УДК 004.432

ОТЛАДКА ПАРАЛЛЕЛЬНЫХ ПРОГРАММ В DVM-СИСТЕМЕ

В.А. Бахтин [1,2], Д.А. Захаров [1], А.А. Ермичев [1,2], В.А. Крюков [1,2]

[1] Институт прикладной математики им. М.В. Келдыша Российской академии наук, г. Москва;
[2] Московский государственный университет им. М.В. Ломоносова, г. Москва

bakhtin@keldysh.ru, s123-93@mail.ru, a.ermich@gmail.com, krukov@keldysh.ru

Аннотация

DVM-система предназначена для разработки параллельных программ научно-технических расчетов на языках C-DVMH и Fortran-DVMH. Эти языки используют единую DVMH-модель параллельного программирования и являются расширением стандартных языков Си и Фортран спецификациями параллелизма, оформленными в виде директив для компилятора. DVMH-модель позволяет создавать эффективные параллельные программы для гетерогенных вычислительных кластеров, в узлах которых в качестве вычислительных устройств наряду с универсальными многоядерными процессорами могут использоваться ускорители, графические процессоры или сопроцессоры Intel Xeon Phi. В статье описаны методика отладки параллельных программ в DVM-системе, а также новые возможности DVM-отладчика.

Ключевые слова*: автоматизация разработки параллельных программ, автоматизация отладки параллельных программ, динамический контроль, сравнительная отладка, DVM-система, ускоритель, ГПУ, Фортран, Си.*

ВВЕДЕНИЕ

Ручные методы отладки, такие, как пошаговое исследование процесса выполнения алгоритма в определенных точках останова или отладочная печать, не позволяют адекватно работать с реальными научными и инженерными параллельными программными комплексами, спроектированными на непрерывную работу в течение часов или даже дней. Параллельные алгоритмы обычно значительно сложнее последовательных вариантов решения тех же задач. Более того, параллельный код может содержать нетипичные для последовательной отладки

ошибки, связанные с некорректным использованием примитивов синхронизации, функций, обеспечивающих параллелизм.

Следует отметить сложность отладки параллельных программ из-за:

- необходимости отслеживать состояние нескольких (очень многих) параллельных процессов/нитей;
- сложности повторного воспроизведения ошибок, вызванной недетерминированностью выполнения;
- влияния отладочных средств на процесс выполнения (меняется время выполнения операторов, возможны внутренние синхронизации).

Для отладки параллельных программ разработаны автоматизированные методы, которые позволяют найти большинство ошибок в программе в автоматическом режиме с минимальным участием программиста. Одним из таких методов является динамический контроль корректности. Данный метод используется во многих инструментах отладки многопоточных программ: Helgrind[1], DRD[2], Intel Parallel Inspector[3]. В DVM-системе этот метод используется при отладке параллельных программ для кластеров с ускорителями.

Другим методом автоматизированной отладки является сравнительная отладка параллельных программ. Принцип работы данного метода заключается в сравнении процесса выполнения двух программ, посредством контроля значений переменных в определенных контролируемых точках. Сравнение может проводиться как между параллельно запущенными программами, так и с использованием файлов трассы, в которые заносятся все необходимые данные об операциях и значениях переменных в контролируемых точках. Подобные методы отладки были реализованы в отладчиках Guard[4] и Wizard[5], в описании которых впервые был использован термин «сравнительная отладка». Этот метод был реализован и в DVM-системе.

При использовании отладчиков Guard и Wizard контролируемые точки, в которых происходит сравнение значений переменных, задаются пользователем. В DVM-отладчике точки сравнения значений устанавливаются автоматически.

В статье кратко описана методика отладки DVMH-программ, представлены проблемы, возникающие при сравнительной отладке DVMH-программ, и предложены пути преодоления данных проблем.

МЕТОДИКА ОТЛАДКИ DVMH-ПРОГРАММ

Для применения автоматизированных методов отладки требуется инструментация параллельной программы. В программу добавляются специальные вызовы к отладчику, которые позволяют контролировать ход ее выполнения. Для выполнения инструментации возможны различные подходы: инструментация бинарного кода, инструментация исходного кода. Отладчики Helgrind, DRD, Intel Parallel Inspector основаны на бинарной инструментации. При использовании высокоуровневых моделей программирования (например, OpenMP [9]) применение бинарной инструментации приводит к определенным сложностям. Так, по бинарному коду программы инструментатор должен восстановить спецификации параллелизма, которые были в программе изначально, чтобы иметь возможность выдавать ошибки в терминах программы пользователя. Для этого инструментатор должен знать логику работы компилятора. Если компилятор и отладчик разработаны одной компанией (например, Intel), то такое восстановление возможно. Если компилятор разработан одной компанией (например, Microsoft), а отладчик другой (например, Intel), то восстановление спецификаций параллелизма может быть затруднено. Избежать данной проблемы можно, если использовать инструментацию исходного кода.

