**Системы, управляемые потоком данных. Язык Dataflow Graph Language.**

**Курсовая работа Андреева М.В.**

**г. Ульяновск, 1999**

**Введение**

Одним из методов организации параллельных вычислений является метод, основанный основанный на принципе управления потоком данных. Обычно в вычислительных системах, управляемых потоком данных, команды машинного уровня управляются доступностью данных, проходящих по дугам графа потока данных (ГПД). Такому принципу управления потоком данных на уровне операций можно противопоставить принцип управления укрупненным потоком данных (Large-Grain Data Flow), в котором единица планирования вычислений крупнее (возможно, намного крупнее), чем одна машинная команда.

ГПД - одна из наиболее распространенных форм представления программы в данной модели вычислений. Вершины ГПД соответствуют отдельным процессам, а дуги задают отношения между ними. Точка вершины, в которую входит дуга, называется входным портом (портом импорта или входом), а точка, из которой она выходит, - выходным (портом экспорта или выходом). По дугам передаются данные из одного процесса в другой.

Данный метод заставляет программиста принять поэтапный подход к программированию, но, с другой стороны, избавляет от сложностей синхронизации, присущих большенству других моделей параллелизма.

**Программное обеспечение**

Система предназначена для работы в сети, в которой любые два компьютера могут обмениваться данными друг с другом. На любом компьютере может быть запушенно несколько процессов. Каждый процесс получает данные через порты импорта и может отслать данные через порты экспорта по дугам данных другим процессам.

Запуск программы осуществляется под управлением диспетчера, который распределяет процессы по компьютерам и устанавливает связи между процессами. Для нормальной работы диспетчера на всех компьютерах должна быть запущена специальная программа - монитор. Монитор по запросу диспетчера запускает процесс, указанный в запросе, на своем компьютере.

Порты импорта используются как очереди, и они, подобно каналам в ОС UNIX, буферизуют одно или неколько сообщений до тех пор, пока их не получит адресат. Объем буфера ограничен долько доступной емкостью памяти. Каждый порт импорта может быть связан с несколькими портами экспорта.

Порты экспорта могут иметь несколько каналов, число которых определяется диспетчером после анализа графа данных на этапе запуска процесса. Каждый канал обязательно связан только с одним портом импорта.

Подготовка прикладной программы к выполнению состоиз из следующих шагов:

конструирование графа потока данных программы

запись графа потока данных на языке графов данных DGL

обработка записи на языке DGL

написание прикладных программ для узловых процессов

компиляция узловых процессов в формат DLL

запуск узловых процессов диспетчером на основе DGL

**Пример параллельной программы**

В качестве примера расмотрим задачу приближенного вычисления числа Пи с использованием правила прямоугольников для вычисления определенного интеграла

где

Согласно правилу прямоугольников,

где , а .

Следует отметить, что это «процессорная» программа. Она не затрагивает многие проблемы параллельного программирования, например критическое влияние процессов ввода-вывода. Тем не менее эта задача поможет ознакомится с основными принципами построения программ, работающих в соответствии с методом управления потоком данных.

Существует множество подходов к решению контрольной задачи. Решение, приведенное ниже, иллюстрирует все основные шаги разработки программы.

**Конструирование графа потока данных программы**

Граф потока данных программы (или граф данных) определяет связи между процессами и дугами данных. Граф данных специфицирует все последуещее конструирование программы прикладной задачи. Его создание может потребовать немало усилий для определения того, как разбить программу на активизируемые данными процессы, чтобы достичь максимального увеличения скорости выполнения.

В пределе разрабатываемая программа может быть создана в виде одного процесса, но при этом теряется параллелелизм. Можно создать множество мелких процессов, таких как один оператор или даже одна арифметическая операция, что приведет к резкому увеличению расходов, связанных с запуском каждого процесса и обменом данных между ними. Следует отметить, что структура решаемой задачи часто наводит на хорошее первое приближение.

После того, как граф данных нарисован, каждый процесс, начало и конец каждой дуги помечаются буквенно-цифровым именем, которое используется в языке DGL. Если выход out имеет несколько каналов, то его i-й канал обозначается на схеме строкой out[i].

Для подсчета числа Пи используется несколько рабочих процессов, которые вычисляют свои части интеграла и пересылают результат суммирующему процессу. Рабочие процессы обращаются за очередным заданием к процессу распределения работ. Вся работа не распределяется заранее равномерно между процессами: один рабочий процесс, если он запущен на более быстрой машине, может выполнить львиную долю работы.

