Введение

Одной из явно прослеживаемых тенденций развития человечества является желание максимально строго моделировать процессы окружающей действительности с целью как улучшения условий жизни в настоящем, так и максимально достоверного предсказания будущего (со сходной конечной целью). Математические методы и приемы цифрового моделирования во многих случаях позволяют разрешать подобные проблемы, однако с течением времени имеет место серьёзное усложнение (как качественное, так и количественное) технологии решения задач. Во многих случаях ограничением является недостаток вычислительных мощностей современных ЭВМ; значимость решаемых задач привлекли огромные финансовые ресурсы в область создания сверхсложных ЭВМ, стоимость разработок которых достигает сотен миллионов долларов.

Однако с некоторых пор повышение быстродействия компьютеров традиционной (именуемой "*фон Неймановской*" и фактически обобщающей результаты исследований английских коллег-союзников и разработок *Конрада фон Цузе*) архитектуры стало чрезмерно дорого вследствие технологических ограничений при производства процессоров, поэтому разработчики обратили внимание на иной путь повышения производительности – комплексирование ЭВМ в *многомашинные комплексы и многопроцессорные системы;* при этом отдельные фрагменты программы параллельно (и одновременно) выполняются на различных процессорах, обмениваясь при этом информацией посредством внутренней *компьютерной сети*.

Идея комплексирования ЭВМ с целью повышения как производительности так и надежности почти так же стара как компьютерный мир (документированные объединения ЭВМ известны с конца 50-х г.г.).

Требования получить максимум производительности при минимальной стоимости привели к разработке многопроцессорных вычислительных комплексов (напр., в форме *вычислительных кластеров* на базе недорогих компонентов персональных ЭВМ); известны системы такого рода, объединяющие вычислительные мощности тысяч отдельных процессоров. Следующим этапом являются попытки объединить миллионы разнородных компьютеров планеты в единый вычислительный комплекс с огромной производительностью (технология *распределённых вычислений, метакомпьютинга*) посредством сети InterNet. На сегодняшний день применение *параллельных вычислительных систем* (*ПВС*) является стратегическим направлением развития вычислительной техники. Развитие "железа" с необходимостью подкрепляются совершенствованием алгоритмической и программной компонент - *технологий параллельного программирования* (*ТПП*).

Хотя идея распараллеливания вычислений отнюдь не нова, организация совместного функционирования множества независимых процессоров требу-

ет проведения серьёзных теоретико-практических исследований, без которых сложная и относительно дорогостоящая *многопроцессорные вычислительные система (МВС)* часто не только не превосходит, а уступает по производительности традиционному компьютеру.

Потенциальная возможность распараллеливания неодинакова для вычислительных задач различного типа – она значительна для научных программ, содержащих много циклов и длительных вычислений и существенно меньше для инженерных задач, для которых характерен расчёт по эмпирическим формулам.

Выделенные курсивом текстовые последовательности фактически являются ключевыми словами для поиска их значений в InterNet, использование явно заданных InterNet-ссылок неминуемо приведёт *настойчивого читателя* к первоисточнику информации. В *сносках* к конкретной странице приведена дополнительная литература, полезная при изучении материала.

1 Общие вопросы решения "больших задач"

Под термином "большие задачи" обычно понимают проблемы, решение которых требует не только построения сложных математических моделей, но и проведения огромного, на многие порядки превышающие характерные для ПЭВМ, количества вычислений (с соответствующими ресурсами ЭВМ – размерами оперативной и внешней памяти, быстродействием линий передачи информации и др.).

Заметим, что (практический) верхний предел количества вычислений для "больших задач" определяется (при одинаковом алгоритме, эффективности и ст**о**йкости к потере точности которого при вычислениях) лишь производительностью существующих на данный момент вычислительных систем. При "прогонке" вычислительных задач в реальных условиях ставится не вопрос "решить задачу вообще", а "решить за приемлемое время" (часы/десятки часов vs месяцы/годы).

1.1 Современные задачи науки и техники, требующие для решения суперкомпьютерных мощностей

Нижеприведены некоторые области человеческой деятельности, где возникают задачи подобного рода, часто именуемые проблемами класса ***GRAND CHALLENGES*** (имеется в виду *вызов окружающему миру*) - фундаментальные научные или инженерные задачи с широкой областью применения, эффективное решение которых возможно исключительно с использованием мощных (суперкомпьютерных, неизбежно использующих технологии многомашинных комплексов и многопроцессорных систем и параллельных вычислений) вычислительных ресурсов [1]:

- Предсказания погоды, климата и глобальных изменений в атмосфере
- Науки о материалах
- Построение полупроводниковых приборов
- Сверхпроводимость
- Структурная биология
- Разработка фармацевтических препаратов
- Генетика человека
- Квантовая хромодинамика
- Астрономия
- Транспортные задачи большой размерности
- Гидро- и газодинамика
- Управляемый термоядерный синтез
- Эффективность систем сгорания топлива

- Разведка нефти и газа
- Вычислительные задачи наук о мировом океане
- Распознавание и синтез речи, распознавание изображений

Одна из серьезнёйших задач – моделирование климатической системы и предсказание погоды. При этом совместно численно решаются уравнения динамики сплошной среды и уравнения равновесной термодинамики. Для моделирования развития атмосферных процессов на протяжении 100 лет и числе элементов дискретизации $2{,}6\times10^{6}$ (сетка с шагом 1^{O} по широте и долготе по всей поверхности Планеты при 20 слоях по высоте, состояние каждого элемента описывается 10 компонентами) в любой момент времени состояние земной атмосферы описывается $2{,}6\times10^{7}$ числами. При шаге дискретизации по времени 10 мин за моделируемый промежуток времени необходимо определить $\approx5\times10^{4}$ ансамблей (т.е. 10^{14} необходимых числовых значений *промежуточных вычислений*). При оценке числа необходимых для получения каждого промежуточного результата вычислительных операций в $10^{2}\div10^{3}$ общее число необходимых для проведения численного эксперимента с глобальной моделью атмосферы вычислений с плавающей точкой доходит до $10^{16}\div10^{17}$. Суперкомпьютер с производительностью 10^{12} оп/сек при идеальном случае (полная загруженность и эффективная алгоритмизация) будет выполнять такой эксперимент в течение нескольких часов; для проведения полного процесса моделирования необходима многократная (десятки/сотни раз) прогонка программы [1].

Классическим примером *большой программы* в этой области может служить программа "Ядерная зима" для отечественной БЭСМ-6, в которой моделировался климатический эффект ядерной войны (акад. *Н.Н.Моисеев* и *В.А Александров*, ВЦ Академии наук); проведённые с помощью этой программы исследования способствовали заключению соглашений об ограничении стратегических вооружений ОСВ-1 (1972) и ОСВ-2 (1979). Известный Earth Simulator (Япония, см. ниже) интенсивно применялся при моделировании поведения "озоновой дыры" над Антарктидой (появление которой считалось следствием выброса в атмосферу хлорфторуглеродных соединений). При подготовке *Киотского протокола* (соглашение о снижении промышленных выбросов парниковых газов в целях противодействию катастрофического потепления климата Планеты) также широко использовалось моделирование с помощью супер-ЭВМ.

Огромные вычислительные ресурсы требуются при моделировании ядерных взрывов, оконтуривании месторождений, обтекания летательных и подводных аппаратов, молекулярных и генетических исследованиях и др.

Проблема супервычислений столь важна, что современной мировой тенденцией является жёсткое курирование работ в области суперкомпьютерных технологий государством: пример - объединяющая три крупнейшие национальные лаборатории (Лос-Аламос, Ливермор, Sandia) американская правительственная программа "Ускоренная стратегическая компьютерная инициатива" (ASCI, *Accelerated Strategic Computing Initiative*, http://www.llnl.gov/asci), программы NPACI (*National Partnership for Advanced Computational Infrastructure*, http://www.npaci.edu), CASC (*Coalition of Academic Supercomputing Centers*, http://www.osc.edu/casc.html) и др. Государственная поддержка прямо связана с тем, что независимость в области производства и использования вычислительной техники отвечает интересам национальной безопасности, а научный потенциал страны непосредственно связан и в большой мере определяется уровнем развития вычислительной техники (а для данного случая это всегда многомашинные комплексы и многопроцессорные системы) и соответствующего математического обеспечения.

Так, в рамках принятой в США ещё в 1995 г. программы ASCI ставилась задача увеличения производительности супер-ЭВМ в 3 раза каждые 18 месяцев и достижение уровня производительности в 100 триллионов (100×10^{12}) операций с плавающей точкой в секунду (100 Терафлопс) к 2004 г. (для сравнения – наиболее мощные "персоналки" имеют производительность не более $1\div10$ Гигафлопс).

С целью объективности при сравнении производительность супер-ЭВМ рассчитывается на основе выполнения заранее известной тестовой задачи ("бенчмарка", от англ. *benchmark*); в качестве последней обычно используется тест LINPACK (http://www.netlib.org/utk/people/JackDongarra/faq-linpack.html), предполагающий решение плотнозаполненных систем линейных алгебраических уравнений (СЛАУ) большой размерности прямым методом (метод исключения Гаусса, другой метод не допускается) с числами с плавающей точкой двойной точности (число выполняемых операций при этом априори известно и равно $2\times n^3/3+2\times n^2$, где n – порядок матрицы, обычно $n\geq1000$; такой тест выбран вследствие того, что большое количество практических задач сводится именно к решению СЛАУ большой ($n\approx10^4\div10^6$) размерности. Для параллельных машин используется параллельная версия LINPACK – пакет HPL (*High Performance Computing Linpack Benchmark*, http://www.netlib.org/benchmark/hpl).

