УДК 004.272.43+004.272.32

Н.Л. Вереник[1], А.И. Гирель[1], Е.Н. Сейткулов[2], М.М. Татур[1]
[1]Белорусский государственный университет информатики и радиоэлектроники, г. Минск, Беларусь
Республика Беларусь, 220013, г. Минск, ул. П. Бровки, 6
[2]Евразийский национальный университет имени Л.Н. Гумилева, Казахстан
Казахстан, 010008, г. Астана,ул. Мунайтпасова, 5

Имитационная модель векторного процессора на примере задачи поиска пути в графе

N.L. Verenik[1], A.I. Girel[1], Y.N. Seitkulov[2], M.M. Tatur[1]
[1]Belarusian State University of Informatics and Radioelectronics, Belarus
Belarus, 220013, Minsk, P. Browki Str., 6
[2]L.N. Gumilyov Eurasian National University, Kazakhstan
Kazakhstan, 010008, Astana, Munaitpasov Str., 5

Simulation Model of Vector Processor Solving the Problem of Path Finding in Graph

Н.Л. Вереник[1], А.І. Гірель[1], Є.Н. Сейткулов[2], М.М. Татур[1]
[1]Білоруський державний університет інформатики і радіоелектроніки, Білорусь
Республіка Білорусь, 220013, м. Мінськ, вул. П. Бровки, 6
[2]Євразійський національний університет імені Л. Гумільова, Казахстан
Казахстан, 010008, м. Астана, вул. Мунайтпасова, 5

Імітаційна модель векторного процесора на прикладі задачі пошуку шляху у графі

В статье рассматривается решение задачи поиска кратчайшего пути в графе на базе имитационной модели разрабатываемого векторного процессора семантической обработки информации. Дается краткое описание архитектуры процессора, формата данных и системы команд процессора, основных принципов функционирования. Строится формальный алгоритм решения с использованием системы команд процессора.
Ключевые слова: семантическая обработка информации, семантическая сеть, векторный процессор, параллельные вычисления.

The article considers the solution of the shortest path in graph problem using simulation model of vector processor for semantic information processing. Brief description is given for processor architecture, processor's data format and instruction set, basic principles of functioning. Regular solution algorithm using the instruction set of the processor is constructed.
Key words: semantic information processing, semantic network, vector processor, parallel computing.

У статті розглядається вирішення задачі пошуку найбільш короткого шляху у графі на базі імітаційної моделі розроблюваного векторного процесора семантичної обробки інформації. Надається короткий опис архітектурі процесора, формату даних та системи команд процесора, основних принципів функціонування. Будується формальний алгоритм вирішення з використанням системи команд процесора.
Ключові слова: семантична обробка інформації, семантична мережа, векторний процесор, паралельні обрахування.

Современные информационные системы, хранилища (репозитории) информации, как правило, обладают рядом интеллектуальных функций. Например, поиск необхо-

димой информации в глобальных базах, категоризация контента, различные фильтры информации, блокировка вредоносной информации, систематизация и структурирование информации, удаление бесполезной информации и многое другое. В основе любой «интеллектуальной» обработки информации можно выделить ряд актуальных базовых задач, таких как представление информации в виде семантических сетей, семантический анализ информации и ассоциативный поиск информации по некоторому ключу. Под семантической сетью понимается графовая структура, вершины и дуги которой в соответствии с определенными правилами наделены некоторой смысловой нагрузкой. Аппарат семантических сетей обеспечивает не только визуализацию структуры информации, но, что главное, позволяет разрабатывать формальные методы и алгоритмы анализа и модификации знаний.

Несмотря на широкую распространенность семантических сетей в качестве модели представления знаний, общепринятой теории, определяющей методы кодирования и переработки информации в семантических сетях, до сих пор не существует. Согласно ряду проведенных исследований [1], теоретическим фундаментом необходимой унификации может выступить концепция платформенной независимости. Создаваемые методы и алгоритмы семантической обработки информации верхнего уровня интеллектуальной системы разрабатываются независимо от их программной и аппаратной реализации на нижнем уровне. Это возможно только при условии, когда семантическая сеть может быть формально преобразована (перекодирована) в некоторый граф. Тогда обработка информации на нижнем уровне интеллектуальной системы будет представлять собой графодинамический процесс, то есть процесс преобразования графовой структуры семантической сети, в ходе которого изменяется не только состояние элементов структуры, но и ее конфигурация (появляются и удаляются вершины, изменяются связи между ними, изменяются атрибуты вершин и связей и т.п.).

На рис. 1 представлена общая схема интеллектуальной системы, основанная на принципе платформенной независимости. Для обозначения математического (алгоритмического) аппарата семантической обработки нижнего уровня предложен удачный (по мнению авторов) термин – абстрактная графодинамическая машина (ГДМ).

