МГУПИ

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ
БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО
ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ПРИБОРОСТРОЕНИЯ И ИНФОРМАТИКИ

Кафедра "Персональные компьютеры и сети"

Баканов В.М.

МНОГОМАШИННЫЕ КОМПЛЕКСЫ И МНОГОПРОЦЕССОРНЫЕ СИСТЕМЫ

Учебное пособие

Москва, 2014

УДК 681.3

Рекомендовано к изданию в качестве учебного пособия
редакционно-издательским советом МГУПИ

Рецензент: профессор, к.т.н. Зеленко Г.В.

Баканов В.М.

Многомашинные комплексы и многопроцессорные системы: учебное пособие. — М.: МГУПИ. Москва, 2014. — 126 с.

В пособии изложены требования науки и промышленности, приводящие к использованию многомашинных комплексов и многопроцессорных систем, которые неизбежно используют принцип параллельности вычислений, история вопроса и современное состояние проблемы, описаны основные подходы к организации многопроцессорных вычислительных систем, разработке параллельных алгоритмов численного решения задач и технологий параллельного программирования.

Табл. 6. Ил. 30. Библиограф.: 8 назв.

УДК 681.3

© Баканов В.М., 2014

© МГУПИ, 2014

Содержание

Стр.

Введение 5
1 Общие вопросы решения "больших задач"………………………. 7
1.1 Современные задачи науки и техники, требующие для решения суперкомпьютерных мощностей..……………………………….. 7
1.2 Параллельная обработка данных……………………………... 13
1.2.1 Принципиальная возможность параллельной обработки…… 13
1.2.2 Абстрактные модели параллельных вычислений……………. 15
1.2.3 Способы параллельной обработки данных, погрешность вычислений………………………………………………………. 19
1.3 Понятие параллельного процесса и гранулы распараллеливания…………………………………………………………………... 25
1.4 Взаимодействие параллельных процессов, синхронизация процессов…………………………………………………………… 28
1.5 Возможное ускорение при параллельных вычислениях (закон Амдаля)……………………………………………………………….. 31
2 Принципы построения многопроцессорных вычислительных систем………………………………………………………………. 36
2.1 Архитектура многопроцессорных вычислительных систем…... 36
2.2 Распределение вычислений и данных в многопроцессорных вычислительных системах с распределенной памятью………………. 41
2.3 Классификация параллельных вычислительных систем……… 42
2.4 Многопроцессорные вычислительные системы с распределенной памятью...................……………………………………………… 46
2.4.1 Массивно-параллельные суперкомпьютеры серии Cray T3… 47
2.4.2 Кластерные системы класса BEOWULF……………………… 49
2.4.3 Коммуникационные технологии, используемые при создании массово-параллельных суперкомпьютеров……………………… 53
2.4.3.1 Транспьютерные системы………………………………… 53
2.4.3.2 Универсальные высокопроизводительные коммуникационные сети: производительность, латентность и цена обмена ……… 55
2.4.4 Стандартные программные пакеты организации вычислительных кластеров……………………………………………………… 57
2.5 Нетрадиционные архитектуры многопроцессорных вычислительных систем………………………………………………………… 58
2.6. Метакомпьютинг и GRID-системы…………………………….. 65
3 Анализ алгоритмов с целью выявления параллелизма………….. 70
3.1 Представление алгоритма графовой структурой……………… 70

3.2 Потенциал распараллеливания циклов. Циклы ParDO……….. 80
3.3 Эквивалентные преобразования алгоритмов, избыточность вычислений и обменов данными ………………………………………. 91
4 Технологии параллельного программирования………………….. 97
4.1 Дополнения известных последовательных алгоритмических языков средствами параллельного программирования ……………… 98
4.2 Системы параллельного программирования на основе обмена сообщениями……………………………………………………….. 104
4.3 Автоматизация распараллеливания алгоритмов………………... 108
4.3.1 Распараллеливающие компиляторы…………………………... 108
4.3.2 Автоматическое распараллеливание с помощью Т-системы... 109
4.3.3 Непроцедурный декларативный язык НОРМА……………… 111
4.4 Стандартные предметно-ориентированные библиотеки параллельных вычислений…………………………………………………. 113
4.5 Параллелизм в системах управления базами данных………….. 116
Заключение... 120
Список использованной литературы.. 121
Приложение А. Примеры последовательной и параллельных реализаций алгоритма Якоби решения сеточных уравнений…………….. 122
Приложение Б. Параллельная Fortran-программа с использованием OpenMP……………………………………………………………….. 124
Приложение В. Программа рекурсивного обхода дерева………….. 124
Приложение Г. НОРМА - программа умножения матриц………… 125

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 стойкости 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 серьезнейших 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 $2{,}6 \times 10^6$ (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 $2{,}6 \times 10^7$ числами. При шаге дискретизации по времени 10 мин за моделируемый промежуток времени необходимо определить $\approx 5 \times 10^4$ ансамблей (т.е. 10^{14} необходимых числовых значений *промежуточных вычислений*). При оценке числа необходимых для получения каждого промежуточного результата вычислительных операций в $10^2 \div 10^3$ общее число необходимых для проведения численного эксперимента с глобальной моделью атмосферы вычислений с плавающей точкой доходит до $10^{16} \div 10^{17}$. 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 ($2006 \div 2010$ 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:

- <u>Параллельное считывание при параллельной записи</u> (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) удаленных 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 предложенная 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 удаленный 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 несет 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.

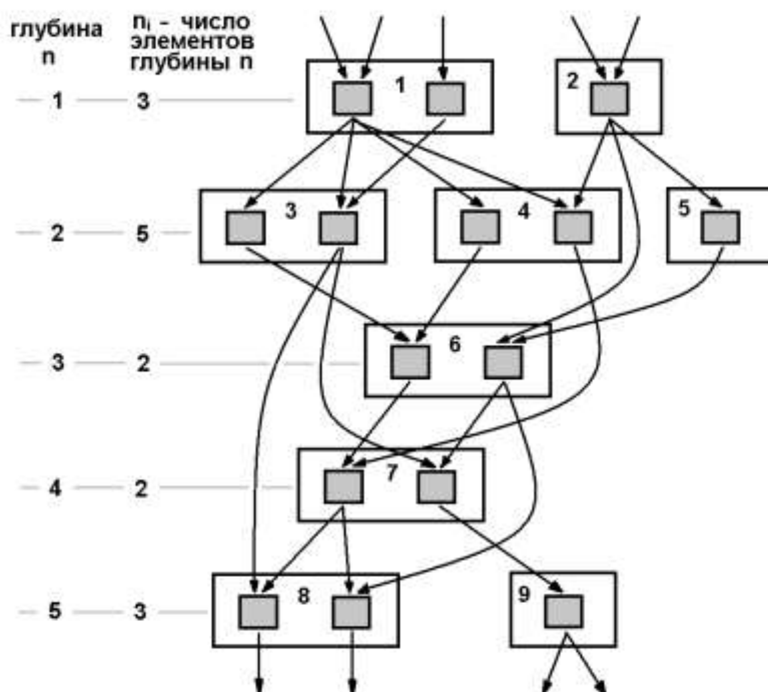


Рисунок 1 — Моделирование схемы размера 15, глубины 5 с двумя процессорами с помощью параллельной машины с произвольным доступом (PRAM - машина)

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 $\frac{1}{\tau}$.

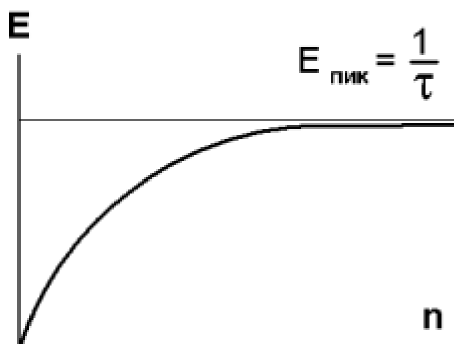


Рисунок 2 — Производительность конвейерного устройства в функции длины входного набора данных

На основе реализации *конвейера команд* были созданы известные конструкции - советская ЭВМ БЭСМ-6 (1957$\div$1966, разработка Института Точной Механики и Вычислительной Техники АН СССР - ИТМВТ) и английская машина ATLAS (1957$\div$1963); арифметический конвейер наиболее полно воплощен в суперкомпьютере CRAY-1 (1972$\div$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 $O(2^{-24}\times10^6)\approx O(10^{-6}\times2^{-4}\times10^6)=O(2^{-4})\approx 6\%$; 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).

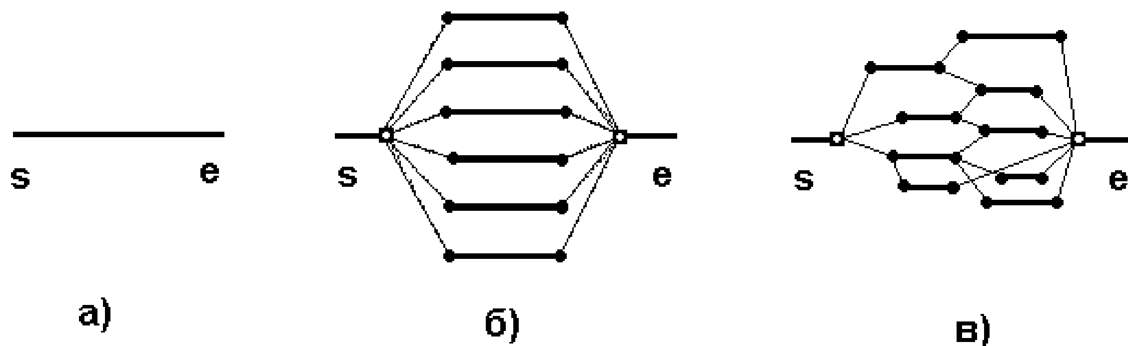


Рисунок 3 — Диаграммы выполнения процессов при последовательном вычислении - а), при близком к идеальному распараллеливании - б) и в общем случае распараллеливания - в)

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_3 выполнения которых единицы микросекунд, время t_O 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 $t_o/t_з$ 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 вызовами) 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 труднооцениваемых 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

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

Анализ выражения (4) показывает (т.к. $\lim\limits_{n\to\infty} S(f,n)=\frac{1}{f}$), 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.

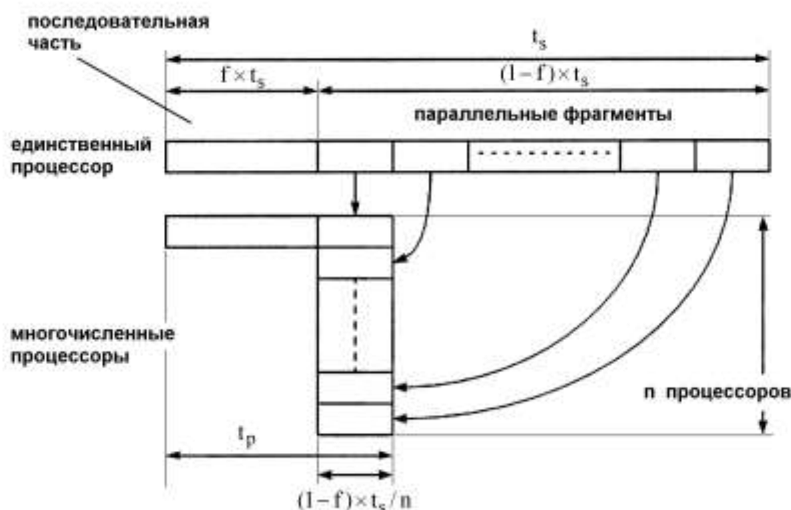


Рисунок 4 — Схема к выводу закона Амдаля

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*.

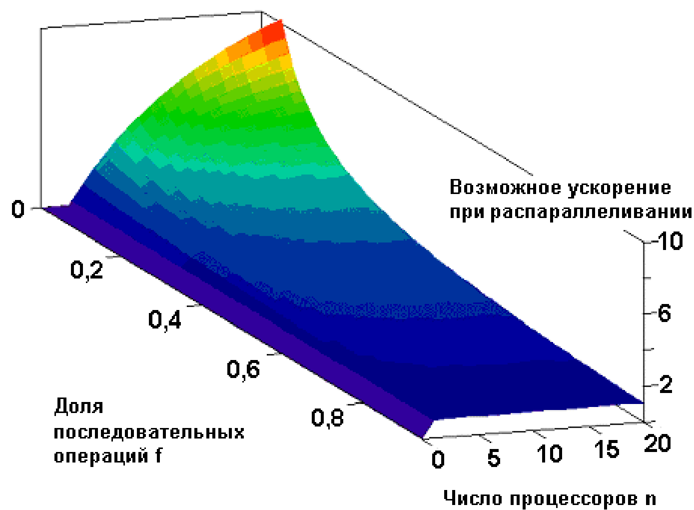


Рисунок 5 — Трехмерный график, количественно отражающий зависимость Амдаля согласно выражения (4)

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*):

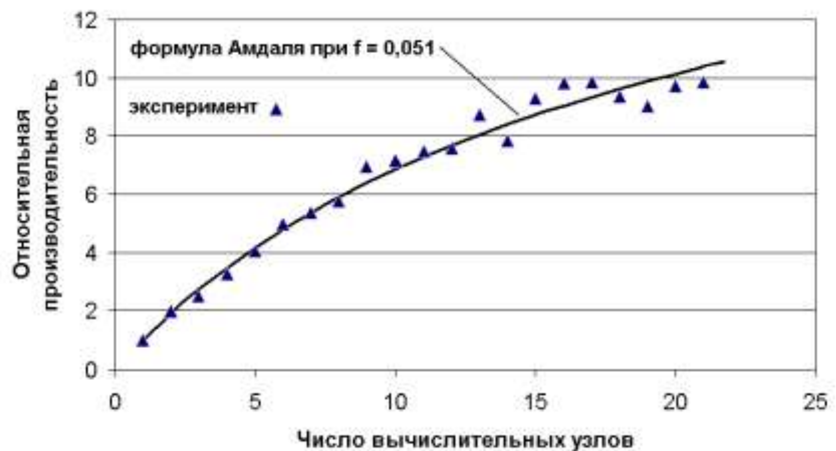


Рисунок 6 — Производительность вычислительной кластерной системы на процедуре умножения матриц (эксперимент и расчёт по формуле Амдаля)

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

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

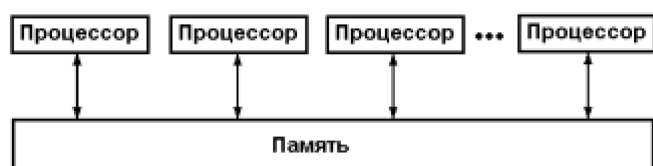
1. Что такое суперкомпьютер? Чем он (*количественно* и *качественно*) отличается от персональной ЭВМ? Почему суперкомпьютер выполняется с использованием технологий многомашинных комплексов и/или многопроцессорных систем? В каких областях человеческой деятельности необходимо применение суперкомпьютеров?
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?

2 Принципы построения многопроцессорных вычислительных систем

2.1 Архитектура многопроцессорных вычислительных систем

Архитектура параллельных компьютеров развивалась практически с самого начала их создания и применения, причём в самых различных направлениях. Наиболее общие положения приводят к двум классам – *компьютеры с общей памятью* и *компьютеры с распределённой памятью* [1,2].

Компьютеры с общей памятью (*мультипроцессоры, компьютеры с разделяемой памятью*) состоят из нескольких (одинаковых, возможно) процессоров, имеющих равноприоритетный доступ к общей памяти с единым адресным пространством (рис.7а). Типичный пример такой архитектуры – компьютеры класса SMP (*Symmetric Multi Processors*), включающие несколько процессоров, но одну память, комплект устройств ввода/вывода и операционную систему ("*symmetric*" означает возможность каждого процессора выполнять то же, что и любой другой и получать равноприоритетный доступ к общей памяти; 2÷4-процессорные серверы SMP-архитектуры легко приобрести почти в любом компьютерном магазине). Достоинством компьютеров с общей памятью является (относительная) *простота программирования параллельных задач* (нет необходимости заниматься организацией пересылок сообщений между процессорами с целью обмена данными), минусом – *недостаточная масштабируемость* (при увеличении числа процессоров возрастает конкуренция за доступ к общим ресурсам – в первую очередь памяти, что ограничивает суммарную производительность системы). Реальные SMP-системы содержат обычно не более 32÷64 процессоров, для дальнейшего наращивания вычислительных мощностей подобных систем используется NUMA-технология (см. ниже). Отечественной SMP-разработкой является ЭВМ "Эльбрус-1" (*В.С.Бурцев*, 1980) с быстродействием до 15 млн. оп./с. (125 млн. оп./с. для "Эльбрус-2"); компьютеры этой модели до сих пор обес-



а)

б)

Рисунок 7 — Параллельные компьютеры: с общей памятью - а) и с распределенной памятью – б)

печивают функционирование систем противоракетной обороны и космических войск России.

В компьютерах с распределённой памятью *(мультикомпьютерные системы)* каждый *вычислительный узел* (ВУ) является полноценным компьютером и включает процессор, память, устройства ввода/вывода, операционную систему и др. (рис.7б). Типичный пример такой архитектуры – компьютерные системы класса MPP (*Massively Parallel Processing*), в которых с помощью некоторой коммуникационной среды объединяются однородные вычислительные узлы. Достоинством компьютеров с распределённой памятью является (почти неограниченная) *масштабируемость*, недостатками – необходимость применения специализированных программных средств (*библиотек обмена сообщениями*) для осуществления обменов информацией между вычислительными узлами. Для МВС с общей и распределённой памятью используются термины *сильно- и слабосвязанные машины* соответственно.

Как было показано, производительность каналов обмена сильно влияет на возможность эффективного распараллеливания, причем это важно для обоих рассмотренных архитектур. Простейшей *системой коммуникации* является *общая шина* (рис.8а), которая связывает все процессоры и память, однако даже при числе устройств на шине больше нескольких десятков производительность шины катастрофически падает вследствие взаимного влияния и конкуренции устройств за монопольное владение шиной при обменах данными.

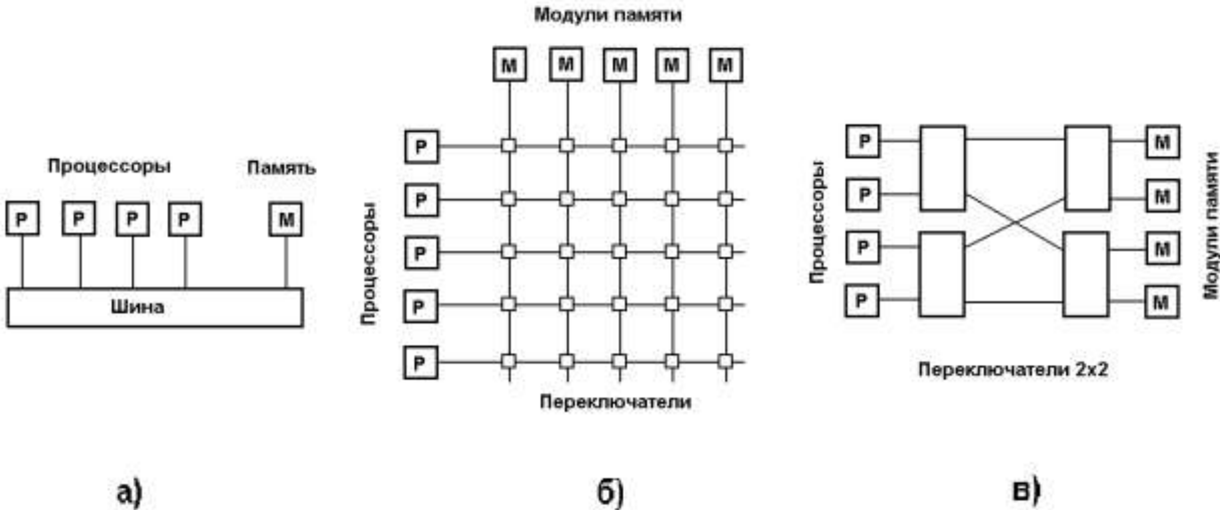


Рисунок 8 — Многопроцессорные системы с общей шиной - а), с матричным коммутатором – б) и с каскадированием коммутаторов (*Омега-сеть*) – в)

При построении более мощных систем (коммутация десятков/сотен устройств) используются более изощренные подходы. Эффективной (но дорогой вследствие значительного объема оборудования) является схема *матричной коммутации* (рис.8б), при которой устройства связываются между собой двунаправленными переключателями, разрешающими или запрещающими пе-

редачу информации между соответствующими модулями. Альтернативой является *каскадирование переключателей*; например, по схеме Омега-сети, рис.8в. При этом каждый переключатель может соединять любой из двух своих входов с любым из двух выходов, для соединения n процессоров с n блоками памяти в этом случае требуется $n \times log_2 n/2$ переключателей (вместо n^2 по схеме рис.8б). Недостаток схем с каскадированием коммутаторов – задержки срабатывания переключателей (на схеме рис.8в при произвольном соединении сигнал вынужденно проходит последовательно через $log_2 n$ переключателей вместо единственного по схеме рис.8б).

Для систем с распределённой памятью используются практически все мыслимые варианты соединений (см. рис.9), при этом параметром качества с точки зрения скорости передачи сообщений служит величина *средней длины пути*, соединяющего произвольные процессоры; при этом имеется в виду именно *физический путь*, т.к. реализовать логическую топологию (программными средствами) не представляет трудностей.

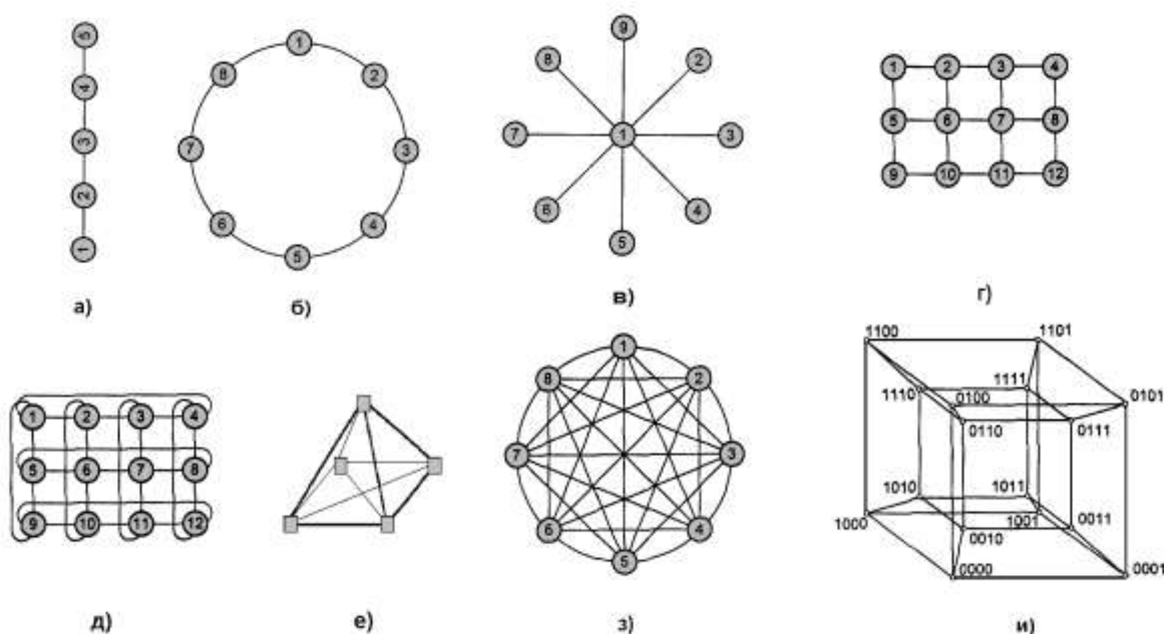


Рисунок 9 — Варианты топологий связи процессоров в многопроцессорных вычислительных системах

Простейшая *линейная топология* (рис.9а) удовлетворительно соответствует многим алгоритмам, для которых характерна связь лишь соседних процессов между собой (одномерные задачи математической физики и многомерные, сводимые к одномерным); недостаток – невозможность передачи сообщений при разрыве в любом месте. Уменьшение вдвое средней длины пути и повышение надежности связи (при нарушении связи сообщения могут быть переданы в противоположном направлении, хотя и с меньшей скоростью) достигается соединение первого узла с последним – получается *топология*

"кольцо" (рис.9б). *Топология "звезда"* (рис.9в) максимально отвечает распределению нагрузки между процессами, характерной для систем "клиент/сервер" (главный узел "раздаёт" задания и "собирает" результаты расчётов, при этом подчинённые узлы друг с другом взаимодействуют минимально).

Топология *"решётка"* (рис.9г) использовалась ещё в начале 90-х г.г. при построении суперкомпьютера Intel Paragon на основе процессоров i860 [1]; нахождение минимального пути передачи данных между процессорами A и B (координаты [1,1,3] и [2,2,3] соответственно) для топологии "трехмерная решетка" иллюстрировано рис.10. Топология *"двумерный тор"* (рис.9д) расширяет "двумерную решетку" дополнительными связями, снижающими длину среднего пути (само собой, возможен и *"трехмерный тор"*) и характерна для сетевой технологии SCI (*Scalable Coherent Interface*), предлагаемой фирмой Dolphin Interconnect Sol. (http://www.dolphinics.com). Применяется (рис.9е) характеризующаяся наличием связи каждого процессора с каждым трехмерная топология "клика" (*Clique*). На рис.9з) приведён общий вид топологии полной связи всех процессоров между собой; такая топология характеризуется наименьшей (тождественно единичной) длиной среднего пути между процессорами, однако *аппаратно практически нереализуема* при значительном числе процессоров вследствие катастрофического роста числа связей (однако часто применяется в качестве *виртуальной топологии,* реализуемый на логическом уровне программными средствами).

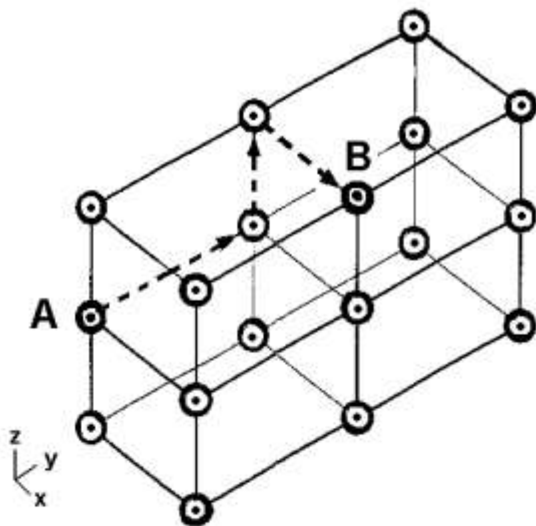


Рисунок 10 — Нахождение минимального пути для передачи сообщений между процессорами в топологии 'трехмерная решетка'

Для топологии *"гиперкуб"* (рис.9и) характерна сокращенная длина среднего пути и близость к структурам многих алгоритмов численных расчетов, что обеспечивает высокую производительность. N-мерный гиперкуб содержит 2^N процессоров. Двухмерный гиперкуб - это квадрат, трехмерный гиперкуб образует обычный куб, а четырёхмерный гиперкуб представляет собой куб в кубе. Для семейства суперкомпьютеров nCube 2 (известная тесным сотрудничеством в области параллельных баз данных с ORACLE фирма nCUBE Corp., сейчас подразделение C-COR, http://www.c-cor.com) гиперкуб максимальной размерности 13 содержит 8192 процессора, в системе nCube 3 число процессоров может достигать 65536 (16-мерный гиперкуб).

В качестве основных характеристик топологии сети передачи данных часто используются следующие показатели:

- Диаметр определяется как максимальное расстояние (обычно кратчайших путь между процессорами) между двумя процессорами в сети, эта величина характеризует максимально необходимое время для передачи данных между процессорами (время передачи в первом приближении прямо пропорционально длине пути).
- Связность (*connectivity*) - показатель, характеризующий наличие разных маршрутов передачи данных между процессорами сети; конкретный вид показателя может быть определён, напр., как минимальное количество дуг, которое надо удалить для разделения сети передачи данных на *две несвязные области*.
- Ширина бинарного деления (*bisection width*) - показатель, определяемый как минимальное количество дуг, которое надо удалить для разделения сети передачи данных на две *несвязные области одинакового размера*.
- Стоимость определяется, напр., как общее количество линий передачи данных в многопроцессорной вычислительной системе.

Естественным представляется объединение преимуществ систем с общей (относительная простота создания параллельных программ) и распределенной памятью (высокая масштабируемость); решением этого вопроса явилось создание компьютеров с архитектурой NUMA (*Non Uniform Memory Access*); в этом смысле классические SMP-компьютеры обладают архитектурой UMA (*Uniform Memory Access*). При этом применяется механизм (обычно аппаратного уровня – что быстрее), позволяющий пользовательским программам рассматривать всю (физически) распределенную между процессорами память как единое адресное пространство. Примерами NUMA-компьютеров является построенная еще в 70-х г.г. и содержащая объединенный межкластерной шиной набор кластеров система Cm* и объединяющий 256 процессоров комплекс BBN Butterfly (1981, фирма BBN Advanced Computers).

Недостатками NUMA-компьютеров является всё же значительная разница времени обращения к собственной (локальной) памяти данного процессора и памяти сторонних процессоров, а также *проблема кэша* (*cache coherence problem*) - в случае сохранения процессором $П_1$ некоего значения в ячейке N_1 при последующей попытке прочтения данных из той же ячейки N_1 процессором $П_2$ последний получит значение из кэша процессора $П_1$, которое может не совпадать с истинным значением переменной в ячейке N_1, если кэш процессора $П_1$ ещё не "сброшен" в память (о чем процессор $П_2$ "знать" не обязан).

Для решения проблемы когерентности (*соответствия, одинаковости*) кэша предложена и реализована архитектура ccNUMA (*cache coherent NUMA*), позволяющая (прозрачными для пользователя) средствами достигать соответствия кэшей процессоров (что требует дополнительных ресурсов и, соответственно, снижает производительность). Типичным образцом ccNUMA-машины является HP Superdome ($2 \div 64$ суперскалярных процессора P-8600/P-8700 с возможностью дальнейшего наращивания, 256 Гбайт оперативной памяти, пиковая производительность 192 Гфлопс в 64-процессорном варианте, http://www.hp.com/go/superdome).

В системах с архитектурой COMA (*Cache-Only Memory Architecture*) на вычислительных узлах предусмотрены дополнительные (построенные подобно структуре кэш-памяти) и называемые AM (*Attraction Memory, "притягивающая память"*) локальные модули памяти. При обращении к фрагменту памяти стороннего процессора поступивший фрагмент размещается как в кэш-памяти запрашивающего процессора, так и в его AM (ранее размещенный фрагмент при этом может быть выгружен); при неудачном поиске (*промахе*) в кэш-памяти контроллер памяти просматривает AM и, если нужного фрагмента нет и там, инициирует запрос на его копирование из локальной памяти соответствующего вычислительного узла.

2.2 Распределение вычислений и данных в многопроцессорных вычислительных системах с распределённой памятью

В случае наличия в составе многопроцессорной вычислительной системы (не-NUMA структуры) вычислительных узлов с локальной оперативной памятью кроме распределения частей вычисления по отдельным ВУ важно *рациональным образом* распределить по имеющимся ВУ данные (например, блоки обрабатываемых матриц значительной размерности). Дело в том, что затраты времени на обмен данными между обрабатывающими эти данные ВУ и ВУ, хранящими эти данные в своей локальной ОП, может на порядки замедлить процесс вычислений.

