ТЕОРЕТИЧЕСКИЙ И ПРИКЛАДНОЙ НАУЧНО-ТЕХНИЧЕСКИЙ ЖУРНАЛ

Издается с сентября 2010 г. DOI 10.17587/issn.2220-3397 ISSN 2220-3397

Журнал издается при поддержке Отделения математических наук РАН, Отделения нанотехнологий и информационных технологий РАН, МГУ имени М.В. Ломоносова, МГТУ имени Н.Э. Баумана

СОДЕРЖАНИЕ

Журнал зарегистрирован в Федеральной службе по надзору в сфере связи, информационных технологий и массовых коммуникаций. Свидетельство о регистрации ПИ № ФС77-38590 от 24 декабря 2009 г.

Журнал распространяется по подписке, которую можно оформить в любом почтовом отделении (индекс по Объединенному каталогу "Пресса России" — **22765**) или непосредственно в редакции.
Тел.: (499) 269-53-97. Факс: (499) 269-55-10.
Http://novtex.ru/prin/rus E-mail: prin@novtex.ru
Журнал включен в систему Российского индекса научного цитирования и базу данных RSCI на платформе Web of Science.
Журнал входит в Перечень научных журналов, в которых по рекомендации ВАК РФ должны быть опубликованы научные результаты диссертаций на соискание ученой степени доктора и кандидата наук.

THEORETICAL AND APPLIED SCIENTIFIC AND TECHNICAL JOURNAL

SOFTWARE ENGINEERING

PROGRAMMNAYA INGENERIA

Vol. 10
N 6
2019

Published since September 2010 **DOI 10.17587/issn.2220-3397** **ISSN 2220-3397**

CONTENTS

Information about the jornal is available online at:
http://novtex.ru/prin/eng e-mail:prin@novtex.ru

УДК 004.023 DOI: 10.17587/prin.10.243-256

В. А. Галатенко, д-р физ.-мат. наук, зав. сектором, galat@niisi.ras.ru,
Н. И. Вьюкова, ст. науч. сотр., e-mail: niva@niisi.ras.ru,
К. А. Костюхин, канд. физ.-мат. наук, ст. науч. сотр., kost@niisi.ras.ru,
Федеральный научный центр Научно-исследовательский институт системных исследований Российской академии наук (ФНЦ НИИСИ РАН), Москва

Схемы программ как инструмент распараллеливания. Механизмы применения*

Рассмотрены традиционные способы и механизмы применения схем программ. Также рассмотрены проблемные вопросы, которые возникают при использовании схем для разработки программ, для распараллеливания унаследованных последовательных программ, для оценки производительности аппаратно-программных систем, поддерживающих параллелизм, для разработки аппаратуры, включающей специализированные ускорители, а также для поддержки учебных курсов по параллельному программированию.

Ключевые слова: *схемы программ, параллельное программирование, распределенные системы, обзор, гетерогенные системы, многоядерные системы, методы распараллеливания*

Введение

Схемы программ — универсальное средство накопления и использования программистских знаний, особенно важное и полезное в сложных предметных областях, каковым является параллельное программирование.

Применение распараллеливаемых схем позволяет:

- поддержать баланс между производительностью аппаратно-программных систем и продуктивностью программистов, сделать разработку параллельного программного обеспечения экономически целесообразной;
- сделать параллельные программы мобильными, облегчить сопровождение и перенос параллельного программного обеспечения на новые аппаратные платформы.

В настоящей работе сделана попытка систематического, многостороннего подхода к схемам программ как инструментарию распараллеливания. Это является наиболее важным результатом, представленным в данной работе.

* Публикация выполнена в рамках государственного задания по проведению фундаментальных научных исследований по теме (проекту) "38. Проблемы создания глобальных и интегрированных информационно-телекоммуникационных систем и сетей, развитие технологий и стандартов GRID. Исследование и реализация программной платформы для перспективных многоядерных процессоров (0065-2019-0002)".

Применение схем для распараллеливания унаследованных последовательных программ

Схемы программ не только помогают писать параллельные программы "с нуля", но и позволяют распараллеливать унаследованные последовательные программы. Идея состоит в том, чтобы находить в этих последних программах определенные шаблоны и (полу)автоматически подбирать для них параллельные схемы. На этой идее основана работа [1].

Важным вопросом является представление последовательной программы, в котором отыскиваются шаблоны для распараллеливания. Чаще всего в этом качестве выступают UML-диаграммы. В работе [1] пошли по другому пути. Там введено понятие элемента обработки данных (*computational unit*, CU) — фрагмента кода, следующего шаблону "чтение — обработка — запись". Программа представляется в виде графа зависимостей, называемого CU-графом, вершинами в котором служат элементы обработки данных, а ребра отражают истинные зависимости по данным между элементами, т. е. показывают, могут ли соответствующие элементы выполняться параллельно. CU-граф представляется в виде матрицы смежности.

На листинге 1 приведен пример фрагмента программы, который порождает CU-граф из трех узлов: `INIT` (особый элемент, в котором присваиваются на-

чальные значения); CUx; CUy. Ребра ведут из CUx и CUy в INIT.

```
// INIT узел графа
x = 3
y = 4
// CUx
a = x + rand () / x
b — x — rand () / x
x = a + b
// CUy
a = y + rand () / y
b — y — rand () / y
y = a + b
```

Листинг 1. Фрагмент программы, по которому строится CU-граф

Для построения CU-графа применяют гибридный (т. е. сочетающий статический и динамический) анализ, отправной точкой которого служит внутреннее представление программы в системе LLVM. Помимо CU-графа, строится также граф передачи управления (рассматриваются только вызовы функций и циклы). Эти два графа в совокупности образуют прошитое дерево выполнения. Построенное дерево зависит от исходных данных. Чтобы уменьшить зависимость, программа запускается несколько раз с различными исходными данными. Результаты запусков объединяются.

Распараллеливаемые шаблоны отыскиваются в дереве выполнения. Анализируется не вся программа, а только "горячие точки", на выполнение которых уходит основная часть времени.

В работе [1] рассмотрен конвейерный параллелизм, а также распараллеливание циклов с независимыми итерациями (do-all).

Рассмотрим, как выявляется конвейерный параллелизм. На листинге 2 представлен цикл обработки последовательности кадров. Его тело состоит из трех элементов обработки данных: CUu вводит (изменяет) текущий кадр, CUe обрабатывает его, CUo выводит результат. Здесь CUe зависит по данным от CUu и CUo, а CUo — от CUu, CUe и себя. Матрица смежности этого CU-графа показана на рис. 1.

```
for (i = 0; i < num _ of _ frames; ++i) {
   // CUu
   if (!pf.Update (i))
     return (0);
   // CUe
   pf.Estimate (estimate);
   // CUo
   WritePose (output, estimate);
   if (outputBMP)
     outputBMP (estimate);
}
```

Листинг 2. Основной цикл отслеживания позы

Конвейер может иметь произвольную длину, поэтому и шаблон для него имеет переменную длину.

	CUu	CUe	CUo
CUu	0	0	0
CUe	1	0	1
CUo	1	1	1

Рис. 1. Матрица смежности CU-графа тела цикла

	A	B	C
A	x	w(1,2)	0
B	1	x	w(2,3)
C	x	1	x

Рис. 2. Матрица смежности шаблона конвейера (матрица конвейера)

Поскольку в данном случае матрица смежности для анализируемого фрагмента программы имеет размер 3×3, шаблон берется того же размера. Он показан на рис. 2 и называется матрицей конвейера.

Строки и столбцы матрицы конвейера представляют стадии конвейера, элементы матрицы — зависимости между стадиями. Элементам матрицы конвейера придается следующий смысл.

- Элемент x обозначает "ни на что не влияет"; эта связь и, соответственно, стоящее здесь значение (0 или 1) не влияет на создание конвейера.
- Элемент 1 характеризует обязательную зависимость. Подобные элементы в совокупности представляют цепочку зависимостей стадий конвейера и называются цепочечными зависимостями.

Элемент w (j, k) указывает на зависимость вперед в конвейере и характеризует вес подобной зависимости. Зависимость вперед существует, если стадия Sj конвейера на итерации i цикла зависит от результата стадии Sk на предыдущей итерации (i – 1) при j < k. Зависимости вперед тормозят работу конвейера, поскольку более ранняя стадия должна ждать результата более поздней стадии, получаемого на предыдущей итерации цикла. Вес рассчитывается по формуле w (j, k) = 1 – (k – j)/(N – 1), где N — общее число стадий конвейера. Вес уменьшается, когда расстояние между двумя стадиями с зависимостью вперед увеличивается, т. е. эффективность функционирования конвейера уменьшается.

- Элемент 0 в последнем столбце первой строки гарантирует отсутствие зависимости первой стадии конвейера от последней и является обязательным, поскольку подобная зависимость по сути делает конвейер последовательным.

По матрицам графа и конвейера создаются вектор графа и вектор конвейера соответственно. Для этого используются цепочечные зависимости и зависимости вперед. Все элементы векторов, кроме последнего, берутся по порядку из позиций цепочечных зависимостей. Последний элемент получает значение 0, если в матрице графа нет зависимостей вперед. Если такие зависимости существуют, последний элемент вектора графа устанавливается равным 1, а для

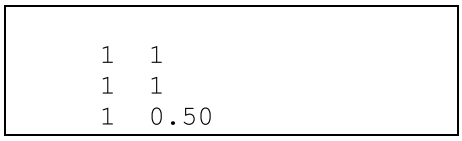

Рис. 3. Вычисленные векторы-столбцы графа и конвейера

вектора конвейера он полагается равным минимуму среди весов ссылок вперед в матрице конвейера, для которых соответствующие элементы матрицы графа имеют ненулевые значения. На рис. 3 показаны векторы графа и конвейера для рассматриваемого примера.

Коэффициент корреляции между этими векторами показывает, можно ли конвейеризовать цикл. В частности, чем меньше последний элемент вектора конвейера, тем меньше коэффициент корреляции. В рассматриваемом примере коэффициент корреляции оказывается равным 0,96. Это означает, что конвейер организовать можно, но, поскольку это значение меньше 1, имеются зависимости вперед, которые необходимо разрешить. Для этого можно, например, слить все стадии, вовлеченные в зависимости вперед.

Дополнительный источник повышения эффективности конвейера — параллельная работа экземпляров стадий конвейера, относящихся к разным итерациям цикла. Подобное распараллеливание возможно, если элемент обработки данных не зависит от себя, т. е. соответствующий диагональный элемент матрицы графа равен 0.

Полное распараллеливание итераций цикла, т. е. применение схемы "do-all", возможно, если в цикле нет зависимостей вперед и зависимостей от себя, т. е. весь верхний треугольник матрицы графа для тела цикла (главная диагональ и выше) состоит из нулей.

Приведенные в работе [1] результаты экспериментов показывают работоспособность описанного подхода. Авторы планируют пополнить набор рассматриваемых схем, включив в него схемы мастер/работник, map/reduce и т. д.

В то же время отметим, что в работе [1] предложено не автоматическое преобразование последовательного кода в параллельный, а выдача рекомендаций, где и какой параллелизм искать и реализовывать. За корректность и эффективность преобразования программы отвечает программист.

Авторы работы [2] поставили перед собой цель автоматического распараллеливания унаследованных (C/C++)-программ. В качестве первого шага они реализовали выявление и аннотирование конвейерного параллелизма в циклах. В работе [2] описан инструментальный механизм анализа схем распараллеливания PPAT (*Parallel Pattern Analyzer Tool*), реализация которого основана на компиляторной инфраструктуре LLVM и библиотеках статического анализатора Clang.

Анализ возможности конвейеризации цикла начинается с выявления стадий потенциального конвейера. Новая стадия открывается всякий раз, когда в теле цикла встречается вызов функции или вложенный цикл. Затем проверяется выполнение нескольких необходимых условий:

- отсутствие модификации глобальных переменных;
- отсутствие обратных связей между итерациями цикла;
- наличие по крайней мере двух стадий в конвейере.

Циклы, которые были отвергнуты в результате анализа, снабжаются комментариями, поясняющими причины отказа. Циклы, успешно прошедшие проверки, аннотируются с использованием следующих атрибутов:

- `rpr::pipeline`: идентифицирует конвейеризуемый цикл;
- `rpr::stream`: идентифицирует поток данных, обрабатываемый конвейером;
- `rpr::stage`: идентифицирует фрагмент исходного кода как стадию конвейера;
- `rpr::plid`: включает аргумент, указывающий, какому конвейеру принадлежит стадия;
- `rpr::in`: указывает входные переменные, передаваемые стадии конвейера;
- `rpr::out`: указывает выходные переменные, передаваемые дальше стадией конвейера.

На листинге 3 приведен пример аннотированного кода.

```
[[ rpr::pipeline (0), rpr::stream (density, density _ energy, velocity, speed _ sqd, pressure) ]]
for (int i = 0; i < nelr; i ++)
{
   float density, density _ energy, speed _ sqd, pressure, speed _ of _ sound;
   float3 velocity, momentum;
   [[ rpr::stage (0), rpr::plid (0), rpr::out (density, density _ energy, velocity) ]]
   {
      density = variables [NVAR * i + VAR _ DENSITY];
      momentum.x = variables [NVAR * i + (VAR _ MOMENTUM + 0)];
      momentum.y = variables [NVAR * i + (VAR _ MOMENTUM + 1)];
      momentum.z = variables [NVAR * i + (VAR _ MOMENTUM + 2)];
      density _ energy = variables [NVAR * i + VAR _ DENSITY _ ENERGY];
      compute _ velocity (density, momentum, velocity);
   }
   [[ rpr::stage (1), rpr::plid (0), rpr::in (velocity), rpr::out (speed _ sqd) ]]
   speed _ sqd = compute _ speed _ sqd (velocity);
```

```
   [[ rpr::stage (2), rpr::plid (0), rpr::in (density, density _ energy, speed _ sqd), rpr::out
(pressure) ]]
   pressure = compute _ pressure (density, density _ energy, speed _ sqd);

   [[ rpr::stage (3), rpr::plid (0), rpr::in (density, pressure), rpr::out (speed _ of _ sound) ]]
   speed _ of _ sound = compute _ speed _ of _ sound (density, pressure);

   [[ rpr::stage (4), rpr::plid (0), rpr::in (speed _ sqd, speed _ of _ sound) ]]
   step _ factors [i] = float (0.5f) / (std::sqrt (areas [i]) * (std::sqrt (speed _ sqd) +
speed _ of _ sound));
}
```

Листинг 3. Пример цикла, допускающего конвейеризацию, с аннотациями

На листинге 4 показан цикл, который не удается конвейеризовать, с комментариями, поясняющими причины неудачи.

```
// It isn't a Pipeline
// Feedback: color
// Feedback: color
// Feedback: h _ cost
while (!wavefront.empty () ) {
   index = wavefront.front ();
   wavefront.pop _ front ();
   // It isn't a Pipeline
   // Feedback: color
   // Feedback: h _ cost
   // Less than 2 stages
   for (int i = h _ graph _ nodes [index].x;
     i <(h _ graph _ nodes [index].y +
     h _ graph _ nodes [index].x); i ++)
     {
     int id = h _ graph _ edges [i].x;
     if (color [id] = = WHITE) {
       h _ cost [id] = h _ cost [index] + 1;
       wavefront.push _ back (id);
       color [id] = GRAY;
     }
   }
   color [index] = BLACK;
}
```

Листинг 4. Пример цикла, не допускающего конвейеризацию, с комментариями

Проведенные авторами настоящей работы эксперименты показали, что для больших программ PPAT выявляет конвейеры лучше человека-программиста. Модульная организация PPAT позволяет добавлять выявление других видов параллелизма и главное — реализовать автоматическое преобразование исходных текстов.

Схемы программ в контексте оценки эффективности распараллеливания

Распараллеливающие схемы программ преследуют цель установить баланс между производительностью компьютерных систем (эффективностью) и продуктивностью программистов (программируемостью), сделать таким образом разработку и выполнение параллельных систем экономически оправданными. Эффективность измеряется временем выполнения программ, программируемость — общим числом строк исходного текста или числом строк, которые потребовалось добавить и/или модифицировать при переходе от последовательной версии программы к параллельной.

Набор тестов производительности параллельных систем из разных предметных областей PARSEC (*Princeton Application Repository for Shared-Memory Computers*) [3] представлен как проект с открытыми исходными текстами [4]. Параллелизм в PARSEC выражен посредством низкоуровневого средства — POSIX-потоков управления (POSIX *threads*, *Pthreads*). Первоначально набор PARSEC использовался для оценки эффективности аппаратных архитектур с разделяемой памятью. Затем его стали применять для оценки выразительной силы инструментальных средств параллельного программирования.

Тесты производительности из набора PARSEC разнообразны в смысле дисциплины доступа к памяти и разделения данных, возможной степени параллелизма, частоты синхронизации и т. п. Они позволяют оценить параллельные системы с разных сторон. В работе [5] сделана попытка использовать PARSEC для оценки распараллеливающих схем программ, их эффективности и программируемости. Для этого был разработан набор тестов производительности P3ARSEC (*Parallel Patterns* PARSEC), также представленный как проект с открытыми исходными текстами [6].

Тексты PARSEC-программ были переписаны с использованием высокоуровневых операций (распараллеливающих схем программ), подготовлены к исполнению с помощью различных инструментальных средств и выполнены на многоядерных аппаратных конфигурациях.

