МИНИСТЕРСТВО ОБЩЕГО И ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ РФ

ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Н. П. Вашкевич, Е. И. Калиниченко

ПРОЕКТИРОВАНИЕ ПАРАЛЛЕЛЬНЫХ АЛГОРИТМОВ В ЗАДАЧАХ ИДЕНТИФИКАЦИИ

Учебное пособие

ПЕНЗА 1999

УДК 519.713
В 23

Вашкевич Н.П., Калиниченко Е.И. Проектирование параллельных алгоритмов в задачах идентификации: Учеб. пособие. - Пенза: Пенз. государ. ун-т, 1999. - 80 с.: 18 ил., библиогр. 5 назв.

Рассматриваются вопросы решения задач распознавания цепочек - образов с использованием теорий регулярных выражений алгебры событий и недетерминированных автоматов. Приводятся примеры решения таких задач. Предлагается методика проектирования параллельных алгоритмов. Описывается инструментальная система, позволяющая автоматизировать разработку алгоритмов с последующей их верификацией.

Учебное пособие подготовлено на кафедре "Вычислительная техника" и предназначено для студентов специальности 2201 при изучении ими дисциплин "Теория автоматов", "Недетерминированные автоматы и их применение", "Системное программирование", выполнения курсового проектирования, а также может быть использовано студентами других специальностей при изучении дисциплин связанных с синтаксическим анализом и т.п.

Рецензент М. М. Бутаев, к.т.н., вед.н.с. ГНПП “ Рубин”

Предисловие

Задачи идентификации языков, реализуемые цифровым автоматом-распознавателем, имеют в настоящее время очень широкое применение. В частности, идентификация цепочек символов входит как составная часть во многие задачи, связанные с редактированием текстов, поиском данных и символьной обработкой. Многие программы для редактирования текстов разрешают пользователю задавать типы замен в цепочке - тексте. Например, пользователю необходимо заменить какое-то слово другим словом во всем тексте или его части. Чтобы выполнить такое действие, программа редактирования текста должна суметь найти вхождение слова и определить его местоположение. Некоторые искусные редактирующие программы разрешают пользователю в качестве множества заменяемых цепочек символов указывать *регулярное множество* [1]. Например, пользователь может поставить задачу: "Заменить [Z*] в тексте W пустой цепочкой", имея в виду, что в W следует стереть пару квадратных скобок и символы между ними. Антивирусной программе, для обнаружения "простых" вирусов необходимо найти последовательность байт (сигнатуру), а при поиске "полиморфных" вирусов обнаружение сигнатуры может входить как одна из составляющих технологии поиска.

В данном пособии приводятся методы и примеры решения таких задач. При проверке алгоритмов использовалась система "СОМПА", в разработке которой принимали самое активное участие студенты Синев С. А. (гр. 95В1), Антонов А. В., Токарев А. Н. (гр. 96ВВ1). Использованы также результаты курсовой работы студента Евдокимова А.С. (гр. 96ВВ3).

1 Задачи идентификации

В типичной задаче идентификации цепочек -образов - задаются входная последовательность W (например, символов или пронумерованные фрагменты графических файлов и т.п.)*)* и множество цепочек-образов $\{z_1, z_2, \ldots z_n\}$. Требуется найти либо хотя бы одно вхождение какой-то цепочки-образа, либо m из n ($1 \le m < n$), либо вхождения всех n цепочек-образов в W. При этом множество цепочек-образов часто является регулярным множеством, которое можно задать регулярным выражением алгебры событий. В главе 1 будут показаны приемы решения такого рода задач

1.1 Язык регулярных выражений алгебры событий и недетерминированные конечные автоматы

Поскольку предлагаемое решение таких задач основано на использовании языка регулярных выражений алгебры событий (РВАС) и модели недетерминированного конечного автомата (НДКА), то далее будут приведены основные положения этих теорий, а более подробный материал по РВАС и НДКА приведен в [1].

Следуя [1], приведем некоторые понятия и определения. Событие в любом алфавите *Z* есть произвольное множество слов в этом алфавите. Элементарными событиями в алфавите $Z=[z_1,z_2, \ldots ,z_n]$ называются события, состоящие из одной любой буквы алфавита Z, и событие s=e*,* где e -пустое слово. Под алгеброй событий в алфавите Z понимается множество всех событий в этом алфавите, на котором определены две двухместные операции: сцепление (конкатенация, умножение) и объединение (дизъюнкция), а также одна одноместная операция, называемая итерацией. Сцепление двух событий $s_1 \bullet s_2$ это событие, состоящее из всех слов вида $k_1 \bullet k_2$, где k_1 любое слово из s_1, а k_2 любое слово события s_2. При этом $s_1 \bullet s_2 \neq s_2 \bullet s_1$. В дальнейшем знак сцепления событий. ($\bullet$) для обозначения этой операции будем опускать. Дизъюнкция двух событий $s_1 \vee s_2$ это теоретико - множественное объединение этих событий. Итерация события s **,** записываемая в виде { s

}, -это событие, состоящее из пустого слова e и всевозможных слов вида: s, ss, и т.д., т.е. $\{s\}=e \vee s \vee ss \vee sss \vee ...$

В алгебре событий, при отсутствии круглых скобок изменяющих порядок действий, установлен следующий приоритет операций в порядке убывания: итерация, конкатенация и потом дизъюнкция. Регулярное событие это событие, которое можно получить из элементарных событий в результате применения к ним конечного числа раз основных операций алгебры событий.

Теперь определим модель цифрового автомата - распознавателя, для которого будут разрабатываться алгоритмы решения задач идентификации цепочек - образов [4]. *Детерминированный конечный автомат - распознаватель* (ДКА) будем рассматривать как устройство, которое в каждый конкретный момент времени находится в одном из конечного множества состояний, и входной ленты (входная последовательность) состоящей из клеток, которая просматривается слева направо блоком чтения этого устройства. Каждый шаг алгоритма распознавания ДКА состоит из перехода в новое состояние, определяемое его текущим состоянием и читаемым входным сигналом, и сдвига блока чтения на одну клетку вправо. Оказывается, что любой язык представим регулярным выражением тогда и только тогда, когда он допускается некоторым конечным автоматом.

Если решается задача только обнаружения наличия цепочки - образа, то достаточно модели *одностороннего автомата,* в которой на каждом шаге алгоритма блок чтения может перемещаться на одну клетку только слева направо. Если же решается задача обнаружения местоположения цепочки - образа, то необходимо использовать модель *двухстороннего автомата,* в которой блок чтения может перемещаться или слева направо или справа налево. Лента, которую читает автомат, должна содержать два специальных символа: "начало" (top) и "конец" (bottom), при достижении которых передвижение головки соответственно влево и вправо невозможно, о чем сообщается выработкой специальных сигналов "*достижение начала"*(y_t)*/"достижение конца*"(y_b).

Важным обобщением понятия ДКА является НДКА. Для каждого состояния и каждого входного сигнала функция переходов НДКА имеет нуль или более вариантов выбора следующего шага алгоритма. В [1,2] предлагается аналитический язык задания НДКА, получивший название языка систем канонических уравнений и систем выходных функций (СКУ и СВФ). Там же предлагается способ перехода от РВАС к СКУ и СВФ. Эта связка двух языков позволяет снять ограничения языка РВАС при формализации ниже перечисленных задач идентификации. Например, хотя алгоритм идентификации задачи типа "1" (смотри ниже) практически напрямую формализуется на языке РВАС, переход к его программной или аппаратной реализации мягко говоря неочевиден.

Основным достоинством языка СКУ и СВФ является то, что алгоритм записанный на этом языке представляет готовое решение для программной или аппаратной реализации, и что самое главное, автором языка предложен простой и эффективный способ перехода от РВАС к СКУ и СВФ. Алгоритм построения СКУ по регулярным выражениям исходной системы событий содержит следующие этапы:

1) В каждом из регулярных выражений исходной системы событий, отмеченных соответствующими выходными сигналами, выделяют параллельные ветви в отдельные регулярные выражения. Получается система выражений, в каждом из которых в правой части (за исключением содержимого итерационных скобок) есть только операции сцепления и итерации

2) Для каждой параллельной ветви (если она заканчивается итерационной скобкой) каждого из регулярных выражений системы событий выполняют операцию замены итерационных скобок вспомогательными переменными, которые будут, играть роль дополнительных входных сигналов.

3) Каждое регулярное выражение системы преобразуется в систему рекуррентных регулярных соотношений вида $s_i=s_jz_k$, где s_j событие заменяющее всю часть выражения за исключением самого правого входного сигнала.

4) Если самый правый входной сигнал - вспомогательная переменная из п.2, то выполняют ее обратную замену на итерационную скобку, которую затем раскрывают с использованием следующего соотношения: если $P=S\{Q\}$, то $P=S \vee PQ$.

5) Если содержимое раскрываемой итерации в п.4 не одноэлементное событие, а сложное выражения включающее в том числе и внутренние итерационные скобки, то для этого события повторяют все операции, предусмотренные в пунктах 1-5 до тех пор, пока не будут раскрыты все итерационные скобки.

6) В итоге каждое полученное уравнение в правой части должно содержать сцепление события и входного сигнала или дизъюнкцию таких сцеплений. В тех случаях, если в процессе преобразований, в правых частях уравнений будут встречаться события без входного сигнала, необходимо сделать подстановку вместо таких событий соответствующих им выражений.

7) В итоге заменяя во всех уравнениях знаки конкатенации на знаки конъюнкции и, вводя дискретное время, получают СКУ, в которой обозначения событий исходной системы отмечаются соответствующими выходными сигналами.

Пример на построение СКУ и СВФ по регулярному выражению приведен далее в разделе 1.3.

1.2 Формулировка задач распознавания цепочек - образов

Рассмотрим более детально формулировку задач распознавания цепочек - образов, в том числе описываемых регулярными выражениями. Пусть дана входная цепочка-последовательность следующего вида $W=topw_1w_2 \ldots w_nbottom$ (w_i - любой символ из алфавита Z) и цепочка-образ, заданная регулярным выражением $R=z_gz_m \ldots z_n$. Надо найти такой наименьший индекс j символа входной последовательности, что для некоторого i подцепочка $w_iw_{i+1}...w_j$, цепочки W принадлежит языку, представленному выражением R [4].

В зависимости от того, какая цель преследуется в результате поиска можно выделить следующие классы задач. ***Обнаружение факта наличия цепочки***, и только. Напри-

мер, наличие в тексте слова или предложения; обнаружение сигнатуры вируса при поиске в файле и т.п. В этом случае достаточно реализовать алгоритм поиска с использованием модели одностороннего автомата, когда при обнаружении цепочки вырабатывается выходной сигнал индицирующий положительный результат поиска. ***Обнаружение факта наличия цепочки и локализация места*** ее расположения, например найти в тексте фразу и заменить ее на другую или обнаружить вирус и "вылечить". Для таких случаев необходимо реализация алгоритма поиска и локализации двухсторонним автоматом, когда при обнаружении цепочки для локализации ее места положения блок чтения автомата начинает перемещаться справа налево (при этом запоминаются позиции ленты) до момента обнаружения начала цепочки. При программной реализации, если этот поиск реализуется как один процесс, после того как обнаружена цепочка - образ блок чтения начинает перемещаться в обратном направлении для поиска начала цепочки. Если запомнить при этом позицию конца цепочки то затем блок чтения можно сразу переместить вправо на позицию следующую за ее концом. Если же операционная система позволяет параллельно выполнить несколько процессов, то тогда можно запустить еще один процесс для локализации цепочки, в то время как родительский продолжает просматривать входную последовательность. Алгоритм работы автомата в прямом направлении реализуется на основе РВАС, а алгоритм работы в обратном на основе "зеркальной" РВАС.

Необходимо отметить что задачи распознавания образов не сводятся только к поиску одной единственной цепочки во входной последовательности. Приведем формулировку таких задач, которые встречаются на практике.

1) ***Поиск любой из нескольких цепочек***, т.е. хотя бы одна из цепочек есть во входной последовательности. Вообще казалось бы эту задачу можно свести к поочередному поиску каждой цепочки за несколько просмотров входной последовательности, но в этом случае время поиска возрастает пропорционально количеству цепочек и их длине, а в некоторых случаях, например "поиск на лету" (без возможности повторных просмотров входной последовательности) он в принципе становится невозможным.

2) ***Поиск наличия*** всех ***заданных цепочек***, т.е. во входной последовательности должны быть все из искомых цепочек до того как на вход автомата поступит "bottom"

3) ***Поиск заданного количества из всех искомых цепочек*** т.е. например ищутся 5 цепочек, а результат поиска может быть положительным, если будет найдены хотя бы 3 из них. Здесь дополнительная трудность при поиске состоит в том, что заранее невозможно определить порядок, в котором они будут находится во входной последовательности. Если же искать их последовательно друг за другом то из-за этого возможны существенные потери времени.

4) ***Поиск*** нескольких ***цепочек, у которых префикс и суффикс одинаковы, а корни различны***. Если префикс обозначить через A, суффикс через C, а корни через B_1,B_2,B_3, то на языке РВАС алгоритм поиска запишется как: $r(y)=A\&(B_1 \vee B_2 \vee B_3)\&C$. Можно,

конечно, выполнить преобразование, раскрыв скобки, но тогда суффиксы и префиксы уйдут в каждую из трех цепочек, что увеличит размерность алгоритма.

5) ***Поиск цепочек, когда во входной последовательности некоторые символы являются "прозрачными"*** (т.е. их как бы нет). Например, игнорирование знаков препинания в тексте или определенных байтовых комбинаций в двоичных файлах.

6) ***Когда поиск во входной последовательности начинается по сигналу "извне"***, а до того времени на каждом шаге осуществляется просто прием очередного символа входной последовательности. Например, когда поиском управляет внешняя среда или когда нужно найти цепочку после пропуска определенного количества символов входной последовательности.

Отметим, что начальное состояние s_0 НДКА, которое назовем ***маркером начала поиска*** имеет очень существенное значение. Маркер позволяет динамически управлять процессом поиска без изменения самого алгоритма поиска (программная или аппаратная реализация не меняется), что обеспечивает гибкость при реализации алгоритма. Например, не меняя алгоритма, а управляя только маркером можно получить решение следующих подзадач, которые могут являться составной частью задач 1-6.

1) Не начинается ли входная последовательность с искомых цепочек, т е. самый первый символ последовательности совпадает с первым символом цепочки. Это может потребоваться если производится такой поиск во многих файлах или даже в том случае если это поиск в одном файле (который в принципе можно сделать вручную), но цепочка достаточно длинная (сотни и более байт) и тогда возможна ошибка из-за человеческого фактора. Такая задача решается установкой маркера начала поиска ($s_0=1$) перед приемом входной последовательности и сбросом его ($s_0=0$) после приема первого символа.

2) Классическая задача поиска, когда входная последовательность может содержать цепочки в любом месте между "top" и "bottom". Эта задача решается установкой маркера ($s_0=1$) перед приемом входной последовательности, без последующего сброса (поиск искомых цепочек начинается с каждого очередного символа).

