каждой нити; начальное значение не определено;

OpenMP

`firstprivate(список)` – переменные, для которых порождается локальная копия в каждой нити; локальные копии инициализируются значениями этих переменных в нити-мастере;

`shared(список)` – переменные, общие для всех нитей;

OpenMP

`reduction(оператор:список)` – задаёт оператор и список общих переменных; для каждой переменной создаются локальные копии в каждой нити; они инициализируются соответственно типу оператора (для аддитивных – **`0`** или аналоги, для мультипликативных – **`1`** или аналоги); после выполнения всех операторов параллельной области выполняется заданный оператор; **`оператор`** это: для языка Си – **`+`**, **`*`**, **`-`**, **`&`**, **`|`**, **`^`**, **`&&`**, **`||`**; порядок выполнения операторов не определён, поэтому результат может отличаться от запуска к запуску.

OpenMP

```
#include <stdio.h>
int main(int argc, char *argv[])
{
   printf("Последовательная область 1\n");
#pragma omp parallel
   {
      printf("Параллельная область\n");
   }
   printf("Последовательная область 2\n");
}
```

OpenMP

```
#include <stdio.h>
int main(int argc, char *argv[])
{
   int count = 0;
#pragma omp parallel reduction (+: count)
   {
      count++;
      printf("Текущее значение count: %d\n",
count);
   }
   printf("Число нитей: %d\n", count);
}
```

OpenMP

Если внутри параллельной области содержится только один параллельный цикл, одна конструкция **`sections`** или одна конструкция **`workshare`**, то можно использовать укороченную запись: **`parallel for`**, **`parallel sections`** или **`parallel workshare`**.

OpenMP

```
void omp_set_num_threads(int num);

#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   omp_set_num_threads(2);
#pragma omp parallel num_threads(3)
   {
      printf("Параллельная область 1\n");
   }
#pragma omp parallel
   {
      printf("Параллельная область 2\n");
   }
}
```

OpenMP

Параллельные области могут быть вложенными; по умолчанию вложенная параллельная область выполняется одной нитью. Это управляется установкой переменной среды **OMP_NESTED**.

```
export OMP_NESTED=true
```

Изменить значение переменной **OMP_NESTED** можно с помощью вызова функции **omp_set_nested()**.

```
void omp_set_nested(int nested);
```

Узнать значение переменной **OMP_NESTED** можно при помощи функции **omp_get_nested()**.

```
int omp_get_nested(void);
```

OpenMP

Функция **omp_in_parallel()** возвращает **1**, если она была вызвана из активной параллельной области программы.

```
int omp_in_parallel(void);
```

```
void mode(void){
   if(omp_in_parallel()) printf("Параллельная область\n");
   else printf("Последовательная область\n");
}

int main(int argc, char *argv[]){
   mode();
#pragma omp parallel
   mode();
}
```

OpenMP

Если в параллельной области какой-либо участок кода должен быть выполнен лишь один раз, то его нужно выделить директивами **single**:

```
#pragma omp single [опция [[,] опция]...]
```

private(список);

firstprivate(список);

copyprivate(список) – новые значения переменных списка будут доступны всем одноименным частным переменным (**private** и **firstprivate**); опция не может использоваться совместно с опцией **nowait**; переменные списка не должны быть перечислены в опциях **private** и **firstprivate** данной директивы **single**;

OpenMP

nowait – после выполнения участка происходит неявная барьерная синхронизация нитей: их дальнейшее выполнение происходит только тогда, когда все достигнут данной точки; если в этом нет необходимости, опция **nowait** позволяет нитям, дошедшим до конца участка, продолжить выполнение без синхронизации.

```
#pragma omp parallel
    {
        printf("Сообщение 1\n");
#pragma omp single nowait
        {
            printf("Одна нить\n");
        }
        printf("Сообщение 2\n");
    }
```

OpenMP

Директива **master** выделяет участок кода, который будет выполнен только нитью-мастером. Остальные нити пропускают данный участок и продолжают работу с оператора, расположенного следом. Неявной синхронизации не предполагает.

```
#pragma omp master
```

```
#pragma omp parallel private(n)
    {
        n=1;
#pragma omp master
        n=2;
        printf("значение n: %d\n", n);
    }
```

OpenMP

Все нити в параллельной области нумеруются последовательными целыми числами от **0** до **N-1**, где **N** — количество нитей, выполняющих данную область.

Вызов функции **omp_get_thread_num()** позволяет нити получить свой уникальный номер в текущей параллельной области.

```
int omp_get_thread_num(void);
```

Вызов функции **omp_get_num_threads()** позволяет нити получить количество нитей в текущей параллельной области.

```
int omp_get_num_threads(void);
```

OpenMP

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   int n=1;
   printf("в посл. области (1): %d\n", n);
#pragma omp parallel private(n)
   {
      printf("Значение n на нити (1): %d\n", n);
/* Присвоим переменной n номер текущей нити */
      n=omp_get_thread_num();
      printf("Значение n на нити (2): %d\n", n);
   }
   printf("в посл. области (2): %d\n", n);
}
```

OpenMP

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   int n=1;
   printf("в посл. области (1): %d\n", n);
#pragma omp parallel firstprivate(n)
   {
      printf("Значение n на нити (1): %d\n", n);
/* Присвоим переменной n номер текущей нити */
      n=omp_get_thread_num();
      printf("Значение n на нити (2): %d\n", n);
   }
   printf("в посл. области (2): %d\n", n);
}
```

OpenMP

Если в параллельной области встретился оператор цикла, то, согласно общему правилу, он будет выполнен всеми нитями текущей группы, то есть, каждая нить выполнит все итерации данного цикла. Для распределения итераций цикла между различными нитями можно использовать директиву **for**.

```
#pragma omp for [опция [[,] опция]...]
```

Эта директива относится к идущему следом за данной директивой блоку, включающему операторы **for**.

OpenMP

`private(список)`;

`firstprivate(список)`;

`lastprivate(список)` – переменным, перечисленным в списке, присваивается результат с последнего витка цикла;

`reduction(оператор:список)`;

`schedule(type[, chunk])` – опция задаёт, каким образом итерации цикла распределяются между нитями;

OpenMP

`collapse(n)` — опция указывает, что **`n`** последовательных тесновложенных циклов ассоциируется с данной директивой; для циклов образуется общее пространство итераций, которое делится между нитями; если опция **`collapse`** не задана, то директива относится только к одному непосредственно следующему за ней циклу;

`ordered` – опция, говорящая о том, что в цикле могут встречаться директивы **`ordered`**; в этом случае определяется блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле;

`nowait`.

OpenMP

На вид параллельных циклов накладываются достаточно жёсткие ограничения. В частности, предполагается, что корректная программа не должна зависеть от того, какая именно нить какую итерацию параллельного цикла выполнит. Нельзя использовать побочный выход из параллельного цикла. Размер блока итераций, указанный в опции **`schedule`**, не должен изменяться в рамках цикла.

Формат параллельных циклов упрощённо можно представить следующим образом:

```
for([целочисленный тип] i =
инвариант цикла; i {<,>,=,<=,>=}
инвариант цикла; i {+,-}= инвариант
цикла)
```

OpenMP

Итеративная переменная распределяемого цикла по смыслу должна быть локальной, поэтому в случае, если она специфицирована общей, то она неявно делается локальной при входе в цикл. После завершения цикла значение итеративной переменной цикла не определено, если она не указана в опции **`lastprivate`**.

OpenMP

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
 int A[10], B[10], C[10], i, n;
 for (i=0; i<10; i++){A[i]=i; B[i]=2*i; C[i]=0;}
#pragma omp parallel shared(A,B,C) private(i,n)
 {
  n=omp_get_thread_num();
#pragma omp for
  for (i=0; i<10; i++){
   C[i]=A[i]+B[i];
   printf("Нить %d сложила элементы %d\n",n,i);
  }
 }
}
```

OpenMP

В опции **`schedule`** параметр **`type`** задаёт следующий тип распределения итераций:

`static` – блочно-циклическое распределение итераций цикла; размер блока – **`chunk`**. Первый блок из **`chunk`** итераций выполняет нулевая нить, второй блок — следующая и т.д. до последней нити, затем распределение снова начинается с нулевой нити. Если значение **`chunk`** не указано, то всё множество итераций делится на непрерывные куски примерно одинакового размера (конкретный способ зависит от реализации), и полученные порции итераций распределяются между нитями.

OpenMP

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
 int i;
#pragma omp parallel private(i)
 {
#pragma omp for schedule (static, 2)
  for (i=0; i<10; i++){
   printf("Нить %d выполнила итерацию %d\n",
                  omp_get_thread_num(), i);
   sleep(1);
  }
 }
}
```

OpenMP

`dynamic` – динамическое распределение итераций с фиксированным размером блока: сначала каждая нить получает **`chunk`** итераций (по умолчанию **`chunk=1`**), та нить, которая заканчивает выполнение своей порции итераций, получает первую свободную порцию из **`chunk`** итераций. Освободившиеся нити получают новые порции итераций до тех пор, пока все порции не будут исчерпаны. Последняя порция может содержать меньше итераций, чем все остальные.

OpenMP

`guided` – динамическое распределение итераций, при котором размер порции уменьшается с некоторого начального значения до величины **`chunk`** (по умолчанию **`chunk=1`**) пропорционально количеству ещё не распределённых итераций, делённому на количество нитей, выполняющих цикл. Размер первоначально выделяемого блока зависит от реализации. В ряде случаев такое распределение позволяет аккуратнее разделить работу и сбалансировать загрузку нитей. Количество итераций в последней порции может оказаться меньше значения **`chunk`**.

OpenMP

`auto` – способ распределения итераций выбирается компилятором и/или системой выполнения. Параметр **`chunk`** при этом не задаётся.

`runtime` – способ распределения итераций выбирается во время работы программы по значению переменной среды **`OMP_SCHEDULE`**. Параметр **`chunk`** при этом не задаётся.

OpenMP

Значение по умолчанию переменной **OMP_SCHEDULE** зависит от реализации. Если переменная задана неправильно, то поведение программы при задании опции **runtime** также зависит от реализации.

Задать значение переменной **OMP_SCHEDULE** в Linux в командной оболочке **bash** можно при помощи команды следующего вида:

```
export OMP_SCHEDULE="dynamic,1"
```

OpenMP

Изменить значение переменной **OMP_SCHEDULE** из программы можно с помощью вызова функции **omp_set_schedule()**.

```
void omp_set_schedule(omp_sched_t type, int chunk);
```

Допустимые значения констант описаны в файле **omp.h**. Как минимум, следующие варианты:

```
typedef enum omp_sched_t {
omp_sched_static = 1,
omp_sched_dynamic = 2,
omp_sched_guided = 3,
omp_sched_auto = 4
} omp_sched_t;
```

OpenMP

При помощи вызова функции **`omp_get_schedule()`** пользователь может узнать текущее значение переменной **`OMP_SCHEDULE`**.

```
void omp_get_schedule(omp_sched_t* type, int* chunk);
```

При распараллеливании цикла нужно убедиться в том, что итерации данного цикла не имеют информационных зависимостей. В этом случае его итерации можно выполнять в любом порядке, в том числе параллельно. Компилятор это не проверяет. Если дать указание компилятору распараллелить цикл, содержащий зависимости, результат работы может оказаться некорректным.

OpenMP

Директива **sections** определяет набор независимых секций кода, каждая из которых выполняется своей нитью.

#pragma omp sections [опция [[,] опция]...]

private(список);

firstprivate(список);

lastprivate(список) – переменным присваивается результат из последней секции;

reduction(оператор:список);

nowait.

OpenMP

Директива **`section`** задаёт участок кода внутри секции **`sections`** для выполнения одной нитью.

```
#pragma omp section
```

Перед первым участком кода в блоке **`sections`** директива **`section`** не обязательна. Какие нити будут задействованы для выполнения какой секции, не специфицируется. Если количество нитей больше количества секций, то часть нитей для выполнения данного блока секций не будет задействована. Если количество нитей меньше количества секций, то некоторым (или всем) нитям достанется более одной секции.

OpenMP

```
 int n;
#pragma omp parallel private(n)
  {
    n=omp_get_thread_num();
#pragma omp sections
    {
#pragma omp section
      printf("Первая секция, процесс %d\n", n);
#pragma omp section
      printf("Вторая секция, процесс %d\n", n);
    }
   printf("Параллельная область, процесс %d\n",
n);
  }
```

OpenMP

```
 int n=0;
#pragma omp parallel
 {
#pragma omp sections lastprivate(n)
  {
#pragma omp section
   n=1;
#pragma omp section
   n=2;
  }
  printf("Значение n на нити %d: %d\n",
              omp_get_thread_num(), n);
 }
 printf("Значение n в конце: %d\n", n);
```

OpenMP

Директива **`task`** применяется для выделения отдельной независимой задачи.

```
#pragma omp task [опция [[,] опция]...]
```

Текущая нить выделяет в качестве задачи ассоциированный с директивой блок операторов. Задача может выполняться немедленно после создания или быть отложенной на неопределённое время и выполняться по частям. Размер таких частей, а также порядок выполнения частей разных отложенных задач определяется реализацией.

OpenMP

`if(условие)` — порождение новой задачи только при выполнении некоторого условия; если условие не выполняется, то задача будет выполнена текущей нитью и немедленно;

`untied` — опция означает, что в случае откладывания задача может быть продолжена любой нитью из числа выполняющих данную параллельную область; если данная опция не указана, то задача может быть продолжена только породившей её нитью;

OpenMP

```
default(shared|none);
private(список);
firstprivate(список);
shared(список).
```

OpenMP

Для гарантированного завершения в точке вызова всех запущенных задач используется директива **`taskwait`**.

```
#pragma omp taskwait
```

Нить, выполнившая данную директиву, приостанавливается до тех пор, пока не будут завершены все ранее запущенные данной нитью независимые задачи.

OpenMP

Самый распространенный способ синхронизации в OpenMP – барьер. Он оформляется с помощью директивы **`barrier`**.

```
#pragma omp barrier
```

Нити, выполняющие текущую параллельную область, дойдя до этой директивы, останавливаются и ждут, пока все нити не дойдут до этой точки программы, после чего разблокируются и продолжают работать дальше. Кроме того, для разблокировки необходимо, чтобы все синхронизируемые нити завершили все порождённые ими задачи (**`task`**).

OpenMP

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
#pragma omp parallel
    {
        printf("Сообщение 1\n");
        printf("Сообщение 2\n");
#pragma omp barrier
        printf("Сообщение 3\n");
    }
}
```

OpenMP

Директива **`ordered`** определяет блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле.

```
#pragma omp ordered
```

Относится к самому внутреннему из объемлющих циклов, а в параллельном цикле должна быть задана опция **`ordered`**. Нить, выполняющая первую итерацию цикла, выполняет операции данного блока. Нить, выполняющая любую следующую итерацию, должна сначала дождаться выполнения всех операций блока всеми нитями, выполняющими предыдущие итерации. Может использоваться, например, для упорядочения вывода от параллельных нитей.

OpenMP

```
#pragma omp parallel private (i, n)
   {
      n=omp_get_thread_num();
#pragma omp for ordered
      for (i=0; i<5; i++)
      {
         printf("Нить %d, итерация %d\n", n, i);
#pragma omp ordered
         {
            printf("ordered: Нить %d, итерация
%d\n", n, i);
         }
      }
   }
}
```

OpenMP

С помощью директивы **`critical`** оформляется критическая секция программы.

```
#pragma omp critical
[(<имя_критической_секции>)]
```

В каждый момент времени в критической секции может находиться не более одной нити. Все другие нити, выполнившие директиву для секции с данным именем, будут заблокированы, пока вошедшая нить не закончит выполнение. Как только работавшая нить выйдет из критической секции, одна из заблокированных нитей войдет в неё. Если на входе стояло несколько нитей, то случайным образом выбирается одна из них, а остальные продолжают ожидание.

OpenMP

Все неименованные критические секции условно ассоциируются с одним и тем же именем. Имеющие одно и тоже имя рассматриваются единой секцией, даже если находятся в разных параллельных областях. Побочные входы и выходы из критической секции запрещены.

```
 int n;
#pragma omp parallel
    {
#pragma omp critical
        {
            n=omp_get_thread_num();
            printf("Нить %d\n", n);
        }
    }
}
```

OpenMP

`#pragma omp atomic`

Данная директива относится к идущему непосредственно за ней оператору присваивания (на используемые в котором конструкции накладываются достаточно понятные ограничения), гарантируя корректную работу с общей переменной, стоящей в его левой части. На время выполнения оператора блокируется доступ к данной переменной всем запущенным в данный момент нитям, кроме нити, выполняющей операцию. Атомарной является только работа с переменной в левой части оператора присваивания, при этом вычисления в правой части не обязаны быть атомарными.

OpenMP

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
    int count = 0;
#pragma omp parallel
        {
#pragma omp atomic
            count++;
        }
        printf("Число нитей: %d\n", count);
}
```

OpenMP

Один из вариантов синхронизации в OpenMP реализуется через механизм *замков* (*locks*). В качестве замков используются общие переменные. Данные переменные должны использоваться только как параметры примитивов синхронизации.

Замок может находиться в одном из трёх состояний: *неинициализированный*, *разблокированный* или *заблокированный*. Разблокированный замок может быть захвачен некоторой нитью. При этом он переходит в заблокированное состояние. Нить, захватившая замок, и только она может его освободить, после чего замок возвращается в разблокированное состояние.

OpenMP

Есть два типа замков: *простые замки* и *множественные замки*. Множественный замок может многократно захватываться одной нитью перед его освобождением, в то время как простой замок может быть захвачен только однажды. Для множественного замка вводится понятие *коэффициента захваченности (nesting count)*. Изначально он устанавливается в ноль, при каждом следующем захватывании увеличивается на единицу, а при каждом освобождении уменьшается на единицу. Множественный замок считается разблокированным, если его коэффициент захваченности равен нулю.

OpenMP

Для инициализации простого или множественного замка используются соответственно функции **`omp_init_lock()`** и **`omp_init_nest_lock()`**.

```
void omp_init_lock(omp_lock_t
*lock);
```

```
void
omp_init_nest_lock(omp_nest_lock_t
*lock);
```

После выполнения функции замок переводится в разблокированное состояние. Для множественного замка коэффициент захваченности устанавливается в ноль.

OpenMP

Функции **omp_destroy_lock()** и **omp_destroy_nest_lock()** используются для переведения простого или множественного замка в неинициализированное состояние.

```
void omp_destroy_lock(omp_lock_t
*lock);
void omp_destroy_nest_lock
(omp_nest_lock_t *lock);
```

OpenMP

Для захватывания замка используются функции **`omp_set_lock()`** и **`omp_set_nest_lock()`**.

```
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock
(omp_nest_lock_t *lock);
```

Вызвавшая эту функцию нить дожидается освобождения замка, а затем захватывает его. Замок при этом переводится в заблокированное состояние. Если множественный замок уже захвачен данной нитью, то нить не блокируется, а коэффициент захваченности увеличивается на единицу.

OpenMP

Для освобождения замка используются функции **`omp_unset_lock()`** и **`omp_unset_nest_lock()`**.

```
void omp_unset_lock(omp_lock_t
*lock);
void omp_unset_nest_lock(omp_lock_t
*lock);
```

Вызов этой функции освобождает простой замок, если он был захвачен вызвавшей нитью. Для множественного замка уменьшает на единицу коэффициент захваченности. Если коэффициент станет равен 0, замок освобождается. Если после освобождения есть нити, заблокированные на операции, захватывающей данный замок, он будет сразу захвачен одной из ожидающих нитей.

OpenMP

```
   omp_lock_t lock;
   int n;
   omp_init_lock(&lock);
#pragma omp parallel private (n)
   {
      n=omp_get_thread_num();
      omp_set_lock(&lock);
      printf("Начало закрытой секции, %d\n", n);
      sleep(5);
      printf("Конец закрытой секции, %d\n", n);
      omp_unset_lock(&lock);
   }
   omp_destroy_lock(&lock);
```

OpenMP

Для неблокирующей попытки захвата замка используются функции **`omp_test_lock()`** и **`omp_test_nest_lock()`**.

```
int omp_test_lock(omp_lock_t *lock);
int omp_test_nest_lock(omp_lock_t
*lock);
```

Данная функция пробует захватить указанный замок. Если это удалось, то для простого замка функция возвращает 1, а для множественного замка – новый коэффициент захваченности. Если замок захватить не удалось, в обоих случаях возвращается 0.

OpenMP

```
   omp_lock_t lock;
   int n;
   omp_init_lock(&lock);
#pragma omp parallel private (n)
   {
      n=omp_get_thread_num();
      while (!omp_test_lock (&lock)){
         printf("Секция закрыта, %d\n", n);
         sleep(2);
      }
      printf("Начало закрытой секции, %d\n", n);
      sleep(5);
      printf("Конец закрытой секции, %d\n", n);
      omp_unset_lock(&lock);
   }
   omp_destroy_lock(&lock);
```

OpenMP

Литература

1. Антонов А.С. Технологии параллельного программирования MPI и OpenMP: Учеб. пособие. Предисл.: В.А.Садовничий. - М.: Издательство Московского университета, 2012.-344 с.- (Серия "Суперкомпьютерное образование").
2. Антонов А.С. Параллельное программирование с использованием технологии OpenMP: Учебное пособие. -М.: Изд-во МГУ, 2009.-77 с.
3. OpenMP Architecture Review Board (http://www.openmp.org/).
4. The Community of OpenMP Users, Researchers, Tool Developers and Providers (http://www.compunity.org/).
5. OpenMP 5.0 Specification (November 2018) (https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf).
6. Barbara Chapman, Gabriele Jost, Ruud van der Pas. Using OpenMP: portable shared memory parallel programming (Scientific and Engineering Computation). Cambridge, Massachusetts: The MIT Press., 2008. - 353 pp.

Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ

(лекция 9)

А.С.Антонов

Вед. н.с. НИВЦ МГУ, к.ф.-м.н.

asa@parallel.ru

ВМК МГУ, 2020

Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ
(лекция 10)

А.С.Антонов

Вед. н.с. НИВЦ МГУ, к.ф.-м.н.

asa@parallel.ru

ВМК МГУ, 2020

Технология программирования MPI

MPI

- MPI - *Message Passing Interface*, интерфейс передачи сообщений.

- Стандарт MPI 3.1 (4 июня 2015 года).
- Более 450 процедур.

- SPMD (Single Program Multiple Data) - основная модель параллельного программирования.

MPI

- Префикс **`MPI_`** .

`#include “mpi.h”`

(**`mpif.h`** для языка Фортран)

- *Процессы*, посылка сообщений.
- *Сообщение* – массив однотипных данных, расположенных в последовательных ячейках памяти.
- *Группа* – упорядоченное множество процессов.

MPI

Коммуникатор – контекст обмена группы.

В операциях обмена используются только коммуникаторы!

Коммуникаторы имеют в языке Си предопределённый тип **`MPI_Comm`** (в языке Фортран – тип **`INTEGER`**).

MPI

MPI_COMM_WORLD – коммуникатор для всех процессов приложения.

MPI_COMM_SELF – коммуникатор, включающий только текущий процесс.

MPI_COMM_NULL – коммуникатор, не содержащий ни одного процесса.

MPI

Каждый процесс может одновременно входить в разные коммуникаторы.

Два основных атрибута процесса: коммуникатор (группа) и номер процесса в коммуникаторе (группе).

Если коммуникатор содержит **`n`** процессов, то номера процессов в нём лежат в пределах от **`0`** до **`n-1`**.

MPI

- *Сообщение* — набор данных некоторого типа.
- Атрибуты сообщения: номер процесса-отправителя, номер процесса-получателя, идентификатор сообщения, коммуниктор.
- Идентификатор сообщения - целое неотрицательное число в диапазоне от **`0`** до **`MPI_TAG_UB`** (не меньше **`32767`**).
- Для работы с атрибутами сообщений введена структура **`MPI_Status`**.

MPI

Возвращаемым значением (в Фортране – последним аргументом) для большинства процедур MPI является информация об успешности завершения.

В случае успешного выполнения процедура вернёт значение **`MPI_SUCCESS`**, иначе - код ошибки.

Предопределённые значения, соответствующие различным ошибочным ситуациям, перечислены в файле **`mpi.h`**

MPI

Общие процедуры MPI

MPI

```
int MPI_Init(int *argc, char ***argv)
```

Инициализация параллельной части программы. Почти все другие процедуры MPI могут быть вызваны только после вызова **`MPI_Init`**. Инициализация параллельной части для каждого приложения должна выполняться только один раз.

MPI

```
int MPI_Finalize(void)
```

Завершение параллельной части приложения. Все последующие обращения к большинству процедур MPI, в том числе к **`MPI_Init`**, запрещены. К моменту вызова **`MPI_Finalize`** каждым процессом программы все действия, требующие его участия в обмене сообщениями, должны быть завершены.

MPI

Простейшая параллельная программа:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
    printf("Before MPI_INIT\n");
    MPI_Init(&argc, &argv);
    printf("Parallel section\n");
    MPI_Finalize();
    printf("After MPI_FINALIZE\n");
}
```

MPI

`int MPI_Initialized(int *flag)`

В аргументе **`flag`** возвращает **`1`**, если вызвана после процедуры **`MPI_Init`**, и **`0`** в противном случае.

`int MPI_Finalized(int *flag)`

В аргументе **`flag`** возвращает **`1`**, если вызвана после процедуры **`MPI_Finalize`**, и **`0`** в противном случае.

Эти процедуры можно вызвать до **`MPI_Init`** и после **`MPI_Finalize`**.

MPI

```
int MPI_Comm_size(MPI_Comm comm,
int *size)
```

В аргументе **size** возвращает число параллельных процессов в коммуникаторе **comm**.

```
int MPI_Comm_rank(MPI_Comm comm,
int *rank)
```

В аргументе **rank** возвращает номер процесса в коммуникаторе **comm** в диапазоне от **0** до **size-1**.

MPI

```
#include <stdio.h>
#include "mpi.h"
#define MAX 100
int main(int argc, char **argv)
{
   int rank, size, n, i, ibeg, iend;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   n=(MAX-1)/size+1;
   ibeg=rank*n+1; iend=(rank+1)*n;
   for(i=ibeg; i<=((iend>MAX)?MAX:iend); i++)
      printf ("process %d, %d^2=%d\n", rank, i, i*i);
   MPI_Finalize();
}
```

MPI

`double MPI_Wtime(void)`

Возвращает для каждого вызвавшего процесса астрономическое время в секундах (вещественное число двойной точности), прошедшее с некоторого момента в прошлом. Момент времени, используемый в качестве точки отсчёта, не будет изменён за время существования процесса.

`double MPI_Wtick(void)`

Возвращает разрешение таймера в секундах.

MPI

```
int MPI_Get_processor_name(char *name, int *len)
```

Возвращает в строке **name** имя узла, на котором запущен вызвавший процесс. В переменной **len** возвращается количество символов в имени, не превышающее константы **MPI_MAX_PROCESSOR_NAME**.

MPI

Определение характеристик системного таймера:

```
#include <stdio.h>
#include "mpi.h"
#define NTIMES 100
int main(int argc, char **argv)
{
   double time_start, time_finish, tick;
   int rank, i;
   int len;
   char *name;
   name =
(char*)malloc(MPI_MAX_PROCESSOR_NAME*sizeof(char));
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Get_processor_name(name, &len);
```

MPI

```
   tick = MPI_Wtick();
   time_start = MPI_Wtime();
   for (i = 0; i<NTIMES; i++)
      time_finish = MPI_Wtime();
   printf ("node %s, process %d: tick= %lf, time= %lf\n",
name, rank, tick, (time_finish-time_start)/NTIMES);
   MPI_Finalize();
}
```

MPI

Передача и приём сообщений с блокировкой

MPI

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int msgtag, MPI_Comm comm)
```

Блокирующая посылка массива **buf** с идентификатором **msgtag**, состоящего из **count** элементов типа **datatype**, процессу с номером **dest** в коммуникаторе **comm**.

MPI

Типы данных:

- **MPI_INT – int**
- **MPI_SHORT – short**
- **MPI_LONG – long**
- **MPI_FLOAT – float**
- **MPI_DOUBLE – double**
- **MPI_CHAR – char**
- **MPI_BYTE –** 8 бит
- **MPI_PACKED –** тип для упакованных данных.

Все типы данных перечислены в файле **mpi.h**.

MPI

Модификации процедуры **MPI_Send**:

- **MPI_Bsend** — передача сообщения с буферизацией.
- **MPI_Ssend** — передача сообщения с синхронизацией.
- **MPI_Rsend** — передача сообщения по готовности.

MPI

```
int MPI_Buffer_attach(void* buf,
int size)
```

Назначение массива **`buf`** размера **`size`** для использования при посылке сообщений с буферизацией. В каждом процессе может быть только один такой буфер.

Размер массива, выделяемого для буферизации, должен превосходить общий размер сообщения как минимум на величину, определяемую константой **`MPI_BSEND_OVERHEAD`**.

MPI

```
int MPI_Buffer_detach(buf, size)
```

Освобождение массива **`buf`** для других целей. Процесс блокируется до того момента, когда все сообщения уйдут из данного буфера.

MPI

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int msgtag, MPI_Comm comm, MPI_Status status)
```

Блокирующий приём в буфер **buf** не более **count** элементов сообщения типа **datatype** с идентификатором **msgtag** от процесса с номером **source** в коммуникаторе **comm** с заполнением структуры атрибутов приходящего сообщения **status**.

MPI

Вместо аргументов **`source`** и **`msgtag`** можно использовать константы:

- **`MPI_ANY_SOURCE`** — признак того, что подходит сообщение от любого процесса;
- **`MPI_ANY_TAG`** — признак того, что подходит сообщение с любым идентификатором.

MPI

Параметры принятого сообщения всегда можно определить по соответствующим элементам структуры **`status`**:

- **`status.MPI_SOURCE`** — номер процесса-отправителя;
- **`status.MPI_TAG`** — идентификатор сообщения;
- **`status.MPI_ERROR`** — код ошибки.

MPI

Если один процесс последовательно посылает два сообщения, соответствующие одному и тому же вызову **`MPI_Recv`**, другому процессу, то первым будет принято сообщение, которое было отправлено раньше.

Если два сообщения были одновременно отправлены разными процессами, то порядок их получения принимающим процессом заранее не определён.

MPI

```
int MPI_Get_count(MPI_Status
*status, MPI_Datatype datatype,
int *count)
```

По значению параметра **status** определяет число **count** уже принятых (после обращения к **MPI_Recv**) или принимаемых (после обращения к **MPI_Probe** или **MPI_Iprobe**) элементов сообщения типа **datatype.**

MPI

```
int MPI_Probe(int source, int
msgtag, MPI_Comm comm,
MPI_Status *status)
```

Получение в параметре **`status`** информации о структуре ожидаемого сообщения с блокировкой. Возврата не произойдёт, пока сообщение с подходящим идентификатором и номером процесса-отправителя не будет доступно для получения.

MPI

Тупиковые ситуации (deadlock):

процесс 0:	процесс 1:
`Recv(1)`	`Recv(0)`
`Send(1)`	`Send(0)`

процесс 0:	процесс 1:
`Send(1)`	`Send(0)`
`Recv(1)`	`Recv(0)`

MPI

Разрешение тупиковых ситуаций:

1.

процесс 0:	процесс 1:
`Send(1)`	`Recv(0)`
`Recv(1)`	`Send(0)`

2. Использование неблокирующих операций

3. Использование функции `MPI_Sendrecv`

MPI

Обмен сообщениями чётных и нечётных процессов:

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
   int size, rank, a, b;
   MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   a = rank;
   b = -1;
```

MPI

```
   if((rank%2) == 0){
      if(rank<size-1)
         MPI_Send(&a, 1, MPI_INT, rank+1, 5,
MPI_COMM_WORLD);
   }
   else
      MPI_Recv(&b, 1, MPI_INT, rank-1, 5, MPI_COMM_WORLD,
&status);
   printf("process %d a = %d, b = %d\n", rank, a, b);
   MPI_Finalize();
}
```

MPI

Передача и приём сообщений без блокировки

MPI

```
int MPI_Isend(void *buf, int
count, MPI_Datatype datatype,
int dest, int msgtag, MPI_Comm
comm, MPI_Request *request)
```

Неблокирующая посылка сообщения. Возврат из функции происходит сразу после инициализации передачи. Переменная **`request`** идентифицирует пересылку.

MPI

Модификации функции **MPI_Isend**:

- **MPI_Ibsend** — передача сообщения с буферизацией;
- **MPI_Issend** — передача сообщения с синхронизацией;
- **MPI_Irsend** — передача сообщения по готовности.

MPI

```
int MPI_Irecv(void *buf, int
count, MPI_Datatype datatype,
int source, int msgtag, MPI_Comm
comm, MPI_Request *request)
```

Неблокирующий приём сообщения. Возврат из функции происходит сразу после инициализации передачи. Переменная **`request`** идентифицирует пересылку.

MPI

Сообщение, отправленное любой из процедур **`MPI_Send`**, **`MPI_Isend`** и любой из трёх их модификаций, может быть принято любой из процедур **`MPI_Recv`** и **`MPI_Irecv`**.

До завершения неблокирующей операции не следует записывать в используемый массив данных!

MPI

```
int MPI_Iprobe(int source, int msgtag, MPI_Comm comm, int *flag, MPI_Status *status)
```

Получение информации о структуре ожидаемого сообщения без блокировки. В аргументе **`flag`** возвращается значение **`1`**, если сообщение с подходящими атрибутами уже может быть принято.

MPI

```
int MPI_Wait(MPI_Request
*request, MPI_Status *status)
```

Ожидание завершения асинхронной операции, ассоциированной с идентификатором **request.** Для неблокирующего приёма определяется параметр **status. request** устанавливается в значение **MPI_REQUEST_NULL.**

MPI

```
int MPI_Waitall(int count,
MPI_Request *requests,
MPI_Status *statuses)
```

Ожидание завершения **`count`** асинхронных операций, ассоциированных с идентификаторами **`requests`**. Для неблокирующих приёмов определяются параметры в массиве **`statuses`**.