Тест производительности Ferret (поиск подходящих аудио- и видеофрагментов) в P3ARSEC реализован в рамках системы FastFlow [7]. Здесь выявлен и использован конвейерный параллелизм (листинг 5). Построен конвейер из шести стадий, начальная и конечная стадии последовательные, экземпляры других четырех стадий могут выполняться параллельно (к ним применяется утилита `make _ Farm`, создающая `n` экземпляров).

```
// Первая стадия конвейера
struct Load: ff _ node _ t <long, load _ data> {
   load _ data *svc (long*) { <содержательный код> };
} In;
// Вторая стадия
struct Segment: ff _ node _ t <load _ data, seg _ data> {
   seg _ data *svc (load _ data *in) { <содержательный код> };
};
// Третья стадия
struct Extract: ff _ node _ t <seg _ data, extr _ data> {
   extr _ data *svc (seg _ data *in) { <содержательный код> };
};
// Четвертая стадия
struct Index: ff _ node _ t <extr _ data, vec _ query _ data> {
   vec _ query _ data *svc (extr _ data *in) { <содержательный код> };
};
// Пятая стадия
struct Rank: ff _ node _ t <vec _ query _ data, rank _ data> {
   rank _ data *svc (vec _ query _ data *in) { <содержательный код> };
};
// Шестая стадия
struct Output: ff _ node _ t <rank _ data> {
   void *svc (rank _ data *in) { <содержательный код> };
} Out;
// Создание конвейера с коллективом работников
ff _ Pipe <> pipe (In,
                   make _ Farm <Segment, n> (),
                   make _ Farm <Extract, n> (),
                   make _ Farm <Index, n> (),
                   make _ Farm <Rank, n> (), Out);
// Запуск конвейера
pipe.run _ and _ wait _ end ();
```

Листинг 5. FastFlow-реализация теста производительности Ferret

Еще один тест производительности, Swaptions, демонстрирует параллелизм по данным и реализован в среде SkePU [8] посредством схемы `map` (листинг 6).

```
// map-функция, работающая с отдельным элементом входной коллекции
MapOutput mapFunction (skepu2::Index1D index, parm elem) { <содержательный код> };

// Подготовка входной и выходной структур данных
skepu2::Vector <parm> swaptions _ sk (swaptions, nSwaptions, false);
skepu2::Vector <MapOutput> output _ sk nSwaptions);
// Создание map-объекта путем предоставления функции для вычисления
auto map = skepu2::Map <> (mapFunction);
// Установка OpenMP-сервера и числа используемых потоков управления
auto spec = skepu2::BackendSpec (skepu2::Backend::Type::OpenMP);
spec.setCPUThreads (nThreads);
map.setBackend (spec);
// Вызов выполнения map
map (output _ sk, swaptions _ sk);
```

Листинг 6. SkePU-реализация теста производительности Swaptions

По сравнению с Pthreads-реализацией в PARSEC, в P3ARSEC удалось в среднем сократить число строк исходных текстов на 26 % (при использовании как FastFlow, так и SkePU). Лучший результат — сокращение числа строк на 87 %. Благодаря использованию схем программ переход от последовательной версии к параллельной также существенно упростился по сравнению с PARSEC — в среднем на 58 %. Иными словами, схемы программ существенно повысили программируемость.

Что касается эффективности, то FastFlow-реализации дали (по сравнению с Pthreads) средний выигрыш 14 % (максимальный выигрыш составил 42 %, максимальный проигрыш — 9 %). При использовании инструментальной среды SkePU соответствующие показатели оказались равны 7, 45 и 11 %). Тем самым было подтверждено, что благодаря распараллеливающим схемам программ улучшения программируемости удается достичь без ущерба для эффективности. На самом деле эффективность даже возрастает, поскольку у разработчика появляется возможность быстрого прототипирования различных вариантов распараллеливания и выбора среди них наилучшего.

Проект P3ARSEC также подтвердил, что для распараллеливания реальных приложений достаточно сочетания небольшого числа схем программ.

В P3ARSEC время выполнения измерялось на трех различных многоядерных системах, представляющих различные классы платформ с разделяемой памятью. Предполагается включить в число целевых платформ конфигурации с графическими процессорами, а также изучить влияние различных (C/C++)-компиляторов.

В работе [9] с помощью набора PARSEC оценивается эффективность и программируемость параллелизма на уровне задач. Средой выполнения служит OmpSs [10] — предвестник задач OpenMP 4.0.

На листинге 7 приведена упрощенная версия теста производительности Ferret, реализованная средствами OmpSs с конвейерным параллелизмом. В OmpSs программист может использовать директивы pragma для идентификации функций, которые должны выполняться асинхронно.

Между задачами могут существовать зависимости по данным, записываемые в директивах `pragma` словами `in`, `out`, `inout`. В рассматриваемом примере зависимости по данным предписывают выполнение задач, порожденных на одной итерации цикла, в последовательном порядке, в то время как задачи из разных итераций могут выполняться параллельно. Исключением является задача `t _ out`, которая разделяет общие выходные данные между всеми экземплярами в outstream и накапливает результаты выполнения Ferret.

```
void load () {
   int i = 0;
   while (load _ image (image [i])) {
   #pragma omp task in (image [i])
                    out (seg _ images [i])
   seg _ images [i] = t _ seg (image [i]);
   #pragma omp task in (seg _ images [i])
                    out (extract _ data [i])
   extract _ data [i] = t _ extract (seg _ images [i]);
   #pragma omp task in (extract _ data [i])
                    out (vectoriz _ data [i])
   vectoriz _ data [i] = t _ vec (extract _ data [i]);
   #pragma omp task in (vectoriz _ data [i])
                    out (rank _ results [i])
   rank _ results [i] = t _ rank (vectoriz _ data [i]);
   #pragma omp task in (rank _ results [i])
                    out (outstream)
   t _ out (rank _ results [i], outstream);
   i++;
   }
   #pragma omp taskwait
}
```

Листинг 7. Реализация теста производительности Ferret в OmpSs

Полезно сопоставить листинги 5 и 7 как два варианта реализации конвейерного параллелизма. В смысле эффективности реализация Ferret в OmpSs не лучше, чем при использовании Pthreads, но масштабируемость последней и так была очень высока (ускорение в 14 раз на 16-ядерной платформе). При этом уменьшение числа строк исходного текста оказалось почти двукратным (46 %).

В работе [9] проведено сравнение эффективности для десяти тестов из набора PARSEC на аппаратных конфигурациях с числом ядер от 1 до 16. Средний выигрыш производительности составил 13 %, в некоторых случаях он достигал 42 %. Улучшилась по сравнению с реализацией средствами Pthreads и программируемость, в среднем число строк исходного текста сократилось на 28 %, а в одном из случаев даже на 81 %.

В работе [5] показано, что число строк, которые пришлось модифицировать и/или добавить при использовании FastFlow или SkePU, в среднем на 34 % меньше, чем для OmpSs-версии при примерном равенстве по эффективности и общему числу строк исходных текстов.

Можно сделать вывод, что высокоуровневые распараллеливающие схемы программ предпочтительны при написании программ "с нуля", так как предоставляют максимум свободы, а средства уровня OpenMP, вероятно, оптимальны при распараллеливании унаследованного (C/C++)-кода.

Представляется, что работы по оценке эффективности распараллеливания находятся в начальной стадии. Первые результаты обнадеживают, однако необходимо расширение спектра целевых платформ, как многоядерных, так и распределенных, а также расширение набора тестов производительности.

Схемы программ в контексте разработки аппаратуры, поддерживающей параллелизм

Один из основных способов повышения вычислительной и энергетической эффективности компьютерных систем — использование специализированных сопроцессоров (ускорителей). Реализация подобных сопроцессоров в виде заказных микросхем сопряжена с высокими затратами ресурсов, длительным циклом разработки и тиражирования и целесообразна только для самых массовых применений.

Решение этой проблемы могут дать реконфигурируемые архитектуры.

Принципиальным вопросом является выбор уровня детализации реконфигурируемых архитектур. Представляется, что детализация с точностью до отдельных бит является чрезмерной, ведущей к вычислительной и энергетической неэффективности. Баланс между эффективностью и реконфигурируемостью достигается для крупноблочных реконфигурируемых архитектур (*coarse-grain reconfigurable architecture*, CGRA), оперирующих на словном уровне детализации.

Еще одна сложная проблема — программирование для реконфигурируемых архитектур. Использование подобных архитектур для создания параллельных систем делает программирование сложным вдвойне. Помочь найти баланс между производительностью аппаратуры и продуктивностью программистов могут распараллеливаемые схемы программ и автоматизация преобразования программ, написанных с использованием схем, в описание целевой аппаратной реконфигурируемой архитектуры. Этот подход выбран в работе [11]. В ней представлена крупноблочная реконфигурируемая архитектура ускорителей Plasticine, оптимизированная для эффективного выполнения распараллеливаемых схем программ.

Plasticine — это двумерный массив реконфигурируемых устройств двух видов: вычислительных (*Pattern Compute Unit*, PCU) и запоминающих (*Pattern Memory Unit*, PMU). Каждое устройство PCU представляет собой реконфигурируемый многостадийный конвейер SIMD функциональных устройств с поддержкой кроссконвейерного сдвига и редукции. Устройства PMU составлены из банков сверхоперативной памяти и специализированной адресной логики и декодеров адресов. Все эти устройства взаимодействуют между собой посредством конвейеризованного межсоединения, точнее, трех видов статических межсоединений: пословного для передачи скаляров; многословного для передачи векторов; побитного для передачи управляющих данных.

Иерархическая организация архитектуры Plasticine способствует поддержке вложенного параллелизма эффективности выполнения. Так, внутренний цикл вычислений может быть отображен на одно устройство PCU, чтобы большинство операндов передавалось напрямую между функциональными устройствами без обращений к памяти и без использования каналов взаимодействия разных PCU.

Отображение программы, написанной с использованием распараллеливаемых схем, в архитектуру Plasticine совершается в несколько этапов. Сначала программа отображается в язык описания аппаратуры *Delite Hardware Definition Language* (DHDL). Этот процесс описан в работе [12].

Конвейеры в DHDL являются либо внешними контроллерами, содержащими только другие конвейеры, либо внутренними контроллерами, не содержащими других конвейеров, а только граф потоков данных, состоящий из вычислений и обращений к памяти. Для отображения DHDL в Plasticine выполняется развертка внешних конвейеров с использованием заданных пользователем или автоматически вычисленных коэффициентов распараллеливания. Результирующее развернутое представление используется для выделения и планирования виртуальных устройств PCU и PMU. Эти виртуальные устройства являются абстрактным представлением устройств в архитектуре Plasticine с неограниченным числом входов, выходов, регистров, вычислительных стадий и цепочек счетчиков.

Поскольку внешние контроллеры не содержат вычислений, они отображаются в виртуальные PCU без вычислительных стадий, только с управляющей логикой и цепочками счетчиков. Вычисления во внутренних контроллерах планируются путем линеаризации графа потоков данных и отображения получившегося списка в виртуальные стадии и регистры. Каждая область локальной памяти отображается в виртуальное устройство PMU.

Затем каждое виртуальное устройство отображается в набор физических устройств путем разделения стадий. Виртуальные устройства PCU разбиваются на несколько физических PCU, в то время как виртуальное устройство PMU отображается на одно физическое PMU и, возможно, несколько вспомогательных PCU. Жадные алгоритмы с несколькими простыми эвристиками обеспечивают хорошее приближение к оптимальному распределению физических устройств.

После выделения физических устройств генерируется управляющая логика, соответствующая иерархии контроллеров. Крупноблочная архитектура Plasticine позволяет завершить процесс компиляции за несколько минут, в отличие от часов, требуемых для стандартных процедур генерации ПЛИС из низкоуровневого описания.

Plasticine — это параметризованная архитектура. В число параметров входят число устройств PCU и PMU, число секций и стадий в PCU, размер и число банков сверхоперативной памяти и т. п. Вообще говоря, значения параметров для разных устройств могут быть разными, но для эффективной реализации распараллеливаемых схем программ в этом нет необходимости. В работе [11] рассмотрена конфигурация из 64 устройств PCU, каждое из которых состоит из 16 6-стадийных секций. Устройств PMU также 64, размер одного банка сверхоперативной памяти 16 Кбайт, число банков — 16. 128 устройств PCU и PMU организованы в массив размера 16×8. Получилась конфигурация с высокой производительностью, умеренным энергопотреблением, отсутствием перерасхода ресурсов. При площади 113 мм^2 и технологии 28 нм максимальная потребляемая мощность составила 49 Вт при тактовой частоте 1 ГГц. По сравнению с ПЛИС, выполненными по аналогичным проектным нормам, производительность Plasticine оказалась выше почти на два порядка, для производительности-на-ватт картина аналогична.

Программы, написанные с использованием распараллеливаемых схем и предназначенные для

аппаратной реализации, необходимо оптимизировать. При этом кроме собственно распараллеливания требуется максимизировать локальность обрабатываемых данных. Одно из возможных решений этой задачи предложено в работе [13]. Основа решения — разбиение на блоки данных и циклов. В результате схема программы преобразуется в пару вложенных схем. При этом во внутреннем цикле обработка идет поблочно, так чтобы каждый блок помещался в сверхоперативную память, а внешний цикл смещается по индексам с большим шагом, переходя от одного блока к другому.

В качестве примера рассмотрим оптимизацию программы, выполняющей кластеризацию по методу *k*-средних. На листинге 8 представлен начальный (неоптимизированный) вариант одной итерации, записанный с использованием распараллеливаемых схем.

```
//matrix: кластеризуемый большой набор данных
val matrix = Matrix.fromFile (path) (parseMatrix)
val clusters = Matrix.fromFunction (numClusters, numFeatures) ((i,j) =>
math.random ())

//данные неявно перераспределяются посредством операции индексирования
matrix (as)
val assigned = matrix.mapRows { row = >
   clusters.mapRows (centroid = > dist(row, centroid)).minIndex
}
val newClusters = clusters.mapIndices { i =>
   val as = assigned.filter (a => a == i)
   matrix (as).sumRows.map (s => s / as.count)
}
```

Листинг 8. Неоптимизированный вариант одной итерации цикла кластеризации по методу *k*-средних, записанный с использованием распараллеливаемых схем

На листинге 9 представлен оптимизированный вариант, записанный на псевдокоде. Кластеризуемые `n` точек разбиты на блоки размера `b0`, а множество из `k` кластеров — на блоки размера `b1`.

```
Для каждого блока из b0 точек:
   Скопировать блок точек в локальную память
   Для каждого блока из b1 центроидов:
     Скопировать блок центроидов в локальную память
     Для каждой точки pt1 в блоке точек:
       Для каждого центроида pt2 в блоке центроидов:
         Вычислить расстояние между pt1 и pt2
         Сохранить ближайшую пару (индекс, расстояние)
       Конец для
     Конец для
     Для каждой точки: сохранить ближайшую пару по всем блокам
   Конец для
   Для каждой точки pt1 в блоке точек:
     Выделить индекс ближайшего центроида
     Добавить pt1 к строке minDistIndex
     Увеличить счетчик в minDistIndex
     Добавить точку и подсчитать суммы по всем блокам
   Конец для
   Добавить точку и подсчитать суммы по всем блокам
Конец для

Для каждого блока из b1 сумм точек и счетчиков:
   Скопировать блок сумм точек в локальную память
   Скопировать блок счетчиков точек в локальную память
   Вычислить каждый новый центроид как сумма / счетчик
Конец для
```

Листинг 9. Оптимизированный вариант одной итерации цикла кластеризации по методу *k*-средних, записанный на псевдокоде

Обратим внимание на дисциплину доступа к центроидам. Вместо того чтобы вычислять для каждой точки ближайший из всех центроидов, данные о последних обрабатываются также поблочно, что позволяет уменьшить число обращений к основной памяти в `b0` раз. Это характерное оптимизирующее преобразование, полезное для многих приложений.

В целом сочетание распараллеливаемых схем программ и крупноблочной реконфигурируемой архитектуры представляется весьма перспективным. Здесь уже имеются практически полезные наработки, достигнута высокая степень автоматизации.

Схемы программ и обучение параллельному программированию

Параллельные архитектуры давно стали нормой, поэтому параллельное программирование является частью базовой подготовки специалистов в области информатики. Профессиональные программисты оперируют схемами программ, значит, и студентам уместно прививать подобную культуру с самого начала обучения.

В работах [14, 15] рассматриваются "патернлеты" — небольшие ("минималистские") программы, представляющие, как правило, одну распараллеливаемую схему. Распараллеливание осуществляется директивами для OpenMP или MPI. Первоначально в исходном тексте эти директивы закомментированы, так что студент может наблюдать поведение последовательной программы, а затем, после раскомментирования, увидеть функционирование параллельной версии и оценить разницу.

