Содержание:

Введение

1. Вычислительная модель потоковой обработки
2. Архитектура потоковых вычислительных систем (ПВС)
3. Статические ПВС
4. Динамические ПВС
   4.1 Архитектура потоковых систем с помеченными токенами
   4.2 Архитектура потоковых систем с явно адресуемыми токенами
5. Новые принципы организации вычислительных процессов высокого параллелизма (Бурцев В.С.)
6. Моделирование и реализация потоковой машины на вычислительном кластере
7. Исследование и разработка программного комплекса системного сопровождения для вычислительной системы с автоматическим распределением ресурсов (ВСАРР)
8. Аппаратно-программные средства преодоления неравномерной загрузки отдельных устройств в параллельной ПВС
9. Язык программирования для модели вычислений, основанной на принципе потока данных DCF

Введение

Идеология вычислений, управляемых потоком данных (потоковой обработки), была разработана в 60-х годах Карпом и Миллером. В начале 70-х годов Деннис, а позже и другие начали разрабатывать компьютерные архитектуры, основанные на вычислительной модели с потоком данных.

Под потоковыми ВС понимают обычно ВС, управляемые потоком данных (Data Flow). Это довольно интересный класс вычислительных систем, в которых очередной поток вычислений в соответствии с алгоритмом инициируется не очередными инструкциями программы, а готовностью к обработке необходимых данных. С каждой операцией в программе связан набор ее операндов, которые снабжаются флагами готовности к обработке. При установке всех флагов операция отправляется на выполнение. Потенциально такой подход позволяет достичь высокой степени параллелизма, так как потоковая архитектура пытается извлечь максимальный параллелизм из всех заданных вычислений.
Эффективность подобных структур во многом определяется эффективным программированием, задачей которого является формулировка задачи в терминах параллельных и независимых операций. На первый план выходит не эффективное построение процедуры вычислений, а выявление взаимосвязей между потоками данных в задаче.
В значительной степени это задача соответствующего оптимизирующего компилятора. Известно много проектов потоковых вычислительных систем, среди них можно отметить Манчестерскую ВС (Манчестерский университет), Tagged Token, Monsoon (Массачусетский технологический институт), Sigma, EMS, EMC-4 (Тсукуба, Япония), RAPID (проект Sharp - Mitsubishi - университет Осаки, Япония) и др. Как утверждает ряд авторов, многие интересные проекты Data Flow не были по достоинству оценены производителями вычислительной техники, в силу нетрадиционности подхода.
С другой стороны, потоковые вычисления нашли применение в современных суперскалярных процессорах и процессорах с длинным командным словом. Причем если в VLIW - структурах задача выявления параллельных потоков в большей степени решается программным обеспечением, то в суперскалярных процессорах логика управления многопотоковым конвейером как раз и реализует механизм, напоминающий управление потоком данных, но средствами самого процессора. Действительно, последовательный поток инструкций программы в суперскалярном процессоре, имеющем много параллельно работающих функциональных устройств и буфер команд, ожидающих готовности операндов, транслируется в параллельные потоки внутренних инструкций, операнды которых снабжаются признаками готовности, которые, в том числе, зависят и от работы механизма динамической оптимизации команд. В качестве примеров таких микропроцессоров можно привести HP PA-8000 и Intel Pentium Pro. Так что системы, управляемые потоком данных, так или иначе, находят применение в современной вычислительной технике.

Вычислительная модель потоковой обработки

В потоковой вычислительной модели для описания вычислений используется ориентированный граф, иногда называемый *графом потоков данных* (dataflow graph). Этот граф состоит из *узлов* или *вершин,* отображающих операции, и *ребер* или *дуг,* показывающих потоки данных между теми вершинами графа, которые они соединяют. Узловые операции выполняются, когда по дугам в узел поступила вся необходимая информация. Обычно узловая операция требует одного или двух операндов, а для условных операций необходимо наличие входного логического значения. По выполнении операции формируются один или два результата. Таким образом, у каждой вершины может быть от одной до трех входящих дуг и одна или две выходящих. После активации вершины и выполнения узловой операции (это иногда называют *инициированием вершины)* результаты передаются по ребрам к ожидающим вершинам. Процесс повторяется, пока не будут инициированы все вершины и получен окончательный результат. Одновременно может быть инициировано несколько узлов, при этом параллелизм в вычислительной модели выявляется автоматически.

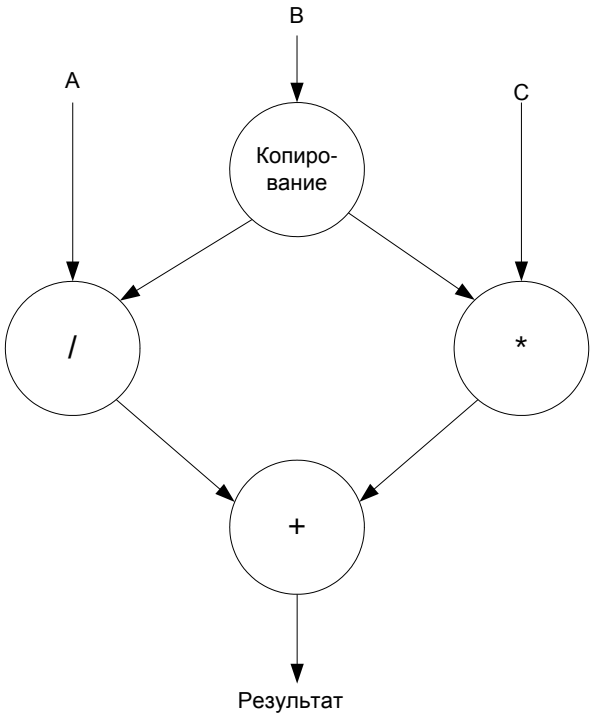


Рисунок 1. Граф потоков данных для выражения A/C+B*C

На рисунке 1 показан простой потоковый граф для вычисления выражения *f=A/B + B*C.* Входами служат переменные A, B и C, изображенные вверху графа. Дуги между вершинами показывают тракты узловых операций. Направление вычислений — сверху вниз. Используются три вычислительные операции: сложение, умножение и деление. Заметим, что B требуется в двух узлах. Вершина «Копирование» предназначена для формирования дополнительной копии переменной B.

Данные (операнды/результаты), перемещаемые вдоль дуг, содержаться в опознавательных информационных кадрах, маркерах специального формата — *«токенах»* (иначе «фишках» или маркерах доступа). Рисунок 2 иллюстрирует движение токенов между узлами. После поступления на граф входной информации, маркер, содержащий значение A, направляется в вершину деления; токен с переменной B — в вершину копирования; токен с переменной C—в вершину умножения. Активирована может быть только вершина «Копирование», поскольку лишь один вход и на нем уже присутствует токен. Когда токены из вершины «Копирование» будут готовы, узлы умножения и деления также получат все необходимые маркеры доступа и могут быть инициированы. Последняя вершина ждет завершения операций умножения и деления, то есть когда на ее входе появятся необходимые токены.

Практические вычисления требуют некоторых дополнительных возможностей, например при выполнении команд условного перехода. По этой причине в потоковых графах предусмотрены вершины-примитивы нескольких типов (рисунок 3):

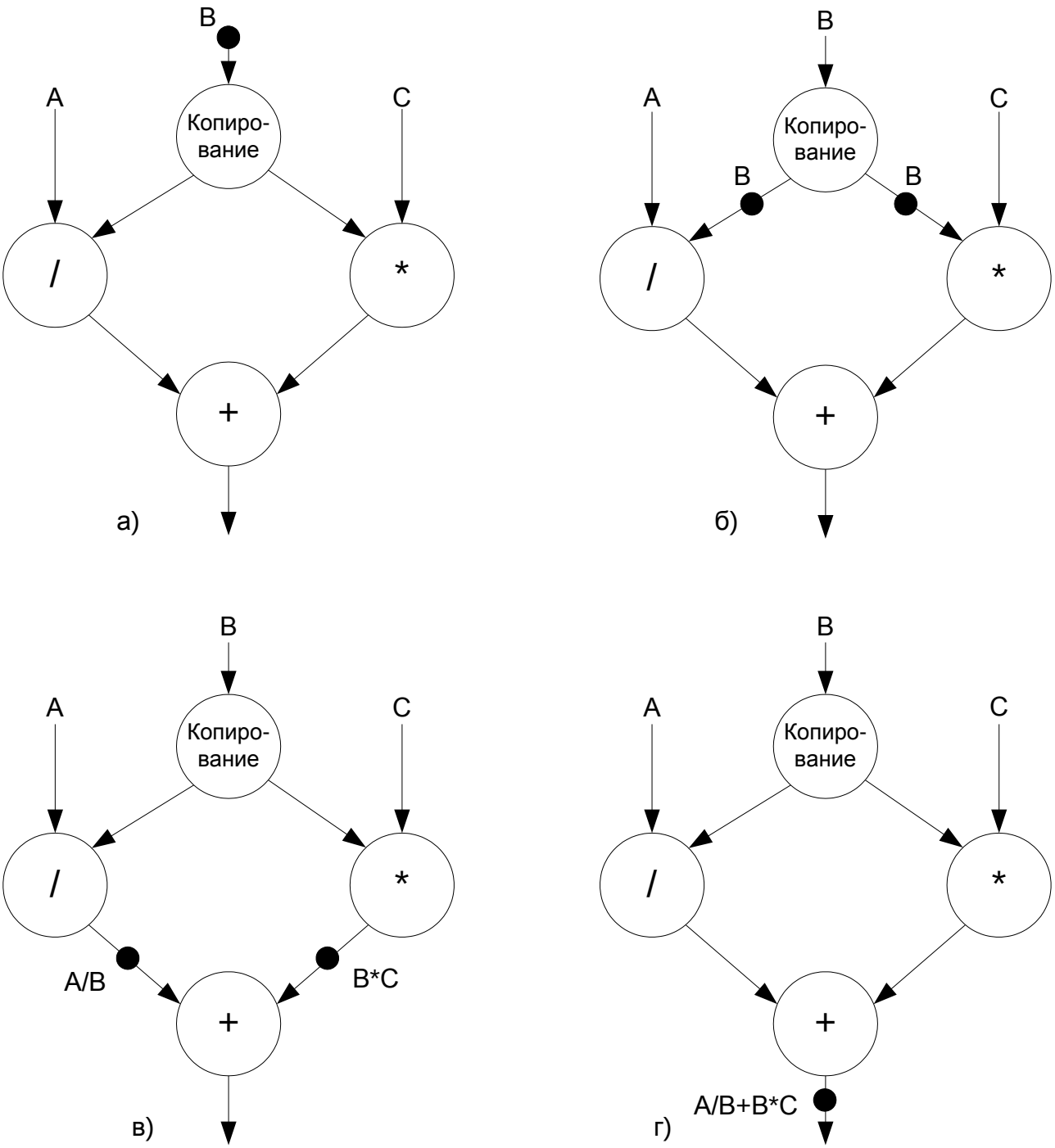


Рисунок 2. Движение маркеров при вычислении A/B+B*C: а – после передачи входных данных; б – после копирования; в – после умножения и деления; г – после суммирования

☑ *двухвходовая операционная вершина* - узел с двумя входами и одним выходом. Операции производятся над данным, поступающими с левой и правой входных дуг, а результат выводится через выходную дугу;

☑ *одновходовая операционная вершина* - узел с одним входом и одним выходом. Операции выполняются над входными данными, результат выводится через выходное ребро;

☑ *вершина ветвления* - узел с одним входом и двумя выходами. Осуществляет копирование входных данных и их вывод через две выходные дуги. Путем комбинации таких узлов можно строить вершины на m выходов;

☑ *вершина слияния* - узел с двумя входами и одним выходом. Данные поступают только с какого-нибудь одного из двух входов. Входные данные без изменения подаются на выход. Комбинируя такие узлы, можно строить вершины слияния с m входами;

☑ *вершина управления* — существует в перечисленных ниже трех вариантах:

- *TF-коммутатор* — узел с двумя входами и одним выходом. Верхний вход — это дуга данных, а правый — дуга управления (логические данные). Если значение правого входа истинно (T — True), то входные данные выводятся через левый выход, а при ложном значении на правом входе (F — False) данные следуют через правый выход;
- *вентиль* — узел с двумя входами и одним выходом. Верхний вход — дуга данных, а правый — дуга управления. При истинном значении на входе управления данные выводятся через выходную дугу;
- *арбитр* — узел с двумя входами и двумя выходами. Все дуги являются дугами данных. Первые поступившие от двух входов данные следуют через левую дугу, а прибывшие впоследствии — через правую выходную дугу. Активация вершины происходит в момент прихода данных с какого-либо одного входа.

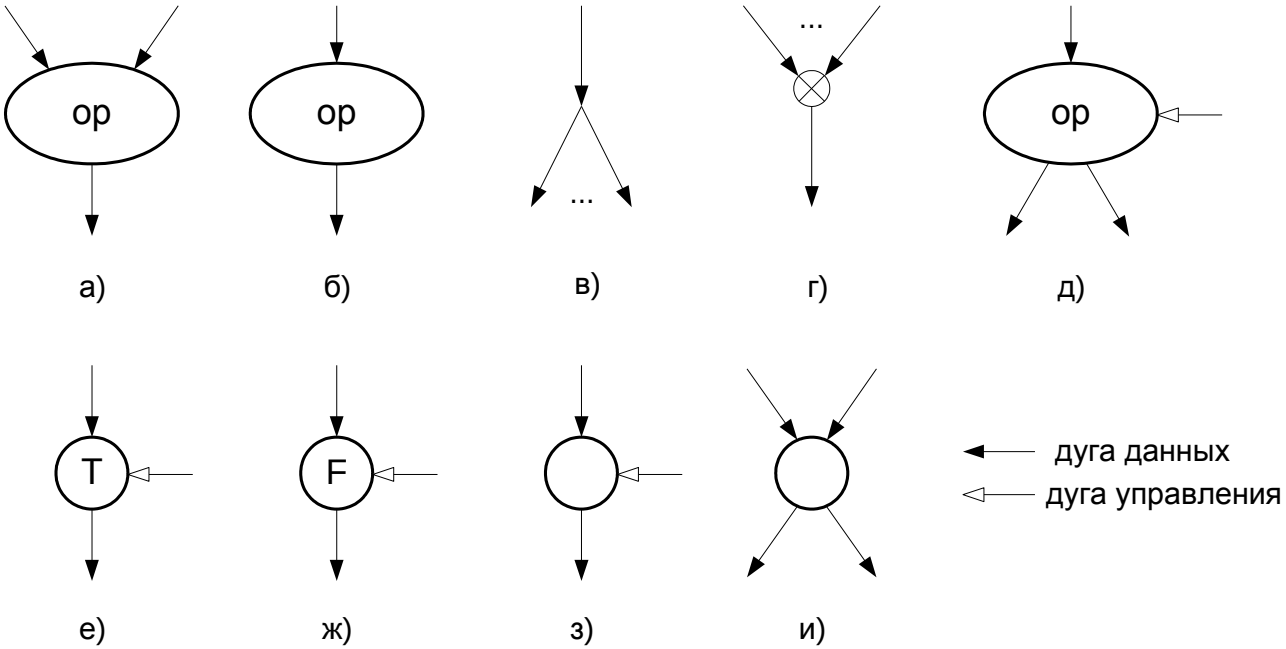


Рисунок 3. Примитивы узлов: а) – двухвходовая операционная вершина; б) – одновходовая операционная машина; в) – вершина ветвления; г) – вершина слияния; д) – TF-коммутатор; е) – T-коммутатор; ж) – F-коммутатор; з) – вентиль; и) – арбитр

Процесс обработки может выполняться аналогично конвейерному режиму: после обработки первого набора входных сигналов на вход графа может быть подан второй и т. д. Отличие состоит в том, что промежуточные результаты (токены) первого вычисления не обрабатываются совместно с промежуточными результатами второго и последующих вычислений. Результаты обычно требуются в последовательности использования входов.

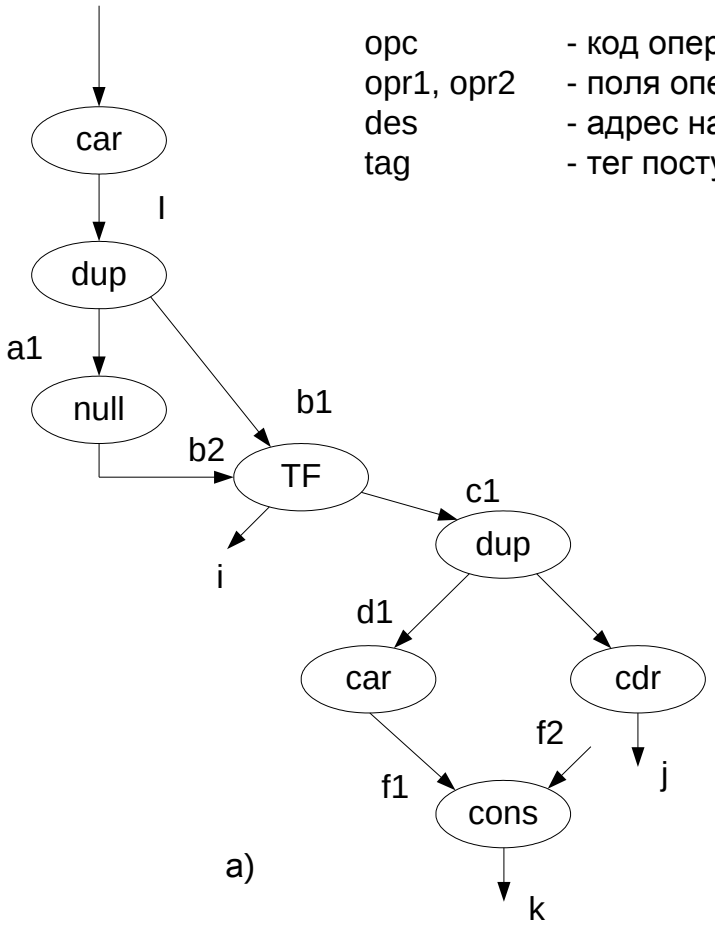
Существует множество случаев, когда определенные вычисления должны повторяться с различными данными, особенно в программных циклах. Программные циклические процессы в типовых языках программирования могут быть воспроизведены путем подачи

результатов обратно на входные узлы. При формировании итерационного кода часто применяются переменные цикла, увеличиваемые после каждого прохода тела цикла. Последний завершается, когда переменная цикла достигает определенного значения.

Все известные потоковые вычислительные системы могут быть отнесены к двум основным типам: *статическим* и *динамическим.* В свою очередь, динамические потоковые ВС обычно реализуются по одной из двух схем: *архитектуре с помеченными токенами* и *архитектуре с явно адресуемыми токенами.*

Архитектура потоковых вычислительных систем

В потоковых ВС программа вычислений соответствует потоковому графу, который хранится в памяти системы в виде таблицы. На рисунке 4 показаны пример графа потоковой программы и содержание адекватной ему таблицы.



opc - код операции;
opr1, opr2 - поля операндов;
des - адрес назначения;
tag - тег поступления данных (2 бита)

	opc	opr1	opr2	des	tag
	car			l	
l	dup			a1, b1	0 ,1
a	null			b2	0 ,1
b	TF			i, c1	0 ,1
c	dup			d1, e1	0 ,1
d	car			f1	0 ,1
e	cdr			j	0 ,1
f	cons			k	0 ,1

б)

Рисунок 4. Пример формы хранения потоковой программы: а) – потоковый граф; б) – память функционального блока

Принципиальная схема потоковой вычислительной системы (рисунок 5) включает в себя блок управления (CS), где хранится потоковый граф, который используется для выборки обрабатываемых команд, а также функциональный блок (FS), выполняющий команду, переданную из CS, и возвращающий результат ее выполнения в CS.

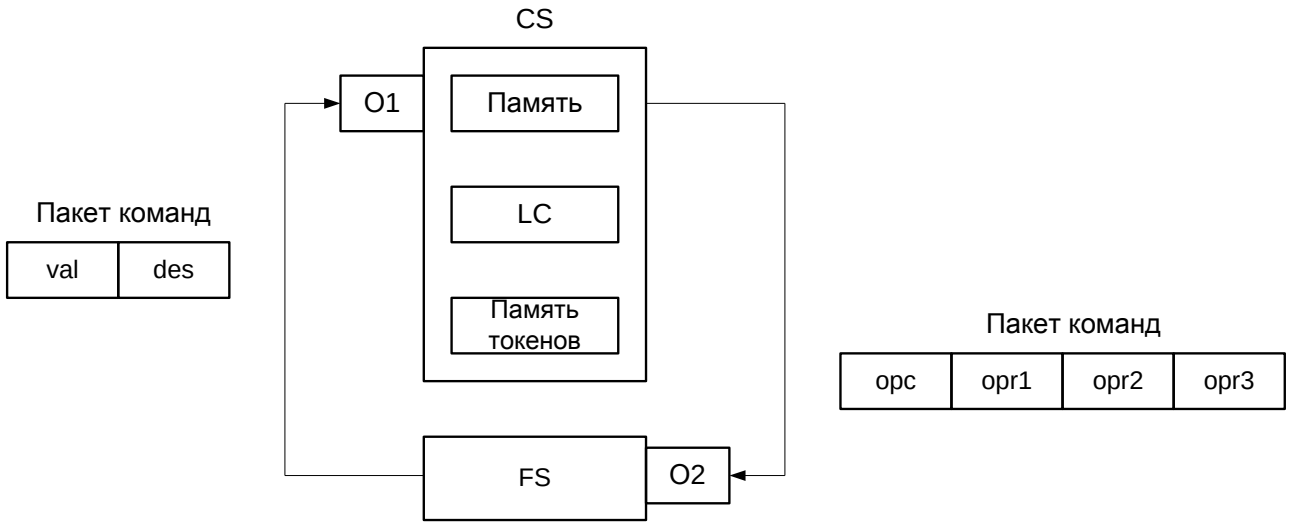


Рисунок 5. Структура потоковой вычислительной машины

Блоки CS и FS работают асинхронно и параллельно, обмениваясь многочисленными пакетами команд и результатами их выполнения. В пакете результата, поступающем из блока FS, содержится значение результата (val) и адрес команды, для которой пакет предназначен (des). На основании этого адреса блок CS проверяет возможность обработки команды. Команда может быть одноперандной или двухоперандной. В последнем случае необходимо подтверждение наличия обоих операндов (opr 1 и opr 2), и для этого устанавливается специальный признак. Блок управления загрузкой (LC) каждый раз при активировании определенной функции загружает из памяти программ код этой функции. Для повышения степени параллелизма блоки CS и FS строятся по модульному принципу, а графы потоковой программы распределяются между модулями с помощью мультиплексирования.

Статические потоковые вычислительные системы.
Статическая потоковая архитектура, известная также под названием «единственный – токен – на - дугу» (single-token-per-arc dataflow), была предложена Деннисом в 1975 году. В ней допускается присутствие на ребре графа не более чем одного токена. Это выражается в правиле активации узла: вершина активируется, когда на всех ее входных дугах присутствует по токену и ни на одном из ее выходов токенов нет. Для указания вершине о том, что ее выходной токен уже востребован последующим узлом (узлами) графа, в ВС обычно прибегают к механизму подтверждения с квитированием связи, как это показано на рисунке 6. Здесь процессорами в ответ на инициирование узлов графа посылаются специальные контрольные токены.