LINPACK-производительность вычисляется как $R_{max}=2\times\frac{n^3/3+n^2}{t}$, где t – время выполнения теста (время выполнения сложений и умножений принимается одинаковым). Пиковое (максимальное) быстродействие в общем случае определяется в виде $R_{peak}=p\Big/\sum_{i=1}^{i=k}\gamma_i t_i$, где p — число процессоров или

АЛУ; k — число различных команд в списке команд ЭВМ, γ_i - удельный вес команд i-го типа в программе, t_i — время выполнения команды типа i; веса команд определяются на основе сбора статистики по частотам команд в реальных программах (известны стандартные смеси команд Гибсона, Флинна и др.), обычно $R_{max} = (0{,}5 \div 0{,}95) \times R_{peak}$. Т.о. пиковая производительность определяется максимальным числом операций, которое может быть выполнено за единичное время при отсутствии связей между функциональными устройствами, характеризует *потенциальные возможности аппаратуры* и (вообще говоря) *не зависит от выполняемой программы*.

Именно на основе LINPACK-теста регулярно составляются мировой список наиболее быстродействующих вычислительных систем "Top-500" (http://www.top500.org) в мире и внутри российский список "Top-50" (http://www.supercomputers.ru); список наиболее энергоэффективных систем Green-500 (http://www.green500.org). Развиваются и новые системы тестов - напр., набор тестов NPB (*NAS Parallel Benchmarks*, http://www.nas.nasa.gov/NAS/NPB), рейтинг Graph500 (http://www.graph500.org) предполагает классификацию по быстродействию на задачах обработки сверхбольших объёмов информации, представленных в виде графа или базы данных.

Недостатком метода оценки пиковой производительности как числа выполняемых компьютером команд в единицу времени (MIPS, *Million Instruction Per Second*) дает только самое общее представление о быстродействии, т.к. не учитывает специфику конкретных программ (априори трудно предсказуемо, в *какое число* и *каких именно* инструкций процессора отобразится пользовательская программа).

Введённая в строй еще в 1999 г. в Sandia National Laboratories многопроцессорная система ASCI Red (Intel Paragon, США) имеет предельную (пиковую) производительность 3,2 триллионов операций в секунду (3,2 Терафлопс), включает 9’632 микропроцессора Pentium Pro, общий объем оперативной памяти 500 Гбайт и оценивается в сумму около 50 млн. $US. Список "Top-500" некоторое время возглавлял созданный для моделирования климатических изменений на основе полученных со спутников данных многопроцессорный комплекс Earth Simulator (NEC Vector, Япония), состоящий из 640 узлов (каждый узел включает 8 микропроцессоров SX-6 производительностью 8 Гфлопс каждый), общий объем оперативной памяти 8 Тбайт, суммарная пиковая производительностью 40 Тфлоп/сек (занимает площадь размерами 65×50 м в специально построенном двухъярусном здании с системами антисейсмики, кондиционирования воздуха и защиты от электромагнитных излучений). 70-ти терафлопный Blue Gene/L заказан Минэнергетики США и установлен в специализирующейся на ядерных проблемах Ливерморской лаборатории. Отечественная система МВС-15000ВМ (установлена в МСЦ -

Межведомственном суперкомпьютерном центре РАН, http://www.jscc.ru) представляет собой кластер из 462 серверов IBM, каждый из которых включает два процессора PowerPC 970 с частотой 2,2 ГГц и 4 Гб оперативной памяти, производительность МВС-15000ВМ в тесте LINPACK равна 5,4 Тфлопс при пиковой производительности в 8,1 Тфлопс на 56 месте в "Top-500" (июнь 2005 г.).

Китай в течение 11-й китайской пятилетки (2006÷2010 г.г.) ввёл в строй суперкомпьютер производительностью не менее 1 Петафлопс (1 Пфлопс=10^{15} "плавающих" операций в секунду), однако Япония приблизительно к этому сроку планировала построить суперкомпьютер на 10 Пфлопс (японцев легко понять – в условиях повышенной сейсмической активности крайне важно уметь предсказывать как сами землетрясения, так и их последствия – напр., цунами).

Вычислительные мощности отечественных супер-ЭВМ (список "Top-50") с 2002 г. возглавлял комплекс МВС-1000М МСЦ (768 процессоров Alpha 21264A 667 MHz, пиковая производительность 1 Тфлопс); в середине 2005 г. он на четвертом месте. На третьем - кластер ANT (НИВЦ МГУ, http://parallel.ru/cluster/ant-config.html), на втором – кластерная система СКИФ К-1000 Объединённого института информационных проблем, Беларусь (http://www.skif.bas-net.by), на первом – система МВС-15000 МСЦ РАН (производительность по LINPACK равна 3,1 Тфлопс, пиковая 4,9 Тфлопс).

Согласно 42-й редакции списка Top500 от ноября 2013 г. на 1-м месте суперкомпьютер Tianhe-2 (он же MilkyWay-2, Китай, Национальный Суперкомпьютерный Центр Гуанчжоу, 3,12 млн ядер; производительность 33,8/54,9 Pflops (max/peak), энергопотребление 17,8 MW), на втором Titan (USA, Oak Ridge National Laboratory, 0,507 млн ядер, 16,6/27,1 Pflops, 8,2 MW) и на третьем Sequoia (USA, 1,6 млн ядер, 17,2/20,1 Pflops; 7,9 MW). Суперкомпьютеры России: 37-е место – Ломоносов (МГУ, 78’660 ядер, 0,9/1,6 Pflops, 2,8 MW), 84-е место – MVS-10P (Объединённый суперкомпьютерный центр РАН, 28’704 ядра, 0,38/0,53 Pflops, 0,22 MW), 127-место – RSC Tornado SUSU (Южно-Уральский университет, 28’032 ядра, 0,29/0,47 Pflops, 0,29 MW).

Все составляющие списки "Top-500" и "Top-50" вычислительные системы являются многомашинными комплексами и/или многопроцессорными системами и реализуют принцип параллельных вычислений, вследствие чего именно это принцип создания суперпроизводительных компьютеров в настоящее время считается наиболее перспективным.

Для очистки совести необходимо отметить, что существуют аргументы против широкого практического применения параллельных вычислений:

- Параллельные вычислительные системы чрезмерно дороги. По подтверждаемому практикой *закону Гроша* (Herb Grosch, 60-е г.г.), производительность компьютера

растёт пропорционально квадрату его стоимости; в результате гораздо выгоднее получить требуемую вычислительную мощность приобретением одного производительного процессора, чем использование нескольких менее быстродействующих процессоров.

Контраргумент. Рост быстродействия последовательных ЭВМ не может продолжаться бесконечно (потолок в настоящее время почти достигнут), компьютеры подвержены быстрому моральному старению и необходимы частые финансовые затраты на покупку новых моделей. Практика создания параллельных вычислительных систем класса Beowulf (http://www.beowulf.org) ясно показала экономичность именно этого пути.

- При организации параллелизма излишне быстро растут потери производительности. По *гипотезе Минского* (Marvin Minsky) достигаемое при использовании параллельной системы ускорение вычислений пропорционально двоичному логарифму от числа процессоров (при 1000 процессорах возможное ускорение оказывается равным всего 10).

  Контраргумент. Приведенная оценка ускорения верна для распараллеливания определенных алгоритмов. Однако существует большое количество задач, при параллельном решении которых достигается близкое к 100% использованию всех имеющихся процессоров параллельной вычислительной системы.
- Последовательные компьютеры постоянно совершенствуются. По широко известному (и подтверждаемому практикой) *закону Мура* (Gordon Moore, 1965) сложность (тесно связанная с быстродействием) последовательных микропроцессоров возрастает вдвое каждые 18 месяцев (исторически быстродействие ЭВМ увеличивалось на порядок каждое 5-летие), поэтому необходимая производительность может быть достигнута и на "обычных" последовательных компьютерах.

  Контраргумент. Аналогичное развитие свойственно и параллельным системам. Однако применение параллелизма позволяет получать необходимое ускорение вычислений *без ожидания разработки* новых более быстродействующих процессоров. К тому же действуют серьезные фундаментальные и технологические ограничения на производство микропроцессор (напр., закон пропорциональности рассеиваемой мощности четвёртой степени тактовой частоты прямо указывает на перспективность многопроцессорности (многоядерности).
- Эффективности параллелизма сильно зависит характерных свойств параллельных систем. Все современные последовательные ЭВМ работают в соответствие с классической схемой фон-Неймана; параллельные системы отличаются существенным разнообразием архитектуры и максимальный эффект от использования параллелизма может быть получен при полном использовании всех особенностей аппаратуры (следствие - перенос параллельных алгоритмов и программ между разными типами систем затруднителен, а иногда и невозможен).

  Контраргумент. При реально имеющемся разнообразии архитектур параллельных систем существуют и определённые "устоявшиеся" способы обеспечения параллелизма (конвейерные вычисления, многопроцессорные системы и т.п.). Инвариантность создаваемого программного обеспечения обеспечивается при помощи использования стандартных программных средств поддержки параллельных вычислений (программные библиотеки PVM, MPI, DVM и др.).
- За десятилетия эксплуатации последовательных ЭВМ накоплено огромное программное обеспечение, ориентировано на последовательные ЭВМ; переработка его для параллельных компьютеров практически нереальна.

Контраргумент. Если эти программы обеспечивают решение поставленных задач, то их переработки вообще не требуется. Однако если последовательные программы не позволяют получать решение задач за приемлемое время или же возникает необходимость решения новых задач, то необходима разработка нового программного обеспечения и оно *изначально* может (и должно!) реализовываться в параллельном исполнении.

