О применении формализма потока данных в разработке параллельных и распределённых программ

М. О. Бахтерев

В докладе рассказывается о формализме потока данных и о конструировании языка программирования и операционной системы, в которых было бы удобно разрабатывать программное обеспечение в рамках этой парадигмы.

Как программировать для GRID систем, состоящих из кластеров разной архитектуры, в которых работают многопроцессорные NUMA узлы, в которых каждый процессор является многоядерным, и в которые встроены специализированные сопроцессоры, которые, скорее всего, тоже имеют не одно ядро? Программирование для таких систем может быть крайне трудоёмким.

И проблема даже не в том, что программы откомпилированные для процессоров одной архитектуры не будут исполнятся устройством с другим набором комманд. Современные компиляторы помогают с этим сладить. Но как справляться с тем, что у различных вычислительных устройств различные методы взаимодействия со внешним миром? И как справляться с тем, что внешний мир современному процессору представляется крайне интересной и неоднородной структурой?

Можно представить себе простую современную систему (см. Рис. 1), которая уже сейчас может состоять из двух двухядерных процессоров, у каждого из которых доступ к локальной памяти осуществляется несколько быстрее, чем к памяти соседа. Пусть к этой системе, к одному из центральных процессоров через PCI-Express подключён вычислительный ускоритель на базе Cell, который, скорее всего, взаимодействует с основной системой через виртуальные tcp/ip каналы. При этом, для управляющего процессорного ядра Cell, вычислительные сопроцессоры выглядят скорее как устройства ввода\вывода.

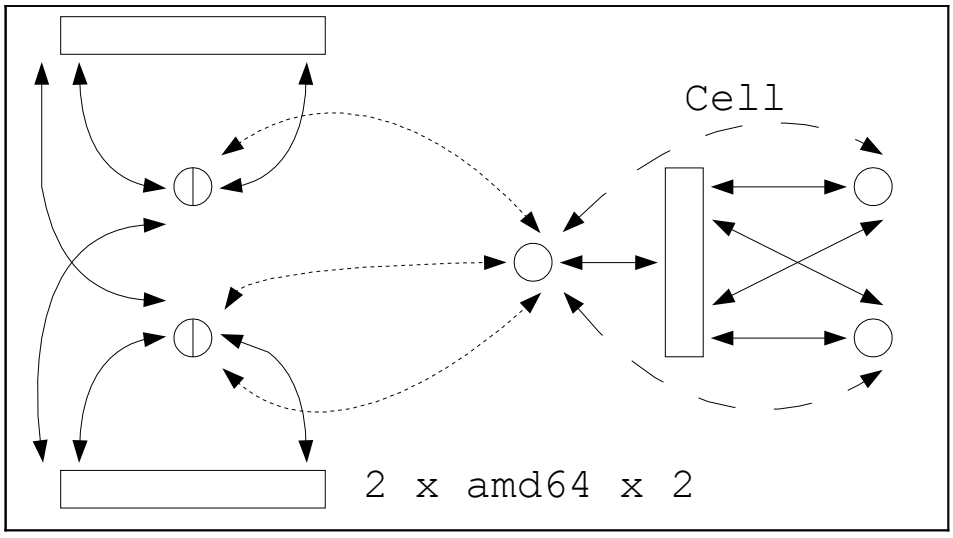

Рис. 1. Схема узла современного суперкомпьютера

Для получения эффективной программы при помощи современных средств разработки программист должен.

(1) На amd64 при помощи NUMA API явно управлять памятью и задачами так, чтобы обеспечивать высокую скорость доступа к данным. OpenMP с его очень простой SMP моделью параллелизма для этих целей не подойдёт.

(2) Обеспечить связь с частью приложения, работающей на Cell, при помощи асинхронной посылки сообщений через MPI.

(3) Через систему асинхронного ввода/вывода POSIX AIO наладить обмен информацией с вычислительными ядрами Cell.

(4) Попытаться всё сделать так, чтобы избежать конфликтов при одновременном доступе к коммуникационным устройствам и памяти.

Насыщенный план действий. Когда же программисту решать свою интересную прикладную проблему? Но даже если время будет найдено, то удручать начнёт тот факт, что при любой попытке изменить принцип распараллеливания программы в большинстве случаев необходимо будет заново выстраивать все алгоритмы для указанных выше задач.