3) Поиск во входной последовательности начинается после поступления в ней определенного символа или символов (точки, отмечающей конец предложения и т.п.). Эта задача решается установкой маркера после поступления этого символа, без последующего сброса.

4) Поиск во входной последовательности начинается после поступления сигнала начать поиск "извне" или наступлении определенного события "внутри" (например, прочитано определенное количество символов в последовательности и т.п.)

Рассмотрим как можно решить задачи 1-5 и подзадачи 1-4. Будем рассматривать задачи идентификации только на обнаружение, т.к. решение задач локализации полностью идентично, только с "зеркальным" отражением.

1.3 Разработка алгоритмов задач идентификации

Решение задачи типа "1". Алгоритм решения этой задачи (без учета поступления символа "bottom") можно непосредственно записать на языке РВАС. Например необходимо определить есть ли во входной последовательности любая из трех следующих цепочек:

1) $z_3z_2z_1 \dots z_1z_3$; 2) $z_2 \dots z_2z_1z_3$; 3) $z_4 \dots z_4z_2 \dots z_2z_3z_1 \dots z_1$,

где $z_i \dots z_i$ повторение символа z_i любое число раз, но не менее одного. При обнаружении любой цепочки должен быть выработан выходной сигнал y.

Тогда на языке РВАС алгоритм поиска запишется:

$r(y)=s_0(z_3z_2z_1\{z_1\}z_3 \vee z_2\{z_2\}z_1z_3 \vee z_4\{z_4\}z_2\{z_2\}z_3z_1\{z_1\})$

Для перехода от РВАС к СКУ и СВФ в соответствие с алгоритмом приведенным в 1.1, преобразуем выражение r в следующую систему из трех уравнений, где s_{k1}, s_{k2}, s_{k3} события, появляющиеся при обнаружении цепочек 1, 2, 3 соответственно:

$s_{k1}(y)=s_0z_3z_2z_1\{z_1\}z_3$

$s_{k2}(y)=s_0z_2\{z_2\}z_1z_3$

$s_{k3}(y)=s_0z_4\{z_4\}z_2\{z_2\}z_3z_1\{z_1\}$.

Дальнейшие преобразования дадут следующие СКУ и СВФ:

СКУ:

$s_1=s_2\&z_1 \vee s_1\&z_1$; $s_2=s_3\&z_2$; $s_3=s_0\&z_3$; $s_{k1}=s_1\&z_3$;

$s_4=s_5\&z_1$; $s_5=s_0\&z_2 \vee s_5\&z_2$; $s_{k2}=s_4\&z_3$;

$s_6=s_7\&z_3$; $s_7=s_8\&z_2 \vee s_7\&z_2$; $s_8=s_0\&z_4 \vee s_8\&z_4$;

$s_{k3}=s_6\&z_1 \vee s_{k3}\&z_1$;

СВФ:

$y=s_{k1} \vee s_{k2} \vee s_{k3}$

Для поиска цепочек в любом месте входной последовательности необходимо устанавливать маркер начала поиска перед приемом каждого следующего символа. Поэтому в СКУ нужно добавить уравнение: $s_0=1$, а чтобы учесть поступление символа "bottom" необходимо добавить еще одно уравнение в СКУ и одно в СВФ:

СКУ -

$s_0=1$

$s_b=s_0\&bottom \vee s_b$;

СВФ - $y_b=s_b$.

При практической реализации алгоритма необязательно повторять переходы из s_b в s_b. Для программной реализации достаточно будет остановить выполнение программы, а при аппаратной реализации блокировать прохождение синхросигнала на элементы памяти.

Решение задачи типа "2". Получить алгоритм решения этой задачи возможно следующим образом:

- на языке РВАС записывается алгоритм поиска любой цепочки, как и в предыдущей задаче;

- по РВАС строится СКУ и СВФ;

- СКУ изменяется (корректируется) так, что событие возникающее при обнаружении какой-либо цепочки устанавливается и остается при каждом следующем поступлении входного символа (тактовом сигнале).

- поиск завершается успешно, если конъюнкция всех таких событий становится равной 1 (все цепочки найдены) и неуспешно если до этого поступит символ "bottom".

Используем тот же пример что и в задаче 1, только теперь все искомые цепочки должны присутствовать во входной последовательности.

СКУ:

$s_0=1$; $s_1=s_2\&z_1 \vee s_1\&z_1$; $s_2=s_3\&z_2$; $s_3=s_0\&z_3$; $s_{k1}=s_1\&z_3 \vee \underline{\mathbf{s_{k1}}}$;

$s_4=s_5\&z_1$; $s_5=s_0\&z_2 \vee s_5\&z_2$; $s_{k2}=s_4\&z_3 \vee \underline{\mathbf{s_{k2}}}$;

$s_6=s_7\&z_3$; $_7=s_8\&z_2 \vee s_7\&z_2$; $s_8=s_0\&z_4 \vee s_8\&z_4$;

$s_{k3}=s_6\&z_1 \vee s_{k3}\&z_1 \vee \underline{\mathbf{s_{k3}}}$;

$s_b=s_0\&\text{bottom} \vee s_b$;

СВФ:

$y=s_{k1}\&s_{k2}\&s_{k3}$; $y_b=s_b$.

Решение задачи типа "3". При разработке алгоритма решения этой задачи наряду с замечаниями к задаче "2" нужно еще учесть, что нельзя заранее знать порядок , в котором появятся цепочки во входной последовательности. Используем тот же пример что и в задаче 1, только теперь необходимо найти хотя бы две цепочки из трех.

При переходе от РВАС к СКУ и СВФ проделаем тоже, что и в задаче 2, но по иному выполним коррекцию СВФ.

СКУ:

$s_0=1$;

$s_1=s_2\&z_1 \vee s_1\&z_1$; $s_2=s_3\&z_2$; $s_3=s_0\&z_3$; $s_{k1}=s_1\&z_3 \vee s_{k1}$;

$s_4=s_5\&z_1$; $s_5=s_0\&z_2 \vee s_5\&z_2$; $s_{k2}=s_4\&z_3 \vee s_{k2}$;

$s_6=s_7\&z_3$; $s_7=s_8\&z_2 \vee s_7\&z_2$; $s_8=s_0\&z_4 \vee s_8\&z_4$;

$s_{k3}=s_6\&z_1 \vee s_{k3}\&z_1 \vee s_{k3}$;

$s_b=s_0\&\text{bottom} \vee s_b$;

СВФ:

$y=\underline{\mathbf{s_{k1}\&s_{k2} \vee s_{k1}\&s_{k3} \vee s_{k2}\&s_{k3}}}$;

$y_b=s_b$.

Решение задачи типа "4". Рассмотрим решение такой задачи когда префикс будет задан регулярным выражением $z_1z_1z_2z_3$, суффикс - $z_2\{z_1\}z_3$, а корни будут из задачи 1. Необходимо найти во входной последовательности любую цепочку из трех. Тогда алгоритм поиска на языке РВАС запишется:

$r(y)=s_0z_1z_1z_2z_3(z_3z_2z_1\{z_1\}z_3 \vee z_2\{z_2\}z_1z_3 \vee z_4\{z_4\}z_2\{z_2\}z_3z_1\{z_1\})z_2\{z_1\}z_3$.

Разобьем выражение r на три части:

$s_p=s_0z_1z_1z_2z_3$- для префикса,

$s_{к1}=s_p(z_3z_2z_1\{z_1\}z_3)$ - для корня 1,

$s_{к2}=s_p(z_2\{z_2\}z_1z_3)$ - для корня 2
$s_{к3}=s_p(z_4\{z_4\}z_2\{z_2\}z_3z_1\{z_1\})$ - для корня 3
$r(y)=s_rz_2\{z_1\}z_3$ - для суффикса.
Для каждого уравнения перейдем от РВАС к СКУ и объединим их. Получим:
СКУ:
$s_0=1$
Для префикса (маркер начала поиска s_0):
$s_p=s_9\&z_3$; $s_9=s_{10}\&z_2$; $s_{10}=s_{11}\&z_1$; $s_{11}=s_0\&z_1$;
Для корней (маркер начала поиска s_p):
$s_1=s_2\&z_1 \vee s_1\&z_1$; $s_2=s_3\&z_2$; $s_3=s_p\&z_3$; $s_{k1}=s_1\&z_3$;
$s_4=s_5\&z_1$; $s_5=s_p\&z_2 \vee s_5\&z_2$; $s_{k2}=s_4\&z_3$;
$s_6=s_7\&z_3$; $s_7=s_8\&z_2 \vee s_7\&z_2$; $s_8=s_p\&z_4 \vee s_8\&z_4$;
$s_{k3}=s_6\&z_1 \vee s_{k3}\&z_1$;
Для суффикса (маркеры начала поиска s_{k1}, s_{k2}, s_{k3}):
$r=s_{12}\&z_3$; $s_{12}=(s_{k1} \vee s_{k2} \vee s_{k3})\&z_2 \vee s_{12}\&z_1$;
Для входного символа "bottom":
$s_b=s_0\&bottom \vee s_b$;
СВФ:
$y=r$; $y_b=s_b$.

Решение задачи типа "5" Рассмотрим решение такой задачи на примере, когда искомая цепочка будет $z_1z_1z_2z_3$, в алфавите $Z=\{z_1,z_2,z_3,z_4,z_5,z_6,z_7,z_8,z_9,z_{10}\}$, а игнорируемый символ при поиске z_{10}. Тогда цепочка во входной последовательности $z_1z_{10}z_{10}z_1z_2z_{10}z_3$ должна быть обнаружена.

Получить алгоритм решения этой задачи возможно следующим образом:

- на языке РВАС записывается алгоритм поиска цепочки

- по РВАС строится СКУ и СВФ;

- СКУ изменяется (корректируется) так, что алгоритма не меняет своего состояния при поступлении на вход z_{10} (символ игнорируется и как бы "прозрачен" для цепочки).

Алгоритм поиска на языке РВАС запишется:
$r(y)=s_0z_1z_1z_2z_3$.
Перейдем от РВАС к СКУ и СВФ и получим:
СКУ:
$s_0=1$; $r=s_1\&z_3$; $s_1=s_2\&z_2$; $s_2=s_3\&z_1$; $s_3=s_0\&z_1$;
$s_b=s_0\&bottom \vee s_b$
СВФ:
$y=r$; $y_b=s_b$.
Корректируем СКУ и получаем:
$s_0=1$; $r=s_1\&z_3 \vee \mathbf{r\&z_{10}}$; $s_1=s_2\&z_2 \vee \mathbf{s_1\&z_{10}}$; $s_2=s_3\&z_1 \vee \mathbf{s_2\&z_{10}}$;
$s_3=s_0\&z_1 \vee \mathbf{s_3\&z_{10}}$; $s_b=s_0\&bottom \vee s_b$.

Решение задачи типа "6". При решение этой задачи алгоритм разрабатывается в том же порядке как и в задачах 1-5.
Но в СКУ добавляется $s_0=0$. Поэтому все символы во входной последовательности игнорируются до тех пока по сигналу "извне" не установится маркер начала поиска ($s_0=1$).

1.4 Выводы

Следует отметить, что хотя разработка алгоритмов решения задач поиска велась для однопроцессорной системы (последовательная реализация алгоритма), полученные алгоритмы внутри являются параллельными.

Но если задача поиска имеет большую размерность (длинные цепочки и много файлов большой длины, внутри которых их надо искать), и есть возможность реализации алгоритма на мультипроцессорной системе, то имеет смысл с точки зрения повышения быстродействия произвести разработку параллельного алгоритма.

2 Методика проектирования параллельных алгоритмов

Одна из важных тенденций развития современной вычислительной техники состоит в расширенном внедрении мультипроцессорных архитектур. Микропроцессоры позволили получить процессорное время по номинальной цене, а их малые габариты делают вполне обоснованной постановку вопроса об их комплексировании в единой системе в количестве десятков, сотен и более. Одним из ключевых факторов способствующих использованию таких систем будет наличие параллельных алгоритмов.

Один из возможных подходов к разработке параллельного алгоритма для решения какой-либо задачи, в том числе задач идентификации базируется на методике, приведенной в [5] и состоит из следующих этапов проектирования: - декомпозиция (разбиение) исходной задачи на элементарные задачи; - определение всех необходимых взаимодействий (коммуникаций) между элементарными задачами; - объединение элементарных задач (агломерация) в соответствие с определенной стратегией, учитывающей в том числе число процессоров в системе; - распределение полученных при агломерации задач на заданное число процессоров; - оценка полученного алгоритма (или алгоритмов, если их получено несколько) на масштабируемость (при увеличении объема исходных данных и/или изменении числа процессоров в мультипроцессорной системе).

2.1 Декомпозиция задачи на элементарные подзадачи

На первом этапе проектирования данные, используемые в задаче и операции над ними, разбиваются на элементарные задачи (не допускающие дальнейшего разбиения) без учета числа процессоров в мультипроцессорной системе. То есть внимание фокусируется на достижении максимального параллелизма как при разбиении данных, исходных и/или промежуточных (***"декомпозиция данных"***), так и при разбиении выполняемых операций в задаче(***"функциональная декомпозиция"***).

При декомпозиции данных входные и промежуточные данные, используемые в задаче разбиваются на элементарные части приблизительно одинакового размера, а затем по этим частям разбиваются операции, выполняемые над ними. Эти данные и операции над ними назовем элементарными задачами (ЭЗ). Нужно учитывать при этом, что если какие-то операции в ЭЗ требуют данных из нескольких полученных при разбиении ЭЗ, то в дальнейшем это потребует организации взаимосвязей (коммуникаций) между ЭЗ на что потребуется время мультипроцессорной системы (это время может занимать существенный процент от общего времени выполнения алгоритма). Идеальным можно считать случай когда коммуникации между ЭЗ отсутствуют. Существует хорошее правило при декомпозиции данных - начинать разбиение с больших структур данных или с тех, которые используются наиболее часто в задаче.

При функциональной декомпозиции разбиение операций, выполняемых над данными, рассматривается раньше чем разбиение самих данных. Задача также разбивается на элементарные ***задачи - функции*** в идеальном случае не зависящие друг от друга.

На этом этапе проектирования для достижения максимального параллелизма необходимо выполнить как декомпозицию данных, так и функциональную декомпозицию. Поскольку декомпозиция данных и функциональная декомпозиция являются интуитивными, то их результаты необходимо сверить со следующими контрольными вопросами, позволяющими оценить качество разбиения.

1) Полученное число ЭЗ меньше или равно числу процессоров в системе? Если да, то на последующих этапах проектирования возможна малая гибкость в принятии решений.

2) Позволяет ли полученная декомпозиция избежать чрезмерного количества операций и/или чрезмерных требований по затратам памяти?. Если нет то результирующий параллельный алгоритм не может быть применим при увеличении входной размерности данных (в литературе используется понятие "алгоритм не масштабируем").

3) Одинакового ли размера полученные задачи? Если нет, то при дальнейшем проектировании будет трудно обеспечить равномерную загрузку процессоров системы.

4) Число ЭЗ пропорционально размеру исходной задачи? В идеальном случае необходимо, чтобы увеличение размерности исходной задачи увеличивало число ЭЗ значительно быстрее, чем размер самой ЭЗ. Если этого нет, то параллельный алгоритм не пригоден при росте размерности задачи и увеличения числа процессоров в системе.