На листинге 10 приведена реализация схемы Fork-Join.

```
include <stdio.h> // printf ()
include <omp.h> // OpenMP

int main (int argc, char** argv) {
   printf ("\nПеред...\n");

   // #pragma omp parallel
   printf ("\nВо время...");

   printf ("\n\nПосле...\n\n");

   return 0;
}
```

Листинг 10. Пример патернлета Fork-Join

Если раскомментировать директиву `#pragma omp parallel`, то на четырехъядерном компьютере программа выдаст результат, показанный на листинге 11.

```
Перед...

Во время...
Во время...
Во время...
Во время...

После...
```

Листинг 11. Возможная выдача патернлета Fork-Join

Патернлет Fork-Join можно использовать для иллюстрации понятия "одна программа — множество данных" (*single program multiple data*, SPMD) (листинг 12).

```
#include <stdio.h> // printf ()
#include <omp.h> // OpenMP функции

int main (int argc, char** argv) {
   printf ("\n");

   // #pragma omp parallel
   {
     int id = omp _ get _ thread _ num();
     int numThreads = omp _ get _ num _ threads ();
     printf ("Привет от потока %d из %d\n", id, numThreads);
   }

   printf ("\n");
   return 0;
}
```

Листинг 12. Иллюстрация понятия "одна программа — множество данных" (OpenMP-версия)

Одним из высокоуровневых средств синхронизации являются барьеры. На листинге 13 показано его использование.

```
#include <stdio.h> // printf ()
#include <omp.h> // OpenMP функции
#include <stdlib.h> // atoi ()
```

```
int main (int argc, char** argv) {
   printf ("\n");
   if (argc > 1) {
     omp _ set _ num _ threads (atoi (argv [1]));
   }

   #pragma omp parallel
   {
     int id = omp _ get _ thread _ num ();
     int numThreads = omp _ get _ num _ threads ();
     printf ("Поток %d из %d ПЕРЕД барьером.\n", id, numThreads);

   // #pragma omp barrier

   printf ("Поток %d из %d ПОСЛЕ барьера.\n", id, numThreads);
   }
   printf ("\n");
   return 0;
}
```

Листинг 13. Пример использования барьеров

Распараллеливание редукции — типичное действие. Пример его реализации приведен на листинге 14.

```
#include <stdio.h> // printf ()
#include <omp.h> // OpenMP
#include <stdlib.h> // rand ()

void initialize (int* a, int n);
int sequentialSum (int* a, int n);
int parallelSum (int* a, int n);

#define SIZE 1000000

int main (int argc, char** argv) {
   int array [SIZE];
   if (argc > 1) {
     omp _ set _ num _ threads (atoi (argv [1]));
   }
   initialize (array, SIZE);
   printf ("\nПоследовательная сумма: \t%d\nПараллельная сумма: \t%d\n\n", sequentialSum
(array, SIZZE), parallelSum (array, SIZE));
    return 0;
}

// Заполнение массива случайными значениями
void initialize (int* a, int n) {
   for (int i = 0; i < n; i++) {
     a [i] = rand () % 1000;
   }
}
// Последовательное суммирование массива
int sequentialSum (int* a, int n) {
   int sum = 0;

   for (int i = 0; i < n; i++) {
     sum += a [i];
   }
   return sum;
}
// Параллельное суммирование массива несколькими потоками
int parallelSum (int* a, int n) {
   int sum = 0;

   // #pragma omp parallel for // reduction (+:sum)
   for (int i = 0; i < n; i ++) {
     sum += a [i];
   }
return sum;
}
```

Листинг 14. Пример распараллеливания редукции с использованием нескольких потоков управления

Обратим внимание, что здесь директива OpenMP закомментирована в два приема. Если раскомментировать только ее начало, получится неверная программа с распараллеленным циклом (функция для последовательного суммирования в этом примере нужна, чтобы обеспечить эталонный результат). Если теперь раскомментировать еще и директиву `reduction`, OpenMP обеспечит корректность параллельного суммирования.

Другие примеры патернлетов можно найти, используя работу [16].

Краткий обзор

Вероятно, первой работой, посвященной формализации понятия схемы программы в контексте параллелизма, стала статья [17].

В работах [18, 19] проведены категорирование и каталогизация распараллеливаемых схем. В книгах [20, 21], помимо систематизации, содержатся детальное рассмотрение и многочисленные примеры использования распараллеливаемых схем. Эти книги могут служить учебниками по параллельному программированию на основе схем программ.

Опубликовано множество работ, посвященных представлению и использованию распараллеливаемых схем программ средствами современных стандартов C++ и библиотеки шаблонов STL. Выделим статьи [22—25]. Последняя содержит описание современного (C++)-интерфейса для OpenCL.

MapReduce [26] можно считать классикой высокоуровневого параллельного программирования. Работа [27] развивает и обобщает этот подход. В работе [28] освещены вопросы компиляции высокоуровневых параллельных программ для кластеров и систем на графических процессорах. Система Data Parallel Haskell [29] поддерживает вложенный параллелизм.

Преобразования программ со вложенным параллелизмом исследованы в работах [30—32]. В работе [33] обобщен подход к оптимизации циклов на основе полиэдрального анализа.

В статье [34] предложена реализация схемы "разделяй и властвуй" для многоядерных систем средствами Intel TBB. В работе [35] та же проблема решена распараллеливанием на уровне задач.

Распараллеливанию унаследованного последовательного кода посвящено множество публикаций. Выделим работу [36], где предложено аппаратно-программное решение. В статье [37] конвейерный параллелизм выявлен средствами адаптивной компиляции. В работе [38] аналогичный подход применен для выявления распараллеливаемых циклов "do-all". Сходную направленность имеет статья [39]. В работе [40] путем анализа зависимостей между вызовами функций выявляется схема "мастер/работник".

Схемы конвейерного параллелизма и тесты производительности для них рассмотрены в работе [41]. В статьях [42, 43] применяются только микротесты производительности. Работ по этой тематике относительно немного, еще меньше внимания уделено анализу продуктивности "параллельных" программистов.

Полезным инструментом для генерации реконфигурируемой аппаратуры на основе схем программ является Chisel [44]. В работе [45] описана реконфигурируемая архитектура ускорителей, сочетающая крупноблочные и детальные элементы. Генерация аппаратуры на основе высокоуровневых специализированных языков программирования с поддержкой распараллеливаемых схем описана в работе [46].

После инициатив, описанных в работах [47, 48] в области обучения информационным технологиям, параллельное программирование перешло из курсов по выбору в разряд обязательных. Появившиеся после этого публикации ориентированы на использование схем программ в учебных курсах. Выделим работу [49], где описана среда визуального программирования с автоматической генерацией OpenMP-кода. Весьма полезна работа [50], в которой представлены схемы не только "как надо", но и "как не надо" программировать. В работе [51] описано web-приложение для автоматической оценки студенческих параллельных программ, написанных с использованием многопотоковости в C++ 11, OpenMP, MPI и CUDA. Оперативная обратная связь повышает эффективность учебного курса. Подход аналогичной направленности с акцентом на оценку эффективности описан в работе [52].

Несомненно, крайне желательна подготовка пространного обзора распараллеливаемых схем программ и их использования.

Заключение

Как показано в настоящей работе, использование схем целесообразно для разработки программ, для распараллеливания унаследованных последовательных программ, для оценки производительности аппаратно-программных систем, поддерживающих параллелизм, для разработки аппаратуры, включающей специализированные ускорители, для поддержки учебных курсов по параллельному программированию.

Распараллеливаемые схемы программ являются зрелой научно-технической областью, продолжающей активно развиваться. По мнению авторов, приоритетными являются следующие направления:

- стандартизация устоявшихся аспектов распараллеливаемых схем, в частности унификация набора высокоуровневых операций;
- расширение набора схем, доступных разработчикам в разных предметных областях;
- повышение уровня автоматизации подбора и настройки схем;
- развитие оптимизирующей трансляции схем для различных целевых платформ;
- разработка учебных курсов на основе распараллеливаемых схем для аспирантов соответствующих специальностей.

Список литературы

1. **Huda Z. U., Jannesari A., Wolf F.** Using template matching to infer parallel design patterns // ACM Transactions on Architecture and Code Optimization. 2015. Vol. 11, Issue 4. Article N 64.
2. **Astorga D. R., Dolz M. F., Sanchez L. M., Garca J. D.** Discovering Pipeline Parallel Patterns in Sequential Legacy C++ Codes // Proc. of PMAM'16, Barcelona, Spain, March 12—16, 2016. P. 11—19.
3. **Bienia C., Kumar S., Singh J. P., Li K.** The PARSEC Benchmark Suite: Characterization and Architectural Implications // Proc. of PACT'08, Toronto, Ontario, Canada, October 25—29, 2008. P. 72—81.
4. **PARSEC.** URL: http://parsec.cs.princeton.edu/overview.htm
5. **De Sensi D., De Matteis T., Torquati M.** et al. Bringing Parallel Patterns Out of the Corner: The P^{3}ARSEC Benchmark Suite // ACM Transactions on Architecture and Code Optimization. 2017. Vol. 14, Issue 4. Article N 33.
6. **P3ARSEC.** URL: https://github.com/ParaGroup/p3arsec
7. **Danelutto M., Torquati M.** Structured Parallel Programming with "core" FastFlow // Central European functional programming school. 2015. Vol. 8606, LNCS. P. 29—75.
8. **Ernstsson A., Li L., Kessler C.** SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems // International Journal of Parallel Programming. 2018. Vol. 46, Issue 1. P. 62—80.
9. **Chasapis D., Casas M., Moret M.** et al. PARSECSs: Evaluating the impact of task parallelism in the PARSEC benchmark suite. // ACM Transactions on Architecture and Code Optimization. 2016. Vol. 12, Issue 4. Article N 41.
10. **Duran A., Ayguad E., Badia R. M.** et al. OmpSs: A proposal for programming heterogeneous multi-core architectures // Parallel Processing Letters. 2011. Vol. 21, Issue 2. P. 173—193.
11. **Prabhakar R., Zhang Y., Koeplinger D.** et al. Plasticine: A Reconfigurable Architecture For Parallel Patterns // Proc. of ISCA'17, Toronto, ON, Canada, June 24—28, 2017. P. 389—402.
12. **Koeplinger D., Prabhakar R., Zhang Y.** et al. Automatic Generation of Efficient Accelerators for Reconfigurable Hardware // Proc. of ACM/IEEE 43rd Annual International Symposium on Computer Architecture, 2016, P. 115—127.
13. **Prabhakar R., Koeplinger D., Brown K. J.** et al. Generating Configurable Hardware from Parallel Patterns // Proc. of ASPLOS'16, Atlanta, Georgia, USA, April 26, 2016. P. 651—665.
14. **Adams J. C.** Injecting Parallel Computing into CS2 // Proc. of SIGCSE14, Atlanta, Georgia, USA, March 5—8, 2014. P. 277—282.
15. **Adams J. C.** Patternlets — A teaching tool for introducing students to parallel design patterns // Journal of Parallel and Distributed Computing. 2017. Vol. 105. P. 31—41.
16. **CSinParallel**. URL: http://csinparallel.org
17. **Baruch O., Katz S.** Partially Interpreted Schemas for CSP Programming // Science of Computer Programming. 1988. Vol. 10, N 1. P. 1—18.
18. **Johnson R.** Center for Programming Models for Scalable Parallel Computing. Parallel Programming Patterns, Final Report. April, 2013. URL: http://hdl.handle.net/2142/43368
19. **Keutzer K., Mattson T.** A pattern language for parallel programming ver2.0. URL: http://parlab.eecs.berkeley.edu/wiki/patterns/patterns
20. **McCool M., Robinson A., Reinders J.** Structured Parallel Programming: Patterns for Efficient Computation. Morgan Kaufmann, Elsevier, Inc. 2012. 433 p.
21. **Barlas G.** Multicore and GPU Programming: An Integrated Approach. Morgan Kaufmann, Elsevier Inc. 2015. 698 p.
22. **Dastgeer U., Kessler C.** Smart Containers and Skeleton Programming for GPU-Based Systems // International Journal of Parallel Programming. 2016. Vol. 44, Issue 3. P. 506—530.
23. **Haidl M., Gorlatch S.** PACXX: towards a unified programming model for programming accelerators using C++ 14 // Proc. of the 2014 LLVM Compiler Infrastructure in HPC, New Orleans, LA, USA, 17 Nov. 2014. P. 1—11.
24. **Hoberock J.** Working draft, technical specification for C++ extensions for parallelism. Technical Report N4505, ISO/IEC JTC1/SC22/WG21, 2015.
25. **Potter R., Keir P., Bradford R. J., Murray A.** Kernel composition in SYCL // Proc. of the 3rd International Workshop on OpenCL, Palo Alto, California, USA, May 12—13, 2015. P. 1—7.
26. **Dean J., Ghemawat S.** MapReduce: Simplified Data Processing on Large Clusters // Communications of the ACM — 50th anniversary issue: 1958—2008. 2008. Vol. 51, Issue 1. P. 137—150.
27. **Isard M., Yu Y.** Distributed data-parallel computing using a high-level programming language // Proc. of SIGMOD'09, Providence, Rhode Island, USA, June 29 — July 2, 2009. P. 987—994.
28. **Rossbach C. J., Yu Y., Currey J.** et al. Dandelion: a compiler and runtime for heterogeneous systems // Proc. of SOSP'13, Farmington, Pennsylvania, USA, Nov. 2013. P. 49—68.
29. **Jones S. L. P., Leshchinskiy R., Keller G., Chakravarty M. M. T.** Harnessing the multicores: Nested data parallelism in Haskell // Proc. of the 6th Asian Symposium on Programming Languages and Systems, Bangalore, India, December 09—11, 2008. P. 383—414.
30. **Rompf T., Sujeeth A. K., Amin N.** et al. Optimizing data structures in high-level programs // Proc. of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Rome, Italy, January 23—25, 2013. P. 497—510.
31. **Bravenboer M., van Dam A., Olmos K., Visser E.** Program transformation with scoped dynamic rewrite rules // Fundamenta Informaticae — Program Transformation: Theoretical Foundations and Basic Techniques. Part 2. 2006. Vol. 69, Issue 1—2. P. 123—178.
32. **Veldhuizen T. L., Siek J. G.** Combining optimizations, combining theories. Technical report, Indiana University, 2008. URL: https://cs.indiana.edu/ftp/techreports/TR582.pdf
33. **Benabderrahmane M.-W., Pouchet L.-N., Cohen A., Bastoul C.** The polyhedral model is more widely applicable than you think// Proc. of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction, Paphos, Cyprus, March 20—28, 2010. P. 283—303.
34. **Gonzalez C. H., Fraguela B. B.** A generic algorithm template for divide-and-conquer in multicore systems // Proc. of the 2010 IEEE 12th International Conference on High Performance Computing and Communications, HPCC'10, Washington, DC, USA, 2010. P. 79—88.
35. **Poldner M., Kuchen H.** Task parallel skeletons for divide and conquer // Proc. of the Workshop of the Working Group Programming Languages and Computing Concepts of the German Computer Science Association GI, Bad Honnef, 2008. URL: http://danae.uni-muenster.de/lehre/kuchen/PUBLICATIONS/Honnef08.pdf
36. **Campanoni S., Brownell K., Kanev S.** et al. HELIX-RC: An architecture-compiler co-design for automatic parallelization of irregular programs // Proc. of the ACM/IEEE 41st International Symposium on Computer Architecture (ISCA'14), 2014. P. 217—228.
37. **Rul S., Vandierendonck H., De Bosschere K.** A profile-based tool for finding pipeline parallelism in sequential programs // Parallel Computing. 2010. Vol. 36, Issue 9. P. 531—551.
38. **Garcia S., Jeon D., Louie C. M., Taylor M. B.** Kremlin: Rethinking and rebooting gprof for the multicore age//Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11), New York, NY, USA. 2011. P. 458—469.
39. **Wang Z., Tournavitis G., Franke B., O'Boyle M. F. P.** Integrating profile-driven parallelism detection and machine-learning-based mapping // ACM Transactions on Architecture and Code Optimization. 2014. Vol. 11, Issue 1. Article 2.
40. **Molitorisz K.** Pattern-based refactoring process of sequential source code//Proc. of the 17th European Conference on Software Maintenance and Reengineering (CSMR'13). URL: http://ps.ipd.kit.edu/backend/tl_files/ipd/Dateien/cat_veroeffentlichungen/Molitorisz_CSMR 013.pdf
41. **Lee I-T. A., Leiserson C. E., Schardl T. B.** et al. On-the-fly pipeline parallelism // ACM Transactions on Parallel Computing. 2015. Vol. 2, Issue 3. Article 17.
42. **Astorga D. R., Dolz M. F., Fernndez J., Garca J. D.** A generic parallel pattern interface for stream and data processing. URL: https://onlinelibrary.wiley.com/doi/full/10.1002/cpe.4175
43. **Leyton M., Piquer J. M.** Skandium: Multi-core programming with algorithmic skeletons // Proc. of the 18th Euromicro Conference on Parallel, Distributed, and Network-Based Processing (PDP'10), 17—19 Feb. 2010. P. 289—296.
44. **Bachrach J., Vo H., Richards B.** et al. Chisel: Constructing hardware in a Scala embedded language. URL: https://chisel.eecs.berkeley.edu/chisel-dac2012.pdf
45. **Gao M., Kozyrakis C.** HRL: Efficient and flexible reconfigurable logic for near-data processing // Proc. of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), Barcelona, Spain, 12—16 March 2016. P. 126—137.
46. **George N., Lee H. J., Novo D.** et al. Hardware system synthesis from Domain-Specific Languages // Proc. of the 24th International Conference on Field Programmable Logic and Applications (FPL), 2—4 Sept. 2014, Munich, Germany, 2014. P. 1—8.