MPI

Обмен по кольцевой топологии при помощи неблокирующих операций:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
   int rank, size, prev, next;
   int buf[2];
   MPI_Request reqs[4];
   MPI_Status stats[4];
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

MPI

```
   prev = rank - 1;
   next = rank + 1;
   if(rank==0) prev = size - 1;
   if(rank==size - 1) next = 0;
   MPI_Irecv(&buf[0], 1, MPI_INT, prev, 5, MPI_COMM_WORLD,
&reqs[0]);
   MPI_Irecv(&buf[1], 1, MPI_INT, next, 6, MPI_COMM_WORLD,
&reqs[1]);
   MPI_Isend(&rank, 1, MPI_INT, prev, 6, MPI_COMM_WORLD,
&reqs[2]);
   MPI_Isend(&rank, 1, MPI_INT, next, 5, MPI_COMM_WORLD,
&reqs[3]);
   MPI_Waitall(4, reqs, stats);
   printf("process %d prev = %d next=%d\n", rank, buf[0],
buf[1]);
   MPI_Finalize();
}
```

MPI

```
int MPI_Waitany(int count,
MPI_Request *requests, int
*index, MPI_Status *status)
```

Ожидание завершения одной из **count** асинхронных операций, ассоциированных с идентификаторами **requests.** Для неблокирующего приёма определяется параметр **status.**

MPI

Если к моменту вызова завершились несколько из ожидаемых операций, то случайным образом будет выбрана одна из них.

Параметр **`index`** содержит номер элемента в массиве **`requests`**, содержащего идентификатор завершённой операции.

MPI

```
int MPI_Waitsome(int incount,
MPI_Request *requests, int
*outcount, int *indexes,
MPI_Status *statuses)
```

Ожидание завершения хотя бы одной из **`incount`** асинхронных операций, ассоциированных с идентификаторами **`requests`**.

MPI

Параметр **`outcount`** содержит число завершённых операций, а первые **`outcount`** элементов массива **`indexes`** содержат номера элементов массива **`requests`** с их идентификаторами.

Первые **`outcount`** элементов массива **`statuses`** содержат параметры завершённых операций (для неблокирующих приёмов).

MPI

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

Проверка завершённости асинхронной операции, ассоциированной с идентификатором **`request`**. В параметре **`flag`** возвращается значение **`1`**, если операция завершена.

MPI

```
int MPI_Testall(int count,
MPI_Request *requests, int
*flag, MPI_Status *statuses)
```

Проверка завершённости **count** асинхронных операций, ассоциированных с идентификаторами **requests.**

MPI

```
int MPI_Testany(int count,
MPI_Request *requests, int
*index, int *flag, MPI_Status
*status)
```

В параметре **flag** возвращается значение **1**, если хотя бы одна из операций асинхронного обмена завершена.

MPI

```
int MPI_Testsome(int incount,
MPI_Request *requests, int
*outcount, int *indexes,
MPI_Status *statuses)
```

Аналог **MPI_Waitsome,** но возврат происходит немедленно. Если ни одна из операций не завершилась, то значение **outcount** будет равно нулю.

MPI

Отложенные запросы на взаимодействие

MPI

```
int MPI_Send_init(void *buf, int count, MPI_Datatype datatype, int dest, int msgtag, MPI_Comm comm, MPI_Request *request)
```

Формирование отложенного запроса на посылку сообщения. Сама операция пересылки не начинается!

MPI

Модификации функции **`MPI_Send_init`**:

- **`MPI_Bsend_init`** — формирование запроса на передачу сообщения с буферизацией;
- **`MPI_Ssend_init`** — формирование запроса на передачу сообщения с синхронизацией;
- **`MPI_Rsend_init`** — формирование запроса на передачу сообщения по готовности.

MPI

```
int MPI_Recv_init(void *buf, int count, MPI_Datatype datatype, int source, int msgtag, MPI_Comm comm, MPI_Request *request)
```

Формирование отложенного запроса на приём сообщения. Сама операция приёма не начинается!

MPI

```
int MPI_Start(MPI_Request *request)
```

Инициализация отложенного запроса на выполнение операции обмена, соответствующей значению параметра **`request.`** Операция запускается как неблокирующая.

MPI

```
int MPI_Startall(int count,
MPI_Request *requests)
```

Инициализация **count** отложенных запросов на выполнение операций обмена, соответствующих значениям первых **count** элементов массива **requests.** Операции запускаются как неблокирующие.

MPI

Сообщение, отправленное при помощи отложенного запроса, может быть принято любой из процедур **`MPI_Recv`** и **`MPI_Irecv`**, и наоборот.

По завершении отложенного запроса значение параметра **`request (requests)`** сохраняется и может использоваться в дальнейшем!

MPI

```
int MPI_Request_free(MPI_Request *request)
```

Удаляет структуры данных, связанные с параметром **request. request** устанавливается в значение **MPI_REQUEST_NULL**. Если операция, связанная с этим запросом, уже выполняется, то она завершится.

MPI

Схема итерационного метода с обменом по кольцевой топологии при помощи отложенных запросов:

```
prev = rank - 1;
next = rank + 1;
if (rank == 0) prev = size - 1;
if (rank == (size - 1)) next = 0;
MPI_Recv_init(&rbuf[0], 1, MPI_FLOAT, prev, tag1,
MPI_COMM_WORLD, &reqs[0]);
MPI_Recv_init(&rbuf[1], 1, MPI_FLOAT, next, tag2,
MPI_COMM_WORLD, &reqs[1]);
MPI_Send_init(&sbuf[0], 1, MPI_FLOAT, prev, tag2,
MPI_COMM_WORLD, &reqs[2]);
MPI_Send_init(&sbuf[1], 1, MPI_FLOAT, next, tag1,
MPI_COMM_WORLD, &reqs[3]);
```

MPI

```
for(…){
 sbuf[0]=…;
 sbuf[1]=…;
 MPI_Startall(4, reqs);
 ...
 MPI_Waitall(4, reqs, stats);
 ...
}
MPI_Request_free(&reqs[0]);
MPI_Request_free(&reqs[1]);
MPI_Request_free(&reqs[2]);
MPI_Request_free(&reqs[3]);
```

MPI

```
int MPI_Sendrecv(void *sbuf, int
scount, MPI_Datatype stype, int
dest, int stag, void *rbuf, int
rcount, MPI_Datatype rtype, int
source, int rtag, MPI_Comm comm,
MPI_Status *status)
```

MPI

Совмещённые приём и передача сообщений с блокировкой. Буферы передачи и приёма не должны пересекаться. Тупиковой ситуации не возникает!

Сообщение, отправленное операцией **`MPI_Sendrecv`**, может быть принято обычным образом, и операция **`MPI_Sendrecv`** может принять сообщение, отправленное обычной операцией.

MPI

```
int MPI_Sendrecv_replace(void
*buf, int count, MPI_Datatype
datatype, int dest, int stag,
int source, int rtag, MPI_comm
comm, MPI_Status *status)
```

Совмещённые приём и передача сообщений с блокировкой через общий буфер **`buf`**.

MPI

Обмен по кольцевой топологии при помощи процедуры **MPI_Sendrecv**:

```
prev = rank - 1;
next = rank + 1;
if (rank == 0) prev = size - 1;
if (rank == (size - 1)) next = 0;
MPI_Sendrecv(&sbuf[0], 1, MPI_FLOAT, prev,
tag2, &rbuf[0], 1, MPI_FLOAT, next, tag2,
MPI_COMM_WORLD, &status1);
MPI_Sendrecv(&sbuf[1], 1, MPI_FLOAT, next,
tag1, &rbuf[1], 1, MPI_FLOAT, prev, tag1,
MPI_COMM_WORLD, &status2);
```

MPI

Специальное значение **MPI_PROC_NULL** для несуществующего процесса. Операции с таким процессом завершаются немедленно с кодом завершения **MPI_SUCCESS**.

```
next = rank + 1;
if(rank == (size - 1)) next=MPI_PROC_NULL;
MPI_Send (&buf, 1, MPI_FLOAT, next, tag,
MPI_COMM_WORLD);
```

Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ
(лекция 10)

А.С.Антонов

Вед. н.с. НИВЦ МГУ, к.ф.-м.н.

asa@parallel.ru

ВМК МГУ, 2020

Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ
(лекция 11)

А.С.Антонов

Вед. н.с. НИВЦ МГУ, к.ф.-м.н.

asa@parallel.ru

ВМК МГУ, 2020

MPI

Коллективные взаимодействия процессов

MPI

В операциях коллективного взаимодействия процессов *участвуют все процессы коммуникатора*!

Как и для блокирующих процедур, возврат означает то, что разрешён свободный доступ к буферу приёма или посылки.

Сообщения, вызванные коллективными операциями, не пересекутся с другими сообщениями.

MPI

Нельзя рассчитывать на синхронизацию процессов с помощью коллективных операций (кроме процедуры **`MPI_Barrier`**).

Если какой-то процесс завершил свое участие в коллективной операции, то это не означает ни того, что данная операция завершена другими процессами коммуникатора, ни даже того, что она ими начата (если это возможно по смыслу операции).

MPI

В коллективных операциях не используются идентификаторы сообщений (теги). Таким образом, коллективные операции строго упорядочены согласно их появлению в тексте программы. Несоблюдение такого порядка может привести к возникновению тупиковых ситуаций.

MPI

`int MPI_Barrier(MPI_Comm comm)`

Работа процессов блокируется до тех пор, пока все оставшиеся процессы коммуникатора **`comm`** не выполнят эту процедуру. Все процессы должны вызвать **`MPI_Barrier`**, хотя реально исполненные различными процессами коммуникатора вызовы могут быть расположены в разных местах программы.

MPI

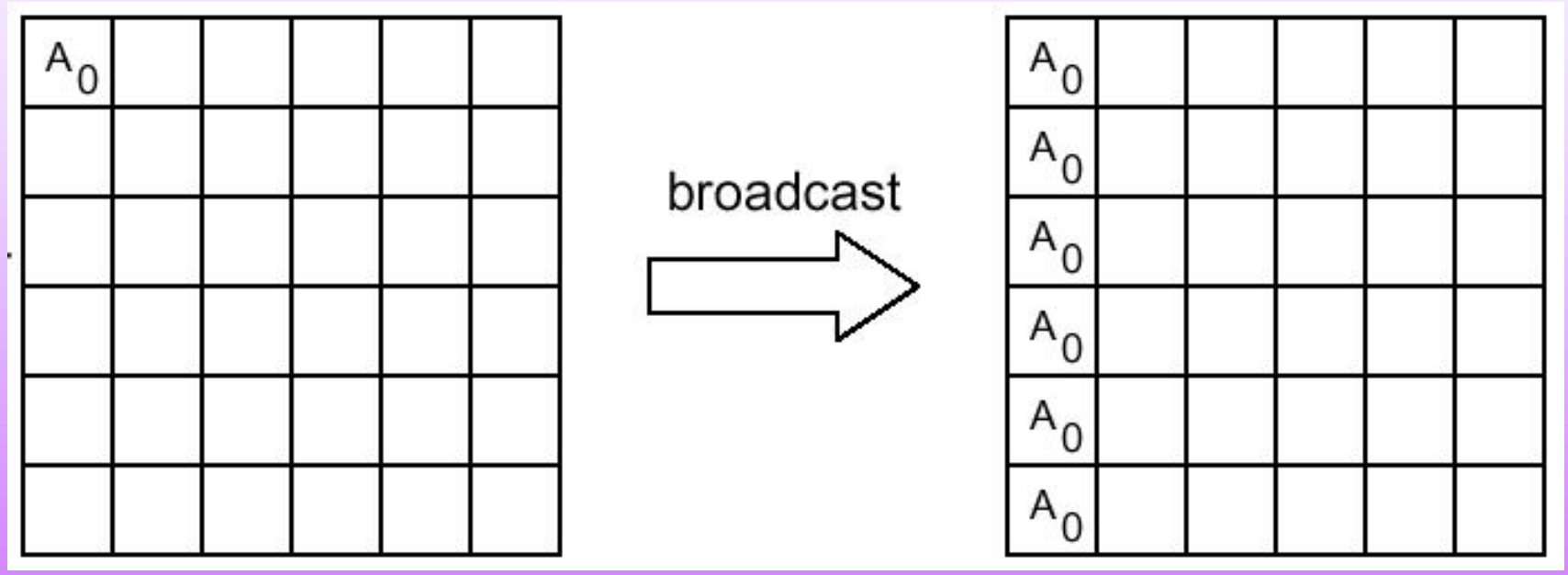
```
int MPI_Bcast(void *buf, int
count, MPI_Datatype datatype,
int root, MPI_Comm comm)
```

Рассылка **count** элементов данных типа **datatype** из массива **buf** от процесса **root** всем процессам данного коммуникатора **comm**, включая сам рассылающий процесс. Значения параметров **count, datatype, root** и **comm** должны быть одинаковыми у всех процессов.

MPI

данные

процессы



MPI

Например, для пересылки от процесса **2** всем остальным процессам приложения массива **buf** из **100** целочисленных элементов, нужно, чтобы во всех процессах встретился следующий вызов:

```
MPI_Bcast(buf, 100, MPI_INT, 2,
MPI_COMM_WORLD);
```

MPI

```
int MPI_Gather(void *sbuf, int scount, MPI_Datatype stype, void *rbuf, int rcount, MPI_Datatype rtype, int root, MPI_Comm comm)
```

Сборка **scount** элементов данных типа **stype** из массивов **sbuf** со всех процессов коммуникатора **comm** в буфер **rbuf** процесса **root**. Данные сохраняются в порядке возрастания номеров процессов.

MPI

На процессе **root** существенными являются значения всех параметров, а на остальных процессах - только значения параметров **sbuf**, **scount**, **stype**, **root** и **comm**. Значения параметров **root** и **comm** должны быть одинаковыми у всех процессов. Параметр **rcount** у процесса **root** обозначает число элементов типа **rtype**, принимаемых от каждого процесса.

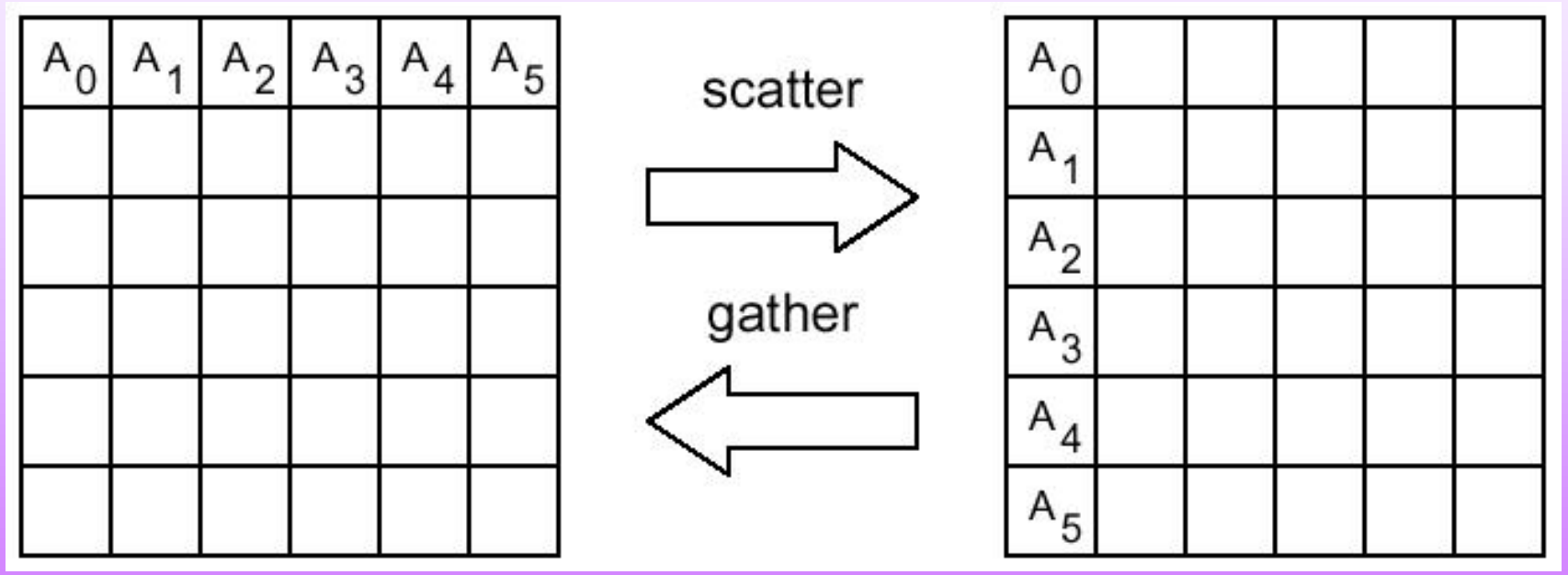
MPI

Если для посылки и приема данных должен использоваться один и тот же буфер, то на месте аргумента **`sbuf`** процесса **`root`** можно указать значение **`MPI_IN_PLACE`**. В этом случае аргументы **`scount`** и **`stype`** игнорируются, и предполагается, что порция данных процесса **`root`** уже расположена в соответствующем месте буфера приема **`rbuf`**.

MPI

данные

процессы



MPI

Например, чтобы процесс **2** собрал в массив **rbuf** по **10** целочисленных элементов массивов **buf** со всех процессов приложения, нужно, чтобы во всех процессах встретился следующий вызов:

```
MPI_Gather(buf, 10, MPI_INT,
rbuf, 10, MPI_INT, 2,
MPI_COMM_WORLD);
```

MPI

```
int MPI_Gatherv(void *sbuf, int scount, MPI_Datatype stype, void *rbuf, int *rcounts, int *displs, MPI_Datatype rtype, int root, MPI_Comm comm)
```

Сборка различного количества данных из массивов **sbuf**. Порядок расположения данных в результирующем буфере **rbuf** задаёт массив **displs**.

MPI

`rcounts` – целочисленный массив, содержащий количество элементов, передаваемых от каждого процесса (индекс равен рангу адресата, длина равна числу процессов в коммуникаторе).

`displs` – целочисленный массив, содержащий смещения относительно начала массива **`rbuf`** (индекс равен рангу адресата, длина равна числу процессов в коммуникаторе).

MPI

```
int MPI_Scatter(void *sbuf, int scount, MPI_Datatype stype, void *rbuf, int rcount, MPI_Datatype rtype, int root, MPI_Comm comm)
```

Рассылка по **scount** элементов данных типа **stype** из массива **sbuf** процесса **root** в массивы **rbuf** всех процессов коммуникатора **comm**, включая сам процесс **root**. Данные рассылаются в порядке возрастания номеров процессов.

MPI

На процессе **`root`** существенными являются значения всех параметров, а на всех остальных процессах — только значения параметров **`rbuf`**, **`rcount`**, **`rtype`**, **`source`** и **`comm`**. Значения параметров **`source`** и **`comm`** должны быть одинаковыми у всех процессов.

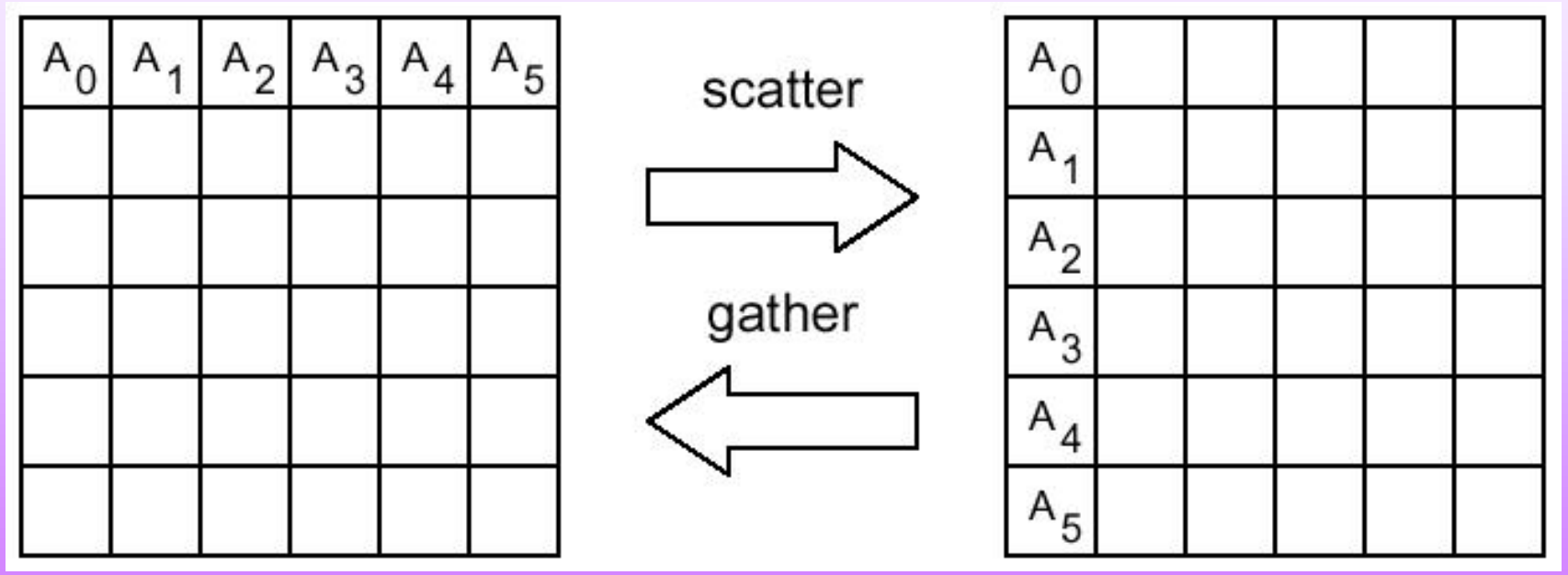
MPI

Если для посылки и приема данных должен использоваться один буфер, то на месте аргумента **`rbuf`** процесса **`root`** можно указать значение **`MPI_IN_PLACE`**. В этом случае аргументы **`rcount`** и **`rtype`** игнорируются, и предполагается, что порция данных процесса **`root`** не пересылается.

MPI

данные

процессы



MPI

```
float sbuf[SIZE][SIZE], rbuf[SIZE];
if(rank == 0)
  for(i=0; i<SIZE; i++)
    for (j=0; j<SIZE; j++)
      sbuf[i][j]=…;
if (numtasks == SIZE)
  MPI_Scatter(sbuf, SIZE, MPI_FLOAT, rbuf,
SIZE, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

MPI

```
int MPI_Scatterv(void *sbuf, int
*scounts, int *displs,
MPI_Datatype stype, void *rbuf,
int rcount, MPI_Datatype rtype,
int root, MPI_Comm comm)
```

Рассылка различного количества данных из массива **sbuf**. Начало рассылаемых порций задает массив **displs**.

MPI

`scounts` – целочисленный массив, содержащий количество элементов, передаваемых каждому процессу (индекс равен рангу адресата, длина равна числу процессов в коммуникаторе).

`displs` – целочисленный массив, содержащий смещения относительно начала массива **`sbuf`** (индекс равен рангу адресата, длина равна числу процессов в коммуникаторе).

MPI

```
int MPI_Allgather(void *sbuf,
int scount, MPI_Datatype stype,
void *rbuf, int rcount,
MPI_Datatype rtype, MPI_Comm
comm)
```

Сборка данных из массивов **sbuf** со всех процессов коммуникатора **comm** в буфере **rbuf** каждого процесса. Данные сохраняются в порядке возрастания номеров процессов.

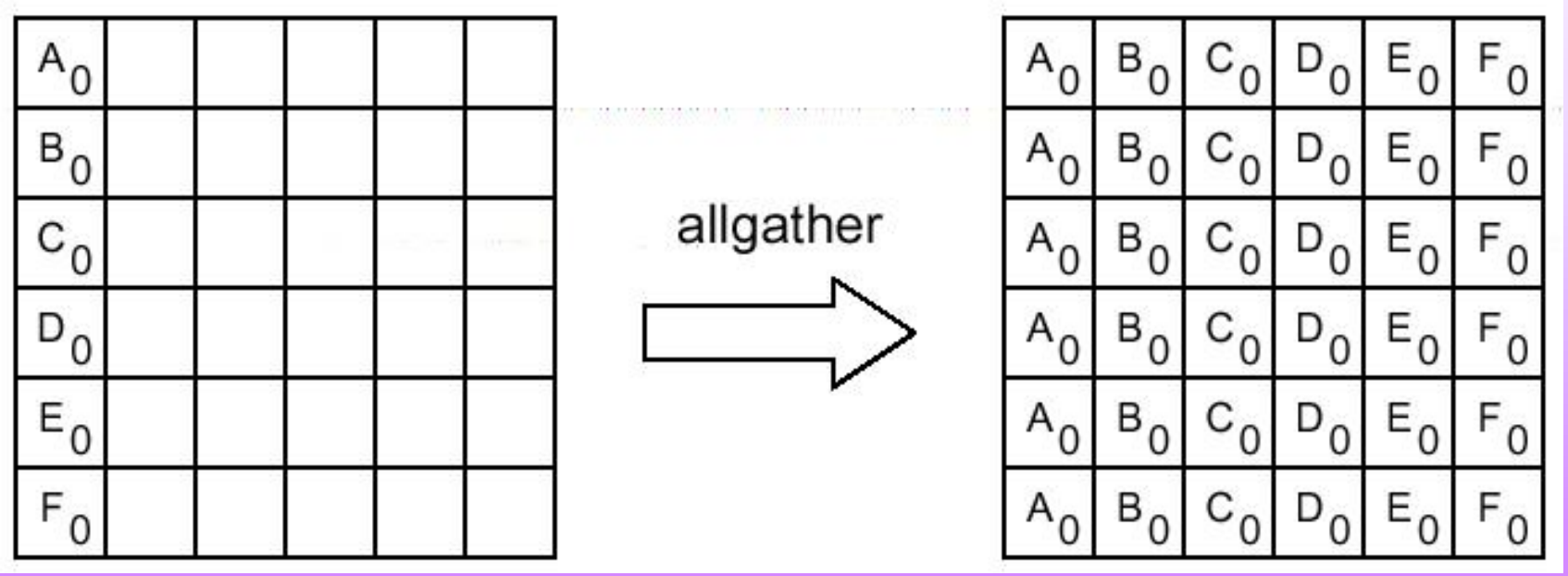
MPI

Если для посылки и приема данных должен использоваться один буфер, то на месте аргумента **`sbuf`** всех процессов можно указать значение **`MPI_IN_PLACE`**. В этом случае аргументы **`scount`** и **`stype`** игнорируются, и предполагается, что порции исходных данных всех процессов уже расположены в соответствующих местах буферов приема **`rbuf`**.

MPI

данные

процессы



MPI

```
int MPI_Allgatherv(void *sbuf,
int scount, MPI_Datatype stype,
void *rbuf, int *rcounts, int
*displs, MPI_Datatype rtype,
MPI_Comm comm)
```

Сборка на всех процессах различного количества данных из **sbuf**. Порядок расположения данных в массиве **rbuf** задаёт массив **displs**.

MPI

```
int MPI_Alltoall(void *sbuf, int scount, MPI_Datatype stype, void *rbuf, int rcount, MPI_Datatype rtype, MPI_Comm comm)
```

Рассылка каждым процессом коммуникатора **comm** различных порций данных всем другим процессам. **j**-й блок массива **sbuf** процесса **i** попадает в **i**-й блок массива **rbuf** процесса **j**.

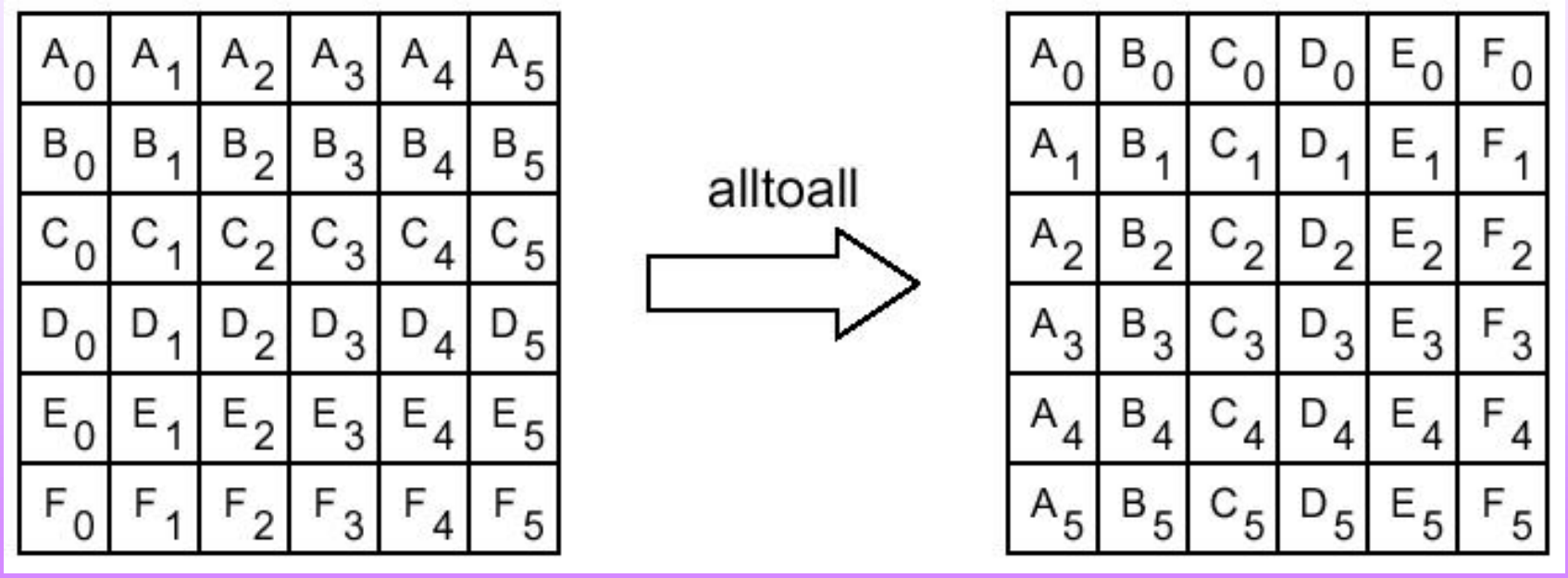
MPI

Если для посылки и приема должен использоваться один буфер, то на месте аргумента **`sbuf`** всех процессов можно указать значение **`MPI_IN_PLACE`**. В этом случае аргументы **`scount`** и **`stype`** игнорируются, и предполагается, что порции исходных данных всех процессов уже расположены в соответствующих местах буферов приема **`rbuf`**.

MPI

данные

процессы



MPI

```
int MPI_Alltoallv(void* sbuf,
int *scounts, int *sdispls,
MPI_Datatype stype, void* rbuf,
int *rcounts, int *rdispls,
MPI_Datatype rtype, MPI_Comm
comm)
```

Рассылка со всех процессов коммуникатора **comm** различного количества данных всем другим процессам. Размещение данных в буфере **sbuf** отсылающего процесса определяется массивом **sdispls**, а в буфере **rbuf** принимающего процесса – массивом **rdispls**.

MPI

```
int MPI_Alltoallw(void *sbuf,
int scounts[], int sdispls[],
MPI_Datatype stypes[], void
*rbuf, int rcounts[], int
rdispls[], MPI_Datatype
rtypes[], MPI_Comm comm)
```

Процедура позволяет осуществить наиболее общий обмен данными с заданием для каждой порции своего размера, типа данных и размещения в буфере. Смещения в массивах **sdispls** и **rdispls** задаются в байтах.

MPI

```
int MPI_Reduce(void *sbuf, void *rbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

Выполнение **count** независимых глобальных операций **op** над соответствующими элементами массивов **sbuf**. Результат операции над **i**-ми элементами массивов **sbuf** получается в **i**-ом элементе массива **rbuf** процесса **root**.

MPI

Типы предопределённых глобальных операций:

MPI_MAX, MPI_MIN – максимальное и минимальное значения;

MPI_SUM, MPI_PROD – глобальная сумма и глобальное произведение;

MPI_LAND, MPI_LOR, MPI_LXOR – логические “И”, “ИЛИ”, искл. “ИЛИ”;

MPI_BAND, MPI_BOR, MPI_BXOR – побитовые “И”, “ИЛИ”, искл. “ИЛИ”.

MPI

Если для посылки и приема данных должен использоваться один буфер, то на месте аргумента **`sbuf`** процесса **`root`** можно указать значение **`MPI_IN_PLACE`**. В этом случае входные данные процесса **`root`** берутся из буфера результата **`rbuf`**.

MPI

```
int MPI_Allreduce(void *sbuf,
void *rbuf, int count,
MPI_Datatype datatype, MPI_Op
op, MPI_Comm comm)
```

Выполнение **count** независимых глобальных операций **op** над соответствующими элементами массивов **sbuf**. Результат получается в массиве **rbuf** каждого процесса.

MPI

```
for(i=0; i<n; i++) s[i]=0.0;
for(i=0; i<n; i++)
  for(j=0; j<m; j++)
    s[i]=s[i]+a[i][j];
MPI_Allreduce(s, r, n, MPI_FLOAT, MPI_SUM,
MPI_COMM_WORLD);
```

MPI

```
int MPI_Reduce_local(void*
inbuf, void* inoutbuf, int
count, MPI_Datatype datatype,
MPI_Op op)
```

Процедура на вызвавшем процессе поэлементно выполняет **op** операций над **count** элементами массивов **inbuf** и **inoutbuf**. Результат помещается в массив **inoutbuf**.

MPI

```
int MPI_Reduce_scatter_block(
void* sbuf, void* rbuf, int
rcount, MPI_Datatype datatype,
MPI_Op op, MPI_Comm comm)
```

Выполнение **`n*rcount`** (**`n`** – число процессов) независимых глобальных операций **`op`** над соответствующими элементами массивов **`sbuf`** всех процессов.

MPI

Сначала выполняются глобальные операции, а затем результат рассылается по процессам.

При этом **i**-ый процесс получает **(i+1)**-ую порцию результатов из **rcount** элементов и помещает ее в массив **rbuf**. Значение **rcount** должно быть одинаковым на всех процессах коммуникатора **comm**.

