Министерство науки и высшего образования Российской Федерации
федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Московский государственный технический университет имени
Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Робототехники и комплексной автоматизации»
КАФЕДРА «Системы автоматизированного проектирования (РК-6)»

НАУЧНО-ИССЛЕДОВАТЕЛЬСКИЕ ЗАМЕТКИ

по направлению «Разработка систем инженерного анализа и ресурсоемкого ПО (rndhpc)»

Авторы (исследователи): Крехтунова Д., Ершов В., Муха В., Тришин И.

Научный(е) руководитель(и): Соколов А.П., Першин А.Ю.

Консультанты: @Фамилия И.О.@

Москва, 2021–2021

RNDHPC
C59

Работа (документирование) над научным направлением начата 20 сентября 2021 г.

Руководители по направлению:

СОКОЛОВ, Александр Павлович	–	канд. физ.-мат. наук, доцент кафедры САПР, МГТУ им. Н.Э. Баумана
ПЕРШИН, Антон Юрьевич	–	PhD, ассистент кафедры САПР, МГТУ им. Н.Э. Баумана

Исследователи (студенты кафедры САПР, МГТУ им. Н.Э. Баумана):
Крехтунова Д., Ершов В., Муха В., Тришин И.

C59 **Крехтунова Д., Ершов В., Муха В., Тришин И.. Разработка систем инженерного анализа и ресурсоемкого ПО (rndhpc)**: Научно-исследовательские заметки. / Под редакцией Соколова А.П. [Электронный ресурс] — Москва: 2021. — 16 с. URL: https://arch.rk6.bmstu.ru (облачный сервис кафедры РК6)

Документ содержит краткие материалы, формируемые обучающимися и исследователями в процессе их работ по одному научному направлению.
Документ разработан для оценки результативности проведения научных исследований по направлению «Разработка систем инженерного анализа и ресурсоемкого ПО» в рамках реализации курсовых работ, курсовых проектов, выпускных квалификационных работ бакалавров и магистров, а также диссертационных исследований аспирантов кафедры «Системы автоматизированного проектирования» (РК6) МГТУ им. Н.Э. Баумана.

RNDHPC

Содержание

1 Разработка графоориентированного дебаггера

2021.11.10: Первичный обзор литературы КП ММАПР

Анализ взвешенных графов вызовов для локализации ошибок программного обеспечения

Далее представлен материал, являющийся результатом анализа работы [1].

Проблема Обнаружение сбоев, которые приводят к ошибочным результатам с некоторыми, но не со всеми входными данными.

Предложенное решение Метод анализа графов вызовов функций с последующим составлением рейтинга методов, которые вероятнее всего содержат ошибку.

Описание решения *Граф вызовов*

Метод основан на анализе графа вызовов. Такой граф отражает структуру вызовов при выполнении конкретной программы. Без какой-либо дополнительной обработки граф вызовов представляет собой упорядоченное дерево с корнем.

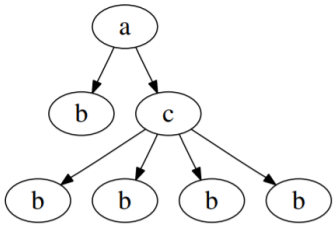


Рис. 1. Граф вызовов

Узлами являются сами методы, а гранями – их вызовы. Метод main() программы обычно является ее корнем, а все методы, вызываемые напрямую, являются ее дочерними элементами.

Редукция графа

Трассировка представляется в виде графов вызовов, затем повторяющиеся вызовы методов, вызванные итерациями, удаляются, вместо этого вводятся веса ребер, представляющие частоту вызовов.

Поиск подграфов

Для поиска подграфов используется фреймворк для ранжирования потенциально ошибочных методов.

После сокращения графов вызовов, полученных от правильного и неудачного выполнения программ, применяется поиск часто встречающихся замкнутых подграфов SG в наборе данных графа G, используя алгоритм CloseGraph. Полученный набор подграфов разделяется на те, которые встречаются при правильном и неудачном выполнении (SGcf), и те, которые возникают только при неудачном выполнении (SGf).