5) Есть ли наличие альтернативных вариантов декомпозиции. Это может обеспечить большее число вариантов на последующих шагах проектирования.

2.2 Необходимые взаимодействия (коммуникации) между элементарными задачами

На втором этапе проектирования определяются необходимые взаимодействия (коммуникации) между ЭЗ. ЭЗ, полученные при разбиении, могут выполняться параллельно, но как правило, являются взаимозависимыми. Операции одной ЭЗ в какой-то момент времени требуют данных от другой ЭЗ, для чего необходима пересылка данных между задачами. Эти информационные потоки определяются на этапе проектирования коммуникаций.

Для спецификации коммуникаций определяются каналы от производителя (кто посылает) к потребителю (кто принимает) и определяются сообщения которые пересылаются по этим каналам.

При декомпозиции данных определение коммуникаций между ЭЗ не является простой задачей, так как существует достаточно много вариантов из которых надо выбрать наиболее оптимальный.

Наоборот при функциональной декомпозиции определение коммуникаций достаточно простая задача, так как коммуникации соответствуют потоку данных между ЭЗ.

В общем случае можно произвести классификацию коммуникаций по следующим признакам: - локальная или глобальная; -структурированная (или же регулярная) или

неструктурированная (нерегулярная); - статическая или динамическая; - синхронная или асинхронная.

При *локальной* коммуникации каждая ЭЗ взаимодействует с небольшим числом задач (ее "соседями"), а при *глобальной* со многими задачами (в пределе с каждой). При *структурированной* коммуникации каждая ЭЗ и ее соседи образуют регулярную структуру (например "дерево" или "сетка"), а при *неструктурированной* нерегулярную структуру (произвольный граф). При *статической* коммуникации взаимосвязи определяются "раз и навсегда" и не меняются при выполнении алгоритма, а при *динамической* могут меняться. При *синхронной* коммуникации есть синхронизирующие сигналы между производителем и потребителем, а при *асинхронной* нет.

Результаты разработки коммуникаций также необходимо сверить со следующими контрольными вопросами для оценки их качества.

1) Все ли элементарные задачи выполняют одинаковые коммуникации? Если нет, то такая несбалансированность дает немасштабируемую конструкцию алгоритма. В этом случае надо пересмотреть коммуникации. Например, если часто используемые данные включить (инкапсулировать) в ЭЗ, то тогда можно рассмотреть возможность распределения или размножения этих данных.

2) Каждая элементарная задача связана с небольшим числом соседей? Если нет, то следует исследовать возможность преобразования глобальных коммуникаций в локальные.

3) Все ли коммуникации могут выполняться параллельно (одновременно)? Если нет, то алгоритм не масштабируем и не эффективен. В этом случае можно использовать технологию “разделяй и властвуй” для создания параллельности.

4) Могут ли вычисления связанные с различными ЭЗ выполняться параллельно? Если нет, то алгоритм также неэффективен и не масштабируем. Для устранения этого необходимо посмотреть, как переупорядочить операции и коммуникации, может быть даже вернувшись к постановке задачи.

2.3 Выбор модели аппаратной реализации параллельных вычислений

На этом этапе необходимо определить наиболее подходящую *модель аппаратной реализации* параллельных алгоритмов из числа следующих: - сетка процессоров; - объединение процессоров через общую шину; - взаимодействие процессоров через общую память.

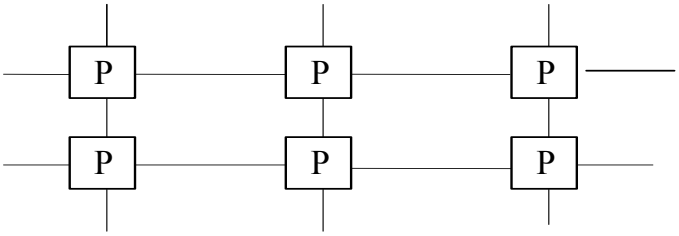


Рис. 2.1. Сетка процессоров

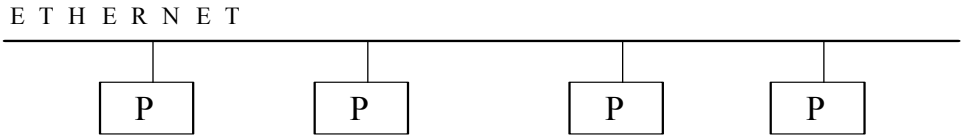


Рис. 2.2. Объединение процессоров через шину

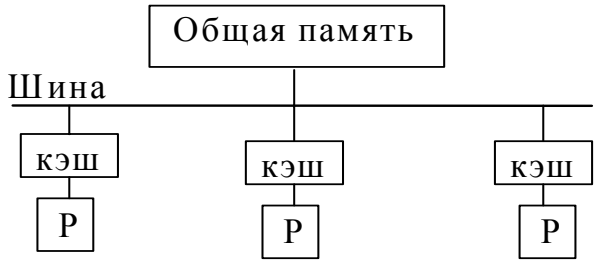
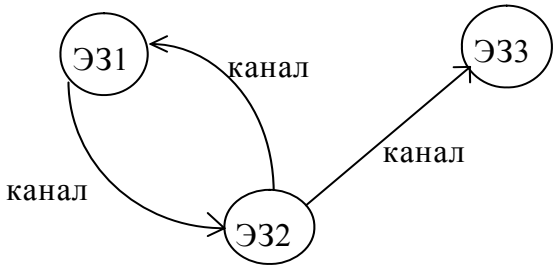


Рис. 2.3. Взаимодействие процессоров через общую память

2.4 Выбор модели программной реализации параллельных вычислений

На этом этапе необходимо определить наиболее подходящую **модель программной реализации** параллельных вычислений (задачи и каналы, пересылка сообщений с использованием стандартных библиотечных утилит, разделяемая память с использованием блокировок и семафоров).



Задачи и каналы

Пересылка сообщений с использованием стандартных библиотечных утилит во многом аналогична модели «задачи и каналы», но для посылки и приема сообщений используются вызовы библиотечных утилит, т.е. есть существуют определенные ограничения на пересылаемые и получаемые сообщения по каналам.

В модели разделяемой памяти с использованием блокировок и семафоров задачи используют общее адресное пространство, в котором они читают и записывают данные

асинхронно. Механизм блокировок и семафоров используется для контроля доступа к общей памяти

2.5 Объединение (агломерация) элементарных задач

На этом этапе проектирования производится объединение (укрупнение) ЭЗ. После этапов разбиения и определения коммуникаций алгоритм, состоящий из ЭЗ является абстрактным в том смысле, что не может быть реализован на любой мультипроцессорной системе. Например он может быть малоприемлем, если создает очень много задач малого размера, а процессоры входящие в систему неэффективны при выполнении таких задач.

Поэтому на этой стадии просматриваются решения принятые на этапах разбиения и определения коммуникаций с целью получения параллельного алгоритма, который эффективно выполнялся бы на некотором классе мультипроцессорных систем. Нужно рассмотреть где полезно скомбинировать задачи полученные на этапе разбиения ***так, чтобы обеспечить в результате наименьшее число задач, каждая из которых имеет наибольший размер***.

Также на этом этапе нужно оценить где при выполнении алгоритма заслуживает внимания размножение данных и/или операций с целью получения конечного результата за минимальное время. При агломерации получаемое число задач сокращается (в идеальном случае одна задача на процессор), но еще может быть больше числа процессоров в системе. В этом случае проектирование предусматривает следующий шаг - распределение задач по процессорам. Соответственно на этапе агломерации нужно стремиться, чтобы в результате была одна задача на процессор.

Три, как правило, взаимоконфликтующих цели преследуются при агломерации и размножении данных и/или операций:

- сокращение стоимости коммуникаций за счет объединения взаимосвязанных ЭЗ (локализация коммуникаций);
- сохранение гибкости по отношению к масштабируемости (изменению размерности задачи) и распределению задач по процессорам;
- сокращении стоимости инженерной разработки программного обеспечения (софта).

Рассмотрим каждую из них.

Локализация коммуникаций. На этапе разбиения алгоритма на ЭЗ усилия сосредотачивались на получении как можно большего числа задач. Это полезно, так как позволяет рассмотреть широкий круг возможностей удобных для параллельного выполнения. Однако, следует заметить, что получение большого числа задач при разбиении не обязательно дает эффективный параллельный алгоритм. Самое негативное влияние на выполнение параллельного алгоритма (по времени), оказывают коммуникации. На большинстве параллельных систем приходится останавливать вычисления чтобы послать или принять сообщения. Поэтому значительно улучшится скорость вычисления, если сократить время на коммуникации. Ясно, что это можно достичь пересылкой меньшего объема данных между задачами, выполняющимися параллельно. Возможно менее очевидно, но сократить время на коммуникации можно и при использовании меньшего числа сообщений при тех же объемах данных. Это объясняется тем, что каждая коммуникация требует время не только пропорциональное объему пересылаемых данных, но имеет фиксированные начальные затраты времени по ее установлению. В дополнение к коммуникационным

временным затратам нужно учитывать и стоимость создания задачи на процессоре (выделение памяти и т.д.).

Исходя из вышесказанного, объединение взаимодействующих задач в одну устраняет необходимость их внешних коммуникаций, так как полученная задача будет выполняться на одном процессоре.

Можно отметить, что при разбиении по данным эффективен подход увеличения локальности за счет увеличении размерности "куска", обрабатываемых регулярных данных (векторы, матрицы и т.п.). Но если данные нерегулярны то задача агломерации становится достаточно трудной.

Можно попытаться использовать размножение данных и/или операций для сокращения коммуникаций и/или для сокращения времени вычисления.

***Сохранение гибкости*.** Хорошо разработанный параллельный алгоритм должен быть гибким по отношению к числу процессоров в системе, что повышает ее надежность в целом. Эта гибкость также полезна и когда создается код для отдельного процессора. Если задачи часто блокируются в ожидании удаленных данных это может быть благоприятным при назначении нескольких таких задач на один процессор, с тем что пока одна задача ожидает данных другая может выполняться на ее месте. И тогда время коммуникации перекрывается вычислением т.е. процессор не простаивает (техника перекрытий вычислений и коммуникаций). Третья полезность создания большего числа задач чем процессоров состоит в обеспечении лучших возможностей при распределении задач по процессорам (балансировка вычислений). Хорошее правило состоит в том, что число задач на порядок больше числа процессоров.

Оптимальное число задач обычно определяется комбинацией аналитического моделирования и эмпирического изучения.

Сокращение инженерной стоимости софта. До сих пор предполагалось, что выбор стратегии агломерации определялся только улучшением скорости вычислений и гибкостью параллельного алгоритма. Если на этом этапе удастся получить идентичные задачи (несколько или все) то при технической реализации алгоритма можно получить большую экономию как по времени, так и по трудовым затратам при разработке машинных кодов задач.

Результаты этапа разработки коммуникаций необходимо сверить со следующими контрольными вопросами.

1) Уменьшаются ли затраты времени на коммуникации при увеличении локальности? Если нет то надо искать другую стратегию.
2) Если агломерация размножает вычисления, то нужно проверить, что польза от размножения перевешивает затраты по отношению к росту размерности задачи и числу процессоров.
3) Если агломерация размножает данные проверьте, что это не компрометирует масштабируемость вашего алгоритма, ограничивая размер задачи или число используемых процессоров.

4) Агломерация производит задачи с одинаковыми затратами вычислений и коммуникаций? Если получается одна задача на процессор, то их размерности должны быть примерно одинаковы.

5) Масштабируется ли число задач, получаемых при агломерации, с ростом входной размерности обрабатываемых данных. Если нет, то трудно использовать такой алгоритм с ростом размерности и числа процессоров.

6) Если агломерация сокращает возможности для параллельных вычислений проверить остается ли достаточно параллельности для вычислений на большем числе компьютеров. Алгоритм с меньшей параллельностью может быть более эффективным по сравнению с алгоритмом с большей параллельностью из-за чрезмерной стоимости коммуникаций последнего.

7) Можно ли еще увеличивать агломерацию без нарушения балансировки загрузки процессоров, увеличения инженерной стоимости программирования или сокращения масштабируемости. При прочих равных условиях, алгоритм создающий несколько больших задач проще и эффективнее того, который создает много задач.

2.6 Распределение задач по процессорам

На заключительной стадии проектирования параллельного алгоритма нужно определить какая задача на каком процессоре будет выполняться . В целом задача распределения остается трудной задачей, которая должна быть точно разрешена при проектировании параллельного алгоритма. Цель распределения состоит в том чтобы минимизировать время выполнения алгоритма (сумма времен вычисления, простоя и коммуникаций) и затраты памяти.

Используются две стратегии для достижения этой цели:

- размещение задач, которые могут выполняться параллельно на разных процессорах;
- размещение задач, которые часто взаимодействуют, на одном и том же процессоре чтобы увеличить коммуникационную локальность.

Ясно что эти стратегии будут конфликтовать и проектирование должно разрешить эти противоречия. Кроме того на распределение задач по процессорам могут влиять доступные ресурсы одного процессора, ограничивая число задач которые могут быть одновременно размещены на нем.

Такая задача распределения известна как NP-задача. Это означает, что задача трудно разрешима, поскольку для нее нет алгоритма решения с полиномиальной зависимостью от входной размерности (число задач и число процессоров). Поэтому при решении таких задач используются эвристические методы.

Так при разбиении по данным используется загрузочно-балансный алгоритм (динамический загрузочно-балансный алгоритм), обеспечивающий равномерную загрузку процессоров.

При функциональной декомпозиции используется алгоритм составления расписаний для задач. Есть централизованный (в одном месте) или распределенный (в нескольких местах) набор (пул) задач, в который новые задачи помещаются и из которого задачи берутся для размещения по процессорам.

Наиболее сложным аспектом при составлении расписания является используемая стратегия при размещении задач по процессорам. В основном выбираемая стратегия представляет компромисс между конфликтными требованиями по независимости задач (чтобы сократить стоимость коммуникаций) и полном знании о состоянии вычислений (чтобы улучшить балансировку загрузки процессоров). Рассмотрим централизованный (manager/workers), иерархический (manager/submanager/workers) и децентрализованный подходы.

Централизованный подход. Manager (управляющий) отвечает за проблему размещения. Каждый worker (исполнитель) постоянно обращается к manager с запросами (например, сообщение об окончании выполнения задачи) получает задачу от него и исполняет ее. В свою очередь workers могут послать новые задачи manager, которые затем могут быть размещены по другим рабочим. Эффективность такой стратегии зависит от числа workers и относительной стоимости получения задач от manager и их выполнения. Результаты могут быть улучшены за счет перекрытия вычислений и коммуникаций, а также кешированием задач в workers так, чтобы workers связывались с manager когда нет локальных (своих) задач.

Иерархический подход. Workers делятся на отдельные группы у которых свой submanager (подуправляющий). Workers из группы связываются только со своим submanager. Submanager периодически связываются с manager и другими submanager чтобы обеспечить балансировку загрузки процессоров за которые они отвечают.