MPI

```
int MPI_Reduce_scatter(void *sbuf, void *rbuf, int *rcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Выполнение $\sum_i$ **`rcounts(i)`** независимых глобальных операций **`op`** над соответствующими элементами массивов **`sbuf`**.

MPI

Сначала выполняются глобальные операции, затем результат рассылается по процессам.

`i`-ый процесс получает **`rcounts(i)`** значений результата и помещает в массив **`rbuf`**.

MPI

```
int MPI_Scan(void *sbuf, void *rbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Выполнение **count** независимых частичных глобальных операций **op** над соответствующими элементами массивов **sbuf**.

MPI

`i`-ый процесс выполняет глобальную операцию над соответствующими элементами массива **`sbuf`** процессов `0`...**`i`** и помещает результат в массив **`rbuf`**.

Окончательный результат глобальной операции получается в массиве **`rbuf`** последнего процесса.

MPI

```
int MPI_Exscan(void *sbuf, void *rbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Выполнение **count** независимых частичных глобальных операций **op** над соответствующими элементами массивов **sbuf**.

MPI

i-ый процесс (**i>0**) выполняет **count** глобальных операций над соответствующими элементами массива **sbuf** процессов с номерами от **0** до **i-1** включительно и помещает полученный результат в массив **rbuf**. На процессе **0** массив **rbuf** не задействуется.

MPI

```
int MPI_Op_create
(MPI_User_function *func, int
commute, MPI_Op *op)
```

Создание пользовательской глобальной операции **op**, которая будет вычисляться функцией **func**. Если **commute=1**, то операция должна быть коммутативной и ассоциативной. Иначе порядок фиксируется по увеличению номеров процессов.

MPI

```
typedef void MPI_User_function
(void *invec, void *inoutvec,
int *len, MPI_Datatype type)
```

Интерфейс пользовательской функции. Первый аргумент **invec**, второй аргумент – **inoutvec**, результат –**inoutvec**. **len** задает количество элементов входного и выходного массивов, а **type** – тип данных. В пользовательской функции не должны производиться никакие обмены.

MPI

```
int MPI_Op_free(MPI_Op *op)
```

Уничтожение пользовательской глобальной операции. После выполнения процедуры переменной **op** присваивается значение **MPI_OP_NULL**.

MPI

```
#include <stdio.h>
#include "mpi.h"
#define n 1000
void smod5(void *in, void *inout, int *l, MPI_Datatype
*type){
   int i;
   for(i=0; i<*l; i++) ((int*)inout)[i] = (((int*)in)[i]
+ ((int*)inout)[i])%5;
}
int main(int argc, char **argv)
{
   int rank, size, i;
   int a[n];
   int b[n];
```

MPI

```
    MPI_Op op;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    for(i=0; i<n; i++) a[i] = i + rank +1;
    printf("process %d a[0] = %d\n", rank, a[0]);
    MPI_Op_create(&smod5, 1, &op);
    MPI_Reduce(a, b, n, MPI_INT, op, 0, MPI_COMM_WORLD);
    MPI_Op_free(&op);
    if(rank==0) printf("b[0] = %d\n", b[0]);
    MPI_Finalize();
}
```

MPI

Группы и коммуникаторы

MPI

```
int MPI_Comm_split(MPI_Comm
comm, int color, int key,
MPI_Comm *newcomm)
```

Разбиение коммуникатора **comm** на несколько по числу значений параметра **color**. В одну подгруппу попадают процессы с одним значением **color**. Процессы с бóльшим значением **key** получат больший ранг.

MPI

Процессы, которые не должны войти в новые группы, указывают в качестве **color** константу **MPI_UNDEFINED**. Им в параметре **newcomm** вернется значение **MPI_COMM_NULL**.

```
MPI_Comm_split(MPI_COMM_WORLD, rank%3, rank,
new_comm)
```

MPI

Пересылка разнотипных данных

MPI

Сообщение – массив однотипных данных, расположенных в последовательных ячейках памяти.

Для пересылки разнотипных данных можно использовать:

- Производные типы данных
- Упаковку данных

MPI

Производные типы данных

MPI

Производные типы данных создаются во время выполнения программы с помощью подпрограмм-конструкторов.

Создание типа:

- Конструирование типа
- Регистрация типа

MPI

Производный тип данных характеризуется последовательностью базовых типов и набором значений смещения относительно начала буфера обмена.

Смещения могут быть как положительными, так и отрицательными, не требуется их упорядоченность.

MPI

```
int MPI_Type_contiguous(int
count, MPI_Datatype type,
MPI_Datatype *newtype)
```

Создание нового типа данных **newtype**, состоящего из **count** последовательно расположенных элементов базового типа данных **type**.

```
MPI_Type_contiguous(5, MPI_INT, &newtype);
```

MPI

```
int MPI_Type_vector(int count,
int blocklen, int stride,
MPI_Datatype type, MPI_Datatype
*newtype)
```

Создание нового типа данных **newtype**, состоящего из **count** блоков по **blocklen** элементов базового типа данных **type**. Следующий блок начинается через **stride** элементов после начала предыдущего.

MPI

```
count=2;
blocklen=3;
stride=5;
MPI_Type_vector(count, blocklen, stride,
MPI_DOUBLE, &newtype);
```

Создание нового типа данных (тип элемента, количество элементов от начала буфера):

{(**MPI_DOUBLE**, 0), (**MPI_DOUBLE**, 1), (**MPI_DOUBLE**, 2),

(**MPI_DOUBLE**, 5), (**MPI_DOUBLE**, 6), (**MPI_DOUBLE**, 7)}

MPI

```
int MPI_Type_create_hvector(int
count, int blocklen, MPI_Aint
stride, MPI_Datatype type,
MPI_Datatype *newtype)
```

Создание нового типа данных **newtype**, состоящего из **count** блоков по **blocklen** элементов базового типа данных **type**. Следующий блок начинается через **stride** байт после начала предыдущего.

MPI

```
int
MPI_Type_create_indexed_block(
int count, int blocklen, int
displs[], MPI_Datatype type,
MPI_Datatype *newtype)
```

Создание нового типа данных **newtype**, состоящего из **count** блоков по **blocklen** элементов базового типа данных **type**. Смещения блоков с начала буфера посылки в количестве элементов базового типа данных **type** задаются в массиве **displs**.

MPI

```
int MPI_Type_indexed(int count,
int *blocklens, int *displs,
MPI_Datatype type, MPI_Datatype
*newtype)
```

Создание нового типа данных **newtype**, состоящего из **count** блоков по **blocklens[i]** элементов базового типа данных. **i**-ый блок начинается через **displs[i]** элементов с начала буфера.

MPI

```
for(i=0; i<n; i++){
   blocklens[i]=n-i;
   displs[i]=(n+1)*i;
}
MPI_Type_indexed(n, blocklens, displs,
MPI_DOUBLE, &newtype)
```

Создание нового типа данных для описания верхнетреугольной матрицы.

MPI

```
int MPI_Type_create_hindexed(int
count, int *blocklens, MPI_Aint
*displs, MPI_Datatype type,
MPI_Datatype *newtype)
```

Создание нового типа данных **newtype**, состоящего из **count** блоков по **blocklens[i]** элементов базового типа данных. **i**-ый блок начинается через **displs[i]** байт с начала буфера.

MPI

```
int MPI_Type_create_struct(int
count, int *blocklens, MPI_Aint
*displs, MPI_Datatype *types,
MPI_Datatype *newtype)
```

Создание структурного типа данных из **`count`** блоков по **`blocklens[i]`** элементов типа **`types[i]`**. **`i`**-ый блок начинается через **`displs[i]`** байт с начала буфера.

MPI

```
blocklens[0]=3;
blocklens[1]=2;
types[0]=MPI_DOUBLE;
types[1]=MPI_CHAR;
displs[0]=0;
displs[1]=24;
MPI_Type_create_struct(2, blocklens, displs,
types, &newtype);
```

Создание нового типа данных (тип элемента, количество байт от начала буфера):

{(**MPI_DOUBLE**, 0), (**MPI_DOUBLE**, 8), (**MPI_DOUBLE**, 16),
(**MPI_CHAR**, 24), (**MPI_CHAR**, 25)}

MPI

```
int MPI_Type_create_subarray(int ndims, int sizes[], int subsizes[], int starts[], int order, MPI_Datatype type, MPI_Datatype *newtype)
```

newtype задаёт **ndims**–мерный подмассив исходного **ndims**–мерного массива. **sizes** задает размеры по каждому измерению исходного массива, **subsizes** – размеры по каждому измерению выделяемого подмассива.

MPI

starts задаёт стартовые координаты каждого измерения выделяемого подмассива в исходном массиве. Все массивы индексируются с **0**. Задаваемые значения не должны выводить подмассив за пределы исходного массива ни по одному из измерений.**order** задаёт порядок хранения элементов многомерного массива:
MPI_ORDER_C (по строкам),
MPI_ORDER_FORTRAN (по столбцам).

type задаёт тип элементов массива.

MPI

```
MPI_Datatype newtype;
int sizes[2], subsizes[2], starts[2];
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
sizes[0] = 100;
sizes[1] = 100;
subsizes[0] = 100;
subsizes[1] = 25;
starts[0] = 0;
starts[1] = rank*subsizes[1];
MPI_Type_create_subarray(2, sizes, subsizes,
starts, MPI_ORDER_C, MPI_DOUBLE, &newtype);
```

MPI

```
int MPI_Type_commit(MPI_Datatype *datatype)
```

Регистрация созданного производного типа данных **`datatype`**. После регистрации этот тип данных можно использовать в операциях обмена. Предопределенные типы данных регистрировать не нужно.

MPI

```
int MPI_Type_free(MPI_Datatype
*datatype)
```

Аннулирование производного типа данных **`datatype`**.

`datatype` устанавливается в значение **`MPI_DATATYPE_NULL`**.

Производные от **`datatype`** типы данных остаются. Предопределённые типы данных не могут быть аннулированы.

MPI

```
int MPI_Type_dup(MPI_Datatype
type, MPI_Datatype *newtype)
```

Создает новый тип данных **newtype**, аналогичный типу данных **type**. При этом производится копирование всех ассоциированных с типом **type** атрибутов. Если исходный тип данных уже был зарегистрирован, то автоматически будет зарегистрирован и создаваемый тип данных.

MPI

```
int MPI_Get_address(void
*location, MPI_Aint *address)
```

Определение абсолютного байт-адреса **`address`** размещения массива **`location`** в оперативной памяти компьютера. Адрес отсчитывается от некоторого базового адреса, значение которого содержится в системной константе **`MPI_BOTTOM`**.

MPI

```
blocklens[0] = 1;
blocklens[1] = 1;
types[0] = MPI_DOUBLE;
types[1] = MPI_CHAR;
MPI_Get_address(dat1, &displs[0]);
MPI_Get_address(dat2, &displs[1]);
MPI_Type_create_struct(2, blocklens, displs,
types, &newtype);
MPI_Type_commit(&newtype);
MPI_Send(MPI_BOTTOM, 1, newtype, dest, tag,
MPI_COMM_WORLD);
```

MPI

```
int MPI_Type_size(MPI_Datatype datatype, int *size)
```

Определение размера типа **`datatype`** в байтах (объёма памяти, занимаемого одним элементом данного типа).

MPI

```
int MPI_Type_get_extent(
MPI_Datatype datatype, MPI_Aint
*lb, MPI_Aint *extent)
```

Для элемента типа данных **`datatype`** определяет смещение от начала буфера данных нижней границы **`lb`** и диапазон **`extent`** (разницу между верхней и нижней границами) в байтах.

MPI

Упаковка данных

MPI

```
int MPI_Pack(void *inbuf, int
incount, MPI_Datatype datatype,
void *outbuf, int outsize, int
*position, MPI_Comm comm)
```

Упаковка **incount** элементов типа **datatype** из массива **inbuf** в массив **outbuf** со сдвигом **position** байт. **outbuf** должен содержать хотя бы **outsize** байт.

MPI

Параметр **`position`** увеличивается на число байт, равное размеру записи. Параметр **`comm`** указывает на коммуникатор, в котором в дальнейшем будет пересылаться сообщение.

Для пересылки упакованных данных используется тип данных **`MPI_PACKED`**.

MPI

```
int MPI_Unpack(void *inbuf, int
insize, int *position, void
*outbuf, int outcount,
MPI_Datatype datatype, MPI_Comm
comm)
```

Распаковка из массива **inbuf** со сдвигом **position** байт в массив **outbuf** **outcount** элементов типа **datatype**. Массив **inbuf** имеет размер не менее **insize** байт.

MPI

```
int MPI_Pack_size(int incount,
MPI_Datatype datatype, MPI_Comm
comm, int *size)
```

Определение необходимого объёма памяти (в байтах) для упаковки **`incount`** элементов типа **`datatype`**. Необходимый для упаковки размер может превышать сумму размеров пакуемых элементов данных.

MPI

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
   int size, rank, position, i;
   float a[10];
   char b[10], buf[100];
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   for(i = 0; i<10; i++){
      a[i] = rank + 1.0;
      if(rank==0) b[i]='a';
      else b[i] = 'b';
   }
   position=0;
```

MPI

```
   if(rank==0){
      MPI_Pack(a, 10, MPI_FLOAT, buf, 100, &position,
MPI_COMM_WORLD);
      MPI_Pack(b, 10, MPI_CHAR, buf, 100, &position,
MPI_COMM_WORLD);
      MPI_Bcast(buf, 100, MPI_PACKED, 0, MPI_COMM_WORLD);
   } else{
      MPI_Bcast(buf, 100, MPI_PACKED, 0, MPI_COMM_WORLD);
      MPI_Unpack(buf, 100, &position, a, 10, MPI_FLOAT,
MPI_COMM_WORLD);
      MPI_Unpack(buf, 100, &position, b, 10, MPI_CHAR,
MPI_COMM_WORLD);
   }
   for(i = 0; i<10; i++) printf("process %d a=%f b=%c\n",
rank, a[i], b[i]);
   MPI_Finalize();}
```

MPI

Производные типы данных:

- сложнее использовать;

+ не нужны дополнительные буфера;

+ нет дополнительных копирований.

Упаковка данных:

+ проще использовать;

- требуется дополнительный буфер;

- требуются дополнительные копированя.

MPI

Литература

1. MPI: A Message-Passing Interface Standard Version 3.1. URL: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf
2. MPI – the complete reference, second edition / Ed. by M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra. – The MIT Press, 1998.
3. Антонов А.С. Технологии параллельного программирования MPI и OpenMP: Учеб. пособие. Предисл.: В.А.Садовничий. - М.: Издательство Московского университета, 2012.-344 с.- (Серия "Суперкомпьютерное образование").
4. Антонов А.С. Введение в параллельные вычисления (методическое пособие). – М.: Изд-во Физического факультета МГУ, 2002.
5. Антонов А.С. Параллельное программирование с использованием технологии MPI: Учебное пособие. – М.: Изд-во МГУ, 2004.

MPI

Литература

6. Букатов А.А., Дацюк В.Н., Жегуло А.И. Программирование многопроцессорных вычислительных систем. – Ростов-на-Дону: Издательство ООО «ЦВВР», 2003.
7. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. – СПб: БХВ-Петербург, 2002.
8. Корнеев В.Д. Параллельное программирование в MPI. – Новосибирск: Изд-во СО РАН, 2000.
9. Немнюгин С.А., Стесик О.Л. Параллельное программирование для многопроцессорных вычислительных систем. – СПб: БХВ-Петербург, 2002.
10. Шпаковский Г.И., Серикова Н.В. Программирование для многопроцессорных систем в стандарте MPI: Пособие. – Минск: БГУ, 2002.

Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ
(лекция 11)

А.С.Антонов

Вед. н.с. НИВЦ МГУ, к.ф.-м.н.

asa@parallel.ru

ВМК МГУ, 2020

Московский государственный университет имени М.В.Ломоносова

Факультет Вычислительной Математики и Кибернетики

СУПЕРКОМПЬЮТЕРЫ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ

Жуматий С.А.

в.н.с. НИВЦ МГУ, к.ф.-м.н.

ВМК МГУ, 2020
serg@parallel.ru

Компоненты и системное программное обеспечение суперкомпьютеров

Компоненты

- Инфраструктура
  - Помещение
  - Электричество
  - Охлаждение
- Вычислитель
  - Сети
  - Узлы

Инфраструктура: помещение

- Площадь
- Шумность
- Доступ для техобслуживания
- Противопожарное оборудование
- Контроль доступа

Инфраструктура: электричество

- Источники питания, проводка, автоматы
- ИБП, PDU, место
- Резервирование
- Расчётное время работы от батарей
- Проводка:фальшпол, короба, организаторы
- Заземление, статика

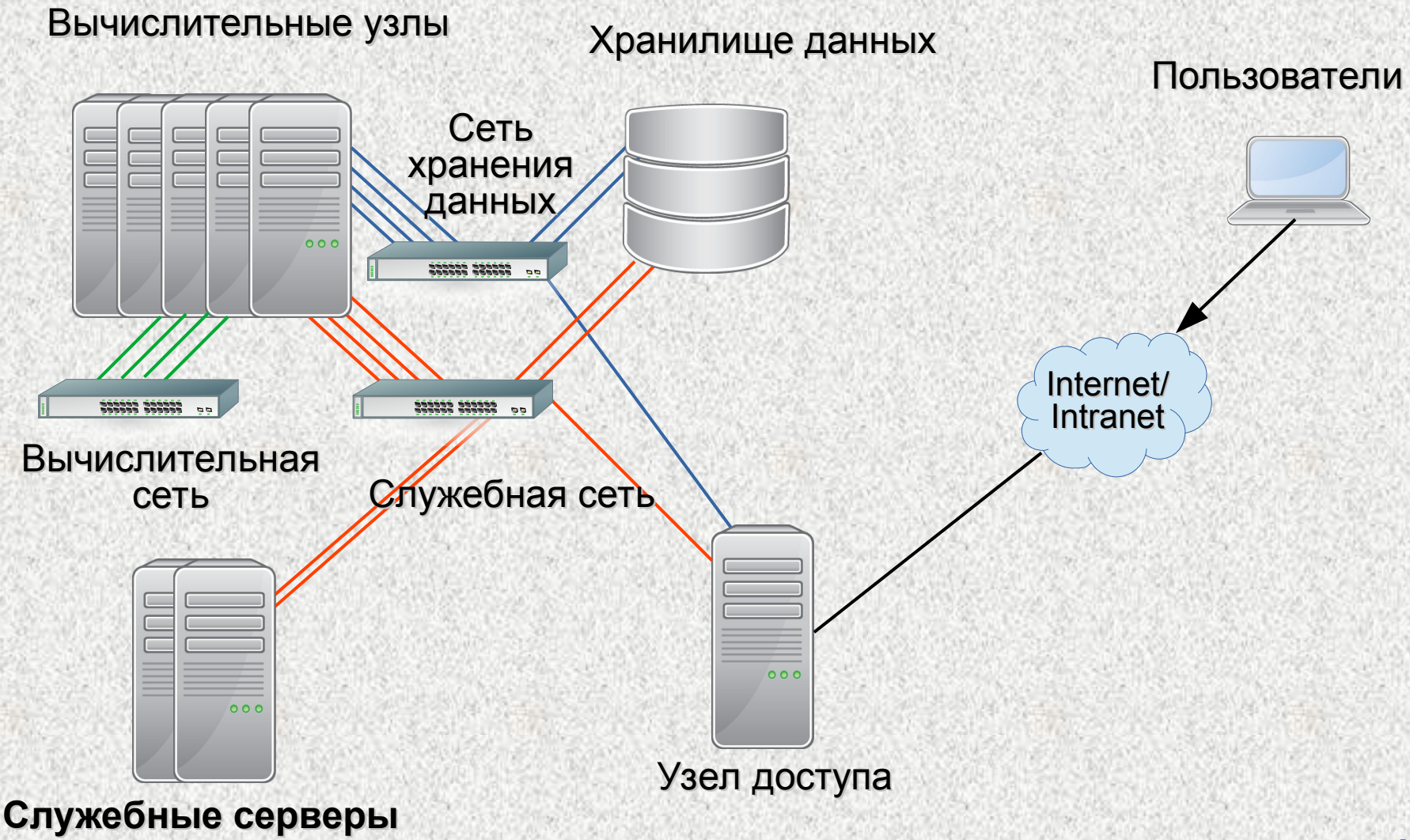
Инфраструктура: охлаждение

- Мощность: потребляемая и реальная
- Влажность
- Центральное отопление, солнце
- Зимнее время
- «Открытое окно»

Структура кластера

- Сети: вычислительная, управляющая, хранения данных, сервисная, другие
- Вычислительные узлы
- Узлы доступа
- Сетевая файловая система
- Служебные узлы
- Система архивирования

Как устроен суперкомпьютер



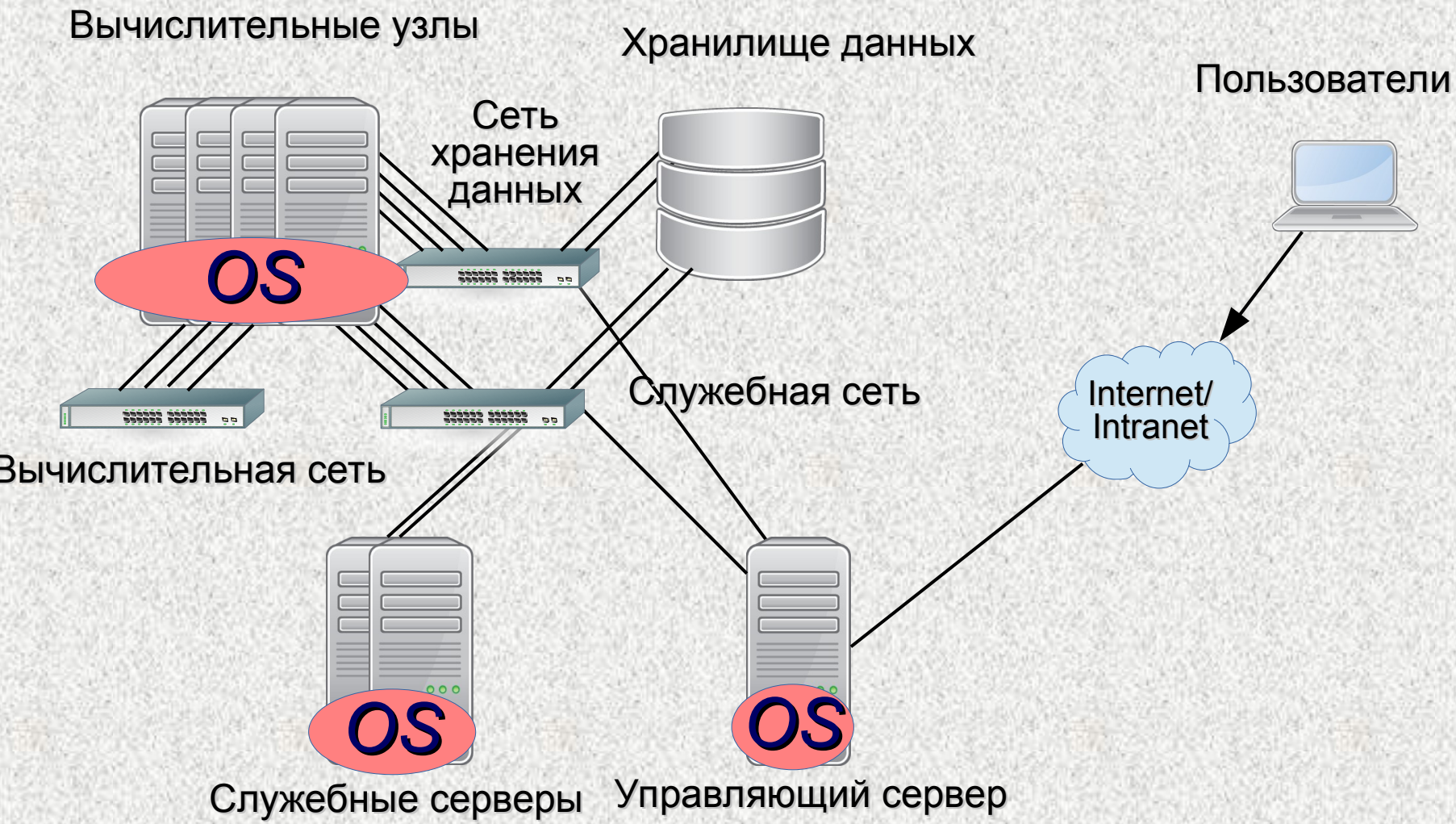
Служебные узлы (1)

- xCAT — управление образами т.п.
- DHCP — управление ip-адресами
- TFTP — загрузка, установка
- NFS — загрузка, root-fs
- DNS — управление dns-именами
- NTP — синхронизация времени
- LDAP — управление учётными записями

Служебные узлы (2)

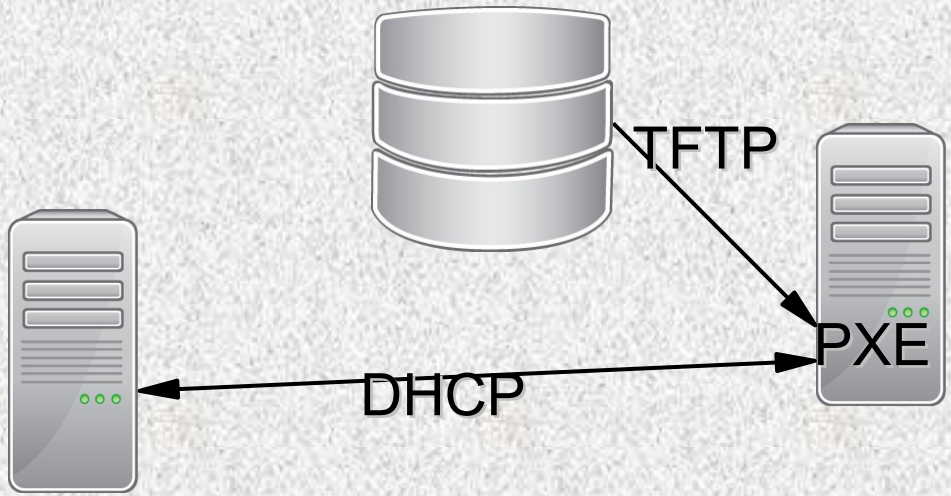
- Доступ (управляющий)
- Управление задачами
- Лицензии (flexlm)
- Мониторинг
- Статистика
- Визуализация
- Копирование данных

Загрузка ОС


Вычислительные узлы
Хранилище данных
Пользователи
Сеть хранения данных
OS
Служебная сеть
Internet/ Intranet
Вычислительная сеть
OS
OS
Служебные серверы
Управляющий сервер

Операционная система

Установлена
локально



Загружается по сети

Операционная система

Установлена локально:

Плюсы

- Быстрая загрузка
- Нет перегруза сети
- Не занимает лишнего места в памяти

Минусы

- Долгая установка и обновление
- Рассинхронизация настроек и версий

Операционная система

Загрузка по сети:

Плюсы

- Единый образ для всех узлов группы
- Возможность быстро обновить всё

Минусы

- Нагрузка на сеть
- Много места в памяти
- Нет возможности сохранить изменения

Установка ОС на диск

- «По одному», клонирование образа
- Автоматизированная установка
- Готовые стеки
- xCAT

Установка ОС на диск: автоматизированная

RH: anaconda-ks.cfg, **SuSE**: autoyast.xml, **Debian**: preseed.file

linux **ks=http://путь-к-файлу.cfg**

linux **autoyast=http://путь-к-файлу.xml**

linux **auto preseed=http://путь-к-файлу**

Установка ОС на диск: готовые стеки

- ROCKS
- PelicanHPC / Parallel Knoppics

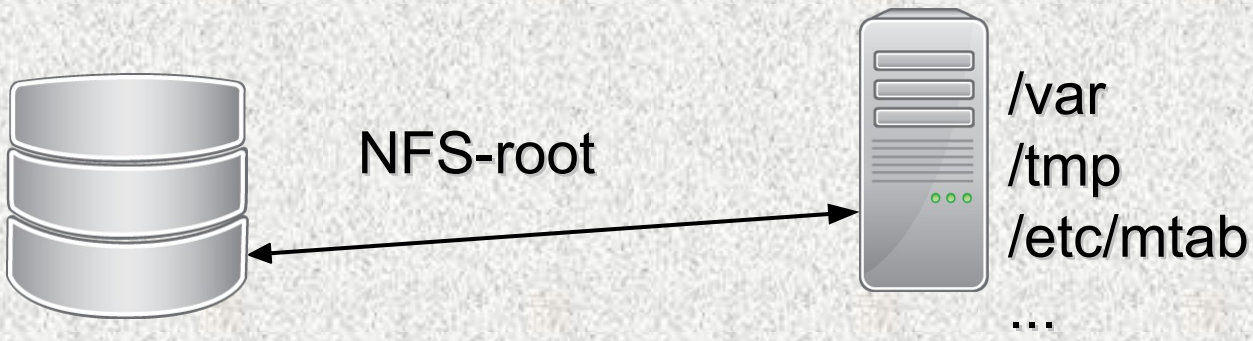
Установка ОС на диск: xCAT

- Использует anaconda/autoyast
- Управляет DNS, DHCP, TFTP
- Поддерживает удалённое управление питанием и консоль (через IPMI, ILO, ...)
- Имеет встроенные «массовые» команды

XCAT

Виды установки ОС:

- Локально (state**full**)
- По сети в память (state**less**)
- По сети, в памяти только изменения (state**lite**)



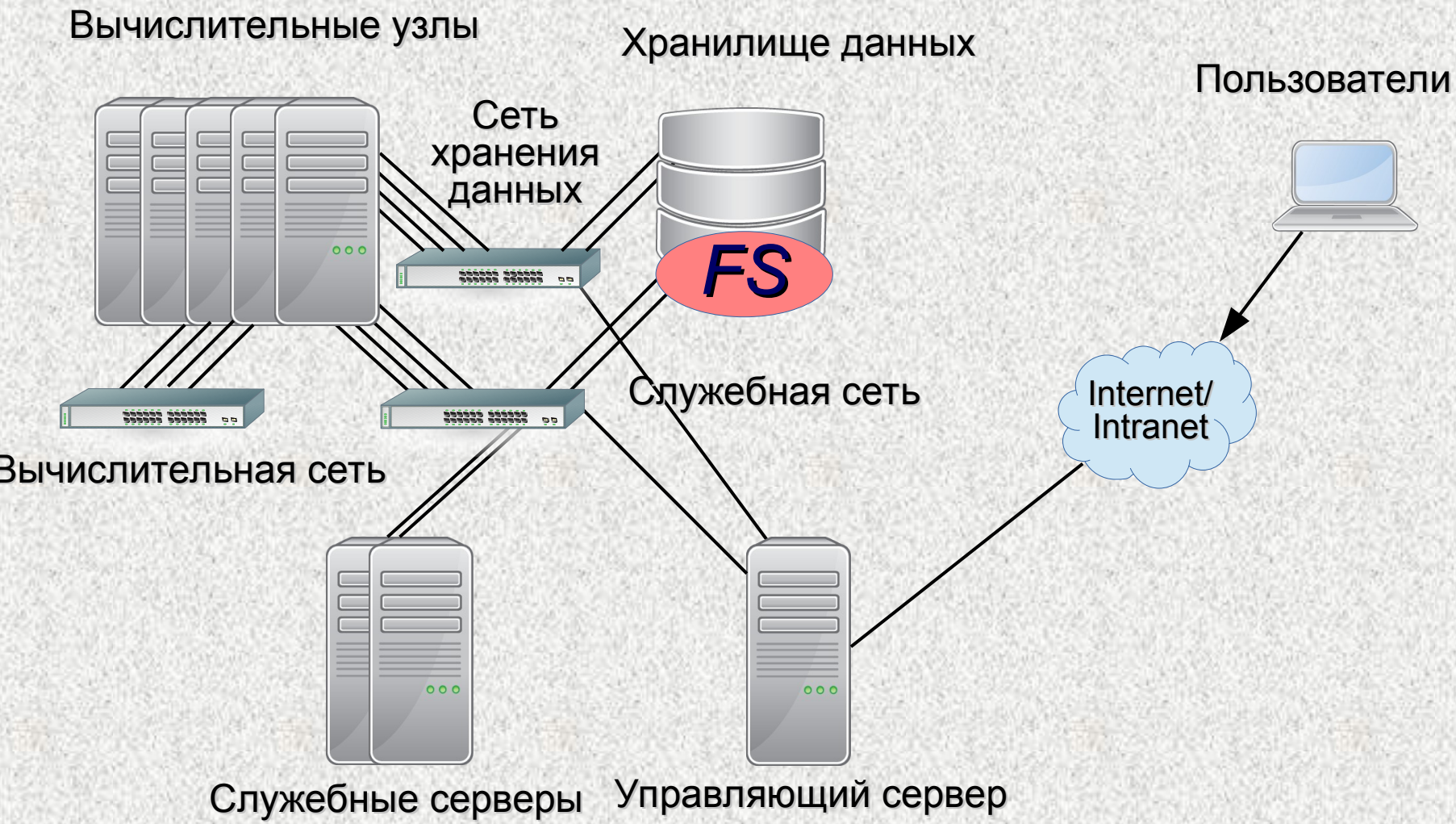
XCAT

Объекты: site, **node**, group, **network**, osdistro, osimage, route...