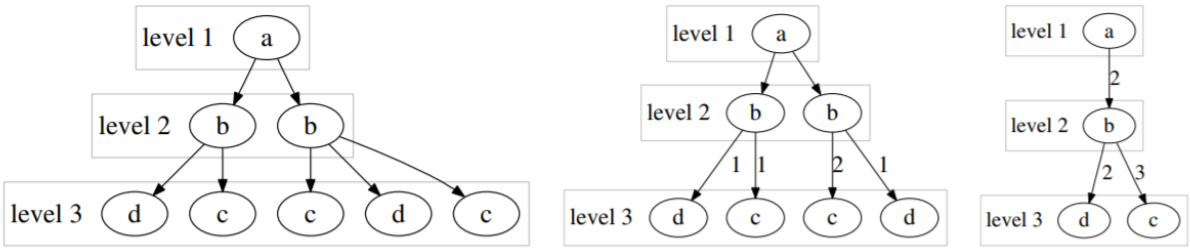


Рис. 2. Редукция графа вызовов

Анализ

Два набора подграфов рассматриваются отдельно.

SGcf используется для построения рейтинга на основе различий в весах ребер при правильном и неудачном выполнении. Графы анализируются: применяется алгоритм выбора характеристик на основе энтропии к весам различных ребер для вычисления вероятности вызова метода и вероятности содержания в нем ошибки.

Алгоритм на основе энтропии не может обнаружить ошибки, которые не влияют на частоту вызовов и не учитывает подграфы, которые появляются только в ошибочной версии (SGf).

Поэтому отдельно рассчитывается оценка для методов, содержащихся только в ошибочной версии. Эта оценка - еще одна вероятность наличия ошибки, основанная на частоте вызовов методов при неудачных выполнениях.

Затем вычисляется общая вероятность наличия ошибки для каждого метода. Этот рейтинг дается разработчику программного обеспечения, который выполняет анализ кода подозрительных методов.

Интегрированная отладка моделей Modelica

Далее представлен материал, являющийся результатом анализа работы [2].

Modelica – объектно-ориентированный, декларативный язык моделирования сложных систем (в частности, систем, содержащих механические, электрические, электронные, гидравлические, тепловые, энергетические компоненты).

Проблема Из за высокого уровня абстракции и оптимизации компиляторов, обеспечивающих простоту использования, ошибки программирования и моделирования часто трудно обнаружить.

Предложенное решение В статье представлена интегрированная среда отладки, сочетающая классическую отладку и специальные техники (для языков основанных на уравнениях) частично основанные на визуализации графов зависимостей.

Описание решения На этапе моделирования пользователь обнаруживает ошибку в нанесенных на график результатах, или код моделирования во время выполнения вызывает ошибку.

Отладчик строит интерактивный граф зависимостей (IDG) по отношению к переменной или выражению с неправильным значением.

Узлы в графе состоят из всех уравнений, функций, определений значений параметров и входных данных, которые использовались для вычисления неправильного значения переменной, начиная с известных значений состояний, параметров и времени.

Переменная с ошибочным значением (или которая вообще не может быть вычислена) отображается в корне графа.

Ребра могут быть следующих двух типов.

- Ребра зависимости данных: направленные ребра, помеченные переменными или параметрами, которые являются входными данными (используются для вычислений в этом уравнении) или выходными данными (вычисляются из этого уравнения) уравнения, отображаемого в узле.

- Исходные ребра: неориентированные ребра, которые связывают узел уравнения с реальной моделью, к которой принадлежит это уравнение. ребра, указывающие из сгенерированного исполняемого кода моделирования на исходные уравнения или части уравнений, участвующие в этом коде

Пользователь может:

- Отобразить результаты симуляции, выбрав имя переменной или параметра(названия ребер). График переменной покажется в дополнительном окне. Пользователь может быстро увидеть, имеет ли переменная ошибочное значение.

- Отобразить код модели следуя по исходным ребрам.

- Вызвать подсистему отладки алгоритмического кода, если пользователь подозревает, что результат переменной, вычисленной в уравнении, содержащем вызов функции, неверен, но уравнение кажется правильным.

Используя эти средства интерактивного графа зависимостей, пользователь может проследить за ошибкой от ее проявления до ее источника.

Систематические методы отладки для масштабных вычислительных фреймворков высокопроизводительных вычислений

Далее представлен материал, являющийся результатом анализа работы [3].

