# Технология PVM

Parallel Virtual Machine (PVM) (дословно виртуальная параллельная машина) — общедоступный программный пакет, позволяющий объединять разнородный набор компьютеров в общий вычислительный ресурс («виртуальную параллельную машину») и предоставляющий возможности управления процессами с помощью механизма передачи сообщений. Существуют реализации PVM для самых различных платформ от лаптопов до суперкомпьютеров Cray. Имеет более расширенные возможности, чем её популярный аналог MPI, в плане контроля вычислений: присутствует специализированная консоль управления параллельной системой и её графический эквивалент XPVM, позволяющий наглядно продемонстрировать работу всей системы.

PVM — плод совместного сотрудничества Окриджской национальной лаборатории, Университета штата Теннесси и Университета Эмори. Работа над проектом началась в Окриджской национальной лаборатории летом 1989 года, и в том же году была выпущена PVM 1.0. Разработкой занимались сотрудники лаборатории Vaidyalingam S. Sunderam и Al Geist. PVM 1.0 использовалась только внутри Лаборатории и не предназначалась для распространения. Версия 2.0, переписанная сотрудниками Университета штата Теннесси, вышла в марте 1991 года и развивалась до версии 2.4. Версия 3, выпущенная в марте 1993 года, была полностью переписана с нуля, поддерживала устойчивость к сбоям (fault tolerance) и проще портировалась на другие платформы. Последней версией PVM является версия 3.4.6, выпущенная в феврале 2009 года.

PVM поддерживает программирование на языках Fortran, C и C++ путём предоставления специальных библиотек.

PVM является свободным ПО и распространяется под двумя лицензиями: BSD Licence и GNU General Public License.

## Методика проектирования программ на PVM:

* алгоритм предварительно приводится к параллельному виду и записывается в виде нескольких ветвей,
* ветви могут между собой обмениваться данными в виде сообщений,
* ветви могут пересекаться и в пересекающихся частях синхронизировать друг с другом свое выполнение,
* все ветви алгоритма сводятся в единую программу (приложение),
* в ОЗУ программа загружается и выполняется в количестве экземпляров, равном количеству ветвей в алгоритме. Эти экземпляры называются задачами (как и в Юниксе),
* задачи объединяются в группу. Каждая группа в PVM объединяет все задачи, работающие над одним алгоритмом, и только их. Возможностей для взаимодействия соседям по группе PVM предоставляет намного больше, чем просто двум задачам,
* каждая задача в зависимости от своего порядкового номера в группе приступает к выполнению той или иной ветви.

В системе PVM каждая задача (которой соответствует исполняемый файл), запущенная на некотором процессоре, идентифицируется уникальным целым числом, которое называется идентификатором задачи (далее используется обозначение TID). Копии одного исполняемого файла, запущенные параллельно на N процессорах PVM, создают N задач с разными TID.

TID - идентификатор задачи (целое число, характеризующее исполняемый файл, запущенный на некотором процессоре).

По смыслу похож на идентификатор процесса (PID) в операционной системе Unix. Конкретные значения TID несущественны, важно лишь, чтобы все задачи, запущенные в PVM, имели различные TID. При этом копии одного исполняемого файла, запущенные параллельно на N процессорах PVM, создают N задач с разными TID .

Для принятой модели взаимодействия задач в PVM считается, что в пределах одной PVM любая задача может передавать сообщения любой другой задаче, причем, размеры и число таких сообщений в принципе не ограничены. Это предположение существенно упрощает реализацию PVM на конкретных вычислительных комплексах, т.к. при этом контроль переполнения буферных устройств и массивов остается только в ведении частных операционных систем.

Существуют следующие типы межзадачного обмена информацией:

* **блокируемый обмен**, при котором функция "Послать сообщение" возвращает значение (т.е. завершает работу) только после того как получена положительная или отрицательная квитанция от получателя сообщения. Такой алгоритм передачи с ожиданием уведомления о доставке предпочтителен в тех случаях, когда длинное сообщение передается несколькими порциями, а также при обмене командами, последовательность выполнения которых во времени должна быть строго фиксированной.
* **неблокируемый обмен** сообщений уменьшаются простои процессоров, вызванные ожиданием реакции "собеседника". Особенно большой эффект это дает на приемной стороне при неизвестном времени прихода сообщения. Можно организовать работу приемного процессора так, чтобы он в ожидании сообщения выполнял текущую работу, лишь время от времени опрашивая приемный буфер.

Существенным является то обстоятельство, что при передаче последовательности сообщение от одной задачи к другой порядок приема сообщение всегда совпадает с порядком их передачи. Более того, если до обращения к функции "Принять сообщение" в приемный буфер принимающей задачи записано несколько сообщений, то функция "Принять сообщение" возвратит ссылку на первое принятое сообщение.

Память для буферных массивов на передающей и приемной стороне выделяется динамически, следовательно, максимальный объем сообщений ограничивается только объемом доступной памяти. Если одна из задач, запущенных в PVM, не может получить требуемую память для общения с другими задачами, то она выдает пользователю соответствующее сообщение об ошибке ("cannot get memory"), но другие задачи об этом событии не извещаются и могут, например, продолжать посылать ей сообщения.