- Существует ограничение на ускорение вычисление при параллельной реализации алгоритма по сравнению с последовательной (*закон Амдаля*, связывающий величину этого ускорения с долей вычислений, которые невозможно распараллелить; подробнее о законе Амдаля см. ниже, подраздел 1.3)

  Контраргумент. В самом деле, алгоритмов вообще без (определённой) доли априори последовательных вычислений не существует. Однако это суть свойство алгоритма и не имеет отношения к возможности параллельного решения задачи вообще. Необходимо разрабатывать и научиться применять новые алгоритмы, более подходящие для решения задач на параллельных системах.

Т.о. на каждое критическое соображение против использования параллельных вычислительных технологий находится более или менее существенный контраргумент. Наиболее серьёзным препятствием к применению технологий параллельных вычислений является всё же (изначальная) нацеленность мышления современных алгоритмистов и программистов на строгую последовательность выполнения вычислений; этому не препятствует даже осознаваемая (на разумном, но отнюдь не на *подсознательном* уровне) реальность широкого применения параллелизма в выполнении микрокоманд современных процессоров даже в ПЭВМ. Очевидно, единственно реальным средством "перестройки мышления" создателей программного обеспечения является практика параллельного программирования; при этом теоретические разработки более чем полезны при совершенствовании технологий параллельного программирования.

1.2 Параллельная обработка данных

1.2.1 Принципиальная возможность параллельной обработки

Практически все разработанные к настоящему времени алгоритмы *являются последовательными*. Например, при вычислении выражения

$$a+b\times c\ ,$$

сначала необходимо выполнить умножение и только потом выполнить сложение. Если в ЭВМ присутствуют узлы сложения и умножения, которые могут работать одновременно, то в данном случае узел сложения будет простаивать в ожидании завершения работы узла умножения. Можно доказать утверждение, состоящее в том, что возможно построить машину (разумеется,

гипотетическую), которая заданный алгоритм будет *обрабатывать параллельно* (∗).

В самом деле, формула $a+b\times c$ фактически задаёт преобразование чисел a,b,c в некоторое другое число $r\leftarrow a+b\times c$ (причём *без конкретизации действий этого преобразования*!). Без ограничений общности считаем, что числа a,b,c заданы в двоичной формуле; тогда речь идет о преобразовании некоторого набора нулей и единиц (*битов*), представляющих собой последовательность чисел a,b,c в некоторый другой набор битов последовательности r (результат вычисления). Возможно построить такое логическое устройство, на вход которого поступает любая допустимая комбинация a,b,c, а на выходе сразу должна появиться комбинация, соответствующая результату r. Согласно *алгебре логики* для любой функции можно построить *дизъюнктивную нормальную формулу*, тогда i-й двоичный разряд (i=1...n) результата r будет рассматриваться как логическая функция

$$r_i = r_i(a_1,a_2...a_n,\ b_1,b_2...b_n,\ c_1,c_2...c_n),$$

где a_j,b_j,c_j являются двоичными разрядами, представляющими возможные значения a,b,c. Учитывая, что любая функция r_i (любой i-тый разряд двоичного r, i=1...m, где m - число разрядов результата) может быть представлена дизъюнктивной нормальной формулой с участием логических операций И, ИЛИ, НЕ, можно построить m процессоров (m логических схем, по одной для каждого бита числа r), которые при одновременной работе выдают нужный результат *за один-единственный такт работы вычислителя*.

Такие "многопроцессорные" машины теоретически можно построить для *каждого конкретного алгоритма* и, казалось бы, "обойти" последовательный характер алгоритмов. Однако не все так просто - конкретных алгоритмов бесконечно много, поэтому развитые выше абстрактные рассуждения имеют ограниченное отношение к практической значимости (хотя позволило разработать теоретические основы построения *спецпроцессоров* - арифметических устройств, предназначенных для сверхбыстрой обработки различных данных по одному алгоритму). Их развитие убедило в самой возможности распараллеливания, явилось основой концепции *неограниченного параллелизма*, дало возможность рассматривать с общих позиций реализацию т.н. *вычислительных сред* - многопроцессорных систем, динамически настраиваемых под конкретный алгоритм.

1.2.2 Абстрактные модели параллельных вычислений

---

* Королев Л.Н. Структуры ЭВМ и их математическое обеспечение. –М.: Наука, 1978, -352 с.

Модель параллельных вычислений обеспечивает высокоуровневый подход к определению характеристик и сравнению времени выполнения различных программ, при этом абстрагируются от *аппаратного обеспечения и деталей выполнения*. Первой важной моделью параллельных вычислений явилась *машина с параллельным случайным доступом* (PRAM, *Parallel Random Access Machine*), которая обеспечивает абстракцию машины с разделяемой памятью (PRAM является расширением модели последовательной машины с произвольным доступом RAM - *Random Access Machine*). Модель BSP (*Bulk Synchronous Parallel*, массовая синхронная параллельная) объединяет абстракции как разделенной, так и распределенной памяти. В LogP моделируются машины с распределённой памятью и некоторым способом оценивается стоимость сетей и взаимодействия. Модель *работы и глубины* NESL основана на структуре программы и не связана с аппаратным обеспечением, на котором выполняется программа.

PRAM (*Fortune, Wyllie, 1978*) является идеализированной моделью *синхронной машины с разделяемой памятью*. Постулируется, что все процессоры выполняют команды синхронно; в случае выполнения одной и той же команды PRAM является абстрактной SIMD-машиной, однако процессоры могут выполнять и различные команды. Основными командами являются *считывание из памяти, запись в память* и обычные *логические и арифметические операции*.

Модель PRAM идеализирована в том смысле, что *каждый процессор в любой момент времени может иметь доступ к любой ячейке памяти* (идеологема "*Операции записи, выполняемые одним процессором, видны всем остальным процессорам в том порядке, в каком они выполнялись, но операции записи, выполняемые разными процессорами, могут быть видны в произвольном порядке*"). Например, каждый процессор в PRAM может считывать данные из ячейки памяти или записывать данные в эту же ячейку. На реальных параллельных машинах такого, конечно, не бывает, поскольку модули памяти на *физическом уровне* упорядочивают доступ к одной и той же ячейке памяти. Более того, время доступа к памяти на реальных машинах неодинаково из-за наличия кэшей и возможной иерархической организации модулей памяти.

Базовая модель PRAM поддерживает *конкуррентные* (в данном контексте - параллельные) считывание и запись (CRCW, *Concurrent Read, Concurrent Write*). Известны подмодели PRAM, учитывающие правила, позволяющие избежать конфликтных ситуаций при одновременном обращении нескольких процессоров к общей памяти:

- Параллельное считывание при параллельной записи (CRCW, *Concurrent Read, Concurrent Write)* – данные каждой ячейки памяти в любой момент

времени могут *считываться и записываться параллельно* любыми процессорами.

- Параллельное считывание при исключительной записи (CREW, *Concurrent Read, Exclusive Write)* - из каждой ячейки памяти в любой момент времени данные могут *считываться параллельно* любыми процессорами, но записываться только *одним процессором*.
- Параллельное считывание при исключительной записи (ERCW, *Exclusive Read, Concurrent Write)* - из каждой ячейки памяти в любой момент времени данные могут считываться *только одним процессором*, однако записываться *параллельно* несколькими.
- Исключительное считывание при исключительной же записи (EREW, *Exclusive Read, Exclusive Write)* - каждая ячейка памяти в любой момент времени доступна *только одному* процессору.

Эти модели более ограничены (и, само собой, более реалистичны), однако и их трудно реализовать на практике. Даже же при этом модель PRAM и её подмодели полезны для анализа и сравнения параллельных алгоритмов. Версией модели PRAM с обменом сообщениями является BPRAM.

В модели массового синхронного параллелизма BSP (*Valiant, 1990*) синхронизация отделена от взаимодействия и учтены влияния иерархии памяти и обмена сообщениями. Модель BSP включает три компонента:

- Процессоры, имеющие локальную память и работающие с одинаковой скоростью.
- Коммуникационная сеть, позволяющая процессорам взаимодействовать друг с другом.
- Механизм синхронизации всех процессоров через регулярные отрезки времени.

Параметрами модели являются число процессоров и их скорость, стоимость взаимодействия и период синхронизации. Вычисление в BSP состоит из последовательности *сверхшагов.* На каждом отдельном сверхшаге процессор выполняет вычисления, которые обращаются к локальной памяти и отправляет сообщения другим процессорам. Сообщения являются запросами на получение копии (операция чтения) или на обновление (запись) удал**е**нных данных. В конце сверхшага процессоры выполняют барьерную синхронизацию и только затем обрабатывают запросы, полученные в течение данного сверхшага; далее процессоры переходят к выполнению следующего сверхшага.

Первоначально предл**о**женная в качестве интересной абстрактной модели, BSP позднее стала *моделью программирования*. Напр., в Оксфордском уни-

верситете (http://www.osc.ox.ac.uk) реализована библиотека взаимодействия и набор протоколирующих инструментов, основанные на модели BSP. Эта библиотека содержит около 20 функций, в которых поддерживается постулируемый BSP-стиль обмена сообщениями и удал**е**нный доступ к памяти. Подмодель E-BSP является расширением BSP, учитывающая несбалансированность схем взаимодействия.