Параллельные вычислительные фреймворки для высокопроизводительных вычислений играют центральную роль в развитии исследований, основанных на моделировании, в науке и технике.

Фреймворк Uintah был создан для решения сложных задач взаимодействия жидких структур с использованием параллельных вычислительных систем.

Проблема Поиск и исправление ошибок в параллельных фреймворках, возникающих из-за параллельного характера кода.

Предложенное решение В статье описывается подход к отладке крупномасштабных параллельных систем, основанный на различиях в исполнении между рабочими и нерабочими версиями. Подход основывается на трассировке стека вызовов.

Исследование проводилось для вычислительной платформы Uintah Computational Framework.

Описание решения Так как количество трассировок стека, которые можно получить при выполнении программы, может быть большим, для лучшего понимания используются графы, которые могут сжать несколько миллионов трассировок стека в одну управляемую фигуру. Метод основан на получении объединенных графов трассировки стека (CSTG).

Хотя сбор и анализ трассировки стека ранее изучались в контексте инструментов и подходов, их внимание не было сосредоточено на кросс-версии (дельта) отладке.

```
void A()                {
       cstg.addStackTrace();
}
void B()                {

     A();
}
int   main() {
    int     x = random();
    if       (x > 0)  B();
    A();
}
```

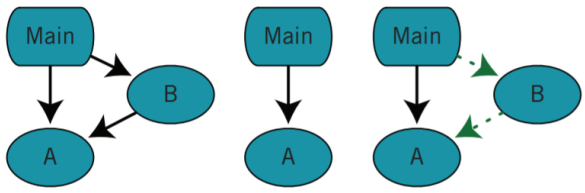


Рис. 3. Пример построения CSTG

CSTG не записывают каждую активацию функции, а только те, что в трассировках стека ведут к интересующей функции (функциям), выбранной пользователем. Каждый узел CSTG представляет все активации конкретного вызова функции. Помимо имен функций, узлы CSTG также помечаются уникальными идентификаторами вызова. Грани представляют собой вызовы между функциями.

Пользователю необходимо вставить в интересующие функции вызовы cstg.addStackTrace(). Инструмент CSTG автоматически запускает тестируемый пример с использованием различных сценариев, создает графы и помогает пользователям увидеть существенные различия между сценариями. Сама ошибка обычно обнаруживается и подтверждается с помощью традиционного отладчика, при этом дельта-группы CSTG наводят на место возникновения ошибки.

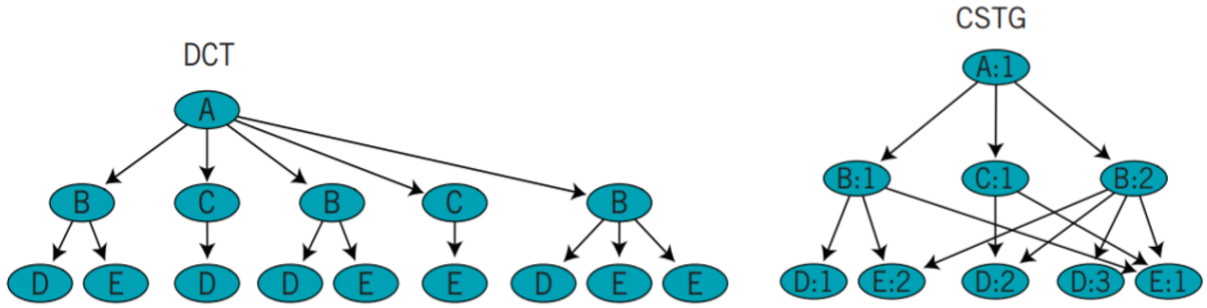


Рис. 4. CSTG

Ориентированный на данные фреймворк для отладки параллельных приложений

Далее представлен материал, являющийся результатом анализа работы [4].

Проблема Обнаружение ошибок в крупномасштабных научных приложениях, работающих на сотнях тысяч вычислительных ядер.

Неэффективность параллельных отладчиков для отладки пета-масштабных приложений.

Предложенное решение В этом исследовании представлена реализация фреймворка для отладки, ориентированного на данные, в котором в качестве основы используются утверждения. Подход, ориентированный на данные можно использовать для повышения производительности параллельных отладчиков.