Кроме обмена сообщениями между двумя задачами в PVM предусмотрены возможность широковещательной передачи сообщений от одной задачи к нескольким другим задачам, а также возможности синхронизации действий в группе задач и совместное выполнение задачами группы некоторых операций над распределенными данными (фрагменты обрабатываемых данных распределены между задачами - членами группы).

## Управление задачами

1) int tid = pvm\_mytid ( void );

возвращает идентификатор задачи tid >= 0.

2) int numt = pvm\_spawn (char \*task, /\*имя исполняемого файла\*/

char \*\*argv,/\*арг-ты командной строки\*/

int flag, /\*опции запуска\*/

char \*where,/\*указывает место запуска\*/

int ntask, /\*число запускаемых копий\*/

int \*tids /\*массив значений tid для

запущенных задач\*/ );

- запускает в PVM ntask копий исполняемого файла с именем "task" с одинаковыми аргументами командной строки в массиве argv и возвращает число запущенных задач numt а также последовательность идентификаторов для запущенных задач. Причем, если numt < ntask, то в последних ntask - numt элементах массива tids записаны отрицательные коды ошибок, объясняющие срыв запуска задачи.

Замечание 1: исполняемый файл для функции pvm\_spawn() должен находиться в строго определенном каталоге. Задавать имя каталога в параметре "task" недопустимо (или, по крайней мере, нежелательно).

Замечание 2: исполняемый файл ищется (и запускается) не только на том компьютере, на котором работает вызвавшая pvm\_spawn() задача, но в зависимости от параметров flag и where, на любом входящем в состав PVM.

Значением параметра flag задается набор опций для запускаемых задач. Каждой опции соответствует целое неотрицательное число - вес опции, и значение flag равно сумме весов выбранных опций.

PvmTaskDefault ( 0 ) - Pvm выбирает по умолчанию, где запустить задачи;

PvmTaskHost ( 1 ) - задачи запускаются на машине, тип которой указан в параметре where;

PvmTaskArch ( 2 ) - задачи запускаются на комплексе, архитектура которого указана в параметре where;

PvmTaskDebug ( 4 ) - задачи стартуют под отладчиком;

PvmTaskTrace ( 8 ) - при выполнении генерируются результаты трассировки;

PvmMppFront ( 16 ) - задачи стартуют на MPP системе;

PvmHostCompl ( 32 ) - дополнение к информации в параметре where.

## Примеры

Пример 1.

numt = pvm\_spawn ("host", 0, PvmTaskHost, "sparky", 2, &tid[0]); - запускаются 2 копии исполняемого файла "host" без параметров в командной строке на вычислительном комплексе типа "sparky"; возвращаемые идентификаторы задач записаны в массиве tid.

Пример 2.

numt = pvm\_spawn ( "node", argv, PvmTaskArch + PvmTaskDebug, "RIOS", 2, &tid[5] );

- запускаются 2 копии исполняемого файла "node" с параметрами командной строки в массиве argv на любом включенном в состав PVM вычислительном комплексе с архитектурой типа "RIOS" с использованием отладчика; возвращаемые идентификаторы задач записаны в tid[5], tid[6].

Пример 3.

numt = pvm\_spawn ( "node", 0, 0, 0, 4, tids );

- запускаются 4 копии исполняемого файла "node" без параметров командной строки с автоматическим выбором используемых вычислительных средств для выполнения задач; возвращаемые идентификаторы задач записаны в массиве tid.

## Управление задачами (окончание)

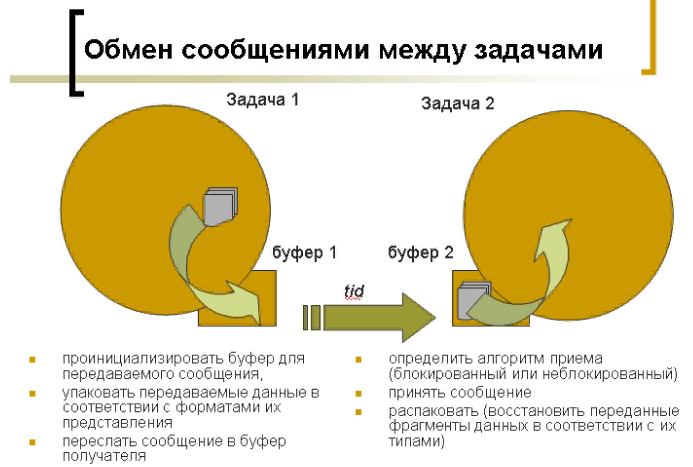
3) int info = pvm\_kill ( int tid );

- завершает выполнение задачи с идентификатором tid; при ошибке возвращает код ошибки info < 0.

Задача не может "убить" себя, поэтому один из возможных сценариев завершения многозадачной работы PVM заключается в том, что одна из задач "убивает" все остальные, после чего вызывает функцию

int info = pvm\_exit ( void );

которая завершает работу PVM, запущенной пользователем, но при этом сама задача продолжает выполняться уже как объект локальной операционной системы и завершает работу, как обычно.



## Многомашинность

1) Программное обеспечение PVM должно быть развернуто на всех машинах

2) Пользователь должен на каждой машине создать свой личный конфигурационный файл PVM

В этом файле указываются:

* имена остальных компьютеров, привлекаемых в состав PVM,
* их производительность, выраженная в неких относительных единицах,
* пароли для входа на эти машины,
* и прочая информация, зачастую не обязательная

Чтобы приложение могло быть запущено на каком-то компьютере, его надо сначала скомпилировать и поместить в каталог $HOME/pvm3/bin/$PVM\_ARCH !