Предпринимаются попытки уложить всё современное топологическое разнообразие архитектур в рамки одного подхода – MPI/OpenMP. И этот подход теоретически позволяет создавать эффективные программы в ряде случаев, но в большинстве ситуаций это не так.

Open MP ориентирован на SMP системы, и не позволяет при описании алгоритма учесть неоднородности, как в аппаратуре, так и в самих алгоритмах и обрабатываемых данных.

MPI лишён этих недостатков, но в его основе лежит концепция копирования данных между различными процессами. А копирования данных значительно снижают производительность в системах с общей памятью. Кроме этого, для организации асинхронного обмена информацией в MPI могут потребоваться логически сложные процедуры. Которые ещё более усложнятся, если понадобится обеспечить балансировку нагрузки между вычислительными модулями в соответсвии с особенностями алгоритмов или данных. Выходит, что вместо решения прикладной задачи программист тратит время на решение системных вопросов.

Возможно, нужен другой подход.

Варианты существуют. Здесь можно упомянуть системы похожие на DVM или NORMA, в которых параллелизм можно задавать указывая на то, как можно разбивать подрасчётные области данных на независимые участки. Но и у этих систем имеются существенные ограничения. Во-первых, они не позволяют динамически переразбивать области в ходе вычислений. Во-вторых, возможности по описанию разбиений ограничены некоторым набором регулярных структур.

Возможно ли предложить подход получше?

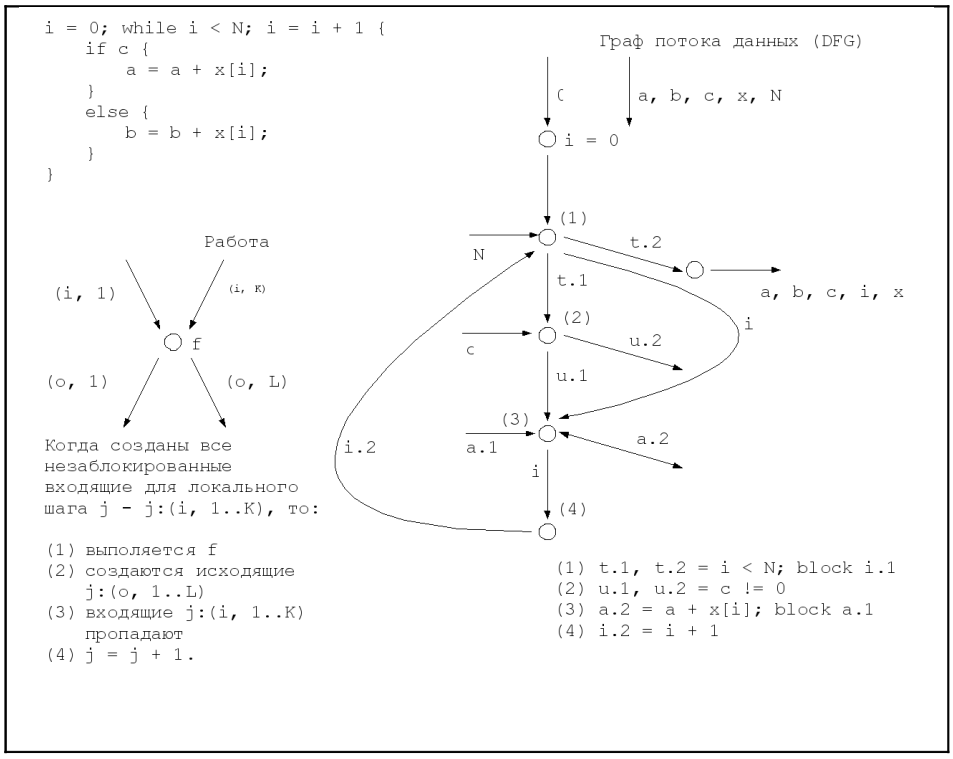

Рис. 2. Программа и её граф потока данных

Структуру, изображённую на Рис. 2. можно найти не только на мелком масштабе в обычных программах. В неявном виде она присутствует везде.