Ясно, что расположение большого пула данных на одном (например, первом в списке) ВУ вряд ли целесообразно вследствие неизбежной значительной потери времени на организацию пересылок отдельных блоков этих данным обрабатывающим ВУ (не говоря уже о возможной нехватке ОП). С другой стороны, чисто формальное разбиение данных на равное числу ВУ число блоков чревато тем же.

Рациональное распределение данных по локальным ОП вычислительных узлов должно совершаться с *учётом частоты обращения каждого ВУ к каждому блоку данных, расположенных на соответствующих ВУ* при стремлении к минимизации числа обменов, что требует определения *тонкой информационной структуры алгоритма*.

Казалось бы, в общем случае возможно построение некоей *функции трудоёмкости* (например, в смысле времени) вычислений, учитывающей как ресурсные затраты на собственно вычисления так и трудоёмкость обменов при заданном распределении данных по ВУ и дальнейшей минимизации этой функции по параметру (параметрам) распределения данных; причём само распределение может являться изменяемым (динамическим). Реально построение подобной функции затруднено вследствие необходимости количественного (для конкретной вычислительной системы) определения временных параметров выполнения операций (в т.ч. сетевых, с учётом реальной зависимости скорости обмена от длины сообщения, тем более при учёте *неоднородности* вычислительных узлов) и выявлении значимых параметров оптимизации. Претендентом на роль подобной функции может служить, напр., вышеописанный *сетевой закон Амдаля*. Некоторые системы программирования (напр., mpC, см. подраздел 4.1) обладают встроенными средствами для осуществления балансировки производительности.

В стандартных пакетах решения задач линейной алгебры ScaLAPACK и AZTEC используются основанные на теоретическом анализе и долговременной практике методы распределения данных по вычислительным узлам (включая специальные технологии, напр., *блочное распределение с теневыми гранями*, [3,7]).

2.3 Классификация параллельных вычислительных систем

Классификация архитектур вычислительных систем преследует цели выявление как основных факторов, характеризующих каждую архитектуру, так и взаимосвязей параллельных вычислительных структур [1,2,6]. Попытки выстроить (возможно непротиворечивые) классификации начались после опубликования в конце 60-х г.г. М.Флинном первого варианта классификации.

Классификация М.Флинна (M.Flynn). Основой этой классификации (1966 г.) является понятие *потока* как последовательности команд (*Instruction stream*) и данных (*Data stream*), обрабатываемая процессором.

SISD (*Single Instruction stream / Single Data stream*) – одиночный поток команд и одиночный поток данных; к SISD-классу относятся классические последовательные (фон-Неймановского типа) машины (напр., всем известная PDP-11). В таких машинах имеется только один поток (последовательно обрабатываемых) команд, каждая из которых инициирует одну скалярную операцию. К этому классу некоторые исследователи относят и машины с технологией конвейерной обработки.

SIMD (*Single Instruction stream / Multiple Data stream*) – одиночный поток команд наряду со множественным потоком данных. При этом сохраняется один поток команд, но включающий векторные команды, выполняющие одну арифметическую операцию сразу над многими данными (напр., отдельными

элементами вектора). Выполнение векторных операций может производиться матрицей процессоров (как в машине ILLIAC IV) или конвейерным способом (Cray-1). Фактически микропроцессоры Pentium VI и Xeon с набором инструкций MMX, SSE, SSE2 являются однокристалльными SIMD-системами [4]. Из отечественных SIMD-систем следует назвать ПС-2000 (Институт проблем управления РАН, *И.В.Прангишвили*, 1972÷1975) – высокопараллельная компьютерная система для обработки информации с производительностью до 200 млн.оп./с.

В начале XXI века популярными стали арифметические ускорители на основе графических плат фирмы nVIDIA и связанная с ними технология разработки параллельных программ CUDA [5]. Выросшие из графических карт арифметические ускорители (GPU - Graphics Processing Unit) этой фирмы имеют сотни/тысячи вычислительных ядер (аппаратно поддерживающих арифметику вещественных чисел двойной точности), выполняющих одинаковый код над различными частями блоками данных; при этом достигается огромное (многие сотни раз) ускорение вычислений (по сравнению с процессорами общего назначения). Появление GPU вызвало к жизни т.н. *гибридные кластеры*, каждый вычислительный узел которых включает (наряду с процессором/процессорами общего назначения) один/несколько CPU [3]. Больщим достоинством технологии CUDA является простота программирования (используется классический язык программирования, дополненный несколькими ключевыми словами). Архитектура GPU хорошо подходит для решения задач линейного программирования и многих иных численных методов.

MISD (*Multiple Instruction stream / Single Data stream*) – множественный поток команд и одиночный поток данных. Архитектура подразумевает наличие многих процессоров, обрабатывающих один и тот же поток данных; считается, что таких машин не существует (хотя с некоторой натяжкой к этому классу можно отнести конвейерные машины).

MIMD (*Multiple Instruction stream / Multiple Data stream*) – множественные потоки как команд, так и данных. К классу MIMD принадлежат машины двух типов: с управлением от потока команд (IF - *instruction flow*) и управлением от потока данных (DF - *data flow*); если в компьютерах первого типа используется традиционное выполнение команд *последовательно их расположения* в программе, то второй тип предполагает активацию операторов *по мере их текущей готовности* (подробнее см. подраздел 2.5 данной работы). Класс предполагает наличие нескольких объединённых в единый комплекс процессоров, работающий каждый со своим потоком команд и данных. Классический пример - система Denelcor HEP (*Heterogeneous Element Processor*); содержит до 16 процессорных модулей (*PEM, Process Execution Module*), через многокаскадный переключатель связанных со 128 модулями памяти данных (*DMM, Data Memory Module*), причём все процессорные модули могут работать независимо друг от друга со своими потоками команд, а каждый

процессорный модуль может поддерживать до 50 потоков команд пользователей. Отечественный представитель машины MIMD-архитектуры – вычислительные системы ЕС-2704, ЕС-2727 (конец 80-х г.г., НИЦЭВТ), позволяющий одновременно использовать сотни процессоров.

Классификация Р.Хокни (R.Hockney). В этом случае классифицируются (более подробно) компьютеры класса MIMD по Флинну [1]. Основа классификации - выделение способов реализации множественного потока команд: единым работающим в режиме разделения для каждого потока конвейерным устройством или несколькими устройствами, обрабатывающими каждое свой поток. Второй вариант представлен двумя реализациями – с переключателями, дающими возможность осуществить прямую связь между всеми процессорами и системами, в которых прямая связь каждого процессора возможна только с ближайшими соседями (доступ к удаленным процессорам осуществляется специальной системой маршрутизации сообщений); каждая реализация имеет подклассы.

Классификация Т.Фенга (T.Feng, 1972). Каждая вычислительная система описывается парой чисел (n,m), где n - число параллельно обрабатываемых единой командой бит в машинном слове, m - число одновременно обрабатываемых слов; произведение $P=n\times m$ суть *максимальная степень параллелизма вычислительной системы* (величина, пропорциональная пиковой производительности). Все вычислительные системы т.о. делятся на:

- разрядно-последовательные, пословно-последовательные ($n=m=1$),
- разрядно-параллельные, пословно-последовательные ($n>1$, $m=1$),
- разрядно-последовательные, пословно параллельные ($n=1$, $m>1$),
- разрядно-параллельные, пословно-параллельные ($n>1$, $m>1$)

Классификацию Фенга считают устаревшей и далеко неполно учитывающей специфику современных многопроцессорных вычислительных систем, однако её преимущество состоит во введении *единообразного числового параметра* для сравнения вычислительных систем (далее эта идея развита в классификации Хандлера).

Согласно *классификации В.Хандлера (W.Handler)* каждая вычислительная система характеризуется тройкой чисел (в расширенной варианте – тройкой групп чисел и арифметико-логических выражений), количественно характеризующей возможности вычислительной системы по параллельной и конвейерной обработке данных (при этом не уточняется тип связей между процессорами и блоками памяти - априорно предполагается, что коммуникационная сеть адекватно выполняет свои функции). Тройка определяет три уровня обработки данных:

- уровень выполнения программы - на основе данных счетчика команд и дополнительных регистров устройство управления (УУ) производит выборку и дешифрацию инструкций,
- уровень выполнения команд - арифметико-логическое устройство (АЛУ) исполняет очередную команду, выданные УУ,
- уровень битовой обработки - элементарной логические схемы (ЭЛС) процессора выполняют операции над отдельными битами данных, составляющих машинное слово

При обозначении числа УУ через k, в каждом из них число АЛУ символом как d и число ЭЛС в каждом АЛУ через w, то конкретная вычислительная система кодируется тройкой чисел (k,d,w), например:

- IBM 701 $\rightarrow$ (1,1,36)
- SOLOMON $\rightarrow$ (1,1024,1)
- ILLAIC IV $\rightarrow$ (1,64,64)

Расширения системы классификации Хандлера позволяют более точно конкретизировать вычислительные системы (учесть число ступеней конвейера, альтернативные режимы работы вычислительной системы и др., [1]).

Классификация Д.Скилликорна (D.Skillicorn, 1989) предполагает описание архитектуры компьютера как абстрактной структуры, состоящей из компонент 4 типов (процессор команд, процессор данных, иерархия памяти, коммутатор):

- процессор команд (IP, *Instruction Processor*) - функциональное устройство, работающее как интерпретатор команд (в системе может отсутствовать),
- процессор данных (DP, *Data Processor*) - функциональное устройство, выполняющее функции преобразования данных в соответствии с арифметическими операциями,
- иерархия памяти (IM, *Instruction Memory*; DM, *Data Memory*) - запоминающее устройство, хранящее пересылаемые между процессорами данные и команды,
- переключатель – (абстрактное) устройство, обеспечивающее связь между процессорами и памятью (Скилликорном определены четыре типа переключателей).

Классификация Д.Скилликорна состоит из двух уровней. На первом уровне она проводится на основе восьми характеристик:

1. Количество процессоров команд (IP).
2. Число запоминающих устройств (модулей памяти) команд (IM).

3. Тип переключателя между IP и IM.
4. Количество процессоров данных (DP).
5. Число запоминающих устройств (модулей памяти) данных (DM).
6. Тип переключателя между DP и DM.
7. Тип переключателя между IP и DP.
8. Тип переключателя между DP и DP.

На втором уровне уточняется описание первого уровня путем добавления возможности конвейерной обработки команд и данных в процессорах.

Скилликорном объявлено, что цель классификации архитектур заключается в выпуклости показа, за счёт каких особенностей структуры достигается повышение производительности различных вычислительных систем; это важно не только для выявления направлений совершенствования аппаратной части компьютеров, но и для создания эффективных параллельных программ.

Дополнительные данные по классификации архитектур вычислительных систем (включая и не цитируемые здесь классификации) см. http://parallel.ru/computers/taxonomy/index.htm.

2.4 Многопроцессорные вычислительные системы с распределённой памятью

С последнего десятилетия 20 века наблюдается тенденция к монополизации архитектур супер-ЭВМ системами с распределённой памятью, причём в качестве процессоров на вычислительных узлах все чаще применяются легкодоступные готовые устройства. Основными преимуществами таких систем является огромная масштабируемость (в зависимости от класса решаемых задач и бюджета пользователь может заказать систему с числом узлов от нескольких десятков до тысяч); что привело к появлению нового названия для рассматриваемых систем – *массивно-параллельные компьютеры* (вычислительные системы архитектуры MPP - *Massively Parallel Processing*).

Дэнни Хиллис, основатель компании Thinking Machines, в 1982 г. продемонстрировал первый суперкомпьютер с массивной параллельной обработкой (*massive multiprocessing*). Система, получившая название Connection Machine (CM-1), была оснащена 64 тыс. процессоров, каждый из который имел собственную память. На своей первой презентации CM-1 выполнила сканирование 16 тыс. статей со сводками последних новостей за 1/20 сек. и разработала интегральную схему процессора с 4 тыс. транзисторов за три минуты.

Выпуклыми представителями MPP-систем являются суперкомпьютеры серии Cray T3 (процессоры Alpha, топология "трехмерный тор", Cray Inc., http://www.cray.com), IBM SP (http://www.ibm.com, http://www.rs6000.ibm.com/sp.html, соединённые иерархической системой высокопроизводительных коммутаторов процессоры PowerPC, P2SC, Power3), In-

tel Paragon (http://www.ssd.intel.com, двумерная прямоугольная решётка процессоров i860) и др.

MPP-система состоит из однородных *вычислительных узлов* (ВУ), включающих:

- один (иногда несколько) центральных процессоров (обычно архитектуры RISC - *Reduced Instruction Set Computing*, для которой характерно длинное командное слово для задания операций, сокращённый набор команд и выполнение большинства операций за один такт процессора),
- локальную память (причём прямой доступ к памяти других узлов невозможен),
- коммуникационный процессор (или сетевой адаптер),
- жёсткие диски (необязательно) и/или другие устройства ввода/вывода

К системе добавляются специальные узлы ввода-вывода и управляющие узлы. Вычислительные узлы связаны некоторой *коммуникационной средой* (высокоскоростная сеть, коммутаторы и т.п.).

Техническое обслуживание многопроцессорных систем является непростой задачей – при числе ВУ сотни/тысячи неизбежен ежедневный отказ нескольких из них; *система управления ресурсами* (программно-аппаратный комплекс) массивно-параллельного компьютера обязана обрабатывать подобные ситуации в обход катастрофического общего рестарта с потерей контекста исполняющихся в данный момент задач.

2.4.1 Массивно-параллельные суперкомпьютеры серии CRAY T3

Основанная Сеймуром Крэем (*Seymour Cray*, $1925 \div 1996$ г.г.) в 1972 г. фирма Cray Research Inc. (сейчас Cray Inc.), прославившаяся разработкой векторного суперкомпьютера Cray 1 (1975 г., оперативная память 64 Мбит, пиковая производительность 160 Мфлопс), в $1993 \div 1995$ г.г. выпустила модели Cray T3D/T3E, полностью реализующие принцип систем с массовым параллелизмом (систем MPP-архитектуры). В максимальной конфигурации эти компьютеры объединяют $32 \div 2048$ процессоров DEC Alpha 21064/150 MHz, 21164/600 MHz, 21164A/675 MHz (в зависимости от модели), вся предварительная обработка и подготовка программ (напр., компиляция) выполняется на управляющей машине (хост-компьютере).

Разработчики серии Cray T3D/T3E пошли по пути создания *виртуальной общей памяти*. Каждый процессор может обращаться напрямую только к своей локальной памяти, но все узлы используют единое адресное пространство. При попытке обращения по принадлежащему локальной памяти другого процессора адресу генерируется специализированное аппаратное прерывание и операционная система выполняет пересылку страницы с одного узла на

другой, причём вследствие чрезвычайно высокого быстродействия (дорогостоящей вследствие этого) коммуникационной системы (пиковая скорость передачи данных между двумя узлами достигает 480 Мбайт/сек при латентности менее 1 мксек) этот подход в целом оправдан. Однако замечен резко снижающий производительность эффект "пинг-понга" – в случае попадания на одну страницу переменных, модифицируемых несколькими процессорами, эта страница непрерывно мигрирует между узлами. Вычислительные узлы выполняют программы пользователя *в монопольном режиме* (однозадачный режим).

Конкретное исполнение компьютеров серии Cray T3 характеризуется тройкой чисел, напр., 24/16/576 (управляющие узлы/узлы операционной системы/вычислительные узлы); при используемой топологии "трехмерный тор" каждый узел (независимо от его расположения) имеет шесть непосредственных соседей. При выборе маршрута между двумя узлами A и B (3D-координаты которых суть [1,2,4] и [2,1,2], рис.11) сетевые машрутизаторы, начиная процесс с начальной вершины A, сначала выполняют смещение по координате X таким образом, пока координаты очередного узла связи и узла B не станут равными (это узел [2,2,4]); затем аналогичные действия выполняются по координате Y и далее по Z (физически подобная маршрутизация происходит одновременно по всем трём координатам). Смещения могут быть и отрицательными, при отказе одной или нескольких связей возможен их обход. Другой интересной особенностью архитектуры Cray T3 является поддержка *барьерной синхронизации* – аппаратная организация ожидания всеми процессами определенной точки в программе, при достижении которой возможна дальнейшая работа [1]. Компьютеры серии T3E демонстрировали производительность $1{,}8 \div 2{,}5$ Тфлопс (на 2048 микропроцессорах Alpha/600 MHz) [4].

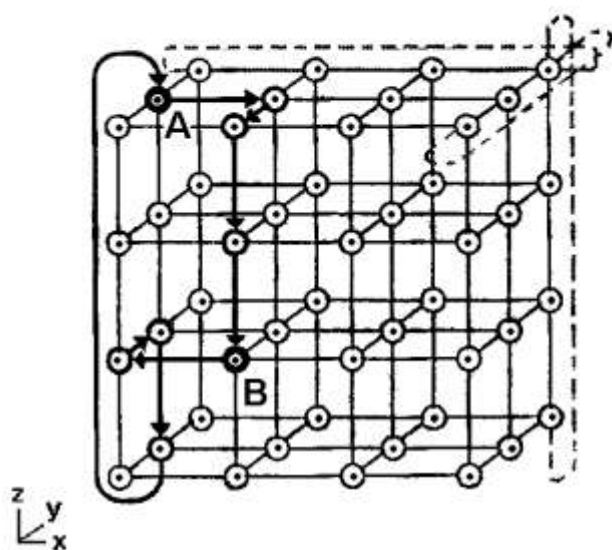


Рисунок 11 — Коммуникационная решетка ‘трехмерный тор’ компьютера Cray T3E

Дальнейшим развитием линии массивно-параллельных компьютеров Cray T3 является суперкомпьютер Cray XT3. Каждый вычислительный узел Cray XT3 включает процессор AMD Opteron, локальную память ($1 \div 8$ Гбайт) и обеспечивающий связь к коммуникационному блоку Cray SeaStar канал

HyperTransport (http://cray.com/products/xt3), имеющий пиковую производительность (для AMD Opteron 2,4 GHz) от 2,6 Тфлопс (548 процессоров, оперативная память 4,3 Тбайт, топология $6 \times 12 \times 8$) до 147 Тфлопс (30'508 процессоров, 239 Тбайт, топология $40 \times 32 \times 24$). Cray XT3 работает под управлением ОС UNICOS/lc, позволяющей объединять до 30'000 вычислительных узлов, применяются компиляторы Fortran'77/90/95 и C/C++, коммуникационные библиотеки MPI (*Message Passing Interface* с поддержкой стандарта MPI 2.0) и ShMem (разработанная Cray Research Inc. библиотека для работы с общей памятью), стандартные библиотеки численных расчётов.

Несмотря на достигнутые столь высокие результаты в области MPP-систем фирма Cray Inc. производит векторно-конвейерный компьютеры (модель C90 с пиковой производительностью 16 Гфлопс при задействовании всех 16 процессоров и её развитие - модель T90 с производительностью до 60 Гфлопс при 32 процессорах), причём эти модели могут быть объединены в MPP-систему. Производительность каждого процессора компьютера Cray SV1 достигает 4 Гфлопс (общая пиковая производительность системы 32 Гфлопс), общее число процессоров может достигать 1000. Векторно-конвейерные процессоры являются основой упоминавшейся выше массивно-параллельной вычислительной системы "the Earth Simulator" производительностью 40 Тфлопс (http://www.es.jamstec.go.jp/esc/eng/index.html).

2.4.2 Кластерные системы класса BEOWULF

Запредельная стоимость промышленных массивно-параллельных компьютеров не давала покоя специалистам, желающим применить в своих исследованиях вычислительные системы сравнимой мощности, но не имеющих возможностей приобрести промышленные супер-ЭВМ. Поиски в этом направлении привели к развитию *вычислительных кластеров* (не путать с *кластерами баз данных* и *WEB-серверов*); технологической основой развития кластеризации стали широкодоступные и относительно недорогие микропроцессоры и коммуникационные (сетевые) технологии, появившиеся в свободной продаже в 90-х г.г.

Вычислительный кластер представляет собой совокупность вычислительных узлов (от десятков до десятков тысяч), объединенных высокоскоростной сетью с целью решения единой вычислительной задачи. Каждый узел вычислительного кластера представляет собой фактически ПЭВМ (часто двух- или четырех- процессорный/ядерный SMP-сервер), работающую со своей собственной операционной системой (в подавляющем большинстве Linux[*]); объ-

---

* Windows-кластеры значительной мощности до настоящего времени остаются экзотикой в силу известных причин (несмотря на активно продвигаемые MS решения класса

единяющую сеть выбирают исходя из требуемого класса решаемых задач и финансовых возможностей, практически всегда реализуется возможность удалённого доступа на кластер посредством InterNet.

Вычислительные узлы и управляющий компьютер обычно объединяют (минимум) две (независимые) сети – *сеть управления* (служит целям управления вычислительными узлами) и (часто более производительная) *коммуникационная сеть* (непосредственный обмен данными между исполняемыми на узлах процессами), дополнительно управляющий узел имеет выход в InterNet для доступа к ресурсам кластера удалённых пользователей, файл-сервер (в несложных кластерах его функции выполняет управляющий компьютер) выполняет функции хранения программ пользователя (рис.12). Администрирование кластера осуществляется с управляющей ЭВМ (или посредством удаленного доступа), пользователи имеют право доступа (в соответствие с присвоенными администратором правами) к ресурсам кластера исключительно через управляющий компьютер.

Одним из первых кластерных проектов явился проект BEOWULF (http://www.beowulf.org, название дано в честь героя скандинавской саги). Проект БЕОВУЛЬФ был заложен в созданном на основе принадлежащей NASA организации GSFC (*Goddard Space Flight Center*, http://www.gsfc.nasa.gov) исследовательском центре CESDIS (*Center of Excellence in Space Data and Information Sciences*) в 1994 г. и стартовал сборкой в GSFC 16-узлового кластера (на процессорах 486DX4/100 MHz, 16 Mb памяти, 3 сетевых адаптера на каждом узле и 3 параллельных 10 Mbit Ethernet-кабелей); вычислительная система предназначалась для проведения работ по проекту ESS (*Earth and Space Sciences Project*, http://sdcd.gsfc.nasa.gov/ESS/overview.html).

Позднее в подразделениях NASA были собраны другие модели BEOWULF-подобных кластеров: напр., theHIVE (*Highly-parallel Integrated Virtual Environment*) из 64 двухпроцессорных (Pentium Pro/200 MHz, 4 Gb памяти и 5 коммутаторов Fast Ethernet в каждом) узлов (общая стоимость 210 тыс. $US). Именно в рамках проекта Beowulf были разработаны драйверы для реализации режима Channel Bonding (см. ниже).

"Беовульф" – типичный образец многопроцессорной системы MIMD (*Multiple Instruction – Multiple Data*), при этом одновременно (и относительно независимо друг от друга) выполняются несколько программных ветвей, в определённые промежутки времени обменивающиеся данными. Многие последующие разработки во всех странах мира фактически являются кланами Beowulf.

Windows Compute Cluster Server - WCCS). Кроме того, автор разделяет известное правило '…не стоит класть все *имеющиеся* яйца в единую корзину'.

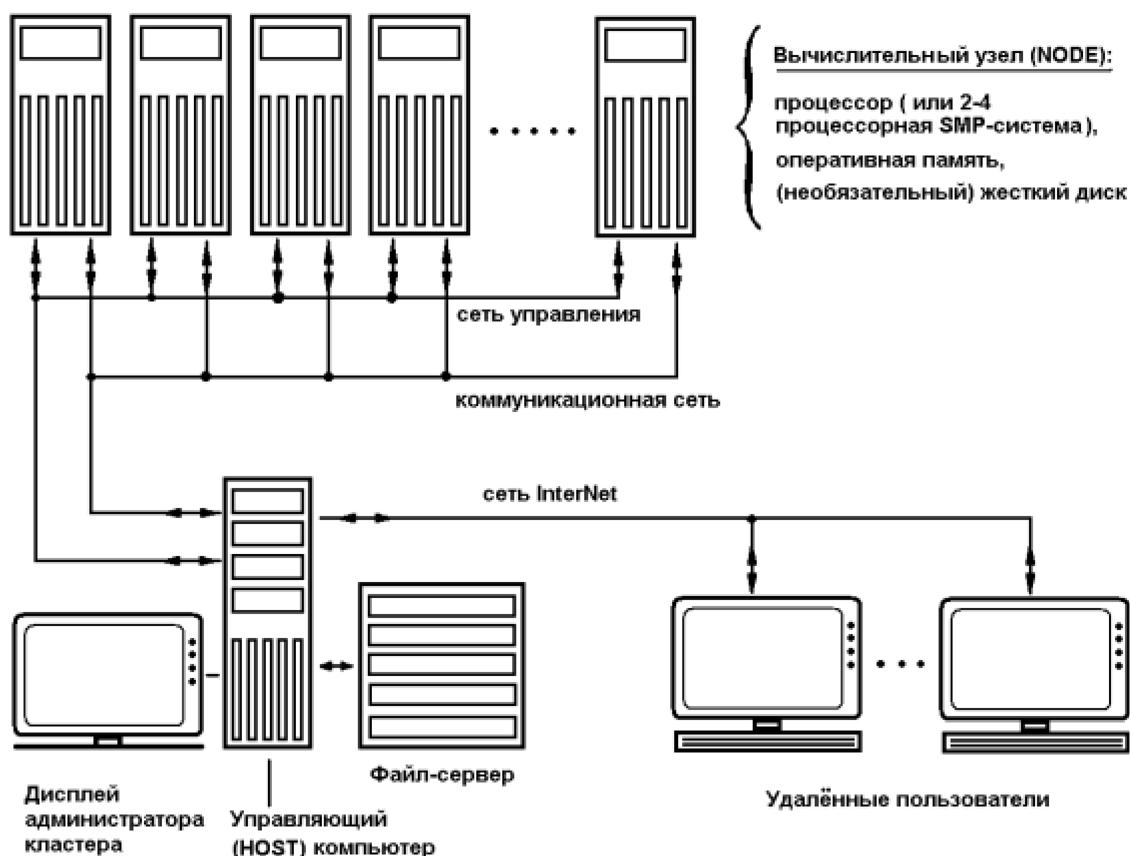


Рисунок 12 — Укрупнённая схема вычислительного кластера

В 1998 г. в национальной лаборатории Лос-Аламос астрофизик Michael Warren (http://qso.lanl.gov/~msw) с сотрудниками группы теоретической астрофизики построили вычислительную систему Avalon (http://cnls.lanl.gov/avalon), представляющую Linux-кластер на процессорах DEC Alpha/533 MHz. Первоначально Avalon состоял из 68 процессоров, затем был расширен до 140, в каждом узле установлено 256 MB оперативной памяти, EIDE-жесткий диск 3,2 Gb, сетевой адаптер фирмы Kingston (общая стоимость узла – 1’700 $US). Узлы соединены с помощью 4-х коммутаторов Fast Ethernet (36 портов каждый) и центрального 12-портового коммутатора Gigabit Ethernet фирмы 3Com (http://www.3com.com). Avalon обошелся в 323 тыс. $US и занял 114 место в 12-й редакции "Top-500" (1998); причем 152-процессорный IBM SP2 занял 113-е место! 70-процессорный Avalon по многим тестам показал такую же производительность, как 64-процессорная система SGI Origin2000 / 195 MHz (http://www.sgi.com/origin/2000) стоимостью выше 1 млн. $US.

Из других смежных проектов следует назвать [1] Berkeley NOW (*Network Of Workstations*, http://now.cs.berkeley.edu), Condor (*High Throughput Computing*, http://www.cs.wisc.edu/condor), MOSIX (программное обеспечение поддержки кластерных вычислений в Linux, Hebrew University, Израиль, http://www.mosix.cs.huji.ac.il), T-Система (программирование на языке T++ и

среда времени выполнения динамически распараллеливаемых программ (*Институт Программных Систем РАН*, Переславль-Залесский, http://www.botik.ru/PSI/PSI.koi8.html).

Типичным образцом массивно-параллельной кластерной вычислительной системы являются МВС-1000М (коммуникационная сеть – Myrinet 2000, скорость обмена информацией 120÷170 Мбайт/сек, http://www.myri.com, вспомогательные – Fast и Gigabit Ethernet) и МВС-15000ВС (см. подраздел 1.1 и http://www.jscc.ru).

Триумфальный успех, явная дешевизна и относительная простота изготовления привели к возникновению в конце XX века в американских околоправительственных кругах "синдрома Беовульфа". В самом деле, вычислительный комплекс типа "Беовульф" с достаточной для решения военных и криптографических задач мощностью может быть собран в любой стране мира (включая т.н. "государства – очаги международного терроризма"). Ситуация усугубляется тем, что используемые при сборке "Беовульфа" комплектующие для персональных ЭВМ не попадают под экспортные ограничения правительства США (а при необходимости могут быть приобретены и в третьих странах).

Однако специалисты считают, что с концом 1-го десятилетия XXI века эпоха кластеров подходит к естественному концу вследствие неадекватности традиционной фон-Неймановской архитектуре современным вычислительным требованиям и коренного изменения коммуникационных технологий [3].

Требование максимальной эффективности использования ресурсов вычислительных мощностей (как процессорных, так и оперативной и дисковой памяти) отдельных процессоров кластера неизбежно приводит к снижению "интеллектуальности" операционной системы вычислительных узлов до уровня *мониторов*; с другой стороны, предлагаются *распределённые кластерные операционные системы* – напр., Amoeba (http://www.cs.vu.nl/pub/amoeba/amoeba.html), Chorus, Mach и др.

Специально для комплектации аппаратной части вычислительных кластеров выпускаются т.н. Bladed - сервера (*) – узкие ("лезвийные") вертикальные платы, включающие процессор, ОП (обычно 256÷512 МБайт при L2-кэше 128÷256 kБайт), дисковую память и микросхемы сетевой поддержки; эти платы устанавливаются в стандартные "корзины" формата 3U шириной 19” и высотой 5,25” до 24 штук на каждую (240 вычислительных узлов на стойку высотою 180 см). Для снижения общего энергопотребления могут применяться расходующие всего *единицы ватт* (против 75 W для P4 или 130 W для кристаллов архитектуры IA-64) процессоры *Transmeta Crusoe*

---

* The Bladed Beowulf: A Cost-Effective Alternative to Traditional Beowulfs. W.Fengy, M.Warrenz, E.Weigley. Advanced Computing Laboratory, Computer & Computational Sciences Division, Los Alamos National Laboratory, Theoretical Astrophysics Group, Theoretical Division, 2002, p.11.

(http://www.transmeta.com) серии TM 5x00 с технологией VLIW (см. подраздел 1.2.2); при этом суммарная потребляемая мощность при 240 вычислительных узлах не превышает 1 kВт.

2.4.3 Коммуникационные технологии, используемые при создании массово-параллельных суперкомпьютеров

Важным компонентом массово-параллельных компьютеров является *коммуникационная среда* – набор аппаратных и программных средств, обеспечивающих обмен сообщениями между процессорами. Фактически коммуникационная среда в данном случае является аналогом общей шины традиционных ЭВМ и поэтому оказывает сильнейшее влияние на производительность всей вычислительной системы.

