Дополнительное распараллеливание MPI-программ с помощью DVM-системы

В.А. Бахтин[1,2], Д.А. Захаров[1], В.А. Крюков[1,2], А.С. Колганов[1], Н.В. Поддерюгина[1], М.Н. Притула[1], А.А. Смирнов[1]

1 ИПМ им. М.В. Келдыша РАН
2 МГУ им. М.В. Ломоносова

Аннотация. DVM-система предназначена для разработки параллельных программ научно-технических расчетов на языках C-DVMH и Fortran-DVMH. Эти языки используют единую модель параллельного программирования (DVMH-модель) и являются расширением стандартных языков Си и Фортран спецификациями параллелизма, оформленными в виде директив компилятору. DVMH-модель позволяет создавать эффективные параллельные программы для гибридных вычислительных кластеров. В статье представлены новые возможности DVM-системы, которые позволяют отображать существующие MPI-программы на кластеры, в узлах которых в качестве вычислительных устройств наряду с универсальными многоядерными процессорами могут использоваться графические ускорители.

Ключевые слова: автоматизация разработки параллельных программ, DVM-система, ускоритель, ГПУ, Фортран, Си, MPI, OpenMP, OpenACC, DVMH

Additional parallelization of MPI programs using DVM-system

V.A. Bakhtin[1,2], D.A. Zakharov[1], V.A. Krukov[1,2], A.S. Kolganov[1], M.N. Pritula[1], N.V. Podderyugina[1], A.A. Smirnov[1]

1 Keldysh Institute of Applied Mathematics
2 Lomonosov Moscow State University

Abstract. DVM-system is designed for the development of parallel programs of scientific and technical calculations in C-DVMH and Fortran-DVMH languages. These languages use a single parallel programming model (DVMH model) and are extensions of the standard C and Fortran languages with parallelism specifications, written in the form of directives to the compiler. The DVMH model makes it possible to create efficient parallel programs for heterogeneous computing clusters. The article presents new features of DVM-system for additional parallelization of MPI programs for clusters where nodes can use graphics accelerators as computing devices along with universal multi-core processors.

Keywords: automation of development of parallel programs, DVM-system, accelerator, GPU, Fortran, C, MPI, OpenMP, OpenACC, DVMH

1. Введение

В настоящее время при разработке программ для высокопроизводительных вычислений на современных кластерах широко используются следующие модели программирования – MPI (для отображения программы по узлам кластера), POSIX Threads (для отображения программы по ядрам процессора), CUDA и OpenCL (для отображения программы по ядрам ускорителя). Все эти модели программирования являются низкоуровневыми. Для отображения программы на все уровни параллелизма программисту приходится использовать комбинацию из перечисленных моделей. Например, MPI+POSIX Threads+CUDA. Технически объединять низкоуровневые модели программирования, реализованные через библиотеки, проще, чем высокоуровневые модели, реализуемые посредством языков и соответствующих компиляторов. Но программировать, отлаживать, сопровождать, переносить на другие ЭВМ такие программы гораздо сложнее. Например, при переходе с графических процессоров фирмы NVIDIA на графические процессоры фирмы AMD придется заменить CUDA на OpenCL. Поэтому важно использовать высокоуровневые модели и языки программирования.

Среди высокоуровневых моделей программирования особое место занимают модели, реализуемые посредством добавления в программы на стандартных последовательных языках спецификаций, управляющих отображением этих программ на параллельные машины. Эти спецификации, оформляемые в виде комментариев в Фортран-программах или директив компилятору (прагм) в программах на языках Си и Си++, не видны для обычных компиляторов, что значительно упрощает внедрение новых моделей параллельного программирования. Примером такой модели для многоядерных процессоров и SMP-систем является модель OpenMP. В последних версиях стандарта OpenMP [1] предложено расширение модели для использования ускорителей. Это расширение позволяет описать параллелизм внутри одного узла кластера: задействовать ядра центрального процессора и ядра ускорителей. Аналогичный подход был реализован совместными усилиями компаний Cray, NVIDIA и AMD. Разработан стандарт OpenACC [2], который описывает набор директив компилятора, предназначенных для упрощения создания гетерогенных параллельных программ, задействующих как центральный, так и графический процессор. Использование высокоуровневых спецификация должно позволить программисту абстрагироваться от особенностей графического процессора, вопросов передачи данных и т.п. С появлением новых высокоуровневых моделей (OpenMP версии 4.+ и OpenACC) процесс разработки параллельных программ может существенно упроститься. Теперь

вместо 3-х моделей программирования можно использовать только две. Например, MPI+OpenMP или MPI+OpenACC.