Из входа num\_iter процесс Summer считывает число частичных сумм, которые он должен просуммировать до завершения своей работы. На вход arg процесса Worker поступает задание: границы и число интервалов. Если число интервалов в задании равно нулю, то процесс завершает работу. Пересылая свой идентификатор через выход demand рабочий процесс обращается за очередным заданием.

**Запись графа потока данных на языке Data Graph Language**

Перевод графа потока данных в язык DGL совершается однозначным образом. В записи на DGL каждый процесс представлен заголовком и списком входных и выходных портов. При описании процесса можно использовать числовые константы, которые определяются в начале программы. Ряд констант задается диспетчером - константа nprocs, например, равна числу доступных процессоров в системе. Синтаксис языка DGL приведен в приложении А.

11 DATAFLOW GRAPH Pi;

12

13 NW = nprocs - 2

14

15 PROCESS Manager

16 EXPORT:

17 worker [NW] --> Worker [c]:arg;

18 num\_iter --> Summer:num\_iter;

19 IMPORT:

20 demand\_list;

21 END

22

23 PROCESS Worker [NW]

24 EXPORT:

25 demand --> Manager:demand\_list;

26 result --> Summer:part\_sum;

27 IMPORT:

28 arg;

29 END

30

31 PROCESS Summer

32 IMPORT:

33 num\_iter;

34 part\_sum;

35 END

**Запись программы вычисления Пи на языке DGL**

В строке 13 определяется константа NW - число рабочих процессов. Ее значение выбирается так, чтобы использовать для решения задачи все компьютеры сети.

В строке 23 описывается процесс Worker. Константа NW, расположенная в квадратных скобках после имени процесса, дает указание диспетчеру создать NW копий данного процесса. Причем, если значение NW меньше 1, то все равно создается одна копия. Все копии нумеруются, номер копии записывается в константу p, которая может быть использована при описании выходов процесса. Рассмотрим пример.

result 🡪 filter[2\*p+1]:arg

Данная запись означает, что выход result р-й копии процесса будет связан со входом arg (2р+1)-й копии процесса filter.

Запись в строке 17 означает, что выход worker процесса Manager будет иметь NW каналов. Причем, если значение NW меньше 1, то все равно будет создан один канал. Все каналы нумеруются, номер канала записывается в константу С. В примере С-й канал выхода worker связан со входом arg С-ой копии процесса Worker.

**Написание тела для каждого процесса**

Для каждого процесса нунжно создать файл-шаблон. Имя такого файла совпадает с именем процесса и имеет расширение frm (можно воспользоваться файлом Process.frm). В нашем случае имеем три файла: Manager.frm, Worker.frm и Summer.frm. В каждом файле есть процедура, имя которой заканчивается на Body. Внутри нее записывается тело процесса.

10 PROCEDURE ManagerBody;

11 VAR

12 Task : RECORD N:cardinal; a,b:real; END;

13 i,WrkId : cardinal;

14 CONST

15 N : cardinal = 10;

16 BEGIN

17 exportNumIter[0].Send (N, SizeOf(N));

18 Task.N := 10\*N;

19 Task.b := 0;

20 FOR i := 1 TO N DO BEGIN

21 Task.a := Task.b;

22 Task.b := i \* 1.0 / N;

23 importDemandList.Receive (WrkId, SizeOf(WrkId));

24 exportWorker[WrkId].Send (Task, SizeOf(Task));

25 END;

26 Task.N := 0;

27 FOR i := 1 TO exportWorker.NChannels DO

28 exportWorker[i-1].Send (Task, SizeOf(Task));

29 END;

 Файл Manager.frm : тело процесса Manager

Переменная Task описывает задание для рабочего процесса: a,b - границы, N - число интервалов. Константа N, описанная в строке 15, равна числу разбиений отрезка [0;1].

В начале работы посылаем процессу Summer число разбиений N (строка 17) . В строке 23 ждем запроса от одного из рабочих процессов. Запрос представляет собой идентификатор запрашивающего процесса. Получив запрос, отсылаем очередное задание соответствующему рабочему (строка 24).

После того, как задания распределены, нужно сообщить об этом всем рабочим процессам. Для этого служат строки 26-28: по всем каналам порта exportWorker посылаем задание с нулевым числом интервалов - сигнал о завершении работы.