2.4.3.1 Транспьютерные системы

В середине 80-х г.г. фирмой Inmos Ltd. (в дальнейшем вошедшей в объединение SGS Thomson) были выпущены специализированные микропроцессорные устройства серий T-2xx, T-4xx и T-8xx названные *транспьютерами*. Транспьютеры содержат на едином кристалле собственно процессор, блок памяти и коммуникационные каналы (*линки*) для объединения транспьютеров в параллельную систему. Линки (два проводника, позволяющие передавать данные последовательно в противоположных направлениях, т.н. линки типа OS, *OverSampled*) позволяют передавать данные с пропускной способностью 10 Мбит/сек (для транспьютера T-805 доступна скорость до 20 Мбит/сек). В 90-е годы SGS Thomson были разработаны более производительные и интеллектуальные транспьютеры (модель T-9000 поддерживала *виртуальные линки,* имела *аппаратный коммутатор линков* C-104 и более производительные двухпроводные DS-линки). Подобные идеи привели к разработке стандарта IEEE P1355, предусматривающего передачу данных со скоростями от 10 Мбит/сек до 1 Гбит/сек [4]. Во второй половине 90-х фирма Texas Instruments выпустила транспьютероподобные *сигнальные микропроцессоры* TMS 320 C4x, Analog Device предлагает подобные устройства серии ADSP 2106x (пропускная способность 20 и 40 Мбит/сек соответственно).

Транспьютеры содержат $4 \div 6$ линков, что позволяет *аппаратным образом* реализовывать топологии "двумерная-" и "трёхмерная решетки" (в некоторых случаях "гиперкуб") вычислительных узлов, а также осуществлять связь с дисковыми массивами.

Значительная пропускная способность транспьютеров (наряду с их высокой "интеллектуальностью" вследствие наличия 32-битового процессора) привели к созданию вычислительных систем, состоящих из транпьютероподобных вычислительных узлов; в этом случае ориентированная на вычисле-

ния компонента реализована мощным вычислительным микропроцессором, а коммуникационная – транспьютером. Типичный пример такой конфигурации – российские суперкомпьютеры семейства МВС-100 и МВС-1000/200 (НИИ "Квант", середина 90-х г.г.), каждый вычислительный узел которых включает вычислительный и коммутационный микропроцессоры (i80860XR/i80869XP фирмы Intel и транспьютеры Т-805/Т-425 для МВС-100 и DEC Alpha 21164 и TMS 320C44/ADSP Sharc 21000 для МВС-1000/200).

Структурный модуль МВС-100 представляет собой двумерную матрицу из 16 вычислительных модулей (в каждом использованы транспьютеры с 4-мя линками - рис.13а, максимальная длина пути равна 3), для присоединения внешних устройств используются 4 линка угловых вычислительных модулей, для связи с внешними структурными модулями остается 8 линков.

Базовый вычислительный блок объединяет 2 структурных модуля (рис.12б, максимальная длина пути равна 5, имеется еще 16 линков для дальнейшего объединения). Схема объединения двух базовых блоков приведена на рис.13в, максимальная длина пути равна 6), при этом дальнейшее наращивание возможно посредством дополнительных транспьютеров (см. выделенный пунктиром компонент рис.13в).

В настоящее время транспьютерные технологии считаются уходящими в прошлое, на смену им пришли более гибкие и технологичные в применении сетевые технологии.

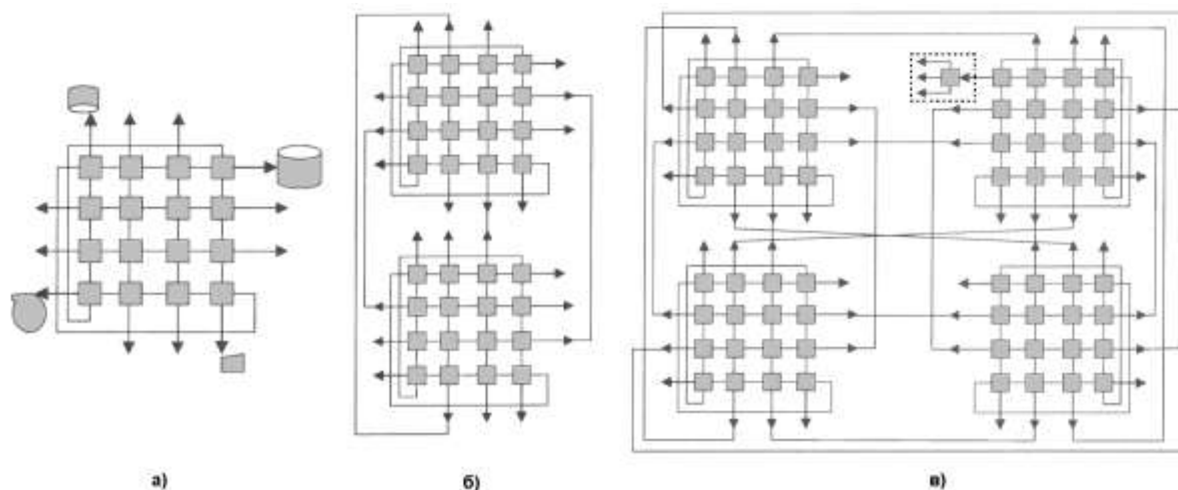


Рисунок 13 — Возможные структурные схемы системы МВС-100: а) – структурный модуль 4×4, б) – базовый вычислительный блок (32 вычислительных модуля), в) – возможная схема объединения двух базовых блоков.

2.4.3.2 Универсальные высокопроизводительные коммуникационные сети: производительность, латентность и цена обмена

Важной характеристикой коммуникационной сети является (экспериментально замеренная) зависимость её производительности от размера сообщений; только при достаточно больших (обычно выше $\approx 0{,}5 \div 1$ Мбайт) размерах сообщений сеть выходит на максимум производительности $S_{\max}$ (рис.14).

Производительность канала обмена данными в простейшем случае определяют на операциях "точка-точка" (обмен между двумя узлами) и замеряют в Мбайт/сек за достаточное продолжительное время (минуты). Однако в реальности межпроцессорный обмен данными характеризуется большим числом относительно коротких (десятки÷тысячи байт) посылок, при этом важное значение начинает играть время "разгона" канала связи.

Время T (сек) передачи сообщения длиной X (Мбайт) от узла А узлу В при пропускной способности сети S (Мбайт/сек) при условии, что никаких иных обменов в сети не происходит, с хорошей точностью описывается формулой:

$$T = \frac{X}{S} + L, \quad (5)$$

где L – время "запуска" обмена (*латентность*); причём L не зависит от длины сообщения X, при $X \to 0$ или $S \to \infty$ имеем $T \to L$ (именно влиянием латентности определяется вид кривой рис.14).

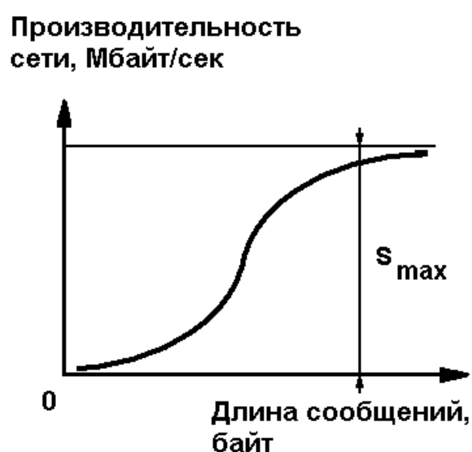


Рисунок 14 — Качественный график зависимости производительности сети от размера передаваемых сообщений

Цена обмена P оценивает число байт, "потерянных" каналом вследствие наличия латентности (естественно стремиться к всемерному снижению латентности):

$$P = L \times S .$$

Оценочные данные для некоторых известных технологий межпроцессорного обмена представлены в табл.3 [5,6].

Таблица 3 — Характеристики некоторых распространённых коммуникационных сетей для типичных представи-

телей MPP-комплексов – *вычислительных кластеров*.

Сетевая технология	*Fast Ethernet*	*SCI (Scalable Coherent Interface,* http://www.dolphinics.com *)*	*Myrinet (Myricom, Inc.,* http://www.myri.com *)*	*InfiniBand,* http://www.infinibandta. org/home
Латентность, μсек	40÷70	3÷10	17	4÷5 (до 1)
Пропускная способность аппаратная (Мбайт/сек)	12	150÷170	170	800÷1000 (однонаправленный обмен) до 2000÷2700 (двунаправленный)
Пропускная способность программная (Мбайт/сек)	10	80	40	-

Технология Gigabit Ethernet позволяет довести пропускную способность сети до 60÷80 Мбайт/сек, однако латентность зачастую в полтора-два раза выше, чем у FastEthernet [5]. Если для повышения быстродействия Fast Ethernet-сети возможно применить технологию Channel Bonding (*связывание каналов*, фактически – их дублирование путём совместного использования сетевых плат; в Linux начиная с ядер версии 2.4.x эта возможность является стандартно включаемой, см. http://linux-cluster.org.ru/bonding.php.htm), то снизить латентность реально лишь с помощью применения более дорогостоящего сетевого оборудования.

Одной из наиболее перспективных технологий считается InfiniBand (http://www.infinibandta.org/home), позволяющая достичь латентности 4÷5 мксек (в перспективе до 1 мксек) и пропускной способности от 800÷1000 Мбайт/сек (однонаправленный обмен) до 2000÷2700 Мбайт/сек (двунаправленные обмены).

Величина латентности определяет класс задач, которые можно эффективно решать на данном многопроцессорном комплексе или необходимый уровень мастерства распараллеливания. Например, при латентности 100 мксек величина зерна распараллеливания должна быть такой, чтобы соответствующая ему последовательность команд выполнялась в среднем за время не менее 1÷10 мсек (т.е. хотя бы на порядок больше времени латентности); это достаточно *крупнозернистый* параллелизм, иначе время обмена сообщениями "съест" все повышение производительности от распараллеливания.

О коренном изменении коммуникационных технологий говорится в работе [3].

2.4.4 Стандартные программные пакеты

организации вычислительных кластеров

В последние годы появилось программные комплексы, позволяющие организовать вычислительные кластеры буквально в течение минут. Это, например, дистрибутивы Parallel Knoppix (http://pareto.uab.es/mcreel/ParallelKnoppix), CLIC (http://clic.mandrakesoft.com/index-en.htm) компании Mandrake (http://mandrake.com), BCCD (http://bccd.cs.uni.edu) разработки Университета Северной Айовы США (http://cs.uni.edu), Rocks Cluster Distribution (http://www.rocksclusters.org) и др.

Особый интерес представляет первая из перечисленных разработок, основанная на технологии openMosix и включающая технологию *миграции заданий*. Кластер openMosix (фактически Linux с 3% изменением исходного кода ядра ОС) представляет собой *однообразную систему* (Single System Image - SSI), при этом ресурсы входящих в кластер ЭВМ могут использоваться более эффективно вследствие возможности *миграции* (перемещения от одной машины к другой) заданий. Особенностью openMosix является то, что пользователь только запускает программу, а ОС кластера решает, где (на каких именно узлах) её выполнить (если не предусмотрено противоположное); разработчики openMosix называют это свойство "разветвись-и-забудь" (*fork-and-forget*). Подобные системы обладают высокой масштабируемостью и могут быть скомпонованы из машин с различными ресурсами (производительность процессора, размер памяти и др.).

Несмотря на лежащие на поверхности явные достоинства подобных программных пакетов, их применение при создании масштабных кластерных систем ограничено (обычная область их применения – демонстрационные и макетные системы и простейшие кластеры). Основная причина этого – серьёзные затруднения при оптимизации построенного с их использованием кластерного ПО (каждый масштабный кластер настраивается и оптимизируется строго индивидуально).

Интересной разработкой ПО кластерных систем является проект МВС-900, разработанный в испытательной лаборатория проекта МВС ИПМ им. М.В.Келдыша РАН, см. http://vbakanov.ru/metods/1441/lacis.zip. Технология МВС-900 использует (относительную) малозагруженность оборудования учебных Windows-классов, техническая реализация МВС-900 основана на применении *диспетчера виртуальных машин* VMware (фирма VMware, Inc., http://www.vmware.com), под которым работает Linux (рекомендуются дистрибутивы Dragon или Slackware, http://sourceforge.net/projects/ dragonlinux, http://www.slackware.com); базовая операционная система – не ниже Windows'2000. При этом Windows и VMware-Linux на физической машине взаимодействуют минимально (переразмечать диски не требуется, используются непересекающиеся адресные пространства, IP-адреса обслуживаю-

щих сетей различны); в случае серьёзной аварии восстановить работоспособность МВС-900 "с нуля" реально за полчаса-час.

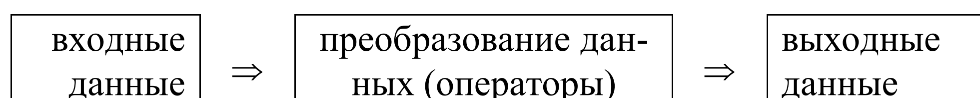
Т.о. *виртуальный кластер* работает совместно с Windows-машинами компьютерного класса, разделяя их ресурсы. Отсутствие значимых накладных расходов при работе Linux под VMware обусловлено ее функционированием не в режиме полной программной эмуляции двоичного кода, а стандартного выполнения на правах обычного Windows-процесса; при этом непривилегированные команды (арифметические, переходов и подобные) выполняются физическим процессором напрямую и только запрещённые к исполнению вне пределов базовой операционной системы привилегированные команды моделируются (эмулируя наличие соответствующего внешнего устройства). В конечном итоге центральным процессором выполняется поток команд (естественно, перемежаемый предписаниями базовой ОС при сохранении и восстановлении контекста), полностью идентичный таков**о**му в настоящих "железных" Linux-кластерах; при этом потери быстродействия процессора на накладные расходы составляют около 7% (по данным разработчиков). Однако формально такой кластер будет *гетерогенным* (производительность каждого узла зависит от загруженности Windows-программами и далеко непостоянна).

Таким образом с помощью технологии МВС-900 реализуется классическая MPP-система коллективного пользования, практически полностью идентичная (по правилам администрирования, программирования и отладки) расположенному в Межведомственном Суперкомпьютерном Центре (МСЦ) суперкластеру МВС-1000М (http://www.jscc.ru).

2.5 Нетрадиционные архитектуры многопроцессорных вычислительных систем

Поиски путей повышения производительности компьютеров вызвали развитие новых нетрадиционных архитектур вычислительных систем; наиболее известными являются *управляемые потоком данных вычислители*, *массивы систолических ячеек* и *многопроцессорные системы с программируемой архитектурой*.

В наиболее общем виде процесс преобразования данных можно записать в виде триады (ср. с известным отношением "*товар*⇒*деньги*⇒*товар*"):



Какой объект (из перечисленных) должен управлять преобразованием? Из общих соображений именно тот, ради которого и "затеяна вся игра" – *данные*

(в этом смысле операторы суть всего лишь конкретизация технологии обработки информации, малоинтересная конечному потребителю).

Метод вычислений, при котором выполнение каждой операции производится при готовности всех ее операндов, называется Data Flow (*вычисления, управляемые потоком данных*), при таком методе *последовательность выполнения команд заранее не задаётся*; используется также понятие Data Driven Computing (*вычисления, управляемые данными*). Впервые графическую модель вычислений, управляемых потоком данных, предложил Дуайн Эдэмс (Стэнфордский университет, 1968). В системе Data Flow вместо императивного указания выполнить некоторую команду используется *разрешение её выполнить,* схема Data Flow фактически противопоставляется традиционной схеме с управлением потоком (Control Flow).

Сама природа метода управления потоком данных подразумевает глубокий параллелизм (в отличие от метода управления потоком команд, заданных человеком). Метод управления потоком данных открывает широкие возможности для организации параллельных вычислительных процессов, причем с *аппаратным распараллеливанием*.

Работы по созданию по созданию программно-аппаратных средств реализации принципа Data Flow велись в государственных и частных исследовательских центрах всего мира: напр., в Массачусетсом технологическом институте (процессор Tagget Token), лабораториях фирмы Texas Instruments (США), в Манчестерском университете (Англия), однако дальше экспериментальных машин дело не пошло. Наиболее известны Data Flow - системы Monsoon, Epsilon (США) и CSRO-2 (Австралия).

В системах Data Flow последовательность выполнения операций зависит от готовности операндов (как только в памяти оказываются необходимые операнды, необходимые и достаточные для выполнения того или иного оператора, исполнение последнего инициируется на одном из нескольких исполнительных устройств - если есть свободные). При этом последовательность выполнения операций программы может отличаться от порядка их записи - например, различные итерации цикла могут выполняться одновременно (более того, $i+1$-я итерация может выполняться ранее i-той – если для неё готовы данные). Data flow - системы включают много исполнительных устройств, работающих параллельно, на каждом из них возможно выполнение любого оператора системы команд машины.

Таким образом удается преодолеть традиционно-узкое место современных скалярных процессоров, существенную часть времени тратящих не на обработку данных, а на *процедуры управления данными* и *спекулятивные вычисления*. В отличие от многомашинных комплексов с традиционной архитектурой распараллеливание вычислительных процессов в системах Data Flow осуществляется аппаратно. При этом работа по организации вычислительного процесса возлагается на особое запоминающее устройство, в ко-

тором происходит накопление операндов и поиск готовых к выполнению операторов (аналог устройства управления в традиционных последовательных ЭВМ). В настоящее время основным препятствием для широкой реализации потоковых вычислителей является отсутствие недорогой ассоциативной памяти достаточного объёма (поиск готовых к выполнению операторов в традиционной RAM-памяти сводит на нет все преимущества DATA FLOW). Для описания потоковых программ используется *язык акторных сетей*.

Наиболее эффективная реализация этого механизма – использование ассоциативной памяти, которая представляет собой несложное, но громоздкое устройство, предназначенное не только для хранения данных с определёнными признаками – тэгами (*tag*, причём тэг указывает, в *каком контексте* используются данные и для выполнения какой команды они предназначены), но и для сравнения тэгов (при их совпадении происходит выборка данных и инициация выполнения соответствующей команды). Заметим, что этот механизм частично используется в некоторых современных высокопроизводительных микропроцессорах (напр., в 64-разрядном Itanium фирмы Intel).

В терминологии Data Flow данные вместе с набором признаков (тэгами) именуются токенам (*token* – признак, метка, *готовый для употребления*). При Data Flow механизм управления данными сильно развит, соответственно имеется большее число управляющих признаков у данных (выполняемая и следующая за ней команды – та, кому необходим результат операции, номер итерации, тип операции - векторная/скалярная, одно/двухоперандная, число токенов результата операции, при этом совокупность признаков токена без кода операции именуется его "окраской"), [7].

На рис.15 приведена схема Data Flow – вычислителя согласно работы (∗), исследования и создание новых архитектур ЭВМ проводились в рамках "Программы Основных направлений фундаментальных исследований и разработок по созданию оптической сверхвысокопроизводительной вычислительной машины Академии наук" (ОСВМ), позднее от реализации ассоциативной памяти на основе оптики отказались в пользу появившихся на рынке высокопроизводительных полупроводниковых модулей троичной ассоциативной памяти (*Ternary Content Addressable Memory,* TCAM) компаний Motorola, Music Semiconductors, Inc., Micron Tech., Inc.

По этой схеме вычислительное устройство представляет собой кольцевую структуру. Токены формируются на основе анализа структуры алгоритма, хранятся в ассоциативной памяти (АП) и при условии набора их комплекта, достаточного для выполнения одного из операторов, направляются в буфер

---

* Бурцев В.С. Выбор новой системы организации выполнения высокопараллельных вычислительных процессов, примеры возможных архитектурных решений построения суперЭВМ. // В кн.: Параллелизм вычислительных процессов и развитие архитектуры суперЭВМ. – М.: ИВВС РАН, 1997.

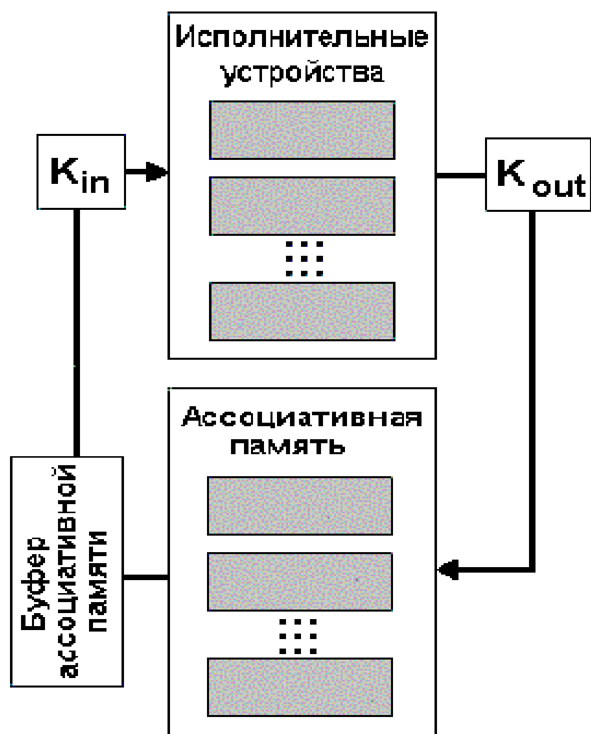


Рисунок 15 — Архитектура вычислительного устройства, реализующего подход Data Flow.

ассоциативной памяти (БАП). В БАП'е накапливаются готовые пакеты для передачи в исполнительной устройство (ИУ); коммутатор K_{in} распределяет пакеты готовых токенов между свободными ИУ.

В ассоциативной памяти поиск нужной информации производится не по адресу ячейки, а по её содержанию (*ассоциативному признаку*), при этом поиск по ассоциативному признаку (или последовательно по отдельным его разрядам) происходит параллельно во времени для всех ячеек запоминающего массива. Данные в ассоциативной памяти интенсивно обрабатываются, поэтому для увеличения быстродействия она разбита на отдельные модули. Набор признаков токенов, полученных в результате обработки ИУ, аппаратно анализируется и коммутатором K_{out} распределяется по модулям АП.

Именно в таком направлении работают сотрудники отдела Института проблем информатики РАН, ряд лет возглавляемые В.С.Бурцевым (при финансовой поддержке американской корпорации Nodal Systems).

Другой путь развития вычислительных систем связывают с отказом от коммутаторов (коммутационных сетей), ограниченное быстродействие которых тормозит рост производительности МВС. Современные технологии позволяют создавать кристаллы с огромным количеством простых *процессорных элементов* (ПЭ), массивы ПЭ с непосредственными соединениями между близлежащими ПЭ называются *систолическими*. Такие массивы обладают очень высоким быстродействием, но каждый из них ориентирован на решение очень узкого класса задач.

В работе [1] описано использование систолических массивов (СМ) для решения специальной задачи умножения и сложения матриц $[D]=[C]+[A]\times[B]$ (частный случай *афинного преобразования*), причем все матрицы — ленточные порядка N; матрица [A] имеет одну диагональ выше и две диагонали ниже главной; матрица [B] — одну диагональ ниже и две диагонали выше главной; матрица [C] по три диагонали выше и ниже главной. Каждый входящий в систолический массив ПЭ выполняет *скалярную* операцию $c+ab$ и одно-

временно осуществляет передачу данных (имеет три входа a,b,c и три выхода a,b,c, причем (in) и выходные (out) данные связаны соотношениями $a_{out}=a_{in}$, $b_{out}=b_{in}$, $c_{out}=c_{in}+a_{in}\times b_{in}$, точка в обозначении систолической ячейки определяет ее ориентация на плоскости, рис.16).

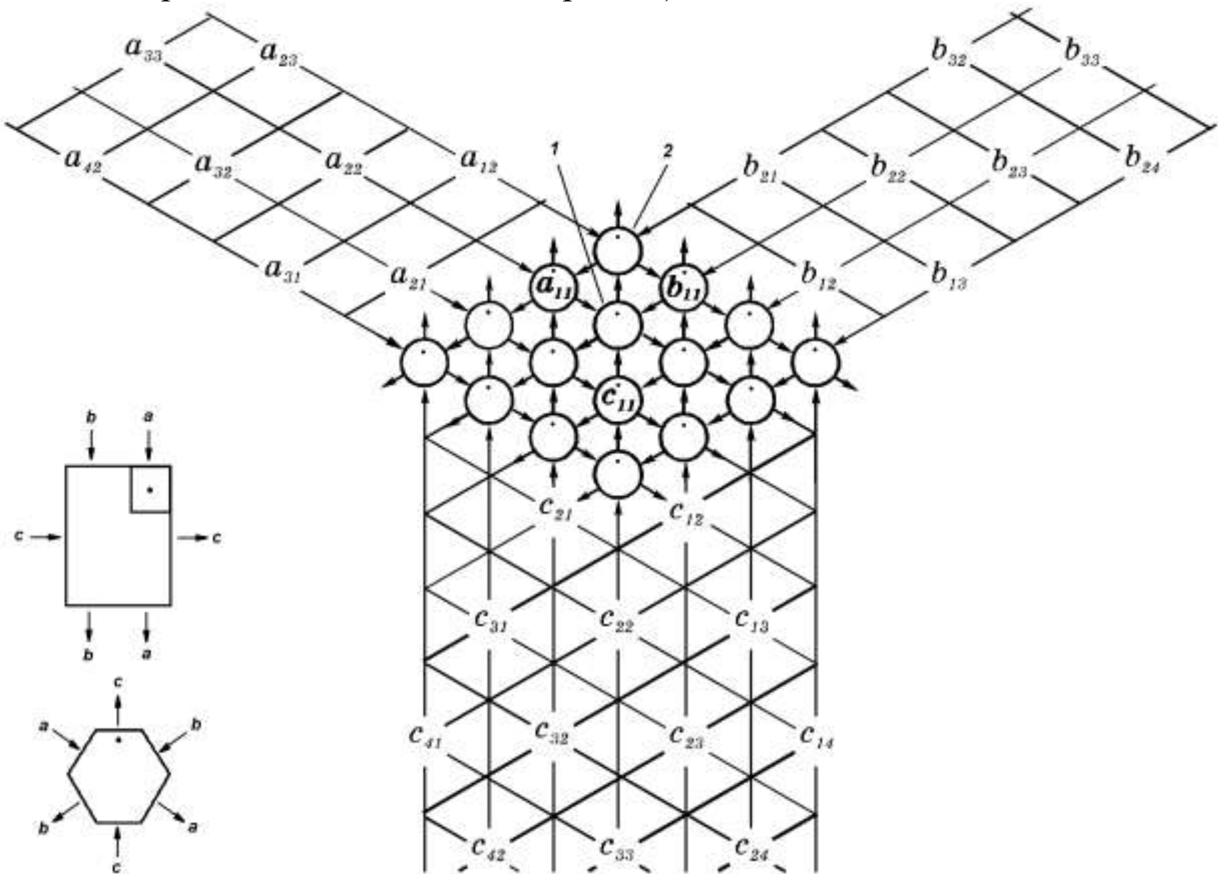


Рисунок 16 — Систолический массив для реализации матричной операции [D]=[C]+[A]×[B] (слева – принятое обозначение систолической ячейки для выполнения скалярной операции c+a×b и передачи данных).

Ячейки расположены в узлах регулярной косоугольной решетки, исходные данные поступают слева сверху, справа сверху и снизу (элементы матрицы [A], [B] и [C] соответственно), за каждый такт все данные перемещаются в соседние узлы по указанным стрелками направлениям. Рис.16 иллюстрирует состояние СМ в некоторый момент времени, при следующем такте все данные переместятся на один узел и элементы a_{11}, b_{11} и c_{11} окажутся в одном ПЭ (помечен номером *1*), находящемся на пересечении штриховых линий (т.е. будет вычислено выражение $c_{11}+a_{11}\times c_{11}$. В этот же такт данные a_{12} и b_{21} вплотную приблизятся в ПЭ, находящемся в вершине систолического массива (помечен символом *2*), в следующий такт все данные снова переместятся на один узел по направлению стрелок и в верхнем ПЭ окажутся a_{12} и b_{21} плюс результат предыдущего срабатывания ПЭ, находящегося снизу, т.е. $c_{11}+a_{11}\times b_{11}$. Следовательно, будет вычислено выражение

$c_{11}+a_{11}\times b_{11}+a_{12}\times b_{21}$, а это и есть искомый элемент d_{11} матрицы [D]. На соответствующих верхней границе СМ выходах ПЭ, периодически через три такта выдаются очередные элементы матрицы [D], на каждом выходе появляются элементы одной и той же диагонали. Примерно через $3\times n$ тактов будет закончено вычисление всей матрицы [D], при этом загруженность каждой систолической ячейки асимптотически приближается к 1/3.

Систолические массивы обладают чертами как процессорных матриц (совокупность связанных ПЭ, выполняющих единую команду), так и явные признаки конвейерного вычислителя (что выражается в потактном получении результата). Систолические массивы являются *специализированными вычислительными системами* и могут выполнять только *регулярные алгоритмы*; математические модели систолических массивов, пригодные для целей построения СМ с заданными свойствами, обсуждаются в работе [1].

Главное отличие *многопроцессорных систем с программируемой архитектурой* является возможность программно устанавливать как связи между процессорными элементами, так и функции, выполняемые данными ПЭ. Важнейшей составной частью такой системы является универсальная коммутационная среда (УКС) (*), которая состоит из однотипных, соединенных друг с другом регулярным образом автоматических коммутационных ячеек, для которых характерно *коллективное поведение*.

Структура многопроцессорной системы, использующей универсальную коммутацию, приведена на рис.17. Настройка системы на выполнение конкретной задачи производится в два этапа:

- На первом этапе производится распределении крупных (макро-) операций (интегрирование, дифференцирование, матричные операции, быстрое преобразование Фурье и т.п.) между ПЭ и настройка имеющихся ПЭ на выполнение этих операций.
- Второй этап заключается в настройке необходимых каналов связи между ПЭ в УКС.

После этого возможен автоматический режим работы многопроцессорной системы, при котором на вход подаётся входная информация, а с выхода снимаются результаты.

Простейший коммутационный элемент (КЭ), необходимый для построения плоской УКС (возможны и многомерные УКС), содержит два входа и два выхода и связан с четырьмя соседними КЭ. В таком элементе возможны девять вариантов внутренней коммутации (рис.18), что и обеспечивает возмож-

---

* Каляев А.В. Многопроцессорные системы с программируемой архитектурой. -М.: Радио и связь, 1984. -283 с.

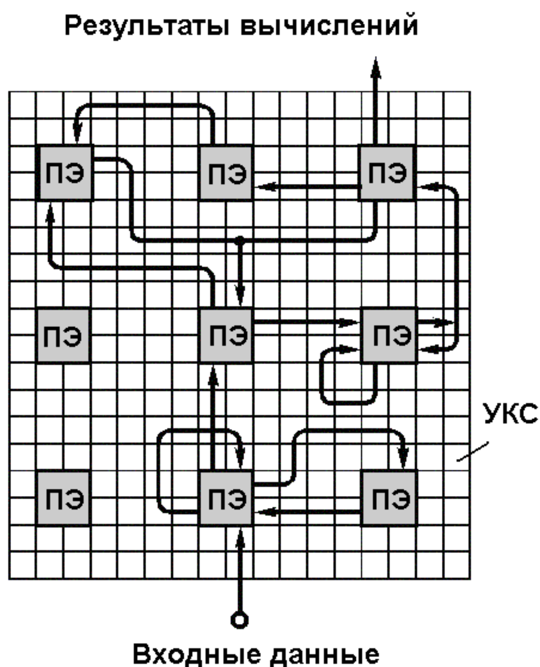


Рисунок 17 — Структура многопроцессорной системы с программируемой архитектурой

ность установления произвольных связей между ПЭ (в т.ч. и изображенных на рис.17).