Более современной является модель LogP (*David Culler, 1996*), т.к. она учитывает характеристики машин с распределенной памятью и содержит больше деталей, связанных со свойствами выполнения в коммуникационных сетях, нежели модель BSP. Процессы в LogP рассматриваются как асинхронные, а не синхронные. Компонентами модели являются процессоры, локальная память и соединительная сеть; своё название модель получила от прописн**ы**х букв своих параметров:

- L - верхняя граница задержки *(Latency)* при передаче от одного процессора к другому сообщения, состоящего из одного слова.
- o - накладные расходы *(overhead),* которые нес**е**т процессор при передаче сообщения (в течение этого промежутка времени процессор не может выполнять иные операции).
- g - минимальный временной интервал *(gap)* между последовательными отправками или получениями сообщений в процессоре.
- P - число пар ‘процессор-память’.

Единицей измерения времени является длительность основного цикла процессоров. Предполагается, что длина сообщений невелика, а сеть имеет конечную пропускную способность.

Модель LogP описывает свойства выполнения в коммуникационной сети, но абстрагируется от её структуры. Таким образом она позволяет моделировать взаимодействие в алгоритме, но не даёт возможности промоделировать время локальных вычислений. Такое ограничение модели было принято поскольку, во-первых, при этом сохраняется простота модели и, во-вторых, локальное (последовательное) время выполнения алгоритмов в процессорах несложно установить и без этой модели.

Моделировать схемы из функциональных элементов с помощью параллельных машин с произвольным доступом (PRAM) позволяет *теорема Брента* (∗). В качестве функциональных элементов могут выступать как 4 основных (осуществляющих логические операции NOT, AND, OR, XOR – *отрицание, логическое* И*, логическое* ИЛИ и *исключающее* ИЛИ соответственно),

---

* Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. –М.: МЦНМО, 2001, -960 с.

более сложные NAND и NOR (И-НЕ и ИЛИ-НЕ), так и любой сложности. В дальнейшем предполагается, что *задержка* (propagation delay, т.е. время срабатывания – время, через которое предусмотренные значения сигналов появляются на выходе элемента после установления значений на входах) одинакова для всех функциональных элементов.

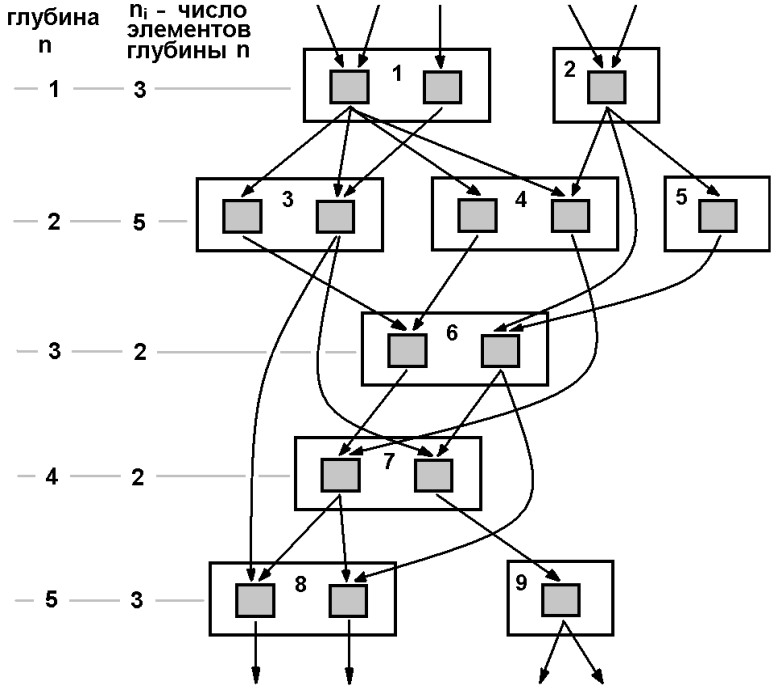

Рисунок 1 — Моделирование схемы размера 15, глубины 5 с двумя процессорами с помощью параллельной машины с произвольным доступом (PRAM - машина)

Рассматривается *схема из функциональных элементов* (combinational circuit), состоящая из *функциональных элементов* (combinational element), соединенных без образования циклов (предполагаем, что *функциональные элементы* имеют любое количество входов, но ровно один выход - элемент с несколькими выходами можно заменить несколькими элементами с единственным выходом), см. рис.1. Число входов определяет *входную степень* (fan-in) элемента, а число входов, к которым подключен выход элемента - его *выходной степенью* (fan-out). Обычно предполагается, что входные степени всех используемых элементов ограничены сверху, выходные же степени могут быть любыми. Под *размером* (size) схемы понимается количество элементов в ней, наибольшее число элементов на путях от входов схемы к выходу элемента называется *глубиной* (depth) этого элемента (глубина схемы равна наибольшей из глубин составляющих ее элементов).

Теорема Брента утверждает (доказательство в цитируемой работе), что при моделировании работы схемы глубиной d и размером n с ограниченными входными степенями элементов с использованием CREW-алгоритма на p процессорах *достаточно* (т.е. *не выше*) времени $O(n/p+d)$. Выражение $O(f)$ говорит, что скорость роста сложности алгоритма ограничена функцией f.

На рис.1 приведён результат моделирования схемы размером (общее количество процессоров) n=15 при глубине схемы (максимальное число элементов на каждом из уровней глубины) d=5 с числом процессоров p=2 (одновременно моделируемые элементы объединены в группы прямоугольными областями, причем для каждой группы указан шаг, на котором моделируются ее эле-

менты; моделирование происходит последовательно сверху вниз в порядке возрастания глубины, на каждой глубине по *p* штук за раз). Согласно теоремы Брента моделирования такой схемы на CREW-машине займет не более *ceil*(15/2+1)=9 шагов.

Там же показано, что выводы Брента верны и для моделирования схем на EREW-машине (при условии, что выходные степени всех элементов также ограничены сверху).

1.2.3 Способы параллельной обработки данных, погрешность вычислений

Возможны следующие режимы выполнения независимых частей программы:

- Параллельное выполнение - в один и тот же момент времени выполняется несколько команд обработки данных; этот режим вычислений может быть обеспечен не только наличием нескольких процессоров, но и с помощью конвейерных и векторных обрабатывающих устройств.
- Распределенные вычисления – этот термин обычно применяют для указания *способа параллельной обработки данных*, при которой используются несколько обрабатывающих устройств, достаточно удалённых друг от друга и в которых передача данных по линиям связи приводит к существенным временным задержкам. При таком способе организации вычислений эффективна обработка данных только для параллельных алгоритмов с *низкой интенсивностью потоков межпроцессорных передач данных*; таким образом функционируют, напр., *многомашинные вычислительные комплексы* (МВС), образуемые объединением нескольких отдельных ЭВМ с помощью каналов связи локальных или глобальных информационных сетей.

Формально к этому списку может быть отнесён и *многозадачный режим (режим разделения времени)*, при котором для выполнения процессов используется единственный процессор (режим является *псевдопараллельным*, ибо реального ускорения выполнения не происходит); в этом режиме удобно отлаживать параллельные приложения (функционирование каждого процессора МВС *имитируется* отдельным процессом многозадачной ОС).

Существует всего-навсего два способа параллельной обработки данных: *собственно параллелизм* и *конвейерность* [1,2].

Собственно параллелизм предполагает наличие *p* одинаковых устройств для обработки данных и алгоритм, позволяющий производить на каждом независимую часть вычислений, в конце обработки частичные данные собираются вместе для получения окончательного результата. В это случае (пренеб-

регая накладными расходами на получение и сохранение данных) получим ускорение процесса в p раз. Далеко не каждый алгоритм может быть успешно распараллелен таким способом (естественным условием распараллеливания является вычисление независимых частей выходных данных по одинаковым – или сходным – процедурам; итерационность или рекурсивность традиционно вызывают наибольшие проблемы при распараллеливании).

Идея *конвейерной обработки* заключается в выделении отдельных этапов выполнения общей операции, причём каждый этап после выполнения своей работы передаёт результат следующему, одновременно принимая новую порцию входных данных. Каждый этап обработки выполняется своей частью устройства обработки данных (*ступенью конвейера*), каждая ступень выполняет определённое действие (*микрооперацию*); общая обработка данных требует срабатывания этих частей (их число – *длина конвейера*) последовательно.

Конвейерность ("принцип *водопровода*", акад. С.А.Лебедев, 1956) при выполнении команд имитирует работу конвейера сборочного завода, на котором изделие последовательно проходит ряд рабочих мест; причём на каждом из них над изделием производится новая операция. Эффект ускорения достигается за счет *одновременной обработки ряда изделий на разных рабочих местах*.

Ускорение вычислений достигается за счёт использования всех ступеней конвейера для потоковой обработки данных (данные потоком поступают на вход конвейера и последовательно обрабатываются на всех ступенях). Конвейеры могут быть *скалярными* или *векторными* устройствами (разница состоит лишь в том, что в последнем случае могут быть *использованы обрабатывающие векторы* команды). В случае длины конвейера ℓ время обработки n независимых операций составит $\ell+n-1$ (каждая ступень срабатывает за единицу времени). При использовании такого устройства для обработки единственной порции входных данных потребуется время $\ell+n$ и только для множества порций получим ускорение вычислений, близкое к ℓ (именно в этом проявляется свойственная конвейерным устройствам сильная зависимость производительности от длины входного набора данных).

Производительность E конвейерного устройства определяется как

$$E = \frac{n}{t} = \frac{1}{\left[\tau + (\sigma + \ell - 1) \times \frac{\tau}{n}\right]}, \qquad (1)$$

где n – число выполненных операций,

t – время их выполнения,

τ - время такта работы компьютера,

σ - время инициализации команды.