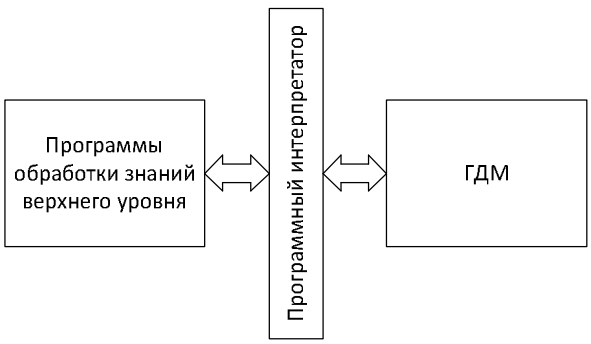

Рисунок 1 – Общая схема интеллектуальной системы, иллюстрирующая принцип платформенной независимости

Полагается, что системы, построенные по такому принципу, могут гибко развиваться, как в верхнем направлении – построение онтологий в различных областях, разработка методов и алгоритмов семантического представления и анализа знаний, так и в нижнем направлении – программно-аппаратная реализация. В качестве реализации абстрактной ГДМ может выступать любая вычислительная система с любой архитектурой, с любой аппаратной и программной платформами. Очевидно, что их выбор будет влиять на трудоемкость программирования и производительность, которую

они смогут обеспечить в конечном итоге. Платформенная независимость реализуется за счет программного интерпретатора, осуществляющего трансляцию (перекодирование) программ семантической обработки верхнего уровня в формат команд конкретной ГДМ.

В настоящей работе мы остановимся на исследовании вопроса отображения алгоритмов семантической обработки на различные варианты архитектуры ГДМ. Как правило, прикладные интеллектуальные системы функционируют на базе универсальных CPU. В особых случаях, когда производительность и обеспечение режима реального времени становятся определяющими факторами системы, используют серийные программно-аппаратные платформы с параллельной архитектурой. Так, наиболее часто исследователи применяют GPU, как особый вид доступной аппаратной платформы с параллельной архитектурой. Действительно, применение универсальных многопроцессорных систем позволяет повысить эффективность решения задач определенного класса, но в итоге приводит к ряду совершенно других принципиальных трудностей. Распределение задач между процессорами, данных между блоками памяти, организация взаимодействия между процессорами, высокая вычислительная сложность и нерегулярность вычислений создают неразрешимые проблемы при распараллеливании алгоритмов. В конечном счете, это ведет к эффекту замедления роста производительности при увеличении числа процессорных элементов, сформулированному в виде закономерностей Густавсона-Барсиса [2] и Амдаля-Уэра [3]. По опыту эксплуатации параллельных систем, в случае нерегулярной структуры графа вычислений, эффективность вычислений достигает максимума при работе четырех-пяти процессоров, а при подключении большего числа процессоров она начинает падать из-за стремительно растущих затрат на обеспечение взаимодействия между процессорами.

Своеобразной «золотой серединой» между универсальными системами и спецпроцессорами, ориентированными на обработку знаний, являются проблемно-ориентированные процессоры [4], [5]. Основная идея в их использовании состоит в том, чтобы обеспечить унификацию процессора в «некоторых» рамках, сохранив при этом достаточный уровень производительности. В то же время достигаемые технические характеристики и издержки на обеспечение универсальности оригинальной архитектуры должны быть конкурентными по сравнению со спецпроцессорами и серийными параллельными процессорами (многоядерными CPU, GPU, DSP).

В статье дается краткое описание оригинальной архитектуры разрабатываемого проблемно-ориентированного SIMD-процессора, а затем рассматривается построенная имитационная модель такого процессора. Моделирование архитектуры на типовых задачах семантической обработки и ассоциативного поиска является ключевым этапом в разработке программно-аппаратной платформы. В работе [6] была показана возможность преобразования произвольной семантической сети (записанной на языке SC) к графу регулярной структуры, а также разработан формальный механизм такого преобразования. Основной идеей трансформации сети является преобразование множества типов и свойств ее узлов и коннекторов (аналоги вершин и дуг соответственно) к обычным вершинам (или дугам) графа, которым ставится в соответствие некоторый конечный набор атрибутов. Фактически, можно судить о возможности сведения задачи семантического анализа к набору классических задач на графах, решение которых, однако, зачастую не является тривиальным.

Для примера (и дальнейшей детализации архитектуры) была выбрана одна из наиболее широко известных задач теории графов – поиск кратчайшего пути в графе. В ходе решения тестовой задачи была верифицирована имитационная модель разрабатываемого векторного процессора.