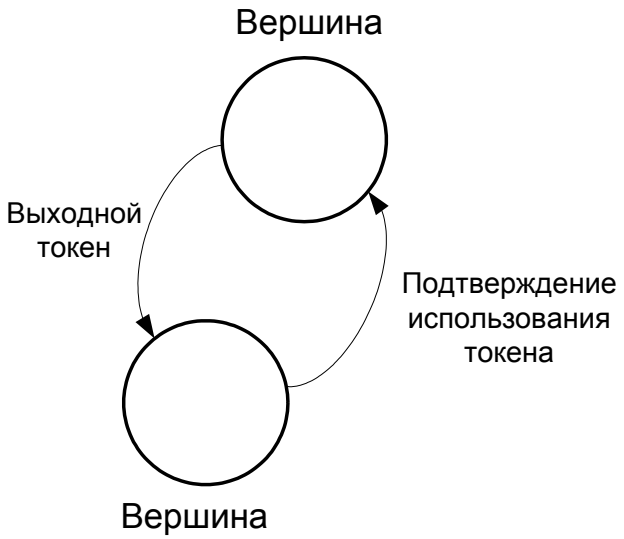


Рисунок 6. Механизм подтверждения с квитированием

Типовую статическую потоковую архитектуру иллюстрирует рисунок 7.

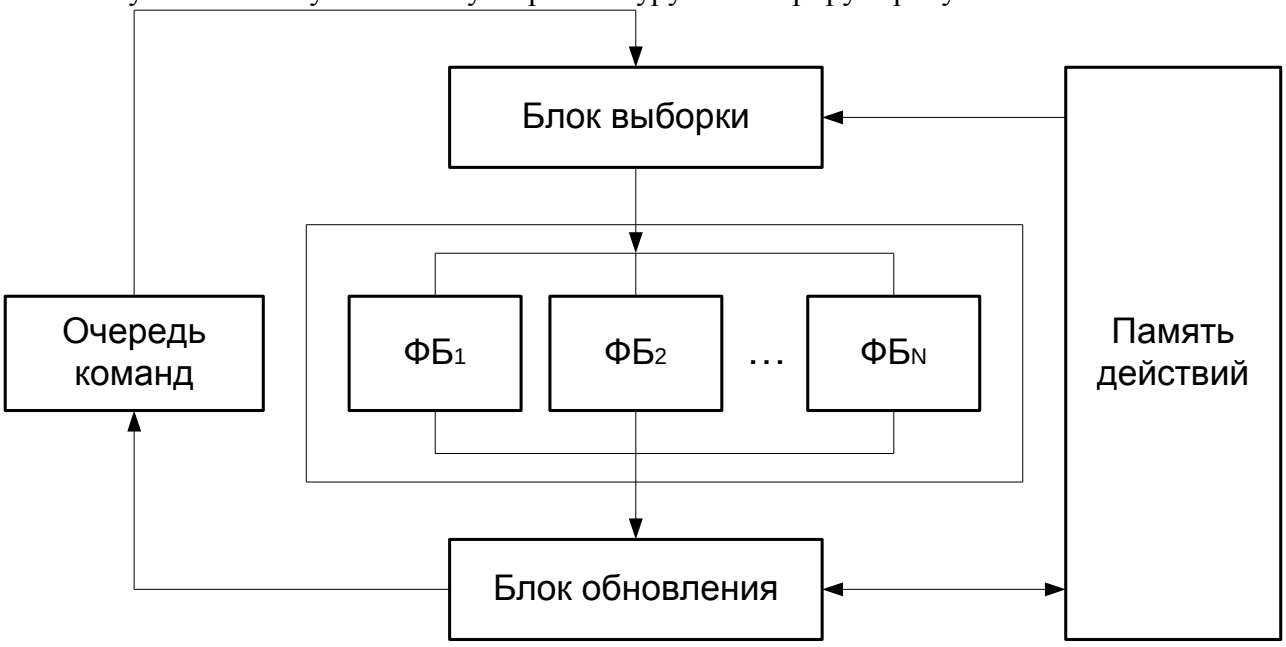


Рисунок 7. Структура процессорного элемента типовой статической потоковой системы

Память действий (узел потоковой машины, хранящий потоковый граф и циркулирующий в нем поток данных) состоит их двух блоков памяти: команд/данных и управляющей памяти. Вершина графа представлена в памяти команд/данных кадром, содержащим следующие поля:
код операции;

операнд 1:
……
операнд N,
вершина/дуга 1;
…….
вершина/дуга К.

Каждому кадру в памяти команд/данных соответствует кадр в управляющей памяти, содержащий биты наличия для всех операндов и всех полей «вершина/ дуга». Бит наличия операнда устанавливается в единицу, если этот операнд доступен, то есть если токен, содержащий данный операнд, уже поступил по входной дуге графа. Для поля «вершина/дуга» установленный бит наличия означает, что выходная дуга, ассоциированная с данным полем, не содержит токена. Вершина графа, описанная в памяти команд/данных, может быть активирована (операция может быть выполнена), если все биты наличия в соответствующем кадре памяти управления установлены в единицу. Когда данная ситуация распознается блоком обновления, он помещает пакет команды в очередь команд. Опираясь на очередь команд и содержимое памяти действий, блок выборки составляет пакет операции и направляет его в один из функциональных блоков. После выполнения требуемой операции функциональный блок создает пакет результата и передает его в блок обновления, который в соответствии с полученным результатом изменяет содержимое памяти действий.

Основное преимущество рассматриваемой модели потоковых вычислений заключается в упрощенном механизме обнаружения активированных узлов. К сожалению, статическая модель обладает множеством серьезных недостатков. Во-первых, данный механизм не допускает параллельного выполнения независимых итераций цикла. Другой нежелательный эффект — колебания трафика токенов. Наконец, в современных языках программирования отсутствует поддержка описанного режима обработки данных. Несмотря на это, несколько образцов статических ВС все-таки были созданы или, по крайней мере, спроектированы.

Архитектуру первой системы, строго соответствующей модели потоковых вычислений типа «единственный-токен-на-дугу», предложил Деннис (рисунок 8).

По изначальной идее, система представляла собой кольцо из процессорных элементов и элементов памяти, в котором информация передается в форме пакетов.

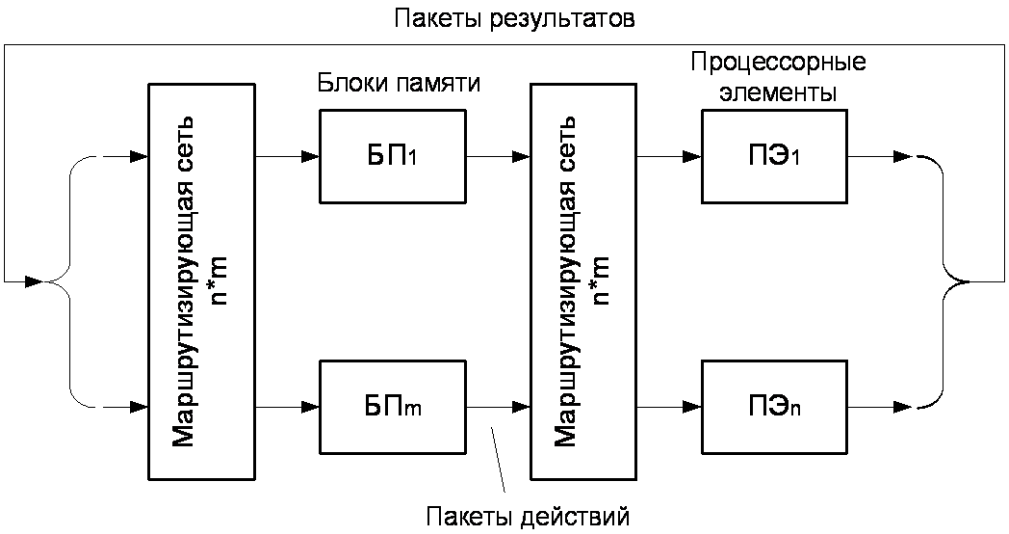


Рисунок 8. Статическая потоковая архитектура по Деннису

Процессорные элементы должны были получать так называемые *пакеты действий* (activity templates) в виде: Код операции • Операнды • Адресат. Здесь точка обозначает

операцию составления целого слова из его частей, «Код операции» определяет подлежащую выполнению операцию, «Операнды» — используемые в операции числа, а «Адресат» — указывает место, куда должен быть направлен результат операции. Отметим, что в полях операндов задаются не адреса ячеек памяти, а непосредственно числа, которые должны участвовать в операции. Это имеет свои преимущества и недостатки. Первое состоит в том, что в каждый момент времени операнды могут быть использованы только одной выбранной вершиной. Изъян идеи — невозможность включения в команду в качестве операндов сложных структур данных, даже простых векторов и массивов.

Пакет результата имеет вид: Значение • Адресат, где в поле «Значение» содержится значение результата, полученное после выполнения операции. Эти пакеты передаются по маршрутизирующей сети в так называемые *ячейки команды* блока памяти, а именно в указанные в их поле «Адресат». Когда получены все входные пакеты (токены), ячейка команды порождает пакет операции. Обычно для генерации пакета операции ячейке команды нужны два входных пакета с операндами. Затем пакет операции маршрутизируется к одному из процессорных элементов (ПЭ). Если все процессорные элементы идентичны (гомогенная система), может быть выбран любой свободный ПЭ. В негомогенных системах со специализированными ПЭ, способными выполнять только определенные функции, выбор нужного ПЭ производится по коду операции, заключенном в пакете операции.

В несколько усовершенствованном варианте рассмотренная ВС была спроектирована под руководством того же Денниса в Массачусетсском технологическом институте (MIT). Система состояла из пяти основных подсистем: секции памяти, секции процессоров, сети арбитража, сети управления и сети распределения. Все коммуникации между подсистемами осуществлялись путем асинхронной передачи пакетов по каналам. Прототип ВС содержал восемь процессорных элементов на базе микропроцессоров и сеть маршрутизации пакетов, построенную на маршутирующих элементах 2x2.

В качестве других примеров статических потоковых ВС можно упомянуть: LAU System, TI`s Distributed Data Processor, DDMI Utah Data Driven Machine, Nec Image Pipelined Processor, Hughes Dataflow Multiprocessor.

Динамические потоковые вычислительные системы

Производительность потоковых систем существенно возрастает, если они в состоянии поддерживать дополнительный уровень параллелизма, соответствующий одновременному выполнению отдельных итераций цикла или параллельной обработке пар элементов в векторных операциях. Кроме того, в современных языках программирования активно используются так называемые *реентерабельные процедуры*, когда в памяти хранится только одна копия кода процедуры, но эта копия является повторно входимой (реентабельной). Это означает, что к процедуре можно еще раз обратиться, не дожидаясь завершения действий в соответствии с предыдущим входом в данную процедуру. Отсюда желательно, чтобы все обращения к реентабельной процедуре также обрабатывались параллельно. Задача обеспечения дополнительного уровня параллелизма решается в динамических потоковых ВС и реализуется двумя вариантами архитектуры потоковой ВС: *архитектуры с помеченными токенами* и *архитектуры с явно адресуемыми токенами.*

Архитектура потоковых систем с помеченными токенами

В архитектуре с помеченными токенами (tagget-token architecture) память хранит один экземпляр потокового графа. Каждый токен содержит *тег* (фишку), состоящий из адреса команды, для которой предназначено заключенное в токене значение, и другой информации, определяющей вычислительный контекст, в котором данное значение используется, например номера итерации цикла. Этот контекст называют «цветом значения», а токен соответственно называют «окрашенным», в силу чего метод имеет еще одно название — *метод окрашенных токенов.* Каждая дуга потокового графа может рассматриваться как вместилище, способное содержать произвольное число токенов с различными тегами. Основное правило активирования вершины в динамических потоковых ВС имеет вид: *вершина активируется, когда на всех ее входных дугах присутствуют токены с идентичными тегами.*

Типовая архитектура потоковой системы с помеченными токенами показана на рисунке 10. Для обнаружения одинаково окрашенных токенов (токенов с одинаковыми тегами) в конвейер процессорного элемента введен согласующий блок. Этот блок получает очередной токен из очереди токенов и проверяет, нет ли в памяти согласования его партнера (токена с идентичным тегом). Если такой партнер не обнаружен, принятый токен заносится в память согласования. Если же токен-партнер уже хранится в памяти, то блок согласования извлекает его оттуда и направляет оба токена с совпавшими тегами в блок выборки. На основе общего тега блок выборки находит в памяти команд/данных соответствующую команду и формирует пакет операции, который затем направляет в функциональный блок. Функциональный блок выполняет операцию, создает токены результата и помещает их в очередь токенов.

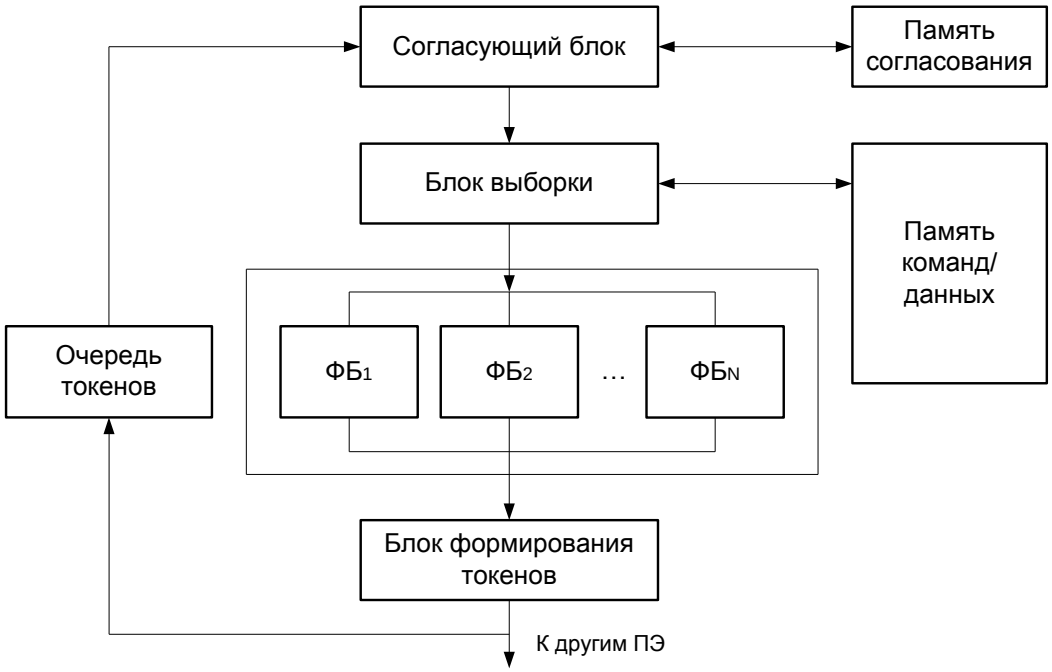


Рисунок 10. Структура процессорного элемента типовой потоковой системы с помеченными токенами

Чтобы учесть возможные ситуации — циклы, элементы массивов, функции и рекурсии, — тег каждого токена должен включать в себя три поля: Уровень итерации • Имя функции • Индекс.

В каждом поле может содержаться число, начиная с нуля. Поле «Уровень итерации» хранит порядковый номер текущей итерации цикла; поле «Имя функция» идентифицирует вызов функции; поле «Индекс» указывает на определенный элемент массива. Первые два поля могут быть объединены в одно.

Индивидуальные поля тега трактуются как числовые значения, пересылаемые от одной вершины к другой. Минимальным требованием к динамической потоковой ВС является наличие операций, позволяющих извлечь значение, содержащееся в каждом поле, или установить в любом поле иное значение. Например, операция «Прочитать поле токена» берет токен и формирует из него другой токен, где определенное поле заполнено содержимым аналогичного поля из входного токена. Операция «Установить поле токена» формирует выходной токен, совпадающий с входным, за исключением указанного поля, куда заносится значение, заданное в данной операции в виде литерала. Применительно к полю «Уровень итерации могут быть предусмотрены операции инкремента и декремента.

Структура динамической потоковой ВС с архитектурой окрашенных токенов, созданной в Манчестерском университете Гурдом и Ватсоном в 1980 году. В этой ВС используется конвейерная кольцевая архитектура, и вычислительные функции отделены от функций согласования токенов и тегов. Каждый процессор обрабатывает пакеты длиной в 166 бит, содержащие два численных операнда, тег, код операции и один или два адреса следующей команды или команд. С каждым пакетом ассоциирован флаг «Система/Вычисление», позволяющий отличить выполняемый пакет (пакет, который должен быть обработан процессором) и пакет, несущий системное сообщение, такое, например, как «Загрузить команду в память». Численными операндами могут быть 32-разрядные целые числа, либо числа в формате с плавающей запятой.

Обработав выполняемый пакет, процессор генерирует один или два пакета (токена) результата. Пакет результата состоит из 96 битов и содержит один численный операнд, тег и адрес (адреса) пункт назначения результата. Каждый пакет результата поступает на

ключ, обеспечивающий либо ввод данных от внешнего источника, либо вьвод на внешний объект (периферийное устройство). Учитывая, что токен результата должен быть передан на другую вершину потокового графа, он направляется в очередь токенов, а оттуда — в блок согласования токенов. Здесь производится поиск других токенов, необходимых для инициирования нор шины (если для этого нужны два токена). Поиск ведется аппаратно, посредством сравнения пункта назначения и тега входного токена с аналогичными параметрами всех хранящихся в памяти токенов. Если совпадение произошло, из пары токенов формируется пакет, в противном случае входной токен запоминается в блоке согласования токенов до момента, когда поступит согласующийся с ним токен Пары токенов и токены с одним операндом передаются в память программ, содержащую узловые команды, и формируется полный выполняемый пакет. Выполняемые пакеты по возможности направляются в свободный процессор в массиве из 15 процессоров.
В качестве других примеров динамических потоковых вычислительных систем следует упомянуть: SIGMA-1, PATTSY Processor Array Tagged-Token System, NTT`s Dataflow Processor Array System, DDDP Distributed Data Driven Processor, SDFA Statelles Data-Flow Architecture.
Основное преимущество динамических потоковых систем — это более высокая производительность, достигаемая за счет допуска присутствия на дуге множества токенов. При этом, однако, основной проблемой становится эффективная реализация блока, который собирает токены с совпадающим цветом (токены с одинаковыми тегами). В плане производительности этой цели наилучшим образом отвечает ассоциативная память. К сожалению, такое решение является слишком дорогостоящим, поскольку число токенов, ожидающих совпадения тегов, как правило, достаточно велико. По этой причине в большинстве вычислительных систем вместо ассоциативных запоминающих устройств (ЗУ) используются обычные адресные ЗУ. В частности, сравнение тегов в рассмотренной манчестерской системе производится с привлечением хэширования, что несколько снижает быстродействие.

В последнее время все более популярной становится другая организация динамической потоковой ВС, позволяющая освободиться от ассоциативной памяти и известная как архитектура с явно адресуемыми токенами.

Архитектура потоковых систем с явно адресуемыми токенами

Значительным шагом в архитектуре потоковых ВС стало изобретение механизма явной адресации токенов (explicit token-store), имеющего и другое название – непосредственное согласование(direct matching). В основе этого механизма лежит то, что все токены в одной и той же итерации цикла и в одном и том же вхождении в реентерабельную процедуру имеют идентичный тег (цвет). При инициализации очередной итерации цикла или очередном обращении к процедуре формируется так называемый кадр токенов, содержащий токены, относящиеся к данной итерации или данному обращению, то есть с одинаковыми тегами. Использование конкретных ячеек внутри кадра задается на этапе компиляции. Каждому кадру выделяется отдельная область в специальной памяти кадров (frame memory), причем раздача памяти под каждый кадр происходит уже на этапе выполнения программы.

В схеме с явной адресацией токенов любое вычисление полностью описывается *указателем команды* (IP, Instruction Pointer) и *указателем кадра* (FP, Frame Pointer). Этот кортеж <FP, IP> входит в тег токена, а сам токен выглядит следующим образом: Значение • FP.IP.

Команды, реализующие потоковый граф, хранятся в памяти команд и имеют формат: Код операции • Индекс в памяти кадров • Адресат.

Здесь «Индекс в памяти кадров» определяет положение ячейки с нужным токеном внутри кадра, то есть какое число нужно добавить к FP, чтобы получить адрес интересующего токена. Поле «Адресат» указывает на местоположение команды, которой должен быть

передан результат обработки данного токена. Адрес в этом поле также задан в виде смещения - числа, которое следует прибавить к текущему значению IP, чтобы получить исполнительный адрес команды назначения в памяти команд. Если потребителей токена несколько, в поле «Адресат» заносится несколько значений смещения.

Каждому слову в памяти кадров придан бит наличия, единичное значение которого удостоверяет, что в ячейке находится токен, ждущий согласования, то есть что одно из искомых значений операндов уже имеется. Как и в архитектуре с окрашенными токенами, определено, что вершины могут иметь максимум две входные дуги. Когда на входную дугу вершины поступает токен <v1, <FP, IP>>, в ячейке памяти кадров с адресом FP + (IP.I) проверяется бит наличия (здесь IP.I означает содержимое поля I в команде, хранящейся по адресу, указанному в IP). Если бит наличия сброшен (ни один из пары токенов еще не поступал), поле значения пришедшего токена (v1) заносится в анализируемую ячейку памяти кадров, а бит наличия в этой ячейке устанавливается в единицу, фиксируя факт, что первый токен из пары уже доступен:

(FP+(IP.I).значение :=v1

(FP+(IP.I).наличие :=1

Этот случай, когда на вершину SUB по левой входной дуге поступил токен <35, <FP, IP>>.