47. **Prasad S. K., Chtchelkanova A., Dehne F.** et al. NSF/IEEE-TCPP curriculum initiative on parallel and distributed computing core topics for undergraduates, version I, 2012. URL: http://www.cs.gsu.edu/~tcpp/curriculum/index.php
48. **Sahami M., Roach S., Cuadro-Vargas E., LeBlanc R.** ACM/IEEE-CS computer science curriculum 2013 // Proc. of the 44th ACM Technical Symposium on Computer Science Education, Denver, CO, March 2013. P. 13—14.
49. **Feng A., Gardner M., Feng W.-C.** Parallel programming with pictures is a Snap! // Journal of Parallel and Distributed Computing. 2017. Vol. 105, Issue C. P. 150—162.
50. **Fagerholm F., Hellas A., Luukkainen M.** et al. Designing and implementing an environment for software start-up education: Patterns and anti-patterns // The Journal of Systems and Software. 2018. Vol. 146. P. 1—13.
51. **Hundt C., Schlarb M., Schmidt B.** SAUCE: A web application for interactive teaching and learning of parallel programming // Journal of Parallel and Distributed Computing. 2017. Vol. 105. P. 163—173.
52. **Chaudhury B., Varma A., Keswani Y.** et al. Let's HPC: A web-based platform to aid parallel, distributed and high performance computing education // Journal of Parallel and Distributed Computing. 2018. Vol. 118. P. 213—232.

Using Program Patterns for Parallel Programming. Practical Usage

V. A. Galatenko, galat@niisi.ras.ru, **N. I. Viukova,** niva@niisi.ras.ru, **K. A. Kostyukhin**, kost@niisi.ras.ru, Federal State Institution "Scientific Research Institute for System Analysis of the Russian Academy of Sciences", Moscow, 117218, Russian Federation,

Corresponding author:

Kostyukhin Konstantin A., Senior Researcher, Federal State Institution "Scientific Research Institute for System Analysis of the Russian Academy of Sciences", Moscow, 117218, Russian Federation,
E-mail: Kost@niisi.ras.ru

Received on January 09, 2019
Accepted on February 19, 2019

This article discusses the traditional mechanisms of the program patterns usage. In addition, the authors consider the problematic issues that arise when using patterns for software development, for parallelization of legacy serial programs, to evaluate the performance of hardware and software systems that support parallelism, to develop equipment that includes specialized accelerators, as well as to support training courses on parallel programming.

Program patterns usage is a universal tool of accumulation and approbation of programming knowledge, especially important and useful in complex subject areas, such as parallel programming.

Usage of the parallelizable patterns allows the developers:

- *to maintain a balance between the performance of hardware and software systems and the productivity of programmers, to make the development of parallel software economically feasible;*
- *to make parallel programs mobile and flexible, facilitate maintenance and porting of parallel software to new hardware platforms.*

This article describes a wide systematic approach to program patterns as a tool of parallelization. This is the most important result presented in this paper.

Keywords: *program patterns, parallel programming, distributed systems, review, heterogeneous systems, multikernel systems, parallelizing methods*

For citation:

Galatenko V. A., Viukova N. I., Kostyukhin K. A. Using Program Patterns for Parallel Programming. Practical Usage, *Programmnaya Ingeneria*, 2019, vol. 10, no. 6, pp. 243—256.

DOI: 10.17587/prin.10.243-256

References

1. **Huda Z. U., Jannesari A., Wolf F.** Using template matching to infer parallel design patterns, *ACM Transactions on Architecture and Code Optimization*, 2015, vol. 11, issue 4, article N 64.
2. **Astorga D. R., Dolz M. F., Sanchez L. M., Garca J. D.** Discovering Pipeline Parallel Patterns in Sequential Legacy C++ Codes, *Proc. of PMAM'16*, Barcelona, Spain, March 12—16, 2016, pp. 11—19.
3. **Bienia C., Kumar S., Singh J. P., Li K.** The PARSEC Benchmark Suite: Characterization and Architectural Implications, *Proc. of PACT'08*, Toronto, Ontario, Canada, October 25—29, 2008, pp. 72—81.
4. **PARSEC**, available at: http://parsec.cs.princeton.edu/overview.htm
5. **De Sensi D., De Matteis T., Torquati M., Mencagli G., Danelutto M.** Bringing Parallel Patterns Out of the Corner: The P3ARSEC Benchmark Suite, *ACM Transactions on Architecture and Code Optimization,* 2017, vol. 14, issue 4, article N 33.
6. **P3ARSEC**, available at: https://github.com/ParaGroup/p3arsec
7. **Danelutto M., Torquati M.** Structured Parallel Programming with "core" FastFlow, *Central European functional programming school,* 2015, vol. 8606, LNCS, pp. 29—75.
8. **Ernstsson A., Li L., Kessler C.** SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems, *International Journal of Parallel Programming,* 2018, vol. 46, issue 1, pp. 62—80.
9. **Chasapis D., Casas M., Moret M., Vidal R., Ayguad E., Labarta J., Valero M.** PARSECSs: Evaluating the impact of task parallel-

ism in the PARSEC benchmark suite, *ACM Transactions on Architecture and Code Optimization,* 2016, vol. 12, issue 4, article N 41.

10. **Duran A., Ayguad E., Badia R. M., Labarta J., Martinell L., Martorell X., Planas J.** OmpSs: A proposal for programming heterogeneous multi-core architectures, *Parallel Processing Letters*, 2011, vol. 21, issue 2, pp. 173—193.

11. **Prabhakar R., Zhang Y., Koeplinger D., Feldman M., Zhao T., Hadjis S., Pedram A., Kozyrakis C., Olukotun K.** Plasticine: A Reconfigurable Architecture For Parallel Patterns, *Proc. of ISCA'17*, Toronto, ON, Canada, June 24—28, 2017, pp. 389—402.

12. **Koeplinger D., Prabhakar R., Zhang Y., Delimitrou C., Kozyrakis C., Olukotun K.** Automatic Generation of Efficient Accelerators for Reconfigurable Hardware, *Proc. of ACM/IEEE 43rd Annual International Symposium on Computer Architecture*, 2016, pp. 115—127.

13. **Prabhakar R., Koeplinger D., Brown K. J., Lee H. J., De Sa C., Kozyrakis C., Olukotun K.** Generating Configurable Hardware from Parallel Patterns, *Proc. of ASPLOS'16*, Atlanta, Georgia, USA, April 26, 2016, pp. 651—665.

14. **Adams J. C.** Injecting Parallel Computing into CS2, *Proc. of SIGCSE14,* Atlanta, Georgia, USA, March 5—8, 2014, p. 277—282.

15. **Adams J. C.** Patternlets — A teaching tool for introducing students to parallel design patterns, Journal of Parallel and Distributed Computing, 2017, vol. 105, pp. 31—41.

16. **CSinParallel**, available at: http://csinparallel.org

17. **Baruch O., Katz S.** Partially Interpreted Schemas for CSP Programming, *Science of Computer Programming*, 1988, vol. 10, no. 1, pp. 1—18.

18. **Johnson R.** *Center for Programming Models for Scalable Parallel Computing*, Parallel Programming Patterns, Final Report, April, 2013, available at: http://hdl.handle.net/2142/43368

19. **Keutzer K., Mattson T.** A pattern language for parallel programming ver2.0, available at: http://parlab.eecs.berkeley.edu/wiki/patterns/patterns

20. **McCool M., Robinson A., Reinders J.** *Structured Parallel Programming: Patterns for Efficient Computation*, Morgan Kaufmann, Elsevier, Inc., 2012, 433 p.

21. **Barlas G.** *Multicore and GPU Programming: An Integrated Approach*, Morgan Kaufmann, Elsevier Inc., 2015, 698 p.

22. **Dastgeer U., Kessler C.** Smart Containers and Skeleton Programming for GPU-Based Systems, *International Journal of Parallel Programming*, 2016, vol. 44, issue 3, pp. 506—530.

23. **Haidl M., Gorlatch S.** PACXX: towards a unified programming model for programming accelerators using C++ 14, *Proc. of the 2014 LLVM Compiler Infrastructure in HPC,* New Orleans, LA, USA, 17 Nov. 2014, pp. 1—11.

24. **Hoberock J.** Working draft, technical specification for C++ extensions for parallelism, Technical Report N4505, ISO/IEC JTC1/SC22/WG21, 2015.

25. **Potter R., Keir P., Bradford R. J., Murray A.** Kernel composition in SYCL, *Proc. of the 3rd International Workshop on OpenCL*, Palo Alto, California, USA, May 12—13, 2015, pp. 1—7.

26. **Dean J., Ghemawat S.** MapReduce: Simplified Data Processing on Large Clusters, *Communications of the ACM — 50th anniversary issue*: 1958—2008, 2008, Vol. 51, Issue 1, pp. 137—150.

27. **Isard M., Yu Y.** Distributed data-parallel computing using a high-level programming language, *Proc. of SIGMOD'09*, Providence, Rhode Island, USA, June 29—July 2, 2009, pp. 987—994.

28. **Rossbach C. J., Yu Y., Currey J., Martin J.-P., Fetterly D.** Dandelion: a compiler and runtime for heterogeneous systems, *Proc. of SOSP'13*, Farmington, Pennsylvania, USA, Nov. 2013, pp. 49—68.

29. **Jones S. L. P., Leshchinskiy R., Keller G., Chakravarty M. M. T.** Harnessing the multicores: Nested data parallelism in Haskell, *Proc. of the 6th Asian Symposium on Programming Languages and Systems,* Bangalore, India, December 09—11, 2008, pp. 383—414.

30. **Rompf T., Sujeeth A. K., Amin N., Brown K., Jovanovic V., Lee H., Jonnalagedda M., Olukotun K., Odersky M.** Optimizing data structures in high-level programs, *Proc. of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Rome, Italy, January 23—25, 2013, pp. 497—510.

31. **Bravenboer M., van Dam A., Olmos K., Visser E.** Program transformation with scoped dynamic rewrite rules, Fundamenta Informaticae — Program Transformation, *Theoretical Foundations and Basic Techniques. Part 2,* 2006, vol. 69, issue 1—2, pp. 123—178.

32. **Veldhuizen T. L., Siek J. G.** Combining optimizations, combining theories. Technical report, Indiana University, 2008, available https://cs.indiana.edu/ftp/techreports/TR582.pdf

33. **Benabderrahmane M.-W., Pouchet L.-N., Cohen A., Bastoul C.** The polyhedral model is more widely applicable than you think, *Proc. of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction*, Paphos, Cyprus, March 20—28, 2010, pp. 283—303.

34. **Gonzalez C. H., Fraguela B. B.** A generic algorithm template for divide-and-conquer in multicore systems, *Proc. of the 2010 IEEE 12th International Conference on High Performance Computing and Communications, HPCC'10*, Washington, DC, USA, 2010, pp. 79—88.

35. **Poldner M., Kuchen H.** Task parallel skeletons for divide and conquer, *Proc. of the Workshop of the Working Group Programming Languages and Computing Concepts of the German Computer Science Association GI,* Bad Honnef, 2008, available http://danae.uni-muenster.de/lehre/kuchen/PUBLICATIONS/Honnef08.pdf

36. **Campanoni S., Brownell K., Kanev S., Jones T. M., Wei G.-Y., Brooks D.** HELIX-RC: An architecture-compiler co-design for automatic parallelization of irregular programs, *Proc. of the ACM/IEEE 41st International Symposium on Computer Architecture (ISCA'14),* 2014, pp. 217—228.

37. **Rul S., Vandierendonck H., De Bosschere K.** A profile-based tool for finding pipeline parallelism in sequential programs, *Parallel Computing,* 2010, vol. 36, issue 9, pp. 531—551.

38. **Garcia S., Jeon D., Louie C. M., Taylor M.** Kremlin: Rethinking and rebooting gprof for the multicore age, *Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11),* New York, NY, USA, 2011, pp. 458—469.

39. **Wang Z., Tournavitis G., Franke B., O'Boyle M. F. P.** Integrating profile-driven parallelism detection and machine-learning-based mapping. *ACM Transactions on Architecture and Code Optimization*, 2014, vol. 11, issue 1, article 2.

40. **Molitorisz K.** Pattern-based refactoring process of sequential source code, *Proc. of the 17th European Conference on Software Maintenance and Reengineering (CSMR'13*), available at: http://ps.ipd.kit.edu/backend/tl_files/ipd/Dateien/cat_veroeffentlichungen/Molitorisz_CSMR 013.pdf

41. **Lee I-T. A., Leiserson C. E., Schardl T. B., Zhang Z., Sukha J.** On-the-fly pipeline parallelism, *ACM Transactions on Parallel Computing*, 2015, vol. 2, issue 3, Article 17.

42. **Astorga D. R., Dolz M. F., Fernndez J., Garca J. D.** A generic parallel pattern interface for stream and data processing, available at: https://onlinelibrary.wiley.com/doi/full/10.1002/cpe.4175

43. **Leyton M., Piquer J. M.** Skandium: Multi-core programming with algorithmic skeletons, *Proc. of the 18th Euromicro Conference on Parallel, Distributed, and Network-Based Processing (PDP'10),* 17—19 Feb. 2010, pp. 289—296.

44. **Bachrach J., Vo H., Richards B., Lee Y., Waterman A., Avizienis R., Wawrzynek J., Asanovic K.** Chisel: Constructing hardware in a Scala embedded language, available at: https://chisel.eecs.berkeley.edu/chisel-dac2012.pdf

45. **Gao M., Kozyrakis C.** HRL: Efficient and flexible reconfigurable logic for near-data processing, *Proc. of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA*), Barcelona, Spain, 12—16 March 2016, pp. 126—137.

46. **George N., Lee H. J., Novo D., Rompf T., Brown K. J., Sujeeth A. K., Odersky M., Olukotun K., Ienne P.** Hardware system synthesis from Domain-Specific Languages, *Proc. of the 24th International Conference on Field Programmable Logic and Applications (FPL)*, Munich, Germany, 2—4 Sept. 2014, pp. 1—8.

47. **Prasad S. K., Chtchelkanova A., Dehne F.** et al. NSF/IEEE-TCPP curriculum initiative on parallel and distributed computing core topics for undergraduates, version I, 2012, available at: http://www.cs.gsu.edu/~tcpp/curriculum/index.php

48. **Sahami M., Roach S., Cuadro-Vargas E., LeBlanc R.** ACM/IEEE-CS computer science curriculum 2013, *Proc. of the 44th ACM Technical Symposium on Computer Science Education,* Denver, CO, March 2013, pp. 13—14.

49. **Feng A., Gardner M., Feng W.-C.** Parallel programming with pictures is a Snap!, *Journal of Parallel and Distributed Computing*, 2017, vol. 105, issue C, pp. 150—162.

50. **Fagerholm F., Hellas A., Luukkainen M., Kyllnen K., Yaman S., Maenpa H.** Designing and implementing an environment for software start-up education: Patterns and anti-patterns, *The Journal of Systems and Software*, 2018, vol. 146, pp. 1—13.

51. **Hundt C., Schlarb M., Schmidt B.** SAUCE: A web application for interactive teaching and learning of parallel programming, *Journal of Parallel and Distributed Computing*, 2017, vol. 105, pp. 163—173.

52. **Chaudhury B., Varma A., Keswani Y., Bhatnagar Y., Parikh S.** Let's HPC: A web-based platform to aid parallel, distributed and high performance computing education, *Journal of Parallel and Distributed Computing,* 2018, vol. 118, pp. 213—232.

УДК 004.4, 004.7 DOI: 10.17587/prin.10.257-264

С. А. Марченков, аспирант, мл. науч. сотр., e-mail: marchenk@cs.karelia.ru,
Петрозаводский государственный университет

Автоматизация процессов программирования агентов на основе кодогенерации при построении семантических сервисов интеллектуальных пространств. Часть 1

Рассмотрено решение задачи упрощения разработки и сопровождения приложений интеллектуальных пространств за счет создания средств автоматизации процессов программирования агентов при построении семантических сервисов. В части 1 статьи сформулированы проблемные вопросы и требования для решения рассматриваемой задачи. В качестве примеров приложений рассмотрены окружения умного музея и интеллектуального зала. Предложено решение, направленное на создание генератора программного кода агентов на основе онтологий сервисов, разрабатываемых с помощью объектно-ориентированных языков программирования.

Ключевые слова: *интеллектуальные пространства, семантические сервисы, автоматизация программирования, онтолого-ориентированная разработка, кодогенерация*

Введение

Интернет вещей (*Internet of Things*, далее — IoT) предоставляет разнообразные сетевые вычислительные среды, в которых может быть развернут определенный класс сервис-ориентированных систем — интеллектуальные пространства (ИП) [1]. Концепция таких пространств, реализующая интеллектуальное окружение, используется в различных проблемных областях. К их числу относятся: совместная деятельность [2, 3]; мобильное здравоохранение [4]; цифровые музеи и культурное наследие [5, 6]. В исследовании, результаты которого представлены в двух частях, рассматриваются ИП, реализуемые на основе архитектуры M3 [1, 7]. Аббревиатура M3 указывает на свойства *multi-device* (множество устройств), *multi-vendor* (множество производителей аппаратуры) и *multi-domain* (множество предметных областей) [8]. Основными участниками в IoT-средах выступают умные объекты, которые обеспечивают пользователей вычислительными и информационными ресурсами при совместном решении возникающих задач [9]. Построение ИП направлено на создание многоагентных сервис-ориентированных приложений на основе умных объектов, когда программные агенты выполняются на разнообразных окружающих IoT-устройствах, совместно создавая востребованные в данный момент сервисы и доставляя их пользователям.