Тогда pvm\_spawn() сможет выбирать наиболее подходящее место выполнения для запускаемой задачи, не ограничиваясь текущей физической машиной.

Стадию переноса можно автоматизировать, воспользовавшись утилитами rdist и rsh.

# Технология DVM

DVM-система предназначена для создания переносимых и эффективных вычислительных приложений на языках C-DVM и Fortran-DVM для параллельных компьютеров с различной архитектурой.

Аббревиатура DVM соответствует двум понятиям: Distributed Virtual Memory и Distributed Virtual Machine. Первое отражает наличие единого адресного пространства. Второе отражает использование виртуальных машин для двухступенчатой схемы отображения данных и вычислений на реальную параллельную машину.

В систему DVM также входят библиотека поддержки LIB-DVM, DVM-отладчик, предсказатель выполнения DVM-программ, анализатор производительности DVM-программ. Система разработана в Институте прикладной математики им. М.В.Келдыша РАН.

DVM – MPP системы, основное понятие – процесс.

## Пример программы на DVM: Решение системы ОДУ

#include <stdio.h>

#include <math.h>

#define DVM(dvmdir)

#define DO(v,l,h,s) for (v=(l); v<(h); v+=s)

#define FOR(v,l) for (v = 0; v < l; v++)

#define N 4

double func ( double x0, double x1, double x2, double x3, int rank){

double w;

switch(rank){

case 0: w = x2; break;

case 1: w = x3; break;

case 2: w = -x0 / pow ((x0\*x0 + x1\*x1),1.5); break

case 3: w = -x1 / pow ((x0\*x0 + x1\*x1),1.5); break

}

return w;

}

DVM (DISTRIBUTE[block]) double y[N];

DVM (DISTRIBUTE[block]) double yy[N];

DVM (DISTRIBUTE[block]) double r1[N];

DVM (DISTRIBUTE[block]) double r2[N];

DVM (DISTRIBUTE[block]) double r3[N];

DVM (DISTRIBUTE[block]) double r4[N];

int i, j;

double tt = 0.0, tmax = 0.1, tau = 0.01;

int main(int argc, char \*\* argv)

{

y[0] = 0.9; y[1] = 0.0; y[2] = 0.0; y[3] = 1.1;

fprintf(DVMSTDERR,”time = %f\n”, tt);

DVM(REMOTE\_ACCESS y[]){

fprintf(DVMSTDERR, “y[0] = %f y[1] = %f y[2] = %f y[3] = %f\n”, y[0],y[1],y[2],y[3]);}

do{

DVM(PARALLEL[i] ON r1[i]; REMOTE\_ACCESS y[])

FOR(i , N){

r1[i] = tau \* func(y[0],y[1],y[2],y[3]);

}

DVM(PARALLEL[i] ON yy[i])

FOR (i , N){

yy[i] = y[i] + 0.5 \* r1[i];

}

DVM(PARALLEL[i] ON r2[i]; REMOTE\_ACCESS y[])

FOR(i , N){

r2[i] = tau \* func(y[0],y[1],y[2],y[3]);

}

DVM(PARALLEL[i] ON yy[i])

FOR (i , N){

yy[i] = y[i] + 0.5 \* r2[i];

}

DVM(PARALLEL[i] ON r3[i]; REMOTE\_ACCESS y[])

FOR(i , N){

r1[i] = tau \* func(y[0],y[1],y[2],y[3]);

}

DVM(PARALLEL[i] ON yy[i])

FOR (i , N){

yy[i] = y[i] + r3[i];

}

DVM(PARALLEL [i] ON y[i])

FOR(i,N){

y[i] += (r1[i] + 2 \* r2[i] + 2 \* r3[i] + r4[i]) / 6.0;

}

tt += tau;

fprintf(DVMSTDERR,”time = %f\n”, tt);

DVM(REMOTE\_ACCESS y[]){

fprintf(DVMSTDERR, “y[0] = %f y[1] = %f y[2] = %f y[3] = %f\n”, y[0],y[1],y[2],y[3]);}

} while ( tt <= tmax);

return 0;

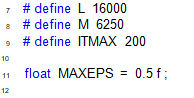
}

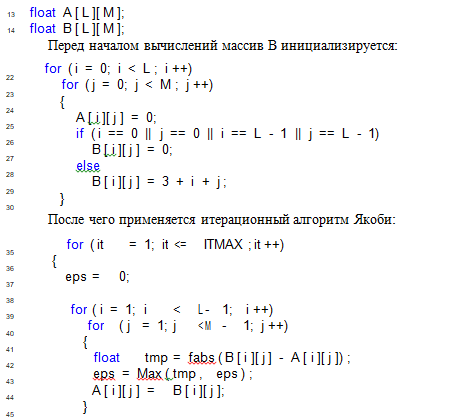
## Распараллеливание алгоритма Якоби на C-DVMH

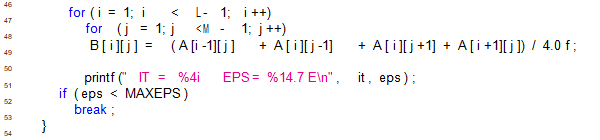
Первое знакомство с DVM-системой лучше всего начать с небольшого примера.