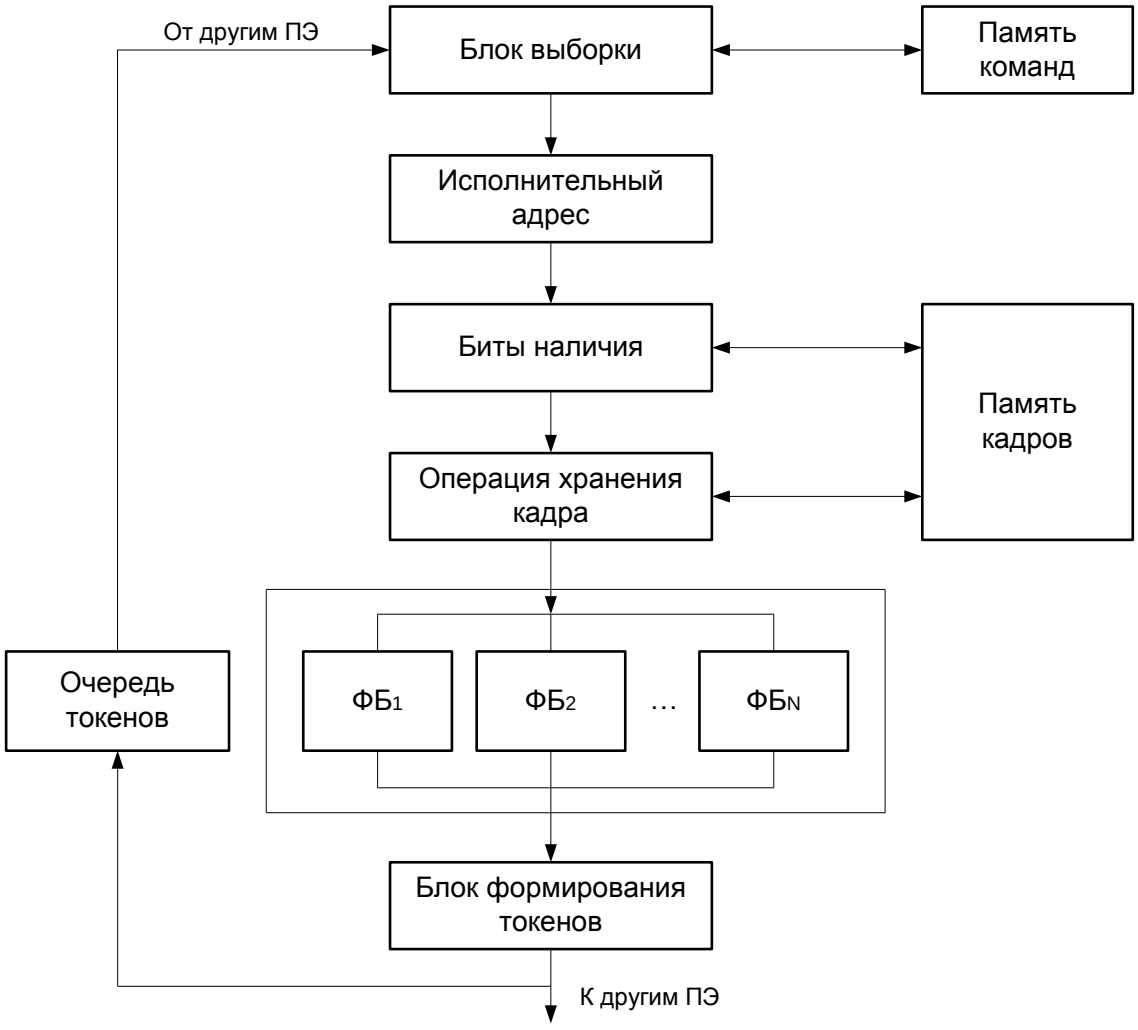


Рисунок 11. Структура процессорного элемента типовой потоковой системы с явной адресацией токенов

Если токен <v2,<FP, IP>> приходит на узел, для которого уже хранится значение v1, команда, представляющая данную вершину, может быть активирована и выполнена с операндами v1 и v2. В этот момент значение v1 извлекается из памяти кадров, бит наличия

сбрасывается, и на функциональный блок, предназначенный для выполнения операции, передается пакет команды <v1, v2, FP, IP, IP.OP, IP.D>, содержащий операнды (v1 и v2), код операции (IP.OP) и адресат ее результата (IP.D). Входящие в этот пакет значения FP и IP нужны, чтобы вместе с IP.D вычислить исполнительный адрес адресата. После выполнения операции функциональный блок пересылает результат в блок формирования токенов.

Типовая архитектура системы с явной адресацией токенов показана на рисуноке 11. Отметим, что функция согласования токенов стала достаточно короткой операцией, что позволяет внедрить ее в виде нескольких ступеней процессорного конвейера.

Новые принципы организации вычислительных процессов высокого параллелизма (Бурцев В.С.).

Цель работы: создание аппаратных и программных вычислительных средств на базе новых
архитектурных и технологических решений, превосходящих по реальной производительности и целому ряду
основных параметров современные вычислительные системы (ВС)

1. Значение высокопроизводительных вычислительных систем

Наступившее столетие безусловно будет веком бурного развития информатизации. Она базируется в
первую очередь на сверхвысокопроизводительные комплексы, построенных на базе суперЭВМ, рабочих
станций, средствами взаимодействия их с человеком через персональные компьютеры и средств
объединения этих комплексов в единую телекоммуникационную систему.

Настоящий проект рассматривает возможность построения суперЭВМ, рабочих станций и телекоммуникационных процессоров **на новых принципах** организации вычислительных процессов
происходящих в них, которые существенно снижают требования к целому ряду важных параметров
элементно-конструкторской базы.

Отставание от мирового уровня в оснащенности высокопроизводительными вычислительными
системами (включая суперЭВМ) научно-исследовательских институтов РАН, отраслевых НИИ и КБ, а также
университетов и вузов страны достигло **критической точки**, и если срочно не принять необходимые меры,
то в самом ближайшем будущем Россия утратит ведущие позиции в науке, в создании наукоемкой
технологии, проектировании сложных объектов и процессов.

Современный этап мирового развития компьютерной техники характеризуется переходом на
высокопроизводительные вычислительные системы высокого параллелизма, что сопровождается
радикальным пересмотром используемых математических методов и информационных технологий. Это
открывает качественно новые возможности для решения ряда глобальных научно-технических проблем. В
качестве таких фундаментальных научных проблем, многие из которых имеют непосредственное
практическое значение, можно назвать:

- прогноз погоды, экологических изменений и природных геофизических явлений, таких как климат,
загрязнение среды, сейсмическая обстановка;
- исследование структуры сложных газодинамических и гидродинамических течений (развитие
гидродинамических неустойчивостей, природы турбулентности и т.д.) с целью создания сложных
аэродинамических комплексов нового поколения;
- изучение свойств вещества в интересах атомной энергетики;
- структурное исследование человеческих генов;

□□молекулярное конструирование лекарств;
□□проектирование сверхсложных радиоэлектронных комплексов и т.д.
Успешное решение этих проблем во многом определяет темпы развития научного, технического и оборонного потенциала страны.
По зарубежному опыту число суперЭВМ в самых развитых странах измеряется десятками штук. Но ни одна перспективная разработка, имеющая государственное и мировое значение, без суперЭВМ обойтись не может.
Этим обстоятельством объясняется жесткое ограничение на продажу суперЭВМ в Россию. К классу суперЭВМ относятся самые быстродействующие ЭВМ на данный период времени.
Поэтому продажа ЭВМ в Россию ограничивается по их производительности. США установили эмбарго на продажу высокопроизводительных вычислительных средств, ограничив производительность на два порядка ниже, чем производительность суперЭВМ находящихся в продаже в Америке при условии, что продающая сторона осуществляет полный контроль над выполняемыми на ней работами.
В настоящее время требуемая производительность суперЭВМ для решения вышеперечисленных проблемных задач оценивается 1012 - 1015 оп/с с соответствующими объемами оперативной памяти высокого быстродействия (1012 - 1015 бит). Причем, эти требования непрерывно увеличиваются в виду того, что чем выше производительность вычислительной системы, тем качественнее может быть поставлен математический эксперимент за более короткое время.
В настоящее время в США начата разработка вычислительной системы производительностью 1015 оп/с. Срок окончания работ 2005-2013 г.г.
Рассчитывать на продажу России вычислительных систем требуемой высокой производительности не приходится, так как та страна, у которой будут иметься более мощные вычислительные системы, будет иметь наивысший темп развития научно-технического потенциала.

Основные технические предпосылки необходимости и возможности реализации процессоров, в которых используются новые принципы организации вычислительных процессов

1.Фактически достигнут физический предел увеличения производительности одного микропроцессора.
В настоящее время, ведущими фирмами Intel и IBM предпринята попытка, увеличить производительность одного микропроцессора за счет схемотехнических решений – увеличить число исполнительных устройств в одном кристалле. На некоторых задачах (далеко не на всех) такие микропроцессоры достигают производительности 3-5 Гфлоп в секунду. На большинстве задач такие процессоры загружены не полностью. Так что этот путь малоэффективен и имеет как схемотехнические так и физические ограничения.
2.Основным методом увеличения производительности вычислительных средств (ВС) становится метод распараллеливания вычислительных процессов.
3.На математическом уровне задача распараллеливания вычислительного процесса успешно решена:
большинство физических задач сводится к решению уравнений математической физики, т.е. к решению задач векторно-матричного характера. К этим же задачам сводятся задачи комбинаторного характера, задачи распознавания образов и задачи, относящиеся к классу искусственного интеллекта. Векторно-матричные вычисления, как известно, обладают высокой степенью параллелизма (перемножение матрицы второго порядка ранга 100 имеет параллелизм равный 10 6). Наши исследования показали, что многие задачи обладают высокой степенью т.н. «скрытого параллелизма», которого человек, при написании задачи не видит. Такой параллелизм может быть выявлен в процессе решения задачи.

4.Фон-неймановский принцип организации вычислительного процесса, принятый в современных микропроцессорах, который сводится к последовательному выполнению команд, не может эффективно работать с задачами, обладающими высоким параллелизмом.
В настоящее время реализация параллельных вычислений, как правило, осуществляется на СуперЭВМ, состоящих из большого количества микропроцессоров, работающих на общую память или многомашинных комплексов, решающих одну задачу.
Ни тот, ни другой принцип построения вычислительного комплекса не позволяет осуществить его достаточно эффективную загрузку по следующим причинам:
Какой бы ни была архитектура вычислительных систем, программист должен решать две задачи (решить которые ему не под силу):
a) первая задача – распределить вычислительные процессы по процессорам или машинам так, чтобы они были равномерно загружены
b) обеспечить синхронизацию вычислительного процесса так, чтобы данные одного процессора поступали на вход другого в соответствии с алгоритмом, определенным задачей.
Учитывая, что продолжительность каждого вычислительного процесса зависит от величин входных обрабатываемых данных, получаемых в процессе вычислений, синхронизации их, а следовательно и задача эффективного использования вычислительных средств для человека (программиста), становится непосильной. Как правило, с увеличением числа микропроцессоров или машин в комплексе, эффективное их использование падает.
В то же время основным методом повышения производительности вычислительных комплексов в мировой практике считается увеличение числа процессоров или машин в них. Фирмы, разрабатывающие СуперЭВМ (основные – Intel и IBM), чуть ли не раз в три года увеличивают производительность СуперЭВМ на порядок, в то время как реальная производительность комплексов для целого ряда очень важных задач фундаментального плана практически остается неизменной.
Пользователь вводится в заблуждение тем, что производитель ВС, как правило, рекламирует максимальную производительность (Pmax), которая вычисляется, как Pmax = P*N (где N – число процессоров или машин в комплексе, а P – производительность каждого процессора или машины) и умалчивает о реальной производительности P реал, которая, как мы видим, не может быть подсчитана по формуле P*N.
В настоящее время пользователь начинает понимать, что путь увеличения производительности комплекса за счет наращивания числа процессоров может быть использован только для решения специальных задач, а для большинства сложных задач, работающих по принципу фон Неймана, необходимо искать иные принципы построения вычислительных систем. Такие поиски сейчас ведутся целым рядом фирм.
5.В течение 10-ти лет наша группа работала над решением проблемы увеличения производительности одного процессора при решении задач обладающих высоким параллелизмом. Работы в 2000 году закончились построением макета, показывающим возможность создания процессора, производительностью выше 1терафлоп в секунду. В настоящее время в связи с бурным развитием микроэлектроники можно говорить о построении такого микропроцессора на одном или нескольких кристаллах.
Несколькими зарубежными фирмами решен вопрос изготовления ассоциативной памяти объемом в 128 тыс. 72-х разрядных слов (ключей), необходимой для построения процессора на новых принципах.
Из вышесказанного можно сформулировать следующие основные требования к процессору с новыми принципами организации вычислительных процессов:
1. Человек (программист) должен быть исключен из распределения вычислительных процессов по вычислительным ресурсам системы, а также решения задачи синхронизации параллельных процессов по данным.

2. Обычное ОЗУ не фиксирует времени записи, и считывания данных, в результате чего использование его в новых системах затруднено.
3. Предельное распараллеливание вычислительных процессов и синхронизация их по данным может быть выполнена только на аппаратном уровне в динамике вычислений.
4. Единственным принципом, отвечающим этим требованиям организации вычислительных процессов, является принцип data flow.
До настоящего времени этот принцип не использовался по следующим причинам:
□□аппаратная реализация на существующей в то время элементной базе была слишком сложной (не было в первую очередь ассоциативной памяти на кристалле).
□□в чистом виде принцип data flow обладает целым рядом особенностей, которые ограничивают его широкое использование в универсальных вычислительных системах. За последние годы нам удалось преодолеть эти ограничения на аппаратном и программном уровне.
Любой вычислительный процесс может быть представлен графом, узлами которого являются операторы, а по дугам графа перемещаются данные (рисунок 12).

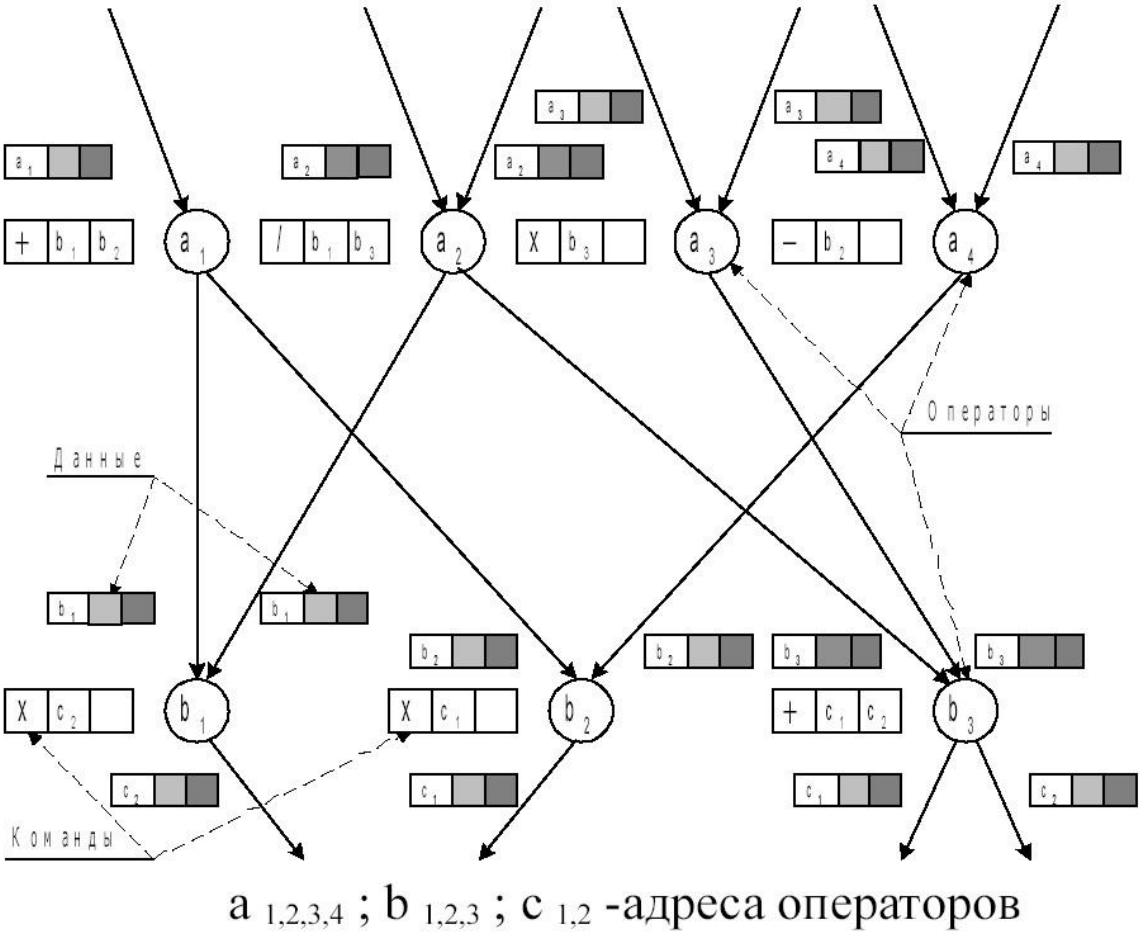


Рисунок 12. Граф вычислительного процесса.

Выполнение такого графа реализует параллельность вычислительных процессов, заданной программой, и исключает конфликтные ситуации по данным, которые возникают из-за того, что, используя обычную оперативную память, мы рискуем при неправильной синхронизации вычислительного процесса (последнее возложено полностью на программиста) вместо новых данных взять из памяти старые данные или наоборот.
Фактически в новой системе концептуально мы не имеем обычной оперативной памяти, работающей с операторами присваивания.
Реализовать такой граф практически невозможно по следующей причине: для того, чтобы такая система вычисляла на протяжении одной секунды со скоростью всего 1 млн.

операций в секунду требуется 1 млн. операторов (исполнительных устройств или микропроцессоров), что реализовать практически невозможно. Однако этот принцип организации вычислений по готовности данных может быть реализован новой архитектурой процессора, представленной на рисунке 13.

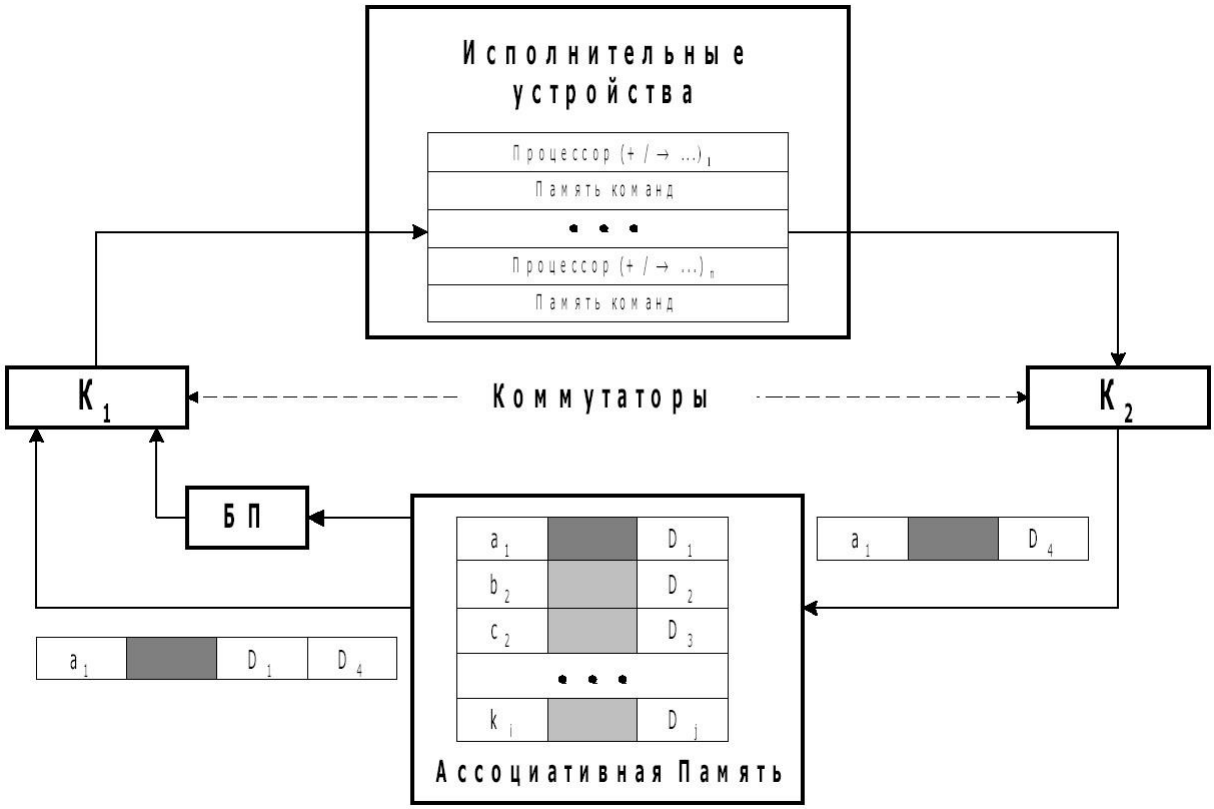


Рисунок 13. Блок-схема высокопроизводительного процессора параллельного счета новой архитектуры.

Работа такой структуры сводится к следующему. Выходящие из исполнительных устройств данные содержат указатели на следующий оператор. Ассоциативная память (АП) объединяет данные, относящиеся к одному и тому же оператору (узлу графа). Если парного данного нет в АП, происходит его запись на свободное место АП. Если парное данное есть, то происходит считывание этого данного со стиранием его в АП и формирование пары данных, направленных к одному и тому же оператору, которая выдается на исполнительное устройство.
В качестве исполнительного устройства может быть микропроцессор с командной и локальной оперативной памятью.
Однако эта упрощенная схема имеет целый ряд проблем при её реализации. Прежде всего, необходимо сделать так, чтобы подпрограммы функций, итераций и циклов не повторялись в командной памяти. С этой целью каждому данному придается «цвет» (код, соответствующий одному или двум данным). Фактически виртуальный адрес в адресном пространстве. В этом случае каждая пара данных имеет свой «цвет», благодаря чему все элементы цикла, итерации и сами функции могут выполняться параллельно, с использованием одних и тех же кодов команд.
Для объединения данных, имеющих единый «цвет» и относящихся к одному и тому же узлу графа, АП должна объединять данные по «цвету» (адресу) и номеру узла (адресу). Если пара по этим признакам найдена, АП выполняет ранее описанный алгоритм. В том случае, если парный операнд не находится в памяти, АП производит запись операнда на любое свободное место. Применение ассоциативной памяти в новой архитектуре способствует решению многих проблем, тормозящих развитие принципов управления по данным в вычислительных средствах:

□□решается, не выходя за рамки принятой концепции, работа с массивами, представляющими общие данные для нескольких параллельных процессов;
□□решается вопрос очищения памяти от засорения данными;
□□достаточно хорошо решается вопрос реализации констант, переменных и др.
Фактически АП берет на себя функции управления вычислительным процессом. Устройство управления, как известно, является самым нерегулярным устройством в любой традиционной архитектуры вычислительных средств. АП, выполняя функцию устройства управления, делает архитектуру вычислительных средств чрезвычайно регулярной структурой, тем самым значительно улучшая технологичность изготовления процессора.
Однако АП является и самым узким местом в части повышения производительности процессора.
Можно сказать, что скорость работы АП обратно пропорциональна её объёму. Поэтому разработан принцип модульного его построения. В этом случае достаточно важным являются два коммутирующих устройства.
Коммутатор К1 распределяет выходящие из модулей АП пары операндов по свободным исполнительным устройствам. Коммутатор К2 распределяет выходящие из исполнительных устройств данные по модулям АП.
Если имеются N исполнительных устройств и N модулей АП, то коммутаторы К1 и К2 являются коммутаторами N x N. В проекте предложены оригинальные меры по сокращению объемов дорогостоящей АП и увеличению скорости её работы в первую очередь её пропускной способности.
Работа всей системы, в отличие от принципа фон-Неймана, основана на конвейерном принципе обработки данных, причем этот принцип распространяется не только на векторные, но и на скалярные операции. Важен тот факт, что операции, выполняемые на этой системе, не взаимосвязаны. Следовательно, загрузка системы зависит только от усредненного параллелизма вычислительного процесса, который, как показала практика, достаточно велик для широкого класса задач и может регулироваться самой системой.
Для сглаживания неравномерности параллелизма выполняемой задачи во времени служит буфер готовых пар (БГП) и сама ассоциативная память (АП).
Производительность конвейера (см. рис.4) целиком зависит от темпа работы каждого его звена (ИУ, К2, АП, К1), пропускная система которых, как правило, делается одной и той же. Таким образом, практически исключается время задержки системы по кольцу ИУ□□К2 □□АП □□К1 при выполнении как векторных, так и скалярных операций при подсчете производительности системы. Пропускная способность (производительность) системы определяется частотой работы устройства в кольце, умноженной на количество колец. Это дает возможность в каждом звене нашей системы (за исключением АП) согласовывать пропускную способность каждого её звена так, чтобы не было узкого горла. Ограничением на количество параллельно работающих устройств в каждом звене системы является только общее требование к среднему параллелизму вычислительных процессов в задаче. Если число колец N умноженное на число ступеней конвейеризации Кк меньше среднего параллелизма Рср задачи, то все исполнительные устройства системы будут полностью загружены.
Теперь рассмотрим ассоциативную память с точки зрения аналогичного решения вопроса пропускной
способности в этом звене системы.
Разделить ассоциативную память на независимые модули невозможно, т.к. каждый ключ может иметь свою пару в любом модуле памяти (запись на свободное место всей памяти). В этом случае каждый ключ должен искаться во всех модулях памяти, что не увеличивает пропускной способности всей памяти.
Единственным способом увеличения пропускной способности путем разбиения памяти на модули является распределение ключей по модулям на основе их уникальных свойств, т.е.