В последнее время для разработки взаимодействующих систем и сервис-ориентированных архитектур все чаще используют стандарты семантических веб-сервисов. Такие стандарты определены в концепции Семантического веба [10]. Интеллектуальные пространства позволяют создавать семантические сервисы за счет того, что обеспечивается осведомленность агентов о задействованных ресурсах и происходящих процессах. Однако в отличие от глобального видения Семантического веба [10], ИП ориентированы на локализованные IoT-окружения, а не на всю систему веб-ресурсов. Использование семантических сервисов меняет принципы создания ИП-приложений, усложняются процессы не только их разработки, но и сопровождения. В связи с постоянно растущим и динамически меняющимся набором участников и источников данных, возникает необходимость в создании новых решений для упрощения и автоматизации процессов разработки и сопровождения приложений.

Настоящая работа состоит из двух частей. В части 1 сформулированы проблемные вопросы и требования, связанные с упрощением процессов разработки и сопровождения ИП-приложений за счет автоматизации процессов программирования агентов при построении семантических сервисов. В качестве иллюстрирующих примеров, на основе которых обосновывается актуальность поставленной задачи, рассмотрены окружения умного музея и интеллектуального зала. Предложено решение, направленное на создание генератора программного кода агентов на основе онтологий сервисов при использовании объектно-ориентированных языков программирования. Генератор позволяет производить, в дополнение к объектной модели предметной области, элементы программной логики агентов, отвечающей за взаимодействие. В части 2 статьи будет предложена унифицированная онтология семантического сервиса интеллектуальных пространств, а также алгоритмы кодогенерации программных агентов.

Интеллектуальные пространства и семантические сервисы

Подход к созданию приложений на основе интеллектуальных пространств объединяет технологии Интернета вещей и методы Семантического веба, создавая определенный класс повсеместных окружений [1]. Интеллектуальный характер такого подхода обусловлен необходимостью обеспечения участников ИП сервисами в условиях их массового использования, наличия неоднородности вычислительных устройств и программных компонентов, их физической распределенности, а также разнообразия используемых ресурсов и возможных средств сетевых коммуникаций для передачи данных между участниками. Участниками взаимодействия выступают программные агенты, называемые процессорами знаний (*Knowledge Processor*, далее — агент КР), которые являются потребителями и производителями общего информационного содержимого. Подсистемы взаимодействующих агентов КР в таком окружении образуют приложения в ИП (ИП-приложения), предназначенные для построения и доставки сервисов в рамках определенной проблемной области.

Промежуточное ПО (платформа) для развертывания ИП представляет собой программный слой, реализующий совместное использование информационного содержимого агентами КР. Платформа поддерживает разнообразные семантические интероперабельные примитивы доступа, в том числе и онтолого-ориентированные. На основе онтолого-ориентированных примитивов поддерживается использование семантических рассуждений, которые позволяют преобразовать интегрированные низкоуровневые данные в высокоуровневые знания для реализации решений возникших задач. В настоящее время доступна открытая программная реализация платформы для создания таких ИП, которая именуется Smart-M3 [8].

Каждый агент КР в ИП-приложении работает в соответствии с определенной проблемной областью и моделью косвенного взаимодействия с другими агентами КР в процессе построения и доставки сервисов [11]. Разработчик программной логики агента КР использует API платформы для доступа к информационному содержимому ИП [12]. С точки зрения агентов КР хранение информации организовано как RDF-граф, как правило, в соответствии с некоторой онтологией, определенной с помощью языка описания OWL. Доступ к информационному содержимому может быть либо низкоуровневым, либо высокоуровневым. Низкоуровневый доступ требует, чтобы код агента работал с RDF-тройками напрямую, используя операции протокола доступа, где тройки являются основными элементами обмена. Высокоуровневый доступ определяет организацию кода агента на основе высокоуровневых объектов онтологии (классы, отношения, индивиды) с предопределенными структурами данных и методами API. Такие методы используют низкоуровневые операции доступа в качестве базового нижележащего слоя.

Построение сервисов в ИП реализуется как распределенный вычислительный процесс [7], который позволяет создавать более сложные системные решения на основе косвенного взаимодействия агентов КР. Другими словами, сервисы создаются в результате совместной работы агентов КР. Такие агенты выполняют пошаговый процесс изменения общего информационного содержимого ИП на основе моделей "классная доска" и "публикация/подписка" в целях реализации прикладной функции и обеспечения взаимодействия с ресурсами вычислительной среды [13]. Потребителями сервисов часто являются пользователи. Поэтому процесс извлечения знаний и доставки сервиса, как правило, персонализирован с учетом приоритетов и предпочтений пользователей, рассматриваемых в контексте текущей ситуации и состояния окружения.

Развитие концепции Семантического веба и связанных с ней таких концепций, как Веб 3.0 [10] и Семантический веб вещей (*Semantic Web of Things*) [14], задает направление развития сервисов ИП в сторону семантических сервисов. Описание семантического сервиса представляется машинно-интерпретируемой онтологией сервиса. Как правило, доступ к таким сервисам осуществляется через глобальную сеть Интернет. Для этого используется семантическое представление сервиса как веб-сайта. Сервисы ИП могут быть определены как семантические, которые должны иметь однозначно описанную семантику, быть доступными среди других разнородных окружений, быть пригодными для автоматизированного поиска, композиции, проактивного построения и упреждающей доставки. Использование семантических сервисов в рамках подхода ИП меняет требования к разработке сервисов, а следовательно, ИП-приложений. В связи с постоянно растущим и динамически меняющимся набором участников в окружении ИП (сервисы, агенты КР) возрастает сложность этапов разработки и сопровождения сервисов. Определим основные требования для разработки ИП-приложений с использованием семантических сервисов.

Унифицированная онтология сервиса. Проектирование семантических сервисов должно выполняться на основе общей унифицированной онтологии. Такая онтология не только определяет интерфейс сервиса с точки зрения передаваемых данных и возвращаемых значений, но и задает назначение сервиса, описывает процесс его построения и однозначно определяет его семантику. Благодаря такому единообразному способу проектирования, сервисы различных приложений ИП приобретают возможность взаимодействовать друг с другом независимо от проблемной области, среды и окружения, в которых развертывается ИП. Обеспечивая таким образом сетевое взаимодействие окружений ИП, можно добиться интеграции как самих ИП, так и их приложений для решения совместных задач на основе семантических сервисов.

Автоматизация процессов программирования агентов. Необходим способ разработки приложений, позволяющий уменьшить количество программно-

го кода, создаваемого прикладным разработчиком во время выполнения рутинных задач, за счет использования инструментальных средств автоматизированного проектирования и программирования. В частности, автоматизация процессов программирования агентов при построении сервисов может быть достигнута за счет использования онтологий семантических сервисов. Онтологии принимаются в качестве входных параметров для генерации объектной модели и шаблонов программного кода для объектно-ориентированных языков программирования. Благодаря пониманию семантики сервиса, а также сведений о доступных ресурсах среды, на основе онтологий может быть достигнута самоорганизация агентов КР путем определения их функциональных ролей, моделей взаимодействия и операций/функций в процессе построения и доставки сервиса [15].

Интеллектуальные окружения совместной деятельности

В настоящей работе интеллектуальные окружения совместной деятельности (СД) выступают как иллюстрирующие примеры, на которых обосновывается актуальность поставленной задачи упрощения разработки и сопровождения ИП-приложений, а также демонстрируются возможности практического применения полученных решений. Такие окружения реализуются в виде ИП-приложения или их набора, обеспечивая информационную поддержку пользователей для достижения целевых установок их совместной деятельности за счет привлечения разнообразных ресурсов. Ресурсы представляют различные сущности (физические, цифровые, а также самих людей-участников) и обеспечивают решения отдельных задач СД. Для организации такой поддержки необходимо обеспечивать участников сервисами контекстной информации, напоминаний и рекомендаций в проактивной и упреждающей манере. В качестве представителей интеллектуальных окружений СД рассматриваются окружения умного музея [6] и интеллектуального зала [3].

Интеллектуальные пространства используются для создания окружений умного музея и интеллектуального зала в виде единого динамического информационного пространства, где всеми участниками (представленными программными агентами КР) обсуждается и определяется смысл используемых в нем объектов в соответствии с общей целью СД. Программная инфраструктура окружений разрабатывается в виде набора взаимодействующих программных агентов, отвечающих за построение сервисов с использованием объединенных разнообразных ресурсов. Разрабатываемые элементы инфраструктуры разворачиваются на сетевых вычислительных устройствах, реализуя неоднородную распределенную систему.

Окружение умного музея реализуется на базе Музея истории Петрозаводского государственного университета, материалы которого помимо исторических фактов описывают повседневную жизнь преподавателей, исследователей и студентов (например, питание, обстановку в комнате общежития, одежду). Программная инфраструктура окружения умного музея обеспечивает построение сервисов для коллективного семантического аннотирования, связывания информации и персонализированного доступа к корпусу источников по истории университета (рис. 1). Сервисы обеспечивают привлечение разнородных видов ресурсов:

- экспертные исторические знания сотрудников музея и индивидуальная информация от посетителей музея;
- музейная информационная система, внешние интернет-ресурсы исторических данных, вики-системы для формирования семантической информации об экспонатах музея;
- различное медиаоборудование и сенсоры для расширения возможностей окружения.

Сервис посещения обеспечивает построение и адаптацию персонализированной программы посещения на основе рекомендуемых экспонатов для каждого посетителя музея. Сервис экспозиции визуализирует информацию об экспонатах на широкоформатных экранах, расположенных в среде музея, а также на мобильных устройствах посетителей. Сервис обогащения обеспечивает дополнение семантической сети сотрудниками музея и посетителями. Посетитель музея может обогатить описания изученных им экспонатов с использованием метода коллективного семантического аннотирования путем добавления видео-, аудио- или текстовой информации.

Окружение интеллектуального зала предназначено для обеспечения информационной поддержки людей в организации таких мероприятий СД, как конференция и семинары. Программная инфраструктура окружения обеспечивает функционирование сервисов и разворачивается в ограниченном физическом пространстве. Такое пространство включает в себя разнородные ресурсы: специализированные вычислительные устройства, внешние интернет-сервисы и ресурсы людей-участников (рис. 2).

Важным классом сервисов являются опорные сервисы и сервисы хранения содержимого. Эти сервисы предоставляют базовые функциональные возможности окружения: обеспечение проведения конференций и семинаров, сбор и распространение материалов и другого мультимедийного контента между участниками. Сервис управления конференцией отвечает за формирование программы выступлений докладчиков, ее адаптацию в ходе конференции и переключение управления докладами между выступающими участниками. Формирование программы выступлений осуществляется на основе взаимодействия с сервисом управления содержимым. Такой сервис отвечает за регистрацию докладов, сбор материалов докладчиков и распространение материалов между всеми участниками.

Сервисы в основном предназначены для формирования информационного содержимого ИП с помощью

Рис. 1. Концептуальная модель окружения умного музея

Рис. 2. Концептуальная модель окружения интеллектуального зала

данных, поступающих от физических устройств окружения и внешних информационных источников. Таким образом реализуются виртуальные образы данных ресурсов, которые семантически связываются друг с другом. Например, сервис определения присутствия и анализа активности участников предоставляет пользователю информацию о состоянии сенсора, о его измерениях в ИП, формирует его виртуальный цифровой образ, а также обнаруживает новые знания об активности участников на основе этой информации [16]. Сервис присутствия использует метод пассивной радиолокации, основанный на измерениях мощности принимаемого сигнала мобильного устройства с помощью сетевого сенсора.

Для рассматриваемых случаев окружений процесс разработки программной инфраструктуры приводит к значительным трудозатратам. В силу отсутствия унифицированной онтологии для проектирования сервиса прикладному разработчику окружений необходимо для каждого сервиса, число которых может динамически изменяться в зависимости от привлекаемых ресурсов, проектировать собственную онтологию. Такая онтология, как правило, не определяет интерфейс сервиса и процесс его построения, а описывает только задействованные ресурсы и другие объекты предметной области. Отсутствие единообразного интерфейса у сервисов (входные и выходные параметры, предусловия и результаты) ограничивает возможности их композиций. Например, для случая интеллектуального зала, результат выполнения сервиса определения присутствия участников (уровень присутствия) может служить входным параметром для сервиса сопровождения и визуализации программы СД. Этот результат инициализирует процесс формирования приветственных сообщений на экранах и используется для предоставления контекста о присутствии участников во время управления программой конференции или семинара.

Далее прикладному разработчику предлагается онтологическая библиотека [12] для создания программных агентов КР, которая реализует семантические примитивы доступа к ИП и генерирует программный код структур данных на основе спроектированной OWL-онтологии. Однако полученные структуры данных не позволяют в полной степени отразить классы, отношения и ограничения онтологии. Эти структуры определяют только полные (с пространством имен) семантические имена классов и отношений для дальнейшего составления соответствующих данным сущностям RDF-троек для публикации в информационное содержимое ИП. Заключительным шагом для разработчика является реализация внутренней логики программных агентов КР для построения сервисов без возможности генерации программного кода функций агентов.

Автоматизация этапов онтолого-ориентированной разработки

Разработка ИП-приложений следует принципам онтолого-ориентированной разработки программного обеспечения. Согласно этим принципам этап проектирования сводится к созданию спецификации определенной предметной области и сервисов в виде описания OWL/RDF. Использование онтологий позволяет добиться общего понимания структуры информационного содержимого между агентами КР, упрощения повторного использования знаний за счет уже определенных в других онтологиях понятий, а также поддержки формальной логики и логических рассуждений [17]. Таким образом, выгодно использовать особенности онтолого-ориентированного подхода в случае применения онтологий на всех этапах разработки.

Для обеспечения автоматизации этапов разработки традиционные методы проектирования и разработки заменяются методами, которые способствуют реализации подхода с широким использованием инструментальных средств автоматизированного проектирования (*Computer-Aided Design*, CAD) [18] и автоматизированного программирования (*Computer-Aided Programming*, CAP) [12]. Такие средства поддерживают прототипирование приложений.

Средства CAD используются для автоматизации процессов, направленных на создание и поддержку различных онтологических и графических представлений во время разработки прикладных систем. В свою очередь, средства CAP ориентированы на упрощение задачи программирования агентов КР. Вместо прямого кодирования исполняемого кода разработчик предоставляет онтологию проблемной области и онтологию спецификации сервиса. Такие онтологии являются входными параметрами для алгоритмов генерации кода, позволяющими создавать корректную объектную модель, структуры данных, методы, функции кода и другие элементы целевого языка программирования. Интеграция CAD- и CAP-средств приводит к автоматическому созданию программного кода непосредственно на этапе разработки спецификаций ИП-приложений. Одним из способов распространения таких средств вместе с платформой является предоставление интегрированной среды разработки, развертывания и управления приложениями [19].

Этап проектирования сводится к созданию спецификаций проблемной области и сервисов в виде RDF/OWL-описания. Существует большое число работ, направленных на решение задач с помощью CAD-средств на этапе проектирования для онтолого-ориентированной разработки программного обеспечения [17]. Например, такие инструментальные средства разработки онтологий, как Protégé [20] и OWL-S editor [21], позволяют создавать необходимые онтологические спецификации, предоставляя программную среду для быстрого прототипирования, в которой разработчики онтологий могут предельно быстро создавать индивидов, экспериментировать с семантическими ограничениями, визуализировать онтологические описания.

На этапе реализации агенты КР создаются из спецификаций и моделей, полученных на этапе проектирования, который предполагает использование языков программирования для кодирования отмеченных проектных решений. На этом этапе существующих

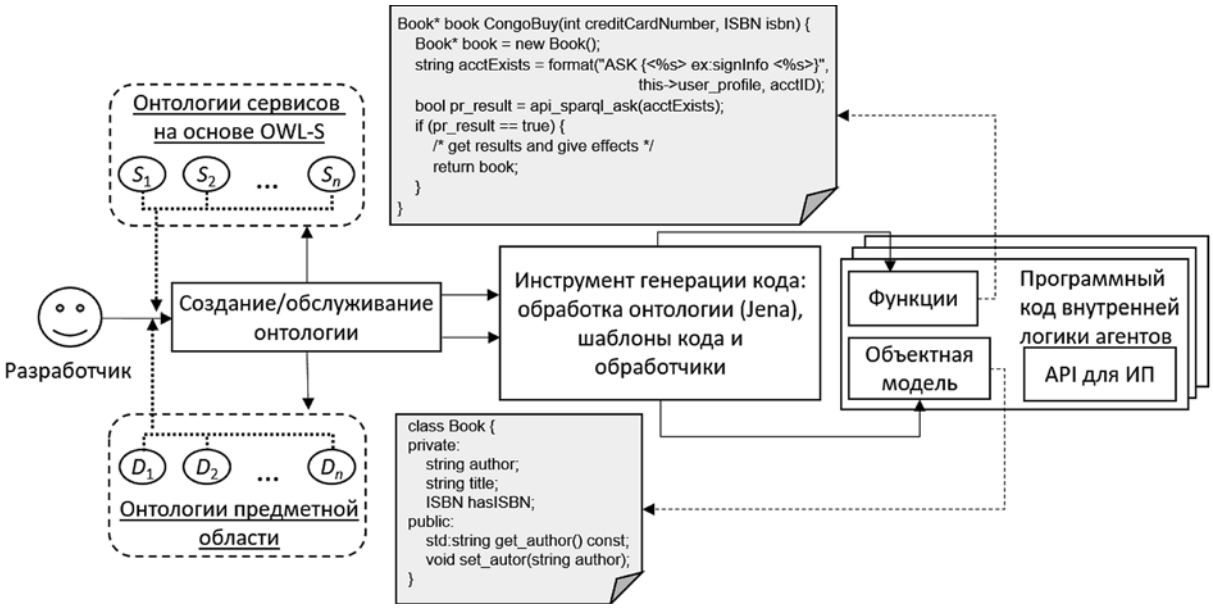

Рис. 3. Общая схема процесса генерации программного кода

CAP-средств [22] недостаточно для решения задачи автоматизации процессов онтолого-ориентированного программирования агентов КР. Такие CAP-средства только частично решают эту задачу путем преобразования OWL-классов, их экземпляров и свойств в соответствующие объекты языка программирования.