Децентрализованный подход. В полностью децентрализованной схеме нет центрального управляющего. Соответственно отдельный пул задач назначается на каждый процессор и незанятые процессоры требуют задач от других процессоров. В результате пул задач становится распределенной структурой данных, которая доступна различным задачам асинхронно. При этом определяется какая-либо процедура доступа. Например worker может требовать задания только от небольшого числа соседей или выбирать другие процессоры по “заданному” случайному закону.

Гибридный централизованно - децентрализованный подход. В гибридной централизованной/децентрализованной схеме запросы посылаются центральному управляющему который размещает их по workers, используя круговой "обход-отбор". Нужно отметить, что хотя в этом случае центральный управляющий будет узким местом в системе при большом числе процессоров, доступ к нему будет требоваться менее часто чем к управляющему в централизованном системе и следовательно это более масштабируемая конструкция.

Доступ к распределенному пулу задач может быть получен несколькими способами. Workers могут быть назначены ответственными как за вычисления так и за управление очередью задач. В этом случае каждый workers должен периодически переключаться с вычислений на проверку ожидающих запросов.

Обнаружение завершения. Составление расписаний требует механизма для обнаружения момента времени когда весь алгоритм будет завершен, в противном случае про-

стаивающие workers никогда не остановятся, требуя задания от других workers . Эта задача легко решается в централизованной схеме, где управляющий определяет, что все workers закончили.

Но это более сложная задача в децентрализованной схеме поскольку нет общей информации что процессоры закончили и требования на выполнение заданий будут "бродить" по системе.

Результаты этапа распределения необходимо сверить со следующими контрольными вопросами.

1) Если используется централизованная загрузочно-балансная схема, нужно проверить, что manager не является узким местом в системе. Также можно сократить коммуникационную стоимость в таких схемах передавая указатели на задачи а не сами задачи manager.

2) Если используется централизованная загрузочно - балансная схема, нужно оценить, относительную стоимость различных стратегий. Вероятностное или циклическое распределение задач является простым и должно быть также рассмотрено, так как это поможет избежать необходимости в повторении загрузочно - балансировочных операций.

3) Если используются вероятностные или циклические методы есть ли достаточное число задач для обеспечения балансировки при загрузке. Обычно для этого требуется по крайней мере не менее 10 задач на процессор.

2.7 Анализ результатов проектирования.

Результатом проектирования будет один или несколько алгоритмов. Однако еще нельзя начинать техническую реализацию алгоритма, так как несколько фаз в проектировании остаются. Во - первых нужно провести анализ, чтобы выбрать один из альтернативных алгоритмов, убедившись, что желаемые цели при проектировании достигнуты. Во - вторых нужно внимательно подумать о начальной стоимости проектирования, о возможностях переиспользования существующих кодов. В-третьих нужно исследовать как поведут себя алгоритмы в больших системах, где они будут являться частью, для чего интерполировать полученные результаты. При этом нужно оценить ***эффективность параллельной реализации*** алгоритма как отношения времени его выполнения при последовательной реализации к времени выполнения при параллельной реализации, а также изменение его эффективности с ***ростом числа процессоров при неизменной размерности исходных данных*** и изменение эффективности с ***ростом размерности исходных данных при неизменности числа процессоров***.

3 Применение методики для задач идентификации

Рассмотрим решение задачи, в которой требуется за один просмотр каждой входной последовательности символов, заканчивающейся "bottom", идентифицировать наличие хотя бы одной из искомых цепочек-образов. Количество последовательностей символов (S) равно 10. Длины последовательностей символов равны 10(2),20(2),30(4),40(2), т.е. две последовательности по 10 символов, две последовательности по 20 символов и т.д. Число процессоров в системе (P) равно 4. Входной алфавит $Z=(z_1, \ldots, z_9)$. Искомые цепочки-образы: $z_1z_2z_2 \ldots z_2z_1$; $z_3z_2z_1$; $z_3z_2z_3 \ldots z_2z_3z_1$. Хранение входных последовательностей символов и результатов поиска цепочек в них производится в одном источнике и одном приемнике. Необходимо выполнить проектирование на событийном уровне параллельного алгоритма решения этой задачи и его исследование. Для разработки параллельного алгоритма рекомендуется выполнить следующие этапы проектирования.

3.1 Разработка алгоритма и рекомендации по его реализации для однопроцессорной системы

На первом этапе разработаем для однопроцессорной системы алгоритм поиска цепочек в одной последовательности символов. Алгоритм поиска любой из трех цепочек-образов можно записать регулярным выражением на языке РВАС:

$r(y)=s_0(z_1z_2\{z_2\}z_1 \vee z_3z_2z_1 \vee z_3\{z_2z_3\}z_1)$.

Разобьем это выражение на части и получим систему РВАС:

$s_{к1}(y)=s_0z_1z_2\{z_2\}z_1$ – поиск первой цепочки образа.

$s_{к2}(y)=s_0z_3z_2z_1$ – поиск второй цепочки образа.

$s_{к3}(y)=s_0z_3\{z_2z_3\}z_1$ – поиск третьей цепочки образа.

Перейдем от системы РВАС к НДСКУ и СВФ, а затем, выполнив их коррекцию, получим алгоритм для однопроцессорной системы на языке НДСКУ:

СКУ:

$s_{к1}=s_1z_1$

$s_1=s_2z_2 \vee s_1z_2$

$s_2=s_0z_1$

$s_{к2}=s_3z_1$

$s_3=s_4z_2$

$s_4=s_0z_3$

$s_{к3}=s_5z_1$

$s_5=s_0z_3 \vee s_6z_3$

$s_6=s_5z_2$

$s_f=(s_0 \vee s_{к1} \vee s_{к2} \vee s_{к3} \vee s_1 \vee s_2 \vee s_3 \vee s_4 \vee s_5 \vee s_6 \vee s_f)$&bottom

СВФ:

$y=s_{к1} \vee s_{к2} \vee s_{к3}$

$y_f=s_f$.

Чтобы убедиться в правильности алгоритма, необходима его отладка с использованием тестирования. Для этого удобно использовать систему отладки и моделирования параллельных алгоритмов "СОМПА" (смотри раздел 4). “СОМПА” позволяет эффективно довести алгоритм до работоспособного состояния. Какие преимущества дает использование этой системы? В окне стандартного редактора для приложений ОС WINDOWS вводится алгоритм на РВАС. По нажатию кнопки выполняется преобразование с языка РВАС на НДСКУ, при этом выявляются синтаксические ошибки в РВАС, которые сразу же можно отредактировать. Этот процесс повторяется до устранения всех синтаксических

ошибок. Затем в окне редактора производится коррекция НДСКУ. Также по нажатию кнопки выполняется преобразование алгоритма из НДСКУ на язык ТПиВ. При этом выявляются синтаксические ошибки, которые могли быть допущены при коррекции НДСКУ. С этого момента можно производить моделирование алгоритма, в том числе с использованием пошагового режима. В случае обнаружения логических ошибок в алгоритме очень быстро производится его корректировка на любом из использованных языков. Таким образом значительно минимизируются затраты времени по сравнению с описанием алгоритма на каком-либо языке программирования и отладкой его в какой либо инструментальной среде. Общая схема работы в системе "СОМПА" приведена на рис. 3.1

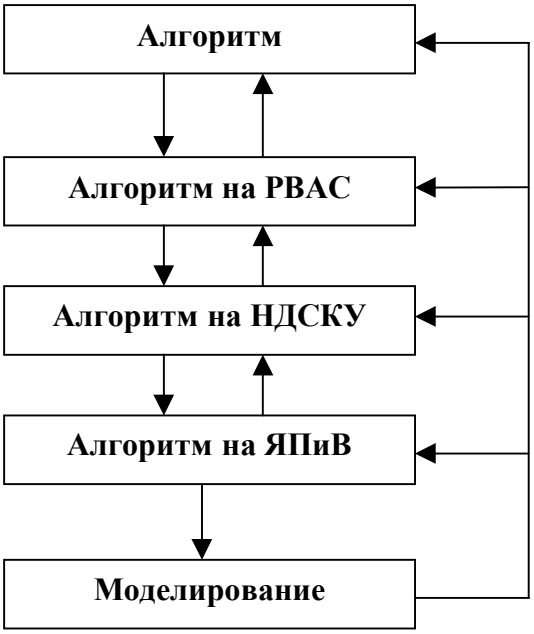


Рис. 3.1 Общая схема работы в системе "СОМПА"

При программной реализации алгоритма на однопроцессорной системе (как и на многопроцессорной системе), возможно несколько вариантов использования полученной НДСКУ. Самый простой состоит в “буквальной” реализации НДСКУ. Для этого в программе создаются два массива для хранения событий в моменты времени **t** и **t+1**. При инициализации программы в массиве для момента времени t значение начальных событий устанавливается в единичное значение, а всех других в нулевое. Для значения очередного входного сигнала вычисляются уравнения НДСКУ в массиве **t+1** используя значения из массива **t** и т.д. Покажем программную реализацию рассмотренного выше примера на языке СИ для ОС MS DOS. Программа обнаруживает наличие в файле (открывается для чтения как "двоичный") искомых цепочек-образов. При этом принято соглашение, что $Z0=00_{16}$, $Z1=01_{16}$, … $Z256=FF_{16}$. Здесь XX_{16} – значение байта в 16-ричной системе. Интерфейс пользователя программы позволяет ввести имя тестируемого файла (контролируется его наличие) и в зависимости от результатов поиска выдается одно из сообщений:

-"Найдена одна из цепочек!",

-"В файле ни одна из цепочек не обнаружена!"

Листинг программы приведен в **приложении 1**. Сразу отметим, что эта программная реализация только иллюстрирует использование НДСКУ и ее нельзя считать эталонной. Покажем пути возможного совершенствования программной реализации.

Во-первых это работа с файловыми потоками (язык C++, библиотека *iostream*) для ускорения работы с дисковыми накопителями. В этом случае использование файлового ввода/вывода (классы *ifstream, ofstream, fstream*) позволяет работать с блоком заданной длины. При этом файл должен быть открыт для чтения в бинарном режиме (*ios::binary*).

Во-вторых несколько усложнив программу можно вычислять только те уравнения НДСКУ, для которых очередной байт из файла будет входным сигналом, т.е. используется в правой части уравнения, что дает значительное сокращение объема вычислений;

В-третьих, возможна любая другая программная интерпретация алгоритма с языка НДСКУ или непосредственно с языка РВАС, в том числе, с использованием функций поиска из библиотек каких-либо инструментальных средств.

В-четвертых, если программное приложение разрабатывается для многозадачной ОС, то возможна организация двух параллельных нитей как показано на рис 3.2.

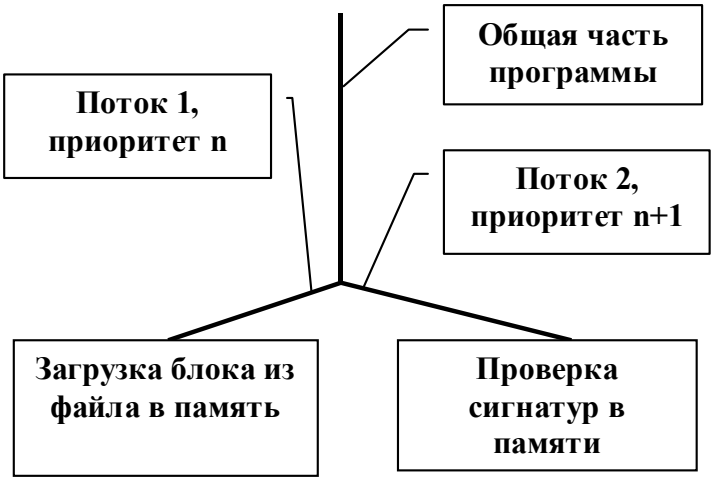


Рис. 3.2 Общая схема работы в системе "СОМПА"

Необходимо отметить, что программное приложение, реализующее алгоритм, должно иметь современный графический интерфейс пользователя, обеспечивающий: идентификацию выполняемого приложения; меню, позволяющее выбрать один из пунктов "тестирование", "лечение", "тестирование и лечение". После выбора пункта меню пользователю должна быть предложена форма с элементами управления, позволяющими выбрать диск, каталог, файл, запустить или прервать поиск, сообщить об обнаружении цепочек-образов с указанием имени файла, где обнаружены и какие именно. Если пользователь выберет на диске каталог, то при поиске, необходимо обеспечить возможность сканирования файлов не только внутри этого каталога, но и во всех вложенных. Дадим необходимую информацию для программной реализации просмотра файлов каталога и файлов всех вложенных в него каталогов. В далее приведенном примере используется инструментальная среда C++Builder 3.0 и также даются сведения необходимые для использования Win32 API.

Пример подготовлен в среде программирования C++Builder 3.0. Для организации выбора каталога были использованы визуальные компоненты TDriveComboBox, TDirectoryListBox, TFileListBox из палитры Win 3.1. Формирование полного имени файла (имя и путь доступа к нему), в котором будет идти поиск цепочки-образа, выполняется в модуле result.cpp. Если выбран файл, то полное имя файла сохраняется в строковой переменной path. Если же выбран каталог, то поиск вирусов будет осуществляться во всех файлах этого каталога и во всех вложенных в него каталогах. В этом случае полное имя файла хранится в строковой переменной buf, меняясь при каждом прохождении тела цик-

ла. Тексты программных модулей для среды программирования C++Builder 3.0 приведены в приложении 2.

Для просмотра содержимого каталога можно использовать функции из Win32 API. Для этого в программном интерфейсе Microsoft Windows NT (Windows 95) предусмотрены функции **FindFirstFile, FindNextFile** и **FindClose.** Просмотр каталога с помощью этих функций выполняется в цикле. Перед началом цикла вызывается функция **FindFirstFile**:

HANDLE FindFirstFile(
LPCTSTR lpFileName, //адрес пути для поиска
LPWIN32_FIND_DATA lpFindFileData); //адрес структуры WIN32_FIND_DATA, куда будет записана информация о файлах

Через параметр **lpFileName** передается адрес строки, содержащей путь к каталогу и шаблон для поиска. В шаблоне можно использовать символы "?" и "*". Через параметр **lpFindFileData** передается адрес структуры типа **WIN32_FIND_DATA**, в которую будет записана информация о найденных файлах. Эта структура определена следующим образом:

```
typedefstruct WIN32_FIND_DATA
{
DWORD dwFileAttributes;      // атрибуты файла
FILETIMEftCreationTime;      // время создания файла
FILETIMEftLastAccessTime; // время доступа
FILETIMEftLastWriteTime;  // время записи
DWORD nFileSizeHigh;         // размер файла (старшее слово)
DWORD nFileSizeLow;          // размер файла (младшее слово)
DWORD dwReserved0;           // зарезервировано
DWORD dwReserved1;           // зарезервировано
TCHAR cFileName[MAX_PATH]; //имя файла
TCHAR cAlternateFileName[14];   // альтернативное имя файла
} WIN32_FIND_DATA;
```

Если поиск завершился успешно, функция **FindFirstFile** возвращает идентификатор поиска, который будет использован в цикле при вызове функции **FindNextFile.** При ошибке возвращается значение **INVALID_HANDLE_VALUE.** Учтите, что поля **cFileName** и **cAlternateFileName** структуры **WIN32_FIND_DATA** содержат соответственно длинное имя файла и короткое, альтернативное имя файла "в формате 8.3".