на основании специальной хеш-функции. В этом случае количеством модулей можно было бы регулировать пропускную способность всей ассоциативной памяти. Однако, при одном и том же объеме ассоциативной памяти системы увеличивая количество модулей, мы уменьшаем объем каждого модуля и тем самым повышаем вероятность его переполнения, имея ввиду, что хеш-функция может распределять ключи по модулям неравномерно.
Следовательно, от качества хеш-функции во многом зависит эффективный объем АП. Чтобы не прекращать решение задачи при переполнении модуля памяти, предусмотрен второй уровень специальных модулей памяти, построенных на базе обычных ОЗУ. Откачка данных переполнившегося модуля практически не ухудшает производительности системы. Следующим уровнем откачки данных из АП в случае её переполнения является обычное ОЗУ и память второго уровня.
Уникальные свойства АП и новые принципы построения процессора позволяют обеспечить фактически виртуальное ОЗУ с минимальными потерями производительности. Тот факт, что мы используем принцип записи на свободное место в АП и выбор любого свободного ИУ при загрузке исполнительных устройств мы обеспечиваем высокую надежность системы и снижаем требования к технологии. Например, система будет работоспособной при выходе из строя нескольких исполнительных устройств или ячеек памяти модуля АП (неисправные устройства и ячейки памяти будут считаться занятыми). Коммутаторы реализуются по распределенному принципу по коммутируемым устройствам.
Это свойство системы можно использовать для снижения требований к технологии при повышении требований к увеличению интеграции. Система может использовать чипы ассоциативной памяти и исполнительных устройств с достаточно высоким процентом бракованных логических вентилей.
Структура системы будет иметь регулярный характер т.к. функции устройств управления выполняет АП. Кроме того, АП берет на себя функции синхронизации вычислительных процессов в процессе их выполнения, освобождая от этих функций программиста. Эти свойства АП так же используются при работе в реальном масштабе времени, с целью сокращения времени реакции на внешние воздействия.

3. Особенности рассматриваемой архитектуры

1. На аппаратном уровне реализуется практически полный параллелизм вычислительных процессов выполняемой задачи, что позволяет создать сверхвысокопроизводительный процессор (1011 - 1013 оп/с в одном чипе).
Ассоциативная память экстрагирует все имеющиеся на данный момент времени параллельные процессы и передает их в ИУ на исполнение, одновременно осуществляя синхронизацию по данным.
2. Программист исключен из распределения ресурсов вычислительных средств. Распределение ресурсов осуществляется автоматически в процессе прохождения вычислительного процесса, что обеспечивает высокий коэффициент использования системы.
3. Обеспечивается высокая структурная надежность и технологичность системы. Выход из строя или повреждение нескольких исполнительных устройств или модулей АП не приводит к нарушению функционирования системы.
Самое нерегулярное и сложное по технологии устройство управления процессором заменяется регулярной системой АП, устойчивой к технологическим дефектам.
4. Конвейеризация вычислительного процесса за счет работы ассоциативной памяти позволяет нивелировать временные задержки в логических схемах.
Из АП на исполнительные устройства поступают не связанные между собой пары данных для исполнения операций. Это позволяет строить систему по конвейерному принципу,

исключая временные задержки из вычислительного процесса (скалярные операции выполняются на фоне параллельных вычислений).
5. АП берет на себя функцию адресной арифметики и целый ряд других функций, управленческого характера, существенно упрощая написание программ и повышая производительность процессоров: проведенные эксперименты подтверждают повышение производительности, приведенной к одному микропроцессору и исполнительному устройству от 1.5 до 5 раз, в зависимости от задачи.
6. Процессор, при работе в масштабе реального времени практически не нуждается в традиционном регистре прерываний, реагируя практически мгновенно на сигналы, приходящие в масштабе реального времени. Эта структура не требует затрат процессорного времени на ожидание прецизионного прерывания.
Переход к необходимой программе обработки сигнала, приходящего в масштабе реального времени
происходит через ячейку АП, в которую направляется этот сигнал.
Эти свойства процессора дают возможность создавать телекоммуникационные процессоры, совмещенные с вычислительными системами, в частности сервера или использовать их в технологически сложных системах реального времени.

Моделирование и реализация потоковой машины на вычислительном кластере

Выполнение вычислительного процесса под управлением графа зависимости по данным (DFG) – один из наиболее часто используемых приемов организации высокопараллельных вычислений. DFG состоит из содержащих последовательности операций вершин (вычислительных узлов) и дуг, по которым данные перемещаются между вершинами.

Российская реализация машины, выполняющей представленные в виде DFG программы, - проект ОСВМ. Эта машина включает следующие основные блоки: исполнительные устройства, модули ассоциативной памяти, буфера распределителей заданий для исполнительных устройств.

Исполнительные устройства выполняют операции вычислительных узлов. Модули ассоциативной памяти, позволяют группировать приходящие в виде *токенов* данные, относящиеся к одному узлу. При готовности всех данных узла эти модули формируют задание (готовую *пару*) на выполнение узла. Модули ассоциативной памяти также позволяют хранить данные с ассоциативной адресацией (запоминающие узлы). Буфера распределителей отвечают за доставку данных к исполнительным устройствам, регулируют их загруженность.

Имеется реализованная на персональной ЭВМ программная модель ОСВМ, позволяющая успешно исследовать выполнение задач небольшого размера. Для повышения скорости моделирования ОСВМ, что требуется для проведения полномасштабных исследований на реальных программах с целью отработки архитектуры и программного обеспечения, необходимо использование более высокопроизводительной техники, например, кластера ТКС-7. ТКС-7 – восьмипроцессорная система на базе Pentium III 800 МГц с коммуникационной сетью SCI 2D тор, использующей адаптеры Dolphin D311/312. Этот кластер позволяет очень быстро выполнять пересылку коротких сообщений, что существенно для моделирования передач токенов и готовых пар.

После проведения исследований ТКС и начальных работ по моделированию ОСВМ стало ясно, что наряду с задачей непосредственного моделирования ОСВМ целесообразна постановка новых задач, их общая направленность – модернизация кластеров серии ТКС с использованием элементов архитектуры ОСВМ. Первая задача - эмуляция ОСВМ на ТКС, вторая – применение ассоциативной памяти как элемента коммуникационной среды.

В предлагаемой модели ОСВМ каждому исполнительному устройству, блоку ассоциативной памяти, буферам распределителя соответствует процесс. Передача данных между блоками осуществляется с помощью межпроцессных пересылок функциями MPI. Выполняемая ОСВМ программа хранится в памяти каждого процесса, моделирующего работу исполнительного устройства. Имеется две группы команд. Команды первой группы активизируются данными, приходящими на вход исполнительного устройства через модули ассоциативной памяти. Команды второй группы – это обычные команды фон-Неймановского типа, которые могут выполняться внутри вычислительного узла после его активизации.

На кластере ТКС-7 реализована упрощенная модель ОСВМ, были решены задачи нахождения максимального числа и перемножения матриц. Разрабатывается полная модель ОСВМ, учитывающая временные задержки моделируемых блоков. Поставленные новые задачи введения в ТКС элементов ОСВМ в настоящее время решаются в рамках исследований мультитредовых вычислительных моделей представления программ.

Использование в будущем вместо MPI при моделировании и эмуляции ОСВМ библиотеки функций API SISCI, поддерживающей модель разделяемой памяти (DSM), позволит снизить по крайней мере в 3 раза потери на межпроцессные пересылки, что существенно для повышения рабочих характеристик.

Исследование и разработка программного комплекса системного сопровождения для вычислительной системы с автоматическим распределением ресурсов

Современные высокопроизводительные вычислительные системы – суперЭВМ – играют значительную роль в ключевых областях науки и техники.
Для решения научных и технических задач создаются многопроцессорные суперкомпьютеры с производительностью 1012-1015 операций в секунду, которая достигается благодаря распараллеливанию вычислительных процессов.
Дальнейшее повышение производительности возможно как за счет увеличения количества содержащихся в вычислительной системе процессоров или машин, так и за счет поиска более эффективных методов распараллеливания вычислений.
В настоящее время во всем мире основное внимание уделяется созданию кластерных и массивно-параллельных систем на основе процессоров общего назначения.
Производительность подобных систем сильно зависит от класса решаемых задач. Так, на задачах пакета LINPACK лучшие из современных суперЭВМ показывают максимальную производительность, равную 60-80% от пиковой производительности. Вместе с тем, на многих других задачах их производительность существенно ниже и может составлять 10-20% от пиковой производительности. Основными причинами этого являются сложность балансировки нагрузки на блоки вычислительной системы, высокие накладные расходы на синхронизацию вычислительных процессов и низкая эффективность механизмов кэширования данных при определенных схемах доступа к памяти (например, при нерегулярном доступе к данным или при частом доступе к глобальным данным из параллельно выполняющихся вычислительных процессов).
Чтобы обойти ограничения, присущие традиционным архитектурам высокопроизводительных вычислительных систем, во всем мире делаются попытки исследования и создания новых, нетрадиционных архитектур: асинхронных, реляционных, рекурсивных, функциональных, потоковых и других. Весьма перспективными являются архитектуры, основанные на модели вычислений, управляемых потоком данных, работа над которыми проводится во многих странах мира, в том числе, в США, Японии, Великобритании и России. Одним из важнейших достоинств потоковых архитектур является возможность распараллеливания вычислений и балансировки загрузки процессоров на аппаратном уровне.
В отделе «Проблем построения информационно-вычислительных систем высокого параллелизма» Института проблем информатики Российской академии наук под руководством академика В.С. Бурцева были разработаны новые принципы организации вычислительного процесса и на их основе создана вычислительная система с автоматическим распределением ресурсов (ВСАРР), использующая гибридную модель вычислений, управляемых потоком данных, с динамически формируемым контекстом. Разработан инструментальный испытательный комплекс ВСАРР, включающий в себя имитационную модель системы, блочно-регистровую модель системы и макет системы на ПЛИС. Созданы инструментальные средства, позволяющие вести программирование для ВСАРР с использованием языка ассемблера и языка повышенного уровня. Таким образом, можно говорить о перспективах применения существующих моделей и макетов в качестве «испытательной лаборатории» для исследования различных научно-технических решений, отработки подходов к практическому применению ВСАРР. Эффективность практического применения высокопроизводительного вычислительного комплекса во многом зависит от его операционной системы.
Основными функциями операционной системы являются управление задачами, управление памятью, организация ввода-вывода, поддержка файловой системы. Как правило, при выполнении своих функций операционная система полагается на имеющиеся в ее распоряжении аппаратные средства, так что структура и алгоритмы работы операционной системы в значительной степени зависят от структуры и

алгоритмов работы аппаратных средств. Следует заметить, что модель вычислений, которая положена в основу ВСАРР, и архитектура ВСАРР обладают рядом принципиальных отличий от моделей вычислений и архитектур других вычислительных систем. Поэтому не представляется возможным использовать для организации вычислений в ВСАРР какую-либо из существующих операционных систем (пусть даже с некоторыми доработками).
С другой стороны, некоторые из функций операционной системы являются в значительной степени нейтральными по отношению к модели вычислений и архитектуре ВСАРР. Например, механизмы удаленного доступа к вычислительной системе, реализованные во всех современных операционных системах, лишь в малой своей части привязаны к конкретным аппаратным средствам.
В данной диссертационной работе решается проблема создания программного комплекса системного сопровождения вычислений для вычислительной системы с автоматическим распределением ресурсов.
Разрабатываемый программный комплекс предназначен для организации многозадачного многопользовательского режима работы ВСАРР с возможностью удаленного доступа к ВСАРР. Программный комплекс обеспечивает возможность одновременной работы с ВСАРР большого количества пользователей, которые могут запускать на ВСАРР различные вычислительные задачи. Программный комплекс выполняет те функции операционной системы, которые являются зависимыми от модели вычислений и архитектуры ВСАРР – управление задачами, управление памятью, ввод-вывод исходных данных и результатов вычислений. При этом остальные функции ввода-вывода и поддержка файловой системы возлагаются на обычную операционную систему – такую, как Windows или Linux, что позволяет существенно сократить размер и сроки создания программного комплекса.

Основные принципы модели вычислений, управляемых потоком данных (*dataflow*), были сформулированы в 1974-1975 годах Джеком Деннисом (США). Главная особенность этой модели вычислений заключается в том, что команда программы поступает на исполнение тогда, когда доступны (вычислены) все ее операнды. Для сравнения, традиционная модель вычислений фон-Неймана основывается на том, что команда программы поступает на исполнение тогда, когда на нее указывает счетчик команд.
По сравнению с традиционной моделью вычислений, модель вычислений, управляемых потоком данных, обладает рядом преимуществ. В частности, выявление параллелизма программ на уровне команд существенно упрощает решение проблемы балансировки нагрузки на процессоры вычислительной системы. Вместе с тем, для модели вычислений, управляемых потоком данных, характерен собственный набор проблем, которые необходимо учитывать при разработке системного программного обеспечения.
Существуют различные виды моделей вычислений, управляемых потоком данных: статическая модель, динамическая модель, гибридная динамическая модель. С точки зрения создания вычислительных систем общего назначения, наиболее перспективными являются различные модификации гибридной динамической модели, которая позволяет рассматривать целые участки программы как команды, которые выполняются от начала и до конца на одном процессоре.
Одной из проблем вычислительных машин с архитектурой потока данных является проблема управления *памятью совпадений*, которая используется для определения готовности команд к выполнению и хранения операндов команд, не готовых к выполнению. Переполнение памяти совпадений в таких вычислительных машинах приводит к блокировке (*deadlock*) вычислительного процесса. Основной причиной переполнения памяти совпадений является то, что параллелизм программ во многих случаях оказывается настолько большим, что имеющегося объема памяти совпадений оказывается недостаточно для хранения всех операндов команд, не готовых к

выполнению. В ряде известных проектов по созданию вычислительных машин с архитектурой потока данных (например, *MIT Tagged-Token Dataflow Architecture* и *Manchester Data-Flow Machine*) для преодоления данной проблемы применялось принудительное ограничение параллелизма программы. Чтобы уменьшить параллелизм программы, использовались различные ограничения, накладываемые на программу на этапе ее исполнения. Например, ограничивалось количество одновременно выполняемых процедур программы или количество одновременно выполняемых итераций циклов с целью исключить саму возможность переполнения памяти совпадений. Недостатком такого подхода являлось то, что параллелизм программы на этапе ее исполнения во многом зависит от входных данных и может варьироваться в широких пределах. Таким образом, чтобы исключить возможность переполнения памяти совпадений, приходилось накладывать на программу очень жесткие ограничения, которые существенно снижали эффективность ее исполнения. Другой важной проблемой вычислительных машин с архитектурой потока данных является проблема распределения функций между аппаратурой и системным программным обеспечением. Во многих проектах потоковых вычислительных машин ряд низкоуровневых операций распределения процессоров, памяти и других ресурсов выполнялся программным образом, что приводило к возникновению значительных накладных расходов и существенному снижению производительности вычислительной машины. Производительность можно повысить, выполняя определенные операции распределения ресурсов на аппаратном уровне.

Модель вычислений ВСАРР

ВСАРР основана на гибридной модели вычислений, управляемых потоком данных, с динамически формируемым контекстом. Программу, написанную для выполнения на ВСАРР, можно представить в виде направленного графа потока данных. В узлах этого графа располагаются операторы – программы, предназначенные для обработки операндов, поступающих на входы узлов. Операторы могут быть простыми, то есть, выполнять над поступившими операндами одну операцию, или сложными – выполнять над поступившими операндами несколько операций. Дуги графа отражают зависимость операторов по данным. Таким образом, основной структурной единицей потоковой программы является узел. С каждым узлом связана программа, которая называется программой узла. Выполнение потоковой программы представляет собой серию активаций программ узлов. Активация и выполнение программы узла происходит при поступлении на узел полного комплекта операндов. Как правило, в процессе вычислений программа узла активируется многократно. Активированные программы узлов выполняются параллельно и независимо друг от друга. Программа узла может выполняться одновременно над несколькими разными комплектами операндов.

Основными единицами информации в ВСАРР являются *токен* и *пара*.

Токены являются носителями операндов. В основном, они предназначены для обмена информацией между узлами. Несколько полей токена объединяются в единую структурную единицу, называемую ключом токена. Ключ токена участвует в операции поиска в памяти совпадений для выявления парных токенов. При выявлении совпадающих по ключу токенов происходит их взаимодействие и формируется пара. Каждая пара включает в себя два операнда и адрес программы узла, обрабатывающей эти операнды. Таким образом, пара содержит комплект операндов, необходимый для активации программы узла.

Принципиальным отличием ВСАРР от других систем, основанных на гибридной модели вычислений, управляемых потоком данных, является возможность *динамического формирования контекста вычислений* программистом. Программа для ВСАРР может самостоятельно формировать содержимое поля контекста токенов в соответствии с алгоритмом своей работы, что устраняет накладные расходы на выделение и освобождение значений контекста и в некоторых случаях дает возможность использовать поле контекста в качестве носителя дополнительного операнда.

Проблема переполнения памяти совпадений в ВСАРР решается за счет введения *виртуальной памяти*. Функционирование виртуальной памяти основано на том, что все множество токенов задачи может быть разбито программистом на набор независимых подмножеств – подзадач, каждое из которых гарантированно умещается в доступный объем памяти совпадений. Если в процессе вычислений возникает угроза переполнения памяти совпадений, некоторые подзадачи могут быть откачаны во внешнюю память. В дальнейшем, когда память совпадений освободится, эти подзадачи могут быть подкачаны обратно. Такой подход позволяет более эффективно использовать имеющийся ресурс памяти совпадений, что, в конечном счете, позволяет достичь более высокой производительности вычислительной системы в целом.

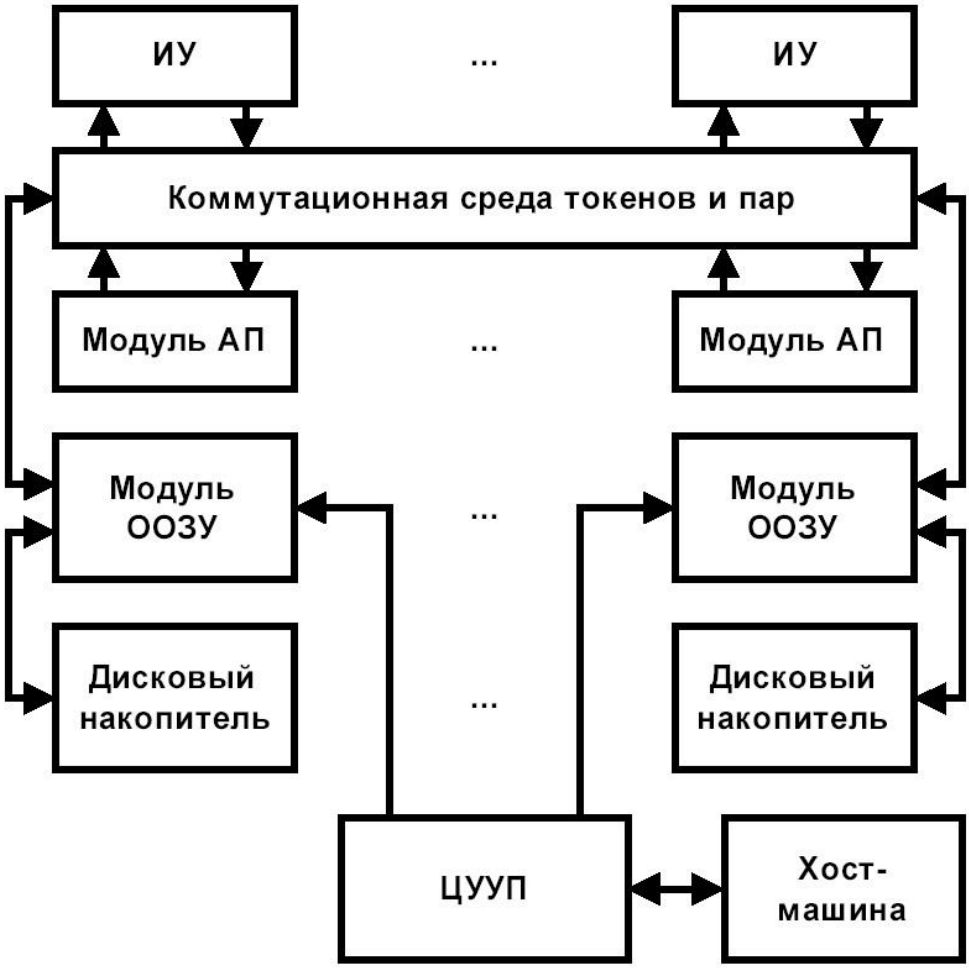


Рисунок 14. Структурная схема ВСАРР

Исполнительные устройства (ИУ) обрабатывают поступающие из коммутационной среды пары, выполняя соответствующие программы узлов. Как правило, результатом выполнения программы узла является один или несколько токенов, которые программа узла генерирует при помощи специальных команд. Через коммутационную среду эти токены направляются на модули ассоциативной памяти.

В качестве памяти совпадений используются модули *ассоциативной памяти* (АП), которые осуществляют хранение токенов и сопоставление хранимых токенов с токенами, поступающими из коммутационной среды.

Результатом операции сопоставления может являться генерация одной или нескольких пар, которые через коммутационную среду направляются на исполнительные устройства.

Коммутационная среда осуществляет доставку пар к исполнительным устройствам и доставку токенов к модулям ассоциативной памяти.

Общее оперативное запоминающее устройство (ООЗУ) и диски в основном предназначены для хранения токенов и пар, откачанных из модулей ассоциативной памяти и внутренних буферов коммутационной среды.
Центральное устройство управления памятью (ЦУУП) поддерживает набор команд управления задачами (загрузка, запуск, останов, сбор статистической информации и другие), а также поддерживает механизмы функционирования виртуальной памяти.
Х*ост-машина* представляет собой обычный персональный компьютер или сервер. Она предназначена для выполнения интерфейсных функций – организации взаимодействия между ВСАРР и другими вычислительными системами, а также, может быть использована для выполнения компонентов системного программного обеспечения ВСАРР.
Принципиальной особенностью ВСАРР является то, что большинство низкоуровневых операций по выделению и освобождению ресурсов выполняется на аппаратном уровне. Так, пары, формируемые в модулях АП, поступают на те ИУ, которые являются наименее занятыми. Токены, генерируемые в ИУ, распределяются между модулями АП в соответствии со значением *хэш-функции* от определенных полей токена. Токен, поступивший в модуль АП, записывается в любую свободную ячейку памяти. Стирание токена из модуля АП заключается в сбросе бита присутствия токена в памяти. Такой подход позволяет практически полностью устранить накладные расходы на распределение ресурсов вычислительной системы между задачами и между различными «ветвями» вычислительного процесса внутри каждой отдельно взятой задачи.