Из рис.2 видно, что производительность конвейера асимптотически растёт с увеличением длины *n* набора данных на его входе, стремясь к теоретическому максимуму производительности $\frac{1}{\tau}$.

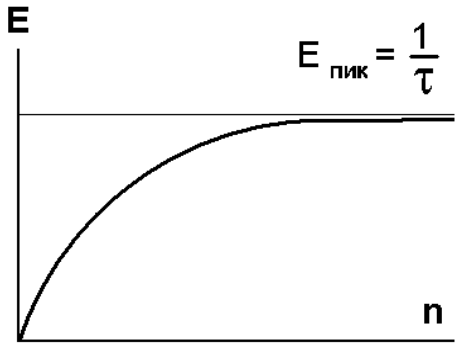

Рисунок 2 — Производительность конвейерного устройства в функции длины входного набора данных

На основе реализации *конвейера команд* были созданы известные конструкции - советская ЭВМ БЭСМ-6 (1957÷1966, разработка Института Точной Механики и Вычислительной Техники АН СССР - ИТМВТ) и английская машина ATLAS (1957÷1963); арифметический конвейер наиболее полно воплощен в суперкомпьютере CRAY-1 (1972÷1976), [1].

Существует ILP (*Instruction Level Parallelism*)-класс архитектур, характерным признаком которых является *параллельность на уровне команды* [2,3], к этому классу относятся *суперскалярные* и *VLIW-процессоры*. IPL-системы реализуют *скрытый от пользователя* параллелизм.

Система команд суперскалярных процессоров не содержит указаний на (возможную) параллельность в обработке, обеспечить динамическую загрузку параллельных программ призваны компилятор и аппаратура микропроцессора без вмешательства программиста. Архитектура современных микропроцессоров (типичный пример – процессоры серии DEC Alpha) изначально спроектирована в расчёте на выполнение как можно большего числа инструкций одновременно (в некоторых случаях в порядке, отличном от исходной последовательности в программе; причем переупорядочение может быть выполнено и транслятором). Дальнейшим развитием суперскалярной архитектуры *является мультитредовая*, в соответствие с которой программа разбивается (аппаратными и программными средствами) на совокупность *тредо*в (единиц обработки информации – частей программы, исполнению которых соответствует непрерывная область динамической последовательности инструкций); тред выполняется определённым *процессорным элементом* (составной частью *мультитредового процессора*) параллельно с другими тредами [4]. Мультитредовые процессоры уже выпускаются – напр., сетевой микропроцессор IPX1200 (фирма Level One, http://www.level1.com), MTA (компания Tera, http://www.tera.com), кристалл для проекта Blue Gene петафлопного суперкомпьютера (фирма IBM).

Архитектура *сверхдлинного командного слова* (VLIW - *Very Long Instruction Word*) берёт свое начало от параллельного микрокода, применявшегося

еще на заре вычислительной техники, а также от суперкомпьютеров Control Data CDC6600 и IBM 360/91. VLIW-команды включают несколько полей (некоторые могут быть пустыми), отвечающих каждое за свою операцию, причём все они выполняются в едином цикле. В начале 70-х г.г. многие вычислительные системы оснащались дополнительными векторными сигнальными процессорами, использующими VLIW-подобные *длинные инструкции*, прошитые в ПЗУ (обычно для выполнения быстрого преобразования Фурье и подобных вычислительных алгоритмов).

Первыми настоящими VLIW-компьютерами стали мини-суперкомпьютеры, выпущенные в начале 80-х г.г. компаниями MultiFlow, Culler и Cydrome (опередившие свое время модели не имели коммерческого успеха). VLIW-компьютер компании MultiFlow 7/300 использовал два арифметико-логических устройства для целых чисел, два АЛУ для чисел с плавающей точкой и блок логического ветвления. Его 256-битное командное слово содержало восемь 32-битовых кодов операций. Модули для обработки целых чисел могли выполнять две операции за один такт 130 нсек, что при обработке целых чисел обеспечивало быстродействие около 30 MIPS. Можно было также комбинировать аппаратные решения таким образом, чтобы получать или 256-битные или 1024-битные вычислительные машины. VLIW-компьютер Cydrome Cydra-5 использовал 256-битную инструкцию и специальный режим, обеспечивающий выполнение команд в виде последовательности из шести 40-битных операций, для чего его компиляторы могли генерировать смесь параллельного кода и обычного последовательного.

Еще один пример машин с VLIW-архитектурой – компьютер AP-120B фирмы Floating Point System (к 1980 г. было поставлено более 1’600 экземпляров, [1]). Команда AP-120B имела длину 64 разряда (6 групп, соответствующих 16-битной целочисленной арифметике, "плавающим" вычислениям, управлением вводом/выводом, командам перехода и работы с оперативной памятью) и выполнялась за 167 нсек. Естественно, последовательность столь сложных команд генерируется специальным "высокоинтеллектуальным" компилятором, выявляющим параллелизм в исходной программе и генерирующим VLIW-инструкции, отдельные поля которых могут быть выполнены независимо друг от друга (фактически параллельно).

Необходимо упомянуть отечественные разработки – параллельную вычислительную машину М-10 (1973) и векторно-конвейерную М-13 (1984) *М.А.Карцева* (Научно-исследовательский институт вычислительных комплексов - НИИВК).

Научной основой диссертации на соискание ученой степени доктора технических наук М.Карцева явилась создание первого этапа (1969 г.) СПРН (*Системы Предупреждений о Ракетных Нападениях*), для чего многие десятки машин ссерии М4 были объединены в работавшую в *реальном масштабе времени* единую вычислительную

сеть каналами длиною в тысячи километров (что, пожалуй, труднодоступно и современному InterNet'у).

На авторитетном совещании в конце 60-х г.г. рассматривалась перспективность двух подходов к созданию суперкомпьютеров – акад. С.А.Лебедев отстаивал однопроцессорный вариант максимального быстродействия (система "Эльбрус"), М.Карцев продвигал многопроцессорную систему М-10 (полностью параллельная вычислительная система с распараллеливанием на уровнях программ, команд, данных и даже слов); акад. В.М.Глушков поддержал оба направления.

Ещё в 1967 г. М.Карцев выдвинул идею создания многомашинного комплекса ВК М-9 (проект "Октябрь"), построенного из вычислительных машин, специально разработанных для совместной работы именно в таком комплексе; исследования показали, что производительность комплекса может достигнуть 10^9 операций в секунду (в то время заканчивалось проектирование БЭСМ-6 с производительностью 10^6 операций в секунду). Такая производительности достигалась разработкой новой структуры вычислительных машин – обрабатывались не отдельные числа, а группы чисел, представляющие собой приближенные представления функций либо многомерные векторы. М-10 разрабатывалась для СПРН и для общего наблюдения за космическим пространством, однако М.Карцев добился разрешения на публикацию материалов о М-10. По его инициативе на М-10 были проведены особосложные научные расчёты по механике сплошной среды (в десятки раз быстрее, чем на ЕС-1040), впервые в мире получены данные по явлению коллапса в плазме (чего не удалось сделать учёным США на мощнейшей в то время CDC-7600).

Важным этапом развития отечественных супер-ЭВМ является проект "Эльбрус-3" (1991) и его современное развитие - процессор E2k ("Эльбрус-2000", *Б.А.Бабаян,* http://www.elbrus.ru), который является определённым этапом развития VLIW-технологии. Близка к описываемым подходам и современная технология EPIC (*Explicitly Parallel Instruction Computing*), реализующая принципы явного параллелизма на уровне операций и широком командном слове (пример - разрабатываемый фирмами Intel и Hewlett Packard проект Merced и одна из его реализаций – процессор Itanium).

Исключительно интересной (и почти забытой сейчас) является "*модулярная ЭВМ*" (использующая принципы *системы счисления остаточных классов - СОК*), построенная в начале 60-х г.г. для целей ПРО (*И.Я.Акушский, Д.И.Юдицкий и Е.С.Андрианов*) и успешно эксплуатирующаяся более 40 лет [6].

В СОК каждое число, являющееся *многоразрядным* в позиционной системе счисления, представляется в виде нескольких *малоразрядных* позиционных чисел, являющихся остатками от деления исходного числа на взаимно простые основания. В традиционной позиционной двоичной системе выполнение операций (напр., сложение двух чисел) производится *последовательно по разрядам, начиная с младшего*; при этом образуется перенос в следующий старший разряд, что определяет *поразрядную последовательность обработки*. Сама сущность СОК подталкивает к распараллеливанию этого процесса: все операции над остатками по каждому основанию выполняются отдельно и независимо и, из-за их малой разрядности, *очень быстро*. Малая разрядность ос-

татков дает возможность реализовать *табличную арифметику*, при которой результат операции не вычисляется каждый раз, но, однажды рассчитанный, помещается в запоминающее устройство и при необходимости из него считывается. Т.о. операция в СОК при табличной арифметике и конвейеризации выполняется за один период синхронизирующей частоты (*машинный такт*). Табличным способом в СОК можно выполнять не только простейшие операции, но и вычислять сложные функции, и также за один машинный такт. В СОК относительно просто ввести функции контроля и исправления ошибок в процессе вычисления арифметических операций (особенно важно для работающих в критичных условиях ЭВМ). Модулярные ЭВМ Т-340А и К-340А имели быстродействие $1{,}2\times10^6$ двойных ($2{,}4\times10^6$ обычных) операций в секунду (в начале 60-х г.г. типовое быстродействие ЭВМ измерялось десятками или сотнями тысяч операций в секунду). В начале 70-х г.г. работами в области модулярной арифметики заинтересовались западные разработчики компьютерной техники, но все предложения о легальной закупке результатов разработки были пресечены глубококомпетентными органами; с тех пор модулярной арифметикой в нашей стране занимаются только отдельные энтузиасты-теоретики.