После вызова функции **FindFirstFile** вы должны выполнить в цикле вызов функции **FindNextFile:**

```
BOOL FindNextFile (
HANDLE hFindFile, // идентификатор поиска
LPWIN32_FIND_DATA lpFindFileData); //адрес структуры WIN32_FIND_DATA
```

Через параметр **hFindFile** этой функции следует передать идентификатор поиска, полученный от функции **FindFirstFile**. Что же касается параметра **lpFindFileData**, то через него вы должны передать адрес той же самой структуры типа **WIN32_FIND_DATA**, что была использована при вызове функции **FindFirstFile.**

Если функция **FindNextFile** завершилась успешно, она возвращает значение TRUE. При ошибке возвращается значение FALSE. Код ошибки вы можете получить от функции **GetLastError**. В том случае, когда были просмотрены все файлы в каталоге, эта функция

возвращает значение **ERROR_NO_MORE_FILES**. Вы должны использовать такую ситуацию для завершения цикла просмотра содержимого каталога.

После завершения цикла просмотра необходимо закрыть идентификатор поиска, вызвав для этого функцию **FindClose:**

BOOL FindClose (HANDLE hFindFile);

В качестве единственного параметра этой функции передается идентификатор поиска, полученный от функции **FindFirstFile.**

В дополнение кратко опишем функции API Microsoft Windows NT, предназначенные для получения информации о дисковых устройствах и состоянии файловой системы. Для того чтобы определить список установленных в системе логических дисковых устройств, вы можете вызвать функцию **GetLogicalDrives:**

DWORD GetLogicalDrives(VOID);

Эта функция не имеет параметров и возвращает 32-разрядное значение, каждый бит которого отвечает за свое логическое устройство. Самый младший, нулевой бит соответствует устройству с идентификатором A: бит с номером 1 - устройству с идентификатором B: и так далее. Если бит установлен, устройство присутствует в системе, если нет - отсутствует. Более развернутую информацию о составе логических дисковых устройств в системе можно получить при помощи функции **GetLogicalDriveStrings:**

DWORD GetLogicalDriveStrings(
DWORD nBufferLength, // размер буфера
LPTSTR lpBuffer); // адрес буфера для записи сведений об устройствах

Если вызвать эту функцию с параметрами **nBufferLength** и **lpBuffer**, равными соответственно 0 и NULL, она вернет размер буфера, необходимый для записи информации о всех логических дисковых устройствах, присутствующих в системе. После этого вы можете вызвать функцию **GetLogicalDriveStrings** еще раз, заказав предварительно буфер нужного размера и указав функции правильный размер буфера. **GetLogicalDriveStrings** заполнит буфер текстовыми строками вида:

A:
B:
C:

Каждая такая строка закрыта двоичным нулем. Последняя строка будет закрыта двумя двоичными нулями.

Определение типа дискового устройства можно выполнить с помощью функции **GetDriveType**:

UINTGetDriveType(LPCTSTRlpRootPathName);

В качестве параметра функции **GetDriveType** нужно передать текстовую строку имени устройства, например полученную при помощи функции **GetLogicalDriveStrings.** В зависимости от типа указанного устройства функция **GetDriveType** может вернуть одно из следующих значений, которые описаны в приведенной ниже таблице:

{PRIVATE}Значение	Описание
0	Тип устройства не удалось определить
1	Указанный корневой каталог не существует
DRIVE_REMOVABLE	Устройство со сменным носителем данных
DRIVE_FIXED	Устройство с несменным носителем данных

DRIVE_REMOTE	Удаленное (сетевое) устройство
DRIVE_CDROM	Устройство чтения CD-ROM
DRIVE_RAMDISK	Электронный диск (RAM)

3.2 Разработка алгоритма и рекомендации по его реализации для многопроцессорной системы

На втором этапе начнем разработку параллельного алгоритма, используя в качестве основы для его проектирования алгоритм для однопроцессорной системы.

На первом шаге выполним декомпозицию алгоритма по данным и по функциям. При этом будем учитывать то, что эти два типа декомпозиции редко используются по отдельности, и чаще всего применяется их комбинация.

Разбиение по данным. На первом (верхнем) уровне можно выполнить разбиение данных по числу просматриваемых входных последовательностей символов ($П_i$). В нашем случае будет 10 таких задач (рис. 3.3).

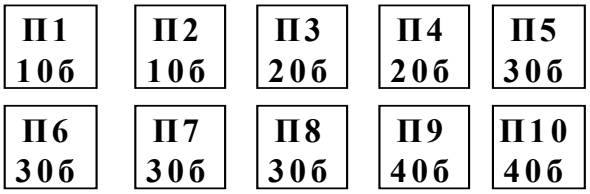


Рис. 3.3. Разбиение по данным

На следующем более низком уровне декомпозиции можно разбить каждую последовательностей на блоки одинаковой длины (за исключением возможно последнего блока). И, казалось бы, в предельном случае минимальный размер блока может достигнуть 1 символа. На самом деле его длина не может быть выбрана любой, т. к. при выборе произвольной длины блока возможен случай, при котором часть цепочки окажется в одном блоке, а другая часть - в другом блоке. Тогда при поиске в каждом из этих блоков цепочка не будет обнаружена. Если выбирать длину блока в зависимости от максимальной длины искомой цепочки-образа можно избежать этого. Максимальная же длина цепочки-образа является постоянной для конкретного алгоритма поиска и поэтому при определении её длины не возникает трудностей. Необходимо учесть еще и то, что брать блоки данных из последовательности нужно с перекрытием, то есть начало следующего блока должно захватывать часть предыдущего, как показано на рис 3.4

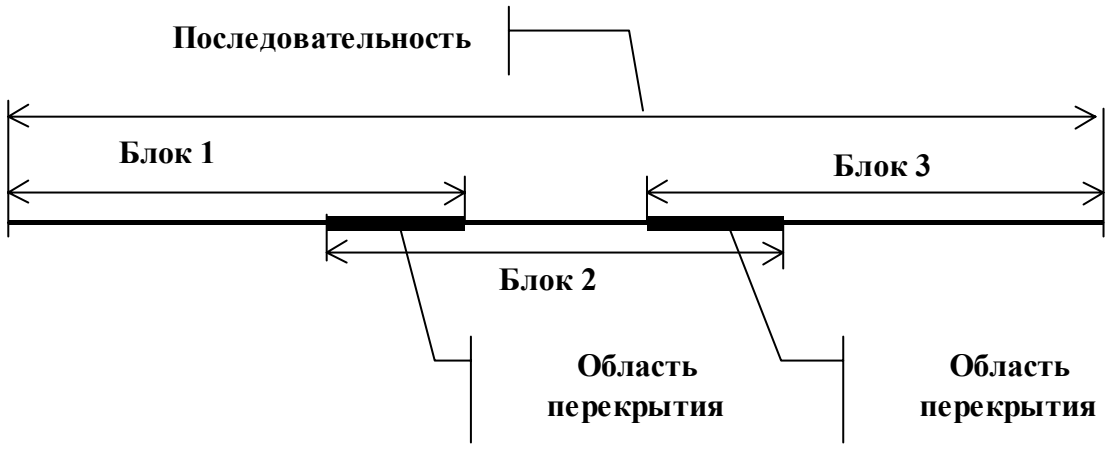


Рис. 3.4

Если цепочки-образы не содержат итераций, то при таком разбиении на блоки поиск может быть выполнен параллельно во всех блоках. Если же цепочка-образ не имеет фиксированной длины (при её задании используется итерация), то алгоритм просмотра блоков последовательности усложняется, и сужаются возможности параллельной обработки. Длина области перекрытия определится в этом случае количеством символов в цепочке предшествующих внешней итерации. Причем если при поиске в блоке эта часть цепочки обнаружена, то в дальнейшем необходимо просматривать последовательно все следующие за этим блоки до тех пор, пока не будет обнаружено, что искомая цепочка прервалась или цепочка найдена.

Исходя из вышесказанного, длину области перекрытия можно определить следующим образом. Пусть максимальная длина цепочки (или её части предшествующей внешней итерации) равна N, тогда длина перекрытия L должна быть равна L=N-1. Единица вычитается для того, чтобы цепочка, если она есть, не вошла в последующий смежный блок. При таком условии невозможен случай, когда часть одной и той же цепочки лежала бы в двух смежных блоках. Естественно, что и длина перекрытия не может быть больше выбранной длины блока.

Такое деление на блоки соответствует условиям реальных задач, поскольку обычно длина искомой цепочки намного меньше длины файла, внутри которого она ищется.

Кроме того, при разбиении последовательности на блоки нужно учесть следующее. Если среди искомых есть цепочки, длины которых меньше N, то при поиске возможен случай, когда такие цепочки целиком будут лежать в двух смежных блоках в области перекрытия. Тогда при поиске они будут учтены дважды. Для решения этой проблемы можно предложить следующее. Если в блоке обнаружена такая цепочка, то нужно проверить - не лежит ли последний символ этой цепочки внутри области перекрытия. Если последний символ окажется в пределах области перекрытия, то полученный результат поиска можно игнорировать, т.к. эта цепочка будет найдена в следующем блоке.

На этом декомпозиция данных завершается, т.к. разбиение символа, как элемента данных, невозможно. Символ, являясь входным сигналом на очередном шаге алгоритма, неделим. Операции, полученные при такой декомпозиции, с входными данными очевидны, а именно, в зависимости от значения очередного входного сигнала происходит изменение состояния алгоритма поиска.

Произведем **разбиение по функциям,** используемым в алгоритме, с учетом их иерархии. В алгоритме можно выделить следующие функции:

1) функция поиска одной цепочки-образа;
2) функция вычисления уравнения НДСКУ;
3) функции, соответствующие операциям при вычислении правой части уравнения НДСКУ.

Разбиение алгоритма по функциям поиска каждой цепочки приведено на рис.3.5.

Рис. 3.5. Разбиение по функциям поиска цепочек

На следующем уровне функцию поиска отдельной цепочки можно разбить по функциям вычисления уравнений НДСКУ. Например, поиск цепочки 1 состоит в вычислении

следующих уравнений: $s_{к1}=s_1z_1$; $s_1=s_2z_2 \vee s_1z_2$; $s_2=s_0z_1$ и соответствующее разбиение приведено на рис.3.6.

Рис. 3.6. Разбиение по функциям вычисления уравнений

Далее, на следующем уровне иерархии, возможно разбиение вычислений правых частей уравнений. Например, уравнение для s_1 можно разбить на функции: конъюнкции, дизъюнкции и присвоения (рис. 3.7).

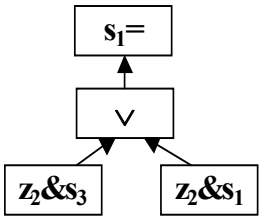


Рис. 3.7. Разбиение правой части для s_1

В конечном итоге полное функциональное разбиение будет представлять собой дерево (рис 3.8), корнем которого будет определение вхождения какой-либо цепочки во входную последовательность, а далее соответственно по ярусам – нахождение конкретной цепочки, затем вычисление уравнений НДСКУ для этой цепочки и затем вычисление их правых частей. На рисунке не приведено разбиение по функциям при вычислении уравнения s_f, т.к. при практической реализации алгоритма достаточно проверять входной сигнал на равенство "bottom", и в случае обнаружения конца последовательности завершать алгоритм поиска. Из рассмотрения этого дерева хорошо видно, что на нижнем уровне выполняется операция конъюнкции, затем дизъюнкции и затем присвоения. Таким образом, в результате разбиения алгоритма по функциям получаются следующие элементарные задачи:

1) конъюнкция $s_i\&z_j$, где s_i- одно из событий НДСКУ, а z_j - входной сигнал;

2) дизъюнкция полученных конъюнкций;

3) присвоение.

Так как эти операции являются элементарными (неделимыми), то дальнейшее функциональное разбиение невозможно и декомпозицию алгоритма как по данным так и по функциям можно считать завершенной.

Анализируя полученные ЭЗ можно утверждать что:

1) число ЭЗ достаточно велико, и это позволит в дальнейшем оптимально распределять ЭЗ между процессорами;

2) возможны слишком большие затраты памяти с ростом числа одновременно обрабатываемых последовательностей (первый уровень декомпозиции данных), и это может привести к ограничению числа разбиений;

3) размеры ЭЗ практически равны, и это позволит в дальнейшем равномерно загрузить процессоры;

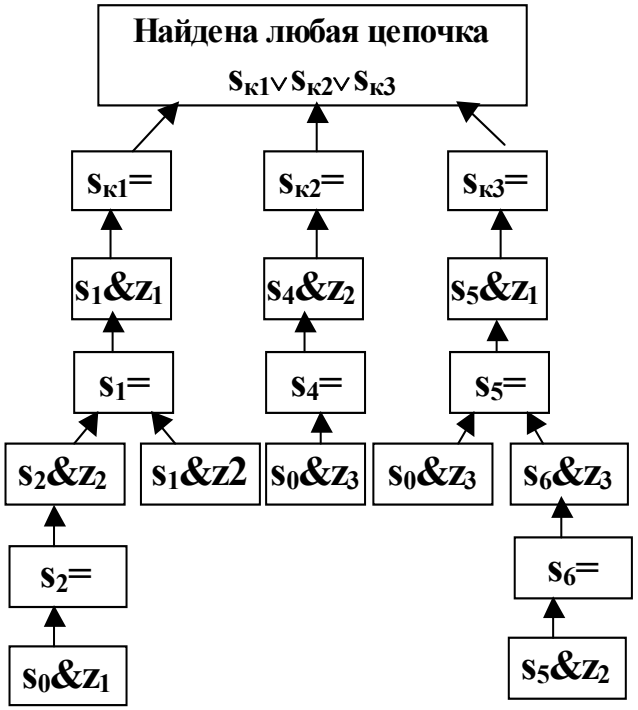


Рис. 3.8. Полное функциональное разбиение

4) при увеличении размерности входных данных возрастает число ЭЗ, а размер Э3 остается неизменным, и это обеспечивает масштабируемость алгоритма поиска.

На втором шаге проектирования параллельного алгоритма определим необходимые коммуникации между ЭЗ, учитывая что:

1) входные последовательности и результаты поиска в каждой из них будут храниться в одном месте (оперативная память, "винт", и т.п.);

2) необходимо определить во входных последовательностях любую из трех цепочек, и поэтому при обнаружении какой либо цепочки в любой последовательности алгоритм завершается.

Определим необходимые коммуникации при разбиении алгоритма по данным (рис. 3.9). При разбиении входной последовательности на блоки коммуникации будут аналогичными (при отсутствии итераций).