В последнее время появилось множество компиляторов, поддерживающих отображение OpenMP и OpenACC-программ на графические ускорители. Например, GNU GCC, Cray CCE, IBM XL, PGI, Clang/Flang и многие другие [3]. Основные сложности, которые возникают у прикладных программистов при использовании данных компиляторов следующие:

- для получения эффективной параллельной программы, как правило, требуется большой опыт использования того или иного OpenMP/OpenACC-компилятора. Существуют спецификации параллелизма, которые предоставляют компилятору некоторую свободу при генерации параллельной версии программы. Например, спецификация kernels в OpenACC определяет в фрагмент в программе, который может выполняться на ускорителе. Такой фрагмент может выполняться последовательно одной нитью, а может – параллельно. В этом случае компилятор создает некоторое множество нитей, которые объединяются в блоки, а количество блоков и нитей компилятор и система поддержки выбирает на свое усмотрение. Опыт использования компилятора позволяет лучше разобраться в логике его работы, научиться анализировать оптимизационные отчеты компилятора, понять, как управлять работой компилятора и «заставить» генерировать более эффективный код;
- не все возможности, описанные в стандартах, реализованы в компиляторах в настоящее время. Например, на сегодняшний день нет ни одного компилятора с полной поддержкой OpenMP версии 5.0. Многие новые возможности стандарта OpenMP версии 4.5 (опубликован в ноябре 2015 года), которые необходимы для работы на ускорителях еще не реализованы ("partially supported", "limited support", "offloading to GPU devices is available with some limitations") [3]. При переходе с одной версии компилятора на другую версию приходится преобразовывать код, чтобы добиться его корректной работы. Некоторые советы, что следует, а что не следует делать в программах при использовании различных компиляторов, можно найти, например, в [4];
- эффективное отображение программы на ускорители требует в большинстве случаев существенного изменения кода программы. Причем такие преобразования зависят от целевой архитектуры, для которой разрабатывается программа. Например, оптимизации, которые будут хорошо работать для сопроцессора Intel Xeon PHI, могут оказаться неприменимыми для графического ускорителя и наоборот. Некоторые исследователи говорят о переносимости OpenMP/OpenACC программ на различные вычислительные устройства, но для обеспечения «эффективной» переносимости параллельных программ

требуется сопровождать несколько версий программы: для центрального процессора, сопроцессора и графического ускорителя;

- в настоящее время не существует удобных, специализированных инструментов для отладки и анализа эффективности OpenMP и OpenACC-программ для ускорителей. Как правило, программистам приходится использовать штатные средства. Например, использовать Nvidia Visual Profiler для анализа производительности программы на графическом ускорителе и разбираться в логике работы компилятора (какое вычислительное CUDA-ядро соответствует тому или иному циклу в программе, какие спецификации параллелизма привели к той или иной операции копирования данных из памяти центрального процессора в память ускорителя и.т.п). Важный вопрос – как понять, какие спецификации параллелизма требуется исправить в исходной программе при обнаружении некорректного выполнения полученной после компиляции CUDA или OpenCL-версии программы?

Таким образом, разработка параллельных программ для кластеров с ускорителями в моделях MPI+OpenMP и MPI+OpenACC на сегодняшний день затруднена.

В качестве альтернативы моделям OpenMP и OpenACC для разработки параллельных программ можно рекомендовать использовать модель DVMH (Distributed Virtual Memory for Heterogeneous Systems). Эта высокоуровневая модель была предложена в 2011 году в ИПМ им. М.В. Келдыша РАН. Модель [5-6] позволяет создавать программы для гетерогенных вычислительных кластеров с различными ускорителями. Прикладные программы, реализованные в данной модели (далее DVMH-программы), не требуют изменений при их переносе с одной вычислительной системы на другую. Задача эффективного отображения программы на все вычислительные устройства суперкомпьютера решается DVMH-компиляторами и системой поддержки выполнения DVMH-программ. Реализованы специализированные отладчики и анализаторы выполнения DVMH-программ, которые выдают диагностики об ошибках и характеристики эффективности в терминах программы пользователя.

В статье представлены новые возможности DVM-системы [7], которые позволяют отображать существующие MPI-программы на кластеры с ускорителями.

В разделе 2 рассматриваются спецификации параллелизма, которые необходимы для дополнительного распараллеливания MPI-программ. В разделе 3 описан процесс запуска MPI/DVMH-программы на счет и приведены основные переменные окружения, которые управляют выполнением параллельной программы. В разделах 4 и 5 показаны возможности инструментальных средств, реализованных в DVM-системе, для анализа эффективности и функциональной отладки MPI/DVMH-программ. В разделе 6 представлены результаты дополнительного распараллеливания программы Himeno.

2. DVM-спецификации параллелизма для дополнительного распараллеливания существующих MPI-программ

В настоящее время, когда параллельные машины уже не одно десятилетие эксплуатируются для проведения расчетов, имеется множество программ, которые уже распараллелены на кластер, однако не имеют распараллеливания по ядрам центрального процессора, а также не используют графические ускорители.

Традиционно в DVM-подходе весь процесс программирования или распараллеливания имеющихся последовательных программ начинается с распределения массивов, а затем отображения на них параллельных вычислений. Это означает, что для использования средств DVM-системы распараллеленные, например, на MPI, программы приходится превращать обратно в последовательные и заменять распределенные вручную данные и вычисления на описанные на DVM-языке распределенные массивы и параллельные циклы.

Однако, во-первых, автору не всегда хочется отказываться от своей параллельной программы, а во-вторых, не всегда удается перевести исходную схему распределения данных и вычислений на DVM-язык. В частности, перевод некоторых задач на нерегулярных сетках в модель DVMH может потребовать применения нетривиальных решений и трюков и не всегда возможно.