Далее идут примеры конструкций в различных парадигмах программирования. OpenMP (см. Рис. 3.1. и 3.2. на этих рисунках m = N / M,A где M – число процессоров, пусть m – целое), MPI и Microsoft Direct3D (Рис 4.(1) и 4.(2)).

```
F(o,N)

#parellel cycle
i = 0; while i < N; i = i + 1 {
      x[i] = f(x[i], y[i], z[i]);
}
```

Рис. 3. (1) Схема OpenMP программы

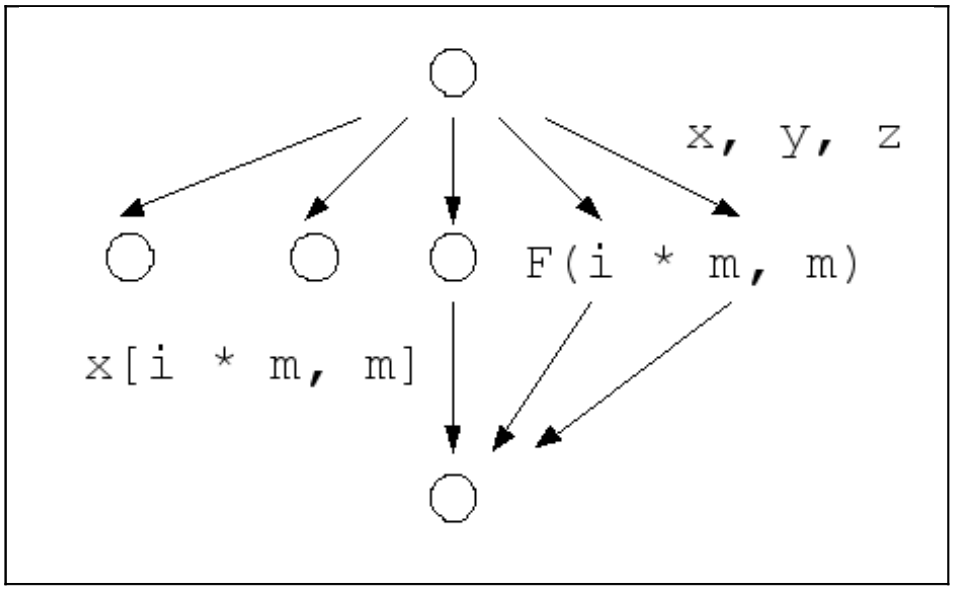

Рис. 3. (2) граф потока OpenMP программы

```
MPI
receive(i.k.1, ..., i.k.(N.k));
{o.k.1, ..., o.k.(M.k)} = f.k(i.k.1, ..., i.k.(N.k));
send(o.k.1, ..., o.k.(M.k));

Direct3D
createbuffer(b.1, b.2);
createsurface(s.1);
setshader(shader);
setstreamsource(b.1, b.2);
drawprimitive(p);
```