Возьмем маленькую программу, которая совершает какие-то преобразования матрицы. В принципе для нас сейчас не имеет значения смысл этих преобразований, но мы хотим, чтобы они выполнялись быстрее, а результат вычислений остался таким же. Как этого добиться? – распараллелить программу. И, естественно, для этого мы будем использовать технологию DVMH.

Рассмотрим исходную программу. Полные исходные коды можно посмотреть в конце этой статьи. Все вычисления производятся над массивом B, кроме него есть вспомогательный массив A:





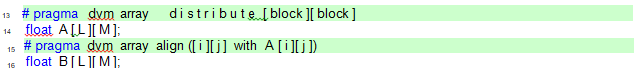


Чтобы получить работающую DVM-программу, необходимо сделать две вещи:

* распределить данные;
* распределить вычисления над этими данными.

Поскольку DVM-программа может выполняться на кластере, то можно себе представить, что наша программа будет состоять из отдельных mpi-процессов, запущенных на разных узлах этого кластера. И наша задача раздать каждому mpi-процессу свою порцию данных и свою часть работы.

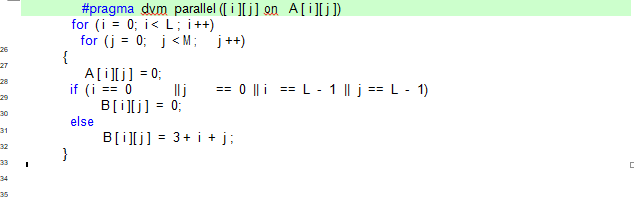
Чтобы осуществить первое (распределить данные) достаточно добавить директивы:



Это означает, что массив A будет равномерно разделен на части по двум измерениям и по-лученные подматрицы будут розданы mpi-процессам. Т.е. каждый mpi-процесс получит свою подматрицу A(i1:i2, j1:j2). Кроме того, массив B будет распределен точно таким же образом и каждая подматрица B(i1:i2, j1:j2) попадет к тому же самому mpi-процессу, что и соответству-ющая подматрица A(i1:i2, j1:j2).

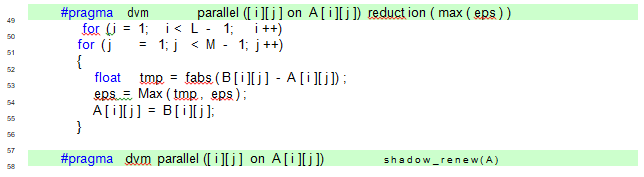
Теперь, чтобы распределить вычисления, нужно найти участки кода, в которых использу-ются распределенные данные. В нашем случае есть три двумерных цикла, которые работают с массивами A и B.

Начнем с цикла инициализации. Этот цикл выполняется только один раз в начале про-граммы и не занимает много времени, но после того, как мы распределили массивы, каждый mpi-процесс должен следить, чтобы у него не происходило обращений к элементам массива, отданным другим mpi-процессам. Поэтому теперь этот цикл будет выполняться гораздо мед-леннее и его неплохо бы распараллелить. Для этого достаточно добавить директиву parallel:



Это означает, что итерацию (i,j) должен выполнять только тот mpi-процесс, которому был отдан элемент A(i,j). Таким образом, mpi-процессы, работая параллельно, будут выполнять предназначенные только им итерации цикла.

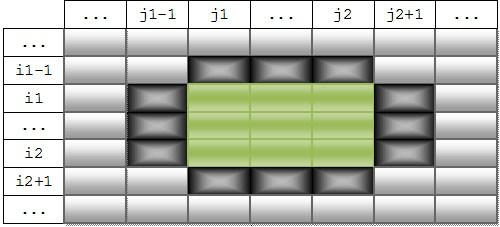
Далее, нужно аналогичным образом распараллелить циклы внутри алгоритма Якоби:





В этих двух циклах пришлось использовать дополнительные указания reduction и shadow\_renew.

В первом цикле встречается переменная eps, которая должна содержать максимальное из вычисленных значений на каждой итерации. Мы знаем, что mpi-процесс выполняет только предназначенные ему итерации параллельного цикла. Следовательно, после первого цикла его переменная eps будет иметь максимальное значение, вычисленное только среди его итераций. Получается, что для вычисления максимума на всех итерациях, нужно после цикла найти мак-симум среди eps всех mpi-процессов. Этим и занимается операция reduction(max(eps)), кото-рая после цикла выполнит необходимые действия, и все mpi-процессы будут иметь одинаковые значения собственных переменных eps, равные максимальному среди вычисленных на всех ите-рациях цикла.

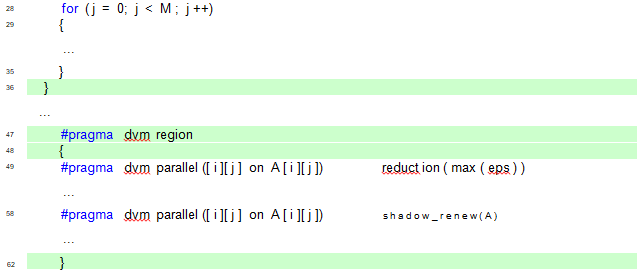
Во втором цикле происходит обращение из итерации (i,j) к элементам массива A(i-1,j), A(i,j-1) и так далее. Т.е. mpi-процесс, который владеет элементом A(i,j) и выполняет итера-цию (i,j), пытается обратиться к элементу A(i-1,j), который может принадлежать соседнему mpi-процессу. Поскольку mpi-процесс не может напрямую обратиться в память другого mpi-процесса, ему приходится предварительно копировать данные к себе. Такое копирование можно обеспечить указанием shadow\_renew. Если mpi-процесс владеет подматрицей A(i1:i2,j1:j2), то shadow\_renew(A) до начала цикла скопирует ему от других mpi-процессов подматрицы A(i1-1,j1:j2), A(i2+1,j1:j2), A(i1:i2,j1-1), A(i1:i2,j2+1).