Параллельные системы априори предназначены для решения задач большой размерности, поэтому среди прочих источников погрешностей заметной становится *погрешность округления*. Известно, что (при *разумных допущениях*) среднеквадратичное значение погрешности округления $\sigma = O(\beta\sqrt{e})$, где β - значение младшего разряда мантиссы числа (для ‘плавающих’ чисел одинарной точности при 3-х байтовой мантиссе $\beta = 2^{-24}$), e – число арифметико-логических операций в задаче [7]. Для компьютера с производительностью всего 1 Гфлопс за час работы величина e достигает 4×10^{12}, а σ будет иметь порядок $O(2^{-24}\times10^6)\approx O(10^{-6}\times2^{-4}\times10^6)=O(2^{-4})\approx6\%$; такая погрешность недопустима, поэтому для параллельных вычислений практически всегда используется двойная точность.

Эмпирически замечено, что до 90% обращений в ОП производится в ограниченную область адресов. Эта область называется *рабочим множеством*, которое медленно перемещается в памяти по мере выполнения программы. Для рабочего множества целесообразно реализовать промежуточную (локальную для данного АЛУ и очень быстродействующую) память небольшого размера (*кэш*), использование кэш-памяти очень эффективно для ускорения обращения к ОП. Во многих случаях рациональные преобразования программы (напр., циклических участков) позволяют многократно (иногда в десятки раз!) повысить быстродействие *даже последовательной* программы за счет "оседания" в кэше часто используемых переменных.

1.3 Понятие параллельного процесса и гранулы распараллеливания

Наиболее общая схема выполнения последовательных и параллельных вычислений приведена на рис.3 (моменты времени S и E – начало и конец задания соответственно).

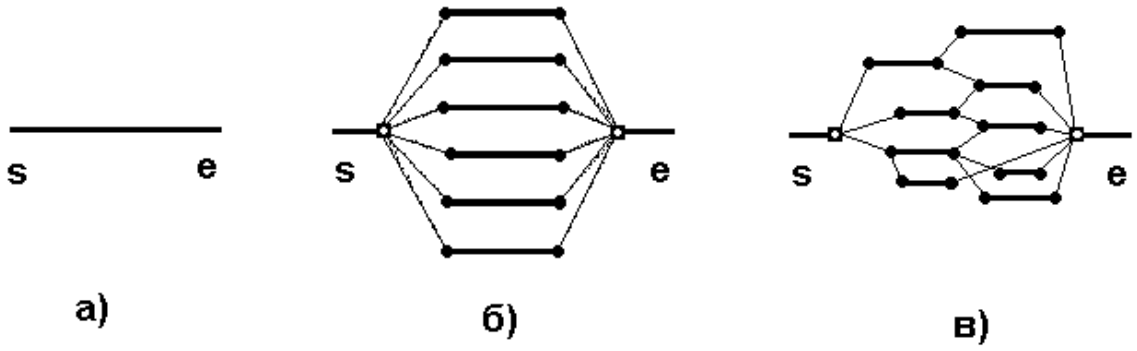

Рисунок 3 — Диаграммы выполнения процессов при последовательном вычислении - а), при близком к идеальному распараллеливании - б) и в общем случае распараллеливания - в)

Процессом называют определённые последовательности команд, наравне с другими процессами претендующие для своего выполнения на использование процессора; команды (инструкции) внутри процесса выполняются последовательно [2,6], внутренний (напр., конвейерный) параллелизм при этом не учитывают.

На рис.3 тонкими линиями показаны действия по созданию процессов и обмена данными между ними, более толстыми – непосредственно исполнение процессов (ось абсцисс - время). В случае последовательных вычислений создается единственный процесс (рис.3а), осуществляющий необходимые действия; при параллельном выполнении алгоритма необходимы несколько (иногда десятки/сотни/тысячи) процессов (*параллельных ветвей* задания). В идеальном случае (*близкое к идеальному распараллеливание*) все процессы создаются одновременно и одновременно завершаются (рис.3б), в общем случае диаграмма процессов значительно сложнее и представляет собой некоторый граф (рис.3в).

Характерная длина последовательно выполняющейся группы команд в каждом из параллельных процессов называется размером гранулы (зерна). В любом случае целесообразно стремиться к "крупнозернистости" (идеал – диаграмма 3б). Обычно размер зерна (*гранулы*) составляет десятки-сотни тысяч машинных операций (что на порядки превышает типичный размер оператора языков Fortran или C/C++, [3]). В зависимости от размеров гранул говорят о мелкозернистом и крупнозернистом параллелизме (*fine-grained parallelism* и *coarse-grained parallelism* соответственно). На размер гранулы влияет и удобство программирования – в виде гранулы часто оформляют некий логически

законченный фрагмент программы. Целесообразно стремиться к равномерной загрузке процессоров (как по количеству вычислительных операций, так и по загрузке оперативной памяти)

Разработка параллельных программ является непростой задачей в связи с трудностью выявления *параллелизма* (обычно *скрытого*) в программе (т.е. выявления частей алгоритма, могущих выполняться независимо друг от друга). Например, для задачи вычислений фрагмента программы (ZnN – вычисленные значения, OpN – операнды, FunN – вычислительные процедуры, DN – арифметико-логические выражения):

```
Zn1 ← Fun1(Op1,Op2) D1 Fun2(Op3) D2 Fun3(Op3); // operator 1        (2)
Zn2 ← Fun4(Op4,Zn1) D3 Fun5(Op3); // operator 2
```

выявить параллелизм "вручную" несложно. Даже на первый взгляд видно, что вычисление оператора Zn2 невозможно без предварительного вычисления Zn1 (при этом говорят, что оператор "Zn1 зависит от Zn2", т.е. существует *зависимость по данным*; подробнее см. подраздел 3.2).

В простейшем случае возможно распределить по процессам вычисления Fun1÷Fun3 в соответствие со схемой рис.3б и далее "собрать" вычисленные значения для определения Zn1, в дальнейшем подобным образом поступить с вычислением Zn2 (что вычисление Fun5 можно совместить с вычислением Fun1÷Fun3); в этом гранулами параллелизма являются вычисления Fun1÷Fun5. Заметим, что автоматизация выявления параллелизма (столь легко проделанная "вручную" для простейшего случая (2)) в реальных алгоритмах непроста и в полном объёме вряд ли может быть решена в ближайшее время. В то же время за десятилетия эксплуатации вычислительной техники разработано такое количество (тщательно отлаженных и оптимизированных) последовательных алгоритмов, что не может быть и речи об их ручном распараллеливании.

Построим простейшую математическую модель для оценки возможного повышения производительности при распараллеливании вычислений с учетом времени обмена данными. Пусть t_O - характерное время обмена (операций пересылки данных между процессами), $t_З$ - время выполнения последовательности команд, составляющих гранулу, n – число гранул. Примем (упрощенно), что при распараллеливании каждая гранула выполняется на отдельном процессоре, время вычисления последовательности команд каждой гранулы одинаково, каждый процесс требует при функционировании не более двух обменов (как показано на рис.3б). Тогда последовательный алгоритм выполнится (в среднем) за время

$$T_{посл} = t_З \times n,$$

а время выполнения параллельного алгоритма

$T_{парр} = t_з + 2 \times t_о$.

Выигрыш в производительности параллельного алгоритма по сравнению с последовательным (*коэффициент ускорения вычислений*) в этом случае:

$$k = \frac{T_{посл}}{T_{парр}} = \frac{t_з \times n}{t_з + 2 \times t_о}. \qquad (3)$$

Расчёт количества процессоров, необходимых для получения *хотя бы* k=1 в функции $t_о/t_з$ дан в табл.1 (значения n не всегда округлены до целого).

Таблица 1 — Минимальное расчётное количество процессоров $n_{k=1}$, необходимых для гарантированного ускорения при параллельном выполнении (расчёт по выражению (3))

$t_о/t_з$	*0,125*	*0,25*	*0,5*	*1,0*	*2,0*	*5,0*	*7,5*	*10,0*
$n_{k=1}$	1,25	1,5	2	3	5	11	15	21
	$t_о/t_з < 1$ (время обмена данными меньше времени выполнения гранулы)				$t_о/t_з > 1$ (время обмена данными превышает время выполнения гранулы)			

Как видно из табл.1, только при малости времени обмена (по сравнению с временем выполнения процесса) эффективность распараллеливания высока (для достижения k=1 требуется мало процессов), при превышении времени обмена времени выполнения процесса достижение хотя бы k=1 требует большого количества процессов.

Таблица 2 — Ускорение вычислений k в зависимости от числа процессов n и величины $t_о/t_з$ (расчет по (3))

n	*2*	*3*	*5*	*7*	*10*	*15*	*20*
$t_о/t_з$=0,1	1,67	2,5	4,17	5,83	8,33	12,5	16,7
$t_о/t_з$=1	0,667	1,0	1,67	2,33	3,33	5,0	6,67
$t_о/t_з$=10	0,095	0,143	0,238	0,333	0,476	0,714	0,952

Как и следовало ожидать (табл.2), снижение отношения $t_о/t_з$ в высшей степени благоприятно для повышения ускорения при параллелизации вычислений.

Т.о. даже простейшая модель (не учитывающая реальной диаграммы распараллеливания, задержек на старт процессов, зависимость времени обменов

от размера сообщения и множества других параметров) параллельных вычислений позволила выявить важное обстоятельство – время обмена данными должно быть как можно меньше времени выполнения последовательности команд, образующих гранулу.