30 PROCEDURE WorkerBody;

31 VAR

32 Task : RECORD N:word; a,b:real; END;

33 S : real;

34 i : word;

35 FUNCTION f(x:real):real;

36 BEGIN

37 Result := 4 / (1 + x\*x);

38 END;

39 BEGIN

40 exportDemand[0].Send (FloLib.CopyNumber, SizeOf(cardinal));

41 WHILE (true) DO WITH Task DO BEGIN

42 importArg.Receive (Task, SizeOf(Task));

43 IF (Task.N = 0) THEN EXIT;

44 h := (b-a)/N;

45 S := 0;

46 FOR i := 1 TO N DO

47 S := S + f(a+(i-0.5)\*h);

48 S := h\*S;

49 exportPartSum[0].Send (S, SizeOf(S));

50 exportDemand[0].Send (FloLib.CopyNumber,SizeOf(cardinal));

51 END;

52 END;

 Файл Worker.frm : тело процесса Worker

Бесконечный цикл 41-51 обеспечивает работу процесса до получения сигнала завершения от распределителя работ Manager.

В строке 42 ждем очередное задание Task. Если число интервалов в задании равно 0, то завершаем работу. В противном случае вычисляем частичную сумму на интервале (Task.a; Task.b) и отсылаем ее суммирующему процессу (строки 44-49). В строке 50 обращаемся к распределителю работ за очередным заданием.

53 PROCEDURE SummerBody;

54 VAR

55 N, i : cardinal;

56 F : TextFile;

57 TotalSum, S : real;

58 BEGIN

59 importNumIter.Receive (N, SizeOf(N));

60 TotalSum := 0;

61 FOR i := 1 TO N DO BEGIN

62 importPartSum.Receive (S, SizeOf(S));

63 TotalSum := TotalSum + S;

64 END;

65 AssignFile (F, ‘Pi.result’);

66 Rewrite (F);

67 WriteLn (F, ‘Pi = ’, TotalSum);

68 CloseFile (F);

69 END;

 Файл Summer.frm : тело процесса Summer

В строках 61-64 собираются частичные суммы от всех рабочих процессов и суммируются в переменной TotalSum. Число частичных сумм записываем в переменну N из порта importNumIter (строка 59).

**Компиляция узловых процессов**

После того, как созданы шаблоны, нужно получить из них файлы, пригодные для компиляции. Для этого используется компилятор с языка DGL:

 dglc Pi.dgl

Компилятор, если нет ошибок, сгенерирует следующие файлы: Pi.dpr, Manager.pas, Worker.pas, Summer.pas.

**Загрузка и выполнение программы**

Сначала на компьютерах сети нужно запустить программу-монитор. Перепишем откомпилироанные файлы и файл Pi.dgl с текстом графа потока данных на языке DGL в один каталог и запустим диспетчер, указав Pi.dgl в качестве параметра. После окончания работы диспетчера должен появится файл Pi.result, в котором записано приближенное значение числа Pi.

**Приложение А**

**Синтаксис языка DGL**

DGL = ["DATAFLOW GRAPH" [identifier] ";"]

 {Definitions}

 {ProcessDecl}

Definitions = identifier "=" ConstExpr

ProcessDecl = "PROCESS" identifier ["AT" path]

 ["[" NumCopies "]" ]

 {"EXPORT:"{ExportDecl} |

 "IMPORT:"{ImportDecl}

 }

 "END"

ExportDecl = identifier ["[" NumCopies "]"]

 "-->"

 identifier ["[" Expression "]"]

 ":"

 identifier ";"

ImportDecl = identifier ";"

NumCopies = ConstExpr

ConstExpr = Expression

Expression = Term [AddOp Term]

Term = Fact [MulOp Fact]

Fact = number | identifier | "(" Expression ")"

AddOp = "+" | "-"

MulOp = "\*" | "/"

Замечания:

number - целое положительное число

все операции языка целочисленные

значение выражения NumCopies должно быть больше нуля, в противном случае оно заменяется на число 1

в выражениях можно использовать следующие переменные: с - номер текущего канала, р - номер текущей копии процесса

**Список литературы**

[1] Роберт Бэб, «Программирование на параллельных вычислительных системах» - Москва: Мир, 1991

[2] А.И.Водяхо, «Высокопроизводительные системы обработки данных» - Москва:Высшая школа, 1997