Модель графодинамической машины

В качестве типовой архитектуры ГДМ предлагается использовать SIMD-архитектуру магистрального типа (Single Instruction Multiply Data, согласно классификации Флинна) с локальной оперативной памятью. Данная архитектура широко применяется во многих вычислительных системах и известна как одна из наиболее технологичных и простых в использовании. В качестве примера достаточно привести SIMD-расширения, используемые в процессорах x86: MMX и EMMX, 3DNow! и E3DNow!, SSE, SSE2, SSE3 и др. На рис. 2 приведен эскиз предлагаемой SIMD-архитектуры ГДМ.

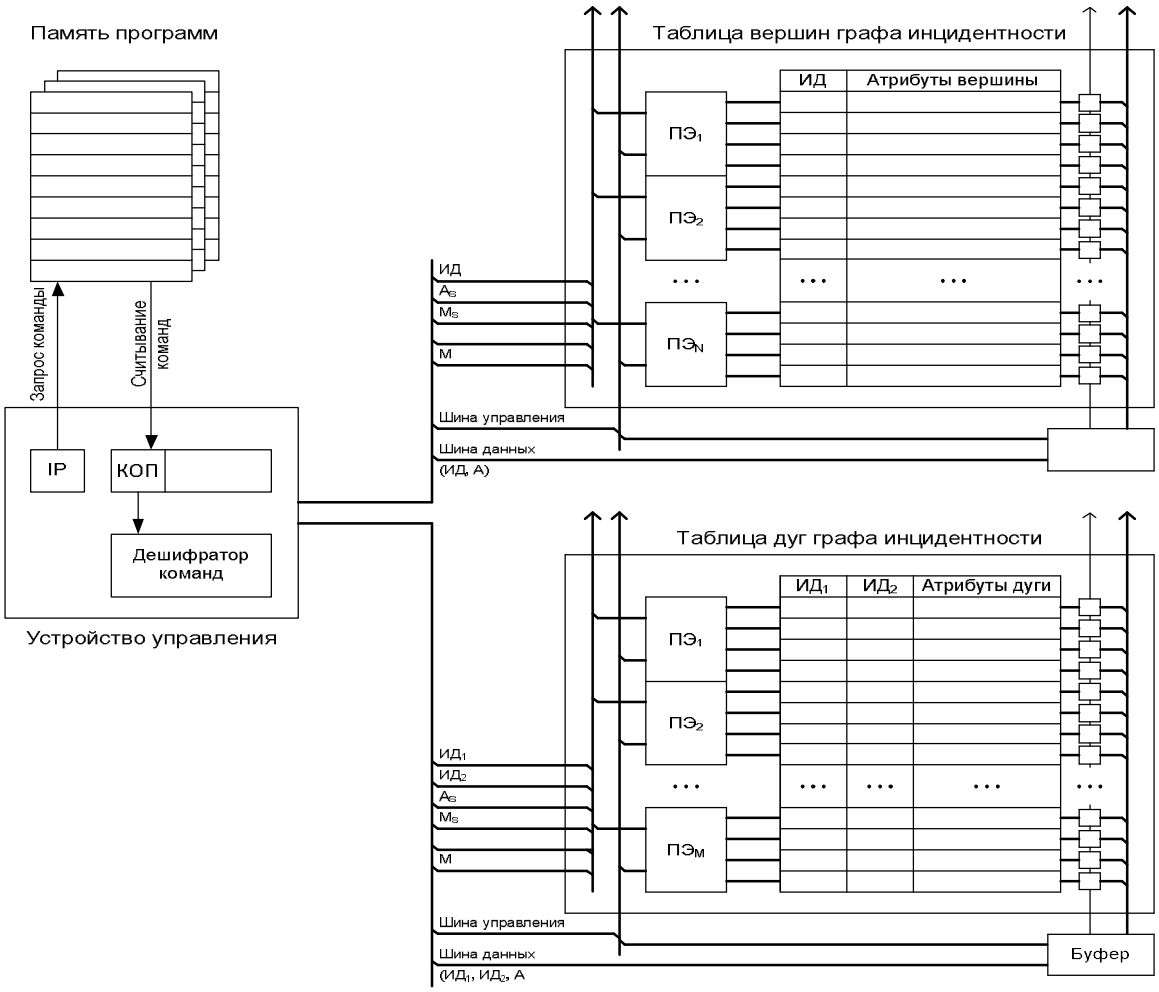

Рисунок 2 – Эскиз архитектуры ГДМ

В системе можно выделить общее устройство управления и множество процессорных элементов (ПЭ), параллельно исполняющих общую команду. Каждый ПЭ имеет свою независимую (локальную) память и отвечает за взаимодействие с ней. По аппаратной реализации ПЭ максимально упрощен (функционально сравним с простым компаратором), а это, в свою очередь, позволяет реализовать огромное их количество на одном чипе. Изменяя количество памяти, приходящееся на один ПЭ, можно регулировать различные характеристики системы, такие как объем обрабатываемой базы знаний, стоимость и т.п., чтобы достичь наилучшего соответствия техническим требованиям. В то же время, свойство регулярности производной графовой структуры, а следовательно, и исполняемых алгоритмов, позволяет обеспечить эффективное распараллеливание вычислительного процесса.