Очевидно, что этот подход сложен как при практической реализации, так и для использования в случае применения статически типизированных компилируемых языков программирования. Однако следует отметить, что он удобен для динамически типизированных интерпретируемых и объектно-ориентированных языков программирования. Такой подход прежде всего предназначен для создания структур данных и элементов объектной модели проблемной области агента. Вместе с тем он не позволяет создавать методы, функции и другие элементы внутренней программной логики. Одним из примеров реализации такого подхода является SmartSlogCodeGen, который является частью онтологической библиотеки SmatSlog, предназначенной для создания ИП-приложений. Его механизмы позволяют создавать структуры данных для конкретных сущностей OWL-онтологии [12].

Автором предлагается решение, направленное на создание генератора программного кода агентов КР по онтологии при использовании объектно-ориентированных языков программирования (C++, Java, Python). Общая схема процесса генерации программного кода показана на рис. 3. Основными особенностями предлагаемой схемы генерации кода являются: использование в качестве входных онтологий, совместно с OWL-онтологиями предметной области, онтологий сервиса ИП на основе онтологии OWL-S [23]; генерация, в дополнение к структурам данных, элементов программной логики агентов на основе API ИП-платформы в целях построения и доставки сервисов, а также объектной модели предметной области.

Разработчик агентов предоставляет спецификацию проблемной области как описание OWL и спецификацию сервисов ИП как описание, основанное на OWL-S. Генератор использует статическую схему шаблонов и обработчиков. Шаблоны кода представляют собой "предварительный код" классов, атрибутов и функций, которые реализуют сущности и свойства онтологий OWL, а также сервиса ИП, основанного на OWL-S. Обработчики могут преобразовывать один или несколько шаблонов в конечный код, заменяя специальные теги именами и элементами, взятыми из онтологий. Код может быть сгенерирован для нескольких агентов, в зависимости от того, какие агенты указаны в спецификации сервиса ИП для его построения и доставки. Преобразование происходит во время обхода RDF-графа онтологии с использованием фреймворка Jena. С его помощью строится метамодель для представления графа. На основе информации, полученной во время обхода генератором узлов метамодели, обработчики преобразуют шаблоны в конечный код.

Заключение

В части 1 настоящей работы сформулированы проблемные вопросы и требования, связанные с упрощением разработки и сопровождения ИП-приложений за счет автоматизации процессов программирования агентов при построении семантических сервисов. На основе референтных примеров окружений умного музея и интеллектуального зала представлена актуальность поставленной задачи. Предложено решение, направленное на создание генератора программного кода агентов для объектно-ориентированных языков программирования на основе онтологий сервисов. Использование предложенного решения позволяет повысить эффективность разработки и прототипирования ИП-приложений за

счет генерации объектной модели сервисов и распределенного взаимодействия агентов при их построении.

В части 2 статьи будут предложены однозначное описание семантики сервиса интеллектуальных пространств в виде онтологии, а также алгоритмы кодогенерации семантических сервисов. Практическое применение полученных решений будет рассматриваться на примерах окружений умного музея и интеллектуального зала.

Исследование поддержано Минобрнауки России в рамках инициативного проекта № 2.5124.2017/8.9 базовой части государственного задания на 2017—2019 гг. Научные результаты получены при финансовой поддержке РФФИ, проект № 19-07-01027. Статья подготовлена в рамках реализации Программы развития опорного университета для Петрозаводского государственного университета на 2017—2021 гг.

Список литературы

1. **Korzun D. G., Balandin S. I., Kashevnik A. M.** et al. Smart spaces-based application development: M3 architecture, design principles, use cases, and evaluation // International Journal of Embedded and Real-Time Communication Systems (IJERTCS). 2017. Vol. 8, N. 2. P. 66—100.

2. **Кашевник А. М., Вальченко Ю., Ситаев М. М., Шилов Н. Г.** Интеллектуальная система автоматизированного проведения конференций // Труды СПИИРАН. 2010. Т. 3, № 14. С. 228—245.

3. **Марченков С. А., Вдовенко А. С., Корзун Д. Ж.** Расширение возможностей совместной деятельности в интеллектуальном зале на основе сервисов электронного туризма // Труды СПИИРАН. 2017. Т. 1, № 50. С. 165—189.

4. **Zavyalova Y. V., Korzun D. G., Meigal A. Y., Borodin A. V.** Towards the development of smart spaces-based socio-cyber-medicine systems // International Journal of Embedded and Real-Time Communication Systems (IJERTCS). 2017. Vol. 8, N. 1. P. 45—63.

5. **Smirnov A., Kashevnik A., Ponomarev A.** et al. Smart space-based tourist recommendation system // International Conference on Next Generation Wired/Wireless Networking. Springer, Cham, 2014. P. 40—51.

6. **Korzun D. G., Marchenkov S. A., Vdovenko A. S., Petrina O. B.** A semantic approach to designing information services for smart museums // International Journal of Embedded and Real-Time Communication Systems (IJERTCS). 2016. Vol. 7, N. 2. P. 15—34.

7. **Корзун Д. Ж.** Формализм сервисов и архитектурные абстракции для программных приложений интеллектуальных пространств // Программная инженерия. 2015. № 2. С. 3—12.

8. **Honkola J., Laine H., Brown R., Tyrkkö O.** Smart-M3 information sharing platform // Proc. IEEE Symp. Computers and Communications, Riccione, Italy, 22—25 June 2010. Washington: IEEE Computer Society, 2010. P. 1041—1046.

9. **Gazis V.** A Survey of Standards for Machine-to-Machine and the Internet of Things // IEEE Communications Surveys & Tutorials. 2017. Vol. 19, N. 1. P. 482—511.

10. **Gyrard A., Serrano M., Atemezing G. A.** Semantic web methodologies, best practices and ontology engineering applied to Internet of Things // 2015 IEEE 2nd World Forum on Internet of Things (WF-IoT). IEEE, 2015. P. 412—417.

11. **Галов И. В.** Применение шаблонов проектирования программных приложений для реализации косвенного взаимодействия агентов в интеллектуальном пространстве // Программная инженерия. 2016. Т. 7, № 8. С. 351—359.

12. **Корзун Д. Ж., Ломов А. А., Ванаг П. И.** Автоматизированная модельно-ориентированная разработка программных агентов для интеллектуальных пространств на платформе Smart-M3 // Программная инженерия. 2012. № 5. С. 6—14.

13. **Ломов А. А., Корзун Д. Ж.** Операция подписки для приложений в интеллектуальных пространствах платформы Smart-M3 // Труды СПИИРАН. 2012. Т. 4, № 23. С. 439—458.

14. **Kujur P., Chhetri B.** Evolution of World Wide Web: Journey from Web 1.0 to Web 4.0 // International Journal of Computer Science and Technology. 2015. Vol. 6, N. 1. P. 134—138.

15. **Городецкий В. И.** Самоорганизация и многоагентные системы. I. Модели многоагентной самоорганизации // Известия РАН. Теория и системы управления. 2012. № 2. С. 92—120.

16. **Марченков С. А., Корзун Д. Ж.** Определение присутствия пользователей в интеллектуальном зале на основе отслеживания активности в беспроводной сети // Ученые записки Петрозаводского государственного университета. 2015. № 2. С. 114—119.

17. **Yang C. W., Dubinin V., Vyatkin V.** Ontology driven approach to generate distributed automation control from substation automation design // IEEE Transactions on Industrial Informatics. 2017. Vol. 13, N. 2. P. 668—679.

18. **Добров Б. В., Лукашевич Н. В., Невзорова О. А., Федунов Б. Е.** Методы и средства автоматизированного проектирования прикладной онтологии // Известия Российской академии наук. Теория и системы управления. 2004. № 2. С. 58—68.

19. **Soldatos J., Kefalakis N., Hauswirth M.** et al. Openiot: Open source internet-of-things in the cloud // Interoperability and open-source solutions for the internet of things. Springer, Cham, 2015. P. 13—25.

20. **Knublauch H.** Ontology-driven software development in the context of the semantic web: An example scenario with Protege/OWL // 1st International workshop on the model-driven semantic web (MDSW2004). Monterey, California, USA, 2004. P. 381—401.

21. **Elenius D., Denker G., Martin D.** et al. The OWL-S editor-a development tool for semantic web services // European Semantic Web Conference. Springer, Berlin, Heidelberg, 2005. P. 78—92.

22. **Křemen P., Kouba Z.** Ontology-driven information system design // IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews). 2012. Vol. 42, N. 3. P. 334—344.

23. **OWL-S:** Semantic Markup for Web Services. URL: https://www.w3.org/Submission/OWL-S (дата обращения 27.03.2019).

Computer-Aided Programming of Software Agents Based on Code Generation in Constructing Semantic Services of Smart Spaces. Part 1

S. A. Marchenkov, marchenk@cs.karelia.ru, Department of Computer Science, Petrozavodsk State University, Petrozavodsk, 185910, Russian Federation

Corresponding author:

Marchenkov Sergei A., Postgraduate Student, Junior Researcher, Petrozavodsk State University, Petrozavodsk, 185910, Russian Federation,
E-mail: marchenk@cs.karelia.ru

Received on February 24, 2019
Accepted on April 03, 2019

The first part of the paper considers the problem of simplifying the development and maintenance of applications of smart spaces at the expense of developing solutions aimed at computer-aided agent programming in the construction of semantic services. Developing applications of smart spaces based on semantic services faces a number of problems. Firstly, the lack of a common ontology with the uniquely described concept of service for smart spaces makes the design phase difficult, making it impossible to create unified solutions based on an interconnected understanding of the resources involved and the processes taking place during the construction and delivery of services. Heterogeneous applications operate with their own descriptions of services, limiting their integration through semantic services to solve common tasks. Secondly, the process of developing applications for smart spaces, whose principles are increasingly deviating from efficiency, consistency and standardization, while increasing the complexity of prototyping, requires specialized tools for computer-aided programming of constructing and delivering semantic services. The article proposes a solution aimed at creating an agent code generator based on the ontologies of services developed using object-oriented programming languages, which allows generating, in addition to the domain object model, elements of the agent's program logic responsible for interaction. The environments of the smart museum and the smart room are considered as the reference examples.

Keywords: *smart spaces, semantic services, computer-aided programming, ontology-oriented development, ontology-driven development, code generation*

Acknowledgements: *The research was financially supported by the Ministry of Education and Science of Russia within project # 2.5124.2017/8.9 of the basic part of state research assignment for 2017—2019. The reported study was funded by RFBR according to the research project #19-07-01027. The article was prepared within the Government Program of Flagship University Development for Petrozavodsk State University in 2017—2021.*

For citation:

Marchenkov S. A. Computer-Aided Programming of Software Agents Based on Code Generation in Constructing Semantic Services of Smart Spaces. Part 1, *Programmnaya Ingeneria,* 2019, vol. 10, no. 6, pp. 257—264.

DOI: 10.17587/prin.10.257-264

References

1. **Korzun D. G., Balandin S. I., Kashevnik A. M., Smirnov A. V., Gurtov A. V.** Smart spaces-based application development: M3 architecture, design principles, use cases, and evaluation, *International Journal of Embedded and Real-Time Communication Systems (IJERTCS)*, 2017, vol. 8, no. 2, pp. 66—100.

2. **Kashevnik A. M., Valcenko J., Sitaev M. M., Shilov N. G.** Intellektual'naya sistema avtomatizirovannogo provedeniya konferencij (Intelligent system for conference management automation), *Trudy SPIIRAN*, 2010, vol. 3, no. 14, pp. 228—245 (in Russian).

3. **Marchenkov S. A., Vdovenko A. S., Korzun D. G.** Enhancing the opportunities of collaborative work in intelligent room using e-Tourism services, *Trudy SPIIRAN*, 2017, vol. 1, no. 50. pp. 165—189 (in Russian).

4. **Zavyalova Y. V., Korzun D. G., Meigal A. Y., Borodin A. V.** Towards the development of smart spaces-based socio-cyber-medicine systems, *International Journal of Embedded and Real-Time Communication Systems (IJERTCS)*, 2017, vol. 8, no. 1, pp. 45—63.

5. **Smirnov A., Kashevnik A., Ponomarev A., Teslya N., Shchekotov M., Balandin S. I.** Smart space-based tourist recommendation system, *International Conference on Next Generation Wired/Wireless Networking*, Springer, Cham, 2014, pp. 40—51.

6. **Korzun D. G., Marchenkov S. A., Vdovenko A. S., Petrina O. B.** A semantic approach to designing information services for smart museums, *International Journal of Embedded and Real-Time Communication Systems (IJERTCS)*, 2016, vol. 7, no. 2, pp. 15—34.

7. **Korzun D. G.** Formalizm servisov i arhitekturnye abstrakcii dlja programmnyh prilozhenij intellektual'nyh prostranstv (Service Formalism and Architectural Abstractions for Smart Space Applications), *Programmnaya ingeneria,* 2015, no. 2, pp. 3—12 (in Russian).

8. **Honkola J., Laine H., Brown R., Tyrkkö O.** Smart-M3 information sharing platform, *Proc. IEEE Symp. Computers and Communications*, Riccione, Italy, 22—25 June 2010, Washington: IEEE Computer Society, 2010, pp. 1041—1046.

9. **Gazis V.** A Survey of Standards for Machine-to-Machine and the Internet of Things, *IEEE Communications Surveys & Tutorials*, 2017, vol. 19, no. 1, pp. 482—511.

10. **Gyrard A., Serrano M., Atemezing G. A.** Semantic web methodologies, best practices and ontology engineering applied to Internet of Things, *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, IEEE, 2015, pp. 412—417.

11. **Galov I. V.** Primenenie shablonov proektirovaniya programmnyh prilozhenij dlya realizacii kosvennogo vzaimodejstviya agentov v intellektual'nom prostranstve (Application of software design patterns for implementation of indirect agent interactions in a smart space), *Programmnaya ingeneria,* 2016, vol. 7, no. 8, pp. 351—359.

12. **Korzun D., Lomov A., Vanag P.** Avtomatizirovannaya model'no-orientirovannaya razrabotka programmnyx agentov dlya intellektual'nyx prostranstv na platforme Smart-M3 (Automated Model-Driven Agent Development for Space-Based Applications on the Smart-M3 Platform), *Programmnaya ingeneria*, 2012, no. 5, pp. 6—14 (in Russian).

13. **Lomov A. A., Korzun D.** Operaciya podpiski dlya prilozhenij v intellektual'nyx prostranstvax platformy Smart-M3 (Subscription operation for applications in smart spaces of Smart-M3 platform), *Trudy SPIIRAN*, 2012, vol. 4, no. 23, pp. 439—458 (in Russian).

14. **Kujur P., Chhetri B.** Evolution of World Wide Web: Journey from Web 1.0 to Web 4.0, *International Journal of Computer Science and Technology*, 2015, vol. 6, no. 1. pp. 134—138.

15. **Gorodetskiy V. I.** Samoorganizatsiya i mnogoagentnye sistemy. I. Modeli mnogo-agentnoy samoorganizatsii (Self-organization and multi-agent systems. I. Multi-agent self-organization models), *Izvestiya RAN. Teoriya i sistemy upravleniya*, 2012, no. 2, pp. 92—120 (in Russian).

16. **Marchenkov S. A., Korzun D. Zh.** Opredelenie prisutstviya pol'zovatelej v intellektual'nom zale na osnove otslezhivaniya aktivnosti v besprovodnoj seti (Network activity tracking detection of user presence in smart room), *Uchenye zapiski Petrozavodskogo gosudarstvennogo universiteta*, 2015, no. 2, pp. 114—119 (in Russian).

17. **Yang C. W., Dubinin V., Vyatkin V.** Ontology driven approach to generate distributed automation control from substation automation design, *IEEE Transactions on Industrial Informatics*, 2017, vol. 13, no. 2, pp. 668—679.

18. **Dobrov B. V., Loukachevitch N. V., Fedunov B. E., Nevzorova O. A.** Metody i sredstva avtomatizirovannogo proektirovaniya prikladnoj ontologii), *Izvestiya RAN. Teoriya i sistemy upravleniya,* 2004, no. 2, pp. 58—68 (in Russian).

19. **Soldatos J., Kefalakis N., Hauswirth M., Serrano M., Calbimonte J.-P., Riahi M., Aberer K., Jayaraman P. P., Zaslavsky A., Zarko I. P., Skorin-Kapov L., Herzog R.** Openiot: Open source internet-of-things in the cloud, *Interoperability and open-source solutions for the internet of things*, Springer, Cham, 2015, pp. 13—25.