Метод, представленный в статье основан на параллельном отладчике Guard, который поддерживает тип утверждения во время отладки, называемые сравнительные утверждения.

Описание решения Утверждение – это утверждение о предполагаемом поведении компонента системы, которое должно быть проверено во время выполнения. В программировании программист определяет утверждение, чтобы гарантировать определенное состояние программы во время выполнения.

Пользователь делает заявления о содержимом структур данных, затем отладчик проверяет достоверность этих утверждений.

В следующем списке показаны примеры утверждений, которые можно использовать для обнаружения ошибок во время выполнения:

- «Содержимое этого массива всегда должно быть положительным».
- «Сумма содержимого этого массива всегда должна быть меньше постоянной границы».

- «Значение в этом скаляре всегда должно быть больше, чем значение в другом скаляре».
- «Содержимое этого массива всегда должно быть таким же, как содержимое другого массива».

В работе используются два новых шаблона утверждений помимо сравнительных: общие специальные утверждения и статистические утверждения.

Общие специальные утверждения позволяют использовать простые арифметические и булевы операции.

```
for (j = 0; j < n; j++) {
  for (i = 0; i < m; i++) {
    pold[j][i] = 50000.;
    pressure[j][i] = 50000.; }}
```

Например, учитывая следующий фрагмент кода из исходного файла init.c и предполагая, что код вызывается с набором процессов $a, можно рассмотреть следующее утверждение:

```
assert $a::pressure@"init.c":180 = 50000
```

Это утверждение гарантирует, что каждый элемент в структуре данных набора процессов $a равен 50 000 в строке 180 исходного файла init.c

Соответственно, *сравнительные утверждения* позволяют сравнивать две отдельные структуры данных во время выполнения.

Следующее утверждение сравнивает данные из big_var в $a в строке 4300 исходного файла ref.c с large_var в $b в строке 4300 исходного файла sus.c.

```
assert $a::big_var@"ref.c":4300=$b::large_var@"sus.c":4300
```

Статистические утверждения - это определяемый пользователем предикаты, состоящие из двух моделей данных в форме либо статистических примитивов (средние значения, значения стандартного отклонения), либо функциональных моделей (гистограммы, функции плотности)

Статистические утверждения позволяют пользователю сравнивать информацию о шаблонах данных между двумя структурами данных, тогда как более ранние утверждения требовали сравнения точных значений.

Например, можно утверждать, что среднее значение большого набора данных находится между определенными границами до или после вызова функции или что количество элементов в массиве должно находиться в определенном диапазоне.

Определение степени и источников недетерминизма в приложениях MPI с помощью ядер графов

Далее представлен материал, являющийся результатом анализа работы [5].

Проблема Выявление причин сбоев воспроизводимости в приложениях на экзафлопсных платформах.

Крупномасштабные приложения MPI обычно принимают гибкие решения во время выполнения том порядке, в котором процессы обмениваются данными, чтобы улучшить свою производительность. Следовательно, недетерминированные коммуникативные модели стали особенностью этих научных приложений в системах высокопроизводительных вычислений.

Недетерминизм мешает разработчикам отслеживать вариации выполнения программы для отладки. Также сложно воспроизвести результаты при повторных запусках, что затрудняет доверие к научным результатам.

Предложенное решение Фреймворк для определения источников недетерминизма с использованием графов событий.

Описание решения Параллельное выполнение программы моделируется в виде ориентированных графов событий, ядра графов используются, чтобы охарактеризовать изменения межпроцессного взаимодействия от запуска к запуску. Ядра могут количественно определять тип и степень недетерминизма, присутствующего в моделях коммуникации MPI.

Фреймворк позволяет найти первопричины недетерминизма в исходном коде без знания коммуникативных паттернов приложения.

Платформа моделирует недетерминизм в следующие три этапа.

Этап первый. Сбор трассировки выполнения. Фреймворк фиксирует трассировку нескольких выполнений недетерминированного приложения с помощью двух модулей трассировки: CSMPI (фиксирует стек вызововов, связанных с вызовами функций MPI) и DUMPI (фиксирует порядок отправляемых и получаемых сообщений для каждого процесса MPI).