При настройке плоской УКС намечается непрерывная цепочка свободных КЭ, соединяющих входы и выходы нужных ПЭ. При этом цепочка должна обходить неисправные КЭ и элементы, уже вошедшие в другие каналы связи. После выбора канала связи во все выбранные КЭ вводится код настройки, далее канал связи функционирует подобно сдвиговому регистру, в котором информация по тактам передается от одного КЭ к другому. Настройка КЭ может производиться вручную или автоматически.

Для автоматического образования одного канала связи между двумя ПЭ хорошо подходит известный *волновой принцип*:

- В УКС указываются входной и выходной КЭ.
- Входной элемент соединяется со всеми исправными и незанятыми соседними элементами (при этом образуется волна подсоединённых КЭ; если в ней не оказалось нужного выходного элемента, то каждый элемент волны соединяется с соседними КЭ, образуя новый фронт волны).
- Продвижение волны продолжается до тех пор, пока её фронт не достигнет нужного выходного КЭ, после чего в элементы, составляющие кратчайший путь между двумя ПЭ, заносятся коды настройки, а остальные установленные связи разрываются.
- Процесс повторяется для всего набора входных и выходных КЭ; таким образом последовательно устанавливаются все связи, необходимые для решения конкретной задачи.

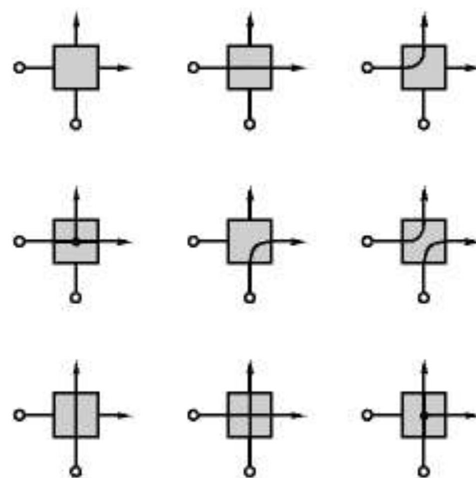


Рисунок 18 — Варианты внутренней коммутации простейшего КЭ

Эффективность систем с программируемой архитектурой зависит от величины коэффициента $K = t_з / t_н$, где $t_з$ — время решения задачи,

$t_н$ — время настройки системы (естественно стремиться к увеличению *K*). Время $t_н$ для систем с программируемой архитектурой относительно велико, поэтому повысить *K* можно только увеличением $t_з$.

Для увеличения $t_з$ следует эксплуатировать многопроцессорную систему с программируемой архитектурой длительное время для решения одной и той же задачи; при этом в каждый такт на вход системы поступает новый набор исходных данных, а с выхода системы в каждый такт снимаются новые результаты – именно этим объясняется очень высокое быстродействие при решении *крупных задач*. Т.о. наиболее выгодным для систем с программируемой архитектурой является *режим конвейерной обработки*, в то же время эти системы принадлежат к многопроцессорным системам типа MIMD в классификации М.Флинна (см. подраздел 2.2 данной работы).

Интересно, что еще в конце 80-х г.г. была создана экспериментальная "из *Ряда* вон выходящая" многопроцессорная (24 вычислительных модуля, 12 коммутационных процессоров и 6 процессоров ввода-вывода) вычислительная система ЕС-2704 (*В.А.Торгашев, В.У.Плюснин*, Научно-исследовательский центр электронной вычислительной техники - НИЦЭВТ), представляющая собой мультипроцессор с *динамической архитектурой* (МДА) и массовым параллелизмом; основой необычной архитектуры составляла адаптивная (подстраивающаяся под логическую структуру конкретной задачи) сеть вычислительных модулей с полностью децентрализованным управлением. Для ЕС-2704 характерно представление задачи не в виде традиционного алгоритма, а в виде сети, каждый элемент которой способен изменять свои связи с другими элементами сети, порождать новые элементы или самоуничтожаться. Аппаратно поддержанная распределённая ОС обеспечивала динамическое управление ресурсами при решения задач, подготовленных для выполнения в сети процессоров, высокую степень распараллеливания и надежность. Дальнейшие разработки привели к появлению нового типа компонента вычислительной системы - процессора с динамической архитектурой (ПДА), который может быть как элементом (модулем) МДА, так и самостоятельным устройством, которое может быть использовано для самых различных приложений; реализации процессоров с динамической архитектурой (DAP-31, DAP-311, DAP-317) применяются в системах обработки радиолокационной информации и телекоммуникаций.

2.6 Метакомпьютинг и GRID-системы

Несмотря на огромную производительность суперкомпьютеров, вполне реально существует огромный резерв вычислительных мощностей, который представляют объединённые сетью InterNet миллионы компьютеров. Их суммарная производительность и объем памяти на порядки превышают соответствующие величины любого мыслимого суперкомпьютера.

Эти миллионы ЭВМ образуют сообщество вычислительных мощностей (*метакомпьютер*), поэтому использующая эти мощности технология получила название *метакомпьютинг*. Для связи ЭВМ при метакомпьютинге

обычно используется InterNet (протокол TCP/IP) как общедоступная и охватывающая всю Планету сеть, хотя для некоторых специальных проектов могут создаваться специализированные компьютерные сети. Метакомпьютер существенно отличается от иных вычислительных систем – он является распределённым по своей природе, динамически меняет свою конфигурацию (специальная программная система отслеживает включение и отключение отдельных ЭВМ таким образом, чтобы пользователь этого не замечал), метакомпьютер неоднороден (используются компьютеры самых различных мощностей и конфигураций с несовпадающими системами инструкций процессоров, операционными системами и т.д.).

Не всякие задачи могут эффективно решаться с помощью метакомпьютинга – вследствие существенно большего (секунды/минуты) времени обмена данными для обработки пригодны задачи, не требующие тесного (частого и интенсивного) взаимодействия параллельных процессов.

Одним из наиболее известных метакомпьютерных проектов является SETI@home (*Search for Extra Terrestrial Intelligence*) – обработка принятых радиотелескопом Аресибо (Пуэрто-Рико) данных (просматривается полоса 2,5 MHz вокруг частоты 1420 MHz, в день записывается около 35 Гбайт информации) с целью выявления признаков сигналов искусственного происхождения (цель – поиск внеземной *разумной* жизни). Принять участие в программе может любой пользователь Windows, UNIX, Mac или OS/2, загрузив клиентскую часть программы с адреса http://setiathome.ssl.berkely.edu (русскоязычное зеркало http://setiathome.spb.ru). Серверная часть ПО загружает на клиентские машины блоки данных размером 0,34 Мбайт и после обработки принимает результаты анализа, расчёты возможны в период простоя ПЭВМ (работа в режиме "хранителя экрана") или постоянно в фоновом режиме; ещё к 2002 г. число клиентов приблизилось к 4 млн., при этом суммарная производительность их ЭВМ намного превосходит производительность всех включенных в "Top-500" суперкомпьютеров.

Проект Einstein@Home (http://einstein.phys.uwm.edu) имеет цель проверки одного из следствий общей теории относительности А.Эйнштейна - существование гравитационных волн. Данные поступают с LIGO (*Laser Interferometer Gravitational wave Oservatore*, две установки в США – Ливингстон и Ханфорд, штаты Луизиана и Вашингтон соответственно) и установки GEO 600 (Ганновер, Германия); метод замера – сравнение времени запаздывания луча лазера в интерферометрах со сверхдлинным плечом $0{,}6 \div 4$ км при анализе гравитационного воздействия от пульсаров или нейтронных звезд. Проект запущен в начале 2005 г., размер клиентского блока данных составляет $12 \div 14$ Мбайт и рассчитан на недельную обработку, приращение участников проекта – 1’000 в день.

Проект Condor (http://www.cs.wisc.edu.condor) распределяет задачи в корпоративной сети, используя неиспользуемые ресурсы ПЭВМ в режиме их про-

стоя (нерабочие дни, ночь и т.п.); программное обеспечение распространяется свободно для платформ Windows NT, Linux, Solaris и др.

В рамках проекта Globus (http://globus.org) разрабатывается *глобальная информационно-вычислительная среда*; созданные в рамках проекта программные средства распространяются свободно (в т.ч. с исходными текстами) в виде пакета Globus Toolkit; это программное обеспечение является основой многих иных проектов в области метакомпьютинга и GRID.

Некоторые из других известных проектов распределенных вычислений:

- Climate Prediction - проект по моделированию влияния выбросов углекислого газа на климат Земли ("парниковый эффект").
- Legion (*A Worldwide Virtual Computer*, http://legion.virginia.edu) - разработка объектно-ориентированного программного обеспечения для построения виртуальных метакомпьютеров на основе миллионов индивидуальных ЭВМ и высокоскоростных сетей, при этом работающий на ПЭВМ пользователь имеет полностью прозрачный доступ ко всем ресурсам метакомпьютера.
- Dictributed.Net (http://distributed.net) - перебор ключей с целью взлома шифров.
- TERRA ONE (http://cerentis.com) – коммерческий проект фирмы Cerentis по созданию метакомпьютера для анализа различной информации (предоставившие для вычислений свои ПЭВМ пользователи получают кредиты TerraPoints для закупок в InterNet-магазинах).
- Find-a-Drug (http://www.find-a-drug.org) - проект по поиску лекарств от различных болезней путем расчета процесса синтеза белков с запланированной молекулярной структурой.
- Folding@Home - проект по расчету свёртывания белков (суммарная вычислительная мощность 200 Тфлопс).
- GIMPS (*Great Internet Mersenne Prime Search*) - поиск простых *чисел Мерсенна* (http://mersenne.org). Фонд *Electronic Frontier Foundation* (http://www.eff.org) предлагает приз в 100 тыс. $US за нахождение числа Мерсенна (простого числа вида 2^P-1) с числом цифр 10 млн.
- SeventeenOrBust (http://www.seventeenorbust.com) - проект, занимающийся решением *задачи Серпински* (нахождении величины n, при котором значение выражения $k \times 2^n + 1$ является простым; в начале проекта это было доказано для всех значений *k*, кроме семнадцати – отсюда название проекта *SeventeenOrBust*).
- ZetaGrid (http://www.zetagrid.net) - проверка *гипотезы Римана* (одна из проблем теории чисел, 1859).

Несмотря на задействованные при метакомпьютинге огромные вычислительные мощности, некоторые задачи не могут быть за приемлемое время решены даже этим способом.

Под GRID (по аналогии с "power grid" – электрическая сеть) понимают распределенную программно-аппаратную компьютерную среду, призванную путём интегрирования географически удалённых компьютерных ресурсов обеспечить всеобщий доступ к вычислительным мощностям для решения крупных научно-технических задач. Для создания надёжной системы GRID особенно важно создание специализированного программного обеспечения промежуточного (*middleware*) уровня для управления заданиями, обеспечения безопасного доступа к данным, перемещения/тиражирования данных огромного объёма в пределах континентов.

Показательным примером использования технологии GRID является организация обработки данных экспериментов Большого Адронного Коллайдера (LHC, *Large Hadron Collider*) в Европейском Центре Ядерных исследований (CERN), $2006 \div 2007$ г.г. На четырёх крупных физических установках в течение $15 \div 20$ лет будет собираться по несколько Петабайт данных ежегодно. Во время проведения эксперимента данные записываются со скоростью $0{,}1 \div 1$ Гбайт/сек и сохраняются как в архивах CERN, так и многих сотен участников – исследовательских институтов и университетов всех стран мира; для хранения и обработки данных создана пятиуровневая иерархическая структура вычислительных центров.

Поддержку LHC осуществляет система EU Data GRID (EDG, http://eu-datagrid.org), основан Европейским Сообществом с целью создания сетевой компьютерной инфраструктуры нового поколения для обработки распределенных тера- и петабайтных баз данных, полученных в результате научных исследований. EDC предполагает хранение и обработку данных физики высоких энергий, биоинформатики и системы наблюдений за Планетой (особенность проекта – разделение информации по различным базам данных, расположенных на различных континентах, цель – кардинальное улучшение эффективности и скорости анализа данных методом распределения мощностей процессоров и систем распределенного хранения с динамически распределением и репликацией). Основой проекта является набор программных средств Globus. В России действует компонент DataGrid, услугами которого пользуются ведущие научные институты - *НИИЯФ МГУ, ОИЯИ, ИТЭФ, ИФВЭ*.

В НИВЦ МГУ разработана GRID-система X-Com, предназначенная для организации распределённых вычислений с использованием неоднородных вычислительных ресурсов. Для функционирования X-Com не требуется специализированных компьютеров, высокоскоростных сетей, сложной настройки задействованных вычислительных ресурсов и необходимости использования определённого языка программирования.

К коммерческим вариантам GRID обычно относят т.н. *облачные вычисления*.

Контрольные вопросы

1. В чем заключается основное отличие многопроцессорных систем с *общей* и *распределённой* памятью? Каковы достоинства и недостатки систем каждого из этих классов? Что такое масштабируем ость многопроцессорных вычислительных систем? В чём преимущества и недостатки компьютеров с NUMA-архитектурой?
2. Допустим, имеются системы параллельного программирования с возможностью преимущественного распределения по вычислительным узлам: а) только вычислений, б) только данных, в) и данных и вычислений. С помощью каких из этих систем проще (*удобнее, быстрее*) разрабатывать эффективные параллельные программы для многопроцессорных вычислительных комплексов архитектуры: а) SMP, б) NUMA, в) MPP?
3. В чем суть использования матричных переключателей и принципа каскадирования при объединении модулей многопроцессорных ЭВМ? Каковы достоинства и недостатки этих технологий?
4. Что такое величина *средней длины пути*, соединяющего произвольные вычислительные узлы многопроцессорных систем? Для каких топологий эта величина больше и для каких она минимальна?
5. В чем различие топологий "двумерная решетка" и "двумерный тор" (соответственно "трёхмерная решетка" и "трёхмерный тор")?
6. На каких характеристиках основываются известные классификации параллельных вычислительных систем?
7. Постарайтесь расширить топологию МВС-100 и МВС-100/200 для транспьютеров с 6-ю линками (представить возможные схемы топологии).
8. Каков физический смысл понятий *латентности* и *цены обмена*? Предложите иную систему (количественных) характеристик для рассматриваемого явления.
9. Назвать распространённые сетевые технологии для связывания вычислительных узлов многопроцессорных систем и дать количественные характеристики основных их параметров (латентности, пропускной способности и др.).

3 Анализ алгоритмов с целью выявления параллелизма

Кардинальным отличием выполнения программы на вычислительной системе параллельной архитектуры от такового последовательного компьютера является *возможность* одновременного выполнения целой группы операций, друг от друга не зависящих (на последовательной машине в каждый момент времени выполняется только одна операция, другие могут находиться в лучшем случае на стадии подготовки). На различных параллельных машинах эти группы и последовательность их выполнения скорее всего *будут разными*. Однако существует (естественное) требование *повторяемости результатов* (алгоритм, приводящий к различным конечным результатам даже при одинаковых входных данных, вряд ли кому нужен – моделирование статистических процессов не рассматриваем).

В общем случае этап разработки параллельной программы должен предваряться процессом выявления блоков (последовательностей исполняемых предписаний), могущих быть выполненными независимо друг от друга и точек (моментов) в программе, в которых необходима синхронизация выполнения этих блоков (для ввода или обмена данными). Только для простейших алгоритмов эта задача может быть выполнена "в уме", в большинстве случае требуется (достаточно сложный) *анализ структуры алгоритма*. В некоторых случаях целесообразно провести *эквивалентные преобразования алгоритма* (замена данного алгоритма – или его части – на алгоритм, гарантирующий получение такого же конечного результата на всех наборах входных данных, причем желательно без снижения точности расчётов).

3.1 Представление алгоритма графовой структурой

Принято и удобно представлять алгоритм в виде графовой структуры. Граф *G* стандартно обозначается *G=(V,E),* где *V* – множество вершин (*vertex*), *E* – множество дуг (*edge*), дуга между вершинами *i* и *j* обозначается как *e(i,j).* В общем случае вершины графа соответствуют некоторым *действиям программы*, а дуги – *отношениям* между этими действиями.

Простейший граф такого рода описывает *информационные зависимости алгоритма* (вершины этого графа соответствуют отдельным операциям алгоритма, наличие дуги между вершинами *i,j* говорит о необходимости при выполнении операции *j* наличия аргументов (операндов), выработанных операцией *i*; в случае независимости выполнения операций *i* и *j* дуга между вершинами отсутствует). Такой граф называют *графом алгоритма* (*вычислительной моделью "операторы - операнды"*), [1,7]. Даже при отсутствии условных операторов (что маловероятно) число выполненных операций (а следовательно, и общее число вершин графа и, соответственно, число дуг) зависит от размеров входных данных, т.е. граф алгоритма (ГА) является *пара-*

метризованным по размеру входных данных. Ацикличность ГА следует из невозможности определения любой величины в алгоритме через саму себя. ГА также является *ориентированным* (все дуги направлены). Различают *детерминированный* ГА (программа не содержит условных операторов) и *недетерминированный* ГА (в противном случае). Для недерминированного ГА не существует взаимно-однозначного соответствия между операциями описывающей его программы и вершинами графа при всех наборах входных параметрах; поэтому чаще всего рассматриваются детерминированные алгоритмы. Не имеющие ни одной входящей или выходящей дуги вершины ГА называются *входными* или *выходными вершинами* соответственно. Построение ГА не является излишне трудоёмкой операцией (чего нельзя сказать о процедурах анализа графа) – любой компилятор (интерпретатор) строит (явно или неявно) его при анализе каждого *выражения* языка программирования высокого уровня

Последовательность вычислений (один из вариантов) нахождения корней полного квадратного уравнения $ax^2+bx+c=0$ в виде $x_{1,2}=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$ приведена в табл.4 и требует 11 операций высокоуровневого языка программирования (сложение, вычитание, умножение, деление, изменение знака, вычисление квадратного корня и т.п.).

Таблица 4 — Последовательность вычисления значений корней полного квадратного уравнения.

№ оператора	*Действие*	*Примечание*
	Ввод a, b, c	Операции ввода не нумеруются
0	a2←2×a	a2 – рабочая переменная
1	a4←4×a	a4 – рабочая переменная
2	b_neg←neg(b)	b_neg – рабочая переменная, neg – операция изменение знака ("унар-ный минус")
3	bb←b×b	bb – рабочая переменная
4	ac4←a4×c	ac4 – рабочая переменная
5	p_sqr←bb-a4	p_sqr – рабочая переменная
6	sq←sqrt(p_sqr)	sq – рабочая переменная, sqrt – операция вычисления квадратного корня
7	w1←b_neg+sq	w1 – рабочая переменная
8	w2←b_neg-sq	w2 – рабочая переменная
9	root_1←w1/a2	root_1 – первый корень уравнения
10	root_1←w2/a2	root_2 – второй корень уравнения

При последовательном вычислении граф алгоритма (рис.19) полностью копирует последовательность действий табл.4; имеет 3 входных вершины (соответствующие вводу коэффициентов a,b и c), две выходные вершины (вычисленные корни x_1 и x_2 исходного уравнения) и 11 вершин, соответствующих операторам алгоритма.

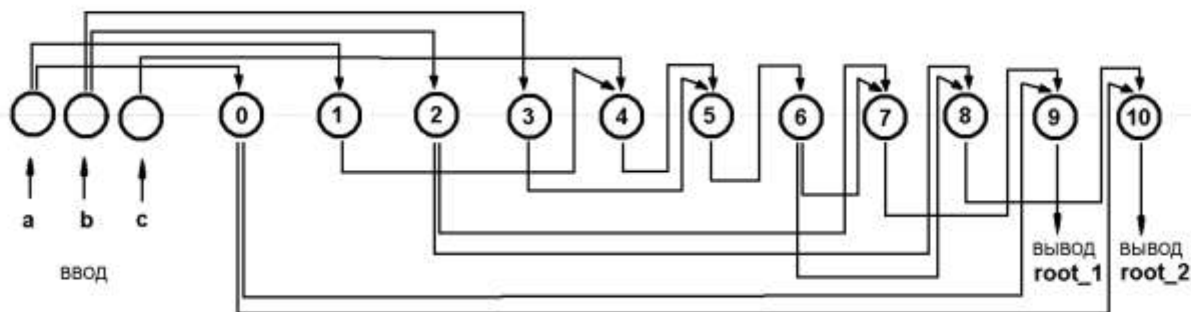


Рисунок 19 — Граф алгоритма (зависимость "операции - операнды") нахождения корней полного квадратного уравнения для последовательного выполнения.

Одним из (неэкономных по использованию памяти при использовании) представлений графа *G=(V,E)* является квадратная *матрица смежности* (нумерация строк и столбцов соответствует нумерации операторов, булева "1" в (*i,j*)-той ячейке соответствует наличию дуги *e(i,j)*, "0" – её отсутствию):

$$\begin{bmatrix} 0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&1&1&0&0 \\ 0&0&0&0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&1&1&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&0&1&1&0&0 \\ 0&0&0&0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&0&0&0&1 \\ 1&0&0&0&0&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0&0&0&0&0 \end{bmatrix}$$

При начальном анализе время выполнения всех операций принимается одинаковым, более точный анализ требует информации о трудоёмкости каждой операции (при этом вершинам и дугам графа могут быть сопоставлены веса – величины, пропорциональные трудоёмкостям операций вычисления и обмена данными по времени соответственно).

Тот же граф представлен на рис.20 в иной форме, причём одновременно проведен *анализ внутренней структуры алгоритма* с целью поиска групп операторов, могущих быть выполненными параллельно. Такой граф пред-

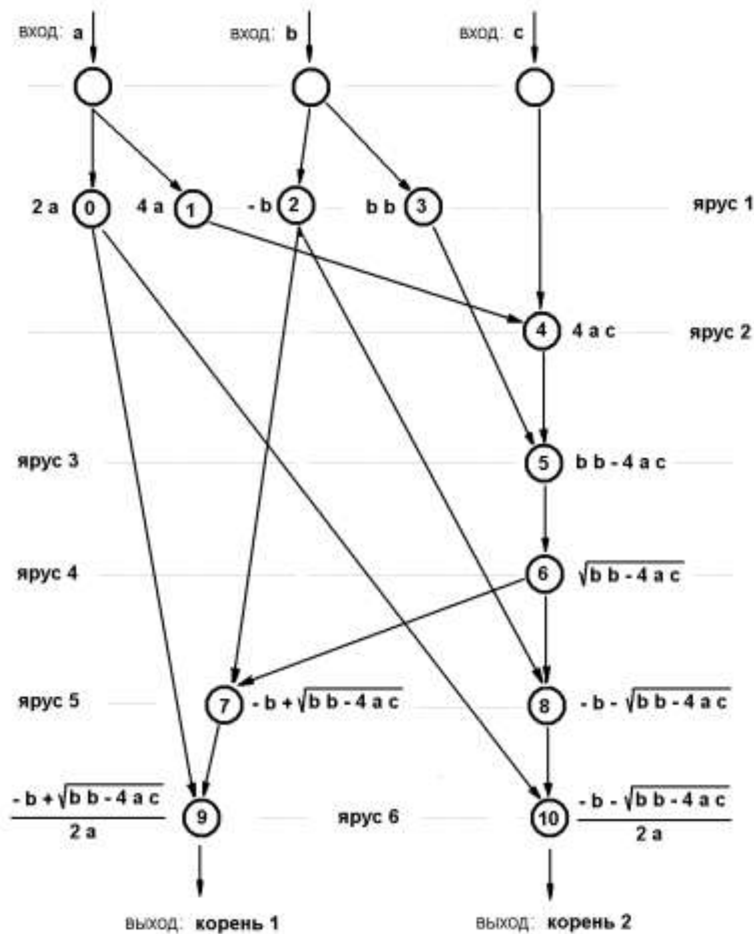


Рисунок 20 — Граф алгоритма (зависимость "операции - операнды") для нахождения корней полного квадратного уравнения с группировкой операций по ярусам

ставляет *ярусно-параллельную форму* (ЯПФ) алгоритма, причем количество ярусов определяет *длину критического пути*.

В *ярусы* собираются операторы, требующие для своего выполнения значений (операндов), которые вычисляются *только на предыдущих ярусах* (всего на рис.20 выделено 6 ярусов); т.о. параллельное выполнение данного алгоритма требует последовательного 6-разового выполнения блоков параллельных операций (в каждом из которых запускаются 4,1,1,1,2,2 независимых процесса соответственно, причём ярусы 2,3,4 вырождаются в последовательное выполнение).

Граф рис.20 позволяет уже сделать некие конкретные выводы о альтернативах данного распараллеливания. Заметим, что ярус 1 перегружен операциями (3 умножения и 1 изменение знака), часть из них (кроме операции 2) можно перенести на более нижние ярусы (варианты: операцию 4 на ярус 2, операцию 3 на ярусы 2,3 или 4, операцию 1 на ярусы 2,3,4 или 5); конкретный вариант должен выбираться исходя из дополнительных данных (например, время выполнения конкретных операций, число задействованных вычислительных модулей, минимизация времени обменов данными между модулями).

Выявление этих ярусов представляет собой один из уровней *анализа внутренней структуры алгоритма*. Нижеприведена *упрощённая* последовательность действий по выявлению могущих выполняться параллельно ярусов графа алгоритма (*вдумчивый читатель* самостоятельно произведёт необходимые уточнения и оптимизацию):

Пункт	*Действие*	*Примечание*
а)	Задать список вершин, зависящих *только от входных данных*; занести его в список *list_1*	Начальные данные для работы алгоритма задаются вручную
б)	Найти вершины, зависимые по дугам от вершин, входящих *только* в список *list_1* и предыдущие списки (если они есть); занести их в список *list_2*	Наиболее трудоёмкий пункт, требующий просмотра матрицы смежности для каждого элемента списка
в)	Если список *list_2* не пуст, скопировать его в *list_1* и идти к пункту б); иначе закончить работу	Цикл по ярусам, пока они могут быть выявлены

Вариант (с целью достижения наибольшей прозрачности оптимизации не производилось) реализации алгоритма на языке С приведён ниже (исходные данные - квадратная булева матрица смежности MS[ ][ ] размерности N_MS, одномерные целочисленные массивы LIST_1[ ] и LIST_2[ ] длиной N_L1 и N_L2 соответственно):

```
001   do {                                        // по ярусам сколько их будет выявлено
002    N_L2=0;
003    for (ii=0; ii<N_L1; ii++) {                // цикл по вершинам в списке LIST_1
004     i_ii=LIST_1[ii];                          // i_ii – номер вершины из списка LIST_1
005     for (j=0; j<N_MS; j++)                    // цикл по столбцам MS (j – номер вершины,
                                                  // к которой  направлено дуга)
006      if (MS[i_ii][j]) {                       // нашли какую-то дугу i_ii → j
007       j1 = j;                                 // запомнили вершину, к которой идет дуга от i_ii
008       for (i1=0; i1<N_MS; i1++)               // по строками MS = исходящим вершинам
009         if (MS[i1][j1]) {                     // нашли какую-то дугу i1 → j1
010           flag=false;
011           for (k=0; k<N_L1; k++)              // цикл по списку LIST_1
012            if (LIST_1[k] == i1)               // если вершина j1 входит в LIST_1…
013             flag=true;                        // при flag=true вершина i1 входит в список LIST_1
014           }  // конец блока  if (MS[i1][j1])
015        if (flag)                              // …если i1 входит в список LIST_1
016          LIST_2[N_L2++] = j1;                 // добавить вершину j1 в список LIST_2
017      }  // конец блока if (MS[i_ii][j])
018     }  // конец блока for (ii=0; ii<N_L1; ii++)
019    for(i=0; i<N_L2; i++)                      // копируем LIST_2 в LIST_1
020     LIST_1[i] = LIST_2[i];
021    N_L1 = N_L2;
022   } while (N_L2);                             // …пока список LIST_2 не пуст
```

Время t_j выполнения операций каждого яруса определяется временем исполнения наиболее длительной операции из расположенных на данном ярусе ($t_j = max(t_{ji})$, где j – номер яруса, i – номер оператора в данном ярусе). При планирования выполнения параллельной программы необходимо учитывать ограниченность по числу процессоров ($P \le P_{max}$), поэтому может быть выгодно объединение двух и более быстровыполняемых операторов для после-

довательного исполнения в рамках одного яруса. Как показано ниже в текущем разделе, подобная задача по трудоемкости относится к NP-полным задачам (точное решение невозможно получить за разумное время). Однако и в этом случае получение решения, значимо (в несколько раз) повышающее производительность, является ценным и должно быть использовано.

Усложняет проблему необходимость учета суперскалярности, присущей практически каждому современному процессору, причеё в разной степени (разработчиками заявлено, что процессор IBM Power5 способен выполнять за такт четыре операции с плавающей точкой, процессоры AMD Opteron и Intel Xeon EM64T - две операции с плавающей точкой за такт и так далее (*).

Конечно, в реальности распараллеливание на уровне отдельных операторов полезно в лучшем случае для параллельных компьютеров с общей памятью (SMP-систем, многоядерных процессоров); рациональнее в роли отдельного оператора рассматривать группу операторов, могущую выполняться последовательно без обмена данными с другими (возможно, параллельно выполняющимися) подобными группами команд (*гранула, зерно распараллеливания*, см. подраздел 1.3). Однако выявить гранулы (зерна) параллелизма (вручную или автоматически) непросто.

Одним из эмпирических методов проектирования алгоритмов *наименьшей высоты* (с минимальным числом ярусов) является процесс *сдваивания*. Для задачи вычисления произведений n чисел $a_1,a_2...a_n$ алгоритм последовательного вычисления для n=8 представляется так (по работе [1]):

исходные данные: $a_1\,a_2\,a_3\,a_4\,a_5\,a_6\,a_7\,a_8$

ярус 1: $a_1 \times a_2$

ярус 2: $(a_1 a_2) \times a_3$

ярус 3: $(a_1 a_2 a_3) \times a_4$

ярус 4: $(a_1 a_2 a_3 a_4) \times a_5$

ярус 5: $(a_1 a_2 a_3 a_4 a_5) \times a_6$

ярус 6: $(a_1 a_2 a_3 a_4 a_5 a_6) \times a_7$

ярус 7: $(a_1 a_2 a_3 a_4 a_5 a_6 a_7) \times a_8$

Высота этой схемы (число ярусов) равна 7, *ширина* (число операций на каждом ярусе) равна 1. Алгоритм может быть реализован на многопроцессорной вычислительной системе, но все кроме одного процессоры будут простаивать.

Использование *другого алгоритма* решения той же задачи (используется свойство ассоциативности умножения) приводит к схеме:

---

* Антонов А.С. Далеко ли до пика? //Журнал ‘Открытые системы’, № 06, 2006.

исходные данные: $a_1\, a_2\, a_3\, a_4\, a_5\, a_6\, a_7\, a_8$

ярус 1: $(a_1 \times a_2)\ \ (a_3 \times a_4)\ \ (a_5 \times a_6)\ \ (a_7 \times a_8)$

ярус 2: $(a_1 a_2) \times (a_3 a_4)\ \ (a_5 a_6) \times (a_7\, a_8)$

ярус 3: $(a_1 a_2 a_3 a_4) \times (a_5 a_6 a_7\, a_8)$

Высота такой схемы равна 3, ширина – те же самые 4. Для произвольного *n* (в этом случае высота равна $ceil(\log_2 n)$ такой алгоритм реализуется на *ceil(n/2)* процессорах (но загрузка процессоров бинарно уменьшается от яруса к ярусу). Процесс сдваивания (известен и применяется также процесс *рекуррентного сдваивания*) позволяет проектировать алгоритмы минимальной высоты для *любой ассоциативной (сочетательной) операции*, но ширина полученного алгоритма чрезмерна (на первых ярусах).