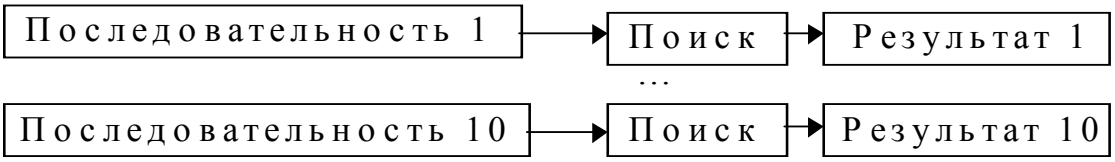


Рис. 3.9. Коммуникации при разбиении алгоритма по данным

Определим необходимые коммуникации при разбиении алгоритма по функциям. При разбиении алгоритма по функциям поиска каждой цепочки коммуникации приведены на рис. 3.10:

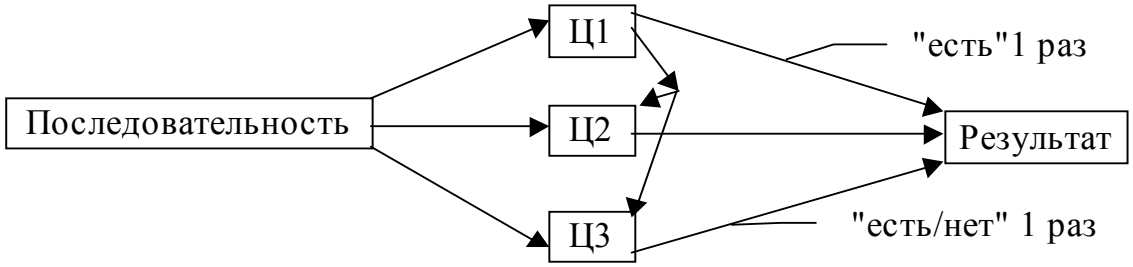


Рис. 3.10. Коммуникации при разбиении по функциям поиска цепочки:

На рисунке 3.9, из-за недостатка места, показана только пересылка сообщений ("цепочка есть") от задачи, обнаруживающей первую цепочку к двум другим. Аналогичные сообщения пересылает каждая задача.

Теперь рассмотрим коммуникации при комбинировании разбиения по данным и разбиения по функциям. Результат установления необходимых коммуникаций представлен на рис 3.11. Для максимального же разбиения по данным, и максимального разбиения по функциям необходимые коммуникации, представлены на рис. 3.12.

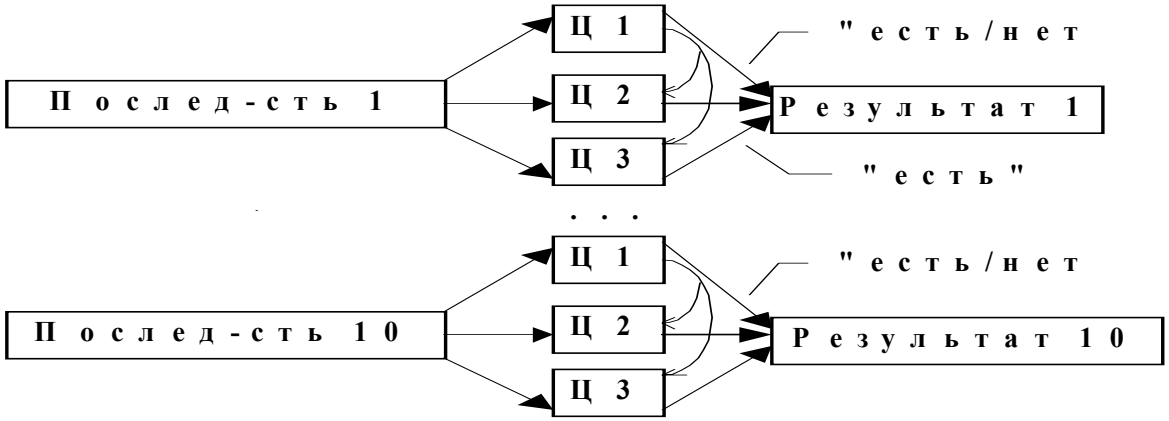


Рис. 3.11. Коммуникации для разбиения по данным и функционального разбиения по цепочкам.

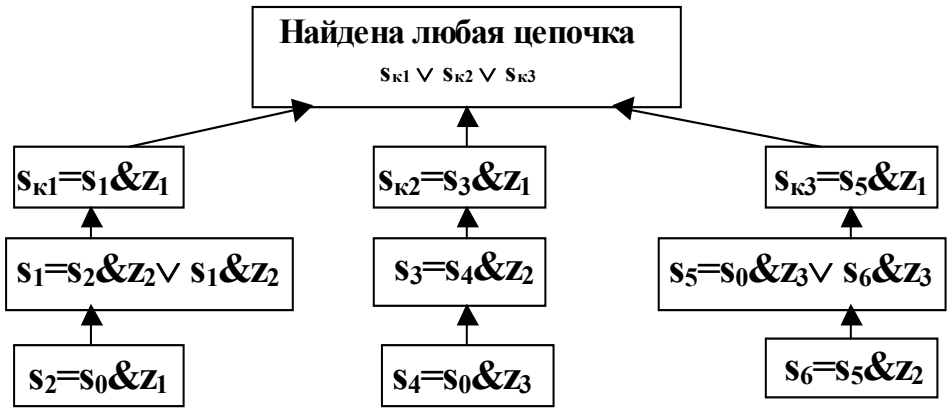


Рис. 3.12. Коммуникации для максимального разбиения по данным и максимального разбиения по функциям.

Полученная при декомпозиции в качестве элементарной задачи конъюнкция требует для своего выполнения два входных параметра. Этими входами являются событие s_i и входной сигнал z_j, которые и определяют коммуникации в алгоритме. Поэтому более детальное рассмотрение коммуникаций по событиям и по входному сигналу в мультипроцессорной системе приводит к следующим выводам.

Вывод 1. В предельном случае (наихудшем) входной сигнал может потребоваться сразу для всех ЭЗ, поэтому для него разумно организовать глобальную коммуникацию. В то же время, используемое событие в каждой ЭЗ своё и поэтому для передачи событий лучше подойдет локальная коммуникация.

Вывод 2. И для входных сигналов и для событий коммуникация получается структурированной. Для входных сигналов это будет дерево, где корнем является входной сигнал, а ветвями - все ЭЗ. Для событий структура коммуникаций будет представлять собой дерево, в корне которого будут события, определяющие наличие какой-либо цепочки во входной последовательности, далее события, определяющие наличие конкретной цепочки, и наконец, события определяющие текущее состояние поиска этой цепочки.

Вывод 3. Т.к. при выполнении алгоритма поиска цепочки-образы не могут быть изменены, то коммуникации будут статическими.

Вывод 4. Если организовать мультипроцессорную систему, на которой будет выполняться алгоритм, таким образом, что интервалы времени между поступлениями очередных входных сигналов будут равны, то возможна использование синхронных коммуникаций. В противном случае необходимы асинхронные коммуникации.

Итоговая оценка полученных коммуникаций между ЭЗ дает следующие результаты:

1) система коммуникаций является структурированной и упорядоченной, что обеспечит в дальнейшем масштабируемость алгоритма;
2) количество коммуникаций для одной ЭЗ мало, что позволит добиться лучшей параллельности выполнения алгоритма;
3) коммуникации нижнего уровня независимы и поэтому могут выполняться параллельно;
4) некоторые коммуникации верхнего уровня не могут выполняться параллельно, но в связи с тем, что их количество в процентном отношении мало, они допустимы в алгоритме;
5) все ЭЗ нижнего уровня выполняются параллельно, т.к. не зависят друг от друга, а также от ЭЗ верхнего уровня.

На третьем шаге проектирования параллельного алгоритма произведем агломерацию ЭЗ. Рассмотрим вначале возможные варианты агломерации ЭЗ, которые были получены при разбиении алгоритма по функциям. Как видно из рис. 3.12, можно провести агломерацию по вертикали или по горизонтали.

При вертикальной агломерации объединяется в одну задачу отдельная ветвь дерева. Таким образом, в каждой ЭЗ осуществляется поиск одной конкретной цепочки. Результаты поиска цепочек объединяются в корне дерева. ЭЗ при этом получаются независимыми, то есть могут работать параллельно. Алгоритм будет достаточно масштабируемым, особенно хорошие результаты получатся, когда количество групп будет равно количеству процессоров. Но у такой агломерации есть существенный недостаток. Дело в том, что ЭЗ нижнего уровня представляет собой конъюнкцию, а конъюнкция является одной из самых быстрых операций в ЭВМ. Следовательно, выигрыш в производительности при такой аг-

ломерации будет незначительным из-за того, что время, затрачиваемое на пересылку входного сигнала в каждую из ЭЗ, будет значительно превосходить время обработки этого сигнала (конъюнкции).

При горизонтальной агломерации в одну ЭЗ объединяется нижний уровень дерева, т.е. все возможные конъюнкции. Это соответствует поиску подцепочки. Вышележащие уровни объединяются в другую ЭЗ, которая будет получать информацию об обнаружения подцепочки, о принадлежности этой подцепочки конкретному образу, анализировать её и по завершении просмотра входных последовательностей выдавать окончательный результат поиска. При такой агломерации ЭЗ нижнего уровня выполняются последовательно, что сужает параллелизм. Основным достоинством такой агломерации является возможность разбиения ЭЗ нижнего уровня по данным. Т.е. одна ЭЗ будет представлять собой поиск всех возможных цепочек в первой последовательности, другая - во второй и так далее.

Рассмотрим теперь варианты агломерации ЭЗ, которые были получены при разбиении алгоритма по данным. На этапе декомпозиции конечным результатом разбиения по данным (теоретически) получился один символ входной последовательности. Естественно, что при агломерации символы можно объединить в блоки. Наиболее очевидным будет объединение в блок целиком входной последовательности. Заметим, что при решении практических задач поиска, этими последовательностями, например, являются файлы. И как отмечалось ранее, параллельная обработка многих файлов (сотен и более) потребует слишком много ресурсов оперативной памяти. Другой возможный вариант – это объединять входные сигналы по принадлежности цепочкам-образам. В этом случае необходимо реализовать фильтр, который анализировал бы входные сигналы и в зависимости от их принадлежности какой-либо цепочке, передавал бы их соответствующему процессу поиска. Этот метод дает хорошие результаты при "вертикальной" функциональной агломерации, когда каждая цепочка ищется отдельным процессом. При таком способе экономится время на пересылку входного сигнала процессам, хотя при большом количестве цепочек он не будет эффективным. Кроме того, этот способ достаточно сложен в практической реализации. Наконец, можно объединять входные данные в последовательности некоторой определенной длины. Этот способ подходит для "горизонтальной" функциональной агломерации. В этом случае входную последовательность можно разбить его на блоки по числу процессоров, каждый из которых способен отыскивать любую из цепочек-образов. Тогда каждый процессор будет обрабатывать свой блок данных, независимо от других, а итоги передавать в процессор, который производит обработку результатов поиска.

Проведя анализ возможных вариантов агломерации выберем для дальнейшего проектирования параллельного алгоритма "горизонтальную" функциональную агломерацию и агломерацию данных в блоки, количество которых соответствует числу процессоров в системе.

На четвертом шаге проектирования параллельного алгоритма произведем выбор модели аппаратной реализации многопроцессорной системы. Модель мультипроцессорной системы будет определяться выбранным вариантом агломерации ЭЗ и требуемыми при этом коммуникациями между ними. Из предлагаемых ранее трех моделей произведем выбор по следующим соображениям. Сетка процессоров не подходит из-за отсутствия в алгоритме "регулярных" коммуникаций (от соседа к соседу). Общая шина не подходит из-за ее общего недостатка - "узости" при коммуникациях. Выбираем модель с общей памятью. Во-первых, она подходит по источникам и приемникам данных для решаемой задачи, а именно исходные данные хранятся в одном источнике, результаты помещаются в

один источник. Во-вторых, каждый процессор может работать независимо от других, за исключением времени обмена с общей памятью. В-третьих, наличие кэша достаточного размера между общей памятью и процессором позволяет сократить временные издержки на коммуникации.

На пятом шаге проектирования параллельного алгоритма работы многопроцессорной системы произведем его разработку в самом общем виде (эскизный проект). Учитывая полученные на предыдущих шагах результаты, можно предложить централизованную мультипроцессорную систему “Управляющий - Рабочие" с общей памятью (рис.3.13).

Рис. 3.13 Архитектура мультипроцессорной системы

Для этой системы предлагается следующий алгоритм.

Для "управляющего" процессора:

1) выполнить начальную инициализацию системы;

2) определить через интерфейс пользователя источник последовательностей (наименование каталогов, файлов и т.д.);

3) открыть очередную входную последовательность (файл);

4) в соответствие с максимальной длиной цепочки - образа определить размер блока, на которые разбивается последовательность;

5) поочередно через общую память передать блоки "рабочим" процессорам, либо если позволяет размер общей памяти переписать всю последовательность в память и передать "рабочим" только начальный адрес блоков;

6) анализировать получаемые от "рабочих" сообщения, в том числе сигнал "конец работы";

7) решать задачу верхнего уровня (устанавливать событие "найдена цепочка");

8) Завершать обработку последовательности, для чего учитывать количество поступивших сигналов "конец работы". Когда их число станет равным числу занятых "рабочих" (это число в принципе может быть меньше общего числа "рабочих"), переходить к обработке следующей последовательности (см. п. 1).

Для "рабочего" процессора:

1) читать очередной символ и осуществлять переходы в алгоритме;

2) если обрабатываемый блок закончился, и нет итерации, передать "управляющему" сигнал завершения обработки блока, иначе запросить блок следующий в обрабатываемой последовательности за предыдущим.

3) если возникло состояние алгоритма "цепочка найдена", то передать сообщение "управляющему" об обнаружении цепочки (возможен вариант алгоритма, когда информация о найденных цепочках будет храниться у "рабочего" до завершения обработки всех блоков, а затем пакетом передаваться "управляющему");

Этот алгоритм можно усовершенствовать. При большом количестве цепочек-образов и достаточном количестве "рабочих" возможно разделение алгоритма работы "управляющего" процессора на алгоритмы двух процессоров: "управляющего" и "анализатора". На "управляющего" будут возлагаться функции загрузки блоков последовательности, а на "анализатор" - прием сообщений от "рабочих" и их обработка. По окончании обработки всех последовательностей "управляющий" получает от "анализатора" результаты. Добавление "анализатора" в мультипроцессорную систему целесообразно в том случае, когда объем оперативной памяти, как общей так и процессоров мал по сравнению с длинами обрабатываемых последовательностей. В этом случае "управляющий" процессор будет тратить много времени на распределение блоков между рабочими процессорами. Архитектура такой системы приведена на рис. 3.14.