Область, образуемая этими подматрицами, называется теневыми гранями. Изменить размер теневых граней можно директивой shadow, но здесь этого не требуется, поскольку текущий размер нас устраивает.

На данном этапе у нас уже получилась DVM-программа, которую можно запустить, напри-мер, на кластере и увидеть, что программа ускорилась. Однако, она использует только MPI для параллельного выполнения, так что, если на этом кластере имеются графические ускорители (GPU), то они останутся не задействованы. Чтобы задействовать и их, нужно совсем немного, именно, указать, какие участки программы следует выполнять на GPU и, при необходимости, указать, какие данные нужно скопировать из оперативной памяти в память GPU или обратно.

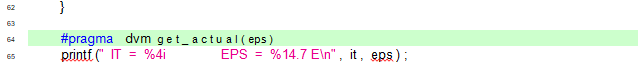
Регионы (участки кода, которые могут быть выполнены на GPU) должны содержать па-раллельные циклы. Если взглянуть на нашу программу, то видно, что параллельные циклы просты и могут быть помещены в регионы. Что мы и делаем:





Расставив регионы, нужно обеспечить копирование данных между оперативной памятью и памятью GPU, чтобы обновленные данные были перемещены туда, где они нужны.

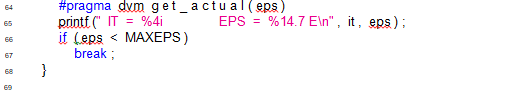
Для цикла инициализации ничего делать не нужно: массивы A и B будут созданы и заполнены на графическом ускорителе. Далее идет регион в алгоритме Якоби, который работает этими массивами, которые уже расположены в памяти GPU, поэтому их копировать тоже не нужно. Однако, в регионе вычисляется значение eps, которое после региона печатается на экран, так что необходимо скопировать eps из региона в оперативную память. Делается это с помощью директивы get\_actual:

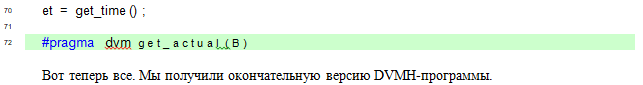


Теперь будет напечатано значение eps, посчитанное на GPU, а не 0, записанный до начала региона. На следующей итерации it, снова выполняется этот же регион. Переменная eps уже встречалась раньше и сохранила свое значение на GPU, так что при чтении будет использоваться оно, а не то, что мы присвоили перед регионом. Чтобы исправить ситуацию необходимо сообщить, что новое значение переменной eps нужно взять из оперативной памяти. Сделать это можно с помощью директивы actual:



* нас получилась работающая реализация алгоритма Якоби, которая может выполняться на кластере с GPU, но вычисленная матрица B по-прежнему располагается в памяти GPU. Если мы хотим ее, например, распечатать, то ее нужно вернуть в оперативную память с помощью директивы get\_actual:





## Запуск DVMH-программы

Попробуем скомпилировать и запустить полученную DVMH-программу. Для этого нам понадобится вычислительная машина, на которой установлена DVM-система

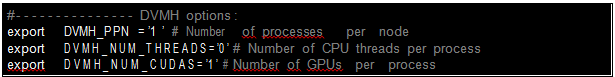
Директорию, где находится наша программа jac2d.cdv, следует скопировать dvm скрипт из директории с установленной DVM-системой.



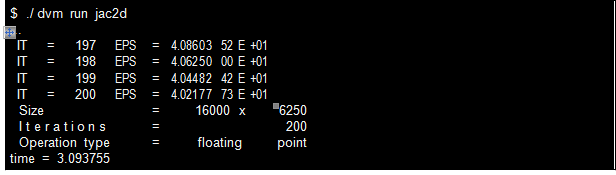
Теперь можно скомпилировать нашу программу.



В текущей директории появился исполняемый файл jac2d, который можно запустить с помощью скрипта dvm, но предварительно надо настроить параметры запуска, отредактировав dvm скрипт.



Задав такие параметры, мы сообщаем, что хотим выполнять регионы только на одном гра-фическом ускорителе. Всё, можем запускать:



# Технология Linda

Разработана в 1980 гг. в Йельском университете (США).

Идея построения системы Linda исключительно проста, а потому красива и очень привлекательна. Параллельная программа есть множество параллельных процессов, и каждый процесс работает согласно обычной последовательной программе. Все процессы имеют доступ к общей памяти, единицей хранения в которой является кортеж. Отсюда происходит и специальное название для общей памяти - пространство кортежей. Каждый кортеж это упорядоченная последовательность значений. Например,

( "Hello", 42, 3.14 ),

( "P", 5, FALSE, 97, 1024, 2),

( "worker", 5 ) .