Для рассмотренных алгоритмов размер гранулы (определённый трудоёмкостью выполнения операций высокоуровневого языка программирования) столь мизерен, что его применение оправдано только для SMP-систем (многопроцессорных вычислительных систем с общей памятью); эффективное распараллеливание в МВС с локальной памятью требуют использование гранул параллелизма намного большего размера.

Подобный подход к проектированию параллельных схем выполнения алгоритмов является изящным, однако абстрагирование от числа процессоров и параметров коммуникационной среды конкретной вычислительной системы делает его практическую применимость весьма ограниченной (напр., для рассмотренного примера при $n \approx 10^3$ на первом ярусе необходимо задействовать 500 процессоров, а на последнем – всего 2!). Подход, основанный на игнорировании архитектуры МВС и количественных параметров ее компонентов, получил название *концепции неограниченного параллелизма.*

При о*ператорном подходе* в любой программе различают два типа действия – *преобразователи* (осуществляют переработку информации, напр., операторы присваивания) и *распознаватели* (определяют последовательность срабатывания преобразователей при работе программы, напр., условные операторы, переключатели и др.). Всего имеются четыре основные (традиционные) модели (см. табл.5) представления алгоритмов в виде графов (*):

Таблица 5 — Основные модели графового представления алгоритмов.

№№	*Название*	*Вершины графа*	*Дуги графа*	*Примечание*
1	Граф управ-	Соответствуют	Передача управления	Граф не зависит

---

* Ершов А.П. Современное состояние теории схем управления. // Проблемы кибернетики, -М.: 1973, № 27, с.87 ÷ 110.

	ления	экземплярам операторов-преобразователей или распознавателей	(наличие направленной дуги между вершинами соответствует выполнению операторов непосредственно один за другим)	от входных данных и создается непосредственно по тексту программы
2	Информационный граф	Соответствуют экземплярам только преобразователей	Информационные связи между операторами (*это и есть ранее рассмотренный граф алгоритма*)	То же самое
3	Операционно-логическая история	Соответствуют каждому срабатыванию (оно необязательно является единственным) каждого оператора	Соответствуют передачам управления	Граф зависит от входных данных и требует для построения отслеживания выполнения всех операторов
4	История реализации	Соответствуют каждому срабатыванию оператора-преобразователя	Соответствуют передачам информации	То же самое

На самом деле традиционные графы зависимостей устроены излишне сложно - в каждую вершину входит слишком большое число дуг, к тому же зависящее от значений внешних переменных. Для таких графов трудно получить приемлемое аналитическое представление и поэтому с их помощью трудно решать сложные содержательные задачи.

В НИВЦ МГУ ряд лет для изучения структуры алгоритмов используют т.н. *минимальные графы зависимостей* [1]. Эти графы являются подграфами традиционных графов зависимостей, однако принципиально отличаются от последних тем, что в каждую вершину такого графа входит лишь *конечное число дуг* и *число их не зависит от значений внешних переменных* программы. Для минимальных графов зависимостей получено их представление в виде конечного набора простейших функций, а оно, в свою очередь, позволяет решать широкий спектр задач в области исследования структуры алгоритмов.

Применение минимальных графов зависимостей позволило решить задачи (включая задачи, решение тесно связано с минимальными графами), [1]:

- Определение параллельной структуры алгоритмов и программ на уровне отдельных операций.
- Расчленение алгоритмов на *крупные параллельно исполняемые фрагменты*.
- Восстановление по программе *исходных математических формул*.
- Оптимизация использования внешней памяти.
- Оценка *параллельной сложности* алгоритмов и программ.
- Быстрое вычисление градиента сложных функций.

- Исследование влияния ошибок округления.
- Построение математических моделей *систолических массивов*.
- Разработка портабельных библиотек программ.

Результаты разработок использованы при создании системы V-Ray (разработка НИВЦ МГУ, http://parallel.ru/v-ray).

В общем случае анализ имеющего m дуг графа алгоритма с целью выявления параллелизма (напр., выявление ярусов) требует порядка m^2 переборов, при этом сам алгоритм выполняется (на однопроцессорном компьютере) за пропорциональное m время. Т.о. *время исследования* структуры алгоритма с целью выявления параллелизма значительно превышает время его выполнения! Как показано в работе (*), решение наиболее практически ценных вариантов задачи оптимального размещения операций по процессорам МВС требует числа переборов порядка $m!$ для только одного набора входных данных (т.е. по трудоемкости относится к задачам, точное решение невозможно получить за разумное время).

В большинстве (из огромного множества наработанных) последовательных алгоритмов присутствует *внутренний параллелизм* (возможность представления алгоритма в параллельной форме при сохранении его *вычислительных свойств* – численной устойчивости и др.), который может быть выявлен методом построения и анализа графа алгоритма. В сложных случаях применяются *эквивалентные преобразования* исходных алгоритмов. Графы алгоритмов для некоторых задач приведены в [1,7]. Для часто применяемых алгоритмов (напр., операции с матрицами) операции распараллеливания проведены и соответствующие программы собраны в *проблемно-ориентированные библиотеки* (*пакеты*).

Практического использование выявленных возможностей распараллеливания требует отображения графа алгоритма на конкретную многопроцессорную вычислительную систему, для чего необходимо иметь описание МВС - количественные параметры процессоров и коммуникационной среды (на первом этапе анализа часто принимается одинаковое время выполнения любых вычислительных операций и мгновенный обмен данными между процессорами, при учете времени обмена данными проводится *анализ коммуникационной трудоёмкости* параллельного алгоритма). Именно на этом этапе можно определить:

- Время выполнения параллельного алгоритма.

---

* Гэри М., Джонсон Д. Вычислительные машины и труднорешаемые задачи. –М.: Мир, 1982, -416 с.

- Ускорение (отношение времени решения задач на скалярной ЭВМ к времени выполнения параллельного алгоритма).
- Эффективность (величина, определяющая *усредненную* долю времени выполнения алгоритма, в течение которой процессоры на самом деле используются для решения задачи).

На основе этих данных строится *описание параллельного выполнения алгоритма*, один из вариантов построения которого приведен в книге (**) и работе [3]).Неравномерность загрузки процессоров снижает эффективность использования МВС, поэтому *проблема балансировки* (равномерности загрузки процессоров) относится к числу важных задач параллельного программирования.

Один из действенных подходов к обеспечению равномерности загрузки процессоров состоит в организации всех готовых к выполнению в системе вычислительных действий в виде *очереди заданий*. При вычислениях любой освободившийся процессор запрашивает для себя работу из этой очереди; появляющиеся по мере обработки данных дополнительные задания дополняют очередь. Эта схема балансировки вычислительной нагрузки между процессорами проста, наглядна и эффективна, что позволяет говорить об использовании очереди заданий как об *общей модели организации параллельных вычислений* для систем с общей памятью.

При теоретическом анализе применяется также понятие *стоимости параллельного алгоритма* (произведение сложности алгоритма на число используемых процессоров). Известно, например, что оптимальный алгоритм сортировки требует $O(n \times \log n)$ операций (n – число сортируемых записей, см. (***)), тогда для параллельного алгоритма со сложностью $O(n)$ коэффициент ускорения составит $O(\log n)$, а стоимость параллельного алгоритма на p процессорах равна $O(n \times p)$. Т.к. стоимость алгоритма последовательной сортировки на одном процессоре совпадает с его сложностью и равна $O(n \times \log n)$, алгоритм параллельной сортировки *обходится дороже параллельной* при $\frac{n \times p}{n \times \log n} = \frac{p}{\log n} > 1$ (т.е. $p > \log n$) и *дешевле* в противоположном случае (при $p < \log n$). В случае $p = n$ стоимость параллельной сортировки всегда выше последовательной (n всегда больше $\log n$).

** Dimitri P.Bertsekas, John N.Tsitsiklis. Parallel and distributed Computation. Numerical Methods. - Prentice Hall, Englewood Cliffs, New Jersey, 1989.

*** Макконел Дж. Основы современных алгоритмов. –М.: Техносфера, 2004, -368 с.

3.2 Потенциал распараллеливания циклов. Циклы ParDO

Практика программирования (особенно научно-технических задач – в первую очередь для сеточных методов) показывает, что наибольший *ресурс параллелизма* сосредоточен в циклах, поэтому распространённым способом распараллеливания является *распределение итераций циклов* (если между итерациями не существует информационных зависимостей, то итерации можно определённым способом распределить разным процессорам для параллельного исполнения).

Циклы, все итерации которых информационно независимы, принято называть *ParDO циклами*, из условия независимости следует, что возможно их выполнять в любом порядке (в т.ч. параллельно). Этот вид параллелизма важен на практике благодаря частой встречаемости и простоты использования. Нахождение циклов ParDO производится с помощью анализа информационной структуры программы (основанный на графовом представлении алгоритмов критерий определения цикла ParDO приведен, напр., в работе [1]). Причем найденное по графу алгоритма свойство ParDO говорит лишь о *принципиальной возможности* независимого выполнения итераций (для реализации возможности необходимо произвести определенные преобразования цикла), полностью готовые к распараллеливанию циклы обозначаются как TrueParDO.

Одними из первых программных средств, обеспечивающих некоторую автоматизацию распараллеливания, явились как раз системы выявления параллелизма в циклах. Обычно используют два типа таких систем:

- Специализированные компиляторы (компиляторы с традиционных языков программирования – обычно Fortran), функциональность которых расширена средствами выявления параллелизма, эквивалентными преобразованиями цикла и созданием соответствующего исполняемого кода (обычно рассчитанного на векторную архитектуру вычислителя).
- Препроцессоры, осуществляющие анализ и отбор циклов на векторизацию или распараллеливание, выявленные блоки часто конвертируются в допускающее параллельную обработку расширение языка (напр., MPI, см. подраздел 4.1); в дальнейшем (преобразованный) исходный текст обрабатывается стандартным компилятором.

Многие такие системы допускают диалоговый режим работы – при этом автоматически выявляются блоки-кандидаты на распараллеливание, а окончательное решение о необходимости и способе преобразования в параллельную форму принимает разработчик.

Одним из известных и мощных препроцессоров-векторизаторов, каждый член которого настроен на конкретную объектную машину, является KAP. Система KAP использует для оптимизации *коллапс циклов*, *разбиение цикла*, *переупорядочение операторов* и *разрушение контуров зависимостей по данным, сегментацию цикла* (преобразование исходного цикла в гнездо из нескольких циклов при соблюдении условия непереполнения кэш-памяти).

Известными системами являются PFC (и основанная на ней инструментальная диалоговая система анализа программ PTOOL), ParaScope, BERT 77, D-система, Polaris и др., (∗). Основой для решений о распараллеливании является граф зависимостей по данным (строится по исходному тексту программы), в некоторых случаях используется дополнительная информация о глубине зависимости для гнезда циклов или векторов направлений зависимости. Исследование графа выявляет как области нераспараллеливания или последовательного исполнения, так и структуру таких областей. Например, изучение контуров зависимости на предмет однородности дуг контура может показать, что контур может быть разрушен. Здесь возможно применение преобразований типа *переименования переменных*.

В нашей стране также разрабатывались системы подобного типа - напр., система программирования для ПС-3000 для МВК ПС-3000 (включает язык программирования Фортран'77ВП, являющийся расширением стандарта Fortran'77 средствами организации векторных и параллельных вычислений), Система М10 (*М.А.Карцев*), конвертер-векторизатор Фора-ЕС (для специализированного векторного процессора СПЕС, Институт прикладной математики АН СССР), векторизующий ПЛ/1-компилятор ИПК АН СССР (для машин Cray, Институт проблем кибернетики АН СССР) и др.

Из современных разработок рекламируется система ОРС (Открытая Распараллеливающая Система, *Open Parallelizing System Group;* Ростовский государственный университет, http://ops.rsu.ru) - программная инструментальная система, ориентированная на разработку распараллеливающих компиляторов, оптимизирующих компиляторов с параллельных языков, систем полуавтоматического распараллеливания (так система позиционируется разработчиками). ОРС базируется на интерактивном характере распараллеливания (разработана мощная система визуализации), использует различные графовые модели программ, эквивалентные преобразования выражений, одномерных циклов и гнезд циклов.

При ручном анализе и преобразовании циклов используют наработанные эмпирические методы, часть из них описана ниже.

∗ В.А.Евстигнеев, И.Л.Мирзуитова. Анализ циклов: выбор кандидатов на распараллеливание. // Препринт. Российская академия наук Сибирское отделение Институт систем информатики им. А. П. Ершова. –Новосибирск, 1999, -41 с.

Приведём пример программирования в технологии MPI (подробнее см. подраздел 4.1), согласно MPI все процессоры выполняют одинаковый исполняемый код, а необходимые каждой параллельной ветви действия определяются использованием условных операторов; один из процессоров (обычно нулевой) является *управляющим*, остальные - *рабочими*.

Задача заключается в суммировании N чисел, последовательно расположенных в массиве A[ ], число процессоров равно N_P (диапазон 0÷N_P-1); предполагается, что каждый процесс "знает" собственный номер (переменная I_AM):

```
int k   = N / N_P;                          // целое от деления N на N_NP
int k1 = N % N_P;                           // остаток от деления N на N_P
 if ((I_AM > 0) && I_AM < N_P-1) {          // для всех процессов кроме нулевого и последне-
                                            го…
  s = 0;
  for (i=k∗(N_P-1); i<k∗N_P; i++)           // суммируем k элементов массива A[ ]
    s += A[i];                              // s – локальная для каждого процесса переменная
  послать s управляющему процессу           // оператор не конкретизируется
  } // конец блока if ()
else
 if (I_AM == N_P-1) {                       // для последнего процессора…
  s = 0;
  for (i= k∗(N_P-1); i< k∗N_P+k1; i++)      // суммируем k+k1 элементов массива A[ ]
    s += A[i];                              // s – локальная для каждого процесса переменная
  послать s управляющему процессу           // оператор не конкретизируется
  } // конец блока if ()
```

Здесь каждый процессор (кроме нулевого и последнего) осуществляет суммирование очередных K элементов массива A[ ], последний процессор суммирует оставшиеся K+K1 элементов. Последним действием (выполняющимся *после всех приведённых* – именно здесь необходима синхронизация) является суммирование всех частичных сумм, полученных N_P-1 процессорами:

```
if (I_AM == 0) {                            // выполняется управляющим процессом
 sum = 0;
 for (j=1; j<N_P; j++) {                    // по всем процессам 1÷N_P
   принять значение s от процесса j         // оператор не конкретизируется
   sum += s;                                // готовое значение – в переменной sum
  } // конец блока for ()
 } // конец блока if ()
```

При излишне большом размере массива A[ ] процесс распределения его про процессам может повторяться многократно (что способствует экономии ОП каждого процессора), общее ускорение выполнения программы будет, конечно, несколько меньше N_P (*сетевой закон Амдаля*, см. подраздел 1.5). При разработке параллельной программы желательно использовать алгоритм, абстрагирующийся от количества процессоров (программа должна выполняться на МВС с числом процессоров ≥ 2).

Приведённый пример иллюстрирует *блочное распределение* итераций по процессорам и является типовым для множества алгоритмов, основными действиями в которых являются обладающие свойством ассоциативности операций сложения/умножения (матричные операции, поиск экстремумов в массивах, интегрирование и многие др.). Преимуществом его является сбалансированность загрузки "рабочих" процессоров (все кроме одного выполняют одинаковую работу, последний "перерабатывает" менее чем на $\frac{100\%}{N_P-1}$).

Заметим, что условием сбалансированности загрузки процессоров является приблизительно равноценные по времени исполнения распределяемые итерации (рассматриваемый пример удовлетворяет этому условию). Таким условиям (с некоторыми оговорками) соответствует, например, распределение по процессорам вычисление одной и той же функции в нескольких точках (при поиске экстремумов функции многих переменных градиентными методами, численного интегрирования или дифференцирования и др.). *Циклическое распределение* итераций предполагает выполнение всеми процессорами по одной последовательной итерации цикла (для простых операций в теле цикла размер гранулы может при этом оказаться чересчур мелким).

В программах часто встречаются *многомерные циклические гнёзда*, причем каждый цикл такого гнезда может содержать некоторый ресурс параллелизма. *Пространство итераций* (Lamport L., 1973) гнезда тесно вложенных циклов называют множество целочисленных векторов, координаты которых задаются значениями параметров циклов данного гнезда. Задача распараллеливания в этом случае сводится к разбиению этого множества векторов на выполняющиеся друг за другом последовательно подмножества, однако для каждого подмножества итерации могут быть выполнены параллельно (если внутри подмножества *не существует информационных зависимостей*).

Среди методов анализа пространства итераций известны *методы гиперплоскостей, координат, параллелепипедов и пирамид* (*). При использовании *метода гиперплоскостей* пространство итераций размерности n разбивается на гиперплоскости размерности $n-1$ таким образом, что все операции, соответствующие точкам одной гиперплоскости, могут выполняться параллельно. *Метод координат* заключается в том, что пространство итераций фрагмента разбивается на гиперплоскости, перпендикулярные одной из координатных осей. *Метод параллелепипедов* является логическим развитием двух предыдущих методов и заключается в разбиении пространства итераций на n-мерные параллелепипеды, объем которых определяет результирующее ускорение программы. При использовании *метода пирамид* находятся итерации, вырабатывающие значения, которые далее в теле гнезда циклов не использу-

---

* А.С.Антонов. Введение в параллельные вычисления (методическое пособие). // Изд. МГУ им.М.В.Ломоносова, НИВЦ. -М.: 2002, -69 с.

ются; каждая такая итерация служит основанием отдельной параллельной ветви, в которую входят и все итерации, информационно влияющие на выбранную (неприятным эффектом при этом является дублирование информации в разных ветвях, что может приводить к потере эффективности).

Известны методы (напр., *метод линейного преобразования*), являющиеся более мощными, нежели метод гиперплоскостей и даже *метод координат Лэмпорта*, (**).

Пространство итераций для простого цикла

```
for (i = 1; i < N; i++)
 for (j = 1; j < M; j++)                                          (6)
   A[i][j] = A[i-1][j] + A[i][j];
```

приведено на рис.21 (окружности соответствуют отдельным операторам вычисления и присваивания, стрелки - информационным зависимостям).

Из рис.21 видно, что разбиение пространства итераций по измерению I приводит к разрыву информационных зависимостей; для измерения J таково**о**го не происходит, поэтому возможно применение метода координат с разбиением пространства итераций гиперплоскостями, перпендикулярными оси J (на рис.21 – пунктир). Эти операции назначаются для выполнения на одном процессоре.

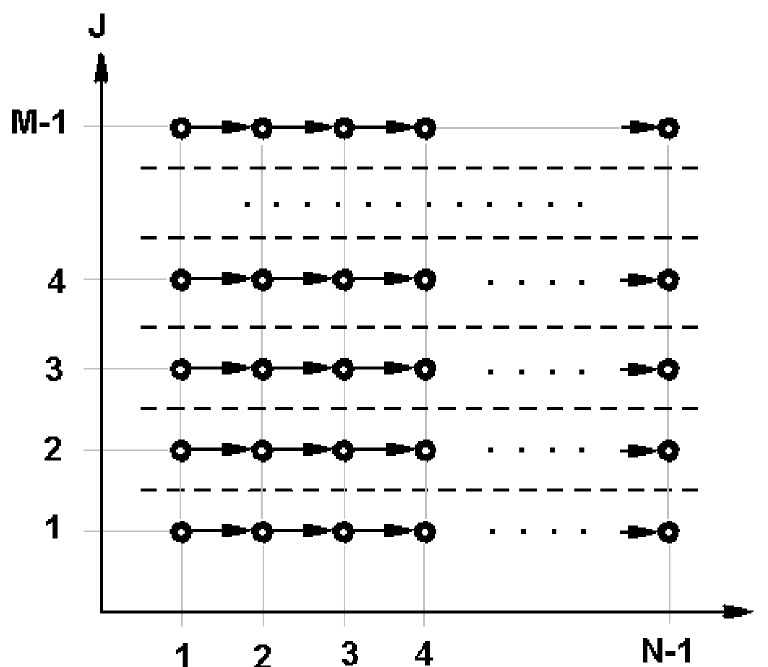


Рисунок 21 — Пространство итераций для фрагмента (6)

Для гнезда циклов

```
for (i = 1; i < N; i++)
  for (j = 1; j < M; j++)
(7)
    A[i][j] = A[i-1][i] + A[i][j-1];
```

метод координат неприменим, т.к. любое разбиение (и по измерению I и по измерению J) перпендикулярными осям координат плоскостями приводит к разрыву информационных зависимостей. Несмотря на это можно заметить, что существуют удовле-

---

** Lamport L. The coordinate method for the parallel execution of DO-loops. // Sagamore computer conference on parallel processing, 1973.

творяющие условию i+j=const гиперплоскости (пунктир на рис.22, проходящие через не содержащие информационных зависимостей вершины; при этом параллельно работающим процессорам следует назначить вычисления тела цикла для каждой из гиперплоскостей i+j=const.

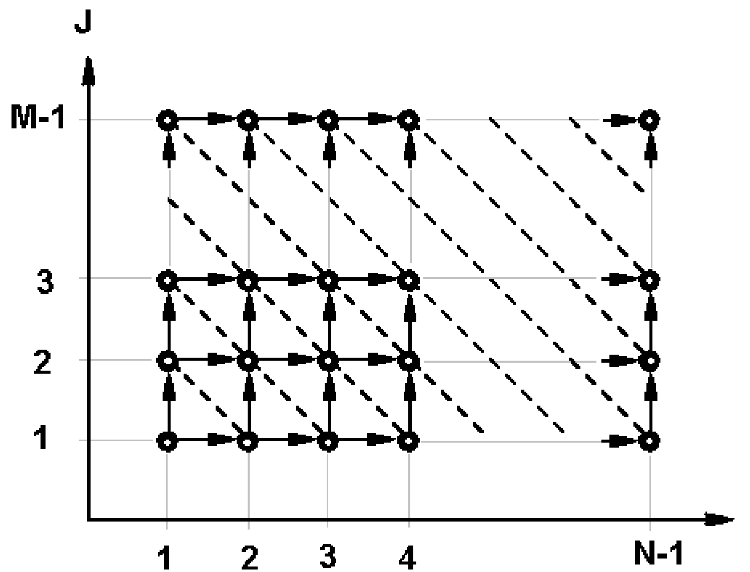


Рисунок 22 — Пространство итераций для фрагмента (7)

Часто удаётся добиться повышения эффективности распараллеливания программы при помощи *эквивалентных преобразований* - таких преобразований кода программы, при которых полностью сохраняется результат ее выполнения при изменении последовательности действий.

Первым этапом работы по распараллеливанию является выявление зависимостей; второй шаг - распараллеливание циклов с использованием результатов первого этапа. Примеры автоматических (выполняемых распараллеливающим компилятором) эквивалентных преобразований в применении к циклам приведены ниже в табл.6.

Если последовательная программа содержит два оператора S_1 и S_2, причем S_1 находится перед S_2, то говорят, что между этими двумя операторами существует *зависимость по данным*, если они считывают или записывают данные в общей области памяти таким образом, что порядок их выполнения нельзя изменять без изменения конечного результата вычислений. Имеются три основных типа зависимости по данным:

- Потоковая зависимость - оператор S_2 потоково зависит от S_1, если S_2 считывает из ячейки, в которую записывает S_1 (такая зависимость называется *истинной*).
- Антизависимость - оператор S_2 является антизависимым относительно S_1, если S_2 записывает в ячейку, из которой S_1 считывает.
- Зависимость по выходу - оператор S_2 зависит по выходу от S_1, если оператор S_2 записывает данные в ту же ячейку памяти, что и S_1.

Часто просто говорят "S_2 зависит от S_1", если между ними существует зависимость по данным (тип зависимости не важен). Зависимости по данным легко определяется в последовательном коде, содержащем ссылки только на скаляры. Намного труднее определить зависимости в циклах и при ссылках на массивы (что обычно встречается совместно), поскольку ссылки в массивы имеют индексы, а в индексах часто используются параметры циклов, т.е. индексы массива имеют различные значения на разных итерациях цикла. Общая проблема вычисления всех зависимостей по данным в программе неразрешима из-за синонимичности имен массивов, которая возникает при использовании указателей или вызовов функций внутри индексных выражений. Даже если указатели не разрешены (как в Fortran'е) и индексные выражения являются линейными функциями, проблема является NP-трудной, т.е. эффективного алгоритма решения её скорее всего не существует.

Ниже рассматриваются несколько полезных преобразований для циклов: *локализация, расширение скаляра, распределение цикла, слияние циклов, развёртка и сжатие, развёртка цикла, разделение на полосы, разделение на блоки*, *перекос цикла*, которые помогают выявлять параллельность, устранять зависимости и оптимизировать использование памяти:

- Перестановка циклов - внешний и внутренний циклы меняются местами.
- Локализация - каждому процессу дается копия переменной.
- Расширение скаляра - скаляр заменяется элементом массива.
- Распределение цикла - один цикл расщепляется на два отдельных цикла.
- Слияние циклов - два цикла объединяются в единый.
- Развертка и сжатие - комбинируются перестановка циклов, разделение на полосы и развертка.
- Развёртка цикла - тело цикла повторяется, благодаря чему выполняется меньше итераций.
- Разделение на полосы - итерации одного цикла разделяются на два вложенных цикла.
- Разделение на блоки - область итераций разбивается на прямоугольные блоки.
- Перекос цикла - границы цикла изменяются таким образом, чтобы выделить параллельность фронта волны.

Распределение циклов используется для устранения зависимости в цикле (фактически для выявления параллелизма). Если исходный цикл имеет вид (здесь и далее используется нотация языка программирования, равноудалённая и от Fortran'а и от C)

```
for [i=1 to n] {
 a[i] = … ;
```

```
 …   = … + a[i-1];
}
```

Здесь второй оператор тела цикла зависит от результата, вычисленного первым на предыдущей итерации. При условии отсутствия иных зависимостей (кроме указанной) цикл распределяется следующим образом (каждый из двух циклов легко распараллеливается, хотя сами циклы должны выполняться последовательно):

```
for [i=1 to n]
 a[i] = … ;
for [i=1 to n]
 …   = … + a[i-1];
```

При *слиянии* циклы, имеющие *одинаковые заголовки* и *независимые тела*, объединяются (сливаются) в единый цикл; слияние приводит к уменьшению числа процессоров и их укрупнению (более крупнозернистый параллелизм).

Развёртка и сжатие приводит к сближению обращений к ячейкам памяти, при этом производительность повышается за счет кэширования. В примере умножения матриц:

```
for [i=1 to n]
 for [j=1 to n]
  for [k=1 to n
   c[i, j] = c[i, j] + a[i, k] * b[k, j];
```

сначала *разворачивается* внешний цикл, затем полученные циклы *сжимаются* (*сливаются*) таким образом, чтобы операторы сложения оказались в цикле максимальной вложенности. После развертки внешнего цикла и сжатия вышерасположенного фрагмента получим (считая n чётным):

```
for [i=1 to n by 2]  // половина всех итераций (только нечётные i)
 for [j=1 to n]
  for [k=1 to n]                                                  (8)
  {
    c[i, j]    = c[i, j]     + a[i, k] + b[k, j];
    c[i+1, j] = c[i+1, j] + a[i, k] + b[k, j];
  }
```

Здесь в каждом внешнем цикле вычисляются два скаляра c[i, j] и c[i+1, j], располагающиеся в смежных ячейках памяти (при условии, что матрицы в ОП *хранятся по столбцам* – что характерно для Fortran'а и отнюдь не для C/C++). А что мешает в каждом внешнем цикле вычислять не 2, а 3,4,5.. скаляров (конечно,

если n не делится нацело на это число, понадобится некоторое дополнение алгоритма)?

В случае *разделения на полосы* цикл делится надвое, причём во внешнем цикле перебираются полосы, а внутренний итерирует все элементы данной полосы. Например, для умножения матриц единократное развёртывание внешнего цикла равноценно разделению полосы шириной 2 итерации (дополнительные итерации производятся по индексу i1):

```
for [i=1 to n by 2] // половина всех итераций (только нечётные i)
 for [i1=i to i+1]   // полоса из двух итераций
  for [j=1 to n]
    for [k=1 to n]
     c[i1, j] = c[i1, j] + a[i1, k] + b[k, j];
```

Преобразование развертки и сжатия равноценны разделению на полосы внешнего цикла, перемещению разделённого цикла на место внутреннего и его развертки (цикл максимальной вложенности использует индекс i1):

```
for [i=1 to n by 2] // половина всех итераций (только нечётные i)
 for [j=1 to n]
   for [k=1 to n]
    for [i1=1 to i+1] // полоса из двух итераций
     c[i1, j] = c[i1, j] + a[i1, k] + b[k, j];
```

После развёртки внутреннего цикла (т.е. замены i1 двумя значениями – i и i+1) получается программа, равноценная полученной преобразованием развёртки и сжатия (8).

При *разделении на блоки* распределение данных в памяти компактируется и, следовательно, повышается эффективность кэширования на МВС с общей памятью или размещение данных в распределенной памяти. В качестве примера рассматривается фрагмент транспонирования матрицы:

```
for [i=1 to n]
 for [j=1 to n]
  b[i, j] = a[j, i];
```

При условии хранения в памяти массивов по строкам (характерно для C/C++) доступ к элементам матрицы b осуществляется с шагом 1 (каждый элемент b размещается рядом с предыдущим вычисленным элементом b в соответствии с внутренним циклом по j). Однако доступ к элементам матрицы a осуществляется с шагом n (длина всей строки). Т.о. при ссылках на a кэш используется неэффективно (в кэш загружается излишне много значений элементов a, из которых реально нужна всего 1/n).

Разбиение матрицы на блоки делит область итераций на прямоугольники (квадраты в частном случае), размер данных в которых *не превышают размера кэша*. При желании полностью поместить в кэш $m \times n$ (само собой, $m<n$) чисел (значений матриц a или b) следует изменить алгоритм транспонирования таким образом:

```
for [i=1 to n by m]  // для каждой m-й строки
 for [j=1 to n by m]  // и n-ного столбца…
  for [i1=i to min(i+m-1, n)]  // блок размером m строк
   for [j1=j to min(j+m-1, n)]   // блок размером m столбцов
     b[i1, j1] = a[j1, i1];
```

Теперь в двух внутренних циклах доступен квадрат из $m \times m$ элементов матриц a и b, который полностью помещается в кэш. Преобразование разделения на блоки является комбинацией разделения на полосы и перестановки циклов.