Рис. 4. (1) Схемы MPI и Direct3D программ

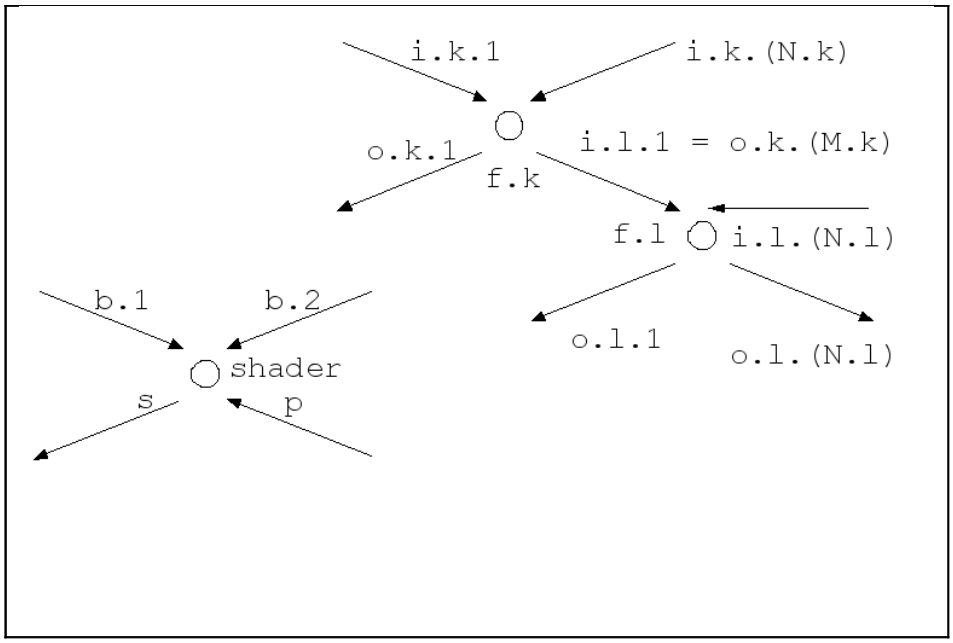

Рис. 4.(2) графы потокв программ с Рис. 4.(1)

Возможно повсеместная распространённость этой структуры связанна с тем, что от графа потоков данных очень просто перейти к схемам выводов из теории типов. Это очень просто сделать автоматически. Например, для верхнего графа на Рис. 4. (2)

Пусть метка каждой дуги – x.y.z – это тип. Вполне законное предположение, потому что каждая дуга символизирует пронумерованный список неких значений.

```
t.i.k = (i.k.1, ..., i.k.(N.k))
t.o.k = (o.k.1, ..., o.k.(M.k))
t.o.s = (i.l.2, ..., i.l.(N.l))
t.o.l = (o.l.1, ..., o.l.(N.l))
```

Рис. 5. дуги, как типы

Тогда можно построить такую схему теории типов.

```
f.k: t.i.k -> t.o.k
```

Рис. 6. вывод соответсвующий преобразованию данных работой f.k

Тогда можно построить такую схему теории типов.Чтобы 'сшить' f.k и f.l нужно формально ввести следующие схемы, которые просто вычисляются из DFG программы.

```
t.o.k -> i.l.1
i.l.1 -> (t.o.s -> t.o.l)
```

Рис. 7. склеивающие работы f.k и f.l схемы выводов

Если теперь объявить t.o.l терминальным типом, а t.i.k и t.o.s - базовыми (это так же делается при помощи простого вычисления по графу), то получается законченная теория типов. А так как известна теорема о изоморфизме Кэрри-Говорда, то можно утверждать, что DFG недалеко ушёл от одной из базовых абстракций в теоретическом программировании – lambda-

вычислений. Вероятно, именно поэтому его очень легко разглядеть в программах решающих совершенно разные задачи. При некотором навыке даже в такой вот программе на языке Haskell 98.

```
fib = 1 : 1 : [a + b | (a, b) <- zip fib (tail fib)]
```

Рис. 8. описание списка на Haskell

Впрочем, математическая логика вопроса не проработана.

Далее можно задаться следующим вопросом. Если примитив DFG вездесущий, то нельзя ли программировать с явным его использованием? Можно. Примеров подобных систем множество: от средств визуального программирования до систем цифровой обработки сигналов. Пара самых распространённых из них - это конвейреы UNIX (UNIX pipes) и Microsoft DirectShow (DS).

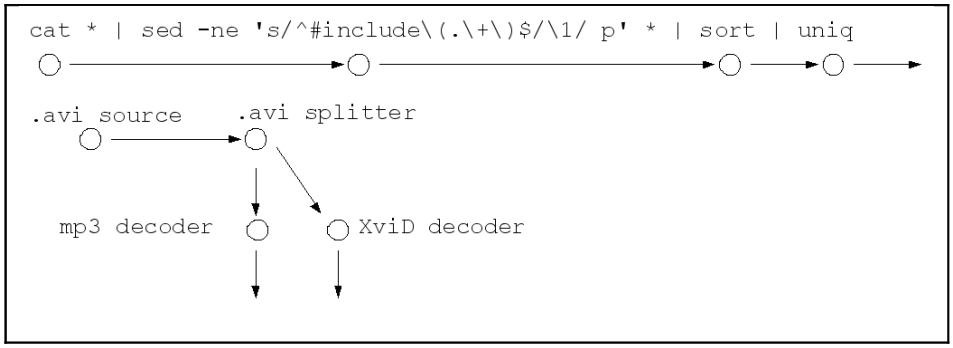

Рис. 9. примеры графов потоков данных, явно используемых в UNIX shell и MS DirectShow

Важным моментом здесь является следующее. Существует внешняя к программам, работающим в узлах графа, среда, связывающая их в соответствии с заданным описанию графа. В UNIX это интерпретатор комманд (shell), в DS – это FilterGraph. После создания среды взаимодействия, программы обращаются к ней для взаимодействия через небольшой набор всегда фиксированных примитивов. В UNIX – это стандартные потоки ввода\вывода, а в DS - контакты (pins).