В DVM-системе инструментация отлаживаемых программ выполняется компиляторами с языков C-DVMH [6] и Fortran-DVMH [7], которые добавляют обращения к функциям DVM-отладчика в следующих точках программы:

- начало последовательного или параллельного циклов;
- завершение последовательного или параллельного циклов;
- начало нового витка цикла;
- обращение к переменной на чтение;
- перед обращением к переменной на запись;
- после обращения к переменной на запись.

Кроме того, обращения к функциям отладчика производятся также и при выполнении многих функций библиотеки Lib-DVMH [8].

Динамический контроль DVMH-указаний основан на анализе последовательности вызовов функций Lib-DVMH и обращений к переменным. Динамический контроль позволяет выявлять ошибки следующих типов:

- необъявленная зависимость по данным в параллельном цикле;
- некорректное использование приватных и редукционных переменных;
- необъявленный доступ к нелокальным элементам распределенного массива;
- некорректная работа с теневыми гранями редукционного массива
- модификация нелокального элемента распределенного массива в последовательной части программы;
- выход за пределы распределенного массива;
- запись в буфер удаленного доступа.

Следует отметить, что не все ошибки могут быть определены в результате динамического контроля. Например, с помощью динамического контроля не могут быть проанализированы процедуры и функции, для которых отсутствует исходный код. Для поиска ошибок в таких программах может быть использован метод сравнительной отладки, который более детально будет рассмотрен в следующем разделе.

СРАВНИТЕЛЬНАЯ ОТЛАДКА DVMH-ПРОГРАММ

Сравнение промежуточных результатов выполнения позволяет обнаруживать ошибки, возникающие из-за некорректных DVMH-указаний, а также ошибки программы, которые не проявлялись при ее последовательном выполнении и не были обнаружены методом динамического контроля.

Общая схема сравнительной отладки выглядит следующим образом:

1) Получение эталонной трассировки (команда **./dvm trc**). При трассировке выполняется сбор информации обо всех чтениях и модификациях переменных, о начале выполнения каждого витка цикла, о начале и конце выполнения параллельного цикла, о начале выполнения каждой параллельной задачи, о начале и конце выполнения группы задач. В качестве эталонной трассировки может выступать трасса последовательного выполнения программы (т. к. DVMH-программа одновременно может быть и последовательной, и параллельной); трасса параллельного выполнения программы, в т.ч. трасса, полученная на другом вычислительном кластере, на котором ошибка не проявляется.

2) Автоматическое сравнение результатов выполнения программы с накопленной ранее эталонной трассировкой (команда **./dvm dif**). Все целочисленные данные сравниваются на совпадение, а вещественные числа сравниваются с заданной точностью по абсолютной и относительной погрешностям. В случае нахождения расхождений выдается информация о найденных различиях.

Файл трассировки создается для каждого процесса и имеет текстовый формат. При накоплении трассировки в начало файла помещается специальный заголовок (рис. 1). Данный заголовок содержит параметры, которые использовались при накоплении трассировки. Также в заголовке сохраняется структура вложенности циклов и областей задач программы в том виде, в котором она представляется отладчику (она определяется по последовательности обращений к нему и передаваемым параметрам, таким, как номер конструкции).

```
<параметры запуска трассировки (архитектура системы, директории)>
MODE = <NONE | MINIMAL | MODIFY | FULL>
<SL | PL | TR> <номер конструкции> (<номер объемлющей конструкции>) [<ранг
цикла>] {<имя файла>, <номер строки>} = <NONE | MINIMAL | MODIFY | FULL>,
(<измерение>:<первый виток>, <последний виток>, <шаг>), …
EL: <номер конструкции>
<SL | PL | TR> <номер конструкции> (<номер объемлющей конструкции>) [<ранг
цикла>] {<имя файла>, <номер строки>} = <NONE | MINIMAL | MODIFY | FULL>,
(<измерение>:<первый виток>, <последний виток>, <шаг>), …
EL: <номер конструкции>
END_HEADER
```

Рис. 1. Формат заголовка файла трассы

Модифицируя значение MODE, можно контролировать глобальный уровень подробности трассировки программы, также можно задавать подробность трассировки для каждого цикла/параллельной области DVMH-программы, поставив соответствующий режим в строке, описывающей нужную конструкцию программы в заголовке.