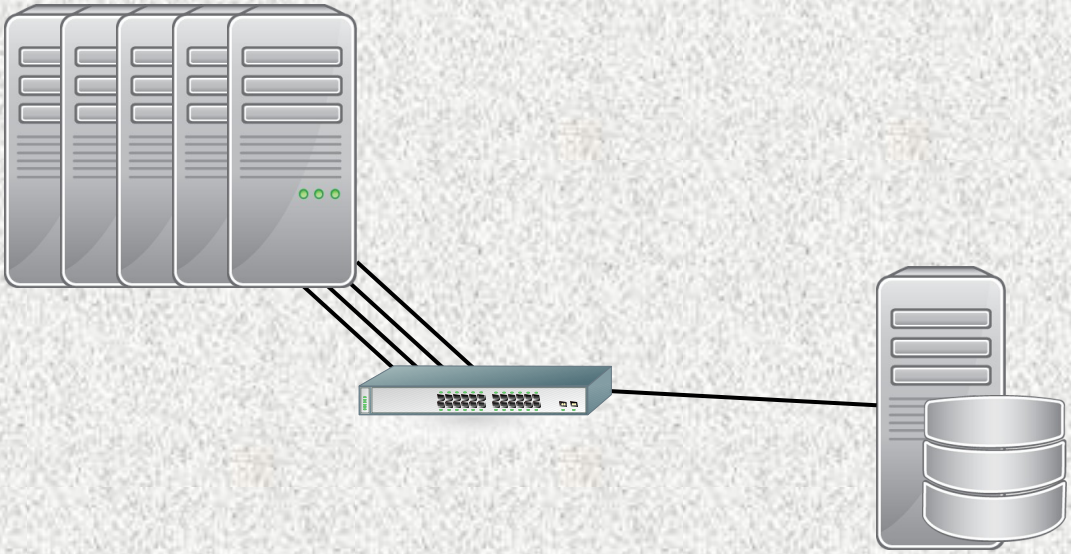
Таблицы: domain, hosts, ipmi, litetree, mac, nodegroup, nodelist, nodetype, postscripts,...

```
mkdef -t node node-1 groups=all,compute arch=x86_64 \
    bmc=node-1-ipmi bmcusername=ADMIN bmcpassword=admin \
    mac=xx:xx:xx:xx:xx:xx mgt=ipmi netboot=pxe \
    provmethod=centos67-x86_64-statelite-compute

chtab node=compute hosts.ip='|\D+(\d+)|10.0.0.(10+$1)|'
```

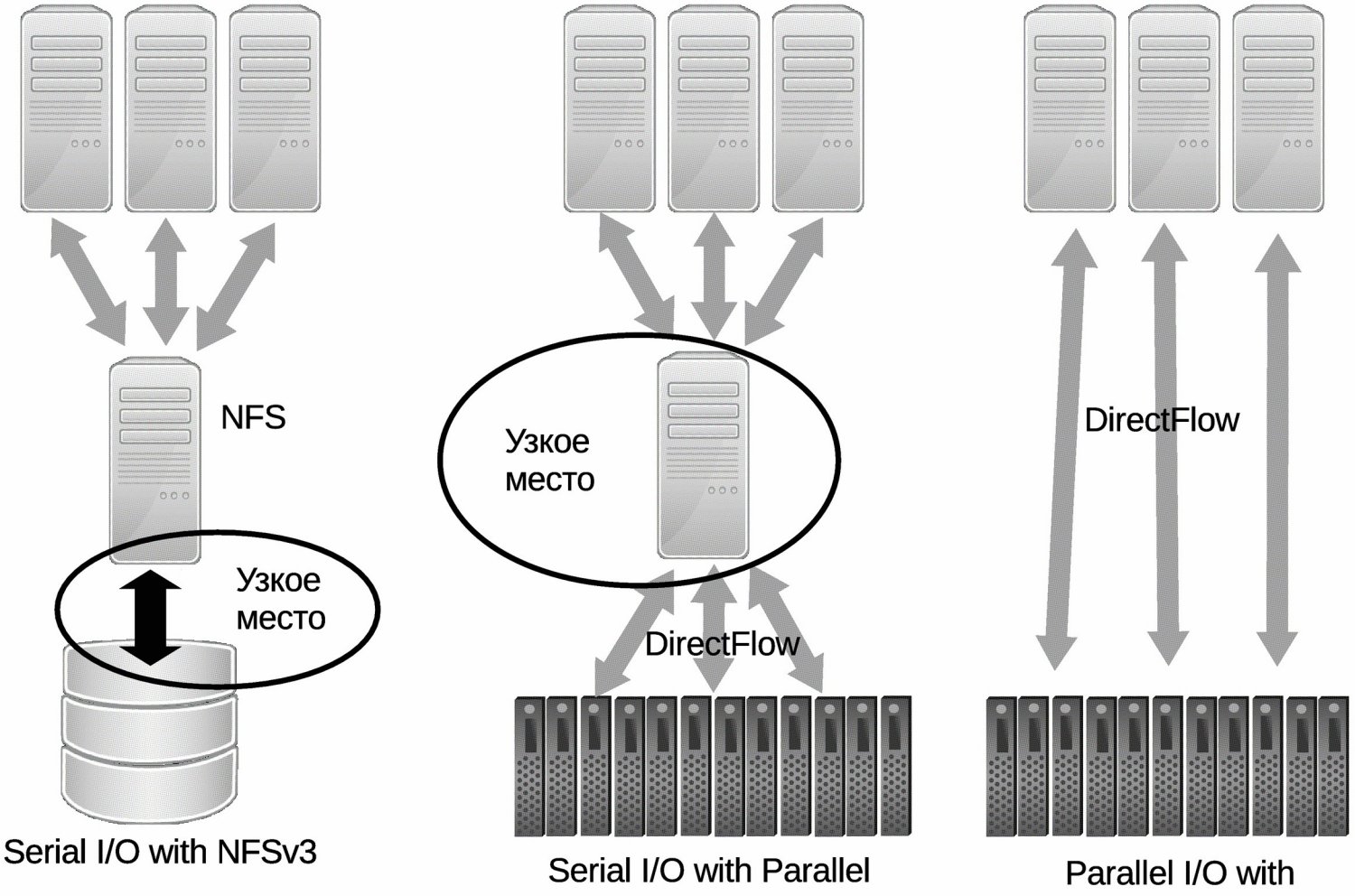
Файловые системы



NFS

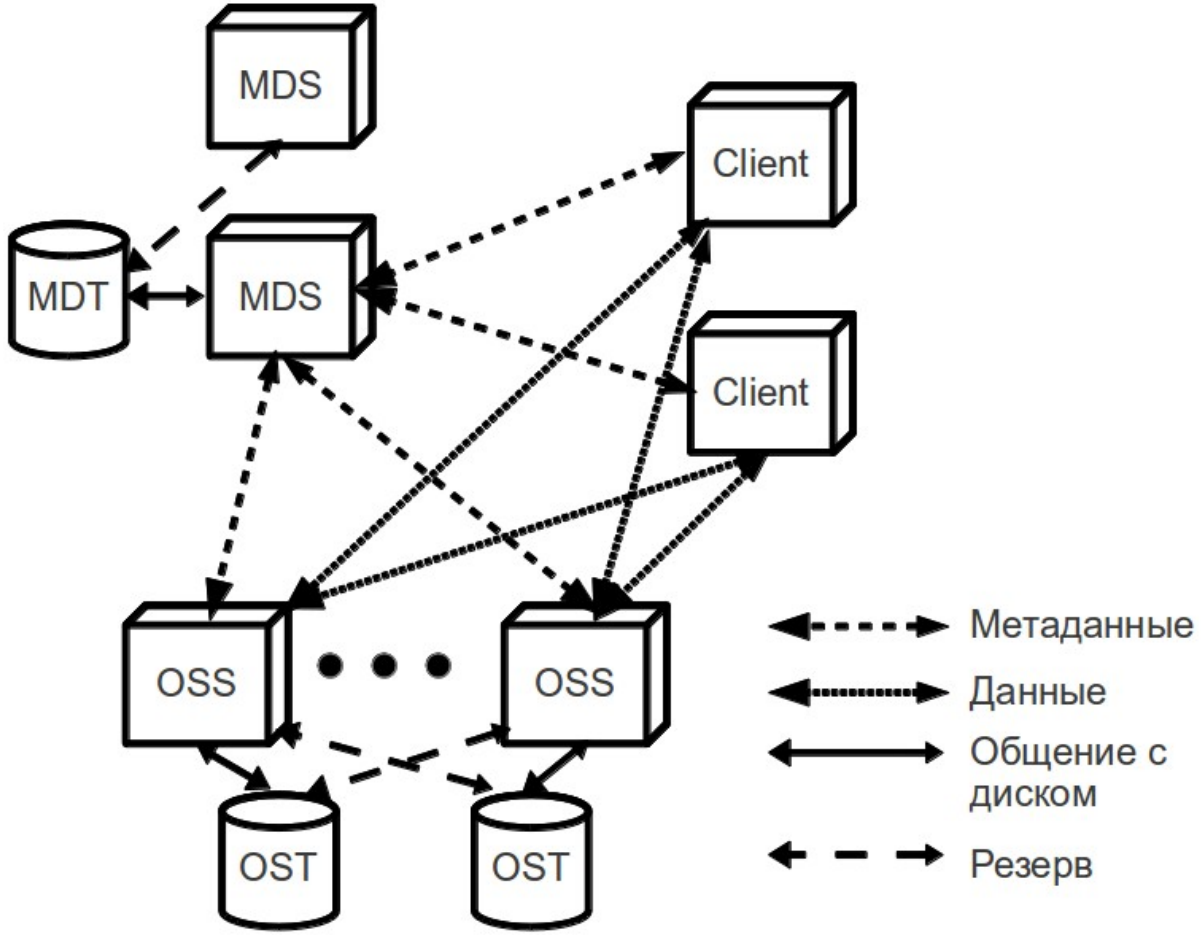


PanFS


NFS
Узкое место
Serial I/O with NFSv3
Узкое место
DirectFlow
Serial I/O with Parallel
DirectFlow
Parallel I/O with

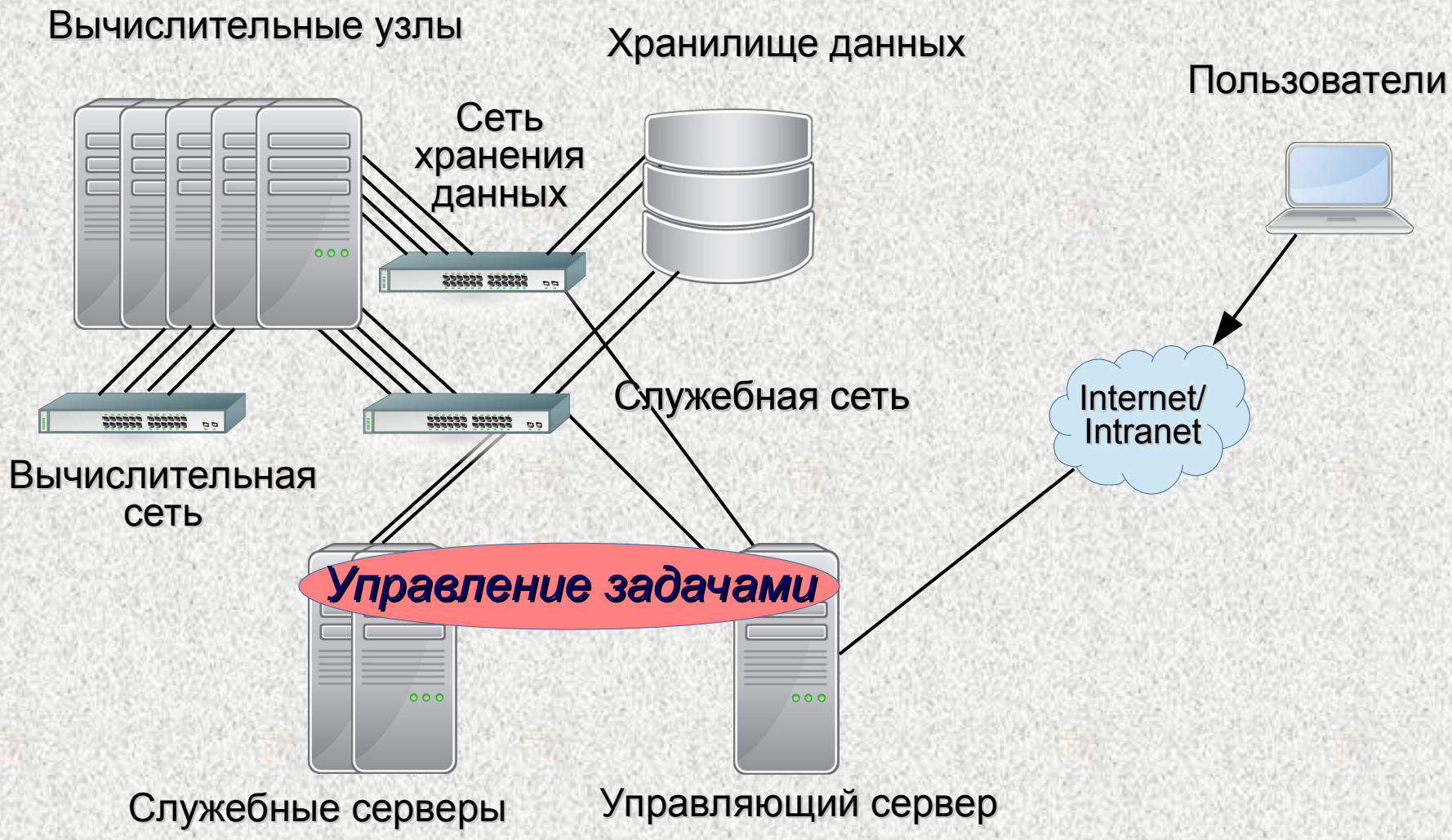
Lustre

Metadata Server(s)



ObjectStorage Servers

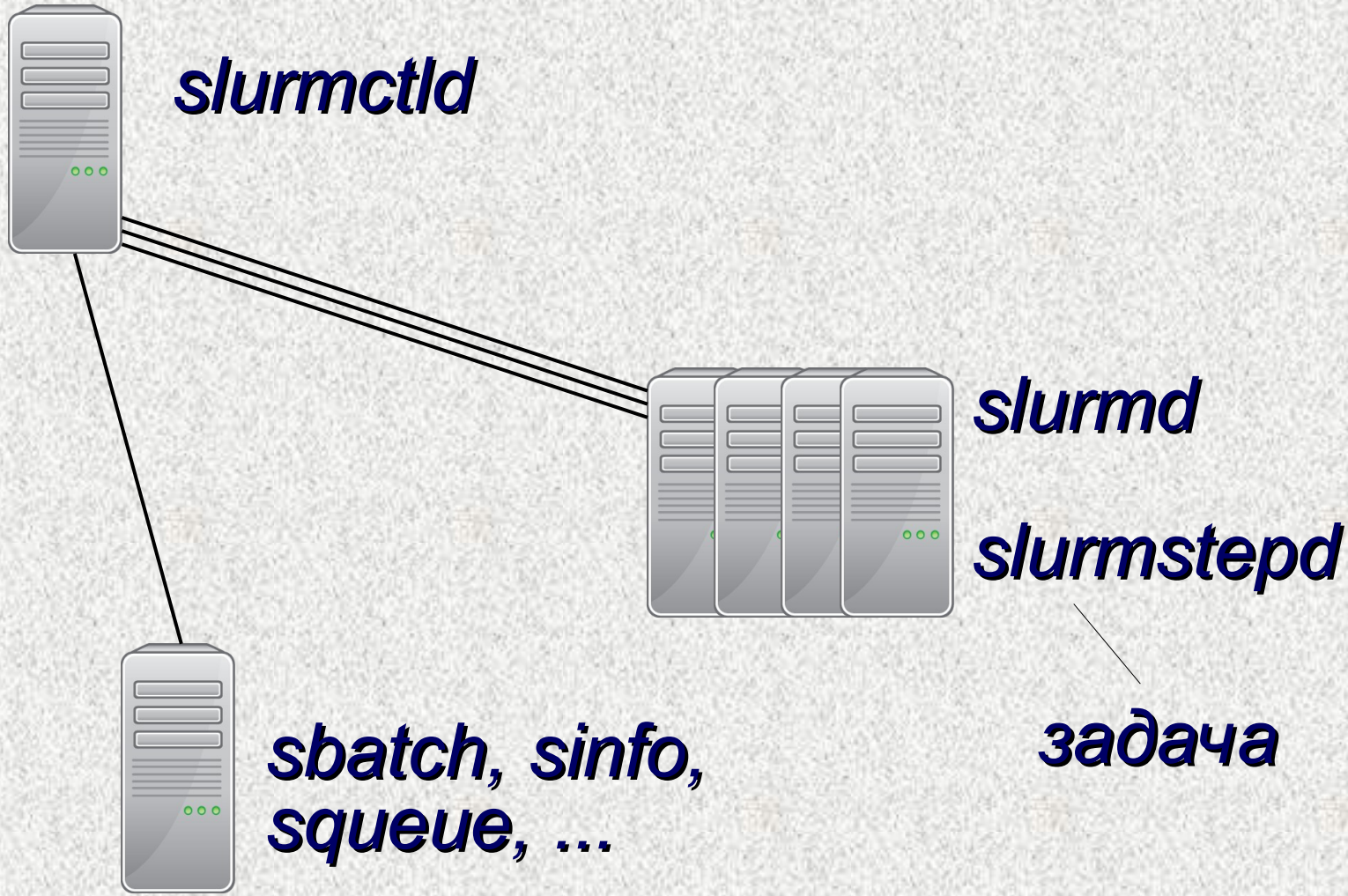
Контроль ресурсов



Управление задачами

- Slurm
- Torque
- OpenPBS
- LSF, BrightManager, Moab, ...

Slurm



Slurm

slurmctld

/etc/slurm/slurm.conf

scontrol

- show node | job | partition |...
- create ...
- delete ...
- update ...

Хранение учётных записей

- passwd
- nis+
- LDAP
- Гибридный метод

Хранение учётных записей

- **passwd**

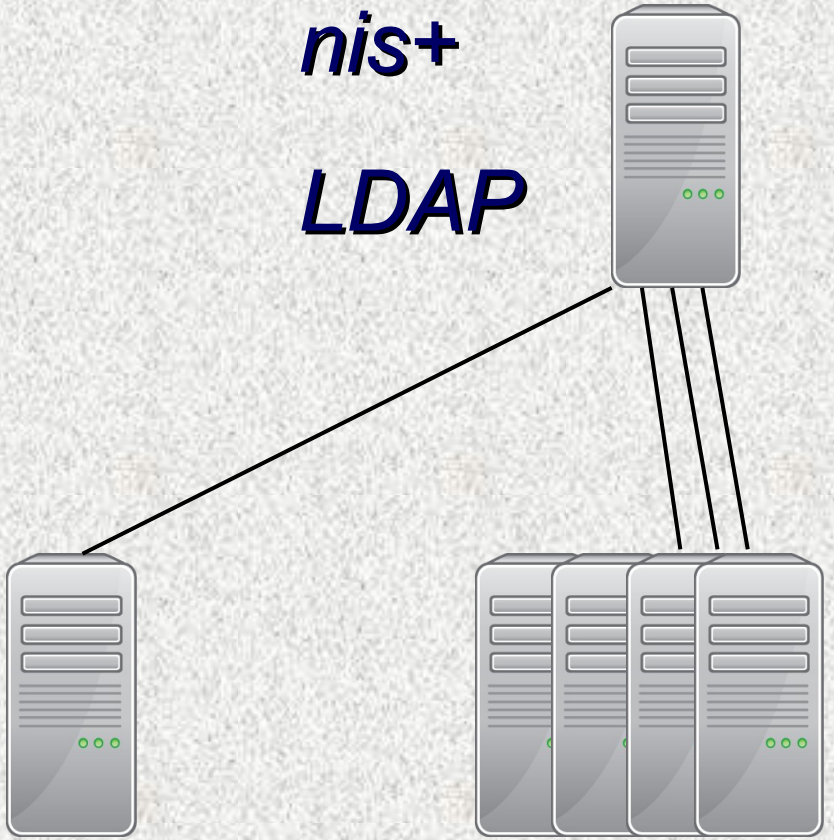
root:x:0:0:root of evil:/root:/bin/bash

- **shadow**

root:*:16219:0:99999:7:::

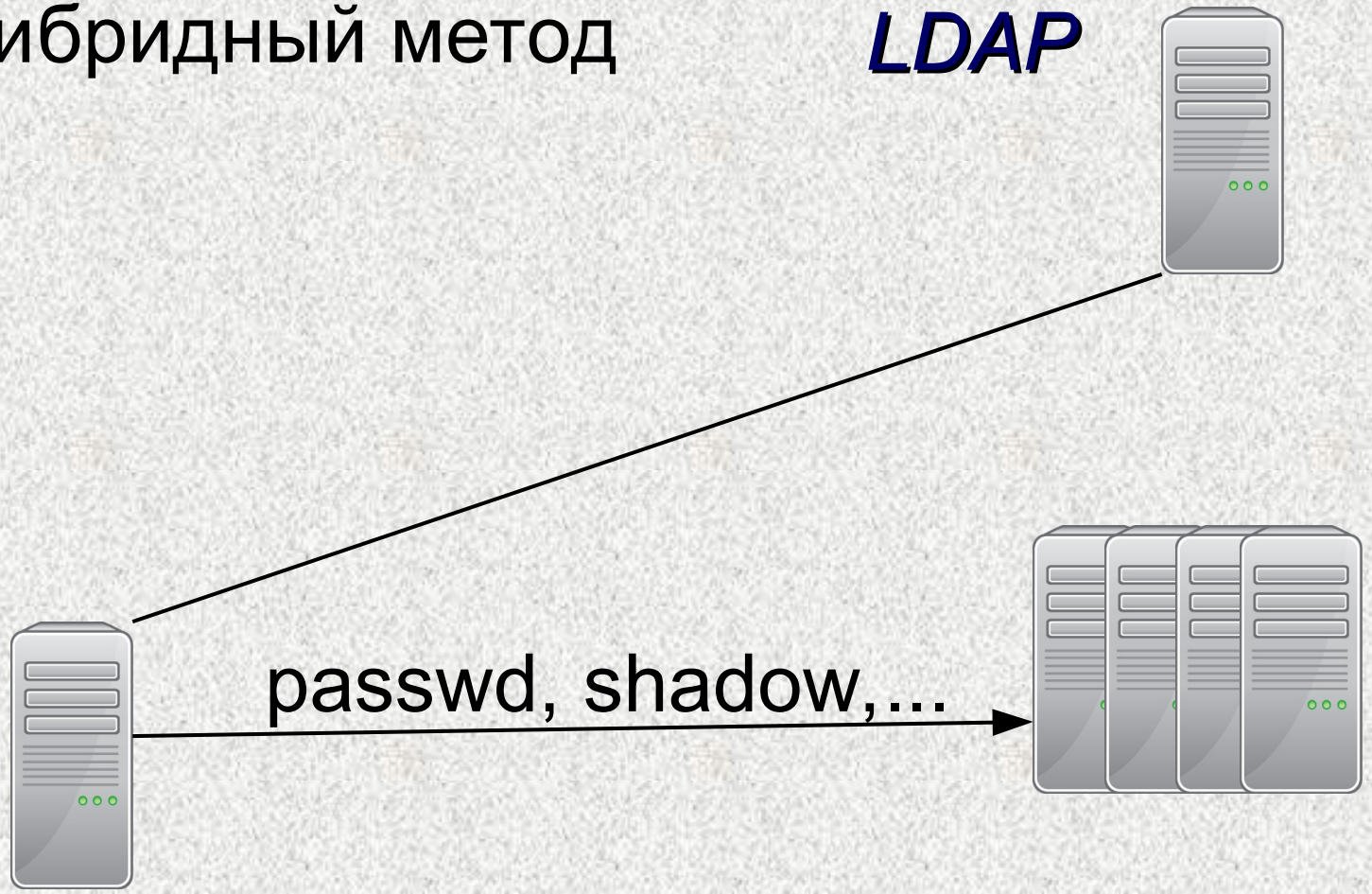
Хранение учётных записей

- LDAP



Хранение учётных записей

- Гибридный метод



Лицензии

- Локальные и сетевые
- Привязка:
  - MAC-адрес
  - Серийный номер материнской платы
  - Файл
- Тип:
  - Список пользователей/серверов
  - Число пользователей (network)
  - Число одновременных запусков (floating)

Московский государственный университет имени М.В.Ломоносова

Факультет Вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ

Жуматий С.А.

в.н.с. НИВЦ МГУ, к.ф.-м.н.

ВМК МГУ, 2020
serg@parallel.ru

Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ И
ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ

ВВЕДЕНИЕ В ТЕОРИЮ АНАЛИЗА СТРУКТУРЫ ПРОГРАММ И АЛГОРИТМОВ

(часть 1)

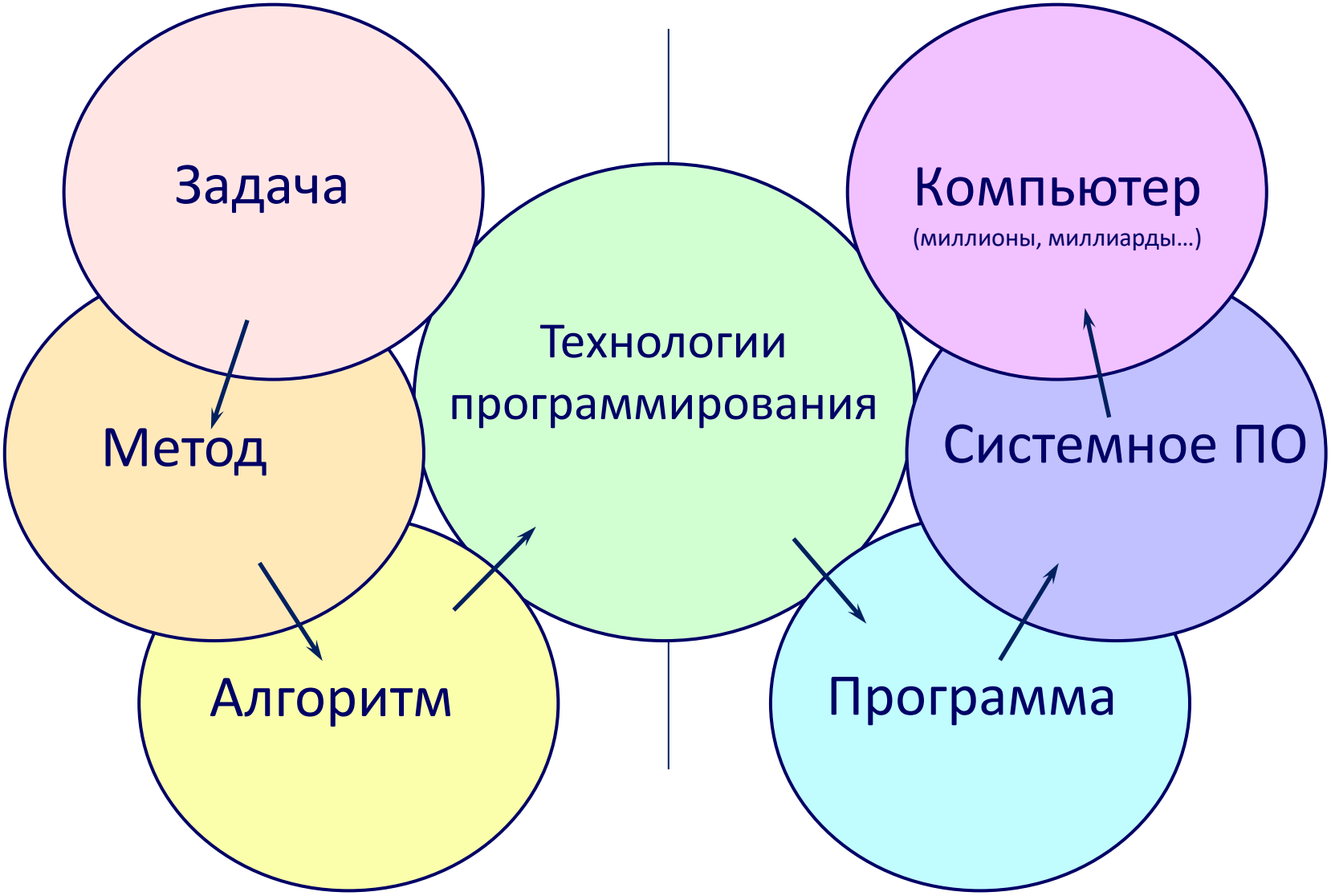
Вл.В.Воеводин

Зав.кафедрой Суперкомпьютеров и квантовой информатики ВМК МГУ
Директор НИВЦ МГУ,
д.ф.-м.н., профессор, чл.-корр. РАН

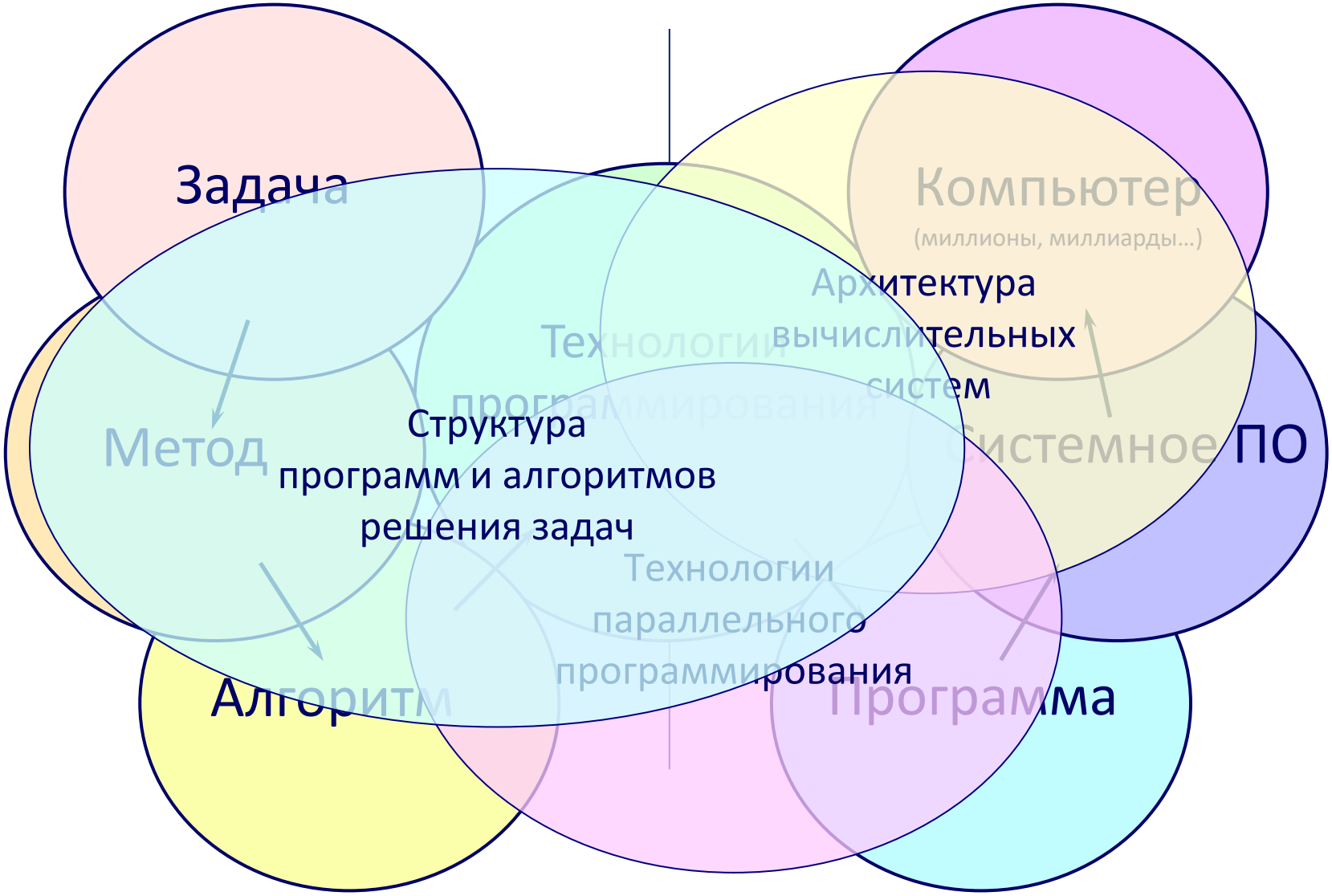
voevodin@parallel.ru

ВМК МГУ, 2020

Решение задачи на компьютере



Решение задачи на компьютере



Почему важно понимать как написаны программы?

Pascal? Fortran? C? C++?

Пользователь: почему?

$A_{ijk} = A_{i-1jk} + B_{jk} + B_{jk}$, i=1,40; j=1,40; k=1,1000

Cray C90, пиковая производительность 960 Mflop/s

```
do k = 1, 1000
   do j = 1, 40
      do i = 1, 40
         A(i,j,k) = A(i-1,j,k)+B(j,k)+B(j,k)
```

Производительность: **20** Mflop/s на Cray C90

Пользователь: почему?

$A_{ijk} = A_{i-1jk} + B_{jk} + B_{jk}$, i=1,40; j=1,40; k=1,1000

Cray C90, пиковая производительность 960 Mflop/s

```
do i = 1, 40, 2
   do j = 1, 40
      do k = 1, 1000
         A(i,j,k) = A(i-1,j,k)+2*B(j,k)
         A(i+1,j,k) = A(i,j,k)+2*B(j,k)
```

Производительность: **700** Mflop/s на Cray C90

Почему важно понимать как написаны программы?

Pascal? Fortran? C? C++?

Почему важно знать как устроены алгоритмы?

Умножение матриц: все ли просто?