Преимущества. Программы независимы друг от друга, а это не только высокий уровень гибкости и масштабируемости системы, но и возможность выполнять программы параллельно без всяких на то инструкций в их коде.

Важная особенность. Обе среды взаимодействия программируются при помощи описания DFG программы, но способы взаимодействия различны. В UNIX – это вариант пересылки сообщений, а в DS – это общая память.

Нужно упомянуть и схему 'мастер – рабочие', которая часто используется в программировании с MPI. (см. Рис. 10.)

Здесь пересылка сообщений с логической (архитектурной, системной?) точки зрения используется лишь для того, чтобы рабочий мог получить от мастера новые данные для обработки и сообщить о результате вычислений. То есть, рабочий – это типичный узел в DFG, а мастер – это то, что обеспечивает 'доставку новых стрелок' к этому узлу. Основная логическая обработка процесса вычислений не связана с абстракциями MPI в этом случае, а полагается именно на понятие работы.

Хороший пример – демонстрационная программа из пакета MPICH, вычисляющая в некотором приближении множество Мандельброта. Кроме прочего эта программа демонстрирует то, как можно балансировать нагрузку в системе, основываясь на очень простом анализе того времени, которое потребовалось на выполнение очередной работы. Этот простой анализ сразу учитывает много факторов: загруженность каналов передачи данных, быстродействие процессоров, загруженность узлов машины другими вычислениями,

особенности обрабатываемых данных, особенности алгоритма, и так далее – всё, что связано со скоростью расчётов.

.

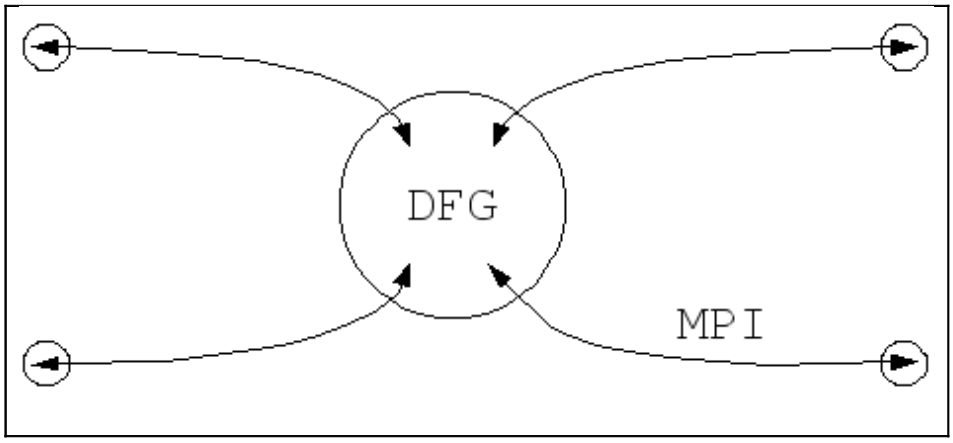

Рис. 10. программирование с мастером и рабочими

Данный документ представляет собой инструкцию для пользователей MS Word (вы можете также воспользоваться инструкцией для пользователей LaTeX) и его можно использовать в качестве шаблона.

Итак, использование DFG в программировании параллельных программ может дать такие преимущества.

(1) Можно использовать одну и ту же программу для выполнения её на системах как с общей, так и с распределённой памятью. При этом использовать асинхронные обмены данными. Более того, весьма вероятно, такие программы можно выполнять на системах смешанного типа, на тех самых GRID-кластерах с многоядерными процессорами.

(2) Скорее всего возможно некоторое решение задачи если не об оптимальном, то об эффективном распределении работ между узлами такой системы. Методики здесь могут быть похожи на алгоритмы выделения регистров (register allocation), используемые в оптимизаторах компиляторов.

(3) Существует возможность балансировать нагрузку в системе с учётом многих факторов, в том числе, особенностей обрабатываемых данных и используемых алгоритмов.