После заголовка в файле содержится последовательность записей, формирующих собственно трассу. Эти записи определяют динамическую структуру программы, то есть последовательность операторов в процессе выполнения программы. На рис. 2 и 3 представлен полный список событий, которые заносятся в трассу.

```
- чтение переменной:
RD: [<тип переменной>] <имя переменной> = <значение>; {<файл>, <строка>}

- начало оператора присваивания нового значения переменной:
BW: [<тип переменной>] <имя переменной>; {<файл>, <строка>}

- запись нового значения переменной:
AW: [<тип переменной>] <имя переменной> = <значение>; {<файл>, <строка>}

- чтение редукционной переменной:
RV_RD: [<тип переменной >] <имя переменной> = <значение>; {<файл>, <строка>}

- начало оператора присваивания нового значения редукционной переменной:
RV_BW: [<тип переменной>] <имя переменной>; {<файл>, <строка>}

- запись нового значения редукционной переменной:
RV_AW: [<тип переменной>] <имя переменной> = <значение>; {<файл>, <строка>}

- результат вычисления редукции:
RV: [<тип переменной>] <значение>; {<файл>, <строка>}
```

Рис. 2. Формат записей событий взаимодействий с переменными

```
- начало параллельного цикла:
PL: <номер конструкции> (<номер объемлющей конструкции>) [<ранг цикла>];
{<файл>, <строка>}

- начало последовательного цикла:
SL: <номер конструкции> (<номер объемлющей конструкции>) [<ранг цикла
(всегда равен единице)>]; {<файл>, <строка>}

- начало области параллельных задач:
TR: <номер конструкции> (<номер объемлющей конструкции>) [<ранг области
 (всегда равен единице)>]; {<файл>, <строка>}

- начало витка или параллельной задачи:
IT: <абсолютный индекс витка (вычисляется из значений всех индексных
переменных цикла) или номер задачи>, (<значение индексной переменной 1>,<
значение индексной переменной 2>,…).

- конец параллельного или последовательного цикла или области задач:
EL: <номер конструкции>; {<файл>, <строка>}

- конец блока собственных вычислений в последовательной части программы:
SKP: {<файл>, <строка>}
```

Рис. 3. Формат записей, описывающих циклы и параллельные области

На рис. 5 показана трасса выполнения тестовой программы, листинг которой приведен на рис. 4. Данная программа состоит из одного параллельного цикла, который начинается на 7-ой строке файла test.c и выполняет инициализацию элементов распределенного массива A. В результате выполнения этого цикла элементы массива A[0] и A[2] получают значение «0», а элемент A[1] становится равным «3».

```
#define L 3
int main(int an, char **as)
{
    #pragma dvm array distribute[block]
    double A[L];
    #pragma dvm parallel([i] on A[i])
    for (int i = 0; i < L; i++)
    {
        if (i == 0 || i == L - 1)
            A[i] = 0;
        else
            A[i] = 2 + i;
    }
    return 0;
}
```

Рис. 4. Фрагмент тестовой программы на языке C-DVMH (файл test.c)

Текстовый формат для файла трассировки выбран не случайно. Такой формат позволяет переносить трассы, полученные на одной вычислительной системе, на другую вычислительную систему, не заботясь о необходимости преобразования данных (различия в размере типов данных, порядке байт в слове и т. п.), а также позволяет, при необходимости, вручную проанализировать ход выполнения отлаживаемой программы (вместо добавления в программу операторов печати).

```
# Begin trace header. Don't modify these records
TRACE_TIME = "Mon Nov 15 00:00:22 2019"
ARCHITECTURE = "Machine x86_64"
USER_HOST = "DVM-COREI7@DVM-COREI7"
WORK_DIR = "/DVM/dvm_current/dvm_sys/demo"
TASK_NAME = "test"
MODE = FULL
PL: 1() [1] {"test.c", 7} = #, (0:0,2,1)
EL: 1
END_HEADER
# End trace header
PL: 1() [1]; {"test.c", 7}, 1.B
  IT: 0, (0)
  BW: [4] "A[i]"; {"test.c", 10}
  AW: [4] "A[i]" = 0; {"test.c", 10}
  IT: 1, (1)
  BW: [4] "A[i]"; {"test.c", 12}
  AW: [4] "A[i]" = 3; {"test.c", 12}
  IT: 2, (2)
  BW: [4] "A[i]"; {"test.c", 10}
  AW: [4] "A[i]" = 0; {"test.c", 10}
EL: 1; {"test.c", 7}, 1.E
END_TRACE
```

Рис. 5. Файл трассы выполнения тестовой программы

Сравнительная отладка показала свою эффективность на простых модельных задачах, но натолкнулась на два препятствия: ресурсы и точность. Рассмотрим подробней суть этих проблем.