Если в (2) качестве Fun1÷Fun5 выступают функции типа sin(), cos(), atan(), log(), pow(), характерное время $t_З$ выполнения которых единицы микросекунд, время $t_О$ обмена данными должно составлять десятые/сотые доли мксек (столь "быстрых" технологий сетевого межпроцессорного обмена не существует); для гранул большего размера допустимо время обменов в десятки (в некоторых случаях – даже сотни) мксек. Для реальных задач (традиционно) характерный размер зерна распараллеливания на несколько порядков превышает характерный размер оператора традиционного языка программирования (C/C++ или Fortran).

Соотношение $t_О/t_З$ определяется в значительной степени структурой и аппаратными возможностями вычислительного комплекса (*архитектурой многопроцессорной системы*), при этом *рациональный размер зерна распараллеливания также зависит от используемой архитектуры*.

1.4 Взаимодействие параллельных процессов, синхронизация процессов

Выполнение команд программы образует *вычислительный процесс*, в случае выполнения нескольких программ на общей или разделяемой памяти и обмене этих программ сообщениями принято говорить о *системе совместно протекающих взаимодействующих процессов* [7].

Порождение и уничтожение процессов UNIX-подобных операционных системах выполняются операторами (системными в**ы**зовами) fork, exec и exit.

Системный вызов fork при выполнении текущим (*родительским, parent process*) процессом запрашивает ядро операционной системы на создание дочернего (*child process*) процесса, являющегося *почти* (за исключение значения идентификатора процесса pid) точной копией текущего. Оба процесса при успешном выполнении fork выполняются одновременно с оператора, непосредственно следующего после вызова fork. Вызовы операторов семейства exec загружают исполняемую программу на место вызывающей, произведший этот вызов (pid процесса и файловые дескрипторы сохраняют свои значения для замещающего процесса), exec-вызовы позволяют передавать порожденному процессу параметры. Создание *двух разных* процессов реализуется последовательностью вызовов fork родительским процессом (создаются два процесса с одинаковыми кодами и данными) и exec дочерним процессом (при этом он завершается и замещается "новым" процессом). Корректность работы с процессами требует вызова родительским процессом wait после fork (что-

бы приостановить работу до момента завершения дочернего процесс и замещения его "новым").

Системный вызов exit завершает выполнение процесса с возвратом заданного значения целочисленного статуса завершения (*exit status*), процесс-родитель имеет возможность анализировать эту величину (при условии выполнения им wait).

Параллелизм часто описывается в терминах макрооператоров FORK и JOIN, в параллельных языках запуск *параллельных ветвей* осуществляется с помощью оператора FORK M1, M2 , ...ML, где M1, M2, ...ML - имена независимых ветвей; каждая ветвь заканчивается оператором JOIN (R,K), исполнение которого вызывает вычитание единицы из ячейки памяти R. Предварительно в R записывается равное количеству ветвей число, при последнем срабатывании оператора JOIN (т.е. когда все ветви выполнены) в R оказывается нуль и управление передаётся оператору K (в некоторых случаях в JOIN описывается подмножество ветвей, при выполнении которого срабатывает этот оператор). В последовательных языках аналогов операторам FORK и JOIN не может быть и ветви выполняются последовательно друг за другом.

В терминах FORK/JOIN итерационная параллельная программа может иметь следующий вид (функции F1, F2 и F3 из-за различной алгоритмической реализации должны вычисляться отдельными программными сегментами, которые логично оформить в виде ветвей параллельной программы), [7]:

```
        X1=X10; X2=X20; X3=X30; R=3;      // инициализация переменных
LL      FORK M1, M2, M3                   // запустить параллельно M1,M2,M3
M1      Z1 = F1 (X1, X2, X3)              // вычислить Z1=F1(X1, X2, X3)
        JOIN (R, KK)                      // R=R-1
M2      Z2 = F2 (X1, X2, X3)              // вычислить Z2=F2(X1, X2, X3)
        JOIN (R, KK)                      // R=R-1
M3      Z3 = F3 (X1, X2, X3)              // вычислить Z3=F3(X1, X2, X3)
        JOIN (R, KK)                      // R=R-1
KK      IF (ABS (Z1-X1) < ε) AND          // если абсолютная точность ε
          (ABS (Z2-X2) < ε) AND           // вычислений достигнута…
          (ABS (Z3-X3) < ε )
        THEN вывод результатов;           // то вывести данные и закончить
             STOP
        ELSE X1=Z1; X2=Z2; X3=Z3;         // иначе переопределить X1,X2,X3
             GO TO LL                     // и повторить цикл вычислений
```

Для многопроцессорных вычислительных систем особое значение имеет обеспечение *синхронизации процессов* (вышеописанный оператор JOIN совместно с ячейкой R как раз и осуществляет такую синхронизацию - состояние R=0 свидетельствует об окончании процессов разной длительности). Например, момент времени наступления обмена данными между процессорами априори никак не согласован и не может быть определён точно (т.к. зависит

от множества трудноо цениваемых параметров функционирования МВС), при этом для обеспечения полноценного рандеву (встречи для обмена информацией) просто необходимо применять синхронизацию. В слабосвязанных МВС вообще нельзя надеяться на абсолютную синхронизацию машинных часов отдельных процессоров, можно говорить только об измерении промежутков времени во временной системе каждого процессора.

Синхронизация является действенным способом предотвращения ситуаций дедлока (*deadlock*, клинч, тупик) – ситуации, когда каждый из взаимодействующих процессов получил в распоряжение часть необходимых ему (разделяемых с другими процессами) ресурсов, но ни ему ни иным процессам этого количества ресурсов недостаточно для завершения обработки (и последующего освобождения ресурсов), [4].

Одним из известных методов синхронизации является использование семафоров (*Е.Дейкстра*, 1965). *Семафор* представляет собой доступный исполняющимся процессам целочисленный объект, для которого определены следующие элементарные (*атомарные, неделимые*) операции:

- Инициализация семафора, в результате которой семафору присваивается неотрицательное значение.
- Операция типа P (от датского слова *proberen* - проверять), уменьшающая значение семафора. Если значение семафора опускается ниже нулевой отметки, выполняющий операцию процесс приостанавливает свою работу.
- Операция типа V (от *verhogen* - увеличивать), увеличивающая значение семафора. Если значение семафора в результате операции становится больше или равно 0, один из процессов, приостановленных во время выполнения операции P, выходит из состояния приостанова.
- Условная операция типа P (сокращенно CP, *conditional P*), уменьшающая значение семафора и возвращающая логическое значение "истина" в том случае, когда значение семафора остаётся положительным. Если в результате операции значение семафора должно стать отрицательным или нулевым, никаких действий над ним не производится и операция возвращает логическое значение "ложь".

В системах параллельного программирования используют существенно более высокоуровневые приемы синхронизации. В технологии параллельного программирования MPI (см. раздел 4.2) применена схема синхронизации обмена информацией между процессами, основанная на использовании примитивов send/receive (послать/принять сообщение). При этом send может быть вызван в любой момент, а receive задерживает выполнение программы до момента реального поступления сообщения (т.н. *блокирующие функции*). В MPI определен оператор Barrier, явным образом блокирующий выполнение

данного процесса до момента, когда все (из заданного множества) процессы не вызовут Barrier. Функция синхронизации в Т-системе (раздел 4.3.2) поддерживается механизмом *неготовых переменных*, в системе программирования Linda (раздел 4.2) при отсутствии явной поддержки синхронизация процессов несложно моделируется языковыми средствами.

Синхронизация может поддерживаться и аппаратно (например, *барьерная синхронизация* в суперкомпьютере Cray T3, с помощью которой осуществляется ожидание всеми процессами определенной точки в программе, после достижения которой возможна дальнейшая работа, см. подраздел 2.4.1).

1.5 Возможное ускорение при параллельных вычислениях (закон Амдаля)

Представляет интерес *оценка* величины возможного повышения производительности с учётом *качественных характеристик* самой исходно последовательной программы.

Закон Амдаля (*Gene Amdahl*, 1967) связывает потенциальное ускорение вычислений при распараллеливании с долей операций, выполняемых *априори последовательно* [1,7]. Пусть f ($0<f<1$) – часть операций алгоритма, которую распараллелить не представляется возможным; тогда распараллеливаемая часть равна ($1-f$); при этом затраты времени на передачу сообщений не учитываются, t_s - время выполнения алгоритма на одном процессоре (последовательный вариант), n – число процессоров параллельной машины.

При переносе алгоритма на параллельную машину время расчёта распределится так:

- $f \times t_s$ - время выполнения части алгоритма, которую распараллелить невозможно,
- $(1-f)\times t_s/n$ - время, затраченное на выполнение распараллеленной части алгоритма.

Время t_p, необходимое для расчёта на параллельной машине с n процессорами, равно (см. рис.4)

$$t_p = f \times t_s + (1-f)\times t_s / n,$$

а ускорение времени расчёта

$$S \le \frac{t_s}{t_p} = \frac{t_s}{f \times t_s + (1-f)\times t_s / n} = \frac{1}{f + \left(\frac{1-f}{n}\right)}. \qquad (4)$$

Анализ выражения (4) показывает (т.к. $\lim_{n\to\infty} S(f,n)=\frac{1}{f}$), что только при малой доли последовательной операций (f<<1) возможно достичь значительного (естественно, не более чем в n раз) ускорения вычислений (рис.5). В случае f=0,5 ни при каком (даже бесконечно большом) количестве процессоров невозможно достичь S>2! Заметим, что эти ограничения носят *фундаментальный характер* (их нельзя обойти для заданного алгоритма), однако практическая оценка доли f последовательных операций априори обычно невозможна.