(4) Можно освободить программиста от управления ресурсами. По описанию DFG просто вычисляется то, сколько нужно выделить памяти и когда её освободить, когда и какими данными нужно обменяться узлам, в какие именно файлы нужно записывать данные. Программы могут быть написаны без системных вызовов malloc, send, receive, wait, read, write и прочих.

(5) Из-за того, что программа распадается на множество независимых вычислительных компонент, появляется возможность манипулировать самим процессом вычисления: создавать контрольные точки, отслеживать вышедшие из строя узлы и перераспределять работы на другие. И это без усилий програмиста.

Всё это весьма неплохие причины для того, чтобы попробовать создать систему для программирования в модели DFG. Но как нужно писать в этой модели параллельные программы? Существующие способы не подходят из-за их специализации для решения других задач или громоздкости.

Как бы могло выглядеть программирование в рамках этой модели? Можно рассмотреть пару модельных задач. Первая из них – программирование простейшего итерационного процесса. Пусть x – вектор из R(N), A – матрица из R(N, N). Пусть нужно запрограммировать следующее вычисление.

```
x = x0; while undone(x) { x = A * x; }
```

Рис. 11. Простейший итерационный процесс

Пусть в системе существует M вычислительных модулей и N / M – целое. Программа в рамках парадигмы DFG могла бы быть такой.

```
перестройка DFG.

m = N / M;
x = x0;
while undone(x) {
      i = 0; while i < M; i = i + 1 {
            y[i * m, m] = node f(A[i * m, m], X, m);
      }
      x = y;
}

Вычислительная компонента.

f(A, x, m, y) {
      i = 0; while i < m; i = i + 1 {
            y[i] = (A[i], x);
      }
}
```

Рис. 12. программа в рамках DFG

Сложнее, чем OpenMP и DVM, но существенно проще MPI.

А если задачу усложнить? Как пересчитать поле нейронов? Решение этой задачи полезно, так как зависимости по данным между нейронами могут быть любыми. На Рис. 13. nf – это поле нейронов. nf[x].d – множество нейронов, от которых зависит нейрон x, представленное в виде списка отрезков [start, length].