ПРОБЛЕМЫ СРАВНИТЕЛЬНОЙ ОТЛАДКИ

После полной инструментации каждый оператор окружается несколькими вызовами подпрограмм трассировщика, которые должны сформировать записи с читаемыми и модифицируемыми значениями переменных. Неудивительно, что при этом время выполнения программы может значительно увеличиться. Например, в результате экспериментов с программами из пакета NAS NPB [11] были получены следующие результаты:

- замедление только от отладочной инструментации (т. е. программа была скомпилирована для отладки, но выполнена без сбора/сравнения трассы) составило 50–100 раз;
- объемы трассы (в среднем несколько десятков байтов на каждый выполненный оператор присваивания) оказались совершенно неприемлемыми для реальных программ; средний размер трассы – порядка нескольких терабайт, и среднее время сбора трасы ~28000 сек. (при среднем времени выполнения исходных программ ~5 сек., т. е. замедление в ~4000 раз!).

Поэтому полная сравнительная отладка оказалась применимой только на «модельных» данных, которые не всегда доступны. Для преодоления данной проблемы были разработаны и реализованы в отладчике DVM-системы средства управления и оптимизации трассы [12]:

- выборочная трассировка, например: только запись, только распределенные массивы и т. п. (режимы **-d1**…**-d4**, которые можно задать при компиляции DVMH-программы);
- локализация инструментации, т. е. выделение отдельных фрагментов программы, инструментируемых для отладки (директивы **DEBUG**<режим отладки>/**END DEBUG**);
- предварительная оценка объема трассы, получение заголовка трассы и его ручная коррекция для отключения трассировки на определенных итерациях и т. д. (команда **./dvm size**);

- автоматический выбор итераций параллельных циклов для трассировки: «границы» и «уголки» (режимы **-dbif1**, **-dbif2**, которые можно задать при компляции DVMH-программы);
- генерация двух тел цикла: одно – без вызовов трассировщика, другое инструментировано; при этом на выбранных для трассировки итерациях работает инструментированное тело цикла, на остальных – исходная программа;
- условный вызов подпрограмм трассировщика (т. е. перед вызовом подпрограммы трассировщика проверяется необходимость ее вызова);
- трассировка контрольных сумм массивов по завершении цикла вместо трассировки элементов массивов при выполнении тела цикла (параметр **TraceOptions.CalcChecksums**).

Использование данных средств расширяет возможности применения сравнительной отладки для реальных приложений. Например, при обработке больших массивов данных в циклах наиболее вероятным местом возникновения и проявления многих ошибок (использование неинициализированных переменных, выход за границы массива) могут стать граничные итерации. Вычислительные алгоритмы часто таковы, что ошибка, возникшая на внутренней итерации цикла, проявляется и на граничной. Для таких задач можно использовать метод выборочной трассировки и сравнения граничных итераций цикла (опция – **dbif**<level>), который позволяет (таблица 1):

- существенно сократить размер трасс (в сотни – сотни тысяч раз);
- существенно сократить время выполнения программ с генерацией трасс (в десятки – тысячи раз);
- сохранить значительное покрытие операторов программы (более 99%).

Таблица 1. Размер и время генерации трасс для тестов NAS NPB класса A при использовании различных режимов инструментации

	Полная инструментация	**«Уголки» ширины 1**	**«Уголки» ширины 2**	**«Грани» ширины 1**	**«Грани» ширины 2**
Среднее покрытие операторов	100%	99,4%	99,8%	99,8%	99,8%
Средний размер трассы, байт	6,57E+12 (~6 Tb)	4,6E+07 (~44 Mb)	9,4E+08 (~894 Mb)	1,7E+10 (~16 Gb)	6,0E+10 (~56 Gb)
Среднее время генерации трассы, сек.	27915 (7,75 ч.)	15	19	105	287

Тем не менее, сравнительная отладка научно-технических программных комплексов, особенно на реальных данных (а не искусственно подобранных «модельных» тестах небольшого размера) все еще не доступна в полном объеме.