Первую экспресс-оценку эффективности процессора можно провести на операции ассоциативного поиска по ключу. В нашем случае, поиск представляет собой одновременный (параллельный) опрос устройством управления всех подключенных к нему ПЭ, что равносильно полному перебору всех знаний системы за одну условную единицу времени. С другой стороны, для последовательной архитектуры потребовалось бы перебрать все ячейки памяти, потратив на это N условных единиц времени. Очевидно, что увеличение количества ПЭ в предлагаемой системе приводит к линейному росту производительности системы в целом.

По своему основному функциональному назначению все команды процессора можно разделить на операции записи (добавление, удаление, изменение содержимого элемента) и чтения. В виду особенностей предлагаемой архитектуры (наращиваемости процессорных элементов, а значит и объема внутренней памяти, что соответствует количеству элементов в таблицах данных) наибольшую проблему с точки зрения реализации представляет операция чтения, которая не может быть выполнена параллельно. То есть, в случае необходимости прочитать результат некоторого поискового запроса, размер которого составляет более одного элемента, нам придется выполнить по одной операции чтения для каждого элемента.

В результате, основной задачей, которую необходимо было решить при построении системы команд процессора, – это обеспечение пользовательского интерфейса, позволяющего минимизировать количество необходимых операций чтения. Решением стало добавление аппаратной поддержки сложных (многоуровневых) поисковых запросов, что позволило пользователю считывать только конечный результат такого запроса, игнорируя промежуточные данные.

На рис. 3 представлена схема вывода данных процессора, позволяющая последовательно считывать содержимое ячеек таблицы. В общем случае приоритетом обладает ячейка, подключенная к шине данных выше остальных. Это означает, что если на команду поиска «ответило» более одного элемента таблицы, то при попытке считывания на шину данных будет подано содержимое наиболее приоритетной ячейки. В этом случае ко всем элементам, расположенным ниже, будет транслирован сигнал, который отключит их от шины данных. Данная схема обладает свойством наращиваемости, что согласуется с общей архитектурной концепцией процессора.

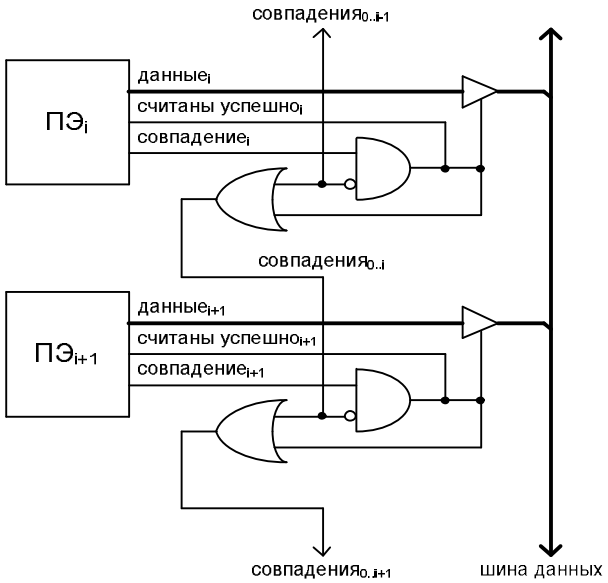

Рисунок 3 – Схема вывода с ячейки памяти процессора (элемента графа)

Система команд процессора

Рассмотрим общий принцип построения команд процессора. Для этого введем следующие обозначения:

– КОП – код операции, используется при дешифрации команды;
– ИД – поле идентификатора вершины;
– А – поле атрибутов вершины (дуги) графа инцидентности;
– М – поле битовой маски для атрибутов.

На рис. 4 приведена схема, согласно которой строятся команды процессора. Условно формат команды можно разделить на 3 части: код операции, адрес целевого элемента и аргументы операции. Код операции используется при дешифрации команды.

Под целевым элементом понимается ячейка (или множество ячеек) таблицы, на которую будет направлено действие операции. Адрес целевого элемента команды может быть представлен одним из двух способов: конкретным идентификатором вершины (или двумя идентификаторами, в случае обращения к дуге), либо поисковым запросом. Поисковый запрос описывается полями A_S и M_S, что соответствует обращению ко всем ячейкам, у которых биты поля атрибутов А, выделенные маской M_S, равны битам A_S. Фактически, поиск представляет собой побитовое сравнение каждым элементом графа с последующей реакцией в зависимости от типа операции. В случае работы с таблицей дуг также возможен вариант обращения сразу ко всем дугам, входящим в определенную вершину, либо выходящим из нее, для чего фиксируется один из соответствующих идентификаторов (ИД1 либо ИД2).