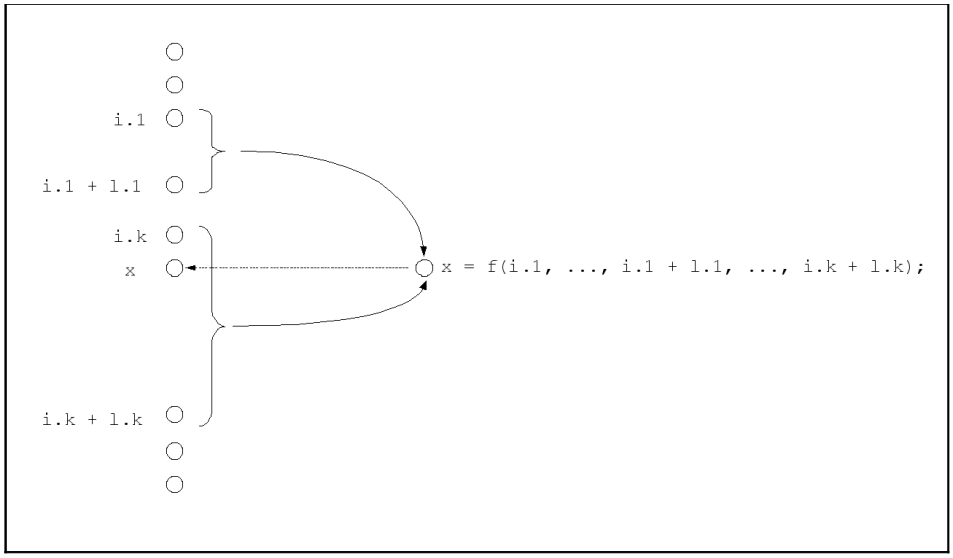

Рис. 13. схема нейронной сети

Пересчёт мог бы производится при помощи программы на Рис. 14. И так как зависимости между нейронами в нейронной сети произвольные, то, раз решение удалось найти, и за каждым оператором в нём стоит вполне определённо программируемый смысл, скорее всего, примитив для разбиения и сшивания данных построить можно. Нужен дополнительный анализ алгоритмов, использующих сетки, но оптимизм ситуация внушает.

```
программа DFG.

m = N / M;
i = 0; while i < m; i = i + 1 {
      D = {};
      j = 0; while j < m; j = j + 1 {
            D = union(D, nf[i * m + j].d);
      }
      y[i * m, m] = node f(nf[D], i, m);
}
nf = y;

программа вычислительной компоненты.

f(nf, j, m, y) {
      i = j; while i < j + m; i = i + 1 {
            sum = 0;
            cs = nf[i].d.head; while cs; cs = cs.next {
                  z = nf[cs.i, cs.l];
                  k = 0; while k < cs.l; k = k + 1 {
                        sum = sum + z[k];
                  }
            }
            y[i – j] = analisys(sum);
      }
}
```

Рис. 14. пересчёт состояния нейронной сети

Впрочем, пока это занимательные теоретические построения. Но кроме теорий и идей, есть и некоторые практические достижения, которые связаны с разработкой этих самых теорий.Основная практическая выгода получается в ходе разработки такой операционной системы, которая предоставила возможность DFG-программам работать максимально эффективно.

Какими для этого должны быть следующие подсистемы?

(1) низкоуровневое управление задачами
(2) ввод/вывод
(3) управление памятью
(4) взаимодействие между задачами

Последнее имеет прямое отношение к обсуждаемому подходу в программировании.

При проектировании примитива взаимодействия между нитями (в данной ОС именно нить является задачей) ставилась задача, чтобы этот примитив мог схватить понятие работы. То есть, он должен позволять сообщить о завершении работы и готовности данных, а так же запускать новые вычисления, когда все необходимые для этого данные получены.

Это сделано в следующей форме (рисунки 15 и 16).

Примитив синхронизации получил называние r3 – request/receive/reply. И суть работы его заключается в следующем. Появление входящей стрелки приводит к разблокированию соответсвующей нити, если она была заблокирована, сделав системный вызов request (его может сделать только ipoint-нить, и она всегда блокируется на нём) или receive (его может сделать только xpoint-нить, которая заблокируется, если очередь запросов пуста). Запросы попадают в очередь с приоритетами.

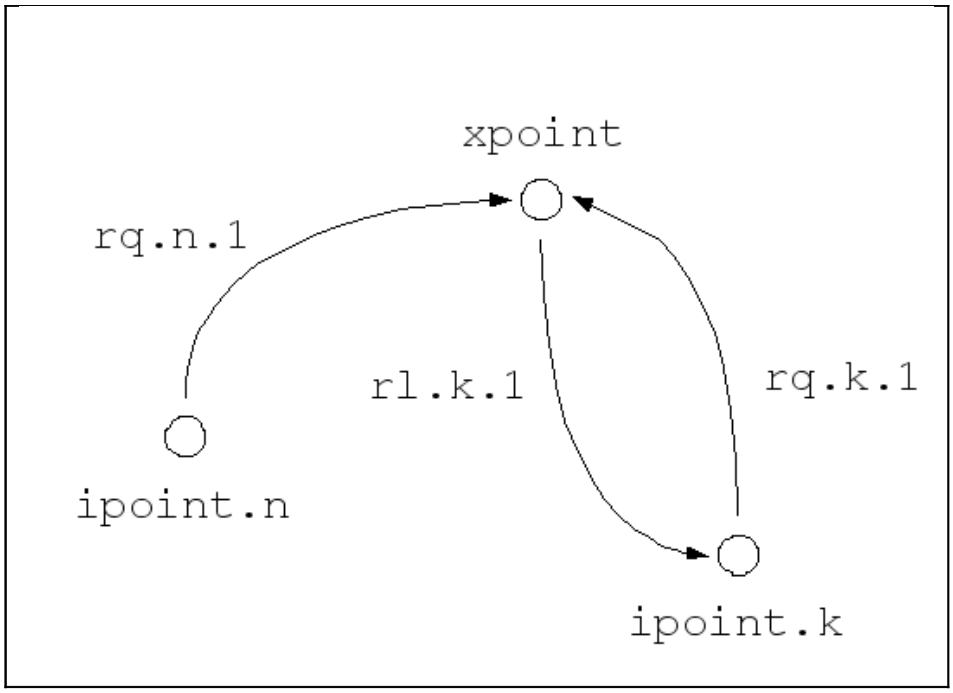

Рис. 15. схема взаимодействия нитей через примитив r3

```
xpoint

while undone {
      r3:receive(prioritymask, .rq, .value);
      rql = process(rq, value, .undone);

      rq = rql.head; while rq; rq = rq.next {
            r3:release(rq);
      }
}

ipoint

while undone {
      r3:request(xid, priority, pointer);
      process(pointer, .undone);
}
```