Программный комплекс системного сопровождения вычислений для ВСАРР

Системное программное обеспечение ВСАРР должно поддерживать многозадачный многопользовательский режим работы вычислительной системы, удаленный доступ к вычислительной системе, механизмы интеграции вычислительной системы в традиционную вычислительную среду и так далее.
Некоторые из этих требований являются нейтральными по отношению к модели вычислений и архитектуре ВСАРР, что позволяет значительно сократить объем работ по созданию системного программного обеспечения ВСАРР, переложив выполнение некоторых функций на операционную систему хост-машины. Например, операционная система хост-машины может поддерживать различные файловые системы, механизмы предотвращения несанкционированного доступа к данным, сетевые протоколы для обмена данными с персональными компьютерами пользователей и многое другое.
Фактически, чтобы продуктивно использовать ВСАРР в составе традиционной вычислительной среды, достаточно разработать набор компонентов системного ПО, позволяющий совместно с операционной системой хост-машины обеспечить выполнение вышеперечисленных требований.
Компоненты системного программного обеспечения ВСАРР могут выполняться непосредственно на потоковой вычислительной машине, на хост-машине и на персональных компьютерах пользователей. Непосредственно на потоковой вычислительной машине обязана выполняться только часть системного ПО – обработчики нештатных ситуаций и различные библиотеки. Остальные компоненты системного ПО могут выполняться на хост-машине; некоторые компоненты системного ПО (например, командные оболочки, различные утилиты) могут выполняться на персональных компьютерах пользователей. Это дает возможность использовать для создания большой части системного ПО стандартные инструменты (языки программирования и среды разработки), так как и хост-машина, и персональные компьютеры пользователей работают под управлением стандартных операционных систем.

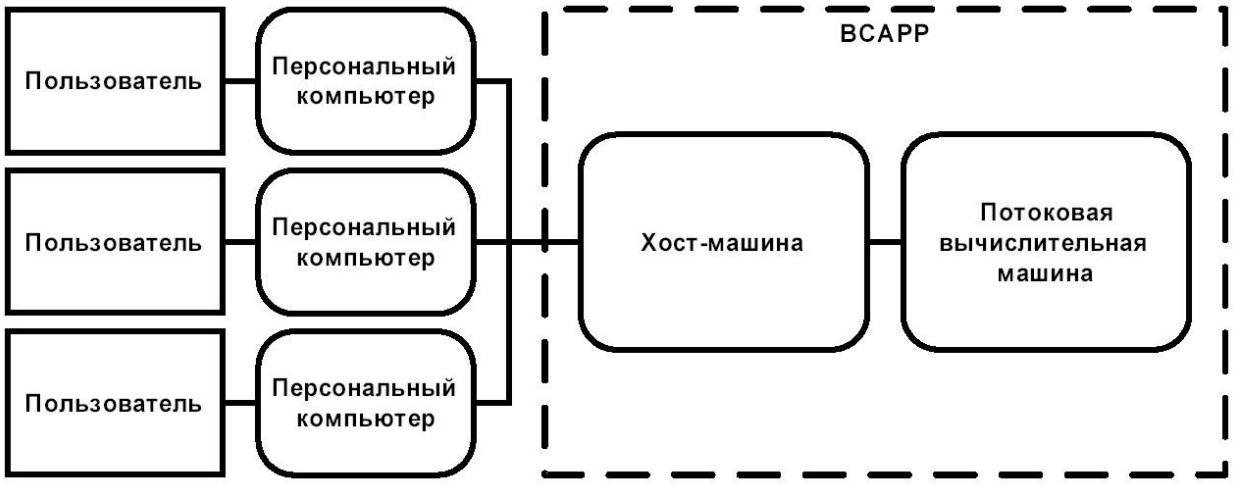


Рисунок 15. Схема доступа пользователей к ВСАРР

Предполагается, что основной целью работы пользователей с ВСАРР является запуск прикладных потоковых программ, позволяющих за сравнительно небольшое время выполнять необходимые пользователям вычисления. Таким образом, весь набор компонентов системного ПО, необходимый для продуктивного использования ВСАРР в составе традиционной вычислительной среды, можно охарактеризовать как *программный комплекс системного сопровождения вычислений для ВСАРР*.

Данный программный комплекс должен предоставлять пользователям следующие основные возможности:

· Доступ к ВСАРР с помощью локальной или глобальной сети со своего персонального компьютера.

· Хранение файлов с потоковыми программами и файлов с исходными данными и результатами вычислений в персональном рабочем пространстве и обмен информацией с другими пользователями.

· Запуск потоковых программ на ВСАРР с определенными исходными данными и получение результатов вычислений.

Программный комплекс должен включать в себя три части:

· Резидентная часть – выполняется на потоковой вычислительной машине, включает в себя обработчики нештатных ситуаций и различные библиотеки.

· Операционная часть – выполняется на хост-машине, осуществляет обработку различных видов пользовательских запросов и контролирует использование ресурсов хост-машины и потоковой вычислительной машины.

· Клиентская часть – выполняется на персональном компьютере пользователя, организует взаимодействие между пользователем и операционной частью программного комплекса.

Потоковая программа в стадии выполнения представляет собой иерархически организованную группу задач, выполняющихся в общем окружении. Иерархия задач представляет собой дерево, построенное на основе отношения «родитель-ребенок». Корнем дерева является так называемая главная задача, которая запускается при вводе пользователем задания на выполнение потоковой программы. Главная задача может запускать дочерние задачи, те могут запускать свои дочерние задачи и так далее. Под окружением выполняющейся потоковой программы понимается совокупность различных системных параметров и таблиц. Поскольку каждая потоковая программа выполняется в собственном окружении, исключается возможность нарушения работы одной программы другой программой.

Управление задачами и памятью в ВСАРР

Работая в многозадачном многопользовательском режиме, программный комплекс системного сопровождения вычислений для ВСАРР должен осуществлять планирование

задач, то есть, предоставлять задачам доступ к вычислительным ресурсам ВСАРР в соответствии с определенными правилами. Существует большое количество различных стратегий планирования, среди которых наиболее общей и универсальной является стратегия планирования в соответствии с приоритетами. Приоритет – некоторое числовое значение, которое пользователь, программист, администратор и системное программное обеспечение могут присваивать задачам. К алгоритмам планирования задач в соответствии с приоритетами предъявляются следующие основные требования:
Распределение ресурсов в соответствии с приоритетами – чем выше приоритет задачи, тем больший объем вычислительных ресурсов системы должен поступать в распоряжение задачи.
Предсказуемость – задачи должны регулярно получать доступ к вычислительным ресурсам системы.
Балансировка нагрузки – вычислительные ресурсы системы должны быть полностью загружены обработкой информации (если это возможно).
Важность выполнения тех или иных требований определяется типом вычислительной системы. Например, для систем реального времени и интерактивных систем чрезвычайно важным является требование предсказуемости, а для систем пакетной обработки – требование балансировки нагрузки.
Планирование задач в ВСАРР связано со следующими проблемами:
· Необходимость четко формулировать, какие виды ресурсов системы считаются *вычислительными* и должны предоставляться задачам в соответствии с выбранным алгоритмом планирования.
· Жесткие требования к скорости работы алгоритма планирования задач, возникающие ввиду ограниченного объема ассоциативной памяти.
Важно отметить, что принципиальной особенностью ВСАРР является то, что задача может выполняться только в том случае, когда ей выделен определенный объем ассоциативной памяти; если задача откачана во внешнюю память, ее выполнение приостанавливается. Таким образом, организация виртуальной памяти ВСАРР и планирование задач в ВСАРР тесно связаны между собой. Организация виртуальной памяти ВСАРР требует решения следующих проблем:
· Проблема распределения внешней памяти между одновременно выполняющимися задачами.
· Проблема выбора алгоритма откачки данных из ассоциативной памяти во внешнюю память и подкачки данных из внешней памяти в ассоциативную память.
В первую очередь, при решении этих проблем следует преследовать цель минимизировать простои вычислительной системы из-за ожидания завершения операции откачки или подкачки. Во вторую очередь, необходимо стремиться сократить объем различных накладных расходов, связанных с управлением памятью. Для этого необходимо искать оптимальное сочетание программных и аппаратных механизмов управления памятью.

Планирование задач в ВСАРР

Характерной особенностью ВСАРР является то, что процесс вычислений происходит как в исполнительных устройствах, в которых происходит обработка пар и генерация токенов, так и в модулях ассоциативной памяти, в которых происходит сопоставление токенов и генерация пар. Более того, в общем случае нельзя сказать, кто вносит основной вклад в получение конечного результата вычислений – ИУ или модули АП, поскольку в данном вопросе все зависит от алгоритма решаемой задачи и конкретных исходных данных. Таким образом, ИУ и модули АП являются *вычислительными* ресурсами ВСАРР, что должен учитывать алгоритм планирования задач. Поскольку распределение доступного физического объема АП между задачами напрямую влияет на скорость выполнения задач, в процессе работы машины необходимо постоянно отслеживать загрузку АП токенами различных задач и перераспределять физический объем АП между задачами. Например,

если низкоприоритетная задача захватила весь доступный физический объем АП, может возникнуть необходимость временно откачать ее внешнюю память, а впоследствии подкачать ее обратно.
Как показывают расчеты, частота принятия решений об откачке и подкачке задач может быть достаточно большой – порядка 106-107 раз в секунду, причиной чего является ограниченный объем АП. Действительно, чтобы обеспечить максимальную интенсивность генерации пар в модулях АП, необходимо поддерживать их в заполненном состоянии. Если заполненность модуля падает ниже определенного порога (допустим, 80%), должна начинаться подкачка токенов в модуль. Вместе с тем, если заполненность модуля поднимается выше определенного порога (допустим, 90%), необходимо начинать откачку токенов из модуля, чтобы избежать блокировки вычислительного процесса в ВСАРР. Таким образом, частота принятия решений об откачке и подкачке определяется той скоростью, с которой меняется заполненность модуля. Данная скорость зависит как от постоянных факторов – то есть, объема модуля и темпа, с которым модуль принимает токены и генерирует пары, так и от переменных факторов – то есть, конкретных задач, которые выполняются в вычислительной системе на определенных наборах входных данных. Поскольку эффективная работа вычислительной системы должна обеспечиваться во всех ситуациях, целесообразно принятие решений об откачке и подкачке задач выполнять аппаратным образом. Эту функцию должно выполнять центральное устройство управления памятью ВСАРР. Системное программное обеспечение ВСАРР должно загружать в ЦУУП значения приоритетов задач, необходимые для принятия решений об откачке и подкачке.
Распределение ресурсов в соответствии с приоритетами в большинстве систем выполняется по следующему принципу: управление передается готовой к выполнению задаче с наивысшим приоритетом. Однако в этом случае необходимо предусматривать механизм предотвращения безостановочной работы такой задачи – например, уменьшать приоритет задачи с течением времени или ограничивать максимальный отрезок времени непрерывной работы задачи. Принципиально другим подходом является так называемое *гарантированное планирование*. Оно основывается на принципе передачи управления той задаче, которая за время своего предыдущего выполнения израсходовала меньше всего ресурсов из того лимита, который должен быть ей предоставлен в соответствии с ее приоритетом. Математически этот принцип может быть сформулирован следующим образом. Пусть в системе выполняется N задач. Обозначим через Pi абсолютный приоритет i-й задачи (присвоенный, допустим, администратором системы), через Qi объем ресурсов, израсходованных i-й задачей за определенный промежуток времени. Qi представляет собой некую интегральную величину, характеризующую загруженность ИУ и модулей АП обработкой токенов и пар i-й задачи, которая рассчитывается на основе значений аппаратных счетчиков активности задач.
Относительный приоритет i-й задачи равен отношению Pi к сумме абсолютных приоритетов всех N задач. Относительный объем израсходованных i-й задачей ресурсов равен отношению Qi к общему объему ресурсов, израсходованных всеми задачами. При постановке задач на выполнение предпочтение должно отдаваться тем задачам, у которых разность между относительным приоритетом и относительным объемом израсходованных ресурсов является наибольшей. Аппаратура ВСАРР (в частности, ЦУУП) может иметь различные встроенные средства, обеспечивающие распределение ресурсов между задачами в соответствии с их приоритетами. Вместе с тем, в силу ограничений реализации, аппаратные механизмы не всегда способны обеспечить достаточный уровень точности данного распределения. Эта проблема может решаться двумя способами.
Первый способ основан на использовании *динамических* приоритетов задач. Динамические приоритеты задач – те приоритеты, которые системное

программное обеспечение ВСАРР загружает в ЦУУП и на основании которых ЦУУП принимает решения об откачке и подкачке. Системное ПО должно отслеживать прохождение задач на вычислительной системе и корректировать динамические приоритеты задач, у которых относительный приоритет не совпадает с относительным объемом израсходованных ресурсов. Для этого можно использовать адаптивный алгоритм: если относительный приоритет задачи приблизительно совпадает с относительным объемом израсходованных ею за определенный промежуток времени ресурсов, то динамический приоритет задачи необходимо оставить неизменным; в противном случае, динамический приоритет задачи необходимо увеличить или уменьшить на определенную величину.

Второй способ основан на использовании *пакетов* задач. Системное программное обеспечение ВСАРР должно периодически набирать из всех готовых к выполнению задач определенный пакет и ставить его на выполнение. Выбирая задачи для очередного пакета, необходимо учитывать приоритеты задач и объемы израсходованных задачами ресурсов. Такой подход аналогичен подходу, используемому в традиционных вычислительных системах, и не предъявляет сколько-нибудь серьезных требований к реализации ЦУУП. Однако такой подход порождает дополнительные накладные расходы, связанные с необходимостью периодически снимать с выполнения предыдущий пакет задач и ставить на выполнение новый пакет. Кроме того, неудачно выбранный пакет может загрузить вычислительную систему не полностью.

Предсказуемость планирования для ВСАРР может быть обеспечена только непрерывной ротацией задач (так же, как и для традиционных вычислительных систем). При использовании механизма динамических приоритетов задач, ротацию должно осуществлять центральное устройство управления памятью. Основываясь на значениях динамических приоритетов задач, ЦУУП должно периодически снимать с выполнения одни задачи и ставить на выполнение другие задачи, стремясь обеспечить как распределение ресурсов в соответствии с приоритетами, так и полную загрузку вычислительной системы. При использовании механизма пакетов задач, ротацию осуществляет системное ПО, выбирая задачи для включения в очередной пакет.

Балансировка нагрузки для ВСАРР сводится к тому, что системное ПО должно постоянно отслеживать загрузку ИУ и модулей АП различными задачами. Для любой конфигурации ВСАРР могут быть подобраны так называемые несбалансированные задачи, которые нагружают ВСАРР неравномерно (недогружают ИУ при полной загрузке модулей АП, недогружают модули АП при полной загрузке ИУ, и так далее). Системное ПО может обнаруживать и соответствующим образом исправлять такие ситуации.

Организация виртуальной памяти в ВСАРР

Виртуальная память ВСАРР предназначена для решения проблемы блокировки вычислительного процесса при переполнении ассоциативной памяти. Переполнение АП может возникать как в многозадачном режиме (если доступного объема АП недостаточно для одновременного прохождения всех выполняющихся задач), так и в монозадачном режиме (если доступного объема АП недостаточно для одновременного прохождения всех подзадач выполняющейся задачи). В случае угрозы переполнения АП часть токенов откачивается из АП в ООЗУ, что предотвращает блокировку вычислительного процесса. Поскольку объем ООЗУ также является ограниченным, в ВСАРР предусмотрена возможность откачки данных из ООЗУ в дисковую память при переполнении ООЗУ. Обмены между АП и ООЗУ целесообразно проводить под контролем ЦУУП ввиду высокой частоты этих обменов. Алгоритм откачки и подкачки, заложенный в ЦУУП,

может быть как достаточно простым, так и весьма сложным. Например, возможен такой простой алгоритм: при угрозе переполнения АП откачивать задачу с наименьшим приоритетом, при освобождении АП подкачивать задачу с наибольшим приоритетом. Такой алгоритм несложно реализовать в аппаратуре, однако ротацию задач в этом случае придется выполнять программным образом. Заложенный в ЦУУП более сложный алгоритм откачки и подкачки, поддерживающий ротацию задач на основе аппаратных счетчиков активности задач, может оказаться более эффективным с точки зрения использования ресурсов вычислительной системы.

Распределение ООЗУ между задачами и между подзадачами каждой отдельно взятой задачи может быть реализовано при помощи общепринятых механизмов страничной адресации памяти. ЦУУП поддерживает так называемую *аппаратную таблицу задач* (АТЗ), в которой хранится информация о задачах, выполняющихся в потоковой вычислительной машине. АТЗ позволяет сопоставить каждой задаче несколько сегментов в ООЗУ, предназначенных для откачки токенов задачи. Системное программное обеспечение ВСАРР должно загружать в АТЗ начальные виртуальные адреса этих сегментов. Если виртуальное адресное пространство ООЗУ достаточно велико, то сегменты всех задач могут быть расположены в одном виртуальном адресном пространстве, а для преобразования виртуальных адресов в физические адреса потребуется одна таблица страниц. В противном случае, необходимо предоставлять каждой задаче свое виртуальное адресное пространство и свою таблицу страничного преобразования адресов. В любом случае, системное ПО должно осуществлять заполнение таблиц страниц и гарантировать, что таблица страниц каждой задачи содержит достаточное количество заполненных элементов (то есть, отслеживать количество страниц, используемых задачей, и заблаговременно выделять задаче дополнительные страницы).

Откачка данных из ООЗУ на диск может производиться под контролем системного ПО ввиду того, что решения об откачке на диск можно принимать достаточно редко. Для этого необходимо, чтобы системное ПО отслеживало заполненность ООЗУ и откачивало целую группу страниц на диск тогда, когда заполненность ООЗУ превышает определенный порог. Дополнительным преимуществом такого подхода является то, что в ООЗУ всегда будет существовать определенное количество свободных страниц. Таким образом, подкачку данных с диска в ООЗУ в тех ситуациях, когда ЦУУП пытается обратиться к странице, откачанной на диск, можно производить без участия системного ПО, что позволяет избежать дополнительных накладных расходов. Основной проблемой, связанной с откачкой страниц на диск, является проблема выбора *алгоритма откачки страниц*. Оптимальный алгоритм откачивает в первую очередь те страницы, которые максимально долго не будут использоваться в будущем (в общем случае, оптимальный алгоритм не может быть практически реализован из-за недоступности информации о том, как в будущем будет протекать процесс вычислений). Алгоритмы откачки, применяемые на практике, обычно основываются на том, что аппаратура выставляет для каждой страницы биты «R» (*referenced*) и «M» (*modified*). Бит «R» устанавливается при любом обращении к странице (по чтению или по записи), бит «M» – при записи в страницу. Этой информации достаточно для того, чтобы реализовать алгоритм, достаточно хорошо аппроксимирующий
оптимальный алгоритм. Например, алгоритм «старения» страниц основан на периодическом опросе состояния битов «R» всех страниц с целью отследить те страницы, которые не использовались дольше всего. *Алгоритм подкачки страниц* в случае недоступности информации о том, как в будущем будет протекать процесс вычислений, сводится к подкачке страниц при обращении к

ним. Если приблизительно известен момент времени в будущем, когда произойдет обращение к странице, то такая страница может быть подкачана заблаговременно.
Применительно к ВСАРР, стандартные алгоритмы откачки и подкачки могут быть определенным образом модифицированы, что повысит их эффективность. Во-первых, при откачке данных из АП в ООЗУ и при подкачке данных из ООЗУ в АП доступ к сегментам памяти ООЗУ, как правило, происходит по последовательным виртуальным адресам. С учетом этого обстоятельства, возникает возможность оптимизировать обмены между ООЗУ и дисками. Например, становится возможной опережающая подкачка страниц с диска в тех случаях, когда необходимо определенный сегмент, находящийся на диске, переместить в АП. Также, при откачке страниц на диск можно откачивать в первую очередь те страницы, которые находятся в «хвостах» сегментов (то есть, гарантированно будут востребованы позже всего). Во-вторых, задачи, обладающие высоким приоритетом, обычно должны откачиваться на диск лишь в редких случаях (например, если задача заблокирована ввиду ожидания внешнего события). Поэтому при выборе страниц, откачиваемых на диск, необходимо совместно анализировать временные метки последнего доступа к страницам и приоритеты задач, которым принадлежат страницы. Например, страницы высокоприоритетных задач могут быть «защищены» от откачки в течение определенного периода времени.
Таким образом, системное ПО выделяет задачам страницы в ООЗУ и откачивает страницы из ООЗУ на диск. ЦУУП осуществляет подкачку страниц с диска в ООЗУ. Чтобы обеспечить корректную совместную работу системного ПО и ЦУУП, необходимо предусмотреть *таблицу свободных страниц ООЗУ*, хранящую физические адреса всех свободных страниц ООЗУ. Системное ПО должно забирать адреса из этой таблицы при выделении задачам новых страниц в ООЗУ и помещать адреса в эту таблицу при откачке страниц из ООЗУ на диск. ЦУУП должно забирать адреса из этой таблицы при подкачке страниц с диска в ООЗУ. Аналогично, необходимо предусмотреть *таблицу свободных страниц на диске*, хранящую свободные адреса страниц на диске. Системное ПО должно забирать адреса из этой таблицы при откачке страниц из ООЗУ на диск, а ЦУУП должно помещать адреса в эту таблицу при подкачке страниц с диска в ООЗУ. Разумеется, все три таблицы – страничного преобразования, свободных страниц в ООЗУ и свободных страниц на диске – должны изменяться согласованно, под контролем ЦУУП.

Структура программного комплекса системного сопровождения вычислений для ВСАРР.

Чтобы обеспечить гибкость и расширяемость программного комплекса, при его разработке использовался компонентный подход. Необходимо было четко разделить функциональные обязанности между компонентами и минимизировать межкомпонентные взаимодействия. На основе анализа требований к программному комплексу, были выделены следующие основные компоненты.
Клиент – компонента, выполняющаяся на персональном компьютере пользователя, предоставляющая программный интерфейс доступа к ВСАРР.
Посредством этого интерфейса любые программы, выполняющиеся на персональном компьютере пользователя, могут взаимодействовать с основной частью программного комплекса, которая выполняется на хост-машине ВСАРР.
Взаимодействие с программным комплексом значительно упрощается, поскольку клиент берет на себя такие функции, как организация удаленного доступа к ВСАРР, буферизация обмена данными с ВСАРР и так далее.