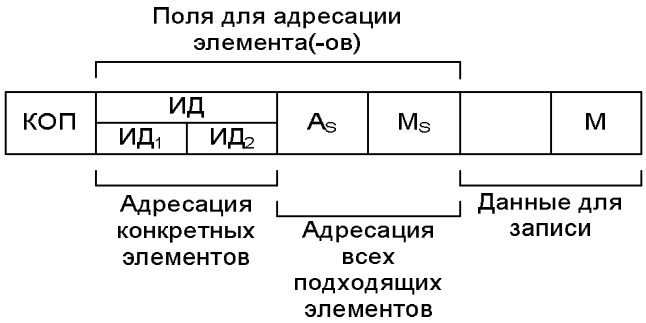

Рисунок 4 – Схема построения формата команды процессора

По аналогии, данные для записи в элемент также представлены двумя полями: атрибутами А и маской М, определяющей те биты поля А, которые будут записаны в целевой элемент.

Перечень всех доступных команд процессора представлен на рис. 5 и 6 для таблиц вершин и дуг соответственно.

Рисунок 5 – Команды доступа к таблице вершин

Подробнее рассмотрим реализацию поискового запроса процессора. В первую очередь выполняется операция записи для элементов графа, которые соответствуют какому-либо условию. Например, для тех элементов, которые содержат значение «5» в первом байте поля атрибутов, выполним запись единицы в i-й бит поля атрибутов (не включенный в байт со значением). Отмеченные таким способом вершины образуют множество не прочитанных результатов изначального поискового запроса (элементы со значением «5»). На втором этапе нам предстоит считать все результаты, для чего выполняется операция чтения для всех элементов со значением «1» в i-м бите поля атрибутов с одновременным сбросом в «0» i-о бита. В результате будет считана только одна ячейка памяти (по описанной выше схеме приоритетов), которая также будет исключена из множества не прочитанных результатов поискового запроса (из-за сброса i-о бита). Следовательно, следующая аналогичная команда чтения вернет очередной, не повторяющийся, элемент из результата первоначального поискового запроса. Таким образом, последовательно выполняя операцию чтения до тех пор, пока будет срабатывать хотя бы одна ячейка таблицы данных, мы получаем возможность считать весь результат поиска.