Первый элемент кортежа всегда является символьной строкой и выступает в роли имени кортежа. Так первый кортеж предыдущего примера состоит из имени ("Hello"), элемента целого типа (42) и вещественного числа (3.14). Во втором кортеже кроме имени "P" есть элемент целого типа (5), элемент логического типа (FALSE) и три целых числа. Последний кортеж состоит из двух элементов: имени ("worker") и целого числа (5). Количество элементов в кортеже может быть любым.

1. Функция OUT помещает кортеж в пространство кортежей.
2. Функция IN ищет подходящий кортеж в пространстве кортежей, присваивает значения его элементов элементам своего параметра-кортежа и удаляет найденный кортеж из пространства кортежей.
3. Функция READ отличается от функции in лишь тем, что выбранный кортеж не удаляется из пространства кортежей. Все остальное точно так же, как и у функции in. Этой функцией удобно пользоваться в том случае, когда значения переменных менять не нужно, но к ним необходим параллельный доступ из нескольких процессов.
4. Функция EVAL похожа на функцию out. Разница заключается лишь в том, что дополнительным элементом кортежа у eval является функция пользователя. Для вычисления значения этой функции система Linda порождает параллельный процесс, на основе работы которого она формирует кортеж и помещает его в пространство кортежей.

## Синхронизация потоков

Не имея в системе Linda никаких явных средств для синхронизации процессов, совсем не сложно их смоделировать самому. Предположим, что в некоторой точке нужно выполнить барьерную синхронизацию N процессов. Какой-то один процесс, например, стартовый, заранее помещает в пространство кортеж ("ForBarrier", N). Подходя к точке синхронизации, каждый процесс выполняет следующий фрагмент, который и будет выполнять функции барьера:

in( "ForBarrier", formal Bar);

Bar = Bar - 1;

if( Bar != 0 ) {

out( "ForBarrier", Bar);

read( "Barrier" );

} else

out( "Barrier" );

Если кортеж с именем "ForBarrier" есть в пространстве, то процесс его изымает, в противном случае блокируется до его появления. Анализируя второй элемент данного кортежа, процесс выполняет одно из двух действий. Если есть процессы, которые еще не дошли до данной точки, то он возвращает кортеж в пространство с уменьшенным на единицу вторым элементом и встает на ожидание кортежа "Barrier". В противном случае он сам помещает кортеж "Barrier" в пространство, который для всех является сигналом к продолжению работы.

# Технология mpC

Язык программирования mpC - это расширение языка Си, разработанное специально для программирования параллельных вычислений на обычных сетях разнородных компьютеров. Основной целью параллельных вычислений является ускорение решения задачи. Именно это отличает параллельные вычисления от распределённых, для которых основной целью является обеспечить совместную работу программных компонент, изначально размещённых на различных компьютерах. В случае параллельных вычислений разбиение программы на компоненты, размещаемые на разных компьютерах, является лишь средством для ускорения работы программы, а не врождённым свойством этой программы. Поэтому, основное внимание в языке mpC уделяется средствам, позволяющим максимально облегчить разработку как можно более эффективных программ для решения задач на обыкновенных сетях компьютеров.

Параллельная программа - множество параллельных процессов, взаимодействующих посредством передачи сообщений.

Программист на mpC не может управлять тем, сколько процессов составляют программу и на каких компьютерах эти процессы выполняются. Это делается внешними по отношению к языку средствами. Исходный код на mpC управляет лишь тем, какие именно вычисления выполняются каждым из процессов, составляющих программу.

# Технология OpenCL

OpenCL (Open Computing Language ) это спецификация, описывающая технологию параллельного программирования, которая в первую очередь ориентирована на GPGPU. Изначально она была разработана компанией Apple, в последствии для развития спецификаций OpenCL был образована группа разработчиков Khronos Compute , в неё вошли Apple, nVidia, AMD, IBM, Intel, ARM, Motorola и др. Первая версия стандарта была опубликована в конце 2008 года.

В отличии от nVidia CUDA, AMD Stream и т.п., в OpenCL изначально закладывалась мультиплатформенность, т.е. OpenCL программа должна без изменений в коде работать на GPU разных типов (разных производителей). Такая программа без изменений должна работать даже на CPU без GPU, хотя в этом случае она может выполняться существенно медленнее чем на GPU.

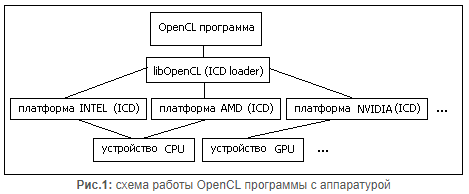
## Схема работы с аппаратурой

Итак - хотим мультиплатформенность и желательно без существенных потерь в производительности. Достигается этот результат следующим образом

OpenCL-программа работает с т.н. **платформами (platform)**. Платформа это программный пакет, который поставляется соответствующим разработчиком аппаратных средств. Например "AMD Accelerated Parallel Processing" или "Intel OpenCL". При этом несколько платформ могут работать одновременно на одной машине.

Каждая платформа включает в себя **ICD (Installable Client Driver)**-- программный интерфейс OpenCL для работы с устройствами, которые эта платформа поддерживает.

В среде Linux список ссылок на ICD, присутствующих в системе, обычно хранится в каталоге */etc/OpenCL/vendors/*, а библиотека *libOpenCL.so* выполняет роль **диспетчера (ICD loader)**, т.е. она направляет вызовы OpenCL функций на устройства, через соответствующие ICD.