Административная подсистема – компонента, которая выполняет идентификацию и аутентификацию пользователей, осуществляющих доступ к ВСАРР. Для этого используются встроенные механизмы операционной системы хост-машины ВСАРР. Когда пользователь пытается подключиться к ВСАРР, административная подсистема определяет, какими правами доступа обладает данный пользователь, и сообщает эту информацию остальным компонентам программного комплекса.
Подсистема регистрации потоковых программ – компонента, поддерживающая совместное использование файлов различного типа (в первую очередь, совместное использование прикладных потоковых программ). Эта компонента позволяет пользователям помещать на хост-машину различные файлы, делая их доступными для считывания другими пользователями. Например, пользователь может зарегистрировать свою потоковую программу под определенным уникальным именем, используя которое, другие пользователи смогут запускать эту программу.
База данных – компонента, осуществляющая хранение различной информации – списка пользователей ВСАРР, списка зарегистрированных потоковых программ, различных административных настроек и так далее.
Процесс-представитель – компонента, которая осуществляет поддержку выполнения потоковых программ и поддерживает обмен данными между потоковыми программами, выполняющимися на потоковой вычислительной машине, и программами, выполняющимися на персональном компьютере пользователя. Процесс-представитель поддерживает окружение потоковой программы – создает начальное окружение и соответствующим образом модифицирует окружение, когда потоковая программа выполняет определенные системные вызовы. Например, процесс-представитель хранит список дескрипторов открытых потоковой программой файлов. Когда потоковая программа выполняет системный вызов доступа к файлу с определенным дескриптором, процесс-представитель непосредственно выполняет запрошенную операцию. Также, процесс-представитель поддерживает преобразование данных из вида, специфичного для потоковой программы (токены определенного формата) в универсальный формат (массивы, списки) и обратно (правила преобразования описываются программистом в самой потоковой программе). Благодаря этому, потоковая программа и программа, выполняющаяся на персональном компьютере пользователя, могут обмениваться данными в удобном для себя формате. *Расширение процесса-представителя* – компонента, подключаемая к процессу-представителю с целью расширения его возможностей. Может поддерживать нестандартные форматы обмена данными, нестандартные системные вызовы и так далее.
Супервизор – ключевая компонента программного комплекса, непосредственно отвечающая за прохождение задач на потоковой вычислительной машине. Основными функциями супервизора являются:
· Взаимодействие с потоковой вычислительной машиной – прием и выдача данных и различной управляющей информации.
· Управление задачами, выполняющимися на потоковой вычислительной машине – в том числе, планирование задач.
· Управление памятью потоковой вычислительной машины – в том числе, совместная с ЦУУП организация виртуальной памяти.
В целом, функционирование программного комплекса выглядит следующим образом. Пользователь выполняет свою работу при помощи определенной программы, запущенной на его персональном компьютере. В определенный момент времени, эта программа обращается к клиенту с запросом на запуск потоковой программы. На основании этого запроса, клиент формирует заявку на выполнение потоковой программы и передает ее процессу-представителю. Процесс-представитель создает начальное окружение потоковой программы, генерирует образ потоковой программы и передает его супервизору. Супервизор загружает полученный образ в потоковую вычислительную машину. В

процессе своего выполнения, потоковая программа может пересылать токены на хост-машину. Супервизор передает эти токены процессу-представителю, который определяет тип полученного токена. Токены системных вызовов собираются в специальном буфере и затем обрабатываются. Токены-данные преобразуются в универсальный формат представления данных и отсылаются клиенту.

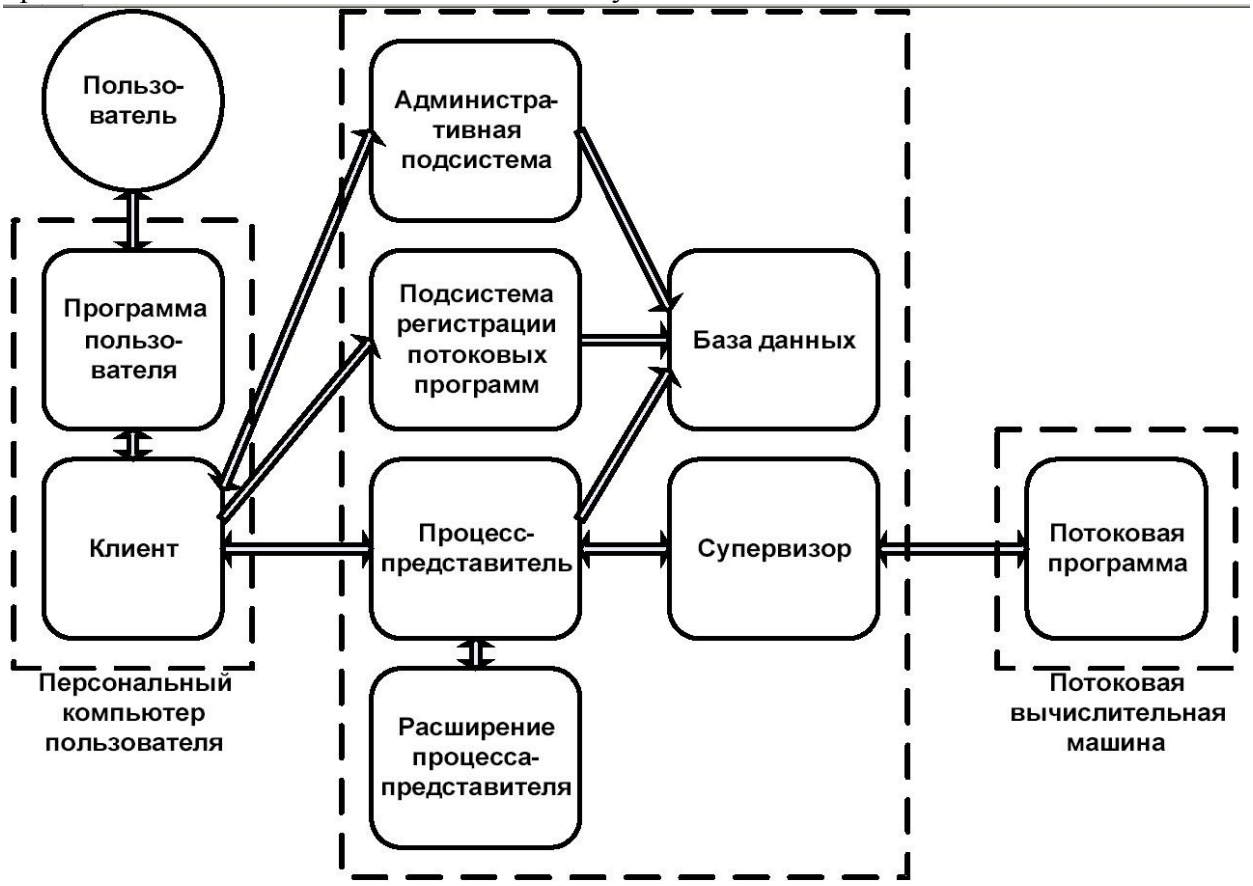


Рисунок 16. Структура программного комплекса

При разработке структуры программного комплекса предполагалось, что основным способом программного доступа к ВСАРР будет доступ через интерфейс клиента, который, как позднее было уточнено, является COM-интерфейсом. Такой интерфейс вполне удовлетворяет требованиям универсальности и надстраиваемости, так как эти требования изначально заложены в COM-технологию. Вместе с тем, COM-интерфейсы не всегда отвечают требованию удобства использования, поскольку не все языки и среды программирования имеют развитую поддержку технологии COM. Поэтому для определенных языков целесообразно было разработать обертки над интерфейсом клиента. Например, такая обертка была разработана для языка Фортран. Эта обертка предоставляет возможность просто и компактно записать процедуру взаимодействия с ВСАРР с использованием языка Фортран.

Аппаратно-программные средства преодоления неравномерной загрузки отдельных устройств в параллельной потоковой вычислительной системе

Работа над созданием потоковой параллельной вычислительной системы (ППВС) ведется на протяжении ряда лет, основные идеи и принципы организации параллельных вычислительных процессов и архитектура вычислительной системы, в которой реализуются эти принципы, были разработаны еще под руководством академика В.С. Бурцева. На рисунке 1 приведена иерархическая система памяти ППВС.

Очевидно, что неравномерность загрузки отдельных устройств вычислительной системы снижает эффективность ее использования. В связи с этим рассмотрим основные виды неравномерностей загрузки отдельных устройств в ППВС, а также способы их преодоления.

Поскольку основными единицами информации в ППВС являются токен и пара, то с этой точки зрения и рассмотрим, как они могут участвовать в создании неравномерности загрузки.

Неравномерность загрузки отдельных устройств системы, связанная с обработкой пар.

Пары в ППВС образуются в модулях ассоциативной памяти (МАП). Затем по цепочке МАП — буфер готовых пар (БГП) — коммутатор пар (КМ пар) — исполнительное устройство (ИУ), поступают в свободные модули ИУ.

На данном этапе возможны следующие неравномерности вычислительного процесса:

- П1) неравномерность, обусловленная программой узла, и, как следствие, различным временем обработки пар в исполнительных устройствах;
- П2) неравномерность, обусловленная взрывным характером параллелизма задачи, проявляющаяся как переполнение буферов готовых пар (БГП);
- П3) неравномерность, обусловленная различным темпом формирования пар в каждом из МАП. На данный фактор оказывают влияние хэш-функция, кратность и маскирование;
- П4) неравномерность, обусловленная недостаточным уровнем параллелизма задачи, что приводит к недозагрузке всей системы, и отдельных ИУ в частности.

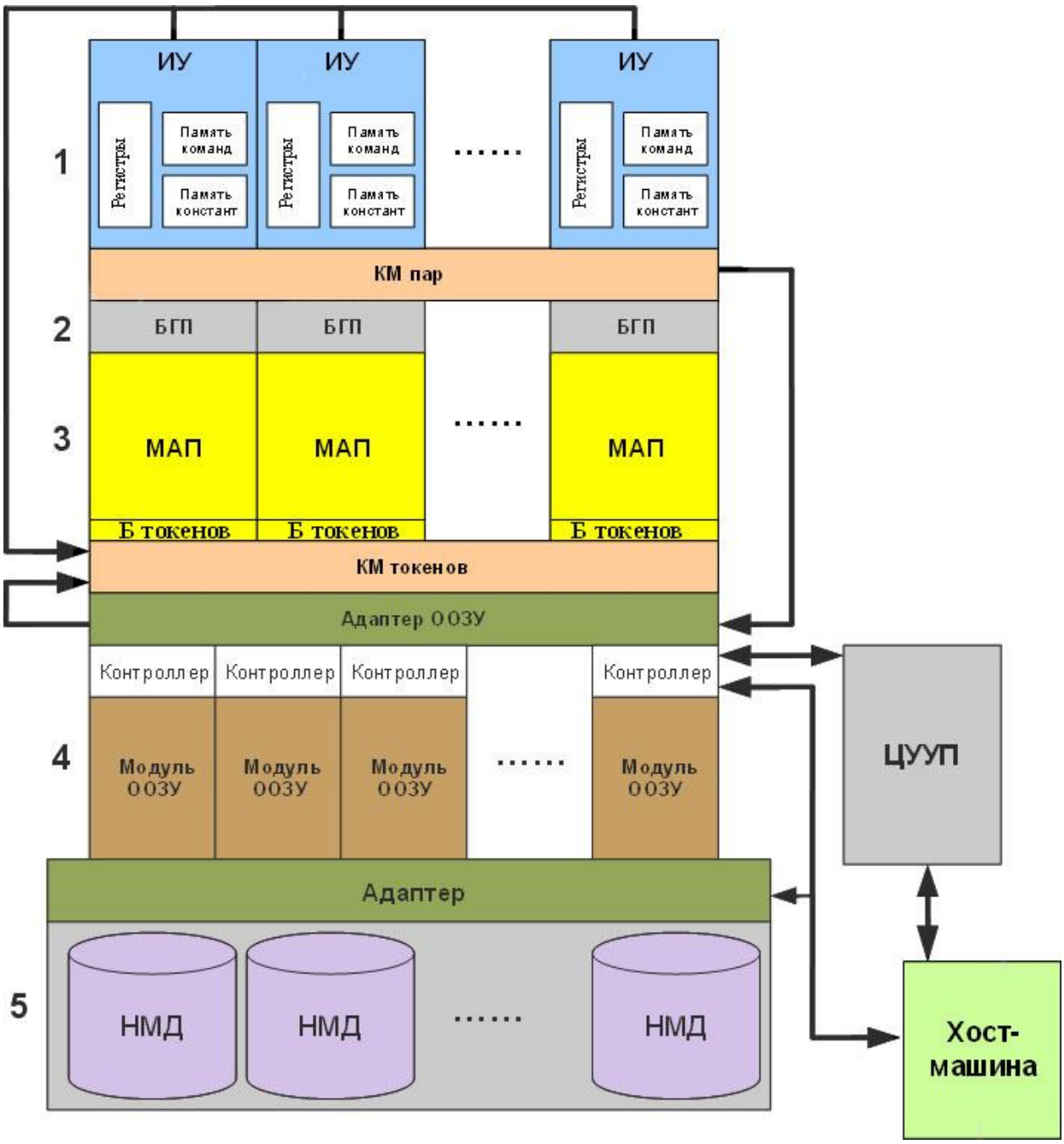


Рисунок 17. Иерархическая система памяти ППВС

Неравномерность загрузки отдельных устройств системы, связанная с обработкой токенов.
Токены в ППВС образуются в исполнительных устройствах, а также поступают извне — с ХОСТ-машины. Затем по цепочке ИУ — коммутатор токенов (КМ токенов) — буфер токенов — МАП, поступают в модули ассоциативной памяти, адреса которых формируются согласно ХЭШ-функции.

На данном этапе возможны следующие неравномерности вычислительного процесса:

Т1) переполнение модуля АП из-за некачественной ХЭШ-функции, в результате чего в группе модулей ассоциативной памяти образуется дисбаланс по количеству токенов;
Т2) программа узла, выполняемая в ИУ, может выдавать как один токен, так и неограниченное количество токенов;

Т3) переполнение буфера токенов может произойти, если узлы, обрабатываемые в ИУ согласно ХЭШ-функции, будут выдавать токены, направленные в одни и те же модули АП;
Т4) неравномерность, обусловленная обработкой множественного отклика в модуле АП, что может привести к задержке приема входных токенов;
Т5) неравномерность, обусловленная аппаратной реализацией группового узла, при которой он размещается в отдельном МАП, создавая повышенную нагрузку на него;
Т6) неравномерность, которая связана с перегрузкой отдельных направлений, приводящей к временной блокировке коммутатора токенов.
Неравномерности создают недозагруженные устройства и, как следствие, происходит падение общей производительности. Но это еще полбеды. Беда, если вследствие неравномерности происходит переполнение одного из МАП, в результате чего программа просто не может продолжать работу. Таким образом, основную проблему представляет неравномерность заполнения МАП, проявляющаяся как переполнение некоторых модулей.

Методы решения — многопоточное ИУ, перелив, откачка пар, ХЭШ-функция, откачка-подкачка токенов.
П1: перелив пар в свободные ИУ.
П2, П3: откачка пар; сдерживание параллелизма.
П4: многозадачный режим.
Т1: иерархия АП (например, откачка-подкачка).
Т2: многопоточное ИУ.
Т4 и Т5: в условиях явно задаваемой функции распределения токенов групповой узел может быть распределен между несколькими МАП.
Т3 и Т6: адаптивная маршрутизация в коммутаторе токенов.

Итак, основная проблема — переполнение МАП — может решаться путем создания иерархии памяти: быстрая небольшая АП (кэш) + большая, но медленная (возможно, с промежуточными уровнями). При переполнении АП одного уровня новые (или старые замещаемые) токены начинают поступать в память следующего уровня. Основная трудность — при отсутствии отклика на некотором уровне неясно, есть ли смысл продолжать поиск на следующих уровнях. Важно исключить поиск, если на следующих уровнях его заведомо нет, поскольку иначе теряется смысл введения иерархии (если все равно требуется всегда продолжать поиск по всем уровням, то нет экономии). Поэтому надо во-первых, при вытеснении на следующий уровень в каком-то виде оставлять об этом информацию на предыдущем уровне, и, во-вторых, при освобождении предыдущего уровня по возможности перемещать ранее вытесненные токены обратно.

Язык программирования для модели вычислений, основанной на принципе потока данных DCF

Язык DCF проектируется как расширение языка последовательного программирования Си специальными возможностями для представления параллелизма в модели dataflow. Возможности языка Си предлагается использовать для программирования вычислений, составляющих тела нитей, а специальные функции и описания, составляющие его расширение – для описания dataflow-механизмов взаимодействия нитей.

Модель dataflow – параллельные вычисления, основанные на принципе потока данных

Модель вычислений dataflow, с момента своего появления примерно 30 лет назад, активно исследовалась и применялась как в области архитектур вычислительных систем, так и программного обеспечения.

Обычно модель вычислений (модель выполнения программы) определяет базовый уровень абстракции программирования, исходя из которого могут быть определены архитектура вычислительной системы, языки программирования, стратегии компиляции, runtime-системы и другие компоненты программного обеспечения. Модель dataflow является параллельной по своей природе - параллелизм заложен в определение dataflow-графа и правил его интерпретации. Надежды на эффективное выполнение программ в этой модели связаны с возможностью решения двух известных проблем последовательной модели вычислений фон Неймана – задержки при обращении к памяти (время от запроса к памяти до получения соответствующего ответа) и синхронизации (необходимость упорядочивать выполнение инструкций в соответствии с зависимостями по данным).

Исследования показали, что поддержка “чистой” модели dataflow в аппаратуре является сложной задачей, и, кроме этого, выполнение некоторых типов вычислений, например, строго последовательного вычисления, в модели dataflow неэффективно. Это привело к появлению “гибридных” моделей dataflow-фон Неймана (data-control flow моделей) и интенсивным исследованиям в этой области. В настоящее время эти исследования сконцентрированы, в основном, вокруг многонитевых (multithreaded) моделей. *Нить* (*thread*) – это набор инструкций, выполняемых последовательно, как в модели фон Неймана, а сами нити могут выполняться параллельно и запускаются при готовности своих входных аргументов, как в модели dataflow. Существуют различные многонитевые модели, допускающие различные правила выполнения тела нити: с блокировкой выполнения нити на время ожидания выполнения “долгих” операций типа обращения к памяти и без блокировки, а также различные правила переключения нитей.

Исследования многонитевых моделей привели в итоге к разработке архитектур многонитевых суперскалярных процессоров, которые, помимо достоинств суперскалярной архитектуры, позволяют выполнять несколько нитей одновременно за счет нескольких контекстов выполнения, а также поддерживают спекулятивные вычисления. Новые вычислительные модели стимулировали также развитие новых языков для программирования в этих моделях и методов компиляции.

Таким образом, вычислительная модель dataflow уже оказала положительное влияние на развитие теории и практики программирования, и в перспективе ее использование также может оказаться весьма полезным. При этом многонитевые архитектуры являются только одним классом dataflow-архитектур, хотя и неплохо продвинутым и проработанным. Весьма интересен и перспективен с этой точки зрения проект dataflow-компьютера, в котором dataflow-механизм инициализации вычисления реализуется при помощи ассоциативной памяти, что может дать принципиально новое по эффективности решение проблемы синхронизации вычислений. Мы считаем необходимым отметить, что знакомство именно с этим проектом стало отправной точкой нашего исследования

вопросов программирования в модели потока данных, первые результаты которых излагаются в данном препринте.

Новые модели вычислений, основанные на принципе dataflow, и языки программирования, основанные на этих моделях, могут оказаться полезными и в ближайшее время для программирования на существующих параллельных вычислительных системах, программирование на которых в настоящее время вызывает значительные трудности.

В данном препринте предпринимается попытка определения основных возможностей языка программирования высокого уровня, предназначенного для программирования в гибридной модели вычислений data-control flow – языка DCF (Data-Control Flow). Язык DCF проектируется достаточно традиционным способом - как расширение языка последовательного программирования Си специальными возможностями для представления параллелизма в модели dataflow. Неформально, возможности языка Си предлагается использовать для программирования вычислений, составляющих тела нитей, а специальные операторы и описания, составляющие его расширение – для описания dataflow-механизмов взаимодействия нитей.

Уровень языка DCF дает возможность использовать его в качестве промежуточного языка представления программ при трансляции традиционных языков программирования высокого уровня, dataflow-языков и декларативных языков программирования высокого уровня для dataflow-компьютеров и многонитевых архитектур, а также для непосредственного программирования. После исследования свойств языка DCF с применением имитационной модели dataflow-компьютера и уточнения его возможностей предполагается использование языка DCF в качестве промежуточного при реализации декларативного языка Норма [20] для dataflow-компьютера.

Основные термины и понятия языка DCF

Основными понятиями рассматриваемой модели вычислений являются *нить* и *токен*.

Нить является логической единицей выполнения программы. Она представляет собой цепочку операций, выполняемых в модели вычислений фон Неймана. При этом различные нити могут выполняться параллельно. Запуск нити происходит при появлении готовых значений всех входных параметров нити.

На уровне языка нить описывается функцией, единственным внешним отличием которой от “обычной” функции языка Си является наличие спецификатора *thread* вместо типа возвращаемого значения. Например:

```
thread Func( int N, float Var ) { … }
```

Главное отличие такой функции (будем называть ее функцией-нитью) от “обычной” функции состоит в том, что можно запустить нить, послав токены, содержащие значения всех аргументов, на имя функции-нити.

Запуск нитей, синхронизация нитей и передача значений между ними осуществляется с помощью токенов - единиц данных для dataflow управления.

Токен представляет собой объект, содержащий тег, идентифицирующий возможного получателя токена, и данные некоторых типов языка Си: int, float, char, адрес и др. Тег содержит два компонента: имя функции-назначения (функции-нити или request-функции), для которой предназначен токен, и некоторую характеристику, называемую цветом.

Токены используются для активации нитей – запуска или возобновления выполнения нитей. Токены порождаются при выполнении специальной стандартной функции.

Если нить передает данные другой нити, то она посылает в “окружающее пространство” токены, рассчитывая, что какая-нибудь другая нить примет их (что, в общем случае, не обязательно должно произойти). Если нить нуждается в какой-то информации, то она ожидает прихода токенов из “окружающего пространства” токенов. Пространство токенов играет в этих процессах активную роль, принимая токены от нитей,

собирая и группируя токены, а затем используя готовые группы токенов для активации нитей. Готовая группа содержит токен для каждого из аргументов активируемой функции-назначения, при этом теги всех этих токенов в некотором смысле совпадают.

Базовый язык

В этом разделе приведено неформальное описание ключевых возможностей языка DCF: описание расширений и ограничений языка Си, который положен в основу языка DCF.

Структура программы

Программа на языке DCF представляет собой набор функций, некоторые из которых имеют спецификатор thread. Такие функции (функции-нити) используются при порождении (запуске) нитей. Для того, чтобы запустить новую нить, необходимо послать на имя этой функции-нити токены, содержащие значения всех ее аргументов; при этом теги посылаемых токенов должны в некотором смысле совпадать (точное определение совпадения тегов будет описано в разделе 3.5). С помощью любой функции-нити можно породить произвольное количество экземпляров нити.

В программе среди описаний функций-нитей обязательно должно быть описание функции-нити с именем main. Запуск программы состоит в активации этой функции-нити. Функция-нить main может иметь аргументы, через которые программа получает параметры из внешней среды выполнения. Повторная активация функции-нити main запрещена.

Таким образом, в любой момент времени выполнения программы происходит параллельное выполнение множества нитей, запущенных с помощью активации функций-нитей.

Программа завершается, когда завершаются все нити.

Активация нити

Активация (запуск) нити осуществляется посылкой на имя функции-нити токенов, каждый из которых содержит значение некоторого аргумента функции-нити. Посылка токенов для запуска нити осуществляется с помощью стандартной функции token:

token(*значение_токена*, *тег_токена* [, *число_копий*])

значение_токена имеет вид *имя_арг : знач_арг*, где
имя_арг - имя или номер аргумента функции-нити,
знач_арг - значение аргумента;
тег_токена имеет вид *имя_функции-нити*([*цвет*])
число_копий указывает количество экземпляров токена. Если этот параметр опущен, то это означает, что *число_копий* равно единице. Если в качестве значения этого параметра указать символ “*”, то порождается “неограниченное” число копий токена.

Примеры: token(Var : 36.6, Func(5));
token(2 : i+j, Func(i), 3);

Первая функция посылает один токен на имя функции-нити Func. Токен имеет цвет, значение которого равно 5. Токен содержит значение 36.6 для аргумента с именем Var.