В предлагаемой мультипроцессорной системе при начале обработки очередной последовательности каждый "рабочий" должен обратиться к "управляющему" с требованием данных. "Управляющий" распределяет имеющиеся блоки по "рабочим". Каждый "рабочий", обработав полученный блок, обращается к "управляющему» с запросом на получение следующего. Если у "управляющего" есть необработанные блоки, он передает очередной

"рабочему'. В противном случае "рабочему" передается

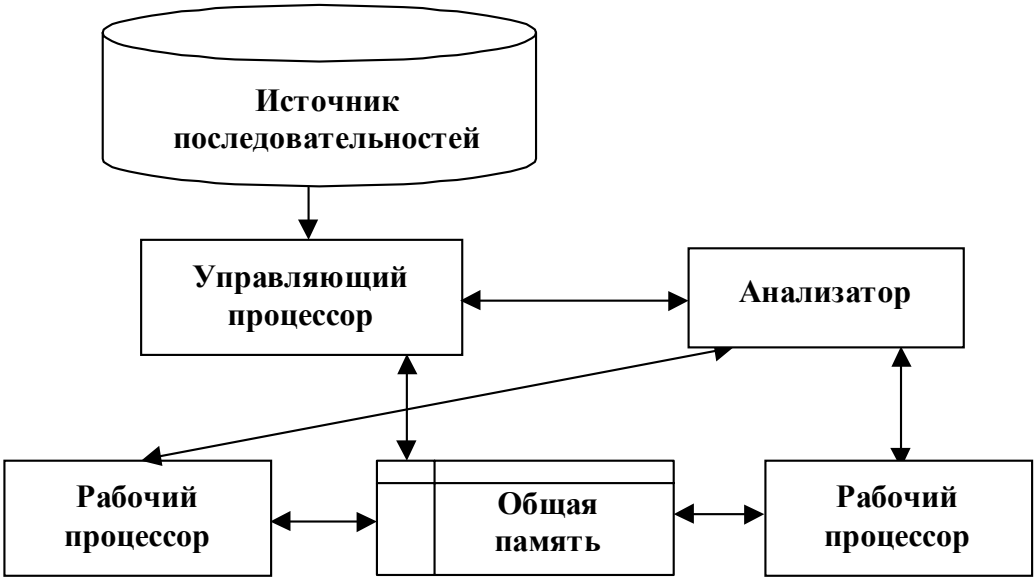


Рис. 3.14 Мультипроцессорная система с анализатором

сообщение об отсутствии необработанных блоков, после чего "рабочий" передает "анализатору" сигнал об окончании работы. "Анализатор" получив
сигналы окончания от всех "рабочих", передает результат поиска "управляющему".

Произведем анализ потоков информации в системе. В ней будет существовать два основных потока информации: передача от "управляющего" к "рабочим" входных данных и передача информационных сообщений между "управляющим", "анализатором" и "рабочими". Эти два потока различаются по требуемой скорости и поэтому разумно организовать их передачу различными способами. Входные данные имеют относительно большой размер и должны передаваться как можно быстрее. Кроме того, блоки передаются последовательно, то есть из файла сначала считывается первый блок, который передается первому "рабочему», затем второй блок, который передается второму "рабочему' и так далее. Информационные сообщения в системе возникают значительно реже и скорость их передачи менее критична.

Если число "рабочих" достаточно большое и производится поиск большого количества цепочек-образов, т.е. высока вероятность обнаружения какой-либо цепочки, то "анализатор" может оказаться узким местом в системе из-за большого числа пересылаемых ему сообщений. В этом случае возможно увеличение числа "анализаторов" в системе по методу "вертикальной" функциональной агломерации когда каждый "анализатор" будет принимать только "свои" сообщения, отобранные по определенному признаку. Например, это могут быть определенные виды цепочек-образов или сообщения от определенной группы "рабочих".

С учетом выше изложенного, предлагается следующий эскизный проект мультипроцессорной системы. В ее составе будут: процессор - "управляющий"; процессор - "анализатор"; два процессора - "рабочие". Максимальная длина цепочки - 3 байта. Длина области перекрытия будет равна 2 байтам. Возможные типы сообщений в системе приведены ниже в таблице.

№	Источник	Приемник	Сообщение	Параметры
1	Рабочий	Управляющий	Свободен	Нет

2	Управляю-щий	Рабочий	Начать работу	Базовый адрес, размер блока
3	Рабочий	Анализатор	Найдена цепочка	Номер цепочки, относительный адрес конца цепочки, базовый адрес
4	Управляю-щий	Рабочий	Работы нет	Нет
5	Рабочий	Анализатор	Работу закончил	Нет
6	Анализатор	Управляю-щий	Поиск завершен	Пакет результатов поиска

Алгоритмы работы процессоров в системе представлены ниже.

Для "управляющего" процессора:

1) произвести инициализацию системы;
2) обеспечить интерфейс пользователя и получить через него указатель на последовательность для поиска;
3) открыть последовательность и определить её длину (L);
4) определить предварительную длину блока S для обработки "рабочим" (S=L/2+2);
5) если S<=2 (длина блока равна или меньше длины области перекрытия), то S=L (при L==0, прекратить выполнение алгоритма);
6) если блок не помещается в оперативную память "рабочего", то перейти к пункту 7;
7) установить базовый адрес - B=0;
8) указатель позиции файла (F) перевести на B+S. Если свободен первый "рабочий", то передать ему текущий блок и затем сообщение типа 2 (с параметрами B и S). В противном случае, если свободен второй "рабочий", то передать текущий блок и затем сообщение типа 2 (с параметрами B и S) второму "рабочему» Если оба "рабочих» заняты перейти в режим ожидания сообщений типа 1 (перейти к пункту 8);
9) если файл закончился, перейти к пункту 10, в противном случае B=F-3, K=L-B (размер оставшейся части файла); если K<S, то S=K перейти к блоку 9.
10) передавать на все запросы типа 1 от "Рабочих" ответы типа 4; ожидать прихода сообщения типа 6; после его получения вывести результаты поиска через интерфейс пользователя.
11) конец алгоритма.

Для "рабочего" процессора:

1) передать сообщение типа 1;

2) ожидать сообщение типа 2 или 4; после получения сообщения типа 4 перейти к пункту 7; после получения сообщения типа 2 перейти к пункту 3;

3) установить указатель адреса на начало оперативной памяти;

4) считать очередной символ из оперативной памяти и выполнить вычисление НДСКУ;

5) если появилось состояние "найдена "цепочка", проверить, не лежит ли она в области перекрытия (цепочка лежит в области перекрытия, если относительный адрес ее окончания (G) меньше либо равен 2); если лежит, то перейти к пункту 4, иначе передать сообщение типа 3 (в качестве параметров относительного и базового адресов передать B и G соответственно);

6) перейти к пункту 4;

7) передать сообщение типа 5;

8) конец.

Для процессора "анализатора":

1) установить счетчик числа "рабочих" равным 2;

2) ожидать сообщения типа 3 или 5;

3) если получено сообщение типа 5, то перейти к пункту 6;

4) если получено сообщение типа 3, то вычислить абсолютный адрес конца цепочки: A=B+G и записать полученный результат вместе с номером цепочки-образа в стек сообщений;

5) перейти к пункту 2;

6) вычесть из счетчика числа "рабочих" единицу и если счетчик равен 0, то перейти к пункту 7, иначе перейти к пункту2;

7) передать сообщение типа 6;

8) конец.

На шестом шаге произведем оценку эффективности многопроцессорной системы. При оценке эффективности многопроцессорной системы относительно однопроцессорной, нужно вначале определить оптимальное число процессоров в системе, т.к. увеличение числа процессоров в системе не дает увеличение скорости работы в такое же количество раз. Для этого выберем критерии, по которым будет определяться число процессоров в системе. Из анализа результатов, полученных при установлении коммуникаций, видно, что скорость работы системы будет в большей степени определяться скоростью передачи

входной последовательности от места её хранения (физического носителя) до "управляющего" процессора. Эта скорость будет намного меньше, чем скорость передачи информации между процессорами системы. Поэтому определение эффективного числа процессоров будет вестись исходя из этого положения. Оптимальным количеством процессоров будет то, при котором первый "рабочий" заканчивает работу в тот момент, когда "управляющий" заканчивает передачу данных последнему "рабочему". В этом случае ни один из процессоров не будет простаивать. Если же: количество процессоров будет меньше, то простаивать будет "управляющий". Но это время может быть использовано "управляющим" процессором для выполнения других функций, кроме решения задач обслуживания "рабочих". Например, для удаления обнаруженных цепочек-образов, обеспечения интерфейса пользователя, и т.д.. Если же количество процессоров будет больше оптимального, то будет простаивать часть "рабочих" процессоров, так как до них задания от "управляющего" не придут. Это будет происходить из-за того, что первый "рабочий" вновь передаст запрос на получение следующего блока раньше, чем процессор сможет загрузить все "рабочие". Таким образом, увеличение числа процессоров не будет повышать скорость работы системы.

Увеличение размера входных данных в общем случае не влияет на величину оптимального количества процессоров. Если длина блока данных не изменяется, а увеличивается только их количество, то в алгоритме работы системы ничего не изменится (увеличится лишь общее время обработки). Если же увеличивается размер блока, то увеличивается время обработки этого блока "рабочим" процессором, но и увеличивается суммарное время передачи блока "рабочему". Если обеспечить в системе, что эти два времени будут равны, то она будет работать без простоев.

Оценим эффективность мультипроцессорной системы при оптимальном количестве процессоров. Время работы однопроцессорной системы T_o:

$T_o=(T_{счит}+T_{обраб})*n$, где $T_{счит}$, $T_{обраб}$ – время считывания и обработки одного байта, а n – число байт в последовательности.

Представим графически работу каждого "рабочего" процессора в мультипроцессорной системе во времени без учета взаимодействия с "управляющим и "анализатором"(рис. 3.15).

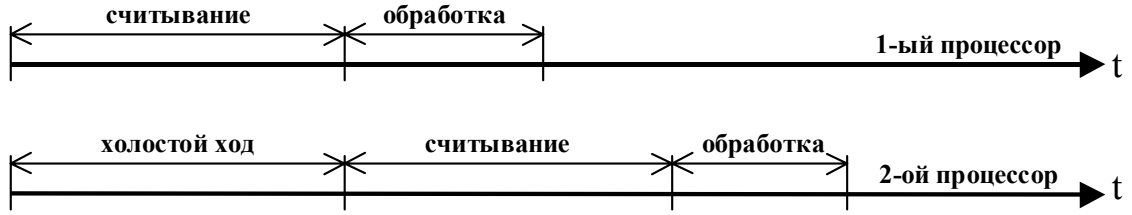


Рис. 3.15 Загрузка работой во времени "рабочих" процессоров

Тогда приближенно время работы многопроцессорной системы $T_м$:

$T_м=T_{счит}*n+T_{обраб}*n/2$

Эффективность выполнения алгоритма при параллельной реализации по сравнению с последовательной будет приблизительно равна:

$$E = \frac{T_o}{T_м} = \frac{2}{\frac{T_{счит}}{T_{счит}+T_{обраб}} + 1}$$

Следует отметить, что полученное выражение показывает максимальную эффективность в результате того, что ранее были введены некоторые допущения. В действительности эффективность будет ниже величины полученной при помощи этой формулы. Из формулы также видно, что если $T_{обраб}$ намного меньше $T_{счит}$, то эффективность системы равна 1, т.е. такой алгоритм работы многопроцессорной системы будет неэффективен.

В заключение сделаем замечания по практической **аппаратной реализации алгоритма**. Аппаратная реализация отлаженного алгоритма в этом случае чрезвычайно проста, если в качестве элементарного автомата памяти использовать D триггер. При этом количество триггеров в схеме соответствует числу уравнений НДСКУ плюс количество начальных состояний в алгоритме. Каждое уравнение будет представлять собой функцию возбуждения соответствующего ему триггера. Таким образом, имеется описание схемы цифрового устройства на входном языке, которое может быть использовано в системах автоматизированного проектирования (CAD/CAM).

4 Инструментальная система разработки и верификации параллельных алгоритмов

Инструментальная система, предназначенная для ускорения разработки алгоритмов с их последующей верификацией путем моделирования, разработана на кафедре "Вычислительная техника" Пензенского государственного университета и получила название "СОМПА" (система описания, моделирования, преобразования алгоритмов). Система предназначена для разработки и верификации алгоритмов, в том числе с использованием параллельной обработки, в различных областях вычислительной техники. Например алгоритмы: - управления цифровыми устройствами в компьютерных системах, - управления цифровыми контроллерами в технологических процессах, - распознавания (поиск вирусов, синтаксический анализ, и т.п.), - и т.д. Система также может использоваться для обучения студентов разработке алгоритмов с их последующей верификацией.

Краткая техническая характеристика системы. Система позволяет: - задавать входные сигналы алгоритма или двоичными, или абстрактными (для абстрактных обязательно использование обозначение z_i), что расширяет диапазон применения системы; - задать (описать) алгоритм на любом из стандартных языков таких, как ГСА, РВАС, СКУ и СВФ; - выполнить преобразование описания алгоритма заданного на одном из языков на любой другой из этих языков; - произвести, если это необходимо, преобразование (детерминизацию) параллельного алгоритма в последовательный (это в ряде случаев полезно и для анализа самого параллельного алгоритма); - произвести упрощение (минимизацию) алгоритма; - проверить правильность (верифицировать) алгоритм моделированием его работы с заданием входа алгоритма по отдельным шагам или на любое число шагов, с возможностью возврата на шаг или на несколько шагов и с возможностью вставки последовательности сигналов в уже промоделированную входную последовательность (при этом автоматически осуществляется моделирование с учетом вставки сигналов).

Система отличается: - высокой эффективностью при создании и верификации алгоритмов, так как не требуется использовать языки программирования; - хорошо интуитивно понимаемым графическим интерфейсом для каждого этапа работы; - удобными встро-

енными редакторами для каждого из используемых языков; - удобными формами для задания входных воздействий на очередном шаге алгоритма; - удобными формами представления результатов моделирования.

Необходимые ресурсы для работы системы "СОМПА": операционные системы - Windows 95(минимальный размер ОЗУ - 8Мб) или Windows NT(минимальный размер ОЗУ - 16Мб)

До запуска системы "СОМПА" необходимо, установить специально разработанный для нее шрифт NDA.ttf, используя "панель управления" WINDOWS и значок "шрифты" в ней. Тогда ввод отрицания двоичного входного сигнала или отрицания события производится при нажатой клавише Shift в латинском алфавите, например < Shift > + <x> (при этом прописные буквы исключены), а операции дизъюнкции,- одновременным нажатием <Shift> + < прерывистая вертикальная черта > также в латинском алфавите.

Приведем описание меню, подменю и вкладок в окнах системы "СОМПА".

При запуске приложения "СОМПА", окно, появляющееся на экране содержит следующие пункты основного меню: "Файл", "Окно", "Помощь". "Файл" позволяет: - вновь задать (подпункт "создать") алгоритм на одном из языков: СКУ и СВФ, РВАС, ГСА по выбору пользователя;, - вновь задать ("создать таблицу") алгоритм на языке ТПиВ; - открыть существующий файл с алгоритмом; - выйти из приложения. Пункты "Окно" и "Помощь" стандартны для приложения под ОС Windows.

Если алгоритм работы задается на языке РВАС то происходит контекстная смена основного меню, которое будет теперь содержать пункты: "Файл", "Правка", "Формат", "Преобразования" "Окно", "Помощь". В "Файл" добавляются пункты подменю, которые позволяют закрыть, сохранить, сохранить с изменением имени создаваемые или открытые файлы. При этом расширение файла формируется автоматически в контекстной зависимости от языка, используемого в активном окне (для РВАС - <имя файла>.rvs). Пункты "Правка", "Формат" основного меню это стандартные подпункты текстовых редакторов, в том числе с возможностью включения / выключения переноса и изменения размера шрифта. Пункт "Преобразования" позволяет выполнить переход от описания алгоритма на языке РВАС к языкам ТПиВ или СКУ и СВФ.