## Структура программы

В OpenCL (аналогично CUDA), программа разделяется на две части: первая часть - управляющая, вторая - вычислительная. В роли управляющего устройства (**host**) выступает центральный процессор (CPU), вычислительное устройство (**device**) выбираем из списка платформ и их устройств. Обычно используется GPU, но не обязательно, это может быть и тот же CPU.

Код, который должен выполняться на device, оформляется специальным способом. Поскольку device могут быть от разных производителей и разных типов, то скомпилировать один бинарник для всех не получится. Решается эта проблема следующим образом.

Текст **ядра (kernel)**, т.е. части программы выполняемой на device, включается в основную часть программы (выполняемой на host) в "чистом" виде т.е. в виде текстовой строки. Этот исходник компилируется средствами OpenCL непосредственно в процессе работы программы (runtime) для выбранного в данный момент вычислительного устройства, это происходит каждый раз при запуске OpenCL-программы.

Так же есть возможность, скомпилированный таким образом, код ядра сохранить и при инициализации вычислительной программы загружать готовый бинарник. Но в этом случае мы теряем универсальность, этот код будет работать только на одном (данном) типе устройств.

Далее рассмотрим общую структуру OpenCL-программы. Она выглядит более сложной чем аналогичная программой CUDA, поскольку необходимо выполнять дополнительные действия по конфигурированию среды выполнения и подготовке кода для device. Эта среда в терминах OpenCL именуется **контекстом (context)**, она включает в себя платформу (platform), вычислительные устройства (device) и буферы памяти для них. Для device в рамках context создаётся **очередь команд на исполнение (command queue)**. Операции с device, такие как чтение/запись данных и запуск ядра, помещаются в эту очередь и последовательно исполняются. Таким образом общая схема OpenCL-программы выглядит так:

1. получить информацию о платформах и устройствах  
   clGetPlatformIDs(),clGetPlatformInfo(), clGetDeviceIDs(),clGetDeviceInfo()
2. выбрать устройства и создать для них контекст  
   clCreateContext()
3. создать ядро из текста программы  
   clCreateProgramWithSource(), clBuildProgram(), clCreateKernel()
4. выделить память для данных на устройствах  
   clCreateBuffer()
5. создать очередь комманд для устройтва  
   clCreateCommandQueue(), clCreateCommandQueueWithProperties()
6. скопировать данные с host на device  
   clEnqueueWriteBuffer()
7. назначить параметры выполнения ядра  
   clSetKernelArg()
8. запуск ядра  
   clEnqueueNDRangeKernel()
9. скопировать результат с device на host  
   clEnqueueReadBuffer()
10. обработка результата
11. завершение работы, освобождение ресурсов   
    clReleaseMemObject(), clReleaseKernel(), clReleaseProgram(), clReleaseCommandQueue(), clReleaseContext()

## Пример программы на OpenCL

В качестве первого примера рассмотрим простую вычислительную задачу: С := d \* A + B, где d - константа, А,В и С векторы заданного размера.

\_\_kernel

void kernel1(const float alpha, \_\_global float \*A, \_\_global float \*B, \_\_global float \*C) {

int idx = get\_global\_id(0);

C[idx] = alpha\* A[idx] + B[idx];

}

Программа запускает много (по количеству элементов векторов) параллельных процессов (тредов) на device, каждый тред получает свой номер (get\_global\_id), исходя из него считывает свою часть данных из глобальной (\_\_global) памяти device, выполняет вычисления и записывает свою часть результата обратно в память.

## Группа процессов и их конфигурация

Каждое вычислительное устройство обладает своими ограничениями по количеству одновременно выполняемых тредов, и хотя их общее количество в программе может быть велико, выполняться они будут частями или группами, максимальный размер группы зависит от конкретного устройства.

Иногда треды удобно формировать в виде решетки. Рассмотрим пример задачи умножения матриц.

// конвертер индексов матрицы в линейный адрес

#define IDX2LIN(i,j,l) (i+j\*l)

\_\_kernel void myGEMM1(const int M, const int N, const int K,

const \_\_global float\* A, const \_\_global float\* B, \_\_global float\* C)

{ // номер треда (2D решетка)

const int r = get\_global\_id(0); // строка 0..M

const int c = get\_global\_id(1); // столбец 0..N

// вычисляем элемент [r,c] результирующей матрицы C

float acc = 0.0f;

for (int i=0; i<K; i++) {

acc += A[ IDX2LIN(r,i,M) ] \* B[ IDX2LIN(i,c,K) ];

}

C[ IDX2LIN(r,c,M) ] = acc; // сохраняем результат

}

Имеем на входе матрицы A[M\*K], B[K\*N] и соответственно буфер для результата C[M\*N]. Программа создаёт M\*N тредов в виде решетки MxN, каждый тред [r,c] отрабатывает одну ячейку в матрице результата.

## Модели памяти и синхронизация процессов

Модель памяти OpenCL имеет несколько типов. Рассмотрим их подробней : global, local, private.