Одним из способов избавиться от обеих проблем является реализация нового режима работы DVM-системы, в котором она не вовлечена в межпроцессорное взаимодействие, а работает локально в каждом процессе.

Данный режим включается заданием специально созданной MPI-библиотеки при сборке DVM-системы. Эта библиотека не производит никаких коммуникаций и не конфликтует с реальными MPI-реализациями. В результате для системы поддержки выполнения DVMH-программ создается иллюзия запуска программы на 1 процессоре.

Кроме такого режима, в языках C-DVMH и Fortran-DVMH введено понятие нераспределенного параллельного цикла, для которого нет необходимости задавать отображение на распределенный массив. Такой цикл по определению выполняется всеми процессорами текущей многопроцессорной системы, но т.к. DVM-система в описанном новом режиме считает многопроцессорной системой ровно один процесс, то такая конструкция не приводит к размножению вычислений, а только лишь позволяет использовать параллелизм внутри одного процесса – использовать ядра центрального процессора или графического ускорителя. Как следствие, появляется возможность не задавать ни одного распределенного в терминах модели DVMH массива и в то же время пользоваться возможностями DVM-системы:

- использовать параллелизм на общей памяти (задействовать ядра центрального процессора) с использованием нитей (OpenMP или POSIX Threads);

- задействовать графические ускорители: не только «наивное» портирование параллельного цикла на ускоритель, но и выполнение автоматической реорганизации данных, упрощенное управление перемещениями данных;
- подбирать оптимизационные параметры;
- использовать удобные средства для отладки и анализа производительности параллельных программ.

Рассмотрим основные спецификации параллелизма, которые могут быть использованы при разработке MPI/DVMH-программы.

2.1 Вычислительный регион

Вычислительный регион выделяет часть программы (с одним входом и одним выходом) для возможного выполнения на многоядерном процессоре/сопроцессоре или графическом ускорителе:

region-directive ::= **region** [in-out-local-clause]
in-out-local-clause ::= **in** (array-range-list)
| **out** (array-range-list)
| **local** (array-range-list)
| **inout** (array-range-list)
| **inlocal** (array-range-list)
array-range ::= var-name[subscript-range]
subscript-range ::= [int-expr [: int-expr]]

Спецификации in-out-local-clause предназначены для указания направления использования данных в регионе: **in** – входные данные; **out** – выходные данные; **local** – локальные данные: значения указанных переменных в регионе изменяются, но эти изменения нигде далее не будут использоваться; **inout** – сокращенная запись одновременно двух спецификаций **in** и **out**; **inlocal**– сокращенная запись одновременно двух спецификаций **in** и **local**. В спецификациях in-out-local-clause не обязательно указывать все используемые в регионе переменные. Для используемых в регионе, но не указанных в спецификациях переменных, действуют следующие правила по умолчанию:

- все используемые массивы считаются используемыми полностью (подмассивы не выделяются);
- всякая переменная, которая используется только на чтение, получает атрибут **in**;
- всякая переменная, которая используется на запись, получает атрибут **inout**;
- всякая переменная, направление использования которой не поддаётся определению, получает атрибут **inout**;
- атрибуты **local** и **out** не проставляются.

В спецификации in-out-local-clause для каждого измерения массива можно указать секцию (subscript-range), задав диапазон индексов. Допускаются составные указания, например, **out**(s[1:5]), **out**(s[7:10]) или **in**(s[1:5]), **out**(s[6:10]) и пересекающиеся указания, например, **out**(s[1:6]), **out**(s[3:10]) или даже **out**(s[1:6]), **out**(s[3:5]). Не допускаются конфликтующие указания, такие как **out**(v), **local**(v).

2.2 Параллельный цикл

Вычислительный регион, как правило, состоит из одного или нескольких параллельных тесно-гнездовых циклов:

parallel-directive ::= **parallel** parallel-map [parallel-clause-list]
parallel-map ::= (int-constant)
| (do-variable-list)
parallel-clause ::= private-clause
| reduction-clause
| across-clause
| tie-clause

Спецификация parallel-map задает количество циклов гнезда, которое ассоциируется с данной директивой.

Спецификация **private** объявляет переменную приватной. Переменная называется приватной, если ее использование локализовано в пределах одного витка цикла:

private-clause ::= **private** (var-name-list)

Очень часто в программе встречаются циклы, в которых выполняются редукционные операции – в некоторой переменной суммируются элементы массива или вычисляется максимальное (минимальное) значение. Витки таких циклов можно распределять и выполнять параллельно, если указать спецификацию **reduction:**

reduction-clause ::= **reduction** (red-spec-list)
red-spec ::= red-func (red-variable)
| red-loc-func (red-variable , loc-variable [, size])
red-variable ::= array-name
| scalar-variable-name
loc-variable ::= array-name
| scalar-variable-name
size ::= int-constant
red-func ::= **sum**
| **product**
| **max**

| **min**
| **and**
| **or**
| **xor**

red-loc-func ::= **maxloc**
| **minloc**

Рассмотрим следующий цикл:

```
for (i = 1; i < N-1; i++)
        for (j = 1; j < N-1; j++)
                A[i][j] =(A[i][j-1]+A[i][j+1]+A[i-1][j]+A[i+1][j])/4.;
```