Для каждого выполнения приложения фреймворк генерирует один файл трассировки CSMPI и один файл трассировки DUMPI для каждого процесса MPI (или ранга). Файлы трассировки впоследствии загружаются конструктором графа событий для восстановления порядка сообщений выполнения.

Этап второй. Построение модели графа событий. На втором этапе фреймворк моделирует выполнение недетерминированного приложения в виде ориентированного ациклического графа (DAG), используя файлы трассировки, созданные на первом этапе.

Здесь вершины представляют собой связи point-to-point, такие как отправка и получение сообщения, а направленные ребра представляют отношения между этими событиями. Модели межпроцессного взаимодействия этой формы обычно называются графами событий.

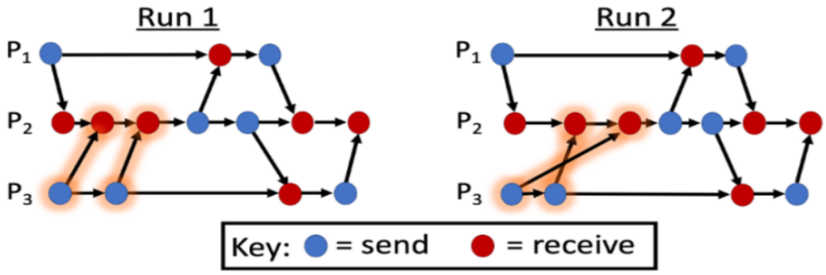


Рис. 5. DGA

Затем используются ядра графов для количественной оценки (несходства) графов событий, тем самым количественно оценивая степень проявления недетерминированности в приложении.

Ядра графов представляют собой семейство методов для измерения структурного сходства графов.

Простыми словами, ядро графа можно рассматривать как функцию, которая подсчитывает совпадающие подструктуры (например, поддеревья) двух входных графов, как показано на рис. 6, сопоставляя пары графов со скалярами, которые количественно определяют, насколько они похожи.

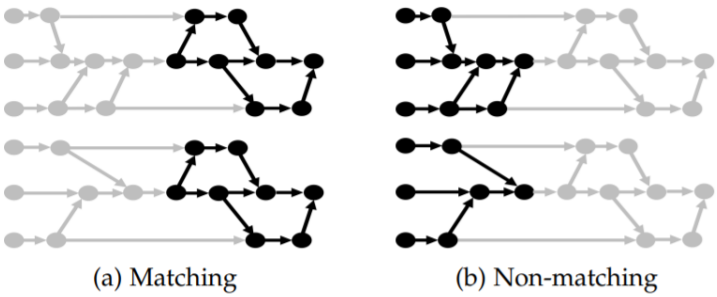


Рис. 6. Сравнение двух графов

Этап третий. Анализ графа событий. На последнем этапе анализ графа событий позволяет ученым количественно оценить недетерминированность многократного выполнения приложения MPI без каких-либо знаний о коммуникативных моделях приложения MPI.

Подготовлено: *Крехтунова Д.Д. (РК6-73Б), 2021.11.10*

2 Разработка web-ориентированного редактора графовых моделей

2021.10.05: Обзор языка описания графов DOT

Язык описания графов DOT предоставляется пакетом утилит Graphviz (Graph Visualization Software). Пакет состоит из набора утилит командной строки и программ с графическим интерфейсом, способных обрабатывать файлы на языке DOT, а также из виджетов и библиотек, облегчающих создание графов и программ для построения графов. Более подробно будет рассмотрена утилита dot.

dot - инструмент для создания многоуровневого графа с возможностью вывода изображения полученного графа в различных форматах (PNG, PDF, PostScript, SVG и др.).

Установка graphviz

Linux: sudo apt install graphviz

MacOS: brew install graphviz

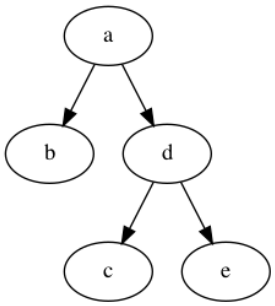
Вызов всех программ Graphviz осуществляется через командную строку, в процессе ознакомления с языком использовалась следующая команда

dot −Tpng <pathToDotFile> −o <imageName>

В результате выполнения этой команды будет создано изображение графа в формате png

Пример описания простого графа

```
digraph G {
    a -> b;
    a -> d -> c;
    d -> e;
}
```