Другая проблема, выявленная в процессе эксплуатации системы отладки, – «допустимое» несовпадение значений переменных. Такими переменными являются, например, редукционные переменные. При вычислении суммы элементов распределенного массива изменяется порядок операций: сначала вычисляются локальные суммы, а потом они суммируются в непредсказуемом порядке. Результаты при этом получаются разными (типично расхождение в 1–2 младших разрядах, хотя возможно и более значительное), но, с точки зрения программиста, эти результаты могут быть одинаково допустимыми. Редукционные операции создают четыре проблемы для сравнительной отладки («ложные тревоги»):

- значения редукционной переменной на промежуточных итерациях не совпадают, потому что вычисляются только частичные суммы или максимум ищется в другом диапазоне и т. п.;
- окончательное значение суммы, как сказано выше, может отличаться (недетерминизм в слабом смысле);
- отличие в одной переменной может сразу распространиться на многие другие (например, если посчитана норма вектора, и затем вектор нормируется);

- если значение переменной, на которую повлияло различие результатов редукции, используется в критерии окончания итераций или при выборе ветви вычислений, то дальше могут быть выбраны разные ветви, и трассы вообще могут оказаться несопоставимыми (недетерминизм в сильном смысле).

Данные проблемы были решены введением особого режима обработки редукции при отладке:

- редукционные переменные распознаются (т. к. они описаны в DVMH-директивах), и операции их чтения и модификации в теле цикла игнорируются сравнительной отладкой;
- после достижения конца параллельного цикла в трассу заносится итоговое значение редукционной переменной (конструкция RV), которое и участвует в сравнении;
- значения редукционных переменных сравниваются с некоторой точностью, которая может быть меньше точности сравнения значений обычных переменных; данная точность может быть задана программистом через специальный конфигурационный параметр;
- в процессе сравнения реального выполнения с трассой выполнения другого варианта реально вычисленное значение редукционной переменной (после успешного сравнения с заданной точностью) заменяется ее значением из «эталонной» трассы для ликвидации потенциально опасного расхождения.

Описанный подход позволил подавить «ложные тревоги», связанные с редукционными переменными. Однако в дальнейшем было обнаружено, что данная проблема может проявиться не только на редукционных переменных. При выполнении отлаживаемой программы на разных машинах или при использовании различных средств компиляции любая арифметическая операция, например, умножение или деление, может вернуть результаты, отличающиеся в одном-двух младших десятичных знаках мантиссы [13].

РАСШИРЕНИЕ ВОЗМОЖНОСТЕЙ СРАВНИТЕЛЬНОЙ ОТЛАДКИ

Для обнаружения ошибок, проявляющихся при выполнении программы на графических ускорителях, в DVM-системе реализован специальный режим работы DVMH-программы, при котором все вычисления в регионах одновременно выполняются на центральном процессоре и графическом ускорителе. Сравнение выходных данных, полученных в регионе при выполнении на ускорителе, с данными, полученными в регионе при выполнении на центральном процессоре, позволяет выявить и локализовать ошибки, которые могут возникать по следующим причинам:

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 результатом, иным по сравнению с работой на центральном процессоре. 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.

При сравнении все целочисленные данные сравниваются на совпадение, а вещественные числа сравниваются с заданной точностью по абсолютной и относительной погрешностям. В случае нахождения расхождений пользователю системы выдается информация о найденных расхождениях. После этого для дальнейшей работы программы берется версия данных, которая была получена при выполнении на центральном процессоре.

Включение и использование данного режима сравнительной отладки не требует от программиста вносить какие-либо изменения в программу, а также заново ее компилировать. Для включения данного режима сравнительной отладки необходимо установить значение переменной окружения

DVMH_COMPARE_DEBUG равным 1 либо использовать команду **./dvm cmph** для запуска программы на выполнение.

Данный режим сравнительной отладки стал очень востребован при разработке DVMH-программ. Одно из главных примуществ данного метода – возможность его использования для больших программных комплексов на реальных данных (т. к. не требуется память для хранения трасс).

В настоящее время в DVM-системе ведется разработка новой версии системы сравнительной отладки [14], которая основана на схожих принципах. Вместо использования файлов с трассами новая версия отладчика организует обмены отладочной информацией между двумя экземплярами программы, которые выполняются одновременно:

1) программа разделяется на блоки (например, в качестве блока могут рассматриваться параллельный цикл программы или вычислительный регион);

2) в процессе выполнения очередного блока происходит накопление очередного отрезка трассы каждым экземпляром отлаживаемой программы;

3) по завершению выполнения блока накопленные трассы пересылаются одному процессу, где и происходит их сравнение.