Рис. 16. схемы программ, выполняющимися ipoint и xpoint нитями

Преимущества.

(1) Очень просто реализовать надёждный вариант. Надёжный вариант r3 гораздо проще в реализации, чем реализация надёжного mutex'a. Это особенно заметно в MP системах. Под надёжностью понимается такой режим работы с примитивом, в котором использующая его нить получает уведомления о всех нештатных ситуациях. Например, не блокируется на уничтоженном mutex'e, как это происходит до сих пор в windows, а получает информацию о том, что примитив уничтожен.

(2) В таком взаимодействии отсутсвуют несколько неприятных ситуаций, которые возникают в многозадачных системах. Например, инверсия приоритетов.

(3) Механизм очень мощный. На стороне xpoint-нити может работать любая программа. Это позволяет выразить через r3 множество схем взаимодействия: от ненадёжных семафоров до программных каналов.

Недостатки.

(1) Примитив не предназначен для организации небольших критических секций. Со всеми вытекающими последствиями.

(2) Нужна дополнительная нить для организации взаимодействия.

НО. Обычно, все маленькие критические секции, при разработке программы в DFG-парадигме переносятся в контролирующую (xpoint) нить. Кроме этого, хотя для взаимодействия и нужна дополнительная нить, сами программы, которые выполняются взаимодействующими нитями получаются более простыми.

С чем это связано? Не известно. В классических математических моделях для взаимодействующих нитей (сети Петри, темпоральная логика Лампорта, pi-вычисления), не получилось найти ответа. Возможно, нужен какой-то другой способ анализа взаимодействующих сущностей.

Теоретически не понятно, почему всё хорошо с примитивом r3, но практика даёт множество подверждений. Одна из недавно решённых задач, где использовалась эта схема взаимодействия – система для обмена сообщениями в реальном времени. Её место в неком изделии показано на Рис. 17.

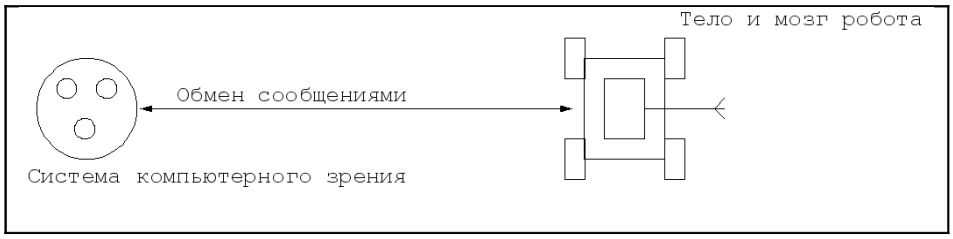

Рис. 17. схема взаимодействия нитей через примитив r3

Как система распознавания образов так и робот – это множество взаимодействующих задач. Поэтому крайне важно было добиться следующих свойств от системы коммуникаций.

(1) Система должна обрабатывать множественные асинхронные запросы с учётом времени.

(2) Пропускать принятые сообщения с минимальными задержками.

Условия работы комплекса далеко неидеальные. Множество помех. Поэтому.

(3) Устойчивость к помехам на линии.

(4) Контроль потока данных.

Так как система работает в реальном времени, то.

(5) Синхронизация часов.

В рамках подхода с DFG, с разбиением задач на асинхронные работ. С программированием (пока вручную) поведения среды взаимодействия с данным графом потока данных. С использованного рождённого в рамкой этой модели примитива r3 для программирования взаимодействия параллельных компонентов задачу удалось решить в не таком уж и сложном и относительно небольшом коде – всего примерно 4000 строк на языке СИ, считая реализованный поверх Win32 и POSIX примитив r3 и созданную вручную среду взаимодействия.