* Память global - основная память уcтройства, самая большая по размеру (512MB для FX1700) и самая медленная, она является общей для всех тредов.
* Память local - общая память для одной группы тредов (shared в терминах CUDA), этот тип быстрее global но существенно меньше по размеру (16KB для FX1700)
* Память private - память треда, быстрая но маленькая (8KB для FX1700)

Вернёмся к предыдущему примеру с умножением матриц. В процессе работы ядро выполняет много повторных чтений из памяти global. Попробуем сократить количество чтений с помощью организации быстрого кэша.

// размер кэша тредов

#define TS 16

// конвертер индексов матрицы в линейный адрес

#define IDX2LIN(i,j,l) (i+j\*l)

\_\_kernel void myGEMM2(const int M, const int N, const int K,

const \_\_global float\* A, const \_\_global float\* B, \_\_global float\* C)

{

// 2D номер треда в группе

const int r = get\_local\_id(0);

const int c = get\_local\_id(1);

// номер ячейки в матрице результата

const int gr = get\_group\_id(0)\*TS + r; // 0..M

const int gc = get\_group\_id(1)\*TS + c; // 0..N

// общий кэш для тредов группы

\_\_local float Asub[TS][TS];

\_\_local float Bsub[TS][TS];

float acc = 0.0f; // результат работы треда

for (int t=0; t<K/TS; t++) { // цикл по всем блокам матриц

const int tr = t\*TS + r;

const int tc = t\*TS + c;

// загружаем блоки в кэш

Asub[c][r] = A[ IDX2LIN(gr,tc,M) ];

Bsub[c][r] = B[ IDX2LIN(tr,gc,K) ];

barrier(CLK\_LOCAL\_MEM\_FENCE); // ждём пока треды группы заполнят общий кэш

// вычисляем

for (int k=0; k<TS; k++) { acc += Asub[k][r] \* Bsub[c][k]; }

barrier(CLK\_LOCAL\_MEM\_FENCE); // ждём пока треды группы завершат вычисления

}

C[ IDX2LIN(gr,gc,M) ] = acc; // сохраняем результат }

Имеем на входе матрицы A[M\*K], B[K\*N] и соответственно результат C[M\*N]. Программа создаёт M\*N тредов в виде решетки MxN, группами размера TSxTS, каждый тред отрабатывает одну ячейку в матрице результата, группа тредов кэширует блоки исходных матриц и далее выполняет операции с этим кэшем.

Для того, что бы кэш корректно был заполнен необходима **синхронизация группы тредов (barrier)**, достигая barrier программа ждет, пока все треды группы соберутся в этой точке и только после этого продолжает выполнение.

# Технология OpenACC

В ноябре 2011 года был анонсирован стандарт OpenACC совместное детище суперкомпьютерных гигантов CRAY, CAPS и PGI и лидера рынка графических процессоров NVIDIA. Сам стандарт призван значительно упростить работу программиста и создать высокоуровневую прослойку над уже известными CUDA и OpenCL.

Стоит отметить, что до недавнего времени стандарт не поддерживался в полной мере ни одним компилятором, но даже то, что уже есть, впечатляет своей простотой и результативностью. Теперь написание программы, выполняемой параллельно на тысячах ядер современных GPU не требует почти никаких усилий и практически полностью перекладывается на компилятор. Все что нужно сделать расставить директивы по коду на манер OpenMP.

Как и его прародители (PGI accelerator и CAPS HMPP) OpenACC поддерживает языки С и Fortran. Итак, все директивы в С-версии стандарта начинаются как обычно с #pragma, далее ставится спецификатор acc и одна из основных директив, дополненная одним, или несколькими условиями. Чаще всего используются 3 директивы: parallel, kernels и data.

## Директивы

Рассмотрим краткое описание некоторых директив и условий к ним:

* Директива parallel указывает на необходимость распараллеливания. Компилятор, проводя анализ кода, определяет необходимость исполнения различных его частей на GPU, или на хосте.
* Директива kernels аналог parallel, указывает на то, что для каждого нового цикла необходимо создать отдельную \_\_device\_\_ функцию.
* Директива loop предшествует оператору цикла и используется для спецификации его свойств. Современные компиляторы не требуют её явного указания.

Несмотря на всю мощь компилятора, иногда нужно подсказывать, какие данные необходимо передать с хоста на устройство и обратно, а поскольку зачастую копирование выполняется дольше расчетов, нужно заранее продумать, где и как оптимизировать доступ к данным. Все условия передачи данных требуют входные данные, выглядящие следующим образом: a[start:length], где a массив, или указатель на него, start номер стартового элемента для копирования, а length длина региона данных, копируемого на GPU, или с него; start и length указываются в элементах массива (для Fortran есть существенное отличие вместо length указывается end конечный элемент). Эти условия можно использовать только с директивами kernels, parallel и data region. Ниже представлены те из них, которые используются наиболее часто:

* copy говорит компилятору скопировать данные на устройство перед выполнением ядра и назад после его завершения.
* copyin указывает, что данные на GPU используются только для чтения, и нет необходимости копировать их обратно на хост.
* copyout данные появятся только в результате выполнения ядра на GPU и никак не зависят от предыдущих значений по этому адресу, их нужно скопировать на хост после выполнения кернела.
* create выделяет в памяти устройства место для данных, не требующих какого-либо копирования, например массив для хранения промежуточных результатов.
* present - подсказывает компилятору, что эти данные уже были переданы на устройство ранее. Вызывает ошибку, если данных на GPU нет.

## Пример программы на OpenACC