Между витками цикла с индексами i1 и i2 (i1<i2) существует зависимость по данным (информационная связь) массива A, если оба эти витка осуществляют обращение к одному элементу массива по схеме запись-чтение или чтение-запись. Если виток i1 записывает значение, а виток i2 читает это значение, то между этими витками существует потоковая зависимость. Если виток i1 читает «старое» значение, а виток i2 записывает «новое» значение, то между этими витками существует обратная зависимость. В обоих случаях виток i2 может выполняться только после витка i1. Значение i2-i1 называется диапазоном или длиной зависимости. Если для любого витка i существует зависимый виток i+d (где d - константа), тогда зависимость называется регулярной или зависимостью с постоянной длиной. Цикл, в котором существуют регулярные зависимости, можно распределять с помощью директивы **parallel**, указывая спецификацию **across**:

across-clause ::= **across** (across-spec-list)
across-spec ::= var-name [subscript-range]
subscript-range ::= [flow-dep-length [: anti-dep-length]]
flow-dep-length ::= int-expr
anti-dep-length ::= int-expr

В спецификации **across** перечисляются все массивы, по которым существует регулярная зависимость по данным. Для каждого измерения массива указывается длина прямой зависимости (flow-dep-length) и длина обратной зависимости (anti-dep-length). Нулевое значение длины зависимости означает отсутствие зависимости по данным.

Для задания соответствия измерений цикла измерениям массивов используется спецификация **tie**:

tie-clause ::= **tie**(tie-array-list)
tie-array ::= var-name [tie-expr]

tie-expr::= do-variable
| –do-variable
| *

В модели DVMH распределение данных выполняется при помощи выравнивания (директивы **align**), которое для каждого элемента массива A ставит в соответствие элемент или секцию массива B. Если на данный процессор распределен элемент B, то на этот же процессор будет распределен элемент массива A, поставленный в соответствие выравниванием. Распределение вычислений в модели DVMH выполняется с учетом распределения данных. Правило отображения параллельного цикла задается при помощи спецификации **on**, которая позволяет связать измерения цикла с измерениями распределенных массивов. Данная информация позволяет компилятору и системе поддержки выполнить ряд оптимизаций, которые могут существенно повысить эффективность выполнения параллельной программы. Например, зная о том, как связаны массивы между собой, и зная правила отображения цикла, система поддержки выполнения DVMH-программ может определить оптимальное для данного цикла представление массивов в памяти вычислительного устройства и выполнить динамическую трансформацию массивов до и после выполнения этого цикла. В результате, например, на графическом ускорителе все обращения к глобальной памяти, выполняемые CUDA-нитями одного варпа, будут объединены, соседние нити блока будут обращаться к соседним ячейкам глобальной памяти и цикл может выполняться до 10 раз быстрее [8].

В MPI/DVMH-программе спецификации **align** и **on** не используются, т.к. распределение данных и распределение вычислений по узлам кластера реализует сам программист, используя средства MPI. Новая оптимизирующая спецификация **tie** позволяет задать соответствие между измерениями цикла и измерениями массивов и переупорядочить их. При помощи данной спецификации программист передает системе поддержки информацию – с каким измерением цикла есть связь у данного измерения массива (и направление этой связи – прямое или обратное), а если связи нет, то указывается “*”.

Для параллельного выполнения циклов с регулярными зависимостями по данным на графических ускорителях в DVM-системе реализован метод гиперплоскостей. Все элементы, лежащие на одной гиперплоскости, могут быть вычислены независимо друг от друга. При таком порядке выполнения витков цикла снова возникает проблема эффективного доступа к глобальной памяти в силу того, что параллельно обрабатываются несоседние элементы массивов. Для эффективного выполнения таких циклов в системе поддержки реализована так называемая диагональная трансформация, в результате которой соседние элементы массивов на диагоналях (в плоскости необходимых двух измерений) располагаются в соседних ячейках памяти, что позволяет применять технику

выполнения цикла с зависимостями по гиперплоскостям без значительной потери производительности на операциях доступа к глобальной памяти графического ускорителя. Обязательным условием эффективного выполнения циклов с зависимостью в MPI/DVMH-программе является наличие спецификации **tie** для всех across-массивов для измерений с ненулевыми длинами зависимостей.

На рис. 1 показан пример параллельного цикла, который может выполняться на многоядерном процессоре или ускорителе.

```
#pragma dvm parallel(3) reduction (max(eps)) // Для языка C-DVMH
for (int i = L1; i <= H1; i++)
    for (int j = L2; j <= H2; j++)
        for (int k = L3; k <= H3; k++)
...
!DVM$ PARALLEL(I,J,K) REDUCTION (MAX(EPS))   ! Для языка Fortran-DVMH
DO I = L1,H1
   DO J = L2, H2
      DO K = L3, H3
...
```

Рис. 1. Нераспределенный параллельный цикл

2.3 Актуальность данных

Фрагменты программы вне вычислительных регионов всегда выполняются на центральном процессоре. Как уже отмечалось, для каждого региона указываются данные, необходимые для его выполнения (входные, выходные, локальные), и перемещения данных между вычислительными регионами осуществляются в соответствии с имеющейся в описании регионов информацией об используемых ими данных. Для управления перемещениями данных между регионами и фрагментами программы вне регионов предусмотрены специальные директивы:

getactual-directive ::= **get_actual** (array-range-list)
actual-directive ::= **actual** (array-range-list)
array-range ::= var-name [subscript-range]
subscript-range ::= [int-expr [: int-expr]]

Директива **get_actual** делает все необходимые обновления для того, чтобы в памяти центрального процессора были актуальные (т.е. самые новые) значения данных в указанных в списке подмассивах и скалярах.