Более подробная информация с примерами представлена в обзоре литературы, который находится по следующему пути:
01 - Курсовые проекты/2021-2022 - Разработка web-ориентированного редактора графовых моделей /0 - Обзор литературы/

Подготовлено: *Ершов В. (РК6-72Б), 2021.10.05*

3 Реализация поддержки различных стратегий распараллеливания в рамках графоориентированного программного каркаса

2021.11.14: Первичный обзор литературы

Известные алгоритмы поиска циклов в ориентированных графах

Существует несколько групп алгоритмов поиска[6]:

1) алгоритмы, основанные на обходе ориентированного графа (ОГ);
2) алгоритмы, основанные на использовании матриц смежности, описывающих конкретный граф.

Одним из представителей алгоритмов первой группы (обхода ОГ) является алгоритм поиска в глубину (англ., depth-first-search (DFS)), который предполагает, что обход осуществляется из фиксированного и заранее заданного узла. Если число узлов ОГ равно n, то сложность алгоритма DFS $O(n^2)$ и $O(n^3)$ – для случая "многократного обхода" из каждого узла[1]. Поэтому для ускорения алгоритма необходимо применение модификаций, например распараллеливание алгоритма поиска в глубину [7].

Зададим граф $G(V, E)$, где $V = \{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \ldots\}$ – множество вершин, $|V| = n$, а $E = \{\alpha\beta : \alpha\beta \in V \times V\}$ – множество рёбер (пример, $\{\boldsymbol{ab}, \boldsymbol{ad}, \ldots, \boldsymbol{ge}\}$).

Для определения ОГ, часто, применяют понятие матриц смежности[**?**]:

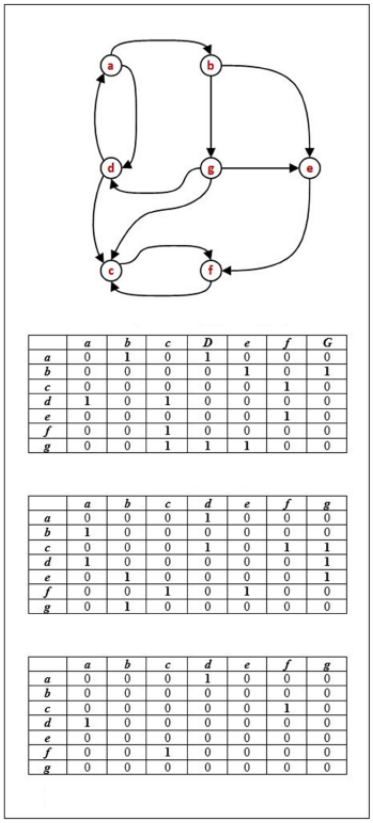
$$\begin{aligned} &\|A\| = \|a_{ij}\|_{n\times n};\\ &a_{ij} = \begin{cases} 1, & \text{– определено ребро с началом в узле } i \text{ и концом в узле } j;\\ 0, & \text{– ребро не определено.} \end{cases} \end{aligned} \tag{1}$$

Пример некоторого графа и ему соответствующая матрица смежности приведены на рис. 7.

$$A = \begin{bmatrix} & \boldsymbol{a} & \boldsymbol{b} & \boldsymbol{c} & \boldsymbol{d} & \boldsymbol{e} & \boldsymbol{f} & \boldsymbol{g} \\ \boldsymbol{a} & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ \boldsymbol{b} & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ \boldsymbol{c} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \boldsymbol{d} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \boldsymbol{e} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \boldsymbol{f} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \boldsymbol{g} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2}$$

Элемент матрицы смежности a_{ij} указывает на факт наличия определённого рёбра с началом в узле i и концом в узле j, тогда как транспонированная матрица A^T задаёт

[1] Для случая ОГ возможность идентификации всех циклов требует осуществления многократных обходов, начиная с каждого из узлов ОГ.