20. **Knublauch H.** Ontology-driven software development in the context of the semantic web: An example scenario with Protege/OWL, *Ist International workshop on the model-driven semantic web (MDSW2004)*, Monterey, California, USA, 2004, pp. 381—401.

21. **Elenius D., Denker G., Martin D., Gilham F., Khouri J., Sadaati S., Senanayake R.** The OWL-S editor-a development tool for semantic web services, *European Semantic Web Conference*, Springer, Berlin, Heidelberg, 2005. pp. 78—92.

22. **Křemen P., Kouba Z.** Ontology-driven information system design, *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews).* 2012, vol. 42, no. 3, pp. 334—344.

23. **OWL-S:** Semantic Markup for Web Services, available at: https://www.w3.org/Submission/OWL-S (accessed 27.03.2019).

УДК 004.75 + 004.41 DOI: 10.17587/prin.10.265-273

А. С. Шундеев, канд. физ.-мат. наук, вед. науч. сотр., e-mail: alex.shundeev@gmail.com,
МГУ имени М. В. Ломоносова

Об изменении размерности векторного представления текстовых данных

В настоящее время интеллектуальный анализ данных является основой для построения широкого спектра прикладных информационных систем. Современным и бурно развивающимся подходом в области анализа текстовых данных является использование векторных представлений слов и текстов. Векторные представления изначально применяли для решения задач определения смысловой близости слов и поиска аналогий, однако они оказались востребованными также и в области решения задачи классификации текстов. Применительно к этой задаче векторные представления рассматриваются в настоящей работе.

Предложен подход к построению векторных представлений текстов, базирующийся на трансформации согласованных с ними векторных представлений слов. Подобные трансформации подразумевают изменение исходной модели и размерности векторного представления и реализуются в виде решения задачи восстановления многомерной регрессии.

Проведенные над тестовыми наборами данных эксперименты позволяют сделать следующие выводы. Построенные с помощью трансформаций векторные представления документов могут иметь меньшую размерность. При этом их использование в решении задачи классификации текстов в большинстве случаев дает более точный результат, чем при использовании исходных векторных представлений.

Ключевые слова: *векторное представление слов, векторное представление документов, классификация текстов, регрессия*

Введение

Сфера приложений интеллектуального анализа текстовых данных включает целый ряд взаимосвязанных задач, для решения которых часто используются общие принципы и подходы. К числу таких задач относятся задача определения смысловой близости слов (*word similarity*), задача поиска аналогий (*word analogies*), задача классификации текстов (*document classification*, *text categorization*). Приведенный список не является исчерпывающим. Общие для выделенных задач методы решения будут представлять объект исследования в контексте настоящей работы.

Одним из популярных и показавших свою практическую эффективность подходов к решению первых двух из числа перечисленных выше задач стало использование так называемых векторных представлений слов (*word embeddings*). Здесь можно отметить модели Word2Vec [1, 2] и модель GloVe [3]. В рамках этих моделей каждому слову ставится в соответствие вещественный вектор фиксированной размерности. При таком подходе предоставляется возможность поставить в соответствие близким по смыслу словам близкие векторы.

Под упомянутыми выше аналогиями понимаются высказывания следующего вида: слово A связано по смыслу со словом B, как слово C связано по смыслу со словом D. В задаче поиска аналогий слова A, B, C заданы, а слово D необходимо подобрать. Например, слово *женщина* относится к слову *королева*, как слово *мужчина* относится к словам *король*, *принц*, *царь*, *герцог*. Слова *король*, *принц*, *царь*, *герцог* являются найденными решениями.

На языке векторных представлений задача поиска аналогий имеет элегантную формулировку и решение. Предположим, что словам A, B, C, D поставлены в соответствие векторы $\mathbf{v}_A$, $\mathbf{v}_B$, $\mathbf{v}_C$, $\mathbf{v}_D$. Тогда делается предположение, что аналогия может быть записана в виде выражения $\mathbf{v}_B - \mathbf{v}_A \approx \mathbf{v}_D - \mathbf{v}_C$. Поэтому в качестве вариантов искомого слова D нужно подбирать слова, векторные представления которых близки вектору $\mathbf{v}_B - \mathbf{v}_A + \mathbf{v}_C$.

Задача классификации текстов [4] состоит в том, чтобы отнести каждый документ из рассматриваемого набора к определенному классу. Число классов при этом фиксировано. В некоторых постановках этой задачи допускается возможность отнести документ сразу к нескольким классам. Однако в дальнейшем этот случай рассматривать не будем. Предположение о том, что в один класс должны попадать тематически близкие документы, позволяет эффективно использовать векторные представления слов для

решения этой задачи. Например, текст может быть представлен как последовательность, составленная из векторов входящих в него слов. Такое представление в частности используется при построении классификаторов на основе сверточных нейронных сетей (*convolutional neural networks*) [5].

Недостатком такого подхода является то обстоятельство, что разным текстам могут соответствовать векторы разной размерности. Модели векторного представления текстов Doc2Vec [6] позволяют обойти подобное ограничение. Оказывается возможным одновременно построить векторные представления одной фиксированной размерности как для текстов, так и для слов, встречающихся в этих текстах. В результате предоставляется возможность определять смысловую близость и аналогии не только между словами, но и между текстами, а также между текстами и наборами ключевых слов. В дальнейшем подобные векторные представления текстов и слов будем называть согласованными.

Создание векторных представлений является трудоемкой вычислительной задачей. В связи с этим обстоятельством появились подходы [7], реализующие так называемую постобработку готовых векторных представлений в целях повышения их качества. В некоторых случаях побочным эффектом постобработки является уменьшение размерности векторного представления.

Основной задачей исследования, результаты которого представлены в настоящей работе, является выработка подхода к созданию векторных представлений текстов, согласованных с заданным векторным представлением слов. Предполагается, что изначально у рассматриваемого векторного представления слов не было согласованного с ним векторного представления текстов. Побочным результатом решения этой задачи оказалась возможность существенно снижать размер исходных векторных представлений без потери их качества. В данном случае качество оценивается с точки зрения возможности эффективного использования векторного представления для решения задачи классификации текстов. В связи с этим предложенный подход можно отнести к области постобработки векторных представлений.

Дальнейшее изложение структурировано следующим образом. В разд. 1 в качестве примеров векторных представлений слов и текстов описаны модели Word2Vec и Doc2Vec. В разд. 2 приведены основные понятия и описаны используемые в работе методы машинного обучения [8]. В разд. 3 описан алгоритм построения векторного представления текстов, согласованный с заданным векторным представлением слов. Приведены результаты тестирования этого алгоритма. В заключении обсуждены полученные результаты и описаны возможные направления для дальнейших исследований.

1. Векторные представления

В качестве примеров векторных представлений слов и документов рассмотрим модели Word2Vec [1, 2] и Doc2Vec [6].

1.1. Модель Word2Vec

В основе построения многих моделей векторных представлений слов лежит так называемая дистрибутивная гипотеза (*distributional hypothesis*) [9]. Согласно этой гипотезе смысл слова определяется распределением слов, в окружении которых оно встречается в текстах. Покажем, как эта гипотеза реализуется в моделях Word2Vec. В качестве примера рассмотрим следующее предложение:

"Лиса схватила зайца, пока он ел морковку".

После предварительной обработки текста, включающей запись слов в нижнем регистре, приведение слов к их словарной форме (лемматизация), удаление часто встречающихся слов и знаков пунктуации, получим последовательность:

"лиса хватать заяц есть морковь".

Предположим, что из этой последовательности было удалено третье слово, и был оставлен только контекст (окружение), в котором оно присутствовало:

"лиса хватать ____ есть морковь".

Видя этот контекст, содержащий такие слова, как *лиса* и *морковь*, можно с высокой долей уверенности предположить, что было пропущено слово *заяц*.

Подобные рассуждения были положены в основу варианта модели Word2Vec под названием *continuous bag-of-words* (CBOW). В рамках модели CBOW проводится анализ всех контекстов заданного фиксированного размера, встречающихся в рассматриваемом наборе текстов. Целевой установкой является оценка условных вероятностей появления разных слов в окружении рассматриваемых контекстов.

В рамках варианта под названием *skip-gram* (SG) модели Word2Vec решается обратная задача. По заданному слову требуется предугадать слова, которые могут встречаться в его контексте. Для слова *заяц* можно с высокой долей уверенности предположить, что в тексте его соседями могут быть такие слова, как *лиса* и *морковь*.

С математической точки зрения наиболее корректное описание моделей CBOW и SG приводится в работе [10]. Следуя этой работе, опишем формальную постановку и решение задачи для модели CBOW в случае, когда контекст состоит из одного слова. Например, для заданного слова (контекста) требуется предсказать, какие слова могут следовать за ним в рассматриваемом наборе текстов.

Пусть D — это словарь, содержащий все рассматриваемые слова. Каждому слову $w \in D$ ставятся в соответствие два вектора $\mathbf{v}_w$, $\acute{\mathbf{v}}_w \in \mathbb{R}^n$. Первый вектор интерпретируется как векторное представление слова w, а второй вектор носит вспомогательный характер. Фиксированная размерность векторов n выбирается заранее.

Условная вероятность появления слова $w \in D$ в контексте слова $c \in D$ моделируется с помощью выражения вида

$$p(w|c) = \frac{\exp\langle \hat{\mathbf{v}}_w, \mathbf{v}_c \rangle}{\sum_{v \in D} \exp\langle \hat{\mathbf{v}}_v, \mathbf{v}_c \rangle}.$$

Приведенное выражение является эвристикой. Правомочность использования этой эвристики подтверждается практикой. Заметим, что векторы, фигурирующие в правой части выражения, изначально не известны и должны быть вычислены. Для этого используются методы математической статистики. Для условных вероятностей записывается функция правдоподобия

$$L(\Theta) = \prod_{(w,c) \in T} p(w|c; \Theta).$$

Оцениваемыми статистическими параметрами являются векторы, которые ставятся в соответствие словам из словаря

$$\Theta = \left\{ (\mathbf{v}_w, \hat{\mathbf{v}}_w) | w \in D \right\}.$$

В определении функции правдоподобия фигурирует множество T. Оно составлено из всевозможных пар вида (w, c), где c — контекст, w — слово, появившееся в контексте c, извлеченных из заданного набора текстов. В рассмотренном ранее примере множество T состоит из пар слов (*хватать*, *лиса*), (*заяц*, *хватать*), (*есть*, *заяц*), (*морковь*, *есть*).

Далее следуют стандартные шаги по переходу к логарифмической функции правдоподобия (переход от произведения множителей к сумме слагаемых) и решению задачи ее максимизации

$$l(\Theta) = \sum_{(w,c) \in T} \ln p(w|c; \Theta) \to \max_{\Theta}.$$

На практике решается эквивалентная задача минимизации функции $-l(\Theta)$. Для этого используется метод стохастического градиентного спуска, идея которого будет обсуждена позже. Следует отметить, что целевая функция $-l(\Theta)$ не является выпуклой. Поэтому говорить о гарантированном нахождении глобального экстремума не приходится. Использование стохастического алгоритма означает, что результат решения задачи будет зависеть от выбора начального значения генератора псевдослучайных чисел. Проведение серии повторных вычислений, при которых варьируется это начальное значение, позволит получить разные варианты решения задачи. Из этих вариантов может быть выбран наилучший.

Итоговое векторное представление образуют векторы $\mathbf{v}_w$ ($w \in D$), а векторы $\hat{\mathbf{v}}_w$ отбрасывают.

1.2. Модель Doc2Vec

Существует два варианта модели Doc2Vec. Вариант *distributed memory* (DM), соответствующий варианту CBOW модели Word2Vec, а также вариант *distributed bag-of-words* (DBOW), который соответствует варианту SG модели Word2Vec.

В обоих упомянутых выше вариантах каждому тексту ставится в соответствие новое уникальное слово, которое интерпретируется как его идентификатор. Для этих идентификаторов и обычных слов, встречающихся в рассматриваемых текстах, совместно строятся векторные представления. Векторное представление идентификатора интерпретируется как векторное представление соответствующего этому идентификатору текста.

В рамках модели DM моделируются условные вероятности появления различных слов в окружении рассматриваемых контекстов. При этом каждый контекст расширяется за счет добавления к нему идентификатора текста, в котором он был обнаружен.

В рамках модели DBOW моделируются условные вероятности появления различных контекстов внутри заданного текста. При этом контексты интерпретируются как неупорядоченные наборы слов.

2. Методы машинного обучения

Приведем основные понятия и методы из области машинного обучения [8], которые будут использоваться в дальнейшем.

Постановка задачи машинного обучения с учителем (*supervised learning*) предполагает наличие множества объектов $\mathcal{X}$, множества целевых значений $\mathcal{Y}$, а также неизвестной функциональной зависимости между объектами и целевыми значениями. Об этой функциональной зависимости можно судить только по конечному множеству обучающих примеров

$$T = \{(x_i, y_i) | i = 1, \ldots, m\} \subset \mathcal{X} \times \mathcal{Y}.$$

Модель машинного обучения фиксирует множество гипотез $\mathcal{H}$ (функций вида $h: \mathcal{X} \to \mathcal{Y}$), среди которых выбирается "приближение" к неизвестной функциональной зависимости. Если $|\mathcal{Y}| < \infty$, то говорят о задаче классификации. В этом случае элементы множества $\mathcal{Y}$ называют классами. Если $\mathcal{Y} = \mathbb{R}$, то говорят о задаче восстановления регрессии. Модель машинного обучения включает алгоритм, который по заданному множеству обучающих примеров T строит гипотезу $h_T \in \mathcal{H}$, которая рассматривается как результат решения задачи машинного обучения.

Должны быть зафиксированы числовые метрики вида $\mathrm{est}(T, h)$, позволяющие судить о том, насколько хорошо гипотеза $h \in \mathcal{H}$ соответствует множеству T. Подобные метрики используются для оценки качества построенного решения h_T, а также могут использоваться в процессе работы алгоритма машинного обучения, в некоторых случаях являясь настраиваемым параметром.

В процессе решения задачи машинного обучения часто приходится сталкиваться с явлениями недообучения (*underfitting*) и переобучения (*overfitting*). Если оценка $\mathrm{est}(T, h_T)$ признается неудовлетворительной, то констатируют случай недообучения. Если построенная гипотеза h_T показывает хорошие результаты

только на обучающих примерах из T, то констатируют случай переобучения. Эти две ситуации являются взаимосвязанными. Меры, направленные на улучшение одной из них, могут приводить к усугублению другой.

Наличие явления переобучения свидетельствует о том, что одной оценки $\text{est}(T, h_T)$ недостаточно, чтобы судить о качестве решения задачи машинного обучения. Чтобы обойти это ограничение были разработаны методы, получившие общее название скользящий контроль (*cross validation*). Один из вариантов скользящего контроля подразумевает случайное разбиение обучающего множества на два непересекающихся подмножества $T = T_{train} \cup T_{test}$. Обычно тренировочная выборка T_{train} содержит 70 % обучающих примеров, а тестовая выборка T_{test} — оставшиеся 30 %.

Для построения решения используется только множество T_{train}. Далее анализируются две оценки $\text{est}\left(T_{train}, h_{T_{train}}\right)$ и $\text{est}\left(T_{test}, h_{T_{train}}\right)$. По первой оценке судят о том, имело ли место недообучение. Сравнивая обе оценки, судят о том, имело ли место переобучение.

В дальнейшем для решения задач классификации будет использоваться метрика точность (*accuracy*):

$$\text{acc}(T, h) = \frac{1}{m}\sum_{i=1}^{m} 1\{h(x_i) = y_i\}.$$

Чем выше точность (максимальное значение равно 1), тем лучше гипотеза h соответствует обучающим примерам.

Для решения задач восстановления регрессии будет использоваться метрика средний квадрат отклонения (*mean squared error*):

$$\text{mse}(T, h) = \frac{1}{m}\sum_{i=1}^{m} |h(x_i) - y_i|^2.$$

Чем ближе средний квадрат отклонения к нулю, тем лучше гипотеза h соответствует обучающим примерам.

2.1. Линейная регрессия

Линейная регрессия представляет собой наиболее популярный и теоретически исследованный метод решения задачи восстановления регрессии. Этот метод предполагает, что $\mathcal{X} = \mathbb{R}^n$ ($n \geqslant 1$), а гипотезами являются линейные функции вида $h_{\boldsymbol{\theta},\,\theta_0}(\mathbf{x}) = \langle \boldsymbol{\theta}, \mathbf{x} \rangle + \theta_0$. Нахождение требуемой гипотезы осуществляется путем решения задачи минимизации

$$\text{mse}\left(T, h_{\boldsymbol{\theta},\,\theta_0}\right) \to \min_{\boldsymbol{\theta},\theta_0}.$$

Эта задача имеет точное аналитическое решение. Однако на практике чаще используется приближенный численный алгоритм под названием "метод градиентного спуска". Этот метод вычисляет последовательные приближения к точке минимума целевой функции J. В текущей точке вычисляется направление наибольшего убывания целевой функции, которое совпадает с направлением отрицательного градиента $-\nabla J$. В качестве очередного приближения к точке минимума выбирается точка, лежащая на этом направлении.