При использовании данного подхода остается возможность применения всех ранее реализованных оптимизаций, связанных с размером трассы (например, использование интегральных характеристик массивов), так как все изменения, вносимые данными режимами, влияют на трассу локально, внутри одного конкретного блока трассы.

Разбиение параллельной DVMH-программы на блоки позволяет выполнять вычисления внутри блока без дополнительных затрат на вызовы отладчика. Значения всех прочитанных и/или модифицированных переменных могут собираться в конце выполнения блока, а не в процессе его выполнения. Отказ от вызовов отладчика перед/после каждого обращения к переменной на чтение/запись внутри блока позволит существенно ускорить процесс отладки программы.

Для решения проблемы расхождения результатов операций с вещественными числами в новой версии отладчика будет реализован режим, расширяющий описанный ранее режим корректировки редукционных переменных на все

результаты вещественных операций. После успешного сравнения значения вещественной переменной с эталонным данное эталонное значение будет подставлено в выполняющуюся программу и использовано для дальнейших вычислений.

Реализация метода сравнения одновременно выполняющихся программ предполагает использование механизма передачи данных между независимо работающими задачами. Данный механизм был реализован на основе библиотеки, написанной для DVM-системы, предназначавшейся для контроля корректности выполнения программ через механизм контрольных точек.

Одной из особенностей данной библиотеки является модуль, предназначенный для обмена данными между программами по модели клиент-сервер, в том числе, и удаленно. Данный модуль был переработан для интеграции с существующей реализацией отладчика DVM-системы и подключен к ней.

Для реализации новых механизмов были реализованы/модифицированы существующие вызовы отладчика:

- добавлены обработчики, накапливающие список задействованных внутри блока переменных;
- в методы dbegpl_ и dendl_, вызываемые в начале и конце параллельных циклов, был добавлен вызов механизма передачи текущего блока другой программе;
- в структуры данных, использующиеся для накопления трассировки, была добавлена возможность хранения, обработки отдельных блоков трассы.

В таблице 1 представлены первые результаты тестирования новой версии системы сравнительной отладки на программе Якоби (одной из тестовых программ DVM-системы), где block – это максимальный размер одного блока, передаваемого между эталонной и отлаживаемой программами, а all – сумарный объем данных, переданных в процесса отладки.

Таблица 1. Сравнение экспериментальной реализации системы сравнительной отладки с существующей версией

	N = 10 100 итераций		N = 100 100 итераций		N = 1000 1000 итераций	
	Размер Трассы	Время работы	Размер трассы	Время работы	Размер трассы	Время работы
Без отладки	-	0.42 s	-	0.45 s	-	3.02 s
Существующая Реализация	4.6 Mb	0.5 s	694 Mb	9.35 s	~800 Gb	~12000 s
Сравнение одновременно выполняющихся программ	1.08 Mb (all) 7.4 K (block)	0.92 s	163 Mb (all) 1.1 Mb (block)	31.2 s	~180 Gb (all) 113 Mb (block)	~18000 s

Тестирование экспериментальной реализации нового режима показало в принципе успешный результат – максимальный объем памяти, требующийся отладчику для корректной работы, снизился до размеров одного блока. Время выполнения отладки увеличилось за счет необходимости постоянной передачи блоков данных, но это не является препятствием для отладки программных комплексов на реальных данных.

ЗАКЛЮЧЕНИЕ

Система автоматизации разработки параллельных программ (DVM-система) существенно упрощает процесс разработки параллельных программ для гетерогенных кластеров с ускорителями. Важным преимуществом DVM-системы является наличие мощных инструментов для отладки, получаемых в процессе распараллеливания программ. Поддерживаются различные варианты сравнительной отладки: с использованием трасс; сравнение результатов выполнения программы на центральном процессоре и графическом ускорителе «на лету».

В настоящее время ведется разработка новой версии системы сравнительной отладки, в которой и эталонная, и отлаживаемая программы будут выполняться одновременно, и сравнение результатов выполнения будет осуществляться «на лету» без трасс. Создаваемая версия системы отладки решит проблему точности сравнения, сделает процесс отладки более гибким и менее требовательным к памяти вычислительного комплекса. Новая версия системы сравнительной

отладки будет допускать как одновременный запуск эталонной и отлаживаемой программ на одной многопроцессорной машине, так и удаленную отладку – сравнение выполнения эталонной программы на одной машине с экспериментальной версией, работающей на другом вычислительном комплексе и использующей иные средства компиляции, при наличии технической возможности создания сетевого соединения между двумя машинами.