Директива **actual** объявляет тот факт, что указанные в списке подмассивы и скаляры имеют самые новые значения в памяти центрального процессора. Значения указанных переменных и элементов массивов, находящиеся в памяти ускорителей, считаются устаревшими и перед использованием будут при необходимости обновлены.

2.4 Функция, вызываемая из вычислительного региона

Функции, вызываемые из вычислительных регионов и параллельных циклов, не должны иметь побочных эффектов и содержать обменов между процессорами (т.н. прозрачные функции). Как следствие этого, прозрачные функции не должны содержать операторов ввода-вывода, обращений к функциям MPI-библиотеки и DVMH-директив (например, вложенных параллельных циклов).

Перед объявлением и определением функций, вызываемых из вычислительного региона, необходимо указать спецификацию **routine**:

routine-directive ::= **routine**

Данная спецификация говорит компилятору о необходимости сгенерировать для данной функции код, который может выполняться на многоядерном процессоре, сопроцессоре и графическом ускорителе.

3. Запуск MPI/DVMH-программ

Для компиляции и запуска MPI/DVMH-программ на выполнение требуется использовать специальную версию DVM-системы. Данная версия не производит никаких коммуникаций и может быть запущена в каждом MPI-процессе.

Перед выполнением MPI/DVMH-программы необходимо настроить следующие переменные окружения (например, их можно задать в скрипте dvm):

1. **DVMH_PPN** – количество MPI-процессов, которые будут запускаться на узле кластера. Положительное целое число или список неотрицательных целых чисел.
2. **DVMH_NUM_THREADS** – закольцованный список неотрицательных целых чисел, задающий количество нитей в каждом из MPI-процессов. Индексируется номером MPI-процесса.
3. **DVMH_NUM_CUDAS** – закольцованный список неотрицательных целых чисел, задающий количество графических ускорителей в каждом из MPI-процессов. Индексируется номером процесса. В текущей реализации максимальное количество графических ускорителей, которые может использовать MPI-процесс, равно 1.

Настройка всех этих переменных является обязательной для MPI/DVMH-программ. Эти переменные позволяют правильным образом распределить вычислительные ресурсы узла кластера между различными MPI-процессами, привязать создаваемые нити к ядрам центрального процессора, распределить графические ускорители между MPI-процессами.

Например, конфигурация:

```
export DVMH_PPN='3'
```

```
export DVMH_NUM_THREADS='0'
export DVMH_NUM_CUDAS='1'
```

позволит запустить на каждом узле кластера 3 MPI-процесса, каждый из которых будет использовать свою графическую карту.

Для узла кластера с 2-мя восьмиядерными процессорами может быть использована следующая конфигурация:

```
export DVMH_PPN='2'
export DVMH_NUM_THREADS='8'
export DVMH_NUM_CUDAS='0'
```

В результате – на каждом узле кластера будут запущены 2 MPI-процесса, каждый из которых создаст по 8 нитей и они будут привязаны к своим ядрам.

Отсутствие или некорректное определение любой из перечисленных выше переменных может приводить к неэффективному выполнению программы. Например, несколько MPI-процессов будут пытаться использовать одну и ту же графическую карту.

Возможность использования списка неотрицательных целых чисел в качестве значений данных переменных позволяет для каждого узла кластера и для каждого MPI-процесса задавать свои значения.

Например, конфигурация:

```
export DVMH_PPN='2'
export DVMH_NUM_THREADS='16,0'
export DVMH_NUM_CUDAS='0,1'
```

запустит на каждом узле кластера 2 MPI-процесса, один из которых будет использовать 16 нитей, а другой – графическую карту. В данном случае задача балансировки нагрузки между различными вычислительными устройствами ложится на плечи MPI-программиста.

После задания указанных выше переменных окружения, MPI/DVMH-программа может быть запущена на счет при помощи команды:

./dvm run < MPI-process number > < program name > [< task parameters >]

В результате выполнения данной команды будет создано необходимое количество MPI-процессов, в каждом из которых система поддержки выполнения DVMH-программ будет распределять вычисления по ядрам центрального процессора или графического ускорителя.

4. Анализ эффективности MPI/DVMH-программ

Для анализа и отладки эффективности выполнения DVMH-программ созданы инструментальные средства, функционирующие следующим образом. Система поддержки выполнения DVMH-программ накапливает информацию с временными характеристиками выполнения программы в оперативной памяти. При завершении выполнения программы эта информация записывается в файл, который затем обрабатывается специальным инструментом – визуализатором производительности.

С помощью визуализатора производительности пользователь имеет возможность получить временные характеристики выполнения его программы с различной степенью подробности (вся программа; параллельные и последовательные циклы; а также любые отмеченные программистом последовательности операторов).