Преобразование *перекос циклов* используется с целью выделения скрытой параллельности в основном вдоль *волновых фронтов* (в этом случае обновления величин в массивах распространяется подобно волне). Типичный пример – решение дифуравнений в частных производных методом Гаусса-Зейделя при 5-точечном шаблоне на равномерной квадратной сетке размером $n \times n$ [3]:

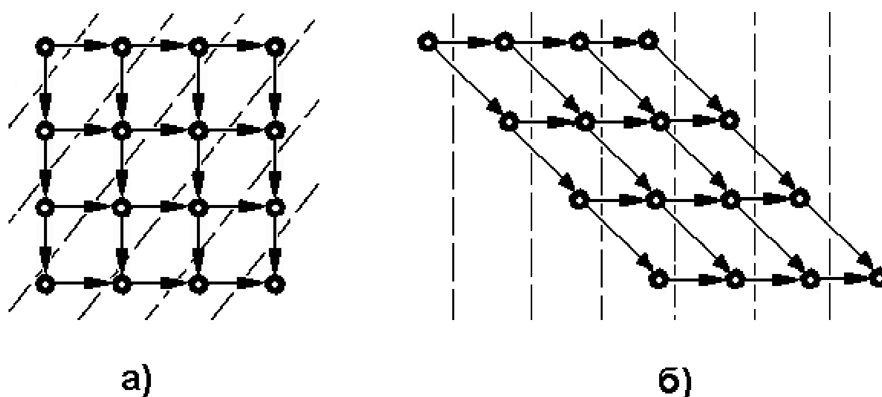


Рисунок 23 — Зависимости по данным в методе итераций Гаусса-Зейделя: а) – исходное пространство итераций, б) – пространство итераций после сдвига цикла.

```
for [i=1 to n]
 for [j=1 to n]
  a[i, j] = 0.25 × (a[i-1, j]) + a[i, j-1] + a[i+1, j] + a[i, j+1]);
```

При перекосе цикла смещаются границы цикла таким образом, чтобы волновые фронты устанавливались не по диагонали, а по столбцам (что иллюстрировано схемой рис.23). Перекос осуществляется методом прибавления к граничным значениям индекса внутреннего цикла числа, кратного величине индекса внешнего цикла, затем в теле внутреннего цикла из значения индекса это число вычитается. Образующий кратное сомножитель называется *сомножителем пе-*

рекоса. Преобразованный т.о. вышеприведенный фрагмент при равном 1 сомножителе перекоса приведен ниже:

```
for [i=1 to n]
 for [j=i+1 to i+n]  // прибавить i к имеющимся границам          (9)
  a[i, j-i] = 0.25 × (a[i-1, j-i]) + a[i, j-1-i] + a[i+1, j-i] + a[i, j+1-i]);
```

Перекос цикла позволил выявить параллельность, ибо после преобразований зависимостей в *каждом столбце* области итераций больше нет, однако зависимости остались во *внешнем цикле*.

Из фрагмента (9) видно, что границы внутреннего цикла зависят от значений индекса внешнего. Однако границы исходных циклов независимы, поэтому выполнить перестановку циклов возможно. После перестановки циклов получаем для общего случая (Li, Ui – нижняя и верхняя границы исходного внешнего цикла, Lj и Uj – то же для внутреннего цикла, k – сомножитель перекоса):

```
for [j = (k×LI + Lj) to (k×Ui + Uj)]
 for [i  = max(Li, ceil((j - Uj) / k)) to min(Ui, ceil(j - Lj) / k))]
  ...
```

Конкретное применение этого преобразования для метода Гаусса-Зейделя приводит к такому коду:

```
for [j=2 to n+n]
 for [i=max(1, j-n) to min(n, j-1)]
  a[i, j-i] = 0.25 × (a[i-1, j-i]) + a[i, j-1-i] + a[i+1, j-i] + a[i, j+1-i]);
```

После этого внутренний цикл несложно распараллелить, напр., путём создания процессов для каждой итерации внешнего цикла.

Несмотря на наличие аппарата выявления информационных зависимостей в циклах, претендующий на опыт распараллеливания алгоритмов программист должен приучаться видеть параллелизм в простых конструкциях, например

```
for (i = 1; i < N; i++) {
 A[i] = A[i] + B[i];                                              (8)
 C[i] = C[i-1] + B[i];
}
```

В приведённом фрагменте существует зависимость по информации между i-той и (i-1)-й итерациями в операторе C[i] = C[i-1] + B[i], поэтому просто распределить итерации между процессорами нельзя. Однако фрагмент (8) легко разбить (операция *распределение циклов*) на два одномерных цикла (т.к. опе-

раторы A[i] = A[i] + B[i] и C[i] = C[i-1] + B[i] информационно независимы), причём первый эффективно распараллеливается:

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

3.3 Эквивалентные преобразования алгоритмов, избыточность вычислений и обменов данными

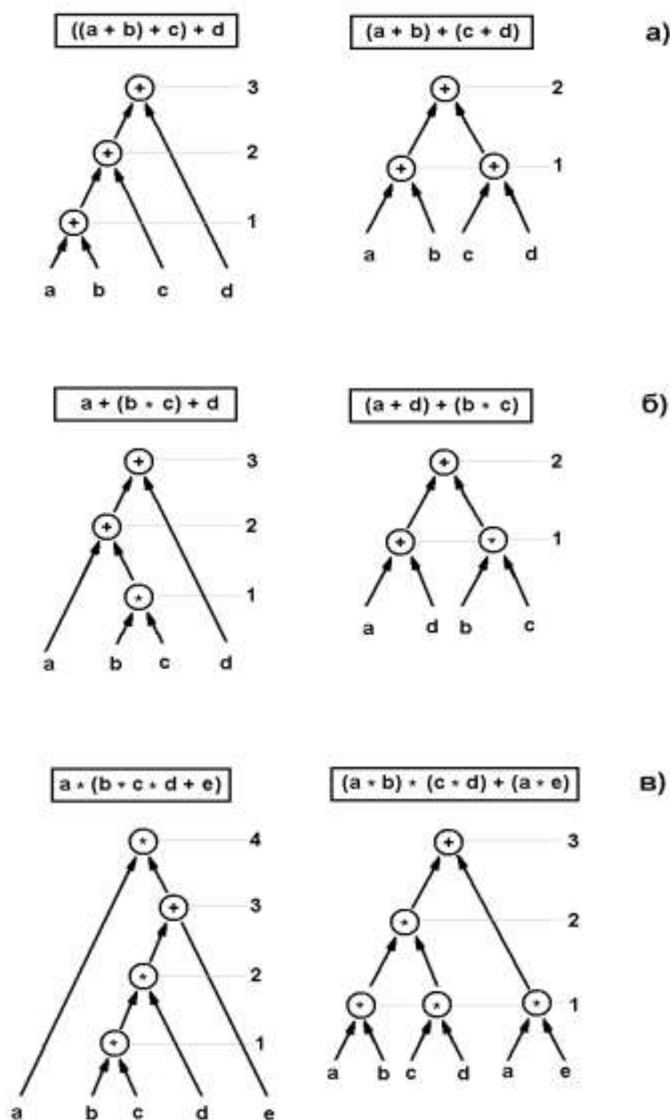


Рисунок 24 — Уменьшение высоты дерева: а) - за счёт ассоциативности, б) - за счёт коммутативности и в) - за счёт дистрибутивности (слева – обычный компилятор, справа - распараллеливающий)

Эквивалентным преобразованием (ЭкП) алгоритма является замена данного алгоритма (или его части) на алгоритм, гарантирующий получение такого же конечного результата на всех наборах входных данных в точной арифметике (учёт ограниченной машинной точности представления данных может накладывать ограничения на ЭкП).

Одним из ЭкП, достаточно часто используемое в распараллеливающих компиляторах, является *редукция высоты дерева*. Один из методов реализации такого преобразования – использование ассоциативных, коммутативных и дистрибутивных свойств арифметики, таким путем компиляторы могут обнаруживать неявный параллелизм в алгебраических выражениях и генерировать объектный (а затем и исполняемый) код для многопроцессорных систем с указанием операций, которые

можно выполнять одновременно.

На рис.24 показано (в виде графа алгоритма), как оттранслировал бы алгебраическое выражение обычный компилятор (слева) и как оно может быть обработано распараллеливающим компилятором (справа); при этом не принимается во внимание возможное изменение точности вычислений. Как видно, во многих случаях возможно уменьшить число ярусов (редуцировать высоту дерева) путём выявления (ранее скрытого) параллелизма в вычислениях.

Не следует забывать, что подобная оптимизация программ совсем не бесплатна – необходимы увеличенные затраты времени и вычислительных ресурсов в период анализа/компиляции.

Очень эффективны эквивалентные преобразования при оптимизации циклов (гнёзд циклов). В табл.6 приведены некоторые преобразования циклов, автоматически осуществляемые распараллеливающим Fortran-компилятором Lahey/Fujitsu Fortran'95 (LF95, http://www.lahey.com) для Linux (LF95'Pro включает возможность автоматической и OpenMP-параллелизации). Распараллеливающие компиляторы постоянно совершенствуются и в настоящее время пригодны для создания эффективных параллельных программ с разделяемыми переменными; особенно это касается научных программ, содержащих много циклов и длительных вычислений.

В табл.6 приведены некоторые преобразования циклов, автоматически осуществляемые распараллеливающим Fortran-компилятором Lahey / Fujitsu Fortran'95 (LF95, http://www.lahey.com) для Linux (LF95'Pro включает возможность автоматической и OpenMP - параллелизации). Распараллеливающие компиляторы постоянно совершенствуются и в настоящее время пригодны для создания эффективных параллельных программ с разделяемыми переменными; особенно это касается научных программ, содержащих много циклов и длительных вычислений.

Таблица 6 — Операции над массивами и автоматическая параллелизация (на примере Fortran-базированной распараллеливающей системы LF95'Pro).

№№	*Преобразование*	*До преобразования*	*После преобразования*	*После распараллеливания*	
1	Сечения (slicing) циклов	do i=1, 50000 a(i)=b(i)+c(i) end do		*Процессор 1:* do i1=1,25000 a(i1)=b(i1)+c(i1) end do	*Процессор 2:* do i2=25001,50000 a(i2)=b(i2)+c(i2) end do
2	Перестановка вложенных (nested) циклов	do i= 2,10000 do j=1,10 a(i,j)=a(i-1,j) +b(i,j)	do j=1,10 do i=2,10000 a(i,j)=a(i-1,j) +b(i,j)	*Процессор 1:* do j=1,5 do i=210000 a(i,j)=a(i-1,j)	*Процессор 2:* do j=6,10 do i=2,10000 a(i,j)=a(i-1,j)

		end do end do	end do end do	+b(i,j) end do end do	+b(i,j) end do end do
3	Слияние (fusion) циклов	do i=1,10000 a(i)=b(i)+c(i) end do do i=1,10000 d(i)=e(i)+f(i) end do	do i=1,10000 a(i)=b(i)+c(i) d(i)=e(i)+f(i) end do	*Процессор 1:* do i=1,5000 a(i)= b(i)+c(i) d(i)=e(i)+f(i) end do	*Процессор 2:* do i=5001,0000 a(i)=b(i)+c(i) d(i)=e(i)+f(i) end do
4	Приведение (reduction) циклов. Приведение циклов может быть причиной изменений в результатах (в пределах ошибок округления).	sum=0 do i=1,10000 sum=sum+a(i) end do		*Процессор 1:* sum1=0 do i=1,5000 sum1=sum1+a(i) end do	*Процессор 2:* sum2=0 do i=5001,0000 sum2=sum2+a(i) end do
				Далее частичные суммы складываются: sum=sum1+ sum2	

Во многих случаях приведённые (лежащие, впрочем, на поверхности) преобразования помогают и разработчику при создании новых программ. Однако перед реальным программированием в любом случае полезно обдумать последовательность вычислений (алгоритмизацию). Классический пример – параллельная сортировка (по аналогии с лежащим на поверхности распараллеливанием алгоритма суммирования элементов массива складывается убеждение в возможности *эффективного крупноблочного распараллеливания* методом сортировки на каждом процессоре части общего массива; однако понимание проблем, связанных с пересечением частично отсортированных множеств приходит обычно позднее); последовательные и мелкозернистые параллельные алгоритмы сортировки описаны, напр., в работе [3].

Распараллеливающие компиляторы хороши в тех случаях, когда имеются (априори корректно работающие, но не могущие быть модифицируемыми вследствие различных причин) параллельные программы и требуется повысить их эффективность путем использования многопроцессорных вычислительных систем. При создании новых программных комплексов для параллельного исполнения разработчик должен руководствоваться *разумными соображениями* при разработке программ. Вряд ли стоит надеяться, что компилятор произведет ЭкП типа известной "задачи юного Гаусса" (сведение суммы индексов к произведению, $\sum_{i=1}^{i=100} i = 50\times(100+1)$ как частный случай)! Использование правила Горнера ($a_0+a_1x+a_2x^2+...a_nx^n \equiv a_0+x(a_1+...x(a_{n-1}+xa_n)))$) хотя и уменьшает число умножений при вычислении значения полинома, но в дальнейшем практически не распараллеливается. Есть ли смысл распараллеливать процедуру сортировки на массиве из 10 (возможно, $10^2 \div 10^3$) чисел? А определение значения вычис-

лительнотрудоёмкой функции нескольких переменных (например, при градиентном методе поиска экстремума функции многих переменных или вычислении определённого её интеграла) распределить по процессорам МВС программист обязан самостоятельно!

Каждый раз (к счастью!) приходится думать!…

Избыточные вычисления возникают в случае неиспользования результатов вычислений в качестве данных для последующих вычислений; в графе алгоритма этому явлению соответствуют вершины (операторы), выходящие из которых дуги (операнды) не входит ни в какую другую вершину. Избыточные вычисления возникают обычно вследствие неаккуратности программиста, ниже приведен пример присвоения четным по сумме индексов элементам двумерной матрицы значений true и нечетным – false:

Присутствуют избыточные вычисления	Избыточные вычисления отсутствуют

`for(i=0;i<i_max;i++)`
` for(j=0;j<j_max;j++)`
`  a[i][j]=false;`
 	
`for(i=0;i<i_max;i++)`
` for(j=0;j<j_max;j++)`
`  if((i+j)%2)`
`   a[i][j]=true;`
	`for(i=0;i<i_max;i++)`
	` for(j=0;j<j_max;j++)`
	`  if((i+j)%2)`
	`   a[i][j]=true;`
	`  else`
	`   a[i][j]=false;`

Кроме избыточных вычислений можно говорить и об *избыточных пересылках* данных (обменах информацией); минимизация обменов данными особенно важна для вычислительных систем с распределенной памятью (из-за относительно больших временных задержек при передаче сообщений интенсивность взаимодействия процессоров должна быть минимальной). На рис.25 приведена упрощённая схема *ленточного алгоритма* вычисления одного из прямоугольных блоков результирующей матрицы [C]. Согласно этому алгоритму каждому процессору МВС пересылается ряд строк a_{ik}, где $i=i1...i2$, $k=1...k_{max}$ (горизонтальная лента) элементов матрицы [A] и ряд столбцов (вертикальная лента) b_{kj}, где $k=1...k_{max}$, $j=j1...j2$ элементов матрицы [B], процессор вычисляет значение элементов одного из прямоугольных блоков результирующей матрицы [C]=[A]×[B] как

```
for(i=i1; i<i2; i++)
 for(j=j1; j<j2; j++)
  for(c[i][j] = 0.0, k=0; k<k_max; k++)                    (9)
   c[i][j] += a[i][k] * b[k][j];
```

Фрагмент (9) повторяется циклически, чтобы вычисляемые прямоугольные блоки заполнили матрицу [C] целиком. Хотя для вычисления каждого блока матрицы [C] в самом деле необходимы указанные на рис.25 горизонтальная лента [A] и вертикальная лента [B], они (ленты) будут пересылаться ещё много раз при вычислении других прямоугольных блоков результирующей матрицы [C]. Одна из модификаций алгоритма лежит буквально "на поверхности" – при заданной вертикальной ленте [B] возможно вычислить не только прямоугольный блок $c_{ij}, i \in [i1 \dots i1], i \in [j1 \dots j2]$, но и *всю вертикальную ленту* $c_{ij}, i \in [1 \dots i_{max}], i \in [j1 \dots j2]$ элементов матрицы [C] (выделено пунктиром на рис.25); для этого достаточно пересылать процессорам только новые горизонтальные ленты матрицы [A] при неизменной вертикальной ленте [B].

Более подробно реализации ленточного метода матричного умножения на вычислительных кластерах рассматривается в работе [8].

Анализ избыточности пересылок привел к модификации методов умножения матриц – блочным алгоритмам Фокса (*Geoffrey Fox*) и Кэннона, в случае применения которых процессам пересылаются для обработки не ленты, а прямоугольные блоки перемножаемых матриц (эти алгоритмы требуют дополнительных обменов данными между процессорами, вычисляющим итоговую матрицу). Наличие интенсивных обменов между процессами характерно и для методов *волновой обработки данных* (wavefront or hyperplane methods).

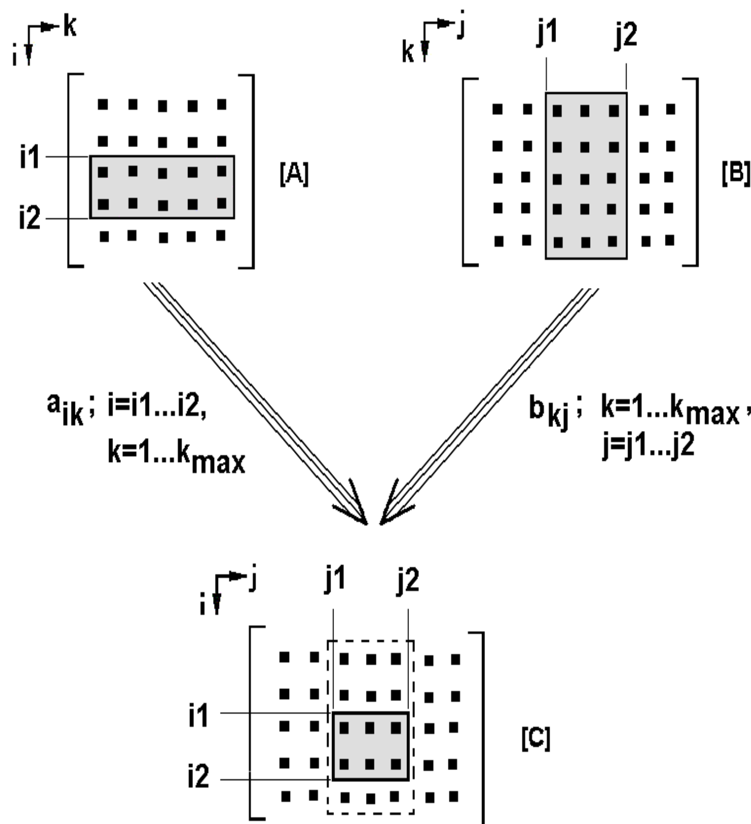


Рисунок 25 — Схема ленточного алгоритма умножения матриц [A]×[B]=[C]

Подобным же целям (конечная решаемая задача – реализация возможно более *крупноблочного параллелизма*) служит метод распределения данных с *теневыми гранями* (окружение основного обрабатываемого блока буферной зоной шириной 1 элемент данных, куда принимаются при обменах копии значений от соседних процессоров), [3].

Контрольные вопросы

1. Что такое граф алгоритма? Почему ГА является ориентированным? Чем определяется принадлежность вершин ГА к множеству входных и выходных? Что такое *ярусно-параллельная форма* алгоритма?
2. Каким образом матрица смежности представляет граф алгоритма?
3. Что такое операторы-преобразователи и операторы-распознаватели? В каких моделях графового представления алгоритмов они используются?
4. В чем заключается потенциал параллелизма в циклах? Какие методы анализа пространства итераций используются для выявления параллелизма? Что такое циклы ParDO?
5. Каким образом выявляется параллелизм системами автоматического распараллеливания? Какие эквивалентные преобразования обычно выполняются этими системами? Что такое редукция высоты дерева и на основе каких свойств арифметики она реализуется?
6. Что такое сбалансированность загрузки процессоров при распараллеливании? Каково условие сбалансированности?
7. Что такое избыточность вычислений и обменов данными? Каким путем можно свести к минимуму эти явления?
8. Количественно определить избыточность пересылок данных в примере рис.24 (пересылку каждого элемента массивов [A] и [B] считать избыточной, если она происходит *более одного раза* за время решения задачи). Как изменится эта *количественная* оценка при *лежащей на поверхности модификации* блочного алгоритма перемножения матриц?
9. Оцените количественно размер гранулы параллелизма для ленточного умножения матриц (единицу измерения выберите сами на основе анализа алгоритма).

4 Технологии параллельного программирования

Производительность параллельных вычислительных систем ещё более зависит от "интеллектуальности" программного обеспечения, нежели для обычных последовательных; при этом обычной ситуацией является разочарование (априори излишне восторженного) программиста, получившего вместо ожидаемого снижения времени вычислений существенное его повышение (даже по сравнению с последовательным вариантом программы).

При разработке программ для параллельных многопроцессорных вычислительных систем (*параллельных программ*) проблема переносимости (возможность исполнения на различных параллельных вычислительных системах) и живучести (возможность долговременного использования) также приобретает большее значение, чем при традиционном (*последовательном*) программировании. Дело, конечно, в "молодости" (и связанным с этим быстрым их прогрессом вширь и вглубь) технологий параллельных вычислений вообще. Детальный обзор технологий параллельного программирования приведён в работах [1,3,7].

Для записи параллельных программ были созданы параллельные языки. Они могут быть совершенно новыми (например, ориентированный на программирование транспьютерных устройств язык OCCAM - *C.Hoar, 1984)*, а могут быть основаны на традиционных языках - существуют параллельный C, параллельные Pascal, Fortran и даже ParJava (Институт Системного Программирования РАН, http://www.ispras.ru/~javap/parallel_java/par_java/parjava.html). Другим языком параллельного программирования, использующим объектно-ориентированный подход и поддерживающим поддерживает параллельность по задачам и по данным, является Orca (*Henri Bal*, Амстердамский университет, 1989, http://www.cs.vu.nl/orca). Язык ZPL (*Larry Snyder*, http://www.cs.washington.edu/research/zpl) обеспечивает параллельность по данным, является переносимым и достаточно производительным. Язык Cilk (*Charles Leiserson*, Massachusetts Institute of Technology, http://supertech.ics.mit.edu/cilk) расширяет ANSI C пятью (ключевые слова *cilk, spawn, synch, inlet* и *abort*) несложными инструментами для параллельного программирования, разработан для эффективного выполнения параллельных программ на симметричных мультипроцессорах с разделяемой памятью, поддерживает параллелизм по вычислениям и по данным, позволяет эффективно работать с параллельной рекурсией; с использованием 5-й версии Cilk создано несколько шахматных программ (одна из них, Cilkchess, показала хорошие результаты на всемирном компьютерном чемпионате по шахматам 14÷20.VI.1999 г. в Paderborn, Germany).

Первым функциональным языком, созданным специально для численных расчетов, стал Sisal (*Streams and Iteration in a Single Assignment Language*, Lawrence Livermore National Laboratory, 1985), вторая версия языка описана в

(*J.T.Feo, D.C.Cann, R.R.Oldehoeft*, 1990). Реализация Sisal основана на модели потоков данных (выражение можно вычислить, как только будут вычислены его операнды); Sisal-программы оказались эффективными, но к концу 90-х г.г. проект был закрыт. Язык NESL (*Guy Blelloch*, Carnegie Mellon University, 1996, http://www.cs.cmu.edu/ ~SCandAL/nesl.html) представляет собой функциональный язык с параллельностью по данным и в целом соответствует модели NESL (*концепция работы и глубины*). Функциональный язык Eden (Philipps Universität Marburg, Германии и Universidad Complutense, Мадрид, 1998, http://www.uni-marburg.de/welcome.html, http://www.ucm.es/UCMD.html) расширяет функциональный подход языка Haskell (см. далее подраздел 4.3.2) путем отмены механизма "ленивых (*отложенных*) вычислений" (*lazy evaluation)* в момент распараллеливания вычислений.

Информацию и документацию по параллельным языкам и свободно распространяемые версии можно найти на http://www.parallel.ru/tech/tech_dev/par_lang.html, http://www.hensa.ac.uk/parallel, полезные ссылки - http://computer.org/parascope.

К настоящему времени выкристаллизировались стандартные технологии, существенного (качественного) изменения которых ожидать в разумное время не приходится. Ниже кратко рассматриваются основанные на распределении данных и вычислений системы параллельного программирования HPF, OpenMP и DVM, специализированный язык mpC высокого уровня для программирования разнородных сетей, низкоуровневая технология программирования обменов сообщениями MPI, PVM и язык Linda, методы автоматизации распараллеливания вычислений (Т-система, НОРМА).

Весьма показательным является связанное с проблемой распараллеливания алгоритмов обращение к результатам (давно разрабатываемой) *теории функционального программирования* (Т-система, НОРМА; чуть подробнее см. подраздел 4.3.2). Практическая реализация положений функционального программирования фактически переводит центр тяжести с *операторного управления* процессом обработки данных на *процесс, управляемый данными* (см. подраздел 2.5).

Можно реализовать Data Flow на чисто алгоритмическом уровне (аппаратная часть остаётся Control Flow и выполняет традиционно императивные инструкции), именно для этого как нельзя подходят идеи *функционального программирования*.

4.1 Дополнения известных последовательных алгоритмических языков средствами параллельного программирования

Простейший (и исторически совершенный) путь создания системы оазработки параллельных программ – дополнить традиционные языки программирования средствами описания распараллеливания вычислений (а возмож-

но, и данных). При этом достигается минимум модифицирования существующих программ (требуется лишь некоторая "переделка") и сохраняется опыт программирования. Недостатки подхода базируются на этих же положениях – для разработки эффективной параллельной программы необходимо учитывать внутреннюю структуру алгоритма, а базовый (исходно последовательный) язык обычно не содержит даже механизмов, с помощью которых возможно реализовать это.

Естественно, на поверхности лежит попытка реализовать параллелизм в циклах (гнёздах циклов) в Fortran-программах (см. ранее подраздел 3.2). Возможность реализовалась использованием дополнительных ключевых слов ParDO (отечественная система М-10), ParDO/ParEND (система параллельного сборочного программирования ИНЯ, (*).

Метод "подсказок" компилятору с целью дополнить его функциональность средствами распараллеливания реализован, например, в системе HPF (*Hight Performance Fortran,* 1995). В HPF-системе традиционный Fortran дополнен специальными директивами, записываемыми в форме комментариев (т.е. не воспринимаемых обычным компилятором – метод решения проблемы переносимости!). С возможностью использовать HPF-программ на последовательных и параллельных машинах без изменений были связаны серьёзные надежды разработчиков, сменившиеся в дальнейшем здоровым скептицизмом (несмотря на это, в 1997 г. был разработан проект стандарта HPF2, расширявший возможности программиста в смысле директив распараллеливания, [6]).

История развития HPF тесно связана с таков**о**й языка Fortran’90. Именно в этом языке появились прототипы языковых конструкций, которые позволили компилятору эффективно генерировать параллельный код. Этот язык не является в прямом смысле параллельным, его создатели ставили себе более общие цели - создать машинно-независимый инструмент обработки числовых научных данных. Основные возможности параллелизации в Fortran’90 связаны с циклами (операциям над массивами, именно на эти действия приходится большее время выполнения научно-технических задач и именно их целесообразно распараллеливать в первую очередь). Мощным средством работы с массивами в Fortran’90 является возможность использования их в арифметических операциях подобно обычным скалярам. Многие встроенные операции в языке Fortran’90 можно применить и к массивам, получая при этом массив такой же формы, каждый элемент которого будет иметь значение, равное результату заданной операции над элементами массивов. Возможно выделять секции в массивах и оперировать с ними как с новыми массивами и др., причём порядок, в котором совершаются поэлементные операции, стан-

---

* В.Э.Малышкин. Основы параллельных вычислений. // Учебное пособие, часть 2. ЦИТ СГГА. –Новосибирск, 2003.

дартом не устанавливается. Это дает возможность компилятору обеспечить их эффективную обработку на векторном или параллельном вычислителе. Синтаксис Fortran'90 помогает компилятору в большинстве случаев избавиться от применения сложных алгоритмов определения параллельных свойств циклических конструкций. Дальнейшее развитие "околофортрановского" направления привели к разработке Fortran'D (1992) и Fortran'Vienna, HPF (1992), инструментального пакета анализа программ ParaScope (*Rice University*), средства автоматического распараллеливания Fortran-программ BERT 77, а также системы Forge (*Applied Parallel Research, Inc*., см. ниже).

Система программирования HPF – типичная система с императивным указанием программиста компилятору, каким образом *распределить данные между процессорами* (вычисления распределяются на основе распределения данных – *модель параллелизма по данным*). Распределение данных управляется директивами ALIGN (определение соответствия между взаимным расположением элементов нескольких массивов) и DISTRIBUTE (отображение группы массивов на решетку процессоров), при этом допустимо "разрезание" массива гиперплоскостями на располагающиеся на различных процессорах блоки и динамическое (в ходе выполнения программы) перераспределение (директивы REALIGN и REDISTRIBUTE). Параллелизм по вычислениям в HPF реализован по следующим конструкциям языка – операции над секциями массивов (параллелизм вычислений детерминируется ранее заданным распределением данных, при необходимости используется пересылка данных), циклы DO и операции FORALL (компилятором производится попытка обобщить эти конструкции в виде операций над секциями массивов). Пример HPF-программы приведен в Приложении 1б (исходно-последовательный вариант программы – Приложение а). HPF-препроцессор Adaptor (http://www.gmd.de/SCAI/lab/adaptor/ adaptor_home.html) распространяется (в упрощенной версии) свободно; автоматическое построение HPF-приложений осуществляется утилитой gmdhpf, поочерёдно вызывающей HPF→Fortran-препроцессор (утилита fadapt), Fortran-компилятор и компоновщик.

Программирование на HPF (существуют варианты для C/C++) с точки зрения программиста близко к идеалу (огромное поле для интеллектуальной деятельности, нет необходимости выписывать громоздкие конструкции обмена данными в стиле MPI), однако эффективность создаваемых HPF-компиляторами параллельных программ невысока (оставшаяся после указаний программиста часть процесса распараллеливания компилятором автоматически распараллелена полностью быть не может). В целом HPF-подход оказался чрезмерно романтичным (с точки зрения соответствия конструкциям языка программирования поставленным целям).

Другой подход основан на модели *параллелизма по управлению* (work-sharing, *модель распределения работы*) и формально заключатся в дополнении языков программирования специальными конструкциями управления –

параллельными циклами и *параллельными секциями*. Глубинная основа подхода заключается в представлении параллельной программы системой *нитей* (нить - *thread* – упрощенный процесс, взаимодействующий с другими нитями посредством общих переменных и простейшей синхронизации). При этом вспомогательную работу (создание и уничтожение нитей, распределение между ними витков параллельных циклов и параллельных секций – вызовов процедур, напр.) выполняет компилятор.

Первая попытка разработки подобного подхода относится еще к 1990 г., но только в 1997 г. появился стандарт языка OpenMP Fortran (позднее появились аналоги для C и Fortran'90/95, http://openmp.org). Программа на OpenMP начинает свое выполнение в виде единственного (именуемого *главной нитью*) процесса. Главная нить выполняется последовательно до тех пор, пока управление не дойдет до начала первой параллельной области программы (описываемой директивами PARALLEL и END PARALLEL). При входе в параллельную область главная нить порождает определённое число подчиненных ей нитей, образующих вместе с ней *текущую группу нитей*, при выходе из параллельной области все порождённые при входе в нее нити сливаются с главной нитью (в целом последовательно реализуется серия параллельных процессов, многократно повторяющая диаграмму рис.3б). Все находящиеся внутри параллельной области операторы (включая вызовы процедур) выполняются всеми нитями текущей группы параллельно до момента выхода из параллельной области или встречи одной из инструкций распределения работы DO (распределение витков цикла между нитями), SECTIONS (распределение между нитями заданных секций программы), SINGLE (указание секции, которая должны выполняться единой нитью). Для организации вычислений используются высокоуровневые директивы синхронизации. Пример простой программы с использованием OpenMP приведен в Приложении 2.