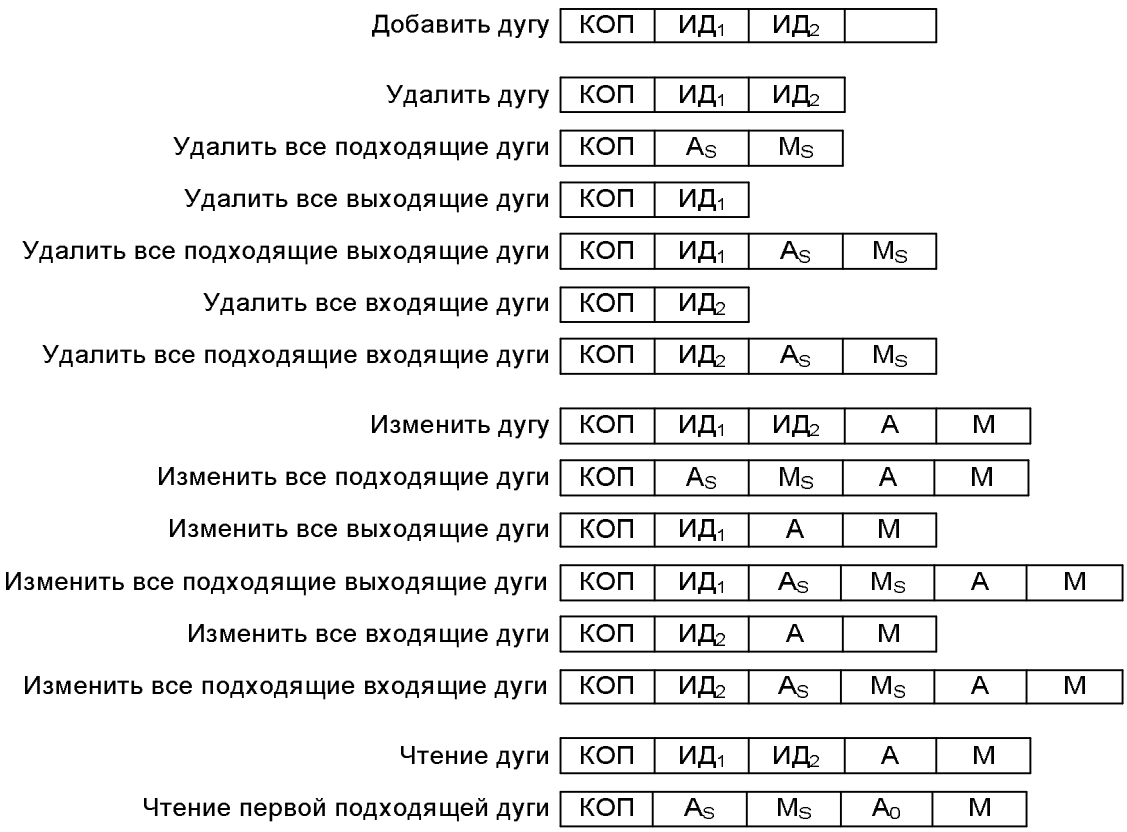

Рисунок 6 – Команды доступа к таблице вершин

Отметим, что предоставление побитового доступа к атрибутам элементов графа позволяет убрать зависимость формата данных процессора от решаемой задачи. Фактически ответственность за то, как интерпретировать атрибуты элементов ложится на программиста. Вместе с тем является возможной реализация некоторых сложных команд (очевидно, являющихся суперпозицией базовых) на уровне устройства управления, что позволяет «настраивать» процессор для более эффективного решения широкого круга задач.

Задача поиска кратчайшего пути в графе

Дан взвешенный граф *G(V,A)* без петель и дуг отрицательного веса. Необходимо найти кратчайшие пути от некоторой вершины *a* графа *G* до всех остальных вершин этого графа.

Введем следующие обозначения:

– *V* – множество вершин графа;

– *A* –множество дуг графа;

– *c[ij]*– стоимость (вес, длина) дуги *ij*;

– *a* – вершина, расстояния от которой ищутся;

– *U* –множество вершин графа;

– *W* – множество вершин, составляющих фронт волны;

– *d[u]* – по окончании работы алгоритма равно длине кратчайшего пути из вершины *a* до вершины *u*;

– *p[u]* – по окончании работы алгоритма содержит кратчайший путь из вершины *a* до вершины *u*.

Псевдокод предлагаемого алгоритма можно записать в следующем виде:

```
Присвоим d[a] ← 0, p[a] ← a
Занесем a в W
Пока ∃u∈W
    удалим u из W
    Для ∀v∈V, uv∈A
        если v∉U или d[v] > d[u] + c[uv] то
            занесем v в U
            занесем v в W
            изменим d[v] ← d[u] + c[uv]
            изменим p[v] ← p[u], v
```

В начале выполнения алгоритма расстояние для начальной вершины устанавливается равным нулю, а все остальные расстояния можно проигнорировать, так как полагается, что если вершина $v \notin U$, то $d[v]=\infty$. Далее формируется так называемый фронт волны, вначале состоящий только из одной вершины *a*. Затем запускается основной цикл.

На каждом шаге цикла происходит распространение волны от вершин, составляющих фронт волны, ко всем вершинам, соединенным с ними исходящими дугами. Расстояние до вершины рассчитывается как расстояние до предыдущей вершины плюс стоимость дуги между ними. В отличие от классического волнового алгоритма, мы рассматриваем каждую вершину не один единственный раз при достижении ее волной. Фактически каждая вершина *v* рассматривается для всех возможных путей волны в графе, но изменяет свое состояние (кратчайший путь) только если расстояние в ней (в *v*) больше, чем сумма расстояний до текущей вершины *u* и длины дуги *uv*. Все вершины, которые изменили свое состояние в текущей итерации цикла, составляют фронт волны для следующей итерации. Таким образом, цикл завершается, когда во всех вершинах будет установлено минимально возможное значение расстояния *d[u]*, что будет соответствовать итерации, на которой не будет изменена ни одна вершина, то есть волна затухнет.

Реализация алгоритма решения

Рассмотрим реализацию описанного выше алгоритма решения на основе архитектуры разрабатываемого векторного процессора. В первую очередь, произведем «настройку» процессора под решаемую задачу. Для этого уточним формат данных процессора (для каждого бита поля атрибутов поставим в соответствие некоторую переменную, используемую при решении) и расширим систему команд (в терминах используемого алгоритма).

ID	Атрибуты вершины				
	v	n	f	$distance_{min}$	prevID
N-1 0				M-1 0	N-1 0

ID1	ID2	Атрибуты дуги	
		c	cost
N-1 0	N-1 0		M-1 0

Рисунок 7 – Детализированный формат данных процессора (интерпретация полей атрибутов программистами ПО)

Детализируем формат данных процессора в соответствии с алгоритмом решения (см. рис. 7):

– ID, ID1, ID2 – идентификаторы вершин, N бит;

– $distance_{min}$ – минимальное найденное расстояние до вершины, M бит;

– prevID – идентификатор вершины, через которую был проведен путь с минимальным расстоянием (используется при построении пути-решения после окончания работы основного алгоритма), N бит;

– v – (visited) флаг, обозначающий, дошла ли волна до вершины (в случае, когда флаг установлен в «0», минимальное найденное расстояние до вершины принимается равным бесконечности), 1 бит;

– f – (wave front) вершины, обозначенные этим флагом, составляют фронт волны для текущей итерации алгоритма, 1 бит;

– n – (next wave front) вершины, обозначенные этим флагом, составляют фронт волны для последующей итерации алгоритма, 1 бит;

– c – (connected) флаг, используемый при поиске всех выходящих из вершины дуг, 1 бит.