Разработка новой системы существенно расширит возможности применения сравнительной отладки для реальных приложений на реальных данных.

СПИСОК ЛИТЕРАТУРЫ

1. Helgrind: a thread error detector. URL: http://www.valgrind.org/docs/ manual/ hg-manual.html

2. DRD: a thread error detector. URL: http://www.valgrind.org/ docs/ manual/drd-manual.html

3. Intel Inspector. Memory and Thread Debugger. URL: https:// software.intel.com/en-us/intel-inspector.

4. Guard Parallel Relative Debugger. URL: http://sourceforge.net/ projects/guardsoft/

5. *Abramson D.A., Sosic R.* Relative Debugging using Multiple Program Versions // Intensional Programming I. Sydney: World Scientific. 1995.

6. Язык C-DVMH. C-DVMH компилятор. Компиляция, выполнение и отладка CDVMH-программ. URL: http://dvm-system.org/static_data/docs/CDVMH-reference-ru.pdf

7. 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

8. Система поддержки выполнения параллельных программ (библиотека Lib-DVM). URL: http://www.keldysh.ru/dvm/dvmhtm1107/rus/sys/libdvm/rtsDDr0.html

9. OpenMP Application Programming Interface. Version 5.0. November, 2018. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

10. The OpenACC Application Programming Interface. Version 2.6. November, 2017. URL: https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final.pdf

11. NAS Parallel Benchmarks, URL: http://www.nas.nasa.gov/publications/npb.html

12. *Крюков В.А., Кудрявцев М.В.* Автоматизация отладки параллельных программ // Вычислительные методы и программирование. 2006. Т. 7. Вып. 4. С. 102–110.

13. *David Monniaux.* The pitfalls of verifying floating-point computations // ACM Transactions on Programming Languages and Systems (TOPLAS), ACM. 2008. V. 30. No 3. 12 p.

14. *Ермичев А.А., Крюков В.А.* Развитие метода сравнительной отладки DVMH-программ // Научный сервис в сети Интернет: труды XIX Всероссийской научной конференции (18–23 сентября 2017г., г. Новороссийск). М.: ИПМ им. М.В. Келдыша, 2017. С. 150–156.

DEBUGGING PARALLEL PROGRAMS IN DVM-SYSTEM

***V.A. Bakhtin*[1,2], *D.A. Zakharov*[1], *A.A. Ermichev*[1,2], *V.A. Krukov*[1,2]**

[1] *Keldysh Institute of Applied Mathematics,*
[2] *Lomonosov Moscow State University*

bakhtin@keldysh.ru, a.ermich@gmail.com, s123-93@mail.ru, krukov@keldysh.ru

Abstract

DVM-system is designed for the development of parallel programs of scientific and technical calculations in the C-DVMH and Fortran-DVMH languages. These languages use a single DVMH-model of parallel programming model and are an extension of the standard C and Fortran languages with parallelism specifications in the form of compiler directives. The DVMH model makes it possible to create efficient parallel programs for heterogeneous computing clusters, in the nodes of which accelerators, graphic processors or Intel Xeon Phi coprocessors can be used as computing devices along with universal multi-core processors. The article describes the method of debugging parallel programs in DVM-system, as well as new features of DVM-debugger.

Keywords*: automation of development of parallel programs, automation of debugging of parallel programs,* dynamic control, relative debugger, *DVM-system, accelerator, GPU, Fortran, C*

REFERENCES

1. Helgrind: a thread error detector. URL: http://www.valgrind.org/docs/ manual/hg-manual.html

2. DRD: a thread error detector. URL: http://www.valgrind.org/docs/ manual/drd-manual.html

3. Intel Inspector. Memory and Thread Debugger. URL: https:// software.intel.com/en-us/intel-inspector

4. Guard Parallel Relative Debugger. URL: http://sourceforge.net/ projects/guardsoft/

5. *Abramson D.A., Sosic R.* Relative Debugging using Multiple Program Versions // Intensional Programming I. Sydney: World Scientific. 1995.