Программирование в технологии OpenMP в целом проще, чем на HPF, однако распределение данных в OpenMP практически никак не описывается (понятно, что OpenMP идеален для многопроцессорных вычислительных систем с общей памятью). Перспективным можно было бы считать дополнение OpenMP возможностями распределения данных, однако реально в настоящее время развивается подход, объединяющий OpenMP (средство программирования процессов) и MPI (механизм объединения процессов) - *гибридная модель параллелизма по управлению с передачей сообщений*, [6].

Примером системы, объединяющей распределение как вычислений так и данных, является DVM (*Distributed Virtual Memory* или *Distributed Virtual Machine*, 1994, http://keldysh.ru/dvm), существуют версии Fortran'DVM и C'DVM. Система разработки параллельных программ DVM разработана в ИПМ им. М.В.Келдыша при участии студентов и аспирантов МГУ [1,4].

В DVM и данные и вычисления по процессорам *виртуальной параллельной машины* распределяет программист и именно на него возлагается полная от-

ветственность за соблюдение (ему известных) правил вычислений. Программист определяет не только общие (вычисляемые на одних, а используемые на других процессорах) данные, но и отмечает точки в последовательной программе, где необходимо обновить (синхронизировать) данные. При этом текст DVM-программы является корректным и для обычных "последовательных" компиляторов (в Fortran'е для задания DVM-директив используются строки комментариев, в С применяется механизм макросов), что позволяет осуществлять сначала отладку последовательной версии программы, а в дальнейшем тестировать DVM-расширения исходного текста и компилировать параллельный вариант программы.

Объявленные в программе переменные (исключая описанные как "распределённые" массивы) копируются по всем процессорам. В момент старта DVM-программы создается единственная её ветвь (*поток управления*), при входе в параллельную конструкцию (напр., параллельный цикл или область параллельных задач) ветвь разбивается на определённое количество параллельных ветвей, каждая из которых исполняется на выделенном процессоре (число процессоров и топология многопроцессорной системы задаётся в командной строке программы); при выходе из параллельной конструкции все ветви вновь сливаются в первоначальную ветвь.

Для распределения массивов в системе DVM используется директива DISTRIBUTE, являющаяся дополнением описательной (неисполняемой) части программы. Например, строка CDVM$ DISTRIBUTE mass_1 (BLOCK) описывает распределение ранее описанного одномерного Fortran-массива mass_1 на решетку процессоров равными блоками (WGT_BLOCK(wb, nwb) задаёт распределение неравными блоками, * - отображение целым измерением), *выравнивание массивов* (расположение нескольких массивов согласованно друг с другом – например, на одном и том же процессоре) достигается применением директивы ALIGN. Директивы REDISTRIBUTE и REALIGN динамически изменяют распределение данных. Параллельное выполнение циклов задается директивой CDVM$ PARALLEL с параметрами, директива MAP определяет соответствие выполняемых задач секции процессоров и т.д. (пример Fortran'DVM-программы приведен в Приложении 1в). Таким образом DVM реализует *модель параллелизма по данным и по управлению*.

В целом DVM-система высокопереносима, достаточно гибка в описании возможностей распараллеливания и, вместе с тем, позволяет сократить время разработки и отладки программ.

В Институте Системного Программирования РАН разработан специальный язык mpC высокого уровня для программирования неоднородных сетей (http://www.ispas.ru/~mpc). Язык mpC использует нотацию С и включает возможности, необходимые для определения всех важных свойств параллельного алгоритма (нужное число параллельных процессов, объём вычислений и передаваемых данных для каждого процесса, сценарий взаимодействия про-

цессов и т.п., причём имеется возможность изменять эти характеристики во время выполнения программы).

В mpC понятие *сети* служит механизмом, позволяющего разработчику абстрагироваться от физических особенностей конкретной вычислительной системы (в простейшем случае сетью является множество *виртуальных процессоров*); после описания сети программист связывает отображение её виртуальных процессоров с реальными процессами параллельной программы. *Автоматическая сеть* определена внутри блока программы, при выходе из этого блока все захваченные под виртуальные процессоры сети освобождаются и сеть может быть переопределена заново; *статическая сеть* существует всё время существования программы. Каждая вновь создаваемая сеть имеет ровно один виртуальный процессор (*родитель* создаваемой сети), общий с уже существующими сетями (через него передаются результаты вычислений при прекращении существования сети). В любой момент выполнения mpC-программы имеется предопределённая сеть host и *виртуальный хост-процессор* (родитель всех сетей). *Размазанная* переменная (массив) определена для всех процессов и её значение одинаково в каждом из них. *Базовые* функции императивно выполняются всеми процессорами, *узловые* функции могут быть вызваны отдельным процессом или группой процессоров, *сетевые* аналогичны базовым, но выполняются на конкретных сетях. Обмен данными между процессами выполняется с помощью механизма *подсетей* – подмножества процессоров, объединяющих две (или несколько) сетей.

Важной особенность mpC состоит в наличии средств для динамического распределения объёмов вычислений по виртуальным процессорам (что необходимо для балансировки вычислений по узлам неоднородных сетей); для этого используется встроенные языковые возможности, реализуемые оператором recon.

В целом mpC представляет программисту огромные возможности разработки параллельных программ, однако требует высокого профессионализма как в логическом программировании так и в практическом использовании языка.

Пропагандируется также система ОРС (Открытая Распараллеливающая Система, *Open Parallelizing System Group*, http://ops.rsu.ru/about.shtml) - программная инструментальная система, ориентированная на разработку распараллеливающих компиляторов, оптимизирующих компиляторов с параллельных языков, систем полуавтоматического распараллеливания.

4.2 Системы параллельного программирования на основе обмена сообщениями

Вышеописанные системы разработки параллельных программ, несмотря на требования явного описания распределения вычислений и данных по процессорам, все же возлагают на "параллельный" компилятор работу по реализации этих описаний. В некоторых случаях бывает полезным иметь возможность самостоятельного максимально полного управления процессом обработки информации (включая распределение данных и вычислений и обмен сообщениями между ветвями программы).

Модель программирования MPI (*Message Passing Interface,* 1994, http://mpiforum.org) основана на передаче сообщений. *Сообщение* состоит из блока (блоков) передаваемых данных и дополнительной информации (тип передаваемых данных, идентификатор сообщения, номер процесса-получателя и др.).

MPI иногда называют "ассемблерным уровнем" в параллельном программировании, основывая это необходимостью максимально подробного описания всех необходимых операций обменов данными между частями программы, выполняющимися на отдельных процессорах; при этом распределение данных и вычислений полностью возлагается на программиста и выполняется средствами базового языка программирования (что очень непросто, поэтому и применяются синтетические подходы – например, вышеупомянутый OpenMP+MPI). Возможности MPI реализованы как набор (размещённых в соответствующей библиотеке) MPI-функций (существует интерфейс с Fortran и C/C++); появившийся в 1997 г. проект стандарта MPI-2 существенно расширяет возможности MPI (напр., динамическое порождение и уничтожение процессов; при этом для MPI-1 диаграмма процессов соответствует рис.3б, а для MPI-2 – рис.3в). В настоящее время существуют две основные реализации MPI – MPICH (*MPI & Chameleon*, http://www-unix.mcs.anl.gov/mpi/mpich) и LAM (*Local Area Machine*, http://www.lam-mpi.org); имеется вариант для Windows. Существуют сведения, что MPI-2 реализован в системе программирования векторно-параллельной системы Earth Simulator.

Вообще говоря, для написания подавляющего большинства программ достаточно 6-ти функций интерфейса MPI:

MPI_Init	- инициализация MPI-библиотеки
MPI_Comm_size	- определение числа процессов
MPI_Comm_rank	- определение процессом собственного номера
MPI_Send	- посылка сообщения
MPI_Recv	- получение сообщения
MPI_Finalize	- завершение программы MPI

среди которых основными являются функции MPI_Send/MPI_Recv обмена сообщениями типа "точка-точка". Однако для удобства программирования в

MPI включен широкий набор функций - широковещательная передача MPI_Bcast, раздача сообщений от одного процесса всем процессам группы MPI_Scatter, сбор данных от всех процессов в группе в один из процессов MPI_Gather и т.п., функции барьерной синхронизации процессов MPI_Barrier, глобальные операции редукции MPI_Reduce, MPI_Allreduce, MPI_Reduce_Scatter, MPI_Scan (конкретная операция редукции может быть переопределена пользователем) и др.

При программировании на MPI программист обязан контролировать исходный текст на наличие структур, вызывающих (труднодетектируемые на работающей программе) дедлоки (*deadlock* – тупиковая ситуация, зависание); возникновение дедлока при MPI-программировании обычно сопряжено с использованием *блокирующих функций* обмена сообщениями. В данном случае дедлок – ситуация, когда первый процесс не может вернуть управление из функции посылки, поскольку второй не начинает приём сообщения, а второй не может начать прием, ибо сам по той же причине не может выполнить посылку.

Дополнительную гибкость даёт MPI возможность определения *виртуальной топологии* процессоров; при этом (независимо от физической топологии процессорной решетки) может вводиться топология n-мерных кубов (торов) или произвольного графа (MPI_CART_CREATE, MPI_GRAPH_CREATE соответственно и функции поддержки их использования). Виртуальные топологии служат как целям упрощения программирования (например, двумерная решетка удобна для произведения матричных операций), так и способствуют повышению производительности (при корректном их отображении на физическую топологию вычислительных узлов многопроцессорной системы).

В целом создание программ с использованием MPI – удел любителей наиболее тонкого, гибкого (низкоуровневого) программирования, однако (вследствие именно этого) имеются возможности разработки надстроек над MPI, существенно упрощающих работу. MPI применён при разработке большого количества проблемно-ориентированных параллельных библиотек. Простой пример Fortran’MPI-программы приведён в Приложении 1г).

К "ручным" (низкоуровневым) технологиям разработки параллельных программ относится и система PVM (*Parallel Virtual Machine*, http://epm.ornl.gov/pvm/pvm_home.html), предложенная исторически ранее MPI (проект –1989, реализация – 1991 г.). PVM стандартизирует не только *интерфейс программиста* (набор и содержание предоставляемых функций), но и *интерфейс пользователя* (команды пользователя, вводимые с клавиатуры для управления параллельной программы), [4]. Функции PVM предоставляются общедоступной библиотекой, существуют реализации PVM для самых различных платформ.

Виртуальной машиной (ВМ) называют совокупность узлов, на которых исполняется параллельная программа; функционирование ВМ достигается

функционированием на каждом узле процесса - *демона PVM*. *Консоль PVM* – специальная программа, позволяющая управлять виртуальной машиной. Имеется возможность программным путем изменять (активизируя и "выключая" узлы) состав ВМ, стартовать и включать в состав ветвей параллельных программ процессы (т.е. динамически порождать ветви параллельной программы) или отсоединять их от ВМ. Любой PVM-процесс может информировать другой ждущий от него сообщений процесс сигналом о возникновении ошибки, этот сигнал является основой механизма избежания зависания или аварийного завершения. Программная реализация посылки сообщений проста, по умолчанию сообщения буферизуемы; вероятность дедлока минимальна. Как и в MPI, кроме обменов "точка-точка" имеются широковещательные и сообщения редукции.

PVM изначально проектировалась разработчиками как система для задач с крупным зерном параллелизма (требования к эффективности коммуникаций не столь высоки – в период разработки PVM применялись 10 Mbit Ethernet-сети); популярность PVM до сих пор высока (основные производители суперкомпьютеров снабжают свои изделия и MPI и PVM).

Системой параллельного программирования на основе передачи сообщений является система Linda (http://cs.yale.edu), разработанная в середине 80-х г.г. в США. Идея Linda основана на простых положениях:

а) параллельная программа представлена множеством процессов, каждый из которых выполняется подобно обычной последовательной программе,
б) все процессы имеют доступ к общей памяти, причём единицей хранения данных является *кортеж* (в этом смысле общая память суть *пространство кортежей*),
в) всем процессам доступны следующие операции с пространством кортежей: *поместить кортеж* (можно поместить имеющийся кортеж, после чего в пространстве кортежей окажется два кортежа с одинаковым именем), *забрать* (после этого кортеж становится недоступным остальным процессам) и *скопировать*,
г) процессы не взаимодействуют друг с другом явно, а только через пространство кортежей.

Т.о. в отличие от MPI (где допустим прямой обмен данными любого процесса с любым), в Linda обмены осуществляются фактически через некий "карман" ("пространство кортежей", причём вследствие возможности при поиске кортежа использовать метод совпадения значения отдельных его полей "пространство кортежей" фактически является *виртуальной ассоциативной памятью*).

Разработчики Linda доказывают, что любой последовательный язык программирования для превращения его в средство параллельного программиро-

вания достаточно добавить лишь четыре новые функции (три функции для операций над пространством кортежей и одна для порождения параллельных процессов) [1]. Функция out(…) помещает кортеж в пространство кортежей, in(…) ищет в пространстве кортежей нужный (используется маска), read(…) аналогична in без удаления кортежа (полезно использовать для параллельного доступа к переменным из нескольких процессов), функция eval(…) порождает отдельный параллельный процесс для вычисления значений функций, перечисленных в списке формальных параметров вызова функции, вычисленное значение помещается в пространство кортежей аналогично вызову out.

Если в пространстве кортежей ни один кортеж не соответствует заданному в in или read образцу, процесс блокируется до момента появления соответствующего кортежа. Параллельная программа в системе Linda завершается, когда все порождённые процесс завершились или все заблокированы функциями in и read.

Указанные языковые средства Linda на самом деле допускают широкие возможности по синхронизации процессов, реализации работающих по схеме "главный/подчиненный" программ и т.д. Linda эффективна для многопроцессорных вычислительных систем, обладающих общей памятью (только в при размещении пространства кортежей в памяти такого типа возможно реализовать быстроту операций с кортежами); при реализации Linda для систем с распределённой памятью возникают проблемы, аналогичные NUMA-системам (VSM - *Virtual Share Memory* - виртуальная разделяемая память в системе Linda реализуется программно).

В настоящее время Linda коммерчески поддерживается образованной в 1980 г. фирмой SCA (Scientific Computing Associates, Inc.), торговая марка TCP Linda (http://www.lindaspaces.com, lunix-support@LindaSpaces.com) включают графическую отладочную среду Tuplescope. На основе VSM фирмой SCA разработаны системы Piranha (средство объединения персональных ЭВМ в многопроцессорную вычислительную систему), JParadise (система поддержки распределенных прикладных Java-программ), SCA предлагает также Windows-вариант системы программирования Linda и (совместно с OptTek Systems, Inc., http://www.opttek.com) параллельную версию библиотеки OptQuest Callable программных пакетов многомерной оптимизации.

Linda использована при разработке системы Gaussian'03 (изучение электронной структуры химических веществ), применена при разработке системы Paradise для параллелизации процесса обнаружения знаний в базах данных (технология *Data Mining*), используется фирмой Motorola, Inc. при проектировании микросхем, для поддержки финансовой деятельности, обработки данных геофизических исследований при морской нефтедобыче и др.

4.3 Автоматизация распараллеливания алгоритмов

4.3.1 Распараллеливающие компиляторы

Желание автоматизировать процесс перевода в параллельный вариант огромного количества наработанного (и вновь разрабатываемого) программного обеспечения является более чем естественным.

Разработка *полностью автоматически распараллеливающего компилятора*, казалось, может снять все проблемы программиста. К сожалению, реальные разработки в этом направлении не впечатляют своей эффективностью. Во-первых, априори невозможно создать такой компилятор универсальным (вследствие наличия значительного количества архитектур ЭВМ и систем команд процессоров, при этом оптимизированный для одной системы компилятор окажется малоэффективным для другой). Во-вторых, (на основе опыта уже разработанных систем параллельного программирования) представляется проблематичной сама идея *полностью автоматического* распараллеливания (подобная система должна обладать существенными чертами искусственного интеллекта и, скорее всего, будет громоздкой и трудно настраиваемой). Весьма проблематична (хотя на сегодняшний день уже скорее чисто *технически*) разработка единой схемы распараллеливания произвольного алгоритма с отображением на архитектуру реальной ЭВМ (именно здесь как нельзя более полезны приёмы классификации и описания архитектур).

Именно поэтому обычно ограничиваются задачей синтаксического анализа исходно-параллельного текста с целью выявления потенциального параллелизма и генерацией распараллеленного исходного текста (часто в стиле MPI), компиляция этой программы осуществляется машинно-зависимым компилятором с поддержкой библиотеки MPI (частично такие системы описаны в подразделе 3.2).

Например, в состав полнофункционального пакета Forge90 (APR - *Applied Parallel Research, Inc.*, http://www.infomall.org/apri//index.html, forge@netcom.com) входят препроцессоры dpf, spf, xhpf (для систем с распределённой памятью, общей памятью и для использования программирования в HPF-стиле соответственно), позволяющие на основе исходного текста последовательного алгоритма сформировать текст, уже распараллеленный в технологии PVM на Fortran’77; подобные пакеты объёмны и дорогостоящи.

4.3.2 Автоматическое распараллеливание с помощью Т-системы

Разработка технологии автоматического динамического распараллеливания программ (названная "Т-система") ведётся с конца 80-х г.г. в Институте программных средств РАН (г. Переславль-Залесский). В целом стиль Т-программирования близок традиционным языкам программирования (Fortran,

С/С++), причм явное указание на параллельность выполнения блоков программы в тексте отсутствует.

Базовые принципы Т-системы основаны на результатах общей теории *функционального программирования* (ФП). Теоретические основы *императивного (операторного) программирования* были заложены в 1930-х г.г. (*Алан Тьюринг, Джон фон Нейман*), положенная в основу функционального подхода теория родилась в 1920÷1930-х г.г. (комбинаторная логика - *Мозес Шенфинкель, Хаскелл Карри*, лямбда-исчисление - *Алонзо Чёрч, 1936*).

Математические функции выражают связь между параметрами (входом) и результатом (выходом) некоторого процесса. Вычисление – также процесс, имеющий вход и выход, при этом функция является вполне подходящим и адекватным средством описания вычислений - именно этот простой принцип положен в основу функциональной парадигмы и функционального стиля программирования. Функциональная программа представляет собой набор определений функций. Функции определяются через другие функции или рекурсивно через самих себя. В процессе выполнения программы функции получают параметры, вычисляют и возвращают результат, в случае необходимости вычисляя значения других функций. Программируя на функциональном языке, программист не должен описывать порядок вычислений - ему необходимо лишь описать желаемый результат в виде системы функций.

Функции в ФП не имеют *побочных эффектов*. Обращение к функции не вызывает иного эффекта, кроме вычисления результата (в этом смысле привычные вызовы традиционных языков программирования суть *процедуры,* а не *функции* в понятии ФП); это устраняет серьёзный источник ошибок и делает *порядок выполнения функций несущественным* (так как побочные эффекты не могут изменять значение выражения, оно *может быть вычислено в любое время*).

Теория функционального программирования долгое время оставалась "гадким утенком" (в противоположность идеям императивного программирования) до момента разработки первого *почти* функционального языка программирования Lisp (*Джон МакКарти,* 1950). К началу 1980-х г.г. появились ФП-языки ML, Scheme, Hope, Miranda, Clean и др.; одной из популярных в настоящее время реализаций ФП-подхода является язык Haskell (*Хаскелл Карри*, начало 1990-х г.г., http://www.haskell.org).

Важной особенностью Haskell является механизм "*ленивых (отложенных) вычислений"*. В традиционных языках программирования (например, Fortran, С/С++) вызов функции приводит к вычислению всех аргументов ("*вызов-по-значению"*); при этом если какой-либо аргумент не использовался в функции, то результат вычислений пропадает, т.е. вычисления были произведены впустую. Альтернативой *вызова-по-значению* является *вызов-по-необходимости* (этом случае аргумент вычисляется, только если он действительно необходим для вычисления результата); понятный программисту пример – выполнение

оператора конъюнкции && из C/C++, который никогда не вычисляет значение второго аргумента, если первый аргумент есть "ложь". Языки, использующие отложенные вычисления, называются *нестрогими* (примеры – Haskell, Gofer, Miranda); если функциональный язык не поддерживает отложенные вычисления, он является *строгим* (языки Scheme, Standard ML, Caml, в таких языках *порядок вычисления строго определён*).

Одно из главенствующих понятий Т-системы – "*чистые" функции* (функции без *побочных эффектов* при вычислениях); чистая функция может быть выполнена параллельно с основной программой. Чистыми (отсутствуют *побочные эффекты*) обычно являются *нестрогие* языки.

В каждый момент времени выделяются готовые к вычислению "подвыражения" и распределяются по имеющимся процессорам, при этом за основу берётся граф, узлы которого представляют выбранные функции, а дуги соответствуют отношению "выражение-подвыражение".

Для использования возможностей функционального программирования в традиционные языки вводится понятие *неготового значения*. В применении к С это выражается введением дополнительного атрибута tval описания переменных, tval int j определяет переменную, значение которой может быть целым числом или неготовым (пока не посчитанным) значением, в описании функции без побочных эффектов необходим атрибут tfun, выходного значения – tout, глобальной ссылки на неготовую переменную - tptr (измененный таким образом С++ получил название Т++, см. Приложение 3).

Т-система запускает чистые функции как ветви параллельной программы, а *неготовые переменные* (НП) являются основным средством синхронизации их выполнения. При чтении НП происходит блокировка процесса вычисления, выполнившего такое обращение, причём ожидание продлится до тех пор, пока переменная не получит готового значения (исключение – операция присваивания НП, при этом блокировка не возникает). При записи обычного значения в НП она становится готовой для всех выражений, в которые она входит; ранее заблокированные на ней процессы выходят из блокировки.

Т-система снабжена развитыми системами отладки – от начальной прогонки программы на последовательной машине (t-атрибуты в этом режиме игнорируются) до использования спецметодов (трассировка, профилировка и др.) для оптимизации при выполнении на многопроцессорной ЭВМ.

Преимущества Т-системы заключается в практически полном освобождении программиста от обдумывания деталей описания распараллеливания, недостатки – в требовании *описания алгоритма в функциональном стиле* (формально необходимо составить программу в виде набора чистых Т-функций, что непривычно). Вычислительная сложность Т-функций (она задаёт размер гранулы параллелизма) также должна определяться программистом (излишне малая приведет к чрезмерным накладным расходам по обмену данными ме-

жду процессорами, слишком большая – к неравномерной загрузке вычислительных узлов).

4.3.3 Непроцедурный декларативный язык НОРМА

Свободно распространяемым инструментом автоматизации создания параллельных MPI-программ является система НОРМА (http://www.keldysh.ru/pages/norma), позволяющая (на основе специализированного декларативного языка) синтезировать исходный Fortran/C - текст с вызовами MPI [9]. Непроцедурный язык НОРМА является проблемно-ориентированным и предназначен для *автоматизации решения сеточных задач* на вычислительных системах с параллельной архитектурой. Этот язык позволяет исключить фазу параллельного программирования, которая необходима при переходе от расчётных формул, заданных прикладным специалистом, к программе.

Концепция непроцедурного языка НОРМА (*Непроцедурное Описание Разностных Моделей Алгоритмов* или *НОРМАльный уровень общения прикладного математика с компьютером*, http://www.keldysh.ru/pages/norma) предложенная еще в 60-х г.г. в ИПМ им.М.В.Келдыша РАН.

Язык НОРМА называют *декларативным* вследствие упора именно на описание правил вычисления значений, а не на исчерпывающе подробную конкретизацию алгоритма. Разработчик прикладных программ абстрагируется от особенностей конкретных ЭВМ и мыслит в привычных терминах своей предметной области (сеточные методы математической физики, в основном метод конечных разностей - МКР). Система НОРМА включает синтезатор, назначением которого является преобразование НОРМА-текста в (один из привычных) языковых стандартов параллельного или последовательного программирования (существует возможность получения Fortran’MPI, Fortran’PVM, Fortran’77 или соответствующих С-текстов). Именно синтезатор учитывает архитектуру конкретной ЭВМ, в самом языке НОРМА не предусмотрено никаких указаний на параллелизм или описания структуры целевого компьютера (см. Приложение 4).

Эффективный автоматический синтез параллельного кода на основе НОРМА-программы достигается определёнными ограничениями языка, важное из которых – отсутствие многократного присваивания (при этом несущественна последовательность операторов, отсутствуют глобальные переменные, запрещена рекурсия, нет побочных эффектов при вычислениях и др., [9]). Исходный текст на НОРМА в высшей степени близок к записи численного метода решения конкретной задачи. В записи на языке НОРМА отсутствуют избыточные информационные связи (полный анализ которых как раз и является "ахиллесовой пятой" систем выявления скрытого параллелизма), что и позволяет реализовать эффективное автоматическое распараллеливание. За-

метим явное сходство парадигмы системы НОРМА с (использованной при разработке Т-системы) идеей функционального программирования.

Программа на НОРМА содержит не менее одного *раздела*. Возможны разделы трех видов: *главный* (ключевые слова MAIN PART), *простой* (PART) и *раздел-функция* (FUNCTION), разделы могут вызывать друг друга по имени (рекурсия запрещена) и обмениваться данными с помощью механизма формальных и фактических параметров или путем использования внешних файлов (описания INPUT и OUTPUT). Все описанные в списке формальных параметров до слова RESULT являются входными данными, далее – выходные (для раздела FUNCTION слово RESULT не используется, т.к. результат вычисления функции связывается с именем и типом последней). Описанные в разделе переменные являются локальными, глобальных переменных не существует.

В теле раздела могут быть заданы (в произвольном порядке) *описания*, *операторы* и *итерации*. Важным понятием НОРМА является одномерная *область*, путем *операции произведения областей* строится многомерная область (с каждой размерностью связывается *имя индекса*). Возможна *операция модификации областей* (добавление/удаление точек, изменение диапазона). *Условная область* представляет собой подобласть, определяемую выполнением некоторого условия над значениями точек области. Мощность языка НОРМА выражена *коллективными операциями* над узлами областей (поддерживается линейная топология и двумерная решетка).

В НОРМА определены имеющие привычные типы REAL, INTEGER, DOUBLE и идентифицируемые именами *скалярные величины* и *величины на области* (последние связаны с конкретной областью и индексированы).

Скалярные операторы вычисляют арифметические значения скаляров, оператор ASSUME вычисляет арифметические значения величин на областях (вычисляются все значения по индексам области). Для выполнения итераций имеется специальная конструкция ITERATION. В НОРМА определены стандартные арифметические функции, *функции редукции* и *внешние функции пользователя* (могут быть вызовами функций на Fortran'е). Заключенные в ограничители # ... # операторы выполняются строго в порядке их следования в программе. Более подробно описание языка см. на сайтах https://parallel.ru/tech/norma и http://www.keldysh.ru/norma.

Однако для создания НОРМА-программы всё же требуется программист, знакомый с правилами языка и формально "набивающий" (после этапа обдумывания) исходный текст программы. Особенности НОРМА позволяют сделать следующий шаг в сторону, противоположную операции разработки программы в виде последовательной записи операторов, т.е. вообще не применять текстовое представление программы. Логично снабдить синтезатор НОРМА интерактивной графической подсистемой, позволяющей манипулировать с программой как с отображением в виде графики и гипертекста сово-

купности объектов (включая формулы, вводимые пользователем в отдельные поля гипертекстовых форм).

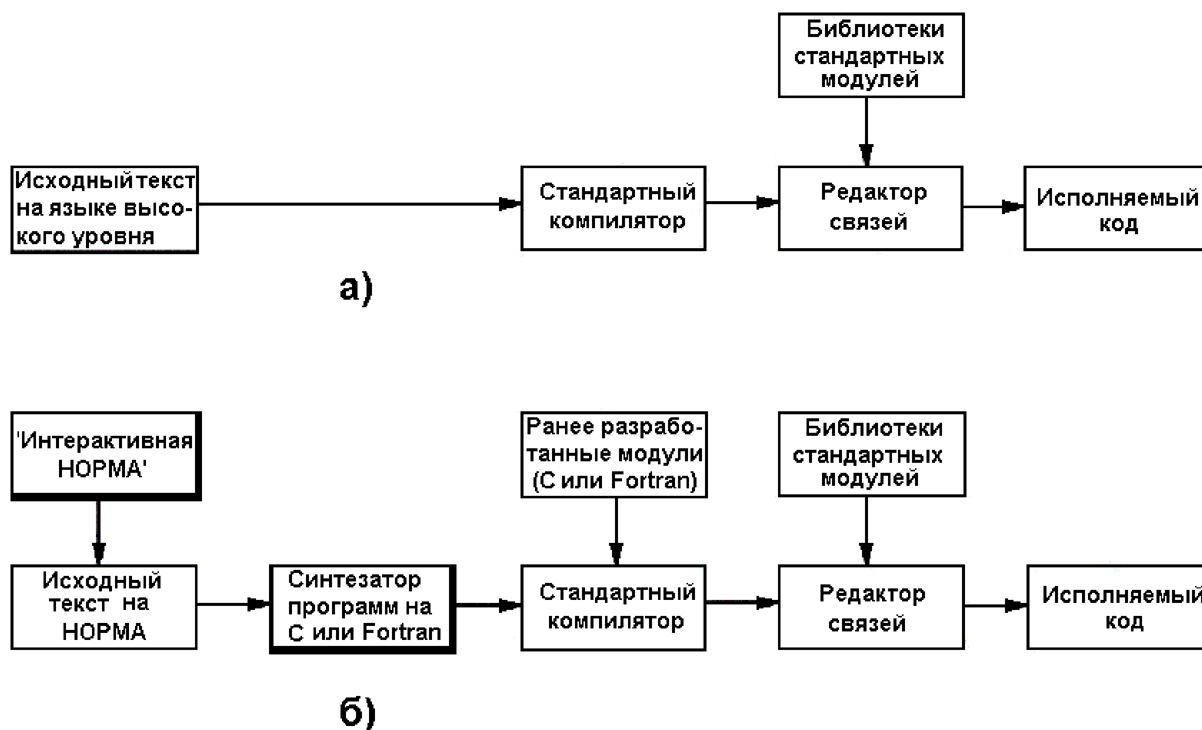


Рисунок 26 — Этапы создания исполняемых программ: а) - классический подход, б) - использование НОРМА-программирования совместно с оболочкой "Интерактивная НОРМА"

С этой целью на кафедре ИТ-4 МГАПИ разрабатывается система "Интерактивная НОРМА" (http://norma.deniz.ru), позволяющая создавать параллельные программы практически без написания исходных текстов на языке программирования (рис.26б).

4.4 Стандартные предметно-ориентированные библиотеки параллельных вычислений

При всей широте средств разработки параллельных программ создавать такие программы (и особенно эффективные!) всё же очень и очень непросто. Одним из путей разработки эффективных параллельных программ является использование *предметно-ориентированных библиотек*.