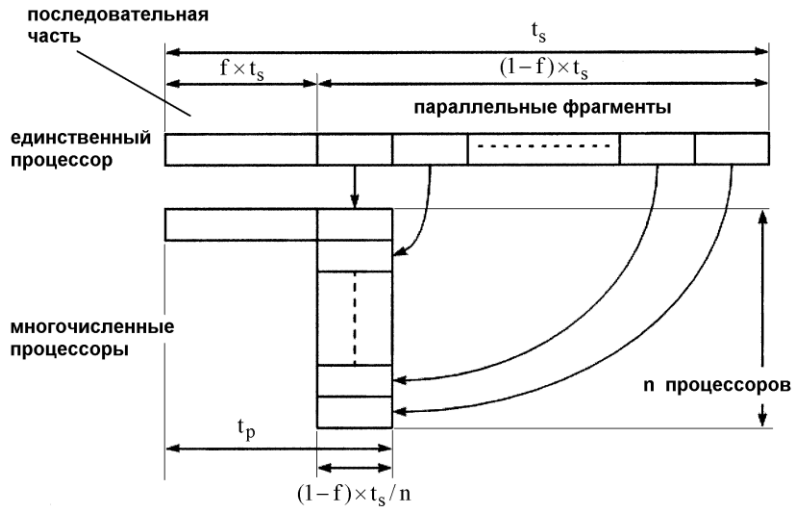

Рисунок 4 — Схема к выводу закона Амдаля

Таким образом качественные характеристики самого алгоритма накладывают ограничения на возможное ускорение при распараллеливании. Например, характерные для инженерных расчётов алгоритмы счета по последовательным формулам распараллеливаются плохо (часть f значима), в то же время сводимые к задачам *линейного программирования* (ЛП – операции с матрицами – умножение, обращение, нахождение собственных значений, решение СЛАУ – систем линейных алгебраических уравнений и т.п.) алгоритмы распараллеливаются вполне удовлетворительно.

Априорно оценить долю последовательных операций f непросто (само понятие части "последовательных" и "параллельных" операций трудноформализируемо и допускает неоднозначные толкования). Однако можно попробовать *формально* использовать формулу Амдаля для решения обратной задачи – определения f по экспериментальным данным производительности; это даёт возможность *количественно* судить о достигнутой *эффективности распараллеливания*.

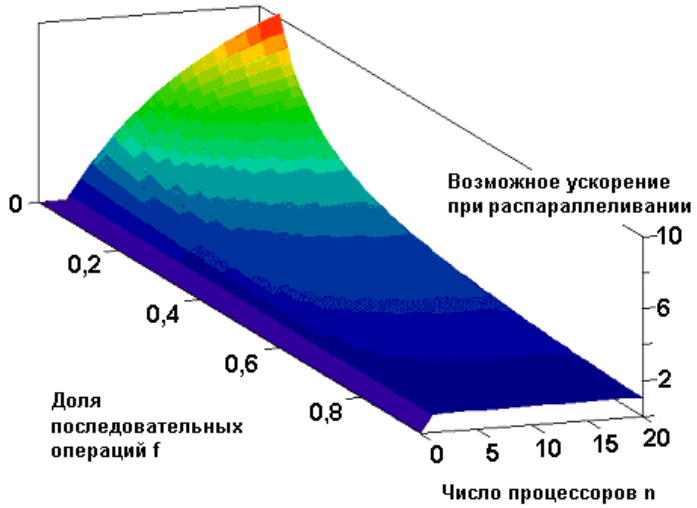

Рисунок 5 — Трехмерный график, количественно отражающий зависимость Амдаля согласно выражения (4)

На рис.6 приведены результаты эксперимента на кластере SCI-MAIN НИВЦ МГУ на задаче умножения матриц по ленточной схеме (порядок матриц 10^3 вещественных чисел двойной точности, серия опытов марта 2005 г.), экспериментальные данные наилучшим образом (использован *метод наименьших квадратов*) соответствуют формуле Амдаля при f=0,051 (вывод - данный алгоритм распараллелен эффективно, т.к. доля параллельных операций составляет ≈95%, что вполне удовлетворительно для алгоритма с вычислительной сложностью уровня $O(n^3)$ операций). *Формальный вывод* – ждать более чем 20-ти кратного увеличения производительности такого алгоритма не следует!

Закон Амдаля удобен для качественного анализа проблемы распараллеливания, недостатком выражения (4) является неучёт потерь времени на межпроцессорный обмен сообщениями (что как раз и выражено знаком ‘меньше или равно’ в (4)); эти потери могут не только снизить ускорение вычислений в параллельном варианте, но и замедлить вычисления по сравнению с последовательным. Более общим является выражение (т.н. *сетевой закон Амдаля*):

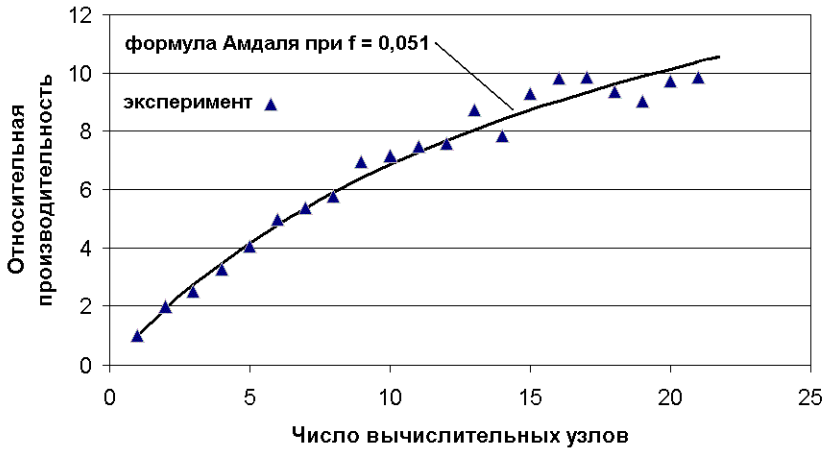

Рисунок 6 — Производительность вычислительной кластерной системы на процедуре умножения матриц (эксперимент и расчёт по формуле Амдаля)

$$S = \frac{1}{f + \left(\frac{1-f}{n}\right) + c},$$

где c – *коэффициент сетевой деградации вычислений*, $c = c_w \times c_t = \frac{W_c \times t_c}{W \times t}$,

W_c - количество передач данных, W – общее число вычислений в задаче, t_c - время одной передачи данных, t – время выполнения одной операции.

Сомножитель $c_w = \frac{W_c}{W}$ (рост которого *снижает* S) определяет составляющую коэффициента деградации, вызванную свойствами алгоритма распараллеливания: $c_t = \frac{t_c}{t}$ - зависящую от соотношения производительности процессора и аппаратуры сети ("*техническую"*) составляющую; значения c_w и c_t могут быть оценены заранее. Т.о. полученные (при неучёте сетевой деградации вычислений) из выражения (4) значения f являются пессимистическими.

Было бы странно, если бы даже при идеальном параллелизме (f=0) сетевое ускорение вычислений могло быть выше значения

$$\lim_{c \to 0} \left[\frac{1}{f + \left(\frac{1 - f}{n} \right) + c} \right] = n$$

Удивительно (для *столь ограниченной* модели!), что в некоторых случаях ускорение времени вычислений на n-процессорной МВС количественно превышает величину числа процессоров (т.н. "*парадокс параллелизма"*)! Объяснения лежат в чисто технической области - напр., обработка однопроцессорной (или SMP, см. подраздел 2.1) системы матриц значительного размера с большой вероятностью приведёт к необходимости сброса части элементов матриц на внешнюю (обычно дисковую) память (своппинг, *swapping*), а длительность этой процедуры на многие порядки превышает время обращения к ОП. При разумном программировании для МВС эти матрицы будут распределены между ОП процессоров, причем каждая часть матриц полностью помещается в ОП и своппинга не будет – в этом и объяснение *"чудесного"* повышения быстродействия.

Контрольные вопросы

1. Что такое суперкомпьютер? Чем он (*количественно* и *качественно*) отличается от персональной ЭВМ? Почему суперкомпьютер выполняется с использованием технологий многомашинных комплексов и/или многопроцессорных систем? В каких областях человеческой деятельности необходимо применение суперкомпьютеров?
2. В каких единицах выражается производительность супер-ЭВМ? Каким образом сравниваются производительности различных суперкомпьютеров? Какие из этих характеристик (хотя бы качественно) наиболее объективны?
3. Какие причины тормозят дальнейшее увеличение вычислительных мощностей однопроцессорных компьютеров? Каковы аргументы против применения параллельных вычислений? Есть ли альтернативы параллельным вычислительным технологиям?
4. Доказать возможность полностью параллельного вычисления любого заданного алгоритма. Какова его практическая ценность?
5. В чем суть теоремы Брента? Каким путем можно уменьшить число шагов при моделировании (*построении*) схемы из функциональных элементов?
6. Дать определение процесса (*параллельного процесса*) и зерна (гранулы) распараллеливания. В *каких единицах* можно определить величину гранулы (привести *несколько вариантов* ответа)?
7. Для чего нужна синхронизация выполнения параллельных процессов? Каким образом синхронизация может помочь в решении проблемы возникновения дедлоков?
8. Чем ограничивается возможность распараллеливания реальных алгоритмов? В чем суть закона Амдаля? Предложите методику "*обхода*" закона Амдаля?
9. Какую (*количественно!*) долю последовательных операций можно считать допустимой для различных вычислительных алгоритмов?