Помимо потерь, связанных с выполнением межпроцессорных обменов, при выполнении программы на ускорителях собираются характеристики, которые позволяют оценить времена выполнения вычислительных регионов, а также потери, связанные с:

- копированием данных из памяти центрального процессора в память ускорителя и обратно (при входе и выходе из вычислительного региона);
- приведением переменных в памяти центрального процессора и ускорителя в согласованное состояние (операции actual/get_actual);
- копированием данных для выполнения операций типа shadow, reduction, remote, across;
- выполнением различных динамических оптимизаций, реализуемых системой поддержки выполнения DVMH-программ для более эффективного использования ресурсов ускорителя (например, реорганизации массивов в памяти ускорителя).

Результат работы визуализатора производительности для программы, о которой пойдет речь в 6 главе, показан на рис. 2.

-- The GPU characteristics ---

Proc: #1

GPU #3 (Tesla V100-PCIE-32GB)

	#	Min	Max	Sum	Average	Productive	Lost
[Region IN] Copy CPU to GPU	619	2.004K	2.020G	7.469G	12.355M	1.9493s	-
GET_ACTUAL	618	2.004K	1.006M	412.805M	683.999K	0.6020s	-
Loop execution	206	0.0022	0.0138	1.4532	0.0071	1.4532s	-
Reduction	206	0.0001	0.0010	0.0144	0.0001	-	0.0144s

Productive time: 4.0044s
Lost time : 0.0144s

Рис. 2. Статистика выполнения программы на графическом ускорителе

Отображается количество выполненных операций, времена выполнения этих операций, а также объемы копируемых данных. Как уже отмечалось, такая информация может быть собрана для любого фрагмента распараллеливаемой программы. Например, если при компиляции программы указать ключ **–e4**, то такая информация будет получена для каждого параллельного цикла в программе.

После завершения выполнения MPI/DVMH-программы каждый процесс сохраняет информацию с временными характеристиками в свой файл, каждый из которых затем может быть проанализирован отдельно. В настоящее время ведется разработка новой диалоговой оболочки для работы со статистиками, которая должна упростить процесс анализа эффективности DVMH-программ. Ключевые возможности разрабатываемой системы: одновременная работа с несколькими статистиками, возможность сравнения характеристик для различных запусков, возможность поинтервального анализа, сортировка интервалов по значимости, а также графическое представление результатов – строятся диаграммы, графики и.т.п. Реализация данной системы должна существенно упростить анализ эффективности MPI/DVMH-программ (например, все статистики, полученные для разных MPI-процессов, могут быть отображены/проанализированы все вместе).

5. Отладка MPI/DVMH-программ

DVMH-модель имеет важное преимущество по сравнению с другими высокоуровневыми моделями программирования для ускорителей. В модели DVMH перемещение данных между памятью центрального процессора и памятью ускорителей не задается явно, а осуществляется автоматически в соответствии со спецификациями использования данных в вычислительных регионах. Например, спецификация **copyin**(A) в модели OpenACC означает, что для дальнейшего выполнения программы необходимо скопировать массив A из памяти центрального процессора в память ускорителя; аналогичная спецификация **in**(A) в модели DVMH означает, что если следующий фрагмент программы будет выполняться на ускорителе и массив A отсутствует в памяти ускорителя, то необходимо скопировать массив A в память ускорителя. Если массив A уже присутствует на ускорителе, или следующий фрагмент программы будет выполняться на центральном процессоре, то никаких операций копирования не возникает. Такой подход дает следующие преимущества:

- позволяет динамически решать, где выгоднее выполнять тот или иной регион;
- позволяет многократно выполнять регион для нахождения оптимального отображения вычислений на графический ускоритель;

– позволяет сравнивать результаты выполнения региона на центральном процессоре и графическом ускорителе с целью обнаружения расхождений в результатах выполнения.

Это дало возможность реализовать специальный режим работы DVMH-программы, в котором все вычисления в регионах одновременно выполняются на центральном процессоре и графическом ускорителе.

Сравнение данных на входе в регион позволяет обнаружить отсутствие спецификации **out** для ранее выполненных регионов или директивы актуализации **actual**.

Сравнение выходных данных, полученных в регионе при выполнении на графическом ускорителе, с данными, полученными в регионе при выполнении на центральном процессоре, позволяет выявить и локализовать ошибки, проявляющиеся при работе на ускорителях.

В сравнение включаются все выходные данные вычислительного региона. При этом целочисленные данные сравниваются на совпадение, а вещественные числа сравниваются с заданной точностью по абсолютной и относительной погрешности. В случае нахождения расхождений пользователю выдается информация об этих расхождениях. Далее в программе используется та версия данных, которая была получена при выполнении на центральном процессоре.

Таким образом, для отладки MPI/DVMH-программ может быть использована следующая методика:

– на первом этапе программа отлаживается как MPI-программа, используя TotalView, Intel Trace Analyzer and Collector или другие инструменты отладки MPI-программ. Это возможно в силу того, что DVMH-директивы не видны обычным компиляторам;
– на втором этапе программа запускается в специальном режиме, когда сравниваются промежуточные результаты параллельного выполнения программы на центральном процессоре и графическом ускорителе с целью обнаружения расхождений в результатах выполнения.