В этом случае оптимизация вообще скрыта от проблемного программиста. Чем больший объём работы внутри программы отводится подпрограммам такой библиотеки, тем большим будет итоговый выигрыш в скорости её работы. Собственно же программа пишется на обычном языке программирования безо всяких упоминаний о MPI и строится стандартным компилятором; от программиста требуется лишь указать при компоновке имя соответствующего библиотечного файла и запускать полученный в итоге исполняемый

файл как MPI-программу. Т.о. чем большую часть вычислений удастся переложить на библиотечные подпрограммы (а они особенно тщательно протестированы, отлажены и оптимизированы), тем более эффективной получится результирующая программа.

Наиболее известны библиотеки программ для решения задач матфизики (матричные операции, численное решение систем дифуравнений и т.п.).

Напр., разработанный в Sandia Laboratories пакет AZTEC (http://www.cs.sandia.gov/CRF/aztec1.html) специализирован на решение с использованием многопроцессорных систем линейных алгебраических уравнение (СЛАУ) с разреженными матрицами. AZTEC включает в себя процедуры, реализующие итерационные методы Крылова:

- метод сопряжённых градиентов (CG),
- обобщенный метод минимальных невязок (GMRES),
- квадратичный метод сопряженных градиентов (CGS),
- метод квазиминимальных невязок (TFQMR),
- метод бисопряжённого градиента (BiCGSTAB) со стабилизацией.

При решении СЛАУ используются как прямой метод LU, так и неполное LU-разложение в подобластях. Матрица системы может быть как общего вида, так и специального, возникающего при конечно-разностной аппроксимации дифференциальных уравнений в частных производных (PDE - *Partial Differential Equations*). Руководство по использованию AZTEC может быть получено с адреса http://rsusu1.rnd.runnet.ru/ncube/aztec/index.html.

Библиотека ScaLAPACK (Scalable LAPACK, http://www.netlib.org/ scalapack/index.html) ориентирована на решение задач линейной алгебры с матрицами общего вида. ScaLAPAC является расширением известной библиотеки LAPACK (как считается, наиболее полно отвечающей архитектуре современных процессоров – напр., библиотечные подпрограммы оптимизированы для эффективного использования кэш-памяти и т.п.) на многопроцессорную архитектуру.

Пакет ScaLAPACK фактически является стандартом в программном обеспечении многопроцессорных систем. В нем почти полностью сохранены состав и структура пакета LAPACK и практически не изменились обращения к подпрограммам верхнего уровня. В основе успеха реализации этого проекта лежали два принципиально важных решения:

- В пакете LAPACK все элементарные векторные и матричные операции выполняются с помощью высокооптимизированных подпрограмм библиотеки BLAS (*Basic Linear Algebra Subprograms*). По аналогии с этим при реализации ScaLAPACK была разработана параллельная версия этой

библиотеки (названная PBLAS), что избавило от необходимости радикальным образом переписывать подпрограммы верхнего уровня.

- Все коммуникационные операции выполняются с использованием подпрограмм из специально разработанной библиотеки BLACS (*Basic Linear Algebra Communication Subprograms*), поэтому перенос пакета на различные многопроцессорные платформы требует настройки только этой библиотеки.

Обозначенные как "глобальные" компоненты пакета выполняются параллельно на заданном наборе процессоров и в качестве аргументов используют векторы и матрицы, распределённые по этим процессорам, подпрограммы из означенных как "локальные" компонентов пакета вызываются на одном процессоре и работают с локальными данными (рис.27). Имеется возможность работы с вещественными и комплексными данными одинарной или двойной точности, матрицами различного вида, выполнение факторизации, вычисление собственных значений и собственных векторов, определение сингулярных значений и др.; руководство пользователя доступно на адресе http://rsusu1.rnd.runnet.ru/ncube/scalapack/scalapack_home.html.

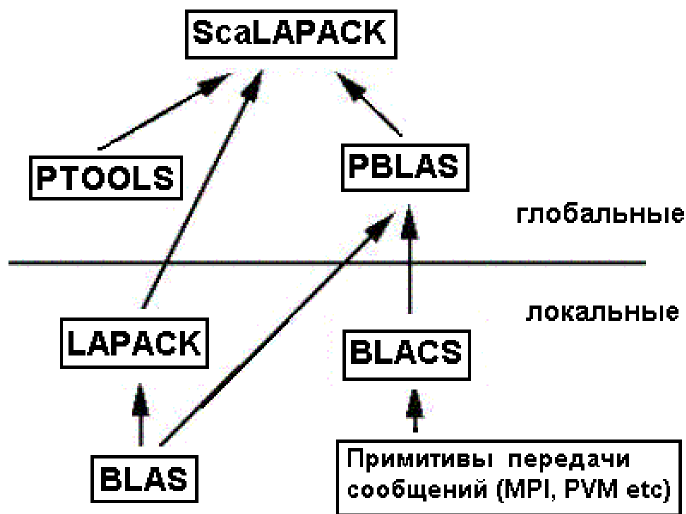


Рисунок 27 — Структура пакета ScaLAPACK

Из иных библиотечных пакетов следует указать Block Solve (решение СЛАУ с редкозаполненной матрицей системы, Argonne National Laboratory, http://www.mcs.anl.gov, http://www-unix.mcs.anl.gov/sumaa3d/BlockSolver), разработанную в математическом отделении университета Бата (Англия) библиотеку DOUG (*Domain On Unstructured Grids*), позиционируемую как "параллельный решатель для систем, возникающих при решении методом конечных элементов" (http://www.math.bath.ac.uk/~mjh/doug) и некоторые другие.

Практически все предметно-ориентированные библиотеки для решения задач матфизики бесплатны и доступны в исходных кодах.

4.5 Параллелизм в системах управления базами данных

Управление большими базами данных – естественное применение многомашинных комплексов и многопроцессорных вычислительных систем. В списке Top500 25-й редакции (июнь 2005) приведены данные 16 инсталляций МВС для применений в области финансов, максимальной LINPACK-производительностью в 4713 GFlops обладает находящаяся на 62 месте система eServer BlueGene Solution (2048 процессоров фирмы IBM, поставлена для NIWS Co. Ltd, Япония в 2005 г.). В СберБанке РФ с 2003 г. эксплуатируется 256-процессорная SMP-система HP SuperDome с процессорами PA-RISK 750 MHz производительностью 440 Gflops.

Параллельная система баз данных - это система управления базами данных (СУБД), реализованная на многопроцессорной системе с высокой степенью связности. Под многопроцессорной системой с высокой степенью связности понимается система, включающую в себя много процессоров и много дисков, в которой процессоры объединены между собой с помощью некоторой соединительной сети, причем *время обмена данными по сети сравнимо со временем обмена данными с диском*. Приведённое определение исключает из рассмотрения распределённые СУБД, реализуемые на нескольких независимых компьютерах, объединённых локальной и/или глобальной сетями (*).

Основная нагрузка в параллельной системе баз данных приходится на выполнение запросов к БД. Обычно рассматривают несколько путей распараллеливания запросов: (**)

- Горизонтальный параллелизм возникает при распределении хранящейся в БД информации по нескольким физическим устройствам (жёстким дискам). При этом информация из одного соотношения разбивается на части по горизонтали (достигается *сегментация данных*), распараллеливание заключается в выполнении *одинаковых операций над разными физически хранимыми данными*. Эти операции независимы и могут выполняться различными процессорами, конечный результат складывается из результатов выполнения отдельных операций (рис.28а).
- Вертикальный параллелизм реализуется конвейерным способом выполнения составляющих запрос пользователя операций. При этом ядро СУБД должно произвести декомпозицию запроса с целью выявления параллельно выполняющихся шагов запроса (рис.28б).

---

* Л.Б.Соколинский. Обзор архитектур параллельных систем баз данных. / ПРОГРАММИРОВАНИЕ, 2004, № 6, с. 49 ÷ 63.

** Т.С.Крюкова. Базы данных: модели, разработка, реализация. –СПб.: Питер, 2001. –304 с.

• <u>Гибридный параллелизм</u> предполагает совместное использование двух первых подходов (рис.28в).

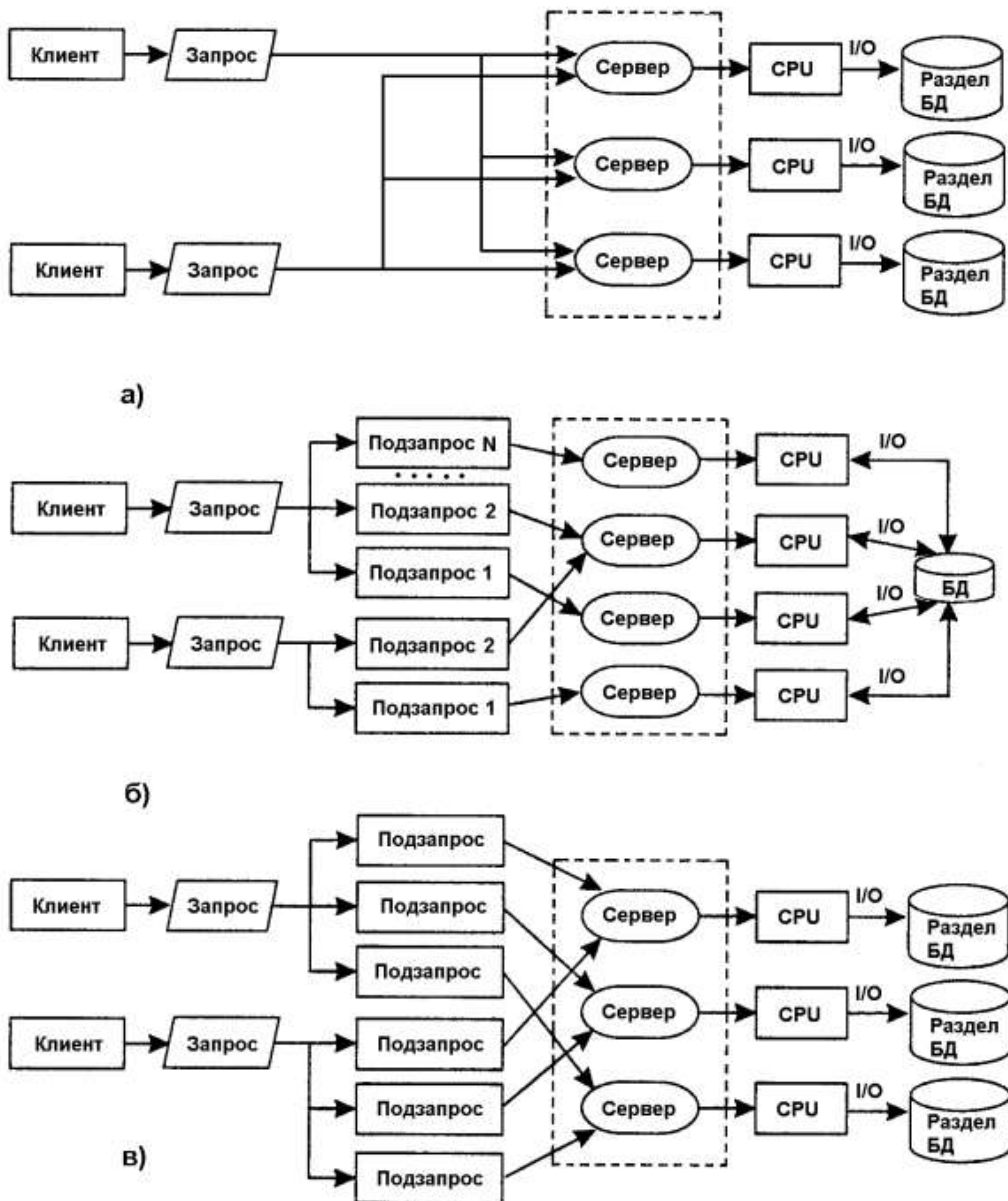


Рисунок 28 — Укрупнённая схема выполнение запроса при а) – горизонтальном, б) - вертикальном и в) - гибридном параллелизме.

Классификация архитектур параллельных машин баз данных разработана в основном Стоунбрейкером (*M. Stonebraker*, 1986). Далее классификация была расширена Ерхардом Рамом (*E. Rahm,* 1992) в сторону *гибридных архитектур* и Копеландом (*Copeland*, 1989) и Келлером (*Keller*) в направление *иерархических архитектур*.

Автор работы (*) развивает подход Копеланда и Келлера, вводя и используя малоизученную до сих пор альтернативную трехуровневую иерархическую архитектуру, в основе которой лежит понятие *Омега-кластера.* Эта архитектура (рис.29, ср. с рис.8в) гает наличие небольшого личества процессорных модулей PM и обслуживающего их модуля дисковой подсистемы DSM (последняя включает сопряжённый с SCSI-шиной коммуникационный процессор, что дает возможность подключения до 7 дисковых накопителей) и является определённым продолжением и развитием идей проекта SDC (*Super Database Computer,* суперкомпьютер баз данных) токийского университета (**).

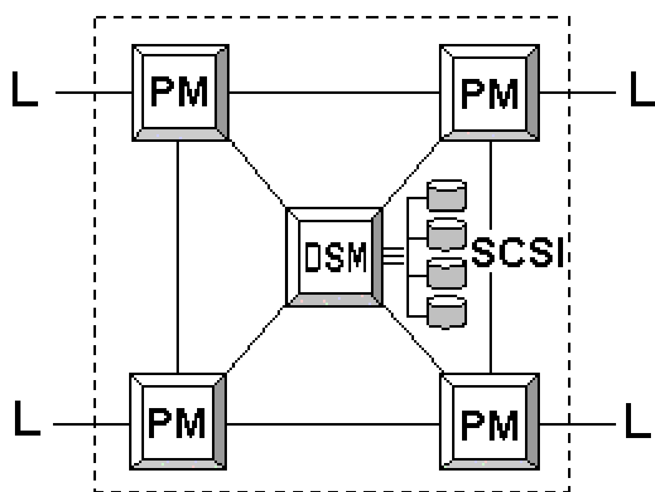


Рисунок 29 — Схема Омега-кластера.

Несколько Омега-кластеров могут объединяться в систему верхнего уровня через свободные линки L процессорных модулей, топология межкластерных соединений может варьироваться от простой линейной до гиперкуба.

Иерархическая архитектура системы Омега предполагает два уровня фрагментации. Каждое отношение может разделяться на фрагменты, размещаемые в различных Омега-кластерах (*межкластерная фрагментация*). В свою очередь, каждый такой фрагмент разделяется на более мелкие фрагменты, распределяемые по различным узлам Омега-кластера (*внутрикластерная фрагментация*). Этот подход делает процесс балансировки загрузки более гибким, поскольку он может выполняться на двух уровнях - локальном, среди процессорных модулей внутри Омега-кластера, и глобальном, среди Омега-кластеров. Описанная архитектура была реализована в прототипе параллельной СУБД Омега на базе МВС-100 в 8-процессорной конфигурации.

---

* Л.Б.Соколинский. Проектирование и анализ архитектур параллельных машин баз данных с высокой отказоустойчивостью. / Челябинский государственный университет.

** Hirano M.S. et al. Architecture of SDC, the super database computer. // In Proceedings of JSPP'90. 1990.

Предложенная в работе (*) классификация форм параллелизма обработки запросов в параллельных БД приведена на рис.30, несколько подробнее о возможных реализациях *транзакций* в распределенных БД реляционного типа см. в работе [7].

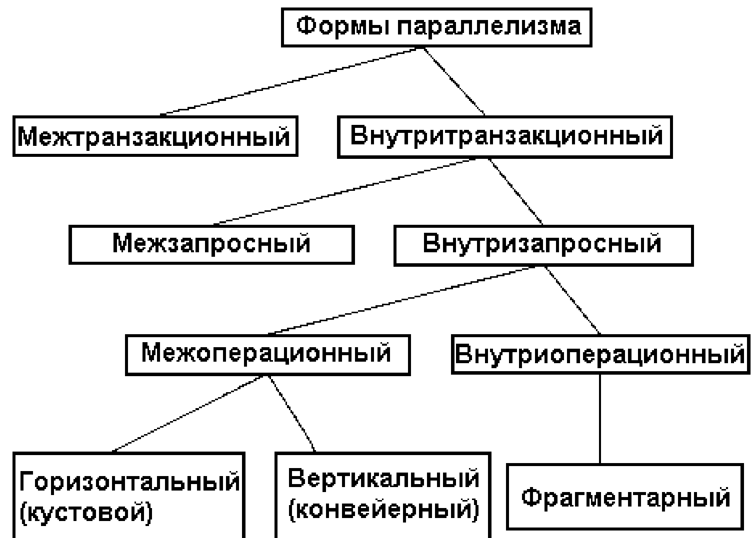


Рисунок 30 — Классификация форм параллелизма при обработке транзакций в параллельных базах данных.

Контрольные вопросы

1. В чём заключается проблема переносимости параллельных программ?
2. В чём суть функционального программирования и отличие его от операторного? В чем заключаются недостатки первого и второго?
3. Что такое "*процесс, управляемый данными*"? В чём отличие его от традиционного подхода к управлению вычислениями? Есть ли недостатки у процесса управления данными?
4. В чём выражается преимущество подхода "Data Flow" и какими средствами оно может быть достигнуто?
5. Каковы основные принципы построения языков (систем) программирования параллельных вычислений?
6. Какие принципы синхронизации независимо выполняющихся процессов известны? Для чего вообще необходима синхронизация?
7. Каким образом параллельная обработка данных применяется в технологии баз данных? Какие формы параллелизма при обработке транзакций известны?

---
* Соколинский Л.Б. Методы организации параллельных систем баз данных на вычислительных системах с массовым параллелизмом. Диссертация на соискание учёной степени доктора физико-математических наук: 05.13.18 // Челябинский государственный университет, -Челябинск: 2003, -247 с.

Заключение

Параллельные вычисления представляют собой целый куст смежных вопросов, включающих аппаратную поддержку, анализ структуры алгоритмов с цель выявления параллелизма и алгоритмические языки для программирования параллельных задач. Технологии параллельных вычислений в настоящее время бурно развиваются в связи с требованиями мировой науки и технологий.

Сегодняшняя проблема – явный недостаток аппаратных средств для параллельных вычислений в учебных и научных учреждениях, что не даёт возможности всестороннего освоения соответствующих технологий; часто практикуемое чисто теоретическое изучение предмета приводит скорее к негативным (отчуждение от предмета из-за схоластичности подхода), чем к позитивным следствиям. Только сейчас появляются технологии, позволяющие ввести практику параллельных вычислений в большинство учебных и производственных лабораторий.

Создание эффективных параллельных программ требует намного более серьёзного и углубленного анализа структуры алгоритмов, нежели при традиционно-последовательном программировании, причём некоторые подходы невозможно реализовать без серьезного изменения мышления программистов. Наряду с теоретическим анализом для получения практически зн**а**чимых результатов необходима постоянная практика в создании параллельных программ.

Список использованной литературы

1. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. — СПб.: БХВ-Петербург, 2011. — 608 с.
2. Э.Таненбаум. Архитектура компьютера. — СПб.: Питер, 2011. — 844 с.
3. Лацис А.О. Параллельная обработка данных. — М.: Издательский центр "Академия", 2010. — 336 с.
4. Шпаковский Г.И. Реализация параллельных вычислений: кластеры, много-ядерные процессоры, грид, квантовые компьютеры. — Минск, БГУ, 2010. –155 с.
5. Боресков А.В., Харламов А.А., Марковский Н.Д. и др. Параллельные вычисления на GPU. Архитектура и программная модель CUDA. Учебное пособие. — М.: Изд. Московского государственного университета, 2012. — 336 с.
6. Жмакин А.П. Архитектура ЭВМ. Сер. Учебная литература для ВУЗ'ов. — СПб.: БХВ-Петербург, 2010. — 352 с.
7. Барский А.Б. Параллельные информационные технологии. // Учебное пособие. — М.: Интернет-Университет информационных технологий: БИНОМ. Лаборатория знаний, 2011. — 503 с.
8. Баканов В.М. Параллельные вычисления (учебно-методическое пособие по выполнению лабораторных работ). — М.: МГУПИ, 2010. — 53 с.

Приложение А. Примеры последовательной и параллельных реализаций алгоритма Якоби решения сеточных уравнений (по материалам работы [6])

а) последовательная программа на языке Fortran'77

```
PROGRAM JAC_F77
PARAMETER (L=8, ITMAX=20)
REAL A(L,L), B(L,L)
PRINT *, '********** TEST_JACOBI **********'
 DO IT = 1, ITMAX
  DO J = 2, L-1
   DO I = 2, L-1
    A(I, J) = B(I, J)
   ENDDO
  ENDDO
  DO J = 2, L-1
    DO I = 2, L-1
      B(I, J) = 0, 25×(A(I-1, J) + A(I, J-1) + A(I+1, J) + A(I, J+1))
    ENDDO
   ENDDO
 ENDDO
END
```

б) параллельная программа на языке HPF,
цветом (насыщенностью) выделены HPF-предписания

```
   PROGRAM JAC_HPF
   PARAMETER (L=8, ITMAX=20)
   REAL A(L,L), B(L,L)
!HPF$ PROCESSORS P(3,3)
!HPF$ DISTRIBUTE ( BLOCK, BLOCK) :: A
!HPF$ ALIGN B(I,J) WITH A(I,J)
C arrays A and B with block distribution
   PRINT *, '********** TEST_JACOBI **********'
   DO IT = 1, ITMAX
!HPF$ INDEPENDENT
   DO J = 2, L-1
!HPF$ INDEPENDENT
   DO I = 2, L-1
     A(I, J) = B(I, J)
   ENDDO
  ENDDO
!HPF$ INDEPENDENT
   DO J = 2, L-1
!HPF$ INDEPENDENT
    DO I = 2, L-1
      B(I, J) = (A(I-1, J) + A(I, J-1) + A(I+1, J) + A(I, J+1)) / 4
    ENDDO
   ENDDO
  ENDDO
 END
```

в) параллельная программа на языке Fortran'DVM,
цветом (насыщенностью) выделены DVM-директивы

```
   PROGRAM JAC_DVM
   PARAMETER (L=8, ITMAX=20)
```

```
  REAL A(L,L), B(L,L)
CDVM$ DISTRIBUTE ( BLOCK, BLOCK) :: A
CDVM$ ALIGN B(I,J) WITH A(I,J)
C arrays A and B with block distribution
  PRINT *, '********** TEST_JACOBI **********'
  DO IT = 1, ITMAX
CDVM$ PARALLEL (J, I) ON A(I, J)
  DO J = 2, L-1
   DO I = 2, L-1
    A(I, J) = B(I, J)
  ENDDO
 ENDDO
CDVM$ PARALLEL (J, I) ON B(I, J), SHADOW_RENEW (A)
C copying shadow elements of A-array from
C neighboring processors before loop execution
  DO J = 2, L-1
    DO I = 2, L-1
     B(I, J) = (A( I-1, J ) + A( I, J-1 ) + A( I+1, J ) + A( I, J+1 )) / 4
   ENDDO
  ENDDO
 ENDDO
END
```

г) параллельная программа на языке Fortran с использованием MPI, цветом (насыщенностью) выделены относящиеся к MPI структуры

```
  PROGRAM JAC_MPI
  include 'mpif.h'
  integer me, nprocs
  PARAMETER (L=8, ITMAX=20, LC=2, LR=2)
  REAL A(0:L/LR+1, 0:L/LC+1), B(L/LR,L/LC)
C arrays A and B with block distribution
  integer dim(2), coords(2)
  logical isper(2)
  integer status(MPI_STATUS_SIZE, 4), req(8),newcomm
  integer srow,lrow,nrow,scol,lcol,ncol
  integer pup,pdown,pleft,pright
  dim(1) = LR
  dim(2) = LC
  isper(1) = .false.
  isper(2) = .false.
  reor = .true.
  call MPI_Init(ierr)
  call MPI_Comm_rank(mpi_comm_world, me, ierr)
  call MPI_Comm_size(mpi_comm_world, nprocs, ierr)
  call MPI_Cart_create(mpi_comm_world,2,dim,isper, .true., newcomm, ierr)
  call MPI_Cart_shift(newcomm,0,1,pup,pdown, ierr)
  call MPI_Cart_shift(newcomm,1,1,pleft,pright, ierr)
  call MPI_Comm_rank (newcomm, me, ierr)
  call MPI_Cart_coords(newcomm,me,2,coords, ierr)
C rows of matrix I have to process
  srow = (coords(1) * L) / dim(1)
  lrow = (((coords(1) + 1) * L) / dim(1))-1
  nrow = lrow - srow + 1
C colomns of matrix I have to process
  scol = (coords(2) * L) / dim(2)
  lcol = (((coords(2) + 1) * L) / dim(2))-1
  ncol = lcol - scol + 1
  call MPI_Type_vector(ncol,1,nrow+2,MPI_DOUBLE_PRECISION,  vectype, ierr)
  call MPI_Type_commit(vectype, ierr)
```

```
  IF (me. eq. 0) PRINT *, '***** TEST_JACOBI *******'
  DO IT = 1, ITMAX
   DO J = 1, ncol
    DO I = 1, nrow
     A(I, J) = B(I, J)
    ENDDO
   ENDDO
C Copying shadow elements of array A from
C neighboring processors before loop execution
  call MPI_Irecv(A(1,0),nrow,MPI_DOUBLE_PRECISION,pleft,1235,MPI_COMM_WORLD,req(1),ierr)
  call MPI_Isend(A(1,ncol),nrow,MPI_DOUBLE_PRECISION,pright,1235,MPI_COMM_WORLD,
 + req(2),ierr)
  call MPI_Irecv(A(1,ncol+1),nrow,MPI_DOUBLE_PRECISION,pright,1236,MPI_COMM_WORLD,
 + req(3),ierr)
  call MPI_Isend(A(1,1),nrow,MPI_DOUBLE_PRECISION,pleft,1236,MPI_COMM_WORLD,req(4),ierr)
  call MPI_Irecv(A(0,1),1,vectype,pup,1237,MPI_COMM_WORLD,req(5),ierr)
  call MPI_Isend(A(nrow,1),1,vectype,pdown,1237,MPI_COMM_WORLD,req(6),ierr)
  call MPI_Irecv(A(nrow+1,1),1,vectype,pdown,1238,MPI_COMM_WORLD,req(7),ierr)
  call MPI_Isend(A(1,1),1,vectype,pup,1238,MPI_COMM_WORLD,req(8),ierr)
  call MPI_Waitall(8,req,status,ierr)
 DO J = 2, ncol-1
  DO I = 2, nrow-1
   B(I, J) = (A( I-1, J ) + A( I, J-1 ) + A( I+1, J) + A( I, J+1 )) / 4
  ENDDO
 ENDDO
ENDDO
  call MPI_Finalize(ierr)
 END
```

Приложение Б. Параллельная Fortran-программа с использованием OpenMP (вычисление числа π, по материалам [9])

Цветом (насыщенностью) выделены относящиеся к OpenMP структуры

```
  PROGRAM COMPUTE_PI
  parameter (n = 1000)
  integer i
  double precision w,x,sum,pi,f,a
  f(a) = 4.D0/(1.D0+a*a)
  w = 1.0D0/n
  sum = 0.0D0;
!$OMP PARALLEL DO PRIVATE(x), SHARED(w)
!$OMP& REDUCTION(+:sum)
  do i=1,n
    x = w*(i - 0.5D0)
    sum = sum + f(x)
  enddo
  pi = w * sum
  print *, 'pi = ',pi
  stop
  end
```

Приложение В. Программа рекурсивного обхода дерева (язык Т++, основная часть кода).

```
// Специальный заголовочный файл txx переопределяет (с помощью макроопределений)
// все ключевые слова, добавленные в язык C++ для обеспечения функциональности T++
  #include <txx>
```

```
#include <stdio.h>
#include <stdlib.h>
struct tree {
struct tree tptr left;
struct tree tptr right;
int value;
};
struct tree tptr create_tree(int deep) {
struct tree tptr res = new tval tree;
res->value = 1;
if (deep <= 1) {
res->left = NULL;
res->right = NULL;
} else {
res->left = create_tree(deep-1);
res->right = create_tree(deep-1);
}
return res;
}
tfun int tsum(struct tree tptr tree) {
tval int leftSum, rightSum;
if (tree->left != NULL) {
leftSum = tsum(tree->left);
} else {
leftSum = tree->value;
}
if (tree->right != NULL) {
rightSum = tsum(tree->right);
} else {
rightSum = tree->value;
}
return leftSum + rightSum;
}
tfun int main (int argc, char* argv[])
{
struct tree tptr tree = create_tree(12);
printf("sum = %d\n", (int) tsum(tree));
return 0;
}
```

Приложение Г. НОРМА- программа умножения матриц [c]=[a]×[b] размерностью a[iMAX][kMAX], b[kMAX][jMAX], c[iMAX][jMAX] на линейке процессоров размерности 12

```
MAIN PART MMmatrix.
! Программа умножения матриц (основной раздел)
BEGIN
  Oi: (i=1..iMAX). ! определение одномерных областей
  Oj: (j=1..jMAX).
  Ok: (k=1..kMAX).
  OA: (Oi; Ok). ! определение двумерных областей
  OB: (Ok; Oj).
```

```
  OC: (Oi; Oj).
  VARIABLE a DEFINED ON OA DOUBLE. ! определение переменных на областях
  VARIABLE b DEFINED ON OB DOUBLE.
  VARIABLE c DEFINED ON OC DOUBLE.
  OUTPUT c(FILE='MMmatrix.out') ON OC. ! операция вывода результирующей матрицы [C]
  DOMAIN PARAMETERS iMAX=500. ! определение констант
  DOMAIN PARAMETERS jMAX=1000.
  DOMAIN PARAMETERS kMAX=1000.
  FOR OA ASSUME a=(i-1)+(k-1). ! коллективная операция для задания
  FOR OB ASSUME b=(k-1)×(j-1). ! начальных значений элементам a[i][k] и b[k][j] матриц
  COMPUTE Pmatrix(a ON OA, b ON OB RESULT c ON OC). ! вызов процедуры Pmatrix
END PART. ! конец основного раздела

PART Pmatrix. ! начало раздела Pmatrix
  a,b RESULT c ! a,b - входные параметры, c - выходной
! тело процедуры Pmatrix
BEGIN
  DOMAIN PARAMETERS iMAX=500.
  DOMAIN PARAMETERS jMAX=1000.
  DOMAIN PARAMETERS kMAX=1000.
  Oi: (i=1..iMAX).
  Oj: (j=1..jMAX).
  Ok: (k=1..kMAX).
  OA: (Oi; Ok).
  OB: (Ok; Oj).
  OC: (Oi; Oj).
  VARIABLE a DEFINED ON OA DOUBLE.
  VARIABLE b DEFINED ON OB DOUBLE.
  VARIABLE c DEFINED ON OC DOUBLE.
  DISTRIBUTION INDEX i=1..12, k=1. ! описание топологии многопроцессорной системы
  FOR OC ASSUME c=SUM((Ok)a[i,k]×b[k,j]). ! коллективная операция
```

$$c_{ij} = \sum_{k=1}^{k=kMAX} a_{ik} \times b_{kj}$$

```
END PART. ! конец раздела Pmatpix
```

Учебное издание.

Баканов Валерий Михайлович

Многомашинные комплексы и многопроцессорные системы

Учебное пособие

ЛР № ________ от __ _______ 2014 г.

Подписано в печать ..2014 г.
Формат 60×84. 1/16.
Объем 7,5 п.л. Тираж 100 экз. Заказ 16.

Московский государственный
университет приборостроения и информатики.
107996, Москва, ул.Стромынка, 20.