6. 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

7. 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

8. Run-time library Lib-DVM. URL: http://www.keldysh.ru/dvm/dvmhtm1107/ eng/sys/libdvm/rtsDDe0.html

9. OpenMP Application Programming Interface. Version 5.0. November, 2018. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

10. The OpenACC Application Programming Interface. Version 2.6. November, 2017. URL: https://www.openacc.org/sites/default/files/inline-files/ OpenACC.2.6.final.pdf

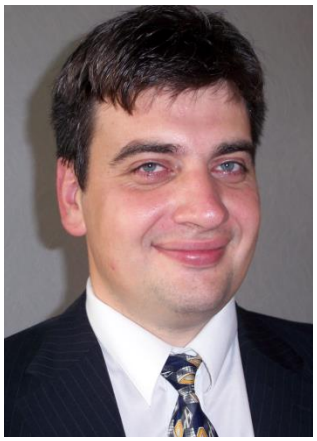
11. NAS Parallel Benchmarks, URL: http://www.nas.nasa.gov/publications/ npb.html

12. *Krukov V.A., Kudryavtsev M.B.* Automated debugging of parallel programs // Vychisl. Metody Programm. 2006. T. 4. S. 102–110.

13. *David Monniaux.* The pitfalls of verifying floating-point computations // ACM Transactions on Programming Languages and Systems (TOPLAS), ACM. 2008. V. 30. No 3. 12 p.

14. *Ermichev A.A., Krjukov V.A.* Razvitie metoda sravnitel'noj otladki DVMH-programm // Nauchnyj servis v seti Internet: trudy XIX Vserossijskoj nauchnoj konferencii (18–23 sentjabrja 2017 g., g. Novorossijsk). M.: IPM im. M.V. Keldysha, 2017. S. 150–156.

СВЕДЕНИЯ ОБ АВТОРАХ



БАХТИН Владимир Александрович – ведущий научный сотрудник ИПМ им. М.В. Келдыша РАН, доцент кафедры системного программирования факультета ВМК МГУ им. М.В. Ломоносова. Сфера научных интересов – математическое обеспечение, программные средства и системы для распределенных вычислений; параллельные алгоритмы; методы, средства и системы обработки данных большого объема.

Vladimir Aleksandrovich BAKHTIN – leading researcher of Keldysh Institute of Applied Mathematics, docent of the faculty of Computational Mathematics and Cybernetics of Lomonosov Moscow State University. Research interests include mathematical software, software and systems for distributed computing; parallel algorithms; methods, tools and systems of large data processing.

email: bakhtin@keldysh.ru

ЗАХАРОВ Дмитрий Александрович – программист ИПМ им. М.В. Келдыша РАН. Сфера научных интересов – программные средства и системы для распределенных вычислений; параллельные алгоритмы; автоматизация параллельного программирования; распараллеливание программ, использующих неструктурные сетки.

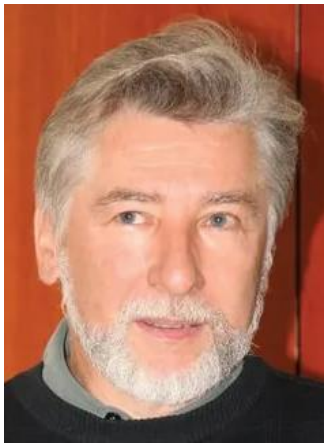
Dmitry Aleksandrovich ZAKHAROV – programmer of Keldysh Institute of Applied Mathematics. Research interests include mathematical software, software and systems for distributed computing; parallel algorithms; automatization of parallel programming; parallelization of unstructured grid applications.

email: s123-93@mail.ru

ЕРМИЧЕВ ***Александр Александрович*** – лаборант ИПМ им. М.В. Келдыша РАН, аспирант кафедры системного программирования факультета ВМК МГУ им. М.В. Ломоносова. Сфера научных интересов – программное обеспечение для распределенных вычислений; распределенные вычислительные системы; параллельные алгоритмы.

Aleksandr Aleksandrovich ERMICHEV – assistant of Keldysh Institute of Applied Mathematics, postgraduate of the Computational Mathematics and Cybernetics faculty of Lomonosov Moscow State University. Research interests include software for distributed computing; distributed computing systems; parallel algorithms.

email: a.ermich@gmail.com



КРЮКОВ Виктор Алексеевич – главный научный сотрудник ИПМ им. М.В. Келдыша РАН, профессор кафедры системного программирования факультета ВМК МГУ им. М.В. Ломоносова. Сфера научных интересов – математическое обеспечение, программные средства и системы для распределенных вычислений; параллельные алгоритмы; методы, средства и системы обработки данных большого объема.

Victor Alekseevich KRUKOV – chief researcher of Keldysh Institute of Applied Mathematics, professor of the faculty of Computational Mathematics and Cybernetics of Lomonosov Moscow State University. Research interests include mathematical software, software and systems for distributed computing; parallel algorithms; methods, tools and systems of large data processing.

email: krukov@keldysh.ru

Материал поступил в редакцию 13 ноября 2019 года