	a	b	c	D	e	f	G
a	0	1	0	1	0	0	0
b	0	0	0	0	1	0	1
c	0	0	0	0	0	1	0
d	1	0	1	0	0	0	0
e	0	0	0	0	0	1	0
f	0	0	1	0	0	0	0
g	0	0	1	1	1	0	0

	a	b	c	d	e	f	g
a	0	0	0	1	0	0	0
b	1	0	0	0	0	0	0
c	0	0	0	1	0	1	1
d	1	0	0	0	0	0	1
e	0	1	0	0	0	0	1
f	0	0	1	0	1	0	0
g	0	1	0	0	0	0	0

	a	b	c	d	e	f	g
a	0	0	0	1	0	0	0
b	0	0	0	0	0	0	0
c	0	0	0	0	0	1	0
d	1	0	0	0	0	0	0
e	0	0	0	0	0	0	0
f	0	0	1	0	0	0	0
g	0	0	0	0	0	0	0

Рис. 7. Граф и его матрица смежности

новый ОГ с теми же узлами и рёбрами, противоположно направленными относительно исходного ОГ.

Введём обозначение матрицы, возведённой в степень m:

$$A^m = \underbrace{A \times A \times \ldots \times A}_{m}. \tag{3}$$

Матрица A^m позволяет определить .... – это число соединений узлов состоящих ровно из m узлов [6]. Для того, чтобы определить циклы порядка $2m$, нужно найти матрицу циклов C_m, определяемую следующим образом:

$$C_m = A^m \wedge (A^T)^m; \tag{4}$$

где $\wedge$ – оператор конъюнкции, применяемый поэлементно.

Таким образом, матрица C_m будет содержать все циклы порядка $2m$. Для поиска циклов порядка $[2, 2n]$ необходимо повторить операцию n раз.

Пересчет циклов в графах в т.ч. при помощи GPU

Замечание 1. Пересчет циклов в графе является NP-полной задачей[7], следовательно не может быть выполнен за полиномиальное время.

В статье [7] представлен подход, позволяющий ускорить процедуру пересчета циклов с использованием GPU, описываемый алгоритмом 1.

Algorithm 1 Алгоритм Thread-Based Cycle Detection поиска циклов в ориентированном графе

1: Создаем массив узлов графа V.
2: Заполняем новый массив V' вершинами со степенью больше чем 2 – это исключит узлы, которые не могут иметь цикла.
3: Для каждого элемента массива V параллельно создаём комбинацию для проверки.
4: Создаём матрицу смежности C для созданной комбинации и проверяем её на наличие циклов.
5: Меняем комбинации перестановками.

e

Подготовлено: *Муха В. (РК6-73Б), 2021.11.14*

4 Реализация графоориентированной технологии определения бизнес-логики работы пользователя в системе

Список литературы

[1] Frank Eichinger, Klemens Bohm, and Matthias Huber. Mining Edge-Weighted Call Graphs to Localise Software Bugs // Lecture Notes in Computer Science. 2008. P. 333–348.

[2] A. Pop, M. Sjolund, A. Asghar, P. Fritzson, F. Casella. Integrated Debugging of Modelica Models // Modeling, Identification and Control. 2014. Vol. 35, no. 2. P. 93–107.

[3] Alan Humphrey, Qingyu Meng, Martin Berzins, Diego Caminha B. de Oliveira, Zvonimir Rakamaric, Ganesh Gopalakrishnan. Systematic debugging methods for large-scale hpc computational frameworks // Computing in Science and Engineering. 2014. Vol. 16, no. 3. P. 48–56.

[4] Minh Ngoc Dinh, David Abramso1, Chao Jin, Andrew Gontarek, Bob Moench, Luiz DeRose. A data-centric framework for debugging highly parallel applications // Software - Practice and Experience. 2013. Vol. 45, no. 4. P. 501–526.

[5] Dylan Chapp, Nigel Tan, Sanjukta Bhowmick, Michela Taufer. Identifying Degree and Sources of Non-Determinism in MPI Applications via Graph Kernels // IEEE Transactions on Parallel and Distributed Systems. 2021. Vol. 32, no. 12. P. 2936–2952.

[6] Davidrajuh R. Detecting Existence of Cycles in Petri Nets // Department of Electrical and Computer Engineering. 2016. Vol. 10, no. 12. P. 2936–2951.

[7] Fahad Mahdi, Maytham Safar, Khaled Mahdi. Detecting Cycles in Graphs Using Parallel Capabilities of GPU // Computer Engineering Department. 2011. Vol. 13, no. 12. P. 2936–2952.