Вторая функция посылает три токена на имя функции-нити Func. Все они содержат значение i+j для второго аргумента функции-нити Func и имеют цвет, равный значению переменной i.

Заметим, что параметр *имя_арг* однозначно определяет тип значения *знач_арг*, поскольку этот тип определен в заголовке функции-нити.

Если функция-нить имеет один аргумент или не имеет его вовсе, то параметр *имя_арг* можно опустить. Функции-нити с пустым списком аргументов активируются посылкой на имя такой функции токена с пустым значением (NULL).

Цвет представляется целочисленным вектором (i,j,k,…). Цвет можно замаскировать, указав в качестве его значения символ “*”. Можно замаскировать только некоторые

элементы вектора цвета, поставив вместо соответствующего элемента символ “*” (смысл маскирования описан ниже в разделе 3.5). Например, функция

token(NULL, thMult(2, *, i, j, *))

пошлет один токен, активирующий не имеющую входных параметров нить thMult. Токен имеет цвет с двумя замаскированными элементами вектора цвета.

Нить может определить цвет активировавших ее токенов, выполнив стандартную функцию getThreadColor. В этом смысле нить “имеет цвет”. Формат обращения к функции:

int getThreadColor(int *V)

Выполнение функции состоит в том, что она последовательно заполняет элементы вектора V значениями элементов вектора цвета до тех пор, пока либо исчерпается вектор V, либо исчерпается вектор цвета. Возвращает функция значение, равное длине вектора цвета.

Если некоторый компонент вектора цвета замаскирован, то соответствующий элемент вектора V получает значение NULL. Значения элементов вектора V, для которых не нашлось значений элементов вектора цвета (т.е. длина вектора V больше длины вектора цвета) не определены.

Пример. Допустим, есть функция-нить myFunc:

```
thread   myFunc( char a )
{   int  lengthColor, myColor[3];
                    lengthColor = getThreadColor( myColor );
}
```

и мы запустили два экземпляра этой нити с помощью токенов

```
token( ‘f’, myFunc(1,*) );
token( ‘s’, myFunc(1,*,3,4,*) );
```

Тогда в первом экземпляре нити переменные lengthColor и myColor приобретут значения:

lengthColor = 2;
myColor = { 1, NULL, *неопределенное_значение* }

а во втором:

lengthColor = 5;
myColor = { 1, NULL, 3 }

Нить может породить другие нити, имеющие такой же цвет, что и она сама. Для этого в функции token параметр *цвет* должен быть опущен, что означает: цвет посылаемого токена равен цвету нити, то есть токенов, запустивших данную нить.

Нить main имеет цвет “NULL”.

В языке имеется стандартная функция **int getNewColor()**, каждый вызов которой генерирует уникальное значение цвета.

С помощью одной функции token можно посылать токены для разных аргументов некоторой функции-нити. Для этого вместо одного *значения_токена* нужно указать несколько элементов *значения_токена*, разделенных запятой.

Например:

token(arg1:value1, arg2:value2, *тег_токена* [, *число_копий*])

Выполнение этой функции идентично выполнению двух функций token:

token(arg1:value1, *тег_токена* [, *число_копий*])
token(arg2:value2, *тег_токена* [, *число_копий*])

Завершение нити

Завершение нити происходит при выполнении функции **exit_thread**. Если эта функция является последней в теле функции-нити, то она может быть опущена.

Нить не может возвратить значение в смысле языка Си, поэтому функции языка Си return(…) и exit(…) в теле функции-нити интерпретируются как функция exit_thread.

Проиллюстрируем средства активации и завершения нитей на следующем примере.

Пример. Пусть необходимо выполнить вычисление заданной функции Func над каждым из значений a[i], где i=0,1, …, 99; при этом мы хотим, чтобы вычисления Func над разными значениями выполнялись параллельно. Программа на языке DCF может выглядеть так:

```
thread main()
{ int i;
  float a[99]={ … };
  for(i=0; i<100; i++)
            token(a[i], Func() );
}
thread Func(float a)
{
  <вычисления над a >
     exit_thread;
}
```

Сложность программы практически не отличается от последовательного варианта, хотя программа и стала параллельной. В этом примере нить main порождает сто параллельных нитей Func, отправив на имя функции-нити Func сто токенов со значениями a[i].

3.4. Синхронизация и обмен данными между нитями

Часто возникает необходимость в синхронизации выполнения нитей и в обмене данными между нитями. Например, в приведенном выше примере может потребоваться обмен данными между нитями Func и нитью main для того, чтобы последняя могла просуммировать результаты вычислений над значениями a[i]. Для организации такого обмена используются функции со спецификатором request вместо типа возвращаемого значения (остановка выполнения нити, ожидание и прием токенов) и уже знакомая функция token (посылка токена).

Описание функции со спецификатором request (будем называть такие функции “request-функциями”) имеет следующий формат:

request *имя* (*адрес_переменной,* ([*цвет*]))

где: *имя* имя request-функции;

адрес_переменной адрес переменной, которой необходимо присвоить значение, присланное токеном;

цвет цвет ожидаемого токена.

Request-функции имеют три важных отличия от «обычных» функций языка Си:

- описание request-функции может появиться в любом месте тела любой функции;
- описание request-функции является одновременно и определением функции и ее вызовом;
- имя request-функции должно быть уникальным в пределах всей программы, поэтому полное имя request-функции имеет вид

имя_функции . идентификатор

где *имя_функции* является именем функции-нити или обычной функции, в которой определена request-функция;

идентификатор является уникальным в пределах функции *имя_функции*.

В пределах функции *имя_функции* первую компоненту полного имени request-функции можно опустить и указывать только компоненту *идентификатор*, однако, посылая токены на request-функцию, необходимо указывать ее полное имя.

Выполнение request-функции состоит в следующем. Функция останавливает выполнение нити и сообщает пространству токенов о потребности в токене с указанным цветом.

Как только в пространстве токенов появляется токен, запрашиваемый ждущей request-функцией (или если он уже там есть), то данные, содержащиеся в токене, присваиваются переменной с адресом *адрес_переменной* и выполнение нити возобновляется.

Если в описании request-функции опущен параметр *цвет*, то это означает, что функция ожидает токен, имеющий тот же цвет, что и нить, в которой выполняется эта функция.

Request-функция позволяет вместо *адрес_переменной* указывать список адресов переменных, т.е. последовательность *адрес_переменной* через запятую. В этом случае request-функция ожидает прихода готовой группы токенов. Готовая группа содержит токен для каждой переменной request-функции, при этом теги всех этих токенов в некотором смысле совпадают (точное определение совпадения тегов будет описано в разделе 3.5).

Формат обращения к функции token при посылке токена(ов) для request-функции имеет практически тот же вид, что и при посылке токенов для запуска нитей (т.е. для функции-нити):

token(*значение_токена*, *тег_токена* [,*число_копий*])

значение_токена имеет вид *номер_арг : знач_арг*, где
номер_арг - номер аргумента request-функции,
знач_арг - значение аргумента;
тег_токена имеет вид *имя_request-функции*([*цвет*])
число_копий указывает количество экземпляров токена.

Обратим внимание на отличия в синтаксисе функции token, используемой при посылке токенов для request-функции:

- поскольку адресатом токенов является request-функция, то *тег_токена* должен включать в себя не имя *функции-нити,* а полное имя *request-функции*, то есть *тег_токена* имеет вид
  имя_функции . *идентификатор* ([*цвет*])

Пример. Дополним приведенный выше пример запуска ста нитей Func таким образом, чтобы нить main могла просуммировать результаты работы всех ста экземпляров нити Func. Программа на языке DCF может выглядеть так:

```
thread main()
{ int i;
  float a[99]={ … };
  float rez, Result=0.0;
//  запускаем нити Func
  for(i=0; i<100; i++)
              token(a[i], Func() );
//  ждем результатов работы нитей Func
     for(i=0; i<100; i++)
              { request main.Sum( &rez, () );
                Result =+ rez;
              }
}
thread Func(float a)
{  float var;
   <вычисления над a и присвоение значения переменной var>
     token( var, main.Sum() );
     exit_thread;
}
```

Маскирование тега токена. Группировка токенов

Тег токена можно расматривать как своего рода “адрес” получателя токена в функции token. В ряде случаев можно добиться заметного упрощения программы и/или увеличения степени ее параллельности, если этот адрес указывать не совсем “точно”, т.е. указывать вместо конкретного отправителя (получателя) множество допустимых отправителей (получателей).

Такая возможность реализуется путем маскирования компонент тега. Маскирование осуществляется путем указания специального значения “*” в качестве значения соответствующего компонента тега. При этом цвет можно маскировать как целиком, указав значение “*” на месте параметра *цвет*, так и отдельные элементы вектора, указывающего цвет.

Точная семантика маскирования компонентов тега определяется логикой поведения пространства токенов, которое получает токены, формирует из них готовые группы и использует эти группы для активации нитей.

В данном разделе мы рассмотрим возможность маскирования только одного компонента тега токена, а именно – цвета. Возможность маскирования другого компонента тега – имени функции-назначения токена – рассматривается в разделе 4.2.

Структура пространства токенов.

Пространство токенов состоит из групп токенов. Каждая группа содержит тег, который так же, как и тег токена, состоит из двух компонентов: имени и цвета.

Построение групп токенов.

Если в пространстве токенов появляется новый токен, то делается попытка включить его в одну из существующих групп. Порядок просмотра групп при этом не определен.

Сначала определим правила включения в группу токена, у которого *число_копий* = 1 (который существует в единственном экземпляре).

Если включить токен в какую-либо группу можно, то он включается в эту группу.

Если включить токен в какую-либо группу не удается, то создается новая группа, токен включается в эту группу и тег группы устанавливается равным тегу токена. Таким образом, имя группы устанавливается равным имени функции назначения, а цвет группы – равным цвету токена.

В случае, если для токена было задано *число_копий* > 1, но *число_копий* ≠ “*” указанная процедура производится для каждой копии токена.

Если для *числа_копий* было задано значение “*” (токен с неограниченным количеством копий), то попытка включения проводится для всех уже существующих групп, а также для всех групп, которые будут возникать в дальнейшем.

При включении токена в группу тег этой группы может измениться.

Включение токена в группу.

Токен можно включить в группу в том и только том случае, если токен подходит группе по имени, по цвету и по номеру аргумента-получателя (то есть аргумента в заголовке функции-нити или переменной в request-функции).

Токен подходит группе по имени, если имя назначения в теге токена совпадает с именем группы.

Если цвет группы равен “*”, то ей подходит (по цвету) токен с любым цветом; при включении токена в такую группу цвет токена становится цветом группы.

Цвет группы (если он не равен “*”) представляет собой в общем случае целочисленный вектор (c1,c2, …, cN), где каждый элемент вектора ci может либо иметь целочисленное значение, либо быть замаскирован (т.е. иметь значение “*”). В такую группу могут быть включены либо токены с замаскированным цветом, либо токены, цвет которых имеет значение (t1,t2,…,tN), где если ci ≠ ti, то либо ci = “*”, либо ti = “*”. Если токен включается в группу, то цвет группы “доопределяется”: замаскированные элементы цвета группы заменяются незамаскированными элементами цвета токена (если ci = “*”, а ti ≠ “*”, то элемент ci заменяется на элемент ti).

Токен подходит группе по номеру аргумента-получателя, если в группе не существует токена с таким же номером аргумента (следовательно, в группе не может быть несколько токенов с одинаковым номером аргумента-получателя).

Выбор готовой группы для запуска нити.

Готовой группой для запуска является группа, в которой есть ровно один токен для каждого аргумента функции-нити с именем, подходящим имени группы.

После включения токена в группу, пространство токенов определяет, возникла ли готовая группа. Если готовая группа возникла, то аргументам функции-нити присваиваются значения, взятые из соответствующих им токенов группы, и запускается новая нить, имеющая цвет, равный цвету группы.

После этого готовая группа и составлявшие ее токены удаляются.

Выбор готовой группы для возобновления нити.

Готовой группой для возобновления является группа, в которой есть ровно один токен для каждой переменной, чей адрес указан в request-функции, с тегом, который подходит тегу, указанному в request-функции.

При выполнении request-функции пространство токенов проверяет, существует ли готовая группа. Если такая группа находится, то переменным, чьи адреса указаны в request-функции, присваиваются значения, взятые из соответствующих им токенов группы, и возобновляется выполнение нити.

После этого готовая группа и составлявшие ее токены удаляются.

Если при выполнении request-функции подходящей группы не находится, то создается новая группа с тегом, равным тегу, указанному в request-функции.

Удаление токенов и групп из пространства токенов.

В языке имеются стандартные функции kill_group и kill_token, позволяющие удалять определенные токены или группы токенов из пространства токенов.

Формат функций:

kill_group(*тег_токена* [, *число_копий*])

kill_token(*тег_токена* [, *число_копий*])

где *тег_токена* - тег токена, возможно, с замаскированными компонентами тега;

число_копий - число удаляемых токенов или групп.

Выполнение функции kill_group приводит к удалению групп токенов указанного типа, чей тег подходит тегу *тег_токена*. Число удаляемых групп задается параметром *число_копий,* при этом если он равен “*”, удаляются все группы. Данная функция может быть использована, в частности, для чистки пространства токенов.

Выполнение функции kill_token приводит к удалению токенов, чей тег подходит тегу *тег_токена*. Число удаляемых токенов также задается параметром *число_копий,* при этом если он равен “*”, удаляются все токены. В последнем случае данная функция позволяет удалить из пространства токенов те токены, которые имели неограниченное количество копий.

Если группа состоит из одного удаляемого токена, то выполнение функции kill_token приводит также к удалению группы, в которой этот токен находится.

Видимость переменных в языке

Согласно стандарту Си, переменные могут быть внешними (описанными вне функций) и автоматическими (описанными внутри функций).

Внешние переменные являются глобальными и доступны для всех нитей; спецификатор static ограничивает видимость только теми нитями, описания которых (функции-нити) находятся в данном файле.

Автоматические переменные порождаются при запуске нити и доступны только внутри данной нити. Спецификатор static в отношении автоматических переменных игнорируется.

Таким образом, обмен данными между нитями осуществляется либо при помощи внешних переменных (в этом случае дисциплину доступа к таким переменным из разных нитей определяет и обеспечивает программист), либо при помощи токенов (в этом случае семантику токенов поддерживает система программирования).

Пример использования базовых средств языка DCF

Допустим, нам нужна функция HowMany, подсчитывающая число вхождений литеры letter в строку. Построим эту функцию следующим образом. Функция запустит нить SplitString, передав ей начало и конец строки.

Нить SplitString анализирует длину полученной строки. Если длина не превышает 10, то нить подсчитывает число вхождений литеры letter и посылает функции HowMany это число, а также длину обработанного участка. Если же длина больше 10, то нить запускает две новые нити SplitString, передав каждой из них соответствующую половину своей строки, и так далее.

Функция HowMany принимает токены от нитей SplitString, суммирует число вхождений и подсчитывает общую длину обработанных участков строки. Когда эта длина станет равной длине исходной строки, функция возвращает искомое число вхождений и завершает свою работу.

И, наконец, сделаем так, чтобы при каждом вызове функции HowMany она порождала нить SplitString уникального цвета (все следующие порождаемые нити будут того же цвета). Тогда этой функцией могут одновременно пользоваться разные функции-нити программы.

```
//функция получает литеру letter, адрес строки S и длину этой строки N
int HowMany(char letter, char *S, int N)
{    int rezCount, rezLength, result=0;
     // Создаем уникальный цвет порождаемых нитей
   int threadColor = getNewColor();
// Запускаем нить SplitString
     token( 1:letter, 2:S, 3:0, 4:N-1, SplitString( threadColor ) );
// Ожидаем токены уникального цвета от нитей SplitString,
// содержащие число вхождений и длину обработанного участка строки
     while(N>0)
            { request F( &rezCount, &rezLength, (threadColor) );
              result = + rezCount;
              N =  -  rezLength;
            }
// Возвращаем результат – число вхождений литеры letter в строку S
     return  result;
}
thread SplitString( char letter, char *S, int ps, int pe )
{    int pm, i, count=0;
     if( pe-ps+1>10 )
// Строка слишком длинная.
// Делим ее пополам и запускаем две нити SplitString
     {   pm = (pe-ps+1)/2;
         token( 1:letter, 2:S, 3: ps, 4:pm-1, SplitString() );
         token( 1:letter, 2:S, 3: pm, 4:pe, SplitString() );
     }
else
// Строка короткая. Считаем сами и посылаем результат нити HowMany
     {   for( i=ps; i<=pe; i++ )
                        if( S[i] == letter) count++;
         token( 1:count, 2:pe-ps+1, HowMany.F() );
```

```
        }
    }
```

Дополнительные возможности

В этом разделе описаны некоторые дополнительные средства языка DCF, позволяющие увеличить его выразительные возможности. Конечно же, возможности расширения языка не исчерпываются описанными ниже средствами; они скорее иллюстрируют некоторые важные направления таких расширений.

Системные токены

В ряде ситуаций пространство токенов само порождает токены, которые нити могут обнаруживать и использовать для идентификации и обработки возникшей ситуации или события. Такие токены называются *системными токенами*.

Программа пользователя также может порождать системные токены в тех случаях, когда требуется получить какой-то сервис со стороны системного окружения.

Системные токены имеют тег с именем, идентифицирующим группу особых событий, и уникальным цветом, идентифицирующем конкретное событие в группе. Значение токена в некоторых случаях несет дополнительную информацию о событии.

Примеры системных токенов:

token(*значение*, THREAD_ERROR(THREAD_ABORT))
: возникает при аварийном завершении какой-то нити;

token(*значение*, SYS_ERROR(IO_ ERROR))
: возникает при появлении ошибки ввода/вывода;

token(*значение*, *имя_системной_нити*(EXCEPTION))
: такой токен посылает пользовательская программа при необходимости обработки исключительной ситуации, определяемой пользователем.

Если программист желает определить свою собственную обработку какой-то исключительной ситуации, то он должен определить функцию-нить со стандартным именем, связанным с данным классом исключительных ситуаций. Эта функция-нить должна на основе анализа цвета запустивших ее системных токенов и полученного значения идентифицировать возникшую ситуацию и выполнить необходимые действия.

Модель вычислений языка «Пифагор»

История языков параллельного программирования насчитывает не один десяток лет. Придуманы и апробированы разнообразные способы написания программ, которые условно можно разделить на три группы.

Последовательное программирование с дальнейшим автоматическим распараллеливанием.
Непосредственное формирование потоков параллельного управления, с учетом особенностей архитектур параллельных вычислительных систем или операционных систем. Зачастую используется расширение традиционных последовательных языков операторами распараллеливания и синхронизации. Уровень параллелизма полностью контролируется разработчиком и может меняться в зависимости от конкретных условий.
Описание параллелизма без использования явного управления обеспечивается заданием только информационных связей. Предполагается, что программа будет выполняться на вычислительных системах с бесконечными ресурсами, операторы будут запускаться немедленно по готовности их исходных данных.
Последовательное программирование выглядит весьма привлекательным, поскольку, на первый взгляд, позволяет: применять опыт, накопленный в течение многолетней эксплуатации вычислительных машин с традиционной архитектурой; использовать продукты, разрабатываемые десятилетиями; разрабатывать программы, не зависящие от особенностей параллельной вычислительной системы или ОС; не заботиться о параллельной организации программ, полагаясь на системы автоматического распараллеливания, решающие все вопросы по эффективному переносу программ на любую параллельную вычислительную систему. В свое время этот подход достаточно широко и разносторонне исследовался, однако многие из возлагавшихся на него надежд не оправдались. Сейчас он практически не используется в промышленном программировании, так как обладает следующими недостатками:

распараллеливание последовательных программ очень редко обеспечивает достижение приемлемого уровня параллелизма из-за ограничений вычислительного метода, выбранного для построения соответствующей программы, что часто обуславливается стереотипами последовательного мышления;
в реальной ситуации учесть особенности различных параллельных систем гораздо труднее, чем это изначально предполагалось.
Помимо этого, современные программы уже трудно считать последовательными. Повсеместно используются процессы, нити и прочие атрибуты параллельного программирования, предоставляемые операционными системами. Сами программы выполняются в многозадачном режиме, распределяются по клиентским и серверным компьютерам и т. д.

Практически во все времена в параллельном программировании наиболее популярными были подходы, учитывающие особенности параллельных вычислительных систем. При этом наряду с языками, применяются библиотечные средства поддержки параллельных процессов, реализованные на уровне операционных систем или специализированных пакетов. Несомненным достоинством здесь является максимальное использование возможностей конкретной архитектуры — программист, опираясь на знание специфической системы, потенциально может весьма эффективно распределять задачу.
Параллельное программирование для конкретных архитектур широко использовалось в конце 70–х — начале 80-х годов. Были разработаны десятки вычислительных систем, значительно отличающихся друг от друга.

В дальнейшем развитие технологии создания однокристальных микропроцессоров привело к резкому снижению эффективности подобных разработок и к нецелесообразности построения нетрадиционных архитектур. Произошел переход к высокопроизводительным системам на основе стандартных аппаратных средств: симметричная многопроцессорность (SMP), кластеры, массовый параллелизм (MPP) и т.д. Это позволило унифицировать методы параллельного программирования, увязав их с семействами архитектур. Появились переносимые инструментальные средства и пакеты, позволяющие выполнять одни и те же программы на вычислительных системах, имеющих разные системы команд, управляемые различными операционными системами. К подобным программным системам следует отнести MPI (ориентация на передачу сообщений в системах с распределенной памятью [1]) и PVM (ориентация на параллельное программирование для систем с общей памятью [2]). Наряду с пакетами, обобщающими семейства архитектур, появились средства, обеспечивающие прозрачную поддержку кластерных вычислений для наиболее популярных операционных систем. К ним следует отнести систему Mosix [3], которая после компиляции с ядром Unix-подобных ОС обеспечивает миграцию по узлам кластера процессов, построенных с использованием стандартных методов их порождения (системный вызов fork).

Вместе с тем, использование явного управления вычислениями обладает рядом недостатков:

программист сам должен формировать все параллельные фрагменты и следить за корректной синхронизацией процессов;
использование в языках такого типа "ручного" управления памятью может привести к конфликтам между процессами в борьбе за общий ресурс (программисту самому приходится тщательно следить за распределением памяти);
программы жестко привязаны к конкретной архитектуре или к семейству архитектур (смена поколений вычислительных систем или появление новых ведут к необходимости переработки накопленного программного обеспечения).
В этой связи неудивительно, что для достижения мобильности параллельных программ, написанных с использованием явного управления, постоянно разрабатывались соответствующие средства, направленные на преодоление трудностей, не существовавших при последовательном программировании.

В последовательных программах управление вычислениями реализовано достаточно просто и легко может быть преобразовано к другим формам. Методы построения параллельных программ ведут к дополнительным проблемам, возникающим при анализе потоков управления.

Наряду с более сложным управлением параллельные программы различного вида имеют и специфические механизмы, обеспечивающие разрешение конфликтов при доступе к общим ресурсам. Выявление этих механизмов и замена их соответствующими эквивалентами также затрудняет трансформацию одних параллельных программ в другие.

Существует еще более жесткая связь между используемыми вычислительными методами и структурой программы, что, в свою очередь, еще сильнее осложняет процесс переноса параллельной программы с одной архитектуры на другую.