При работе с языком СКУ и СВФ представляются аналогичные возможности (для СКУ и СВФ расширение - .sku).

При работе с языком ТПиВ (расширение - .tbl) в основное меню добавляется: - в пункт "Правка" подпункты "вставить строку" (перед текущей),"добавить строку" (в конец таблицы),"удалить строку" (текущую); - пункт "Моделирование"; - в пункт "Преобразования" добавляются подпункты "детерминизация" и "список переходов". Отметим, что моделирование алгоритма возможно только при его задании на языке ТПиВ.

При выборе пункта "Моделирование" и подпункта "стандартное" открывается окно "автоматическое моделирование" имеюшее четыре вкладки. Вкладка "общее" позволяет просмотреть имя таблицы, число состояний, число входных сигналов и их тип, число выходных сигналов. Вкладка "алфавит" позволяет просмотреть алфавит состояний, алфавит входных сигналов (если входные сигналы абстрактные, то предоставляется возможность изменить алфавит, добавить или удалить сигналы) и алфавит выходных сигналов. Вкладка "переходы" позволяет просмотреть все возможные переходы из одного состояния в другое. Вкладка "автоматическое моделирование" позволяет формировать входную последовательность сигналов для алгоритма. Для этого используются кнопки "изменить", "вставить перед текущим сигналом", "удалить (текущий сигнал)", "добавить (в конец последовательности)", "удалить (последний сигнал)". Перед началом моделирования с

использованием кнопки "S0" задается начальное множество состояний (переводом из окна "пассивные" в окно "активные"). С этого момента производится моделирование алгоритма для всей сформированной входной последовательности, после чего можно модифицировать входную последовательность добавлением, удалением или вставкой входных сигналов. Входную последовательность можно сохранить в файле (расширение .sgn) для последующей работы (кнопка "сохранить сигналы") и загрузить из файла (кнопка "загрузить сигналы"). Результаты моделирования можно сохранить в файле (кнопка "отчет о моделировании"). По умолчанию результаты моделирования представляются вертикальной таблицей, но возможна и горизонтальная таблица (кнопка "горизонтальная таблица"). С помощью кнопки "закрыть" окно моделирования закрывается.

При выборе пункта "Моделирование" и подпункта "по шагам" открывается окно в котором можно с использованием соответствующих кнопок установить входной сигнал на шаге алгоритма, задать множество начальных состояний, выполнить шаг моделирования, вернуться назад на шаг, сохранить историю переходов и выйти из окна моделирования.

При выборе пункта "Преобразования" и подпункта "детерминизация" открывается окно содержащее три вкладки. Вкладка "общее" дает общую информацию по алгоритму. Вкладка "алфавит" позволяет просмотреть алфавиты состояний и входных сигналов. Вкладка "главная" позволяет выполнить детерминизацию алгоритма. Для этого вначале с использованием кнопки S0 ("установить исходное событие") необходимо установить, перенеся из правого окна в левое все события, которые в начальный момент времени равны единице. Затем нажать кнопку "детерминизация". При обнаружении ошибок выдается сообщение, в соответствие с которым нужно внести исправления в алгоритм. Если детерминизация прошла успешно, о чем выдается сообщение, то нажав кнопку "сформировать таблицу" получаем детерминированный алгоритм на языке ТПиВ. Если предварительно установить checkbox "разрешить преобразование имен", а в текстовом поле справа задать шаблон имени, то в таблице детерминированного алгоритма будет произведена подстановка вместо сочетаний событий параллельного алгоритма соответствующего события последовательного алгоритма. С использованием радиокнопок можно установить тип триггера с учетом которого производится детерминизация.

Как и в любом приложении под ОС WINDOWS в главном окне ниже основного меню располагается контекстно - зависимая панель инструментов, позволяющая получить быстрый доступ к пунктам меню. Поддерживается также вызов контекстного меню при нажатии правой кнопки мышки.

5 ЛИТЕРАТУРА

1. *Вашкевич Н. П.* Синтез микропрограммных управляющих автоматов: Учеб. пособие. - Пенза: Пенз. политехн. ун-т, 1990. - 115 с.
2. *Вашкевич Н. П.*, *Вашкевич С. Н.* Недетерминированные автоматы и их использование для синтеза систем управления: Учеб. пособие. - Пенза: Пенз. государст. техн. ун-т, 1996. - 88 с.
3. *Глушков В. М.* Синтез цифровых автоматов. - М.: Физматгиз, 1962. - 476 с.
4. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. - М.: Мир, 1979. - 535 с.
5. Foster Ian "Designing and Building Parallel Programs": Конспект лекций. Электронный адрес - www.mcs.anl.gov.

ПРИЛОЖЕНИЕ 1

```
-----------------------------------------------------------------------------------------------
// ЗАГОЛОВОЧНЫЙ ФАЙЛ
// nda.h
//
#ifndef NDA_H
#define NDA_H
#define NUM_OF_EVENTS 10     //число уравнений СКУ.
#define BOOL short int
#define TRUE 0xFFFF
#define FALSE 0
#define Sk1 0    //определение констант препроцессора для более удобной
#define S1 1    //адресации массива событий СКУ.
#define S2 2
#define Sk2 3
#define S3 4
#define S4 5
#define Sk3 6
#define S5 7
#define S6 8
#define Sf 9
#define S0 10
#endif
//
// nda.cpp  - основной файл.
#include "nda.h"
#include <stdio.h
#include <conio.h
void main()
{
   FILE *fl=NULL;    //указатель на файл, в котором ищутся цепочки-образы.
   char name[81];    //буфер для ввода имени файла.
   BOOL Sprev[NUM_OF_EVENTS];    //массив для событий СКУ от (t-1).
   BOOL Snext[NUM_OF_EVENTS];    //массив для событий СКУ от (t).
   BOOL Z[256];    //массив для входных сигналов с Z0 по Z255.
   int ch=0;    //переменная для хранения очередного байта файла.
   int flag=0;    //признак обнаружения цепочки-образа.
   int i=0;     //счетчик цикла.
   //ввод имени файла для поиска.
   printf("Пример программы поиска цепочек-образов в файле \n");
   printf("Введите имя файла (полный путь): ");
   gets(name);
   fl=fopen(name,"r");
   if(fl==NULL){
      perror("Вы ошиблись! ");
      getch();
      return;}
   //инициализация начальных значений S(t) и S(t-1) СКУ.
   for(i=0;i<NUM_OF_EVENTS;Snext[i]=FALSE,i++);
   for(i=0;i<NUM_OF_EVENTS;Sprev[i]=FALSE,i++);
   //установка начальных событий (в данном примере одного).
   Sprev[S0]=TRUE;
//чтение байта из файла
```

```
    while((ch=fgetc(fl))!=EOF){
      for(i=0; i<256; i++) (i==ch)?(Z[ch]=TRUE):(Z[i]=FALSE);
      //вычисление СКУ
      Sprev[S0]=TRUE;
      Snext[SK1]=Sprev[S1]&Z[1];
      Snext[S1]=(Sprev[S2]&Z[2])|(Sprev[S1]&Z[2]);
      Snext[S2]=Sprev[S0]&Z[1];
      Snext[Sk2]= Sprev[S3]&Z[1];
      Snext[S3]=Sprev[S4]&Z[2];
      Snext[S4]=Sprev[S0]&Z[3];
      Snext[Sk3]=Sprev[S5]&Z[1];
      Snext[S5]=(Sprev[S0]&Z[3])|(Sprev[S6]&Z[3]);
      Snext[S6]=Sprev[S5]&Z[2];
  //проверка обнаружения цепочки-образа
      if(Snext[SK1]|Snext[SK2]| Snext[SK3]){
        printf("Найдена одна из цепочек \n");
        flag=1;    //флаг обнаружения установлен
      }
      else
                  //перепись S(t) в S(t-1)
        for(i=0;i<NUM_OF_EVENTS;Sprev[i]=Snext[i],i++);
    }
    if(flag==0)
      printf("Ни одна из цепочек не найдена \n");
    i=getch();
  }
-------------------------------------------------------------------------------------------
```

```
//Scandir.cpp - основной модуль проекта для C++Builder 3.0
//---------------------------------------------------------------------------
#include <vcl.h
#pragma hdrstop
USERES("Scandir.res");
USEFORM("main.cpp", Form1);
USEFORM("result.cpp", OKBottomDlg);
//---------------------------------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
try
{
Application-Initialize();
Application-Title = "Выбор диска и каталога";
Application-CreateForm(__classid(TForm1), &Form1);
Application-Run();
}
catch (Exception &exception)
{
Application-ShowException(&exception);
}
return 0;
}
//---------------------------------------------------------------------------
main.h - заголовочный файл
//---------------------------------------------------------------------------
#ifndef mainH
#define mainH
//---------------------------------------------------------------------------
#include <Classes.hpp
#include <Controls.hpp
#include <StdCtrls.hpp
#include <Forms.hpp
#include <ComCtrls.hpp
#include <Dialogs.hpp
#include "cdiroutl.h"
#include <Grids.hpp
#include <Outline.hpp
#include <FileCtrl.hpp
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
__published: // IDE-managed Components
TEdit *Edit1;
TLabel *Label1;
TButton *Button1;
TButton *Button2;
TDriveComboBox *DriveComboBox1;
TDirectoryListBox *DirectoryListBox1;
TFileListBox *FileListBox1;
void __fastcall Button2Click(TObject *Sender);
void __fastcall Button1Click(TObject *Sender);
```

```
void __fastcall DriveComboBox1Change(TObject *Sender);
void __fastcall DirectoryListBox1Change(TObject *Sender);
void __fastcall FileListBox1Change(TObject *Sender);
private: // User declarations
bool file;
public: // User declarations
__fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

main.cpp - главная форма приложения "Выбор диска и каталога"

```
//---------------------------------------------------------------------------
#include <vcl.h
#pragma hdrstop
#include "main.h"
#include "result.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma link "cdiroutl"
#pragma link "DriveComboBox1"
#pragma link "DriveComboBox1"
#pragma link "DirectoryListBox1"
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
: TForm(Owner)
{
file=false;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
Release();
Application-Terminate();
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
OKBottomDlg=new TOKBottomDlg(Application,Edit1-Text.c_str(),file);
OKBottomDlg-ShowModal();
delete OKBottomDlg;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::DriveComboBox1Change(TObject *Sender)
{
DirectoryListBox1-Drive = DriveComboBox1-Drive;
FileListBox1-Drive = DriveComboBox1-Drive;
FileListBox1-Directory = DirectoryListBox1-Directory;
Edit1-Text=DirectoryListBox1-Directory;
file=false;
}
//---------------------------------------------------------------------------
```

```
void __fastcall TForm1::DirectoryListBox1Change(TObject *Sender)
{
DirectoryListBox1-Drive = DriveComboBox1-Drive;
FileListBox1-Drive = DriveComboBox1-Drive;
FileListBox1-Directory = DirectoryListBox1-Directory;
Edit1-Text=DirectoryListBox1-Directory;
file=false;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FileListBox1Change(TObject *Sender)
{
DirectoryListBox1-Drive = DriveComboBox1-Drive;
FileListBox1-Drive = DriveComboBox1-Drive;
FileListBox1-Directory = DirectoryListBox1-Directory;
Edit1-Text=FileListBox1-FileName;
file=true;
}
//---------------------------------------------------------------------------
result.h - заголовочный файл
//---------------------------------------------------------------------------
#ifndef OCBH
#define OCBH
//---------------------------------------------------------------------------
#include <System.hpp
#include <Windows.hpp
#include <SysUtils.hpp
#include <Classes.hpp
#include <Graphics.hpp
#include <StdCtrls.hpp
#include <Forms.hpp
#include <Controls.hpp
#include <Buttons.hpp
#include <ExtCtrls.hpp
#include <dir.h
//---------------------------------------------------------------------------
class TOKBottomDlg : public TForm
{
__published:
TButton *CancelBtn;
TListBox *ListBox1;
TLabel *Label1;
private:
char path[256];
TStringList *dirItem;
void scanDir(char *,TStringList *);
public:
virtual __fastcall TOKBottomDlg(TComponent* ,char *,bool);
};
//---------------------------------------------------------------------------
extern PACKAGE TOKBottomDlg *OKBottomDlg;
//---------------------------------------------------------------------------
#endif
result.cpp - форма "Результат полного сканирования каталога"
//---------------------------------------------------------------------
#include <vcl.h
```

```
#pragma hdrstop
#include <dir.h
#include <dos.h
#include <string.h
#include "result.h"
//---------------------------------------------------------------------
#pragma resource "*.dfm"
TOKBottomDlg *OKBottomDlg;
//---------------------------------------------------------------------
__fastcall TOKBottomDlg::TOKBottomDlg(TComponent* AOwner,char *ptr,bool file)
: TForm(AOwner)
{
strcpy(path,ptr);
ListBox1-Items-BeginUpdate();
try
{
ListBox1-Items-Clear();
if(!file)
{
dirItem=new TStringList();
dirItem-Clear();
dirItem-Sorted = true;
scanDir(path,dirItem);
for(int i=0; i<dirItem-Count;i++)
ListBox1-Items-Add(dirItem-Strings[i]);
}
else
/*************************************************************************
ЕСЛИ ВЫБРАН файл то:
строковая переменная path, заканчивающаяся нулем, содержит полный путь доступа к выбранному файлу. Значение переменной path может быть использовано в качестве параметра для функции поиска цепочек-образов.
*************************************************************************/
ListBox1-Items-Add(path);
ListBox1-Items-EndUpdate();
}
catch(...)
{
ListBox1-Items-EndUpdate();
throw;
}
}
//---------------------------------------------------------------------
void TOKBottomDlg::scanDir(char *path,TStringList *dirItem)
{
int err,count;
ffblk fileInfo;
char buf[1024];
if(path[strlen(path)-1]!='\\')
strcat(path,"\\");
strcpy(buf,path);
count=strlen(buf);
strcat(path,"*");
err = findfirst(path,&fileInfo,FA_DIREC);
while (!err)
```

```
{
if(strcmp(fileInfo.ff_name,".") && strcmp(fileInfo.ff_name,".."))
{
strcat(buf,fileInfo.ff_name);
if(fileInfo.ff_attrib==FA_DIREC)
scanDir(buf,dirItem);
else
/*************************************************************************
ЕСЛИ ВЫБРАН КАТАЛОГ ТО:
строковая переменная buf, заканчивающаяся нулем, содержит полный путь доступа к очередному файлу. Значение переменной buf может быть использовано в качестве параметра для функции поиска цепочек-образов.
*************************************************************************/
dirItem-Add(buf);
buf[count]='\0';
}
err = findnext(&fileInfo);
}
findclose(&fileInfo);
}
//-----------------------------------------------------------------------------------------------------
```

С О Д Е Р Ж А Н И Е