Фрагмент исходного текста:

```
for( i = 0; i < n; ++i)
    for( j = 0; j < n; ++j)
        for( k = 0; k < n; ++k)
            A[i][j] = A[i][j] + B[i][k]*C[k][j]
```

Порядок циклов: (i, j, k)

Возможен ли порядок:

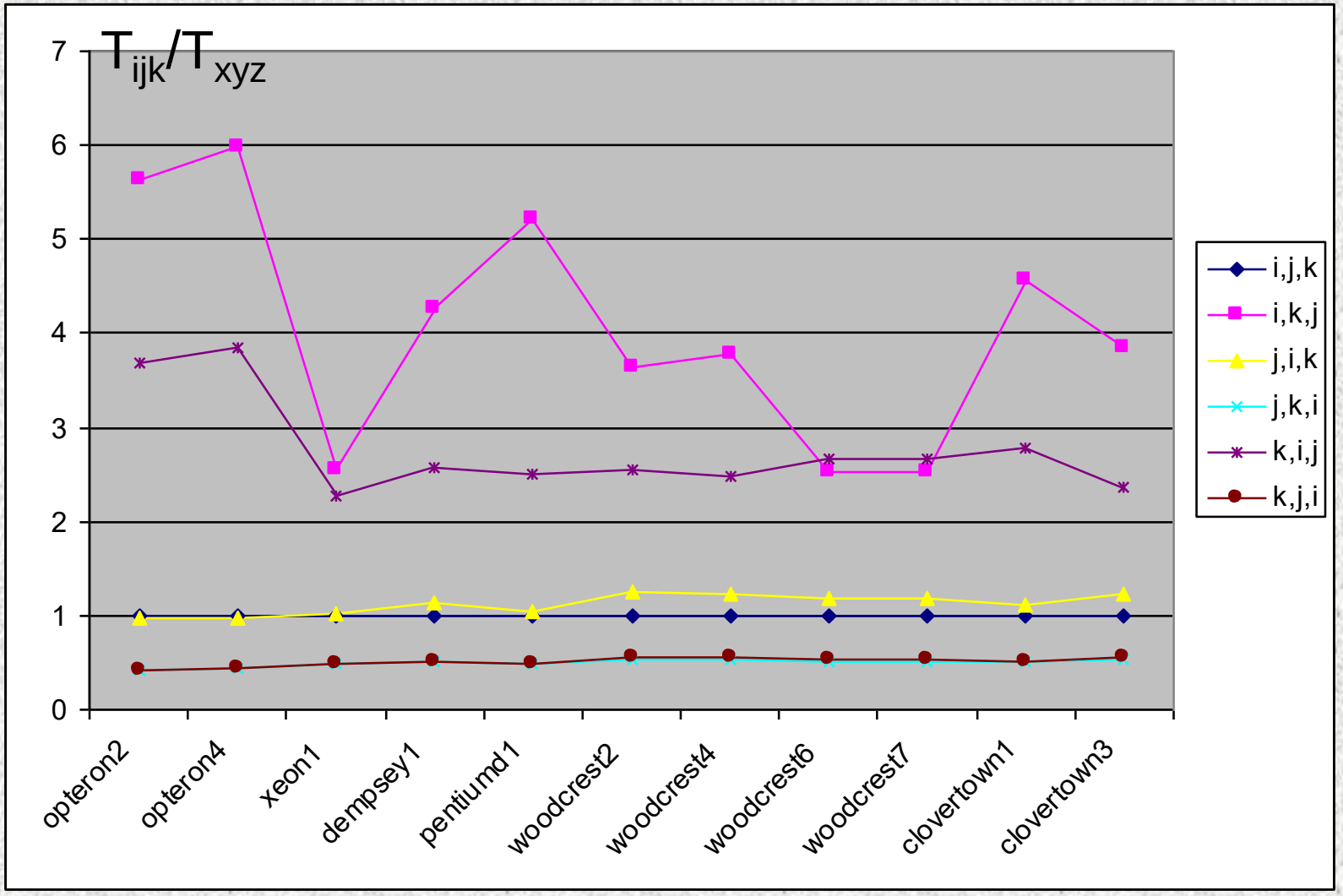
(i, k, j) - ?	ДА
(k, i, j) - ?	ДА
(k, j, i) - ?	ДА
(j, i, k) - ?	ДА
(j, k, i) - ?	ДА

Почему возможен другой порядок?

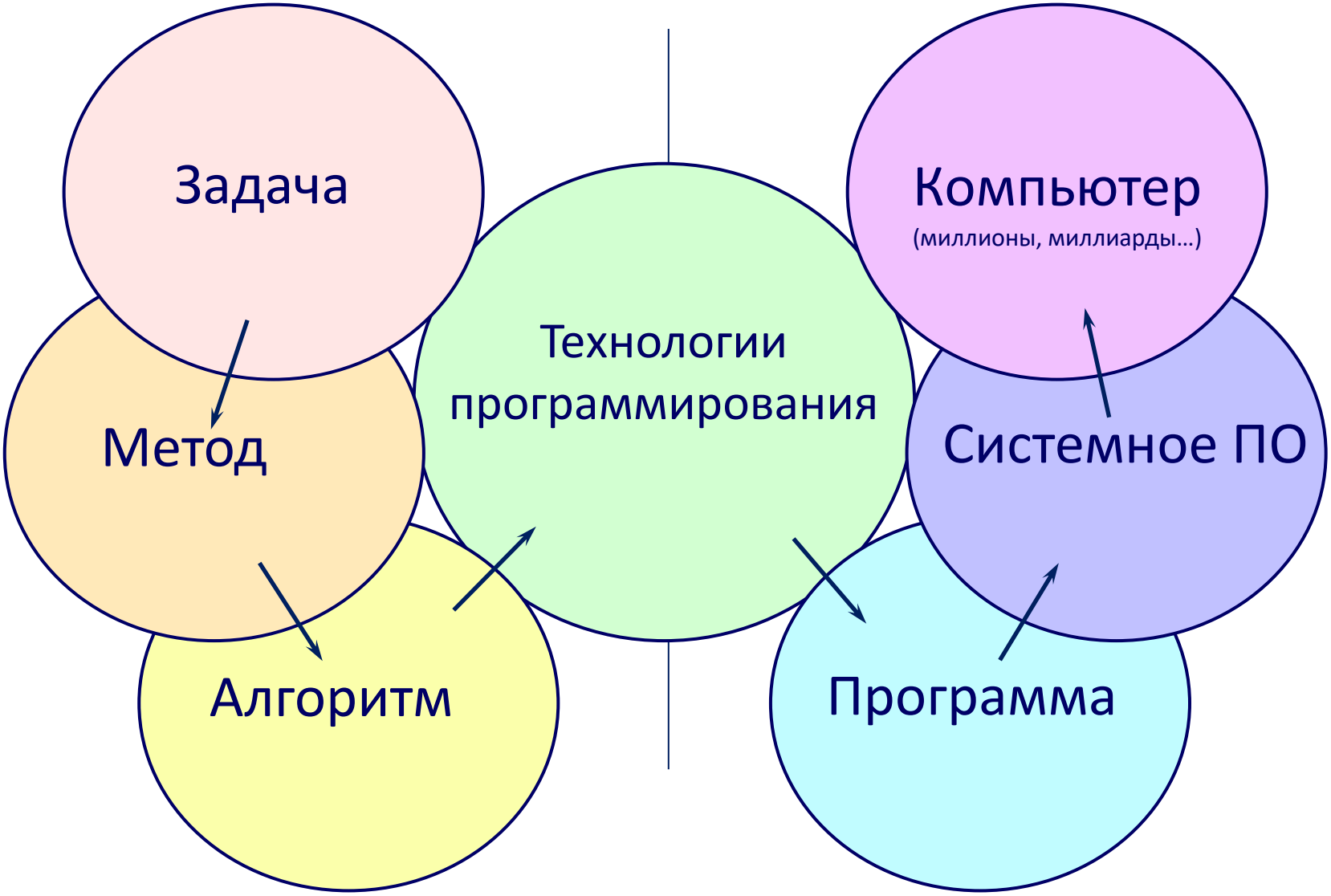
А зачем нужен другой порядок?

Умножение матриц: все ли просто?

(сравнение с порядком (i, j, k))



Решение задачи на компьютере



Графовые модели программ

Будем представлять программы с помощью графов: набор вершин и множество соединяющих их направленных дуг.

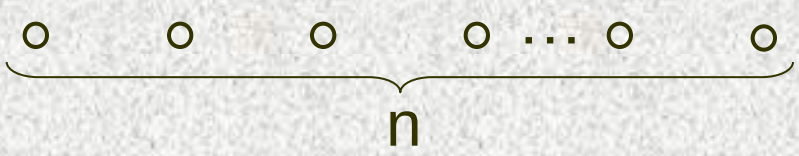
Вершины: процедуры, циклы, линейные участки, операторы, итерации циклов, срабатывания операторов…

Графовые модели программ

Будем представлять программы с помощью графов: набор вершин и множество соединяющих их направленных дуг.

Вершины: итерации циклов.

```
for( i = 0; i < n; ++i) {
    A[i] = A[i – 1] + 2;
    B[i] = B[i] + A[i];
}
```



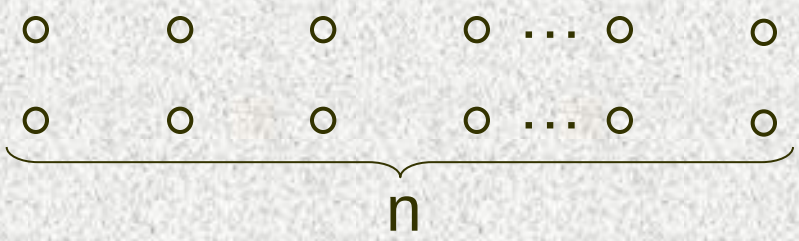
Каждая вершина соответствует двум операторам (телу цикла), выполненным на одной и той же итерации цикла.

Графовые модели программ

Будем представлять программы с помощью графов: набор вершин и множество соединяющих их направленных дуг.

Вершины: срабатывания операторов.

```
for( i = 0; i < n; ++i) {
    A[i] = A[i – 1] + 2;
    B[i] = B[i] + A[i];
}
```



Каждая вершина соответствует одному из двух операторов тела данного цикла, выполненному на некоторой итерации.

Графовые модели программ

Будем представлять программы с помощью графов: набор вершин и множество соединяющих их направленных дуг.

Вершины: процедуры, циклы, линейные участки, операторы, итерации циклов, срабатывания операторов…

Дуги: отражают связь (отношение) между вершинами. Выделяют два типа отношений:

- *операционное отношение,*
- *информационное отношение.*

Графовые модели программ

Будем представлять программы с помощью графов: набор вершин и множество соединяющих их направленных дуг.

Дуги: операционное отношение:

Две вершины A и B соединяются направленной дугой тогда и только тогда, когда вершина B может быть выполнена сразу после вершины A.

Операционное отношение = отношение по передаче управления.

Графовые модели программ

Будем представлять программы с помощью графов: набор вершин и множество соединяющих их направленных дуг.

Дуги: операционное отношение:

x(i) = a + b(i) (1)
y(i) = 2*x(i) – 3 (2)
t1 = y(i)*y(i) + 1 (3)
t2 = b(i) – y(i)*a (4)

Графовые модели программ

Будем представлять программы с помощью графов: набор вершин и множество соединяющих их направленных дуг.

Дуги: информационное отношение:

Две вершины A и B соединяются направленной дугой тогда и только тогда, когда вершина B использует в качестве аргумента некоторое значение, полученное в вершине A.

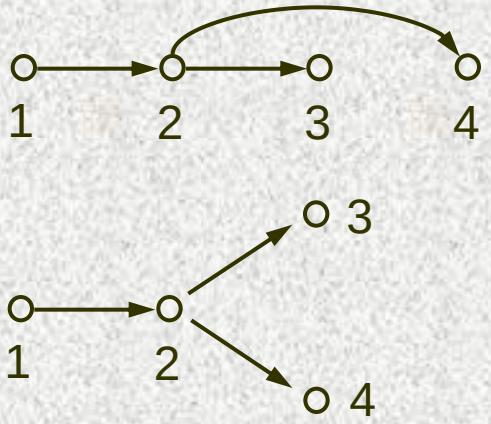
Информационное отношение = отношение по передаче данных.

Графовые модели программ

Будем представлять программы с помощью графов: набор вершин и множество соединяющих их направленных дуг.

Дуги: информационное отношение:

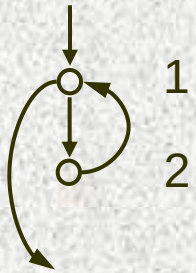
x(i) = a + b(i) (1)
y(i) = 2*x(i) – 3 (2)
t1 = y(i)*y(i) + 1 (3)
t2 = b(i) – y(i)*a (4)



Четыре основные модели программ

Граф управления программы.
Вершины: операторы
Дуги: операционное отношение

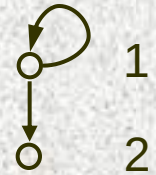
```
for( i = 0; i < n; ++i) {
    A[i] = A[i – 1] + 2;        (1)
    B[i] = B[i] + A[i];         (2)
}
```



Четыре основные модели программ

Информационный граф программы.
Вершины: операторы
Дуги: информационное отношение

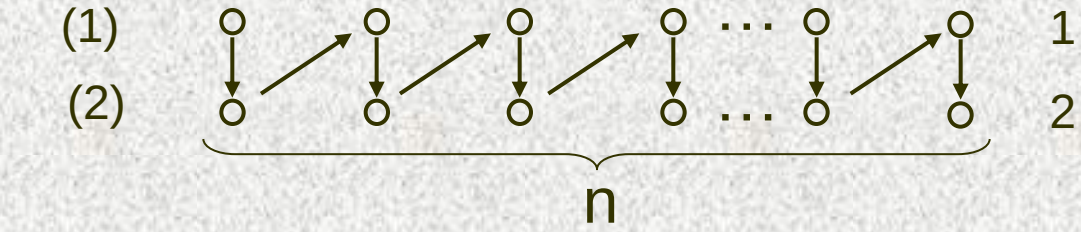
```
for( i = 0; i < n; ++i) {
    A[i] = A[i – 1] + 2;          (1)
    B[i] = B[i] + A[i];           (2)
}
```



Четыре основные модели программ

Операционная история программы.
Вершины: срабатывания операторов
Дуги: операционное отношение

```
for( i = 0; i < n; ++i) {
    A[i] = A[i – 1] + 2;
    B[i] = B[i] + A[i];
}
```

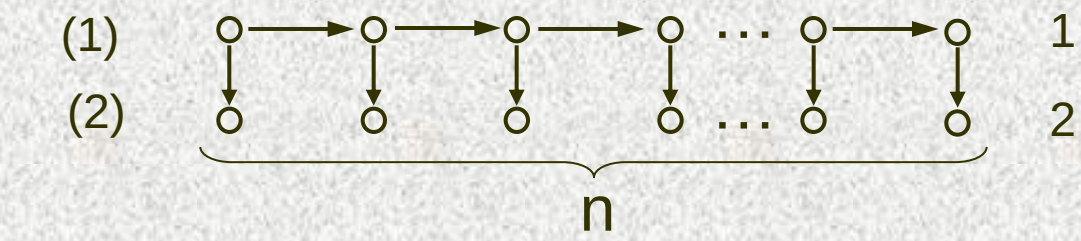


Свойства операционной истории:
- одна начальная вершина, у которой нет входящей дуги,
- одна конечная вершина, у которой нет исходящей дуги,
- у всех остальных вершин есть ровно одна входящая дуга и одна исходящая дуга.

Четыре основные модели программ

Информационная история программы.
Вершины: срабатывания операторов
Дуги: информационное отношение

```
for( i = 0; i < n; ++i) {
    A[i] = A[i – 1] + 2;
    B[i] = B[i] + A[i];
}
```



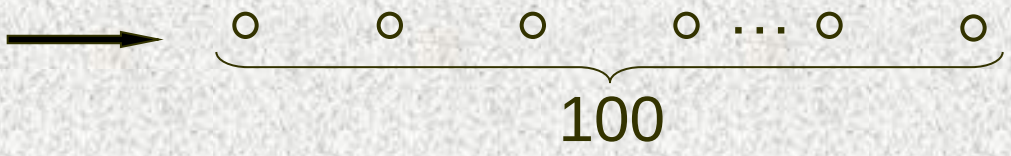
Свойства информационной истории:
- ациклический граф,
- нет кратных дуг.

Несколько вопросов…

Может ли информационная история некоторого фрагмента программы иметь 100 вершин и ни одной дуги?

ДА.

```
for( i = 0; i < 100; ++i)
    A[i] = B[i] + C[i]*x;
```

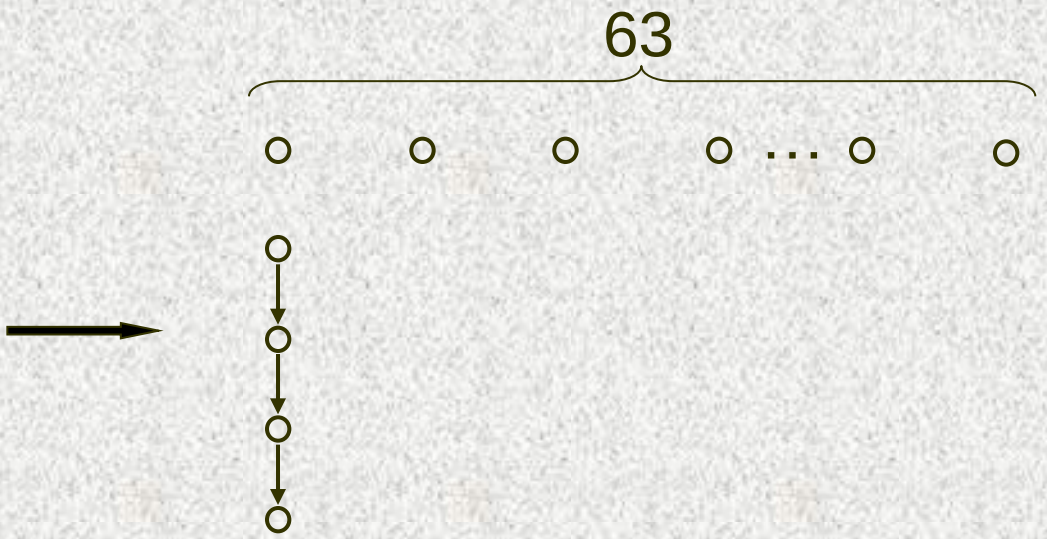


Несколько вопросов…

Может ли информационная история некоторого фрагмента программы иметь 67 вершин и 3 дуги?

ДА.

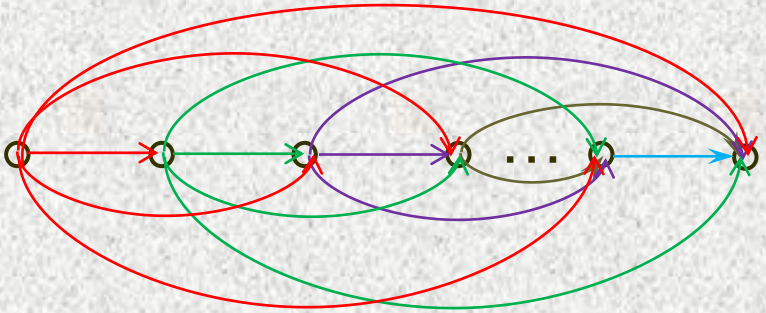
```
for( i = 0; i < 63; ++i)
    A[i] = B[i] + C[i]*x;
x1 = 10;
x2 = x1+1;
x3 = x2+2;
x4 = x3+3;
```



Несколько вопросов…

Может ли информационная история некоторого фрагмента программы иметь 20 вершин и 200 дуг?

Вспомним свойства информационной истории:
- ациклический граф,
- нет кратных дуг.



Макс.число дуг: (n-1) + (n-2) + (n-3) + … + 2 + 1 = n(n-1)/2*

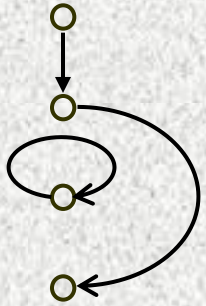
Ответ: НЕТ

Несколько вопросов…

Может ли граф управления некоторого фрагмента программы состоять из нескольких компонент связности?

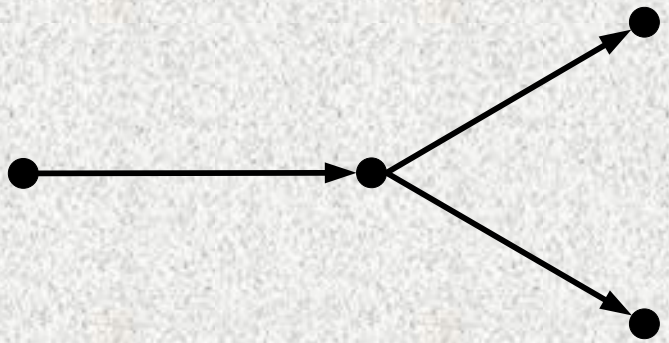
ДА.

```
x1 = 10;
x2 = x1+1; goto A;
B: x3 = x2+2; goto B;
A: x4 = x2+3;
```



Несколько вопросов…

Модель некоторого фрагмента программы в качестве подграфа содержит следующий граф:



Какой моделью могла бы быть исходная модель?

ГУ ИГ ~~ОИ~~ ИИ

Множество графовых моделей программ

(опорные точки)

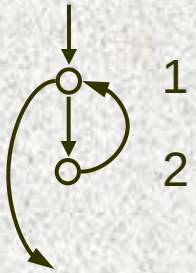
Операционные модели

Информационные модели

Компактные модели

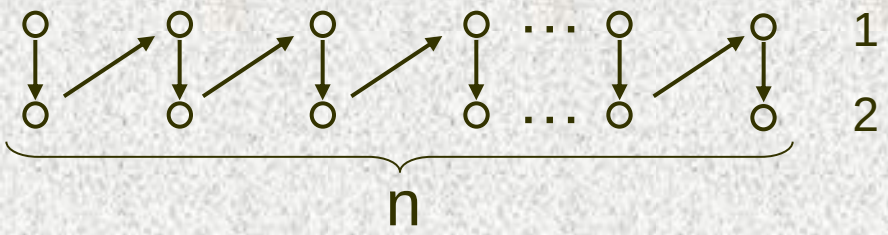
Истории

Граф управления

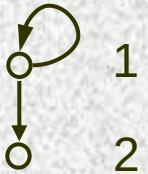


…

Операционная история

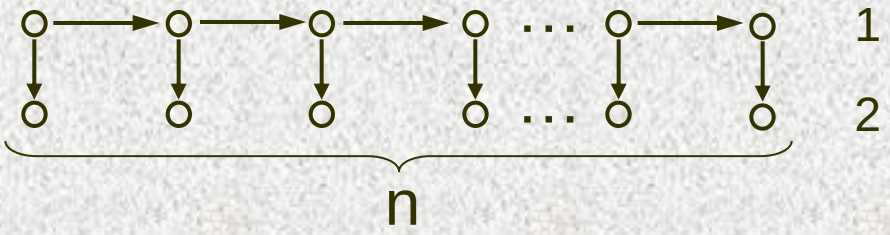


Информационный граф



…

Информационная история



Какое отношение выбрать для описания свойств программ?

Операционное отношение?

x(i) = a + b(i) (1)
y(i) = 2*x(i) – 3 (2)
t1 = y(i)*y(i) + 1 (3)
t2 = b(i) – y(i)*a (4)

Какое отношение выбрать для описания свойств программ?

Информационная структура – это основа анализа свойств программ и алгоритмов.

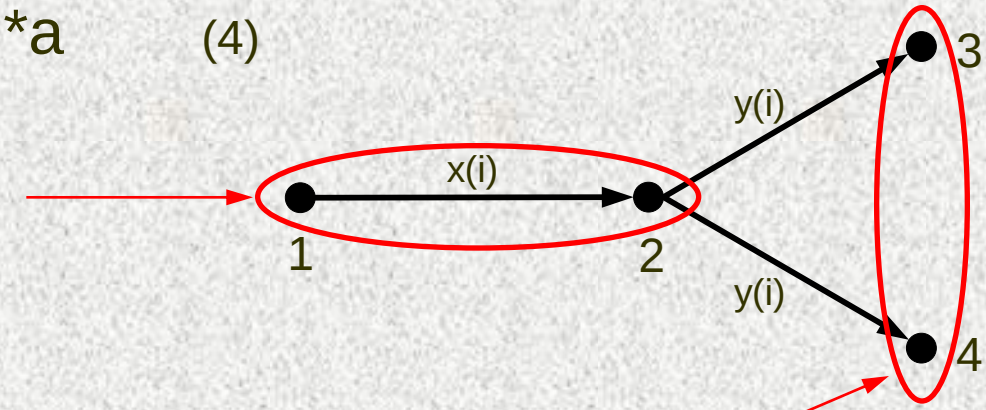
$$x(i) = a + b(i) \quad (1)$$

$$y(i) = 2*x(i) - 3 \quad (2)$$

$$t1 = y(i)*y(i) + 1 \quad (3)$$

$$t2 = b(i) - y(i)*a \quad (4)$$

Исполнять только последовательно !

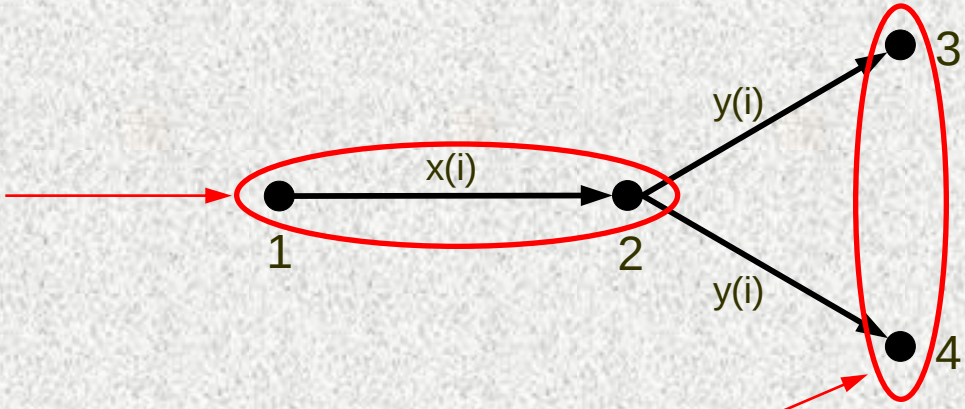


Можно исполнять параллельно !

Какое отношение выбрать для описания свойств программ?

Информационная структура – это основа анализа свойств программ и алгоритмов.

Исполнять только последовательно !



Можно исполнять параллельно !

Информационная зависимость определяет критерий эквивалентности преобразований программ.

Информационная независимость определяет ресурс параллелизма программы.

От компактных до историй: что выбрать для описания свойств программ?

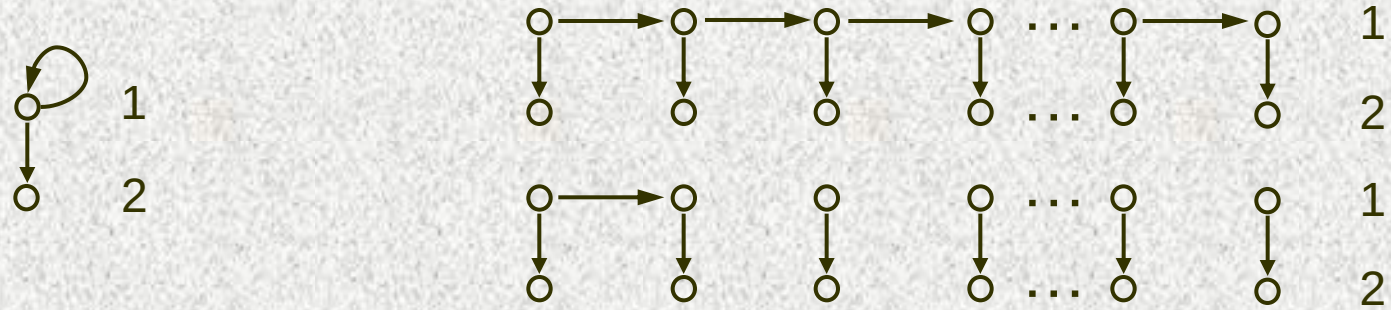
Аргументы для выбора степени компактности модели:

- *компактность описания,*
- *информативность,*
- *сложность построения.*

От компактных до историй: что выбрать для описания свойств программ?

Аргументы для выбора степени компактности модели:

- *компактность описания,*
- *информативность,*



- *сложность построения.*

От компактных до историй: что выбрать для описания свойств программ?

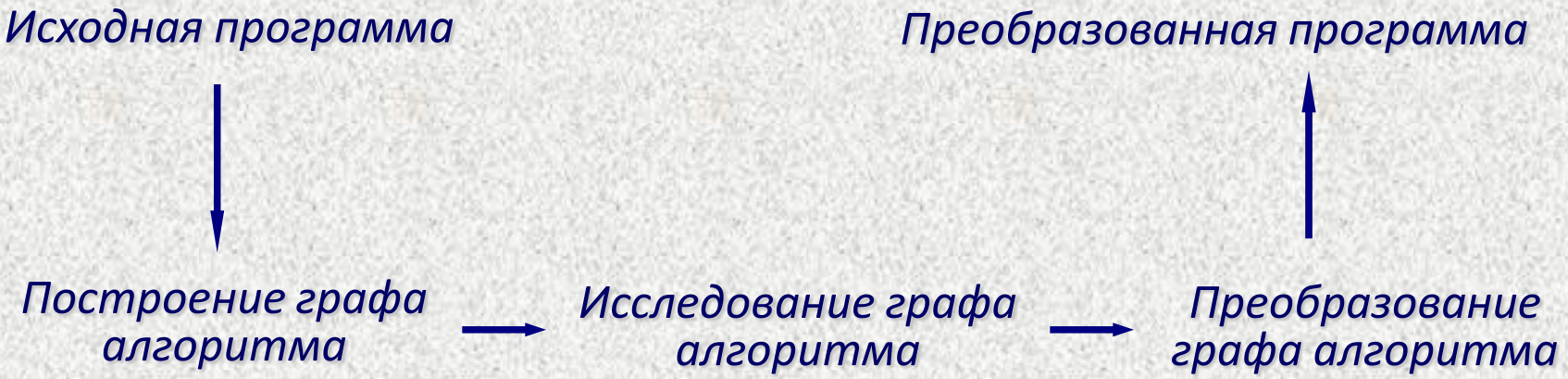
Аргументы для выбора степени компактности модели:

- *компактность описания,* *(компактные +)*
- *информативность,* *(истории +)*
- *сложность построения.* *(компактные +)*

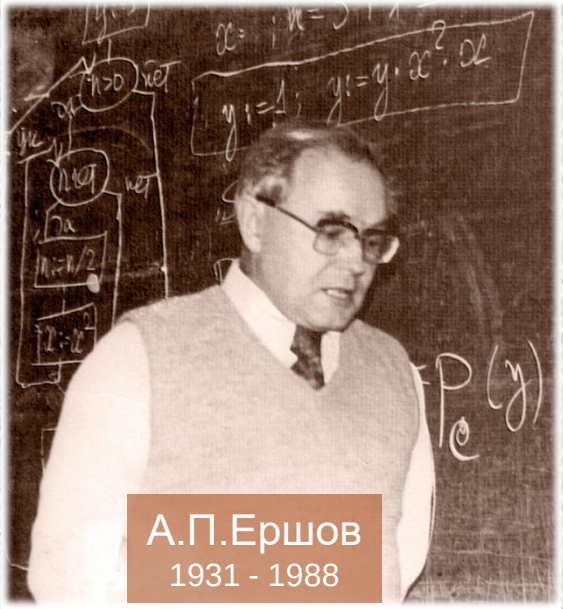
Граф алгоритма – это параметризованная информационная история:

- *компактность описания за счет параметризации,*
- *имеет информативность истории,*
- *разработана методика построения графа алгоритма по исходному тексту программ.*

Общая схема анализа и преобразования структуры программ



Основатели теории анализа структуры программ и алгоритмов



Ершов Андрей Петрович, академик, создатель сибирской школы системного и теоретического программирования. Многие его работы посвящены методам изучения свойств и структуры программ. Еще в 60-х годах он рассматривал задачу преобразования схем программ над общей и распределенной памятью, изучал фундаментальные основы графовых моделей программ.

Воеводин Валентин Васильевич, академик, создатель математической теории информационной структуры программ и алгоритмов. Разработал методы нахождения и описания информационной структуры программ по их исходному тексту, методы определения потенциала параллелизма и эквивалентного преобразования программ.

Теорема о построении графа алгоритма

<u>Теорема.</u> Если фрагмент принадлежит к линейному классу программ, то на основе статического анализа можно построить компактное описание его графа алгоритма в следующем виде:
для каждого входа каждого оператора фрагмента будет указано конечное множество троек вида

$$(N, \Delta(N), F(\Delta, N))_k ,$$

где:
N – линейный выпуклый многогранник в пространстве внешних переменных фрагмента,
$\Delta(N)$ – линейный выпуклый многогранник в пространстве итераций фрагмента,
$F(\Delta, N)$ – линейная векторная функция, описывающая входящие дуги.

Теорема о построении графа алгоритма

(…простыми словами…)

<u>Теорема.</u> Если фрагмент принадлежит к линейному классу программ, то на основе статического анализа можно построить компактное описание его графа алгоритма в следующем виде:
для каждого входа каждого оператора фрагмента будет указано конечное множество троек вида

$$(N, \Delta(N), F(\Delta, N))_k ,$$

где:
N – линейный выпуклый многогранник в пространстве внешних переменных фрагмента,
Δ(N) – линейный выпуклый многогранник в пространстве итераций фрагмента,
F(Δ, N) – линейная векторная функция, описывающая входящие дуги.

аргументы операторов

входные данные программы

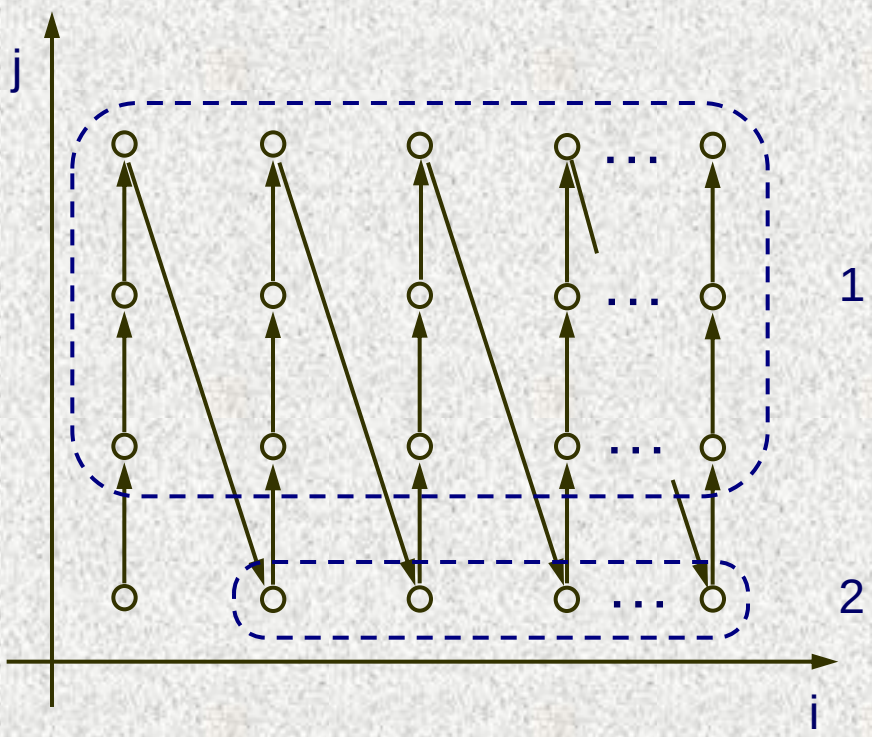
срабатывания операторов

информационное отношение

Как выполняется описание структуры программ на практике?