Обнаруживаются следующие типы ошибок:

1. Программистом произведено некорректное распараллеливание, не подходящее для массивно-параллельного выполнения в общей памяти.
2. Программист некорректно указал приватные или редукционные переменные в параллельном цикле.
3. Арифметические операции или математические функции на ускорителе отработали с иным по сравнению с работой на центральном процессоре результатом. Это может происходить из-за различий в системе команд, приводящих к различным результатам (в пределах точности округлений).
4. Программист указал неверные директивы актуализации данных **get_actual** и **actual**, вследствие чего обрабатываемые данные на центральном процессоре и ускорителе оказались разными.

Включение и использование данного режима сравнительной отладки не требует от программиста вносить какие-либо изменения в программу, инструментировать ее, а также заново ее компилировать. Для включения режима сравнительной отладки необходимо установить значение переменной окружения DVMH_COMPARE_DEBUG равным 1, либо для запуска программы на выполнение использовать команду: **./dvm cmph**.

В случае обнаружения ошибок информация о них выдается в стандартный поток вывода ошибок или в файл. Имя этого файла можно указать в переменнной окружения DVMH_LOGFILE. Выводится информация о наличии ошибок и множество индексов элементов массивов с расхождениями в компактной форме.

Точность сравнения переменных можно изменить, указав значения переменных окружения DVMH_COMPARE_FLOATS_EPS, DVMH_COMPARE_DOUBLES_EPS, DVMH_COMPARE_LONGDOUBLES_EPS.

6. Апробация подхода

Рассмотрим процесс дополнительного распараллеливания программ в модели MPI/DVMH на примере программы Himeno, которая решает задачу Пуассона в трехмерной области с использованием итерационного метода Якоби [9]. Существует множество различных версий данной программы: на Фортране 77, Фортране 90, Си; MPI, OpenMP; со статическими, динамическими массивами. Для данного исследования использовалась MPI-версия программы, реализованная на языке Фортран 90. Исходный код программы составляет 814 строк.

Дополнительное распараллеливание данной программы потребовало:

1. Объявить 2 параллельных цикла в процедуре jacobi:

```
!DVM$ PARALLEL (K,J,I), PRIVATE(S0,SS), REDUCTION(SUM(WGOSA))
!DVM$ PARALLEL (K,J,I)
```

2. Объединить эти 2 параллельных цикла в вычислительный регион:

```
!DVM$ REGION
!DVM$ END REGION
```

3. Привести в согласованное состояние в памяти центрального процессора и ускорителя значение редукционной переменной WGOSA, которая используется для контроля сходимости итерационного метода:

```
!DVM$ ACTUAL(wgosa)
!DVM$ GET_ACTUAL(wgosa)
```

   Данные спецификации используются после обнуления переменной wgosa на центральном процессоре на каждой итерации и перед выполнением функции mpi_allreduce.

4. Скопировать необходимые элементы массива P (теневые грани) перед их отправкой и после их получения от соседних процессоров:

```
!DVM$ GET_ACTUAL(P(:,:,2),P(:,:,kmax-1))
!DVM$ ACTUAL(P(:,:,1),P(:,:,kmax))
!DVM$ GET_ACTUAL(P(:,2,:),P(:,jmax-1,:))
!DVM$ ACTUAL(P(:,1,:),P(:,jmax,:))
!DVM$ GET_ACTUAL(P(2,:,:),P(imax-1,:,:))
!DVM$ ACTUAL(P(1,:,:),P(imax,:,:))
```

Для данной программы имеется возможность задать: какие измерения массивов и на сколько частей необходимо распределить (параметр DDM pattern – «1 1 2»). В зависимости от распределения массивов вызываются 3 разные процедуры, которые пересылают необходимые данные. В каждой из этих процедур была добавлена одна директива ACTUAL и одна директива GET_ACTUAL. Если отказаться от данного параметра, например, всегда распределять массивы по 1-му измерению, то можно обойтись 2-мя директивами вместо 6.

Таким образом, для распараллеливания данного теста в модели MPI/DVMH потребовалось добавить в программу 12 DVMH-директив. Самым сложным оказалось определение данных, которые необходимо копировать из памяти ускорителя в память центрального процессора и обратно для выполнения межпроцессорных обменов. В таблице 1 показаны времена выполнения 100 итераций данной программы для массивов размером 1025x1025x525 на вычислительном кластере K60 (ИПМ им. М.В. Келдыша РАН) при использовании различного числа ядер и графических карт.

Таблица 1. Времена выполнения программы Himeno на гибридном кластере K60

Конфигурация	1 ядро	2 ядра	4 ядра	8 ядер	16 ядер	1 GPU	2 GPU	4 GPU	8 GPU
Время, в сек.	297.73	279.13	146.46	78.49	43.31	5.94	6.43	3.44	1.97

За счет использования 8 графических процессоров nVidia Volta GV100GL удалось ускорить выполнение программы почти в 22 раза по сравнению с ее выполнением на 16 ядрах центрального процессора Intel Xeon Gold 6142. При использовании 1 графического ускорителя программа выполняется почти в 50 раз быстрее, чем на одном ядре центрального процессора.