Таким образом, размер одной ячейки таблицы вершин составит (N+2M+3) бит. Размер одной ячейки таблицы дуг – (2N+M+1) бит.

Введение дополнительных команд процессора, ориентированных на конкретный алгоритм решения, позволяет избежать работы напрямую с битами поля атрибутов элемента графа, тем самым упрощая конечное решение задачи программистом. Список полученных таким образом команд приведен в табл. 1.

Далее представлен фрагмент программы, реализующей рассмотренный алгоритм на базе построенной имитационной модели процессора. На вход функции поступают объекты vertexTable и arcTable, предоставляющие интерфейс для доступа к данным процессора (к таблице вершин и таблице дуг соответственно), а также вспомогательные методы для расшифровки поля атрибутов.

Таблица 1 –Расширенная система команд процессора

Команды для работы с таблицей вершин		
`createVertex(ID)`		Добавляет вершину в граф процессора.
`ID`	идентификатор новой вершины	
`init`		Для всех вершин графа процессора сбрасывает флаги v, n, f и переменные $distance_{min}$, prevID в ноль.
`setMinDistance(ID, distance, prevID)`		Записывает значение `distance` и `prevID` в вершину `ID`.
`ID`	идентификатор целевой вершины	
`distance`	минимальное расстояние до вершины	
`prevID`	идентификатор предыдущей вершины в пути	
`setWaveStartVertex(ID)`		Включает вершину `ID` во фронт волны (устанавливает бит f в «1»).
`ID`	идентификатор целевой вершины	
`readNextVertexFromWavefront`		Считывает следующую вершину из множества вершин, составляющих фронт волны (бит f установлен в«1»), одновременно сбрасывая бит в «0».
`readVertex`		Считывает вершину `ID`
`ID`	идентификатор целевой вершины	
`moveWavefront`		Для всех вершин, помеченных битом n, выполняет сброс бита n в «0» и установку бита f в «1».
Команды для работы с таблицей дуг		
`createArc`		Добавляет дугу (выходящую из вершины `ID1`, входящую в вершину `ID2`, со стоимостью `cost`) в граф процессора.
`ID1`	идентификатор вершины, инцидентной дуге	
`ID2`	идентификатор вершины, инцидентной дуге	
`cost`	цена дуги (вес, длина)	
`findAllOutputArcs`		Помечает все дуги, исходящие из вершины `ID`, флагом c.
`ID`	Идентификатор целевой вершины	
`readNextOutputArc`		Считывает следующую дугу из множества дуг, помеченных битом c, одновременно сбрасывая бит в «0».

Переменные ID1 и ID2 представляют идентификаторы вершин, путь между которыми необходимо найти. На выход функция возвращает булевую переменную, обозначающую, был ли найден путь.

```
bool findPath(MVertexTable&vertexTable, MArcTable&arcTable, uint ID1, uint ID2)
{
    vertexTable.init();
    vertexTable.setWaveStartVertex(ID1);
    while (true)
    {
        while (vertexTable.readNextVertexFromWavefront() == true)
        {
            MVertexsrcVertex = vertexTable.getBuffer();
            arcTable.findAllOutputArcs(srcVertex.m_ID);
            while (arcTable.readNextOutputArc() == true)
            {
                MArc arc = arcTable.getBuffer();
                uint distance = vertexTable.getMinDistance(srcVertex.m_attributes) +
arcTable.getCost(arc.m_attributes);

                vertexTable.readVertex(arc.m_ID2);
                MVertexdstVertex = vertexTable.getBuffer();
                if ((vertexTable.isVertexVisited(dstVertex.m_attributes) == false) ||
(distance <vertexTable.getMinDistance(dstVertex.m_attributes)))
                {
                    vertexTable.setMinDistance(dstVertex.m_ID, distance, srcVertex.m_ID);
                }
            }
        }
        if (vertexTable.moveWavefront() == false)
            break;
    }
    if (vertexTable.readVertex(ID2) == true)
    {
        MVertex vertex = vertexTable.getBuffer();
        returnvertexTable.isVertexVisited(vertex.m_attributes);
    }
    return false;
}
```

Заключение

В настоящей работе был изложен подход к построению прикладных интеллектуальных систем (систем, ориентированных на интеллектуальную обработку информации), основанный на принципе платформенной независимости. В качестве аппаратной реализации ГДМ (графодинамической машины) использован векторный проблемно-ориентированный процессор с оригинальной архитектурой.