Программы и их графы алгоритма

```
Do i = 1, n
   Do j = 1, m
      s = s + A(i, j)
```



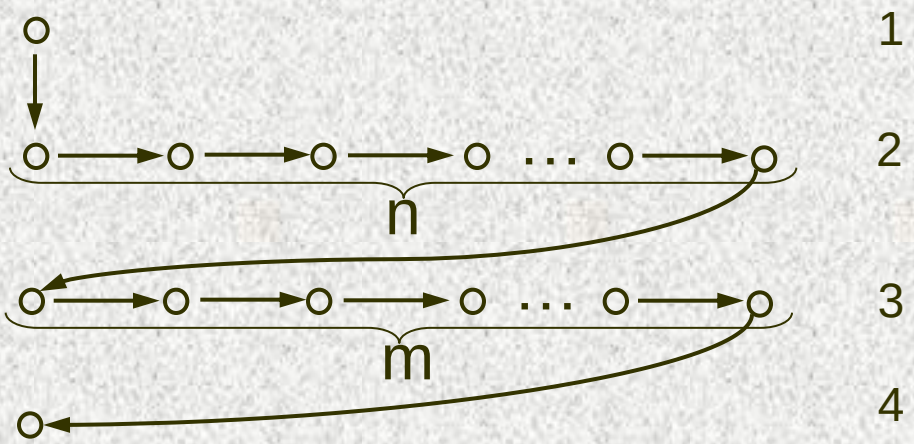
Для входа s:

$$N_1 = \begin{cases} n \ge 1 \\ m \ge 2 \end{cases} \quad I_1 = \begin{cases} 1 \le i \le n \\ 2 \le j \le m \end{cases} \quad F_1 = \begin{cases} i' = i \\ j' = j - 1 \end{cases}$$

$$N_2 = \begin{cases} n \ge 2 \\ m \ge 1 \end{cases} \quad I_2 = \begin{cases} 2 \le i \le n \\ j = 1 \end{cases} \quad F_2 = \begin{cases} i' = i - 1 \\ j' = m \end{cases}$$

Программы и их графы алгоритма

```
s = 0                    (1)
Do i = 1, n
   s = s + 1             (2)
Do i = 1, m
   s = s + 1             (3)
s = s + 1                (4)
```



$$\begin{cases} m \geq 1 \\ j_1 = m \\ из\ 3 \end{cases} \qquad \begin{cases} m < 1 \\ n \geq 1 \\ j_1 = n \\ из\ 2 \end{cases} \qquad \begin{cases} m < 1 \\ n < 1 \\ из\ 1 \end{cases}$$

Вспомним прошлые лекции…

(Умножение матриц: все ли просто?)

Фрагмент исходного текста:

```
for( i = 0; i < n; ++i)
    for( j = 0; j < n; ++j)
        for( k = 0; k < n; ++k)
            A[i][j] = A[i][j] + B[i][k]*C[k][j]
```

Порядок циклов: (i, j, k)

Возможен ли порядок:

(i, k, j) - ?	ДА
(k, i, j) - ?	ДА
(k, j, i) - ?	ДА
(j, i, k) - ?	ДА
(j, k, i) - ?	ДА

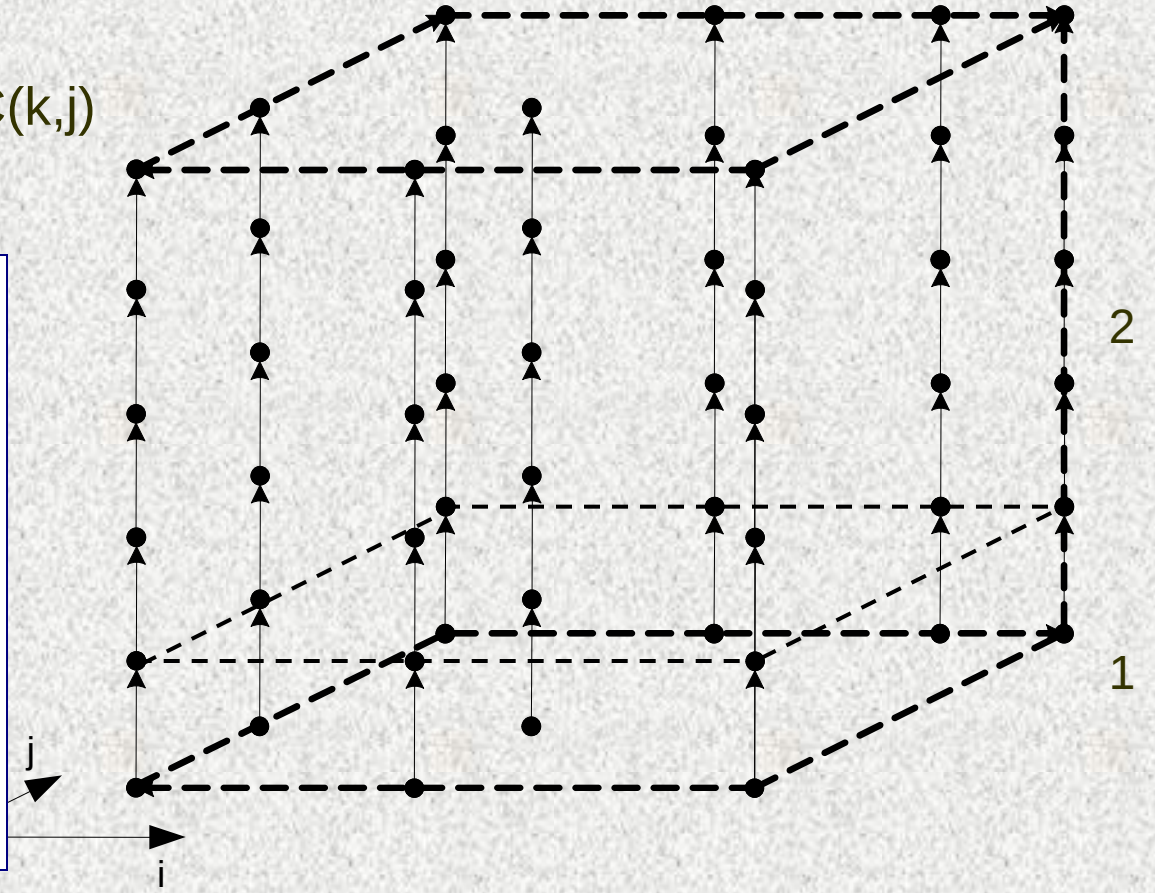
Почему возможен другой порядок?

А зачем нужен другой порядок?

Программы и их графы алгоритма
(умножение матриц)

```
    Do i = 1, n
       Do j = 1, n
1      A(i,j) = 0
         Do k = 1, n
2          A(i,j) = A(i,j) + B(i,k)*C(k,j)
```

$$\begin{cases} 1 \le i \le n \\ 1 \le j \le n \\ k = 1 \\ \begin{cases} i_1 = i \\ j_1 = j \end{cases} \\ из\ (1) \end{cases} \qquad \begin{cases} 1 \le i \le n \\ 1 \le j \le n \\ 2 \le k \le n \\ \begin{cases} i_1 = i \\ j_1 = j \\ k_1 = k - 1 \end{cases} \\ из\ (2) \end{cases}$$



Предварительные объявления о зачете…

- *Крайний срок сдачи отчетов по практическому заданию – 3 декабря.*
- *Заявлены три выступления в формате стенд-ап, которые пройдут на второй половине лекции 8-го декабря, 10 минут на каждое выступление:*

1. Черкасов Владислав, 416 группа. Как суперкомпьютер справляется с приземлёнными задачами?
2. Раева Анастасия, 412 группа. Суперкомпьютеры и их применение.
3. Гринев Тимофей, 416 группа. Введение в теорию суперлюдей (пародия на теорию суперкомпьютеров)

- *Список вопросов к зачету опубликован на https://parallel.ru/vmk20*
- *Консультация по курсу состоится 12 декабря, суббота, в 14 часов. Ссылку вышлем дополнительно.*
- *Итоги по конкурсу изображений, выступлениям в формате стенд-ап и выполнению практических заданий будут подведены к 12 декабря.*
- *Электронное тестирование пройдет 15-го декабря в 14:35, пароли будут высланы старостам групп.*
- *Устные зачеты (экзамены) – 24 и 28 декабря с 10:00, расписание по группам и ссылки будут высланы позже. Необходима включенная камера на стороне студентов и преподавателей.*

Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ И
ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ

ВВЕДЕНИЕ В ТЕОРИЮ АНАЛИЗА СТРУКТУРЫ ПРОГРАММ И АЛГОРИТМОВ

(часть 1)

Вл.В.Воеводин

Зав.кафедрой Суперкомпьютеров и квантовой информатики ВМК МГУ
Директор НИВЦ МГУ,
д.ф.-м.н., профессор, чл.-корр. РАН

voevodin@parallel.ru

ВМК МГУ, 2020

Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ И
ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ

ВВЕДЕНИЕ В ТЕОРИЮ АНАЛИЗА СТРУКТУРЫ ПРОГРАММ И АЛГОРИТМОВ

(часть 2)

Вл.В.Воеводин

Зав.кафедрой Суперкомпьютеров и квантовой информатики ВМК МГУ
Директор НИВЦ МГУ,
д.ф.-м.н., профессор, чл.-корр. РАН

voevodin@parallel.ru

ВМК МГУ, 2020

Множество графовых моделей программ

(опорные точки)

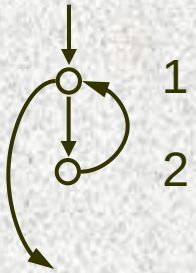
Операционные модели

Информационные модели

Компактные модели

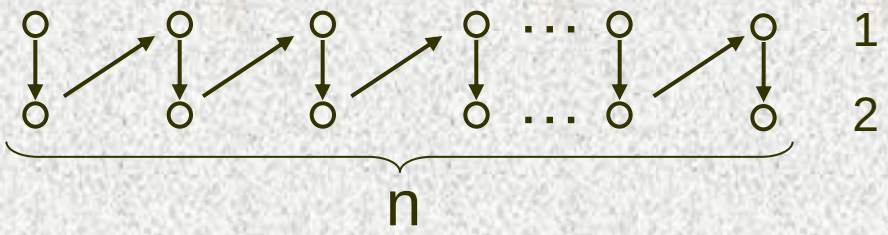
Истории

Граф управления

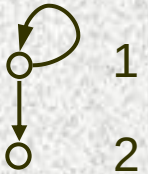


…

Операционная история

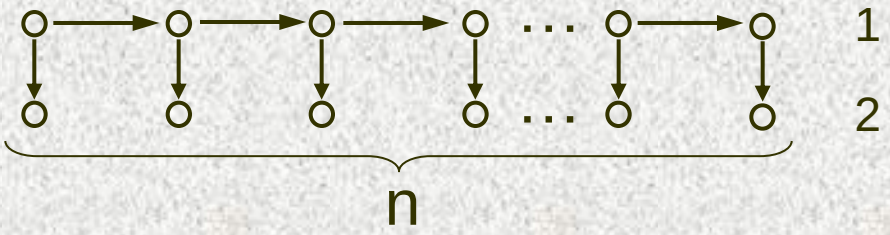


Информационный граф



…

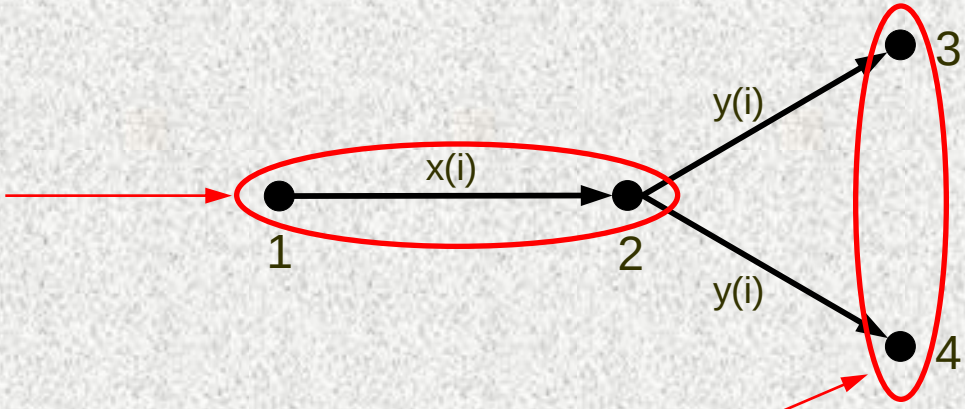
Информационная история



Какое отношение выбрать для описания свойств программ?

Информационная структура – это основа анализа свойств программ и алгоритмов.

Исполнять только последовательно !



Можно исполнять параллельно !

Информационная зависимость определяет критерий эквивалентности преобразований программ.

Информационная независимость определяет ресурс параллелизма программы.

От компактных до историй: что выбрать для описания свойств программ?

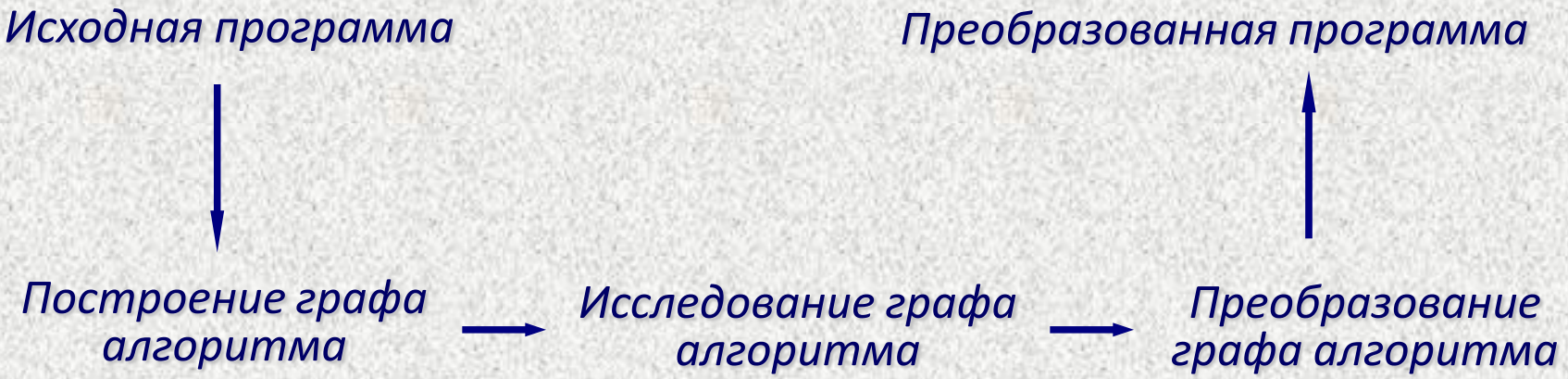
Аргументы для выбора степени компактности модели:

- *компактность описания,* *(компактные +)*
- *информативность,* *(истории +)*
- *сложность построения.* *(компактные +)*

Граф алгоритма – это параметризованная информационная история:

- *компактность описания за счет параметризации,*
- *имеет информативность истории,*
- *разработана методика построения графа алгоритма по исходному тексту программ.*

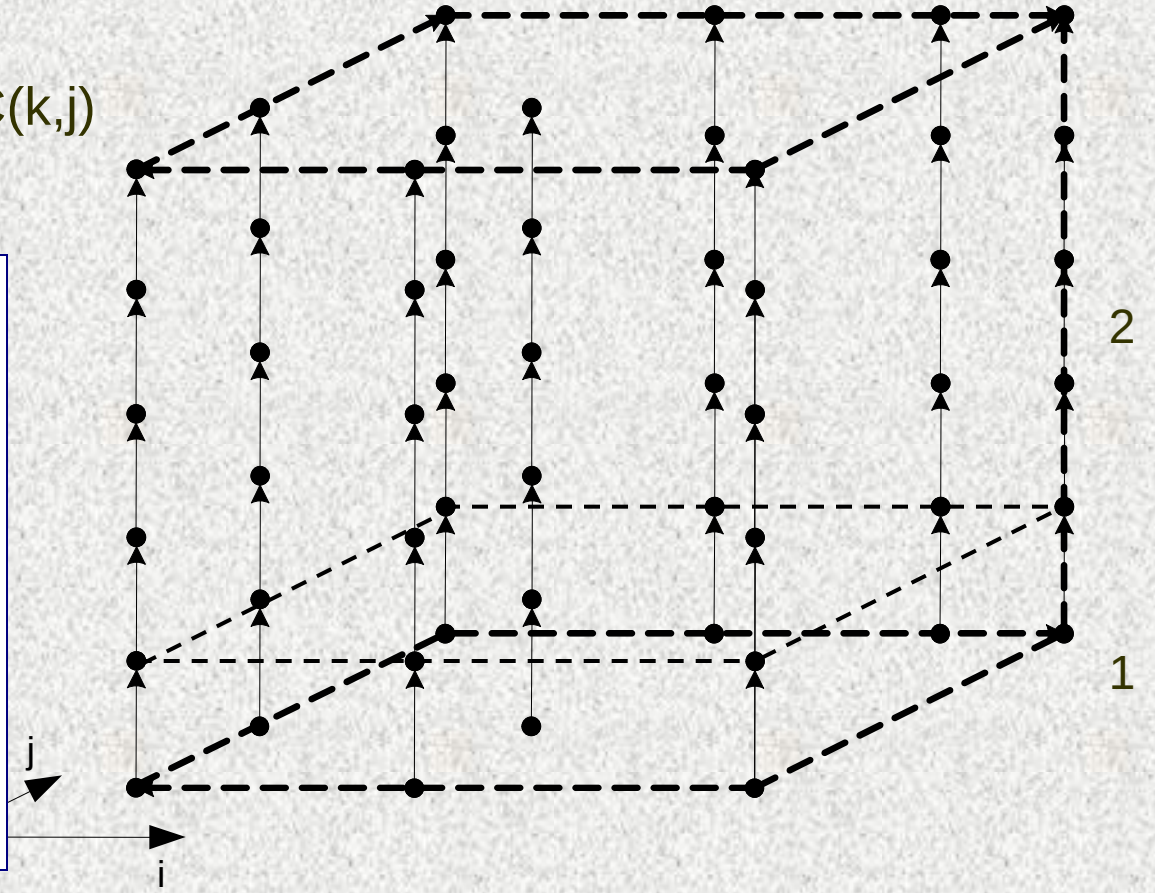
Общая схема анализа и преобразования структуры программ



Программы и их графы алгоритма
(умножение матриц)

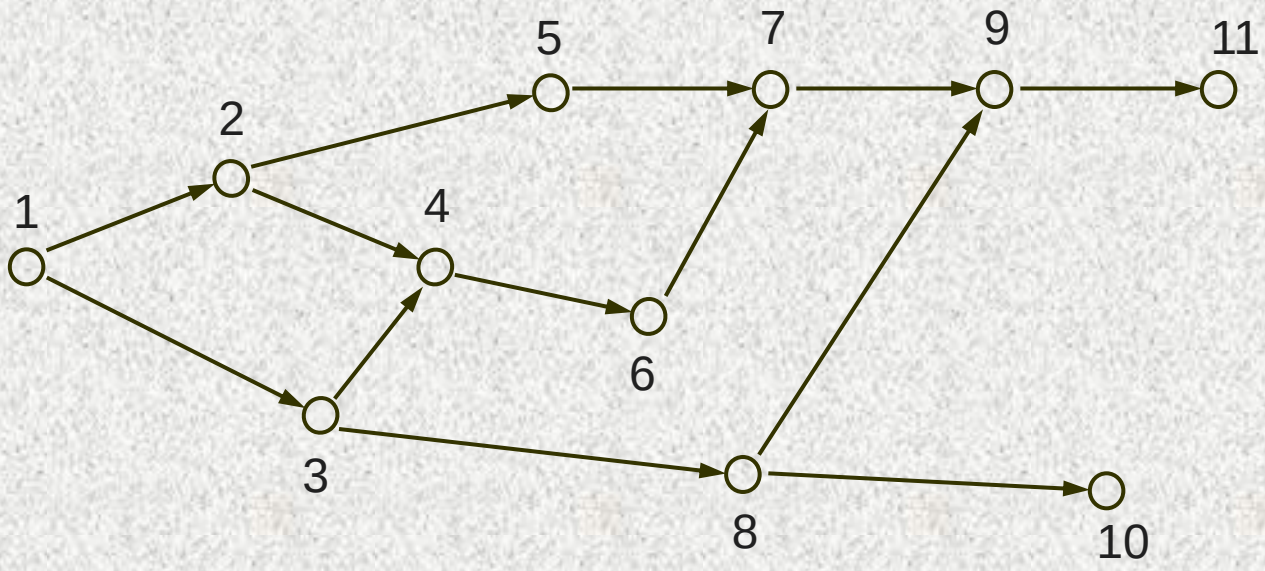
```
    Do i = 1, n
       Do j = 1, n
1      A(i,j) = 0
         Do k = 1, n
2          A(i,j) = A(i,j) + B(i,k)*C(k,j)
```

$$\begin{cases} 1 \le i \le n \\ 1 \le j \le n \\ k = 1 \\ \begin{cases} i_1 = i \\ j_1 = j \end{cases} \\ из\ (1) \end{cases} \qquad \begin{cases} 1 \le i \le n \\ 1 \le j \le n \\ 2 \le k \le n \\ \begin{cases} i_1 = i \\ j_1 = j \\ k_1 = k - 1 \end{cases} \\ из\ (2) \end{cases}$$



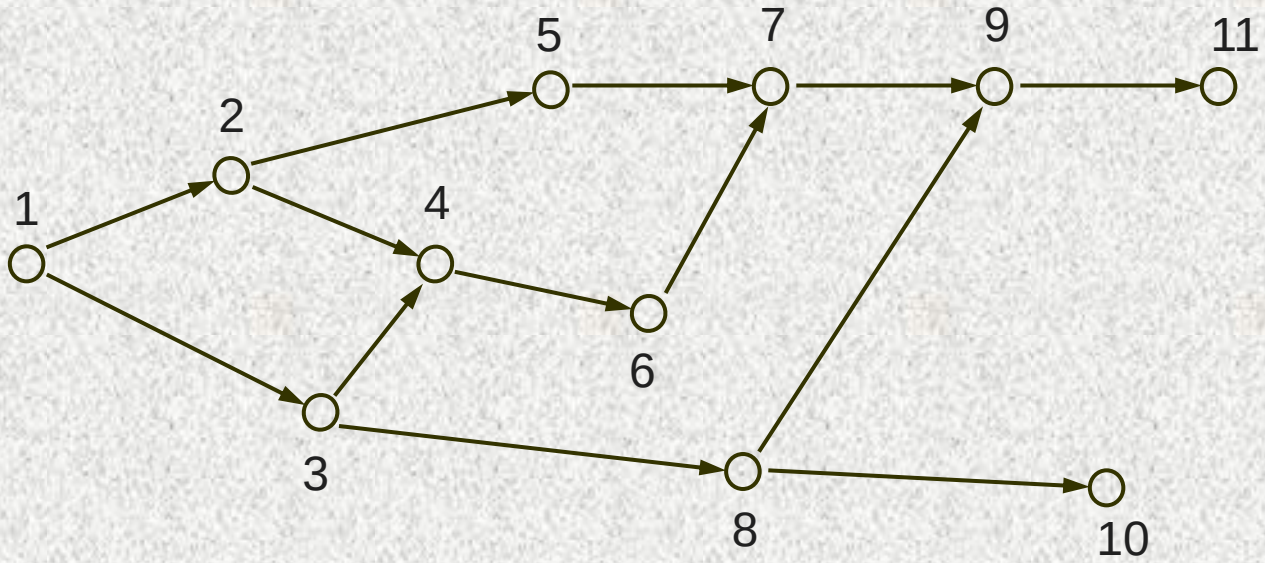
Как описать ресурс параллелизма программ и алгоритмов?

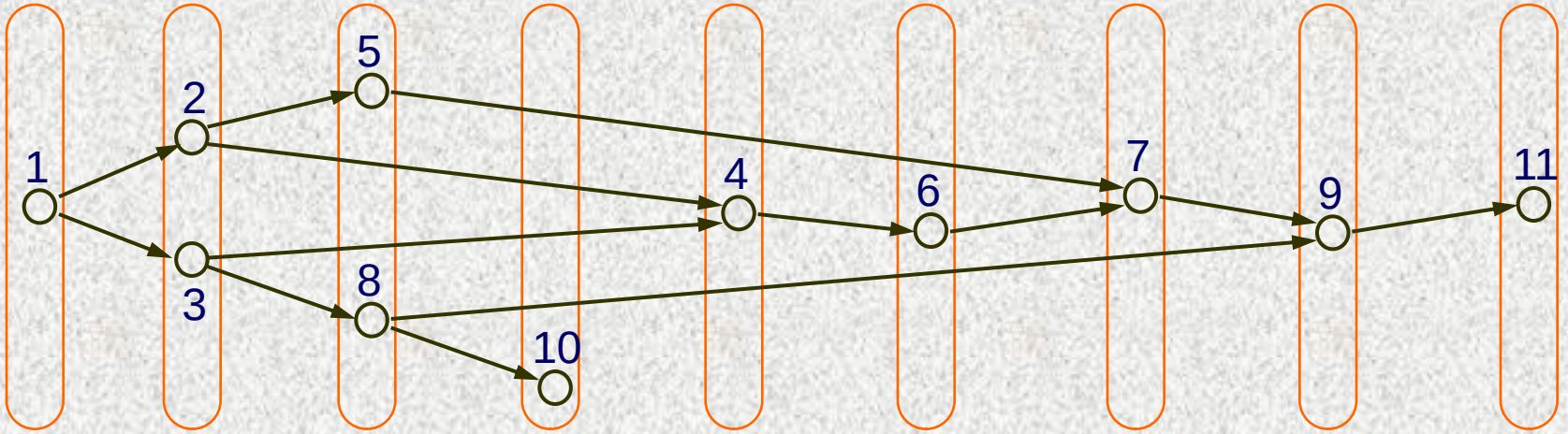
Ярусно-параллельная форма графа алгоритма



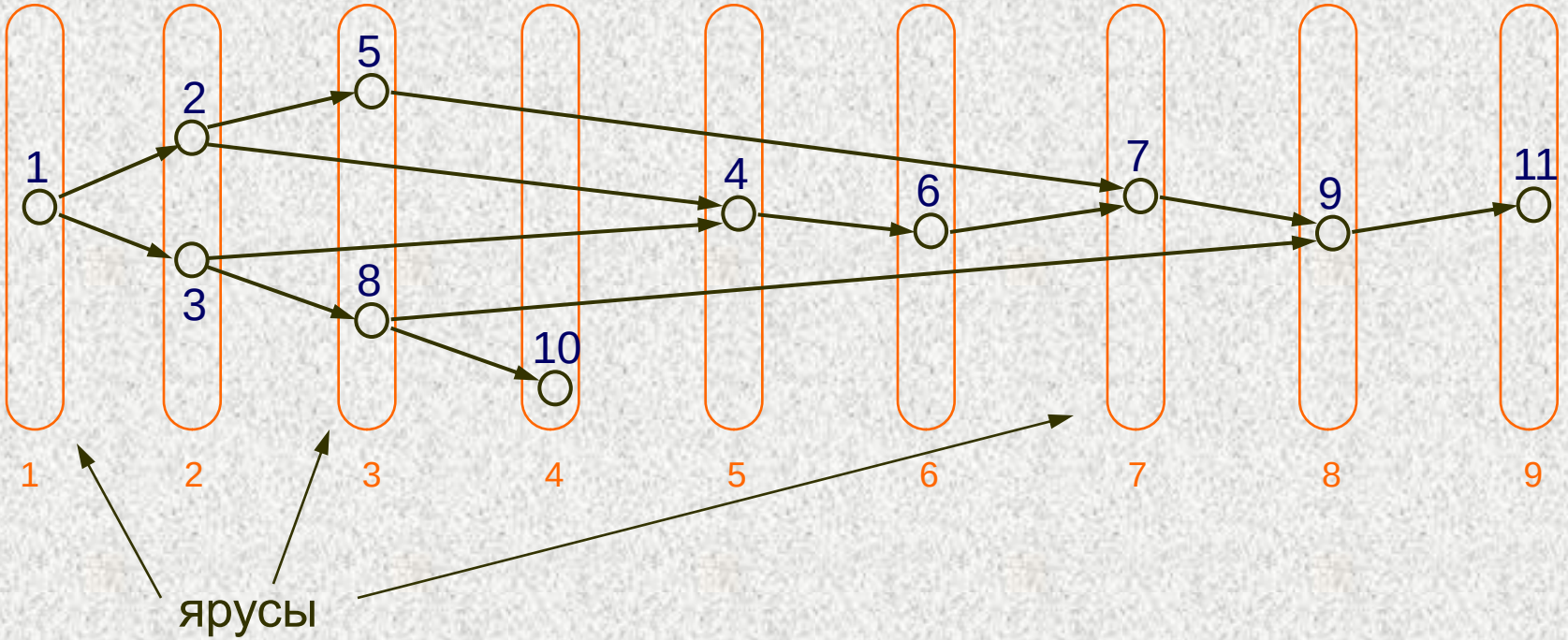
Как определить и сделать понятным ресурс параллелизма в графе алгоритма (в программе, в алгоритме) ?

Ярусно-параллельная форма графа алгоритма





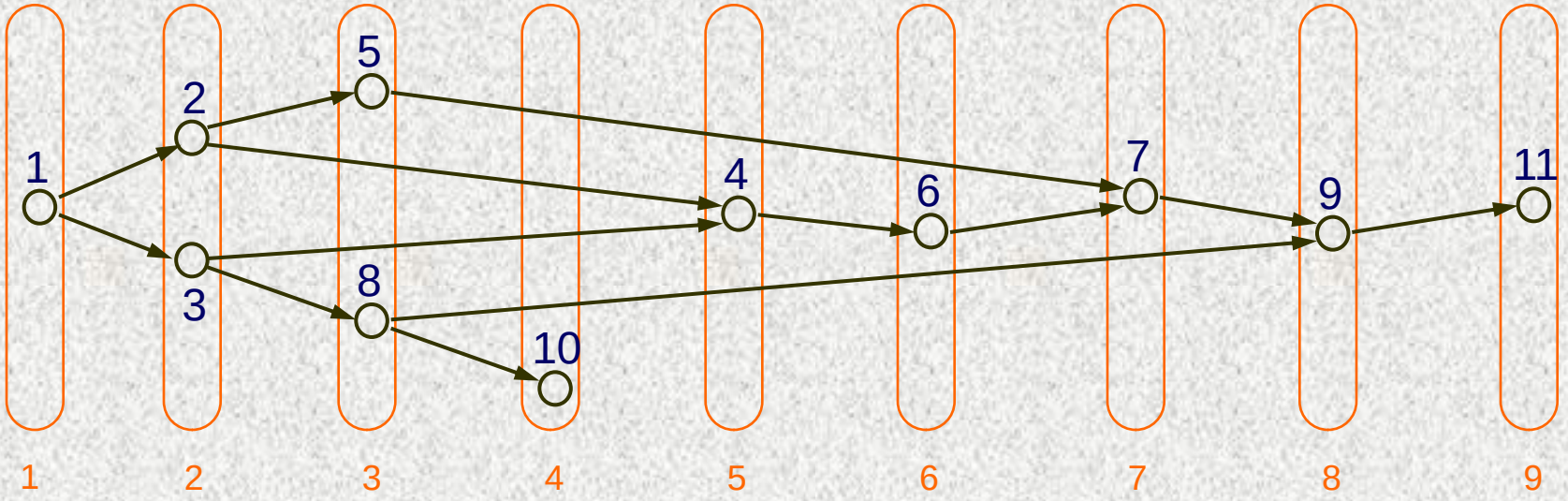
Ярусно-параллельная форма графа алгоритма



- начальная вершина каждой дуги расположена на ярусе с номером меньшим, чем номер яруса конечной вершины,

- между вершинами, расположенными на одном ярусе, не может быть дуг.