Заключение

Появление гибридных кластеров с ускорителями серьезно усложнило процесс разработки параллельных программ. Помимо распределения данных, распределения вычислений и выполнения межпроцессорных обменов между узлами кластера теперь требуется дополнительное распараллеливание. Необходимо дополнительно распределить данные и вычисления по вычислительным устройствам узла кластера (графическим ускорителям, сопроцессорам, многоядерным процессорам и др.) и организовать их параллельную обработку в рамках конкретного вычислительного устройства.

В статье были представлены новые возможности DVM-системы, которые могут быть использованы при таком дополнительном распараллеливании существующих MPI-программ, показаны преимущества модели DVMH по сравнению с моделями OpenMP и OpenACC:

1. Высокая эффективность получаемых программ, достигаемая за счет различных оптимизаций, которые выполняются как при компиляции DVMH-программ, так и во время выполнения. Например, динамическая реорганизация данных, динамическая компиляция кода CUDA-обработчиков во время выполнения программы и другие. Некоторые из этих оптимизаций становятся возможными за счет дополнительной информации в DVMH-программе: о зависимостях, о соответствии измерений цикла измерениям массивов.
2. Возможность распараллеливания циклов с зависимостью по данным на графических ускорителях.
3. Наличие инструментальных средств для анализа эффективности, которые работают в понятных пользователю терминах, собирают характеристики эффективности параллельного выполнения, которые связаны с конструкциями DVMH-языков.
4. Наличие инструментальных средств для автоматизированной отладки параллельных программ.
5. Простота DVMH-спецификаций параллелизма. Все основные DVMH-конструкции для распараллеливания MPI-программ были представлены в данной статье. Для сравнения –описание последней версии стандарта OpenACC занимает около 150 страниц, а полное описание стандарта OpenMP занимает более 600 страниц.

Таким образом, можно смело рекомендовать использовать модель DVMH для дополнительного распараллеливания MPI-программ как альтернативу моделям OpenMP и OpenACC.

Работа выполнена при поддержке Российского фонда фундаментальных исследований, проекты 19-07-00889-а, 19-07-01101-а и 20-01-00631-а.

Литература

1. OpenMP Application Programming Interface. Version 5.0. November, 2018. — URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf .
2. The OpenACC Application Programming Interface. Version 3.0. November, 2019. — URL: https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC.3.0.pdf .
3. OpenMP Compilers & Tools — URL: https://www.openmp.org/resources/openmp-compilers-tools/
4. Rahulkumar Gayatri, Charlene Yang. Optimizing Large Reductions in BerkeleyGW on GPUs Using OpenMP and OpenACC. — URL:

https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9626-optimizing-large-reductions-in-berkeleygw-with-cuda-openacc-openmp-and-kokkos.pdf .
5. Язык C-DVMH. C-DVMH компилятор. Компиляция, выполнение и отладка CDVMH-программ. — URL: http://dvm-system.org/static_data/docs/CDVMH-reference-ru.pdf .
6. Язык Fortran-DVMH. Fortran-DVMH компилятор. Компиляция, выполнение и отладка DVMH-программ. — URL: http://dvm-system.org/static_data/docs/FDVMH-user-guide-ru.pdf .
7. Система автоматизации разработки параллельных программ (DVM-система). — URL: http://dvm-system.org .
8. В.А. Бахтин, А.С. Колганов, В.А. Крюков, Н.В. Поддерюгина, М.Н. Притула. Методы динамической настройки DVMH-програм на кластеры с ускорителями // Суперкомпьютерные дни в России: Труды международной конференции (28-29 сентября 2015 г., г. Москва), М.: Изд-во МГУ, 2015, С. 257-268 .
9. Himeno benchmark. — URL: http://accc.riken.jp/en/supercom/documents/himenobmt/ .

References

1. OpenMP Application Programming Interface. Version 5.0. November, 2018. — URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf .
2. The OpenACC Application Programming Interface. Version 3.0. November, 2019. — URL: https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC.3.0.pdf .
3. OpenMP Compilers & Tools — URL: https://www.openmp.org/resources/openmp-compilers-tools/
4. Rahulkumar Gayatri, Charlene Yang. Optimizing Large Reductions in BerkeleyGW on GPUs Using OpenMP and OpenACC. — URL: https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9626-optimizing-large-reductions-in-berkeleygw-with-cuda-openacc-openmp-and-kokkos.pdf .
5. C-DVMH language, C-DVMH compiler, compilation, execution and debugging of DVMH programs. — URL: http://dvm-system.org/static_data/docs/CDVMH-reference-en.pdf .
6. Fortran DVMH langauge, Fortran DVMH compiler, compilation, execution and debugging of DVMH programs. — URL: http://dvm-system.org/static_data/docs/FDVMH-user-guide-en.pdf .
7. System for automating the development of parallel programs (DVM-system). — URL: http://dvm-system.org .

8. V.A. Bakhtin, A.S. Kolganov, V.A. Krukov, N.V. Podderugina, M.N. Pritula. Methods of dynamic tuning of DVMH programs on clusters with accelerators // Russian Supercomputing Days: Proccedings of International conference (28-29 september 2015, Moscow), Moscow: Moscow University Press, 2015, P. 257-268.
9. Himeno benchmark. — URL: http://accc.riken.jp/en/supercom/documents/himenobmt/ .