Для верификации и оценки эффективности предлагаемой архитектуры была построена программная имитационная модель процессора, которая позволила детализировать формат данных и систему команд, апробировать основные принципы функционирования процессора в целом. В качестве целевой задачи была выбрана одна из типовых задач семантической обработки информации, в частности задача поиска кратчайшего пути в графе. Для ее решения был построен параллельный алгоритм решения (на основе алгоритма Дейкстры и волнового алгоритма), ориентированный на систему команд процессора.

В дальнейшем планируется продолжить детализацию возможностей процессора посредством имитационного моделирования других типовых задач семантической обработки. Планируется формализовать и автоматизировать процесс проведения тестовых испытаний, что позволит систематизировать получаемые результаты исследований прироста производительности при различных конфигурациях исходного графа. Конечной целью исследования является построение аппаратного прототипа семантического процессора с SIMD-архитектурой.

Литература

1. Голенков В.В. Графодинамические модели параллельной обработки знаний: принципы построения, реализации и проектирования / В.В. Голенков, Н.А. Гулякина // Открытые семантические технологии проектирования интеллектуальных систем : материалы II Междунар. научн.-техн. конф. (Минск, 16 – 18 февраля 2012 г.) – Минск : БГУИР, 2012. – С. 23-52.
2. John L. Gustafson. Reevaluating Amdahl's Law / John L. Gustafson // Communications of the ACM. – № 31(5). – P. 532-533.
3. Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities / Gene M. Amdahl // AFIPS Conference Proceedings (30). – P. 483-485.
4. Параллельные процессоры для построения интеллектуальных систем / [С.А. Байрак, Д. Н. Одинец, М.М. Татур и др.] // Открытые семантические технологии проектирования интеллектуальных систем : материалы II Междунар. научн.-техн. конф. (Минск, 16 – 18 февраля 2012 г.)– Минск : БГУИР, 2012. – С. 135-140.
5. Mikhail M. Tatur. Synthesis and Analysis of Classifiers Based on Generalized Model of Identification / [M. Tatur, D. Adzinets, M. Lukashevich, S. Bairak] // Advances in intelligent and soft computing. – 2010. – Vol. 71. – P. 529-536.
6. Вереник Н.Л. Разработка проблемно-ориентированных процессоров семантической обработки информации / Н.Л. Вереник, Е.Н. Сейткулов, М.М. Татур // Электроника инфо. – 2012. – № 8. – С. 95-98.

Literature

1. Vladimir V. Golenkov. Graphodynamical models of parallel knowledge processing / V.V. Golenkov, N.A. Guliakina // Open Semantic Technologies for Intelligent Systems (OSTIS-2012). – Minsk : BSUIR, 2012. – P. 23-52.
2. JohnL. Gustafson. Reevaluating Amdahl's Law / John L. Gustafson // Communications of the ACM 31(5). – P. 532-533.
3. Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities / Gene M. Amdahl // AFIPS Conference Proceedings (30). –P. 483-485.
4. Sergei A. Bairak. Parallel processors for intelligent systems development / S. A. Bairak, D. N.Adzinets, M. M. Tatur, P. Philipoff, M. Munoz // Open Semantic Technologies for Intelligent Systems (OSTIS-2012).– Minsk : BSUIR, 2012. – P. 135-140.
5. Mikhail M. Tatur. Synthesis and Analysis of Classifiers Based on Generalized Model of Identification / M. Tatur, D. Adzinets, M. Lukashevich, S. Bairak // Advances in intelligent and soft computing. – 2010. – Vol. 71. – P. 529-536.
6. Nick L. Verenik. Development of ASIP for semantic information processing / Nick L. Verenik, Yerzhan N. Seitkulov, Mikhail M. Tatur // Electronics info. – 2012. – № 8. – P. 95-98.

RESUME

N.L. Verenik, A.I. Girel, Y.N. Seitkulov, M.M. Tatur

Simulation Model of Vector Processor Solving the Problem of Path Finding in Graph

The article briefly describes the existing problem of effective parallel hardware platform creation which is oriented on certain class of problems related with semantic information processing. Using of ASIP is proposed to solve this issue [4], [5]. In many cases such solution is more technically and economically profitable than using of general-purpose CPUs, specialized processors or serial parallel processors (multi-core CPU, GPU, DSP).

Using of original developed SIMD-class architecture as typical processor architecture is proposed. In the article description of this architecture is given, the principles of data format and instruction set construction are considered.

In the paper classic problem of the shortest path finding in graph is solving on simulation model of processor using specially designed parallel algorithm. Source code of solution is given.

Статья поступила в редакцию 15.04.2013.