Ярусно-параллельная форма графа алгоритма



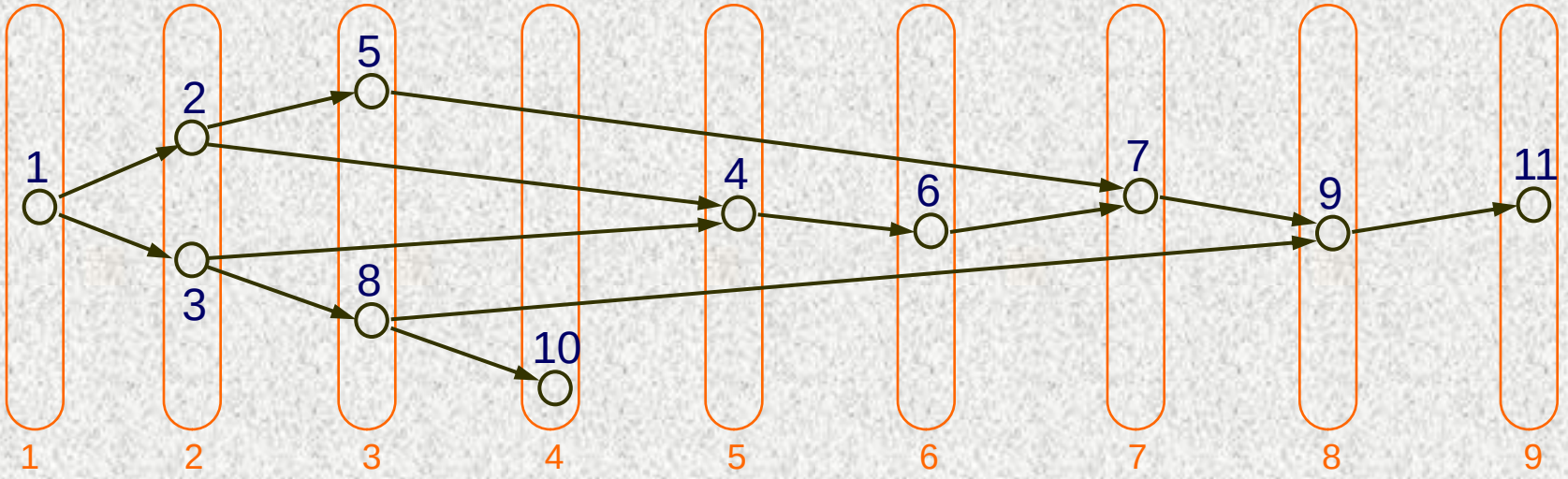
Высота ЯПФ – это число ярусов,

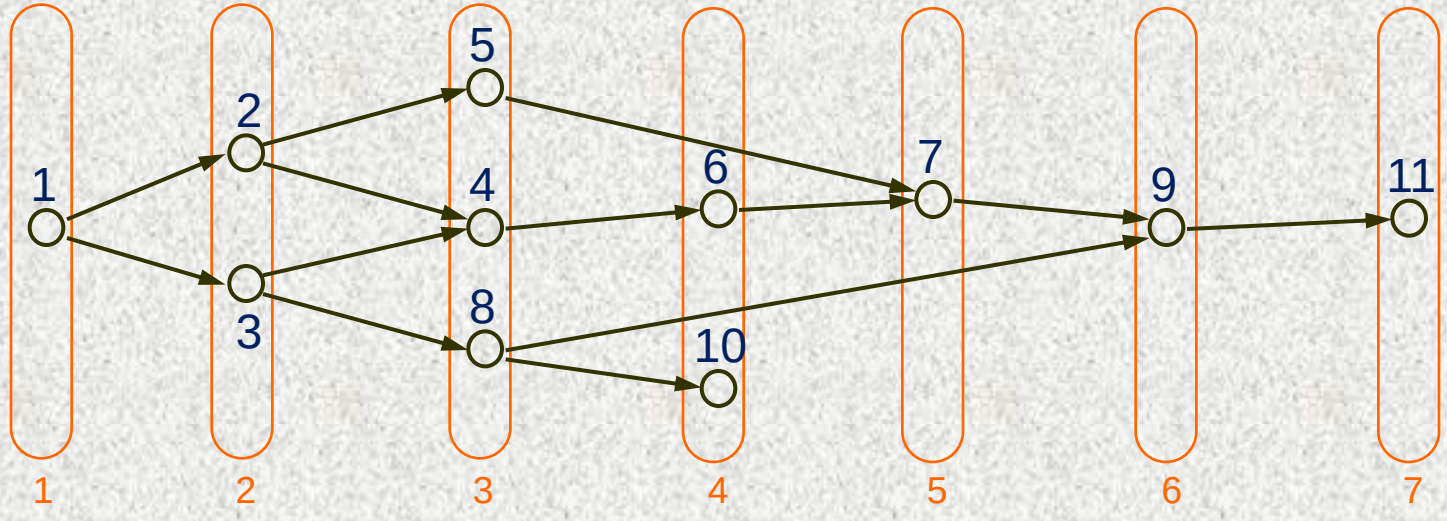
Ширина яруса – число вершин, расположенных на ярусе,

Ширина ЯПФ – это максимальная ширина ярусов в ЯПФ.

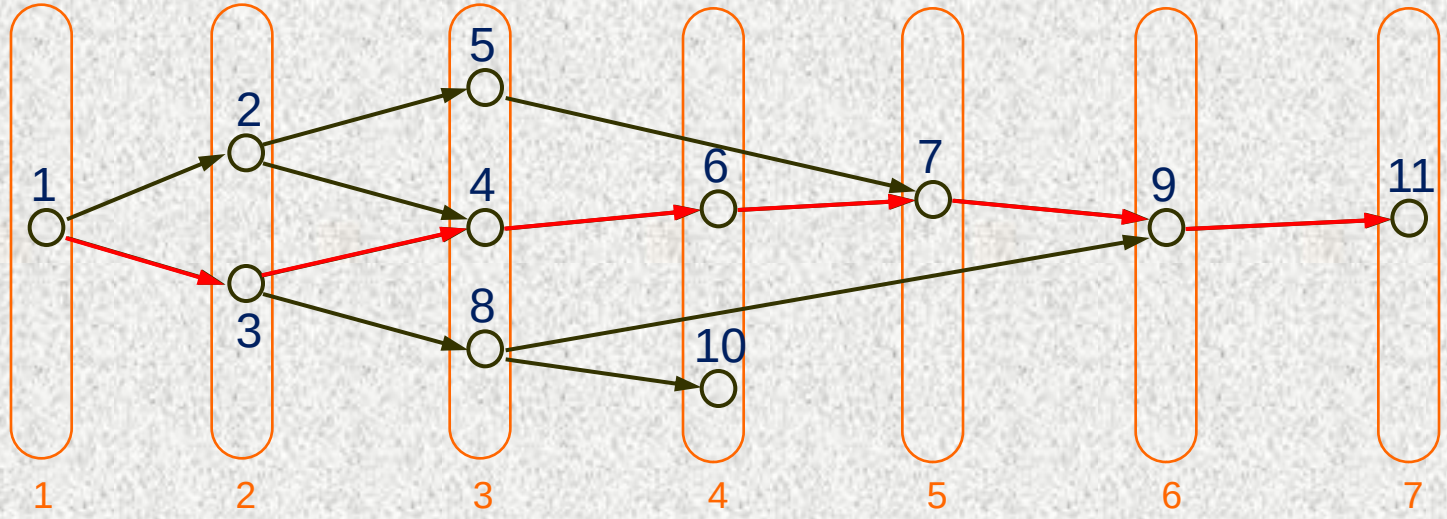
Высота ярусно-параллельной формы - это сложность параллельной реализации алгоритма/программы.

Ярусно-параллельная форма графа алгоритма определяется неоднозначно





Каноническая ярусно-параллельная форма графа алгоритма



Ярусно-параллельная форма называется канонической, если у любой вершины, кроме вершин первого яруса, есть входная дуга, идущая с предыдущего яруса.

Высота канонической ЯПФ = длине критического пути + 1.

Критический путь в ориентированном ациклическом графе – это путь максимальной длины.

Каноническая ЯПФ и степень параллелизма

(каков у фрагмента ресурс параллелизма?)

```
for( i = 0; i < n; ++i)
    for( j = 0; j < m; ++j)
        A[i][j] = A[i][j–1] + C[i][j]*x;
```

Чему, согласно закону Амдала, равно максимальное ускорение, которое можно получить при исполнении данного фрагмента на параллельной вычислительной системе?

Закон Амдала:

$$S \le \frac{1}{\alpha + \frac{(1-\alpha)}{p}}$$

где:

α *– доля последовательных операций,*

p – число процессоров в системе.

Каноническая ЯПФ и степень параллелизма

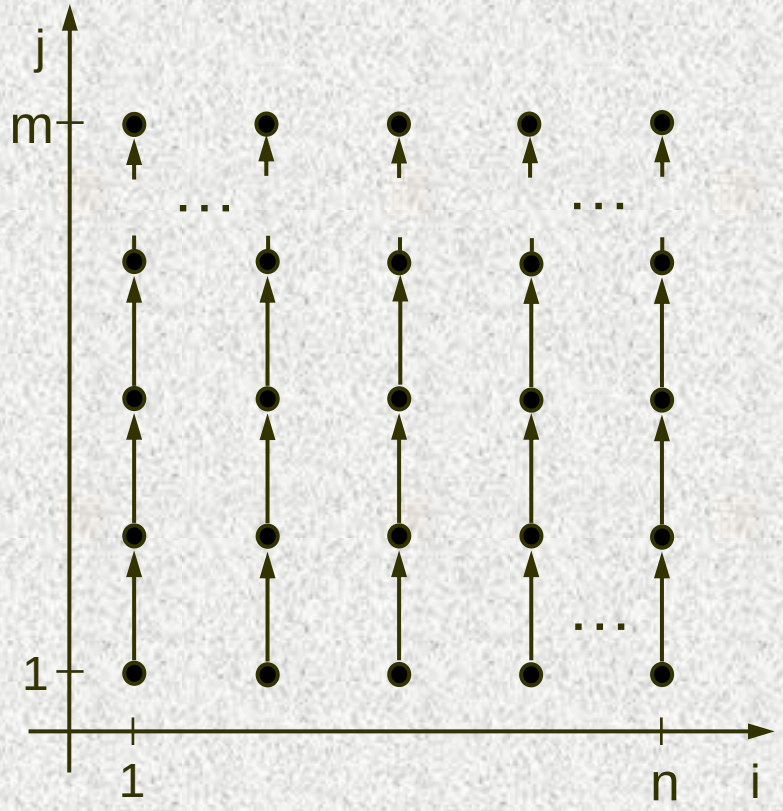
(каков у фрагмента ресурс параллелизма?)

```
for( i = 0; i < n; ++i)
    for( j = 0; j < m; ++j)
        A[i][j] = A[i][j–1] + C[i][j]*x;
```

$$S \approx \frac{1}{\alpha}$$

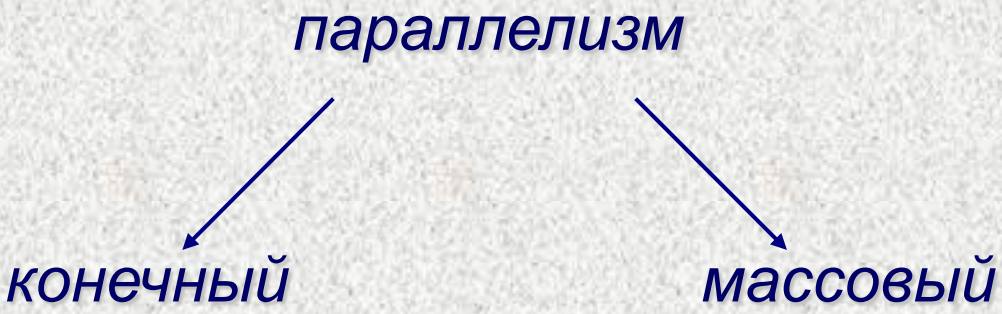
$$\alpha = \frac{\text{Число последовательных операций}}{\text{Общее число операций}} = \frac{m}{n*m} = \frac{1}{n}$$

$$S \approx n$$



Какого рода параллелизм встречается в программах?

Виды параллелизма в алгоритмах и программах



Конечный параллелизм определяется информационной независимостью некоторых фрагментов в тексте программы.

Массовый параллелизм определяется информационной независимостью итераций циклов программы.

Виды параллелизма в алгоритмах и программах

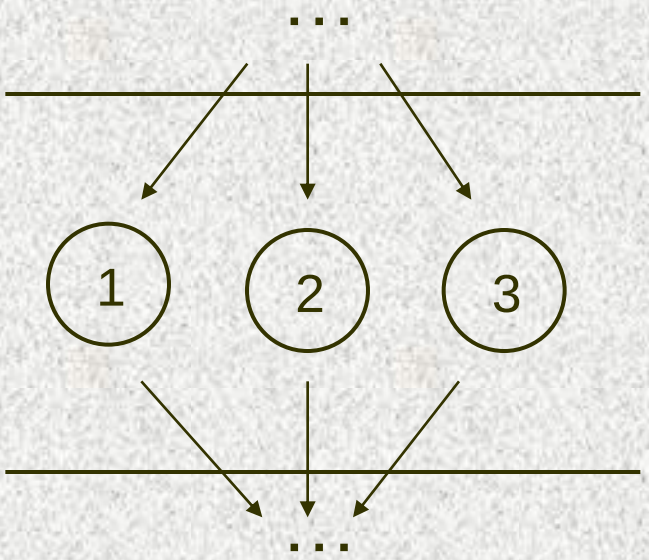
Конечный параллелизм.

```
cout << "N=" << N << endl;
cycleTestWithUnroll_KJI("k");
cycleTestWithUnroll_KJI("j");
cycleTestWithUnroll_KJI("i");
cycleTestWithUnroll_KJI_3("k");
cycleTestWithUnroll_KJI_3("j");
cycleTestWithUnroll_KJI_3("i");
cycleTest("j,i,k");
cycleTest("i,k,j");
cycleTest("k,j,i");
cycleTest("i,j,k");
cycleTest("k,i,j");
cycleTest("j,k,i");
float ***a12=new float**[N];
for (i=0;i<N;i++) {
    a12[i]=new float*[N];
    for (j=0;j<N;j++) {
        a12[i][j]=new float[N];
        for (k=0;k<N;k++) {
            a12[i][j][k]=(float)1/(i+j+k+1);
        }
    }
}
for (i=1;i<N;i++)
    for (j=1;j<N;j++)
        for (k=1;k<N;k++) {
            testee[i][k] = testee[i][k] + S[k]*A[k][j][i] + P[i][j]*A[k][j][i-1] +
                P[i][k]*A[k][j-1][i] + P[j][k]*A[k-1][j][i];
        }
…
```

1

2

3

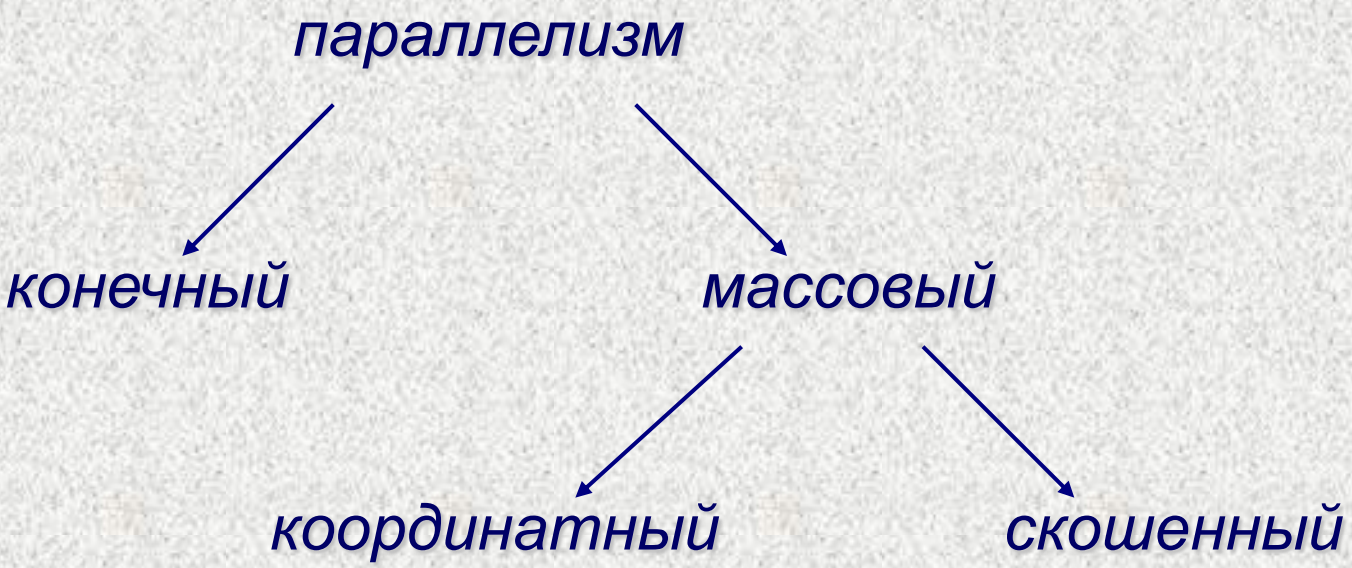


Виды параллелизма в алгоритмах и программах

Массовый параллелизм.

```
for( i = 0; i < n; ++i)
    A[i] = B[i] + C[i]*x;
```

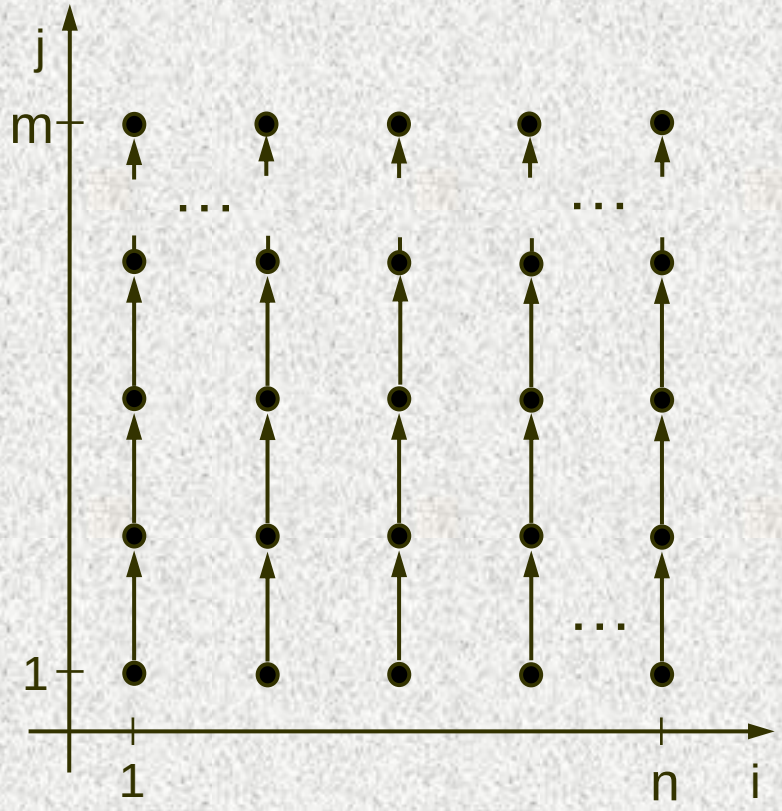
Виды параллелизма в алгоритмах и программах



Виды параллелизма в алгоритмах и программах

Координатный параллелизм.

```
#pragma omp parallel for
for( i = 0; i < n; ++i)
    for( j = 0; j < m; ++j)
        A[i][j] = A[i][j–1] + C[i][j]*x;
```



Виды параллелизма в алгоритмах и программах

Координатный параллелизм.

***Утверждение**: для того чтобы цикл был параллельным необходимо и достаточно, чтобы для любой тройки графа алгоритма данного цикла включение* $\Delta_i \subset G_i$ *было верным, где*

Δ_i *— это многогранник из тройки,*

G_i *= {*$f_1 = i_1$*,*

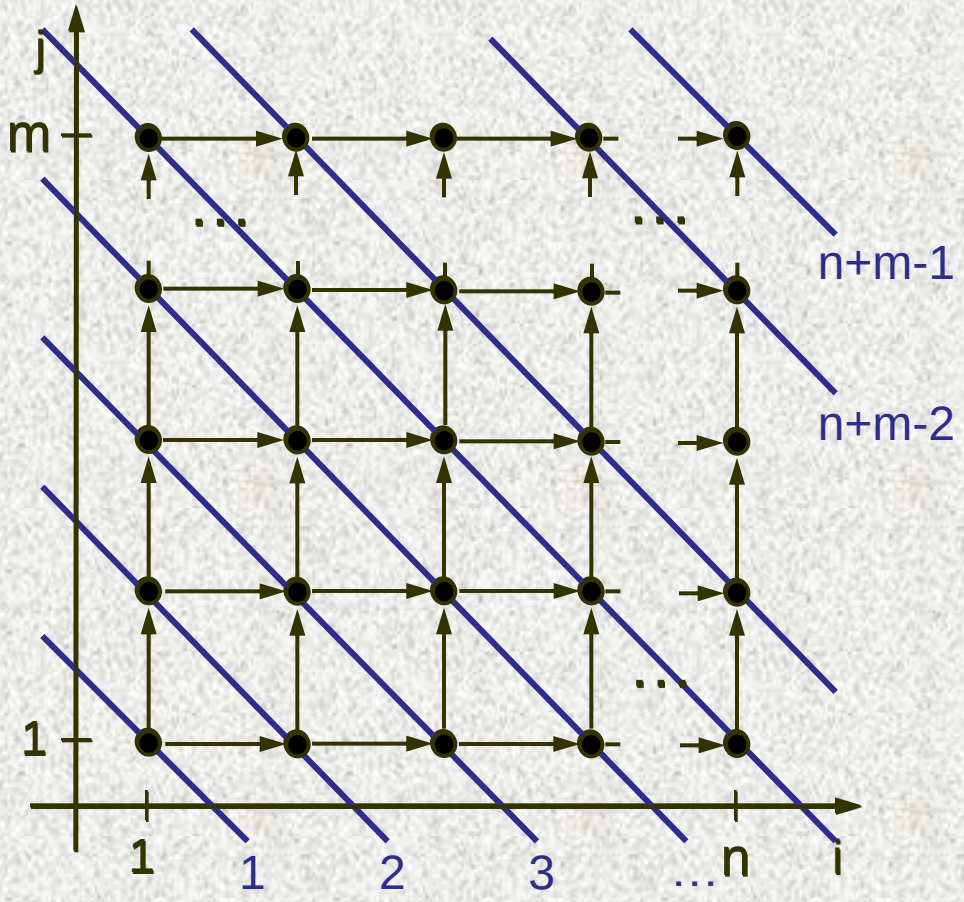
i_1 *— это параметр анализируемого цикла,*

f_1 *— это первая компонента векторной функции* F_i *из тройки.*

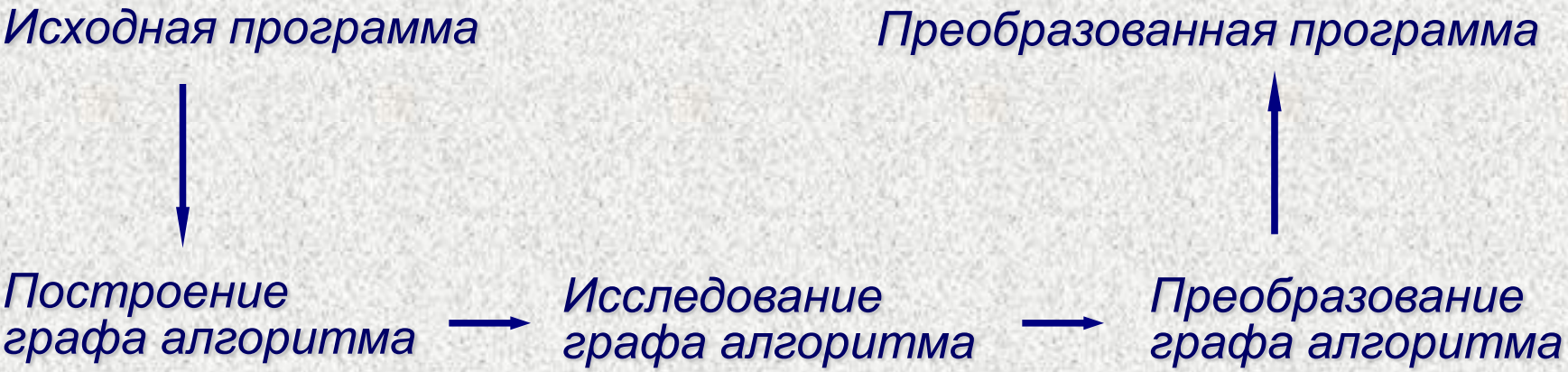
Виды параллелизма в алгоритмах и программах

Скошенный параллелизм.

```
for( i = 0; i < n; ++i)
    for( j = 0; j < m; ++j)
        A[i][j] = A[i][j–1] + A[i-1][j]*x;
```



Эквивалентные преобразования программ



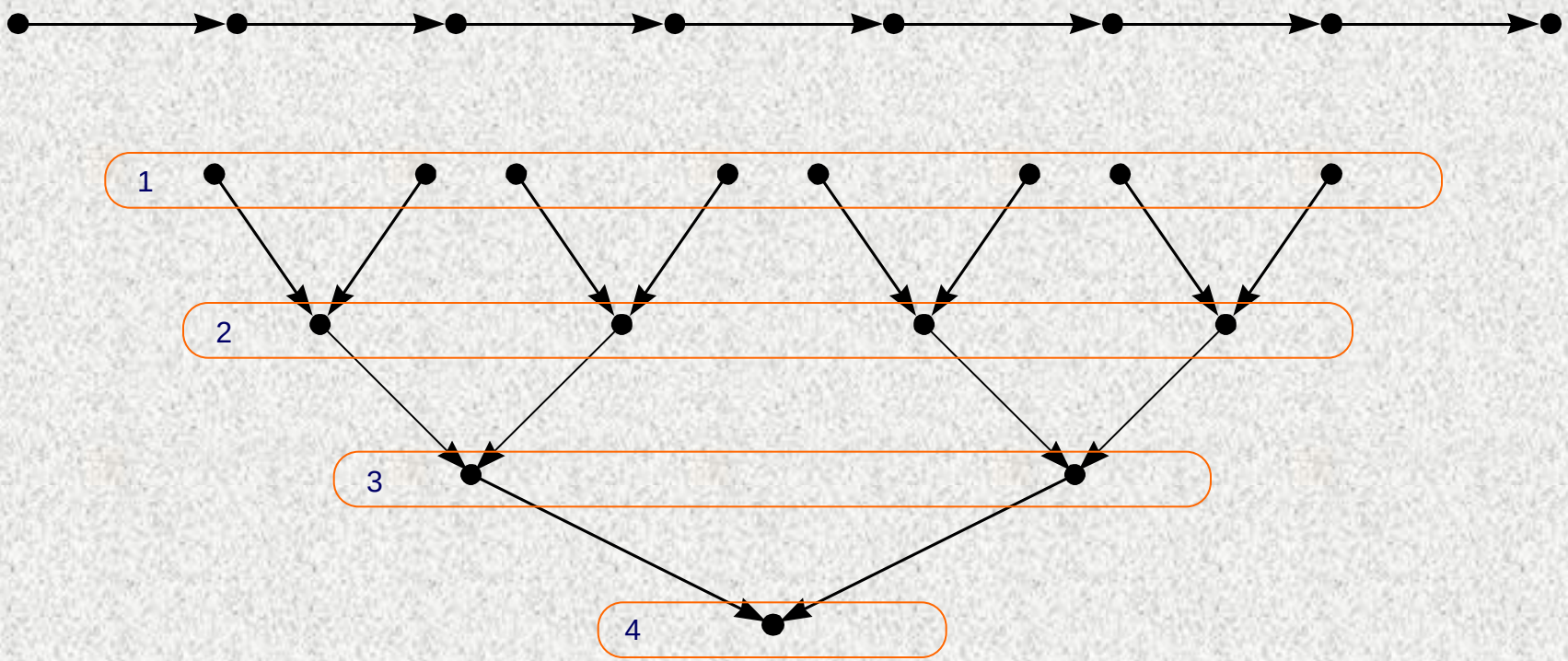
Эквивалентные преобразования программ

(суммирование элементов массива)

```
s = 0.0;
for ( i = 0; i < n; ++i )
    s = s + A[ i ];
```

Чисто последовательный алгоритм. Что делать? Не использовать суммирования на параллельных вычислительных системах? Невозможно…

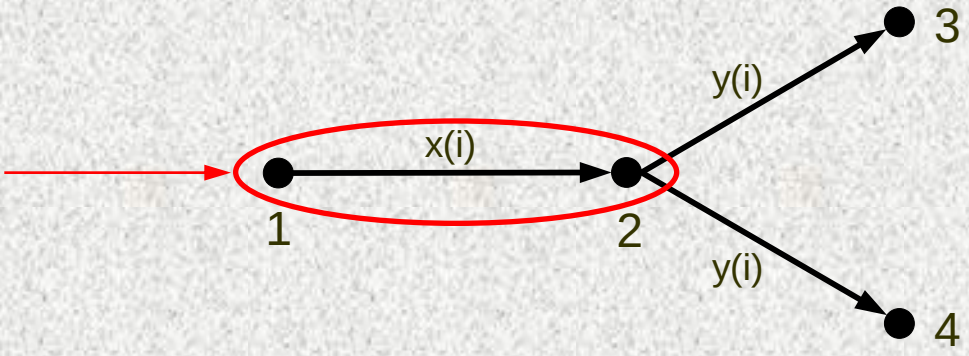
Эквивалентные преобразования программ
(суммирование элементов массива)



В данном случае это не есть эквивалентное преобразование! Использовано два разных метода с разной информационной структурой, разной параллельной сложностью, разными ошибками округления…

Эквивалентные преобразования программ

Исполнять только последовательно !



Информационная зависимость определяет критерий эквивалентности преобразований программ.

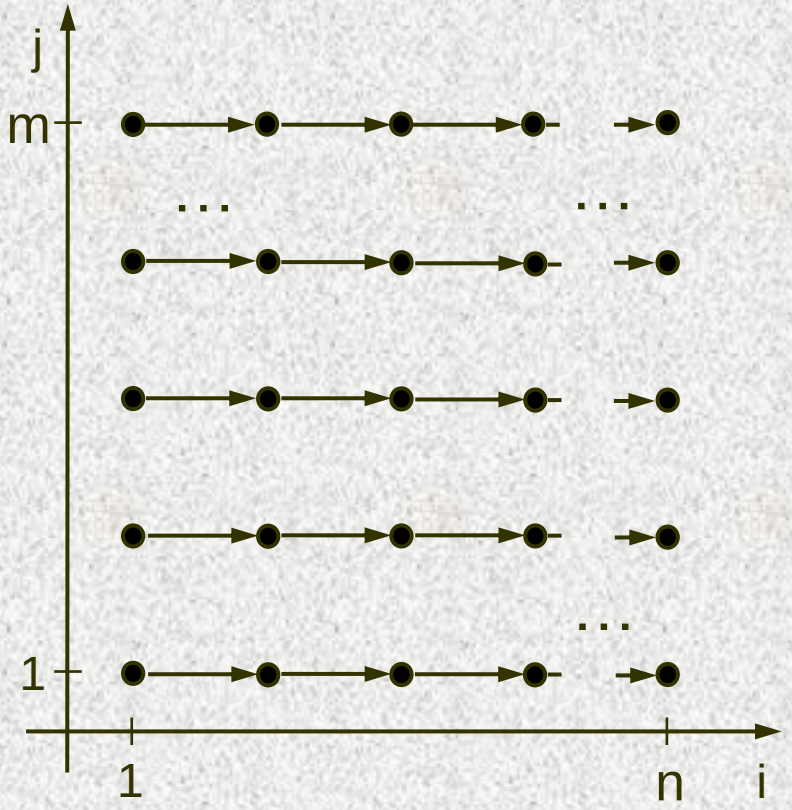
Информационная независимость определяет ресурс параллелизма программы.

Эквивалентные преобразования программ на практике…

Элементарные преобразования циклов

Перестановка циклов.

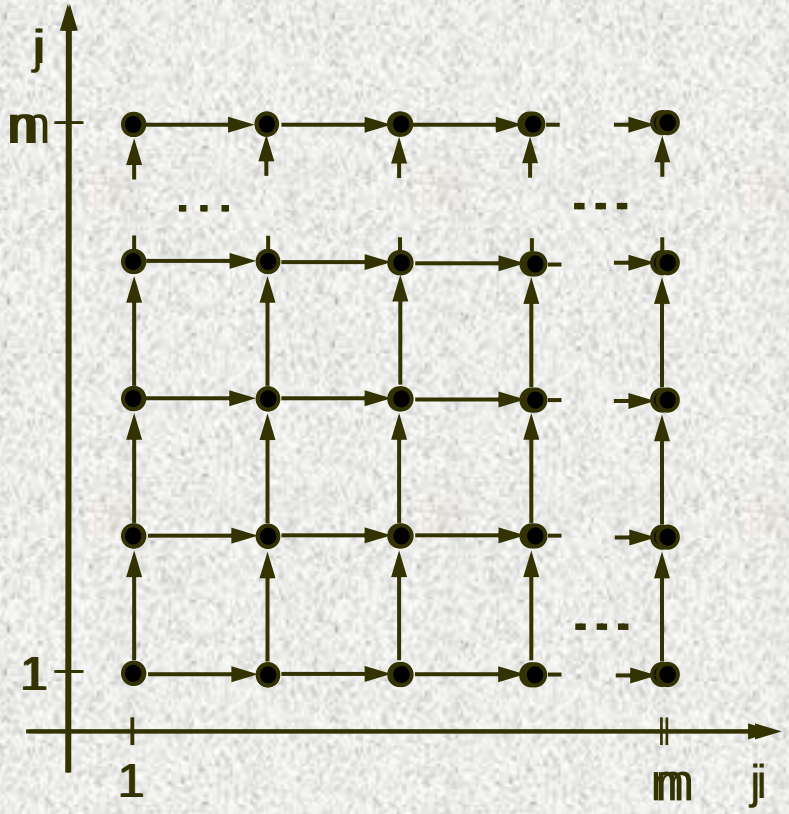
```
for( i = 0; i < n; ++i)
#pragma omp parallel for
    for( j = 0; j < m; ++j)
        A[i][j] = A[i-1][j] + C[i][j]*x;
```



Элементарные преобразования циклов

Перестановка циклов.