Для достижения высокой эффективности необходим разумный баланс между параллелизмом и последовательными фрагментами: система должна учитывать особенности реализации и производительность отдельных подсистем конкретной

архитектуры и, в соответствии с этим, ориентироваться на распараллеливание вычислений или наоборот, последовательную реализацию этих же фрагментов программы.

Мобильность и неявное управление

Разработка переносимых параллельных программ возможна только в том случае, если язык программирования позволяет скрыть механизмы управления вычислениями, специфичные для конкретных архитектур. Особенности обычно проявляются в явном управлении различными ресурсами. Дополнительные «шумы» вносятся управлением последовательностью вычислений по собственному усмотрению. Для того чтобы «избавить» программиста от подобных возможностей, были предложены языки с неявным описанием параллелизма, предусматривающие указание только информационной взаимосвязи между функциями, осуществляющими преобразование данных.
Использование таких языков позволяет создавать программы, поддерживающие параллелизм на уровне операций, ограниченный лишь методом решения задачи; обеспечивать перенос на конкретную архитектуру, не распараллеливая программу, а сжимая ее максимальный параллелизм; проводить оптимизацию программы по множеству параметров с учетом архитектуры параллельной вычислительной системы, для которой осуществляется трансляция. Работы в этой области условно можно разделить на два направления: языки и методы функционального программирования; методы управления вычислениями по готовности данных.

Функциональное программирование

Функциональное программирование, имеющее давнюю историю, определяется как «способ построения программ, в которых единственным действием является вызов функции, единственным способом расчленения программы на части является введение имени для функции и задание для этого имени выражения, вычисляющего значение функции, а единственным правилом композиции — оператор суперпозиции функции» [4]. Среди первых языков, обеспечивших поддержку ограниченного параллелизма, был FP [5], возможность писать параллеллельные программы реализована в одной из версий Haskell (www.haskell.org). Параллелизм существующих функциональных языков ограничен рядом общих факторов.

Набор базовых операций и структур данных изначально ориентирован на поддержку последовательных вычислений. Так, списки состоят из "головы" и "хвоста", что требует рекурсивной развертки и затрудняет одновременную обработку всех элементов. Поэтому последующие расширения таких языков могут служить только надстройкой, не поддерживающей параллелизм на уровне отдельных операций.
Включение механизмов поддержки параллелизма во время создания ряда функциональных языков осуществлялось без полного анализа всех возможных вариантов. Как результат этим языкам присущи только ограниченные формы организации параллельных вычислений.
Ограничения накладываются и синтаксисом языков. В некоторых из них просто отсутствует возможность описать параллелизм, присущий задаче.
Управление по готовности данных
Для того чтобы язык программирования мог обеспечить формирование параллелизма любого типа, он должен на уровне элементарных операций содержать механизмы размножения информационных потоков, их группирования в списки данных различного уровня вложенности, одновременного выделения из списков нескольких независимых и разнородных по составу групп. Обычно такие возможности заложены в потоковых моделях вычислений и языках, построенных на их основе.

Известны различные модели вычислений, поддерживающие управление по готовности данных [6]. В каждый из этих подходов заложен определенный метод порождения и синхронизации параллельных процессов. Однако в некоторых случаях реализованная модель вычислений не позволяет обеспечить максимальный параллелизм из-за ограничений, присущих отдельным механизмам организации параллельных вычислений. Эти проблемы решаются введением модели «идеального» вычислителя, обладающего бесконечными ресурсами, мгновенно выделяемыми по первому требованию для выполнения любой операции и хранения любых промежуточных данных.

Пути решения проблемы

Отсутствие практического интереса к созданию «идеальных» систем, обеспечивающих разработку переносимых параллельных программ, во многом обуславливается семантическим разрывом между методами написания программ, которые должны использовать параллелизм на уровне операций, и современными техническими средствами, обеспечивающими реальную поддержку только крупноблочного распараллеливания. Эффективнее и проще осуществлять кодирование, непосредственно поддерживающее архитектурно зависимые методы.

Вместе с тем, исследования в области создания переносимых параллельных программ представляют интерес, так как, в перспективе позволяют избежать переписывания кода при изменении архитектур вычислительных систем. Усилия должны быть направлены на создание языка, который не является основой машинной архитектуры, а предназначен для решения следующих задач:

исследования и разработки систем программирования, обеспечивающих написание программ, не зависящих от архитектур конкретных параллельных вычислительных систем;
создания методов преобразования архитектурно независимых параллельных программ в архитектурно зависимые;
разработки промежуточных интерпретирующих средств, позволяющих выполнять архитектурно независимые программы на реальных системах.

Отсутствие средств, обеспечивающих решение этих задач, привело авторов к разработке функционального языка «Пифагор» (http://www.softcraft.ru/parallel.shtml), позволяющего распараллеливать программы на уровне операций, имеющего архитектурную независимость, достигаемую за счет описания только информационных связей, обладающего асинхронным параллелизмом, полученным путем выполнения операций по готовности данных. В языке отсутствуют переменные, что позволяет избежать конфликтов, связанных с совместным использованием памяти параллельными процессами.

Модель вычислений языка «Пифагор»

Особенности языка во многом определяются используемой моделью управления вычислениями. Именно она определяет подход к построению переносимых программ. Программа задается информационным графом, определяющим потоки передачи данных между вершинами - операторами. Операторы обеспечивают преобразования данных, их структуризацию и размножение. Предусмотрены следующие типы вершин: оператор интерпретации; константный оператор; оператор копирования данных; оператор группировки в список; оператор группировки в параллельный список; оператор группировки в список задержанных вычислений.

Динамика выполнения операторов задается механизмом продвижения начальной разметки графа по дугам модели. Она определяет обработку исходных данных, получение

результатов и их использование в дальнейших вычислениях. Моделирование вычислительного процесса осуществляется в соответствии с рядом правил.

Если все входные дуги вершины имеют разметку (исходные данные), то на выходных дугах происходит формирование разметки (результата) в соответствии с правилами срабатывания, определяемые оператором.
Если входные разметки имеют кратность, превышающую единицу (одновременно сформировано несколько независимых наборов данных), то для заданной вершины формирование выходной разметки может начинаться при неполной входной разметке независимо для каждого из сформированных наборов входных данных, и осуществляется в соответствии с аксиомами срабатывания операторов.
В процессе распространения разметка не убирается и не замещается. Каждая дуга графа может получить разметку только один раз. Из требования о недопустимости повторной разметки вытекает требование ацикличности информационного графа.
Процесс распространения разметки заканчивается, когда все дуги графа имеют полную разметку в соответствии с предписанной кратностью или при невозможности распространения разметки.
Поддержка разметкой дуги нескольких независимых наборов данных позволяет описывать на уровне модели вычислений массовый параллелизм. При этом инициализация вычислений в вершине может начинаться до формирования полной разметки, так как обработка каждого из входных наборов осуществляется независимо.

Каждый из операторов модели обладает своими свойствами.

Оператор интерпретации описывает функциональное преобразование аргумента и имеет два входа, на которые через информационные дуги поступают функция F и аргумент X.

Аргумент и функция могут быть результатами предшествующих вычислений. Оператор запускается по готовности данных. Получение результата задается разметкой выходной дуги. При текстовом описании оператор интерпретации имеет две формы: постфиксную, обозначаемую знаком «:», и префиксную, при которой функция отделяется от аргумента знаком «^». Наличие двух способов записи позволяет в дальнейшем комбинировать их для получения более наглядного текста программы. Выражение F(X) оператор интерпретации задает как X:F или F^X.

Константный оператор не имеет входов (рис. 2). У него есть только выход, на котором постоянно находится разметка, определяющая значение константы. В языковом представлении константный оператор задается величиной предопределенного типа или константным выражением.

Оператор копирования осуществляет передачу данных с единственного входа на множество выходов. В текстовой форме копирование определяется через именование передаваемой величины и дальнейшее использование введенного обозначения. Используется постфиксное именование размножаемого объекта в форме: величина >> имя, и его префиксный эквивалент, имеющий вид: имя << величина. Например: y << F^x; (x, y):+ >> c. В графическом представлении (рис. 3) передача фиксируется установкой разметки на дугах, связанных с выходами вершины при размеченной входной дуге.

Оператор группировки в список (рис. 4) имеет несколько входов и один выход. Он обеспечивает структуризацию и упорядочение данных, поступающих по дугам из различных источников. Порядок элементов определяется номерами входов, каждому из которых соответствует натуральное число в диапазоне от 1 до N.

В текстовом виде оператор задается ограничением элементов списка круглыми скобками. Нумерация элементов списка в данном случае задается неявно в соответствии с порядком их следования слева направо; это соглашение действует и в графическом представлении при отсутствии явной нумерации входов.

Оператор создания параллельного списка (рис. 5) обеспечивает группирование элементов, аналогичное списку данных. В текстовом виде группировка в параллельный список задается ограничением его элементов квадратными скобками. Кратность разметки, определяющая выходной набор данных равна сумме кратностей разметок всех входных дуг. Считается, что функция использует каждый элемент данного списка как независимый аргумент:

[x1,x2,...,xn]:f = [x1:f,x2:f,...,xn:f]

Если же параллельный список определяет набор функций, то все они выполняются одновременно над одним и тем же аргументом:

X:[f1,f2,...,fn] = [X:f1,X:f2,...,X:fn]

Оператор группировки в задержанный список задается вершиной, содержащей допустимый вычисляемый подграф, в котором возможно несколько входов и выходов. Входы связаны с дугами, определяющими поступление аргументов, а выходы определяют выдаваемый из подграфа результат (рис. 6). Спецификой такой группировки является то, что ограниченные оператором вершины (на графе ограничение задается контуром), представляющие другие операторы, не могут выполняться, даже при наличии всех аргументов. Их активизация возможна только при снятии задержки (раскрытие контура), когда ограниченный подграф становится частью всего вычисляемого графа.

Первоначально задержанный подграф создает на своем единственном выходе константную разметку, которая является его образом. Эта разметка распространяется по дугам графа от одного оператора к другому, размножаясь, входя в различные списки и выделяясь из них до тех пор, пока не поступит на один из входов оператора интерпретации. При наличии на его обоих входах готовых данных происходит подстановка, вместо образа, самого задержанного графа с сохранением его входных связей. Опоясывающий контур при этом «убирается», и происходит выполнение активированных операторов. В результате на выходной дуге раскрытого подграфа вновь формируется результирующая разметка, которая и является одним из окончательных операндов оператора интерпретации. Данная процедура называется раскрытием задержанного подграфа. В текстовом виде группирование в список задержанных вычислений (для краткости будем называть его также «задержанным списком») задается охватом соответствующих операторов фигурными скобками. Его раскрытие ведет к образованию параллельного списка. Наличие этой конструкции позволяет откладывать момент начала некоторых вычислений или вообще не начинать их, что используется при организации выборочной обработки данных.

Наряду с особенностями, присущими различным операторам, следует остановиться на использовании оператора интерпретации. Он обеспечивает преобразование аргумента X, в результат Y, в соответствии с функцией F. В постфиксной нотации, выбранной для дальнейших иллюстраций, данное преобразование записывается так: X:F => Y.

Множество унарных функций F полезно разделить на два подмножества: F = {f1, f2}, где f1 — множество предопределенных функций языка, для каждой из которых аксиоматически задаются области определения и изменения, а f2 — множество функций, порождаемых при программировании. Следует отметить: выбор базового набора предопределенных функций осуществлен до некоторой степени субъективно, из соображений удобства. Введены арифметические функции, функции сравнения и прочие, аналогично тому, как это сделано и в других языках программирования. Например, функция сложения двух чисел x1, x2 задается так: (x1, x2):+, где первый аргумент оператора интерпретации является двухэлементным списком данных. Каждый элемент этого списка должен быть числом. Второй аргумент оператора интерпретации является функцией сложения, обозначенной значком «+». Результат функции является атомарным элементом.

Основные приемы функционально-параллельного программирования
Использование параллельных списков
Отсутствие операторов цикла в функциональных языках ведет к использованию рекурсии. В ряде случаев ее можно избежать, если алгоритм задачи предусматривает выполнение одной функции или списка функций над списком независимых аргументов. Тогда можно воспользоваться механизмом параллельных списков.

Параллельный список обеспечивает одновременное выполнение одной операции над всеми его элементами. Рассмотрим пример скалярного произведения двух векторов одинаковой длины. Аргументом функции ScalMultVec является двухэлементный список данных: W=((x1, x2, ... xn), (y1, y2, ... yn)). Функциональная программа будет организована следующим образом:

```
ScalMultVec << funcdef W { (W:#:[]:*) >> return }
```

Двухэлементный список W, состоящий из двух векторов, транспонируется (операция «#»), образуя множество пар (xi, yi). Следующая затем операция «[]» создает параллельный список пар, над которыми одновременно выполняется операция умножения «*». Результат, оказывается размещенным внутри круглых скобок, определяющих сформированный вектор. Его обозначение ключевым словом return ассоциируется с выдачей из функции. Пример:

```
((3, 5), (-4, 2)): ScalMultVec => (-12, 10)
```

Функции тоже могут задаваться параллельным списком, определяя тем самым множество потоков функций над одним потоком данных. Следующий пример иллюстрирует одновременное нахождение суммы, разности, произведения и частного двух чисел:

```
ParAddSumMultDiv << funcdef Param {(Param:[+,-,*,/]) >>return }
```

Результатом вычислений является четырехэлементный вектор, полученный применением разных операций к одному и тому же двухэлементному числовому списку, например:

```
(3, 5): ParAddSumMultDiv => (8, -2, 15, 6.000000e-001).
```

Использование задержанных списков
Для программирования вычислительных алгоритмов, предусматривающих ветвление, применяются задержанные списки с последующим выбором и раскрытием элемента,

соответствующего дальнейшим вычислениям. Рассмотрим пример функции, находящей абсолютное значение скалярного аргумента:

```
Abs << funcdef Param {
({Param:-}, Param):
[(Param,0):(<,=>):?]:. >>return
};
```

Осуществляется одновременное сравнение аргумента с нулем на «меньше» и «больше или равно». Размещение этих операций внутри круглых скобок позволяет получить список данных из двух булевских величин. Операция «?» обеспечивает формирование параллельного списка, состоящего из порядковых номеров элементов, имеющих истинное значение. В данной ситуации возможен только один вариант. Полученное значение используется для выбора одного из элементов списка. Использование задержанного списка позволяет предотваратить немедленное вычисление унарного минуса. Оно произойдет только в том случае, если проверяемый параметр окажется отрицательным числом. Задержка второй альтернативы не нужна, так как параметр представлен уже сформированным значением. Функция «.» используется в качестве пустой операции. Она «раскрывает» задержанные списки и пропускает без изменения прочие аргументы. Рассмотрим выполнение программы при обработке отрицательного аргумента, равного –5.

```
({-5:-}, -5):[(-5,0)(<,>=):?]:.
=> ({-5:-}, -5):[(true,false):?]:.
=> ({-5:-}, -5):[1]:.
=> {-5:-}:. => -5:- => 5
```

При положительном аргументе, например, равном 3, результат будет получаться иначе.

```
({3:-}, 3):[(3,0)(<,>=):?]:.
=> ({3:-}, 3):[(false,true):?]:.
=> ({3:-}, 3):[2]:.
=> 3:. => 3
```

Использование параллельной рекурсии

Если вычислительный алгоритм предусматривает древовидное или рекуррентное использование функции для множества однотипных аргументов, число которых может быть произвольным (например, суммирование всех элементов вектора), то применяется параллельная рекурсивная декомпозиция списка аргументов, на самом нижнем уровне которой выполняется операция над одноэлементными или двухэлементными списками, полученными в результате разложения. После этого следует обратный ход со сверткой возвращаемых результатов. Рассмотрим вычисление суммы элементов числового вектора произвольной длины.

```
VecSum << funcdef Param {                          // 1
// Формат аргумента: (x1,... ,xn), где x1,... ,xn - числа      2
Len<<Param:|;                                      // 3
return<< .^[((Len,2):[<,=,>]):?]^                  // 4
(                                                  // 5
{Param:[]},                                        // 6
{Param:+},                                         // 7
{                                                  // 8
block {                                            // 9
```

```
OddVec << Param:[(1,Len,2):..];                    // 10
EvenVec << Param:[(2,Len,2):..];                   // 11
([OddVec,EvenVec]: VecSum):+                       // 12
>>break                                  // 13
} // конец блока                              14
} // конец задержанного списка                    15
)                                        // 16
}                                        // 17
```

В строке 3 определяется длина вектора (функция «|»), что позволяет в дальнейшем выбрать один из трех вариантов вычислений. При одноэлементном векторе возвращается значение атома, размещенного в списке (строка 6). Для двух элементов в списке (строка 7) осуществляется их суммирование. При большей длине (строки 9-12) происходит разбиение вектора на два (состоящих из четных и нечетных элементов), для каждого из которых одновременно осуществляется рекурсивный вызов функции VecSum, и суммирование возвращаемых результатов (строка 12). Строка 4, записанная в префиксной форме, обеспечивает проверку длины, селекцию одного из вариантов, раскрытие задержанного списка и возврат результата. Конструкция block используется для группировки операторов. Пример выполнения:

(-3, 6, 10, 25, 0): VecSum => 38

Наряду с представленными возможностями допустимо использование функций в качестве аргументов, динамический контроль типов данных, перегрузка функций с одинаковой сигнатурой. Последнее обеспечивает эволюционное расширение программы без изменения уже написанного кода. Поддерживаются типы, динамически порождаемые пользователем.

Преобразования функциональных программ

Использование функционального языка изменяет не только стиль программирования, но и облегчает различные эквивалентные преобразования, что позволяет настраивать программу под конкретную архитекуру. В качестве примера рассмотрим векторное произведение, которое в последовательном программировании можно описать следующим псевдокодом:

c := 0;

для i от 1 до N выполнить c := c + A[i]*B[i];

Предполагается, что A и B - векторы длины N, c - скаляр.

Даже такой простой код реализуется совершенно по-иному в различных параллельных системах. Например, векторная ЭВМ осуществляет попарное перемножение элементов в конвейерном умножителе и их непосредственную передачу на сложение в конвейерный сумматор. Кластерная система на основе MPI обычно использует распределение векторов по всем узлам с последующим раздельным перемножением и суммированием подвекторов. Окончательное суммирование осуществляется на одном из узлов кластера. В связи с тем, что автоматическое распараллеливание возможно только в простейших случаях и для узкого круга архитектур, программистам каждый раз приходится перерабатывать программу. Ситуация еще больше усугубляется при увеличении сложности задачи. В [1] показано, что при использовании только MPI, достаточно простая задача перемножения двух прямоугольных матриц может быть запрограммированна

несколькими способами, эффективность применения каждого из которых зависит от размерности задачи и параметров подсистем кластера.

Ипользование функционального языка позволяет в большей степени сосредоточиться на проблемах конструирования каркаса программы, который может быть адаптирован к требованиям архитектуры путем дальнейших преобразований. Сама программа является предметом дальнейших исследований и преобразований. Например, векторное произведение первоначально может быть записано в виде следующей программы (с использованием уже представленных функций):

```
VecMult<< funcdef x {x:ScalMultVec:VecSum >>return}
```

В последующем возможны изменения этой программы с учетом эквивалентной реализации некоторых операций. В частности, параллельное суммирование VecSum можно заменить на последовательный аналог SeqVecSum, написанный следующим образом:

```
SeqVecSum << funcdef Param {
// Формат аргумента: (x1,... ,xn), где x1,... ,
xn - числа
Len<< Param:|;
return<< .^[((Len,2):[<,>=]):?]^
(
{Param:[]},
{(Param:1, Param:-1:SeqVecSum):+} //
 голова+Сумма(хвост)
)
}
```

Эквивалентные реализации одних и тех же функций можно связать с методами их выполнения на различных архитектурах. Это позволит осуществить автоматические подмены в процессе адаптации исходной программы перед генерацией окончательного кода. Использование функций высших порядков позволяет создавать не конкретные вычислительные процедуры, а обобщенные операции, определяющие принципы формирования вычислений для различных функций. Например, параллельное суммирование элементов вектора VecSum может рассматриваться как конкретная реализация бинарной древовидной свертки. Последовательное суммирование SeqVecSum - это пример правой свертки.

Инструментальные средства

Использование языка для построения параллельных программ требует адекватного инструментария: транслятора, последовательного интерпретатора с возможностью пошагового выполнения, интегрированной среды. При создании такого инструментария всегда надо помнить о необходимости поддержки реального, а не виртуального параллелизма. Среди существующих средств, обеспечивающих поддержку кластерных и распределенных вычислений, можно отметить систему динамического распараллеливания Mosix [3] и библиотеку MPI [1]. Обе поддерживают распараллеливание на уровне процессов, но при этом имеют свои характеристики, влияющие на реализацию параллельного интерпретатора. Например, в языке «Пифагор» требуется, чтобы исполнительная система эффективно и, по возможности, автоматически поддерживала видимость бесконечных ресурсов, обеспечивая при этом реальное сжатие параллелизма с учетом существующих ограничений.

При решении подобных задач вряд ли имеет смысл ориентироваться на системы статического распараллеливания. Несмотря на эффективность «жесткого планирования», они мало подходят для решения задач, в которых параллелизм описывается динамически. Поэтому для реализации интерпретатора была выбрана система Mosix, обеспечивающая автоматическое распределение процессов по узлам кластера. Порождение процессов осуществляется стандартными средствами ОС Linux, что позволяет запускать интерпретатор как на кластере, так и на однопроцессорных системах, не использующих данный пакет.

Следует учесть, что поддержка параллелизма на уровне отдельных операций не обеспечивает эффективного выполнения функциональных программ, поскольку время создания параллельного процесса для элементарных операций будет много больше времени ее обычного последовательного выполнения. Для повышения производительности требуется реализовать автоматическое ограничение числа порождаемых процессов с учетом особенностей кластерной системы.

Распараллеливание может быть установлено на уровне независимых наборов данных, которые должны быть параллельными списками. Функция, осуществляющая обработку этих данных, не должна быть предопределенной.
Распараллеливание проводится на уровне функций. Они должны быть представлены в виде параллельных списков, и не являться предопределенными.
Еще одним фактором, влияющим на выполнение функциональных программ, является глубина распараллеливания рекурсивных функций. Параллельная рекурсия будет эффективна в том случае, если число процессов, порожденных в ходе интерпретации, не будут превышать количества узлов кластера, а время выполнения каждого из этих процессов будет больше времени миграции порожденного процесса на свободный узел кластера. Параметр, ограничивающий глубину распараллеливания, может накладывать дополнительные ограничения на уровень распараллеливания.

Опциональный ввод параметров, ограничивающих распараллеливание и глубину параллельного выполнения рекурсивных функций до определенного уровня, позволяет дополнительно настраивать интерпретатор на основе информации, полученной в ходе анализа алгоритма решаемой задачи и текущей архитектуры кластера. Использование специального языка, опирающегося на функциональную модель параллельных вычислений, на наш взгляд, является основным фактором, необходимым для переноса параллельных программ на различные архитектуры.

Перечень сокращений

COM – Component Object Model (компонентная объектная модель).
АП – ассоциативная память.
АТЗ – аппаратная таблица задач.
ВСАРР – вычислительная система с автоматическим распределением ресурсов.
ИУ – исполнительное устройство.
МАП – модуль ассоциативной памяти.
ООЗУ – общее оперативное запоминающее устройство.
ПЛИС – программируемая логическая интегральная схема.
ПО – программное обеспечение.
ЦУУП – центральное устройство управления памятью.