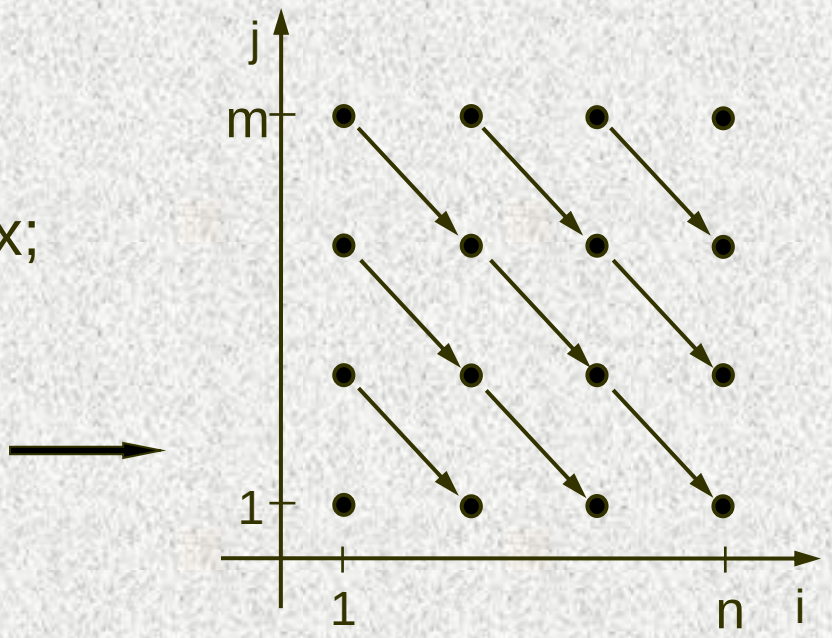
```
#pragma omp parallel for
for( i = 0; i < n; ++i)
    for( j = 0; j < m; ++j)
        A[i][j] = A[i–1][j] + C[i][j]*x;
```



Элементарные преобразования циклов

Всегда ли перестановка циклов является эквивалентным преобразованием?

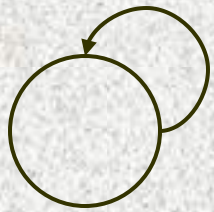
```
for( i = 0; i < n; ++i)
    for( j = 0; j < m; ++j)
        A[i][j] = A[i+c1][j+c2] + C[i][j]*x;

for( i = 0; i < n; ++i)
    for( j = 0; j < m; ++j)
        A[i][j] = A[i–1][j+1] + C[i][j]*x;
```



Элементарные преобразования циклов

Распределение циклов.

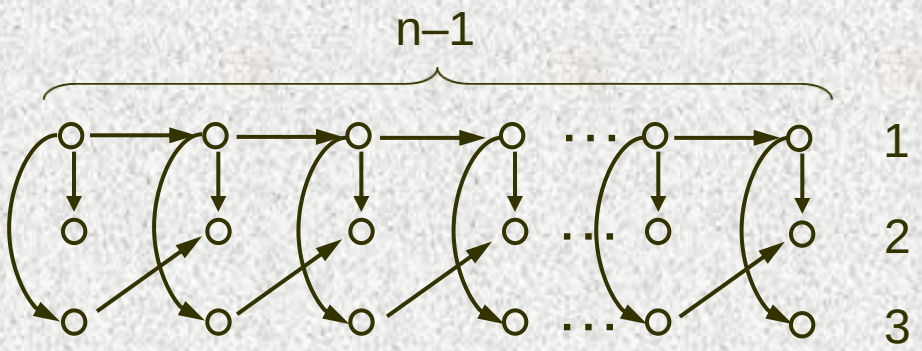
```
   for( i = 1; i < n; ++i) {
1      A[i] = A[i–1]*p + q;
2      C[i] = (A[i] + B[i–1])*s;
3      B[i] = (A[i] – B[i])*t;
   }
```



Элементарные преобразования циклов

Распределение циклов.

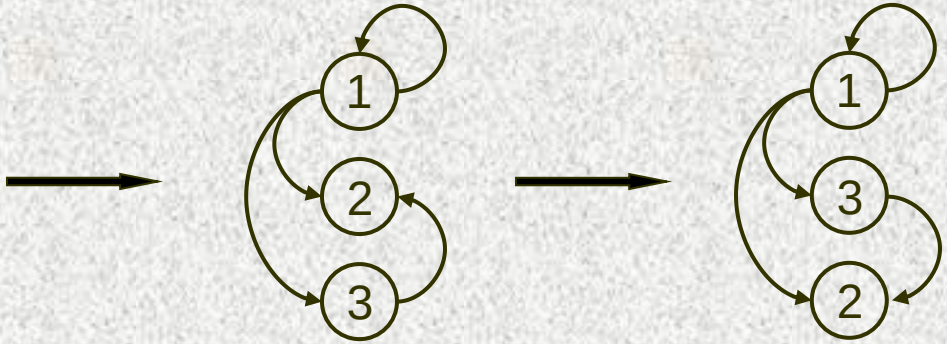
```
   for( i = 1; i < n; ++i) {
1      A[i] = A[i–1]*p + q;
2      C[i] = (A[i] + B[i–1])*s;
3      B[i] = (A[i] – B[i])*t;
   }
```



Элементарные преобразования циклов

Распределение циклов.

```
   for( i = 1; i < n; ++i) {
1      A[i] = A[i–1]*p + q;
2      C[i] = (A[i] + B[i–1])*s;
3      B[i] = (A[i] – B[i])*t;
   }
```



***Утверждение**: для того чтобы можно было выполнить распределение цикла необходимо и достаточно, чтобы распределяемые части находились в разных компонентах сильной связности информационного графа тела данного цикла.*

Элементарные преобразования циклов

Распределение циклов.

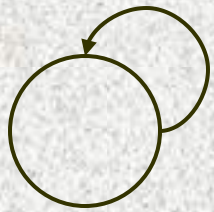
```
for( i = 1; i < n; ++i) {
    A[i] = A[i–1]*p + q;
    C[i] = (A[i] + B[i–1])*s;
    B[i] = (A[i] – B[i])*t;
}
```

```
for( i = 1; i < n; ++i)
    A[i] = A[i–1]*p + q;
#pragma omp parallel for
for( i = 1; i < n; ++i)
    B[i] = (A[i] – B[i])*t;
#pragma omp parallel for
for( i = 1; i < n; ++i)
    C[i] = (A[i] + B[i–1])*s;
```

Элементарные преобразования циклов

Расщепление циклов.

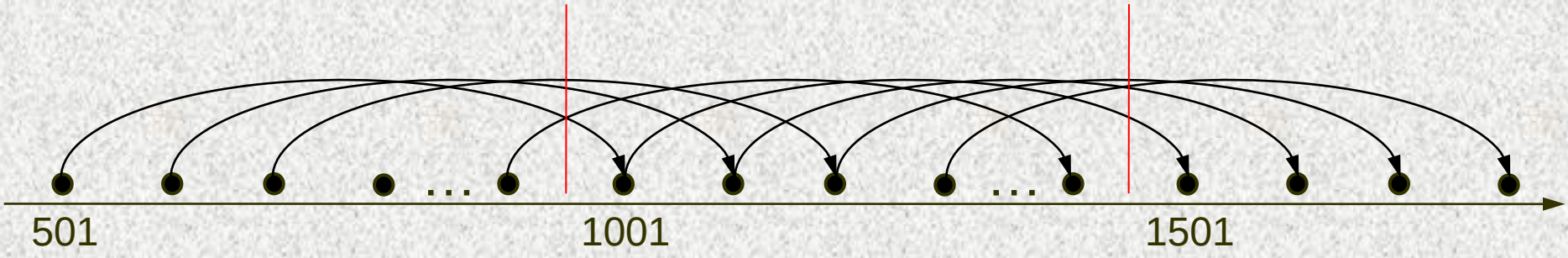
```
for( i = 501; i <= 2000; ++i)
    A[i] = A[i] + A[i–500];
```



Элементарные преобразования циклов

Расщепление циклов.

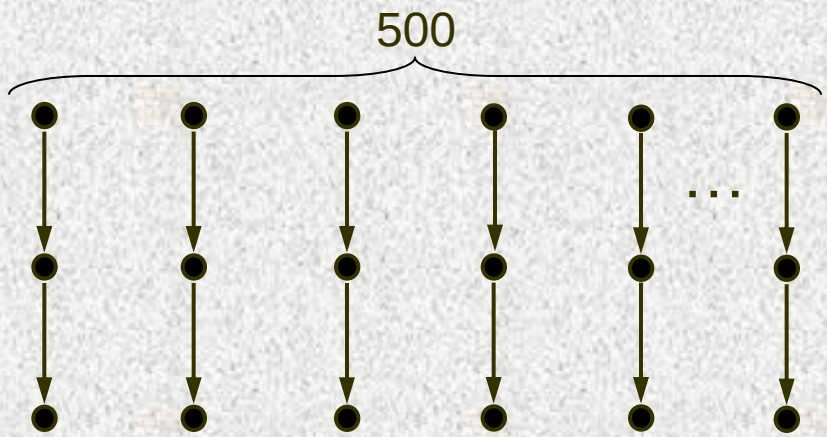
```
for( i = 501; i <= 2000; ++i)
    A[i] = A[i] + A[i–500];
```



Элементарные преобразования циклов

Расщепление циклов.

```
for( i = 501; i <= 2000; ++i)
   A[i] = A[i] + A[i–500];
```




```
#pragma omp parallel for
 for( i = 501; i <= 1000; ++i)
    A[i] = A[i] + A[i–500];
#pragma omp parallel for
 for( i = 1001; i <= 1500; ++i)
    A[i] = A[i] + A[i–500];
#pragma omp parallel for
 for( i = 1501; i <= 2000; ++i)
    A[i] = A[i] + A[i–500];
```

Эквивалентные преобразования программ

Эквивалентно ли преобразование?

```
for( i = 0; i < n; ++i)
    D[i] = D[i] * F[i];
if( m == 3 )
    for( i = 0; i < n; ++i)
        R[i] = P[i] + D[i];
else
     for( i = 0; i < n; ++i)
        R[i] = Q[i] – D[i];
```

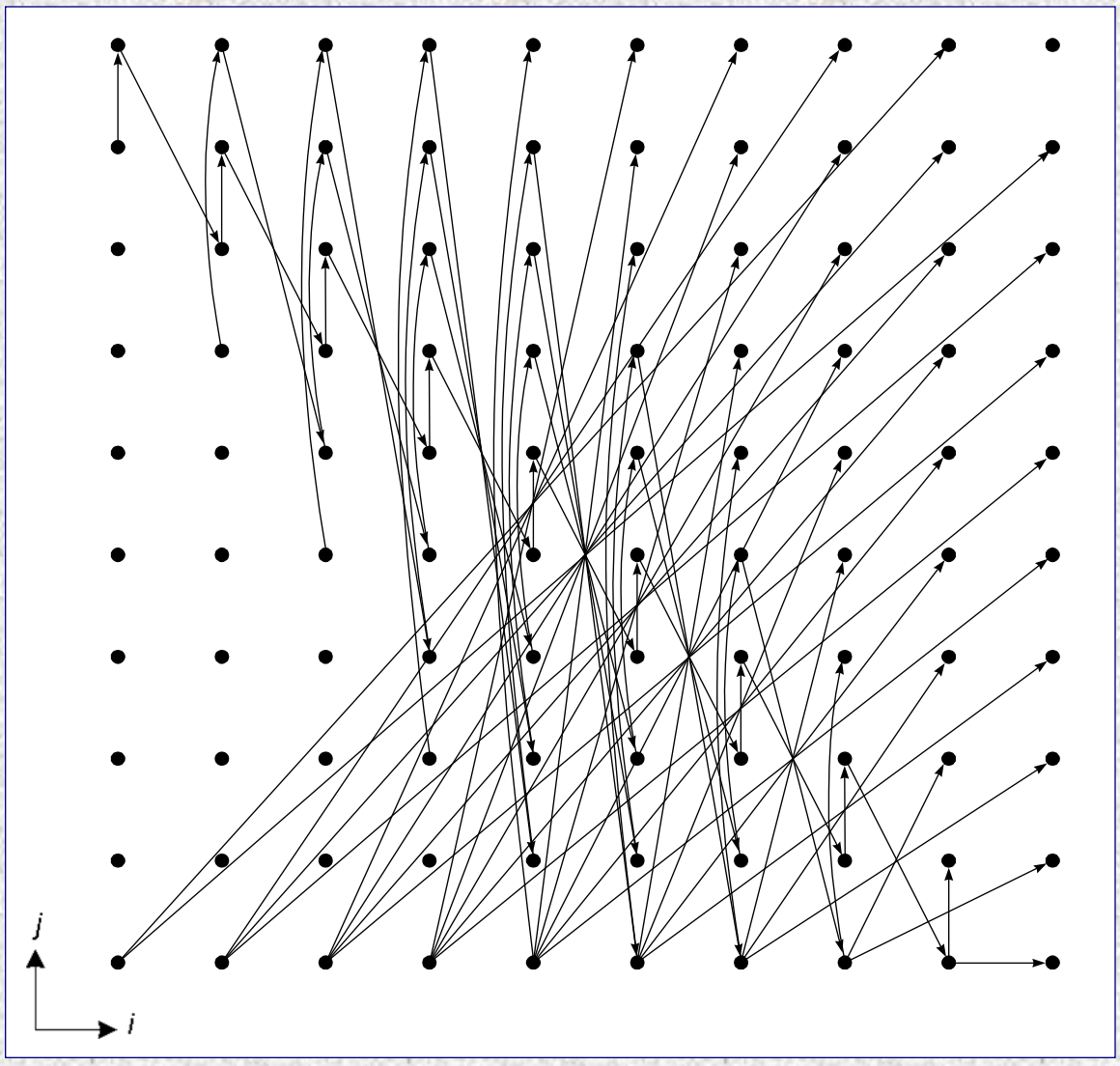
```
if( m == 3 )
    for( i = 0; i < n; ++i)
        R[i] = P[i] + D[i] * F[i];
else
     for( i = 0; i < n; ++i)
        R[i] = Q[i] – D[i] * F[i];
```

Означает ли малый размер программы простоту ее информационной структуры?

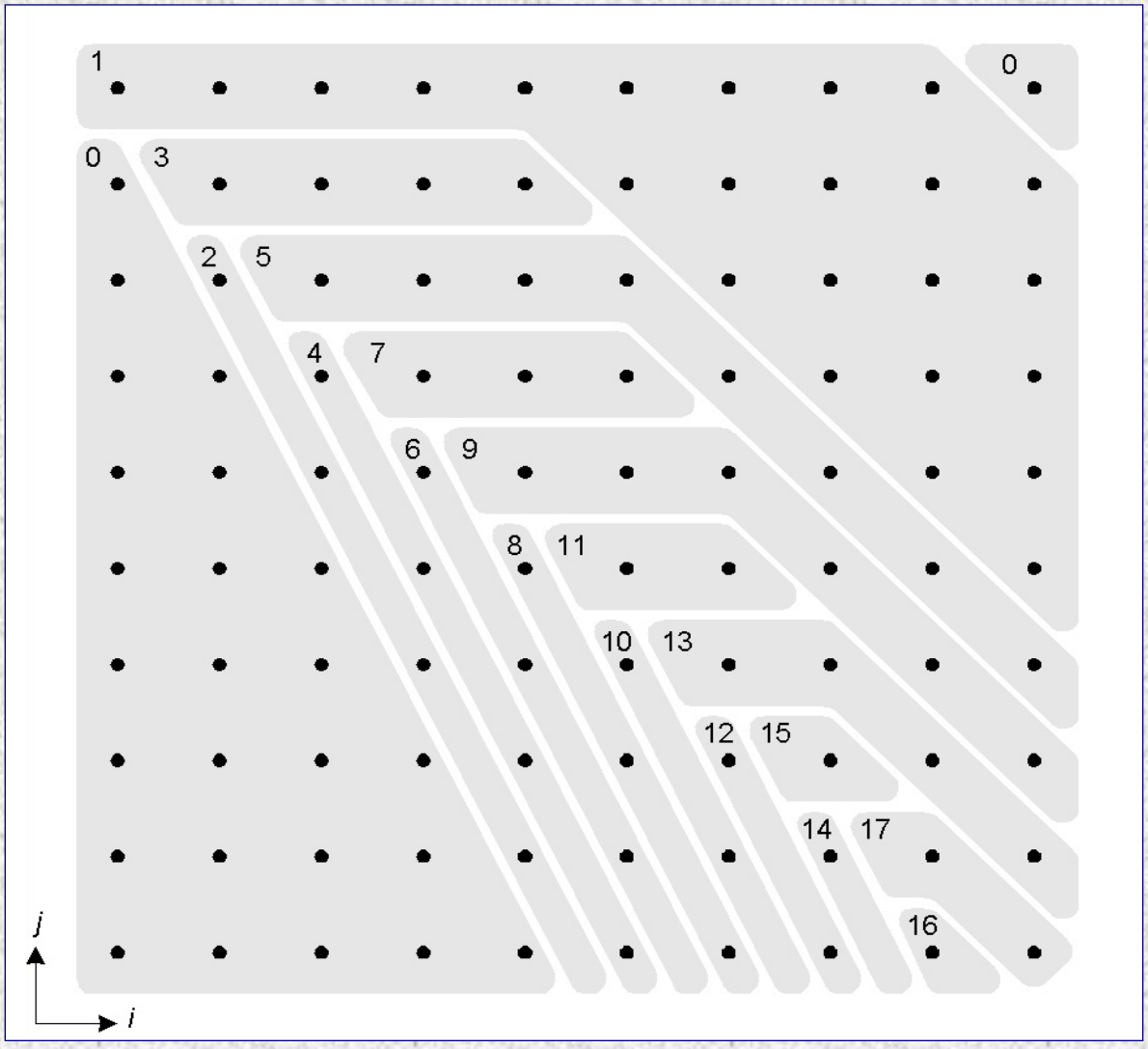
Простой пример…
(последовательный вариант)

```
DO  i = 1, n
    DO j = 1, n
        U( i + j ) = U( 2*n – i – j + 1)*q + p
    EndDO
EndDO
```

Информационная структура примера…



Ярусно-параллельная форма примера…



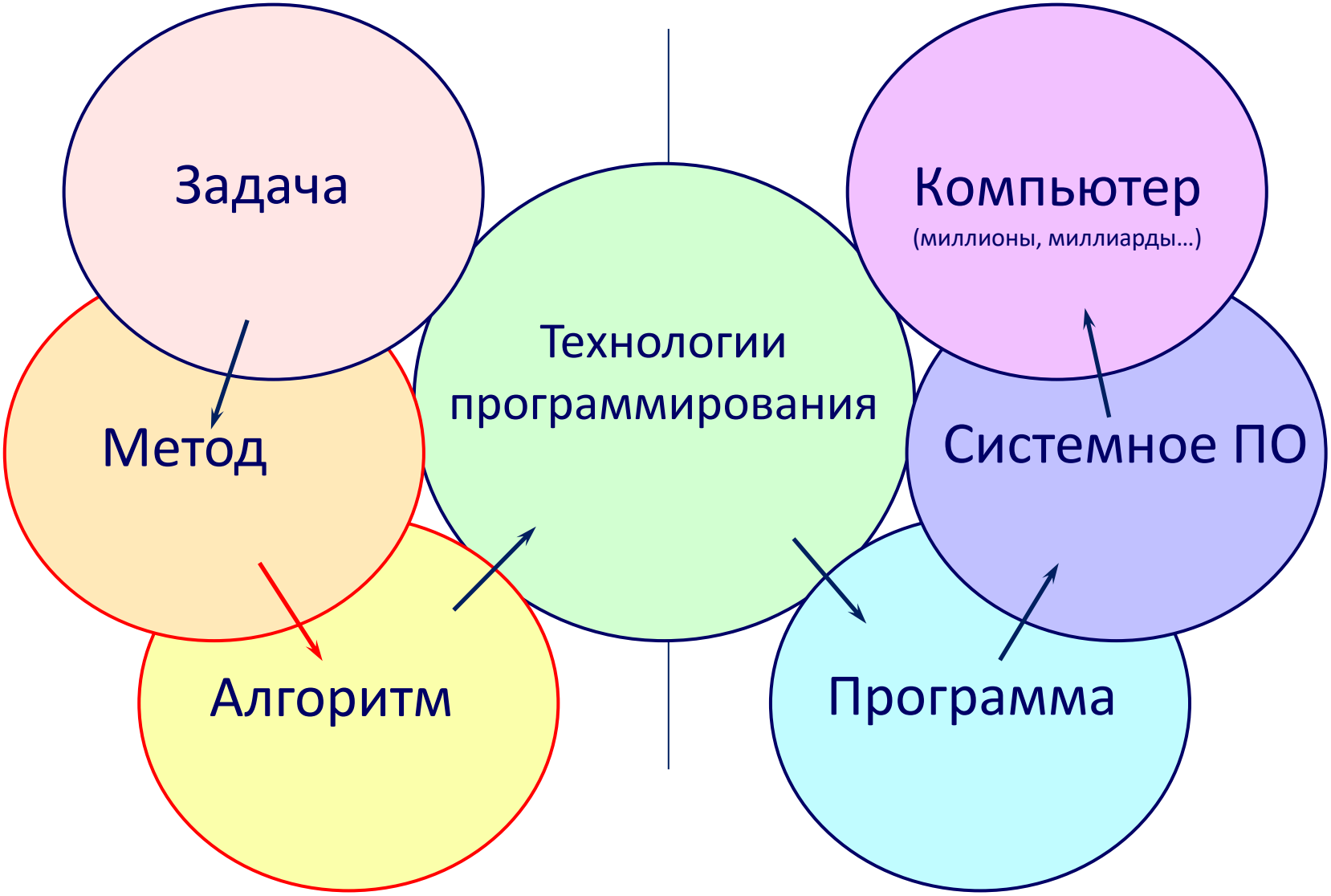
Совсем не простой пример…
(параллельный вариант)

```
DO  i = 1, n
    DO  j = 1, n – i        Параллельный цикл !
        U( i + j ) = U( 2*n – i – j + 1)*q + p
    End DO
    DO j = n – i + 1, n     Параллельный цикл !
        U( i + j ) = U( 2*n – i – j + 1)*q + p
    End DO
End DO
```

Вспомним прошлые лекции…

Почему, говоря о решении задач на параллельных компьютерах, мы разделяем этапы “Метод” и “Алгоритм” ?

Решение задачи на компьютере



Рассмотрим решение верхней треугольной системы методом Гаусса…

Решение СЛАУ: от метода к алгоритму
(информационная структура)

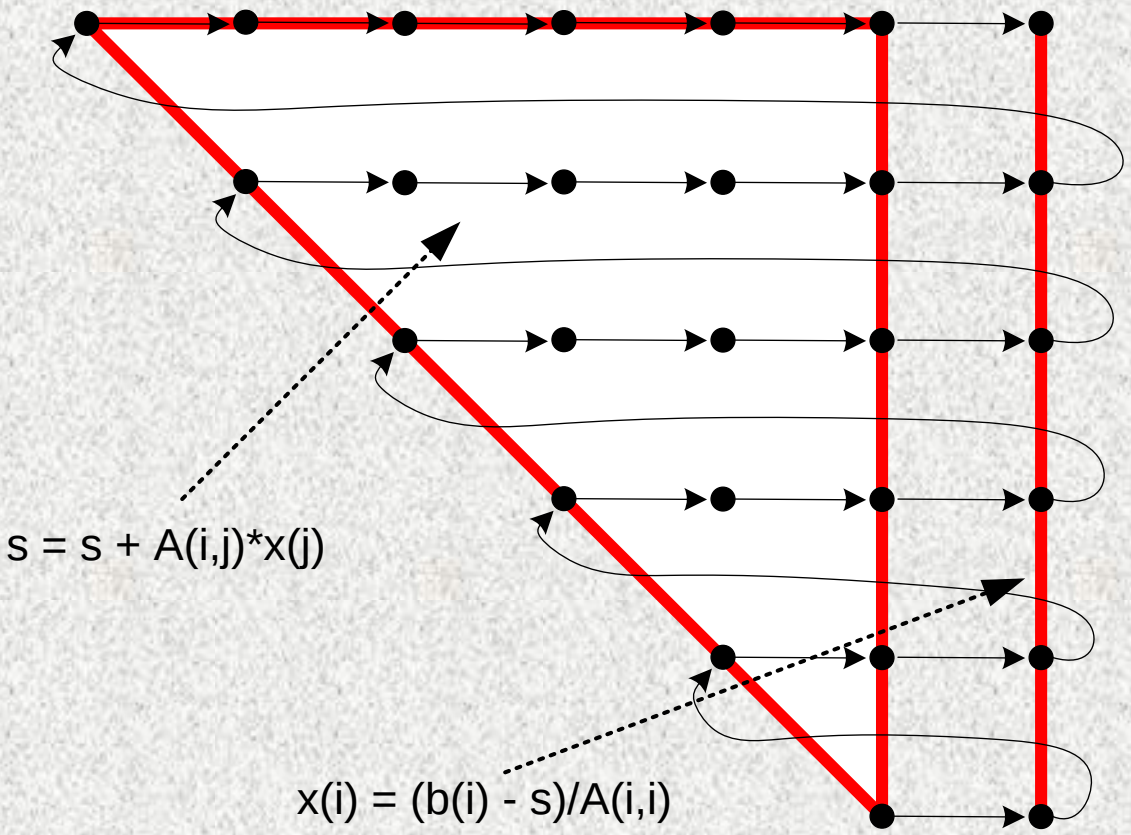


Схема программы:

```
do i = n, 1, -1
   s = 0
   do j = i+1, n
      s = s + A(i,j)*x(j)
   end do
   x(i) = (b(i) - s)/A(i,i)
end do
```

Критический путь графа алгоритма проходит через все вершины, следовательно такую программу нет смысла исполнять на параллельной вычислительной системе!

С точки зрения метода Гаусса
не имеет никакого значения,
в каком порядке выполнять итерации
внутреннего цикла по j (суммирование)

Изменим в программе только этот порядок
и определим информационную структуру.

Решение СЛАУ: от метода к алгоритму

(информационная структура)

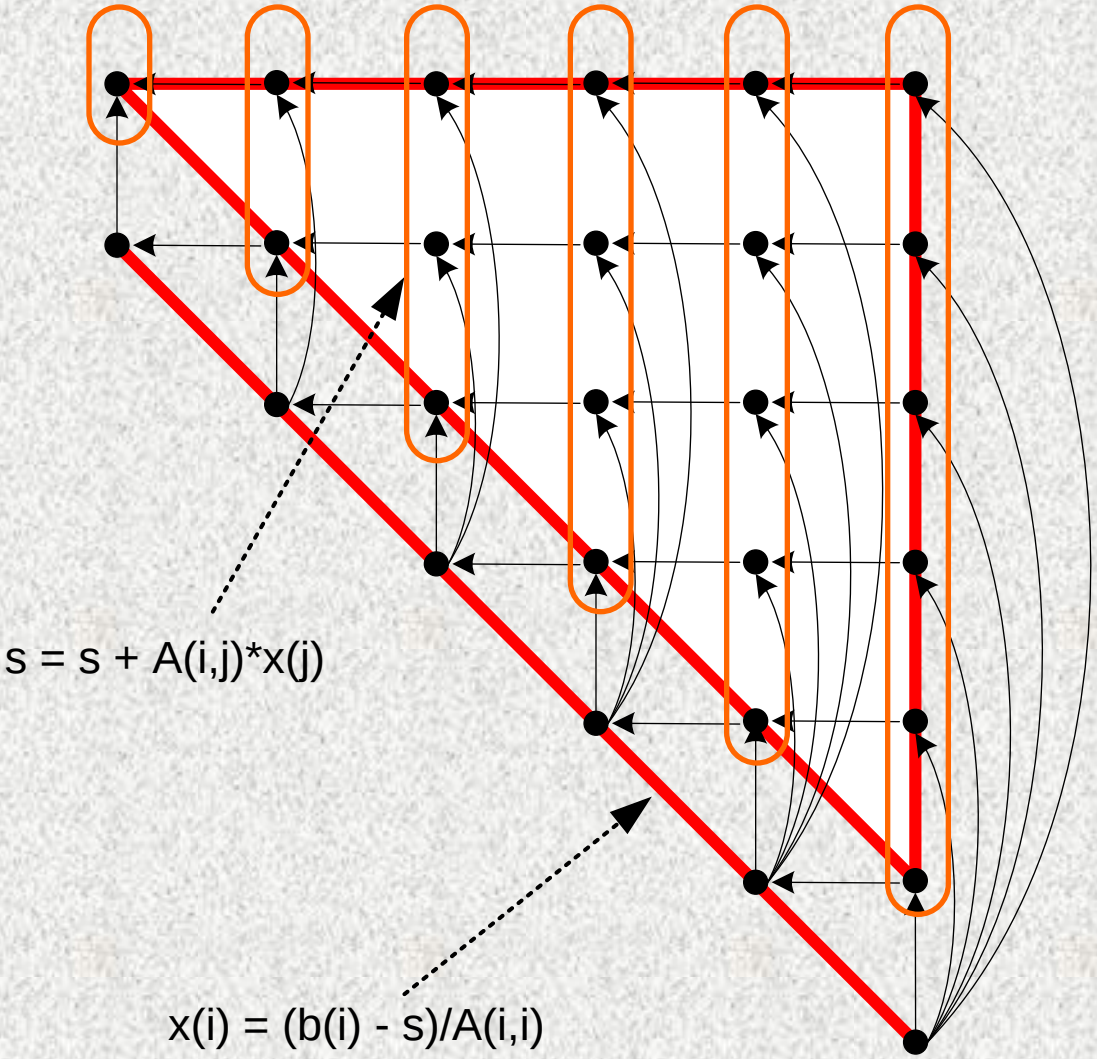


Схема программы:

```
do i = n, 1, -1
   s = 0
   do j = n, i+1, -1
      s = s + A(i,j)*x(j)
   end do
   x(i) = (b(i) - s)/A(i,i)
end do
```

Критический путь графа алгоритма имеет длину o(n), следовательно данная программа обладает хорошим ресурсом параллелизма!

Файл Правка Вид Журнал Закладки Инструменты Справка

Алговики

https://algowiki-project.org/ru/Открытая_энциклопедия_свойств_алгоритмов

Поиск

Создать учётную запись Войти

AlgoWiki

Main page Обсуждение Читать Просмотр История Поиск

Открытая энциклопедия свойств алгоритмов

Заглавная страница
Общий форум
Технический форум
Справка
Случайная статья
Свежие правки

Хранилище файлов
Новые файлы
Загрузить файл

Инструменты
Ссылки сюда
Связанные правки
Спецстраницы
Версия для печати
Постоянная ссылка
Сведения о странице

На других языках
English

Добро пожаловать! Присоединяйтесь!

AlgoWiki - это открытая энциклопедия по **свойствам алгоритмов и особенностям их реализации** на различных программно-аппаратных платформах от мобильных платформ до экзафлопсных суперкомпьютерных систем с возможностью коллективной работы всего мирового вычислительного сообщества.

Цель **AlgoWiki** - дать исчерпывающее описание алгоритма, которое поможет оценить его потенциал применительно к конкретной параллельной вычислительной платформе. Кроме классических свойств алгоритмов, например, последовательной сложности, в AlgoWiki представлены дополнительные сведения, составляющие в совокупности полную картину об алгоритме: параллельная сложность, параллельная структура, детерминированность, оценки локальности данных, эффективность и масштабируемость, коммуникационный профиль конкретных реализаций и многие другие.

Читать подробнее: О проекте

Структура проекта

Классификация алгоритмов - основной раздел AlgoWiki, содержащий описания всех алгоритмов. Алгоритмы добавляются в подходящий раздел классификации, при необходимости классификация расширяется за счет новых разделов.

Образцовая статья

Разложение Холецкого (метод квадратного корня)

1 Свойства и структура алгоритма

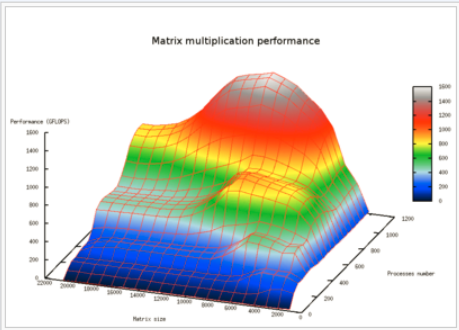
1.1 Общее описание алгоритма

Разложение Холецкого впервые предложено французским офицером и математиком Андре-Луи Холецким в конце Первой Мировой войны, незадолго до его гибели в бою в августе 1918 г. Идея этого разложения была опубликована в 1924 г. его сослуживцем. Потом оно было использовано поляком Т. Банашевичем

Свойства алгоритма:

- Последовательная сложность алгоритма: $O(n^3)$
- Высота ярусно-параллельной формы: $O(n)$
- Ширина ярусно-параллельной формы: $O(n^2)$
- Объём входных данных: $\frac{n(n+1)}{2}$
- Объём выходных данных: $\frac{n(n+1)}{2}$

Изображение дня



Производительность умножения плотных матриц

Организация работы

- Структура описания свойств алгоритмов
- Руководства по заполнению разделов описания
- Готовность статей
- Результаты прогона алгоритмов
- Глоссарий
- Помощь

Участники проекта

Руководители:

Предварительные объявления о зачете…

- *Список вопросов к зачету опубликован на https://parallel.ru/vmk20*

- *Консультация по курсу состоится 12 декабря, суббота, в 14 часов. Ссылку вышлем дополнительно.*

- *Итоги по конкурсу изображений, выступлениям в формате стенд-ап и выполнению практических заданий будут подведены к 12 декабря.*

- *Электронное тестирование пройдет 15-го декабря в 14:35, пароли будут высланы старостам групп.*

- *Устные зачеты (экзамены) – 24 и 28 декабря с 10:00, расписание по группам и ссылки будут высланы позже. Необходима включенная камера на стороне студентов и преподавателей.*

Стенд-ап (стендап, stand up) по теме курса “Суперкомпьютеры и параллельная обработка данных”

Заявлены три выступления в формате стенд-ап, 10 минут на каждое выступление:

1. Черкасов Владислав, 416 группа. Как суперкомпьютер справляется с приземлёнными задачами?

2. Раева Анастасия, 412 группа. Суперкомпьютеры и их применение.

3. Гринев Тимофей, 416 группа. Введение в теорию суперлюдей (пародия на теорию суперкомпьютеров)

Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ И
ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ

ВВЕДЕНИЕ В ТЕОРИЮ АНАЛИЗА СТРУКТУРЫ ПРОГРАММ И АЛГОРИТМОВ

(часть 2)

Вл.В.Воеводин

Зав.кафедрой Суперкомпьютеров и квантовой информатики ВМК МГУ
Директор НИВЦ МГУ,
д.ф.-м.н., профессор, чл.-корр. РАН

voevodin@parallel.ru

ВМК МГУ, 2020