Для минимизации среднего квадрата отклонения mse можно напрямую использовать метод градиентного спуска. Однако, если размерность пространства объектов n велика, вычисление градиента будет иметь большую погрешность. Чтобы устранить это ограничение, используется модификация этого метода, которая называется методом стохастического градиентного спуска.

Идея метода стохастического градиентного спуска состоит в следующем. Целевая функция mse представляет собой сумму неотрицательных слагаемых, каждому из которых соответствует свой обучающий пример. На каждом шаге вычислительного процесса можно случайным образом выбирать подмножество обучающих примеров. Для очередного приближения будет строиться направление убывания не всей целевой функции, а только суммы слагаемых, соответствующих выбранным обучающим примерам.

Целевая функция mse является выпуклой, поэтому применение метода стохастического градиентного спуска будет порождать последовательность приближений, сходящихся к глобальной точке минимума.

Задача восстановления регрессии может быть обобщена. В качестве множества целевых значений может выступать множество $\mathbb{R}^s$ ($s > 1$). Такую задачу будем называть задачей восстановления многомерной регрессии. Простейший подход к ее решению сводится к решению s отдельных задач восстановления регрессии. Каждой координате целевых значений соответствует своя отдельная задача.

2.2. Логистическая регрессия

Одним из популярных способов решения задачи классификации является метод логистической регрессии. В базовой постановке решается задача бинарной классификации. Предполагается, что $\mathcal{X} = \mathbb{R}^n$ ($n \geqslant 1$), $\mathcal{Y} = \{0, 1\}$. В качестве гипотез выступают функции вида

$$h_{\boldsymbol{\theta},\theta_0}(\mathbf{x}) = \frac{1}{1 + \exp\left(-\langle \boldsymbol{\theta}, \mathbf{x} \rangle - \theta_0\right)}.$$

Гипотеза интерпретируется как условная вероятность принадлежности объекта $\mathbf{x}$ классу 1, а именно

$$p\left(y = 1 | \mathbf{x};\, \boldsymbol{\theta},\, \theta_0\right) = h_{\boldsymbol{\theta},\theta_0}(\mathbf{x}) \quad \text{и}$$

$$p\left(y = 0 \mid \mathbf{x};\ \boldsymbol{\theta},\, \theta_0\right) = 1 - h_{\boldsymbol{\theta},\theta_0}(\mathbf{x}).$$

Если $p(y = 1|\mathbf{x};\, \boldsymbol{\theta},\, \theta_0) > 0{,}5$, то считается, что объект x принадлежит классу 1, иначе он принадлежит классу 0.

Для нахождения подходящей гипотезы по множеству обучающих примеров строится функция правдоподобия, для которой, в свою очередь, решается задача максимизации

$$L(\boldsymbol{\theta},\, \theta_0) = \prod_{i=1}^{m} p\left(y_i | x_i;\, \boldsymbol{\theta},\, \theta_i\right) \to \max_{\boldsymbol{\theta},\theta_i}.$$

Метод логистической регрессии естественным образом обобщается на случай $|\mathcal{Y}| > 2$.

3. Изменение модели и размерности векторного представления

В данном разделе описан алгоритм построения векторного представления текстов, согласованный с заданным векторным представлением слов. Приведены результаты анализа тестирования этого алгоритма.

3.1. Построение векторного представления набора текстов

Пусть $\mathcal{T}$ — набор текстов, а $\mathcal{D}$ — словарь, состоящий из всех слов, встречающихся в текстах из набора $\mathcal{T}$. Отображение вида $\tau: \mathcal{T} \to \mathbb{R}^n$ будем называть векторным представлением текстов, а отображение вида $\delta: \mathcal{D} \to \mathbb{R}^n$ будем называть векторным представлением слов. При этом число n будем называть размерностью векторного представления.

Алгоритм

Вход: δ_1, δ_2 — векторные представления слов, соответственно размерности n и m; τ_1 — векторное представление текстов размерности n; $\mathcal{R}$ — модель решения задачи восстановления многомерной регрессии (не обязательно линейной).

Выход: τ_2 — векторное представление текстов размерности m.

Начало.

1. Построим $T := \{(\delta_1(w), \delta_2(w)) | w \in \mathcal{D}\}$.
2. С помощью модели $\mathcal{R}$ и множества обучающих примеров T построим многомерную регрессию ρ: $\mathbb{R}^n \to \mathbb{R}^m$.
3. Положим $\tau_2 := \rho \circ \tau_1$.

Конец.

В основе описанного алгоритма лежит простая идея, согласно которой каждому слову из рассматриваемого словаря поставлено в соответствие по два вектора. Первый вектор получается с помощью исходного векторного представления τ_1, а второй вектор — с помощью целевого векторного представления τ_2. Подобные пары векторов рассматриваются как обучающие примеры, на основе которых строится многомерная регрессия. Полученная регрессия применяется к векторному представлению текстов, согласованному с исходным векторным представлением слов. Можно сделать эвристически обоснованное предположение, что получившееся векторное представление текстов будет согласовано с целевым векторным представлением слов. Требуется выработать способ проверки степени обоснованности выдвинутого предположения. Точнее, необходимо выработать подход к оценке качества получившегося в результате применения алгоритма векторного представления текстов.

В рамках предлагаемого подхода качество векторного представления текстов оценивается с точки зрения решения задачи классификации. Поэтому будем предполагать, что набор текстов $\mathcal{T}$ разбит на конечное число попарно непересекающихся классов, а также сформированы тренировочная T_{train} и тестовая T_{test} выборки.

Через $\tau(T)$ будем обозначать следующую модификацию множества обучающих примеров T с помощью векторного представления текстов τ. В каждом обучающем примере текст t заменяется на его векторное представление $\tau(t)$.

Зафиксируем некоторую модель C решения задачи классификации. Тогда векторное представление текстов τ можно оценивать через точность решения задачи классификации текстов в рамках модели C. При этом предполагается, что объектами обучающих примеров являются векторы текстов, полученные с помощью представления τ.

Метод скользящего контроля предписывает одновременно оценивать две следующие величины:

$$a_{train}(\tau) = \mathrm{acc}\left(\tau(T_{train}),\ h_{\tau(T_{train})}\right)$$

$$\text{и}\ \ a_{test}(\tau) = \mathrm{acc}\left(\tau(T_{test}),\ h_{\tau(T_{train})}\right)$$

с целью выявления случаев недообучения и переобучения.

Поэтому, если требуется определить, насколько построенное целевое представление текстов τ_2 "лучше" ("хуже") исходного представления τ_1, то разумным выглядит подход, когда в качестве соответствующей оценки берутся значения двух следующих величин:

$$e_{train}(\tau_1, \tau_2) = \frac{a_{train}(\tau_2)}{a_{train}(\tau_1)} \ \text{и}\ \ e_{test}(\tau_1, \tau_2) = \frac{a_{test}(\tau_2)}{a_{test}(\tau_1)}.$$

Если значения построенных величин приблизительно равны единице, то можно считать, что качество исходного и целевого векторных представлений одинаково. Если эти значения строго больше (меньше) единицы, то можно считать, что построенное целевое векторное представление лучше (хуже), чем исходное.

Если имеется набор различных моделей решения задачи классификации, то величины e_{train} и e_{test} могут быть вычислены для каждой из них. В этом случае будем рассматривать максимальное и среднее значение этих величин.

Далее представим результаты экспериментов над тестовыми наборами данных, которые получены в целях анализа эффективности построенного алгоритма.

3.2. Эксперименты

В ходе проведения экспериментов использовали два набора тестовых данных[1]. Каждый набор был предварительно разделен на тренировочную и тестовую выборки. Набор *movies* содержит рецензии к кинофильмам. Каждая рецензия отнесена к одному из шести жанров. Набор состоит из 44 012 элементов. Набор *twitter* состоит из сообщений одноименной социальной сети. Каждое сообщение оценено как позитивное или негативное. Набор состоит из 1 596 753 элементов.

Для каждого набора данных всегда можно построить тривиальный классификатор. Этот классификатор относит все объекты к одному классу, содержащему наибольшее число элементов. Точность тривиального классификатора равна отношению раз-

[1] Наборы тестовых данных находятся в сети Интернет по адресу https://github.com/group112/se2019

мера класса с наибольшим числом элементов к размеру всего набора данных. Точность тривиального классификатора задает своеобразную нижнюю границу. Классификаторы с меньшей точностью следует рассматривать как неадекватные. Для набора *movies* точность тривиального классификатора равна 0,48, а для набора *twitter* — 0,50.

Для генерации векторных представлений типа Word2Vec и Doc2Vec использовалась библиотека Gensim[2]. Для генерации векторных представлений типа Glove использовалась разработанная авторами этой модели программа[3].

Для набора *movies* были созданы исходные векторные представления типа DBOW размерностей 50, 100, 200 и 300. При этом создавались векторные представления текстов и согласованные с ними векторные представления слов. В качестве целевых были созданы векторные представления слов типа CBOW, SG и GloVe размерностей 50, 100 и 200. Для набора *twitter* были созданы векторные представления аналогичных типов, но только размерностей 50 и 100.

В ходе проведения экспериментов были использованы модели машинного обучения, реализованные в библиотеке Scikit-Learn[4]. Использовался класс LinearRegression, реализующий модель линейной регрессии, и класс MultiOutputRegressor, реализующий модель многомерной регрессии. В качестве модели решения задачи классификации была использована логистическая регрессия, реализованная в классе LogisticRegression. Этот класс имеет набор конфигурационных параметров. В частности, может быть выбран алгоритм решения соответствующей оптимизационной задачи (*newton-cg*, *lbfgs*, *liblinear*, *sag*, *saga*), задан генератор псевдослучайных чисел, а также установлен параметр регуляризации. Выбор различных комбинаций значений этих параметров будет приводить к созданию разных классификаторов. В ходе проведения экспериментов рассматривали все предустановленные алгоритмы решения оптимизационной задачи, пять различных начальных значений генератора псевдослучайных чисел и девять различных значений параметра регуляризации в диапазоне $[10^{-5}, 1000]$.

В табл. 1 собраны результаты тестирования алгоритма на наборе данных *movies*. Строка таблицы соответствует исходному векторному представлению, а столбец — целевому. Таким образом, ячейка соответствует регрессии, отображающей исходное векторное представление в целевое векторное представление. Каждая ячейка содержит пять чисел. Первые четыре числа являются характеристиками построенных классификаторов. Это, соответственно, максимальное и среднее значения величины e_{train}, а также максимальное и среднее значения величины e_{test}. Пятое число — это средний квадрат отклонения построенной регрессии.

Построенные классификаторы имели следующие характеристики. На тренировочной (тестовой) выборке минимальная, максимальная и средняя точности равны соответственно 0,48, 0,96 и 0,86 (0,48, 0,87 и 0,79). Все эти числа не меньше точности тривиального классификатора. Было принято решение для последующего анализа оставить результаты только "сильных" классификаторов, точность которых превосходит средние значения. Показатели каждой из ячеек со столбцами типа CBOW и SG были сформированы на основе обобщения результатов 300 пар классификаторов, а показатели ячеек со столбцами типа GloVe — 150 пар классификаторов.

Следует обратить внимание на то, что классификаторы, построенные для целевых векторных представлений, имеют строго большую точность, чем классификаторы, построенные для исходных векторных представлений. Более того, показатели точности зависят только от типа целевого векторного представления и практически не зависят от размерностей исходного и целевого векторных представлений. Этот результат сложно назвать ожидаемым. Построенные целевые векторные представления текстов оказались с точки зрения решения задачи классификации лучше, чем исходные представления.

Неожиданное поведение продемонстрировал также показатель точности регрессии. В каждом столбце он одинаковый. Этот факт означает, что точность регрессии никак не зависит от размерности исходного векторного представления. Если анализировать точность регрессии построчно, то можно заметить, что в границах одного типа целевого векторного представления с ростом его размерности эта точность улучшается.

Результаты тестирования алгоритма на наборе данных *twitter*, приведенные в табл. 2, в общем, подтверждают результаты, полученные на наборе данных *movies*. Некоторое отличие состоит в том, что целевые векторные представления имеют приблизительно те же самые характеристики, что и исходные представления. Нет особых улучшений, но и ухудшения незначительны.

Во всех рассмотренных случаях нельзя сделать вывод, что построенные регрессии обладают большой точностью. Оценки их точности лежат в диапазоне [0,046, 0,390] для набора *movies* и в диапазоне [0,052, 0,656] для набора *twitter*. Можно предположить, что с точки зрения решения задачи классификации какую-либо роль играют только старшие разряды (после запятой) значений координат векторных представлений текстов. Поэтому младшие разряды можно просто отбросить, проведя округления значений координат. Для проверки этой гипотезы была проведена дополнительная серия экспериментов.

Если τ — векторное представление текстов, а n — натуральное число, то через τ_n будем обозначать векторное представление текстов, полученное из τ путем округления значений его координат. В результате округления остаются только n первых цифр после запятой. Качество получившегося векторного представления можно оценить с помощью двух параметров $e_{train}(\tau, \tau_n)$ и $e_{test}(\tau, \tau_n)$.

В табл. 3 приведены результаты округления целевых векторных представлений типа GloVe, созданных

[2] https://radimrehurek.com/gensim

[3] https://nlp.stanford.edu/projects/glove

[4] https://scikit-learn.org

Таблица 1

Результаты тестирования алгоритма на наборе данных *movies*

Исходное представление DBOW (размерность)	Целевое представление (модель, размерность)								
	CBOW 50	CBOW 100	CBOW 200	SG 50	SG 100	SG 200	GloVe 50	GloVe 100	GloVe 200
50	1,030	1,031	1,031	1,019	1,023	1,027	1,029	1,030	1,030
	1,005	1,005	1,006	1,003	1,004	1,005	1,005	1,005	1,005
	1,033	1,034	1,035	1,021	1,026	1,030	1,032	1,034	1,033
	1,005	1,005	1,005	1,003	1,004	1,004	1,005	1,005	1,005
	0,390	**0,232**	**0,129**	**0,101**	**0,069**	**0,046**	**0,282**	**0,175**	**0,099**
100	1,033	1,033	1,033	1,030	1,031	1,032	1,032	1,033	1,034
	1,006	1,006	1,006	1,005	1,006	1,006	1,006	1,006	1,006
	1,035	1,036	1,036	1,032	1,034	1,035	1,035	1,036	1,036
	1,005	1,005	1,006	1,005	1,005	1,005	1,005	1,005	1,006
	0,390	**0,232**	**0,129**	**0,100**	**0,069**	**0,046**	**0,282**	**0,175**	**0,099**
200	1,034	1,035	1,035	1,033	1,034	1,034	1,034	1,034	1,034
	1,006	1,006	1,007	1,006	1,006	1,006	1,006	1,006	1,006
	1,038	1,037	1,037	1,036	1,037	1,037	1,037	1,037	1,037
	1,005	1,005	1,005	1,005	1,005	1,005	1,005	1,005	1,005
	0,389	**0,231**	**0,128**	**0,100**	**0,068**	**0,046**	**0,281**	**0,175**	**0,098**
300	1,036	1,036	1,036	1,035	1,035	1,035	1,035	1,035	1,035
	1,006	1,006	1,007	1,006	1,006	1,007	1,006	1,006	1,007
	1,039	1,039	1,038	1,038	1,038	1,038	1,037	1,037	1,038
	1,006	1,005	1,005	1,005	1,005	1,005	1,005	1,005	1,005
	0,388	**0,231**	**0,128**	**0,100**	**0,068**	**0,046**	**0,280**	**0,174**	**0,098**

Таблица 2

Результаты тестирования алгоритма на наборе данных *twitter*

Исходное представление DBOW (размерность)	Целевое представление (модель, размерность)					
	CBOW 50	CBOW 100	SG 50	SG 100	GloVe 50	GloVe 100
50	1,002	1,003	1,000	1,001	1,001	1,003
	1,000	1,000	0,999	1,000	1,000	1,000
	1,003	1,003	1,000	1,001	1,001	1,003
	1,000	1,000	0,999	1,000	1,000	1,000
	0,657	**0,394**	**0,076**	**0,052**	**0,204**	**0,118**
100	1,004	1,004	1,003	1,004	1,003	1,004
	1,000	1,001	1,000	1,000	1,000	1,000
	1,004	1,004	1,003	1,004	1,003	1,004
	1,000	1,001	1,000	1,000	1,000	1,001
	0,656	**0,394**	**0,076**	**0,052**	**0,204**	**0,118**

Таблица 3

Результаты округления координат целевого векторного представления текстов для набора данных *twitter*

Исходное представление DBOW (размерность)	Целевое представление GloVe (размерность, округление)					
	50, 1	50, 3	50, 5	100, 1	100, 3	100, 5
50	0,988	1,000	1,000	0,993	1,000	1,000
	0,986	1,000	1,000	0,991	1,000	1,000
	0,988	1,000	1,000	0,993	1,000	1,000
	0,986	1,000	1,000	0,992	1,000	1,000
100	0,997	1,000	1,000	0,998	1,000	1,000
	0,996	1,000	1,000	0,996	1,000	1,000
	0,997	1,000	1,000	0,997	1,000	1,000
	0,996	1,000	1,000	0,995	1,000	1,000

для набора данных *twitter*. Округления проводили до 1, 3 и 5 цифр после запятой. Как и в предыдущих таблицах, строка таблицы соответствует исходному векторному представлению, а столбец соответствует целевому векторному представлению и параметру округления. Каждая ячейка содержит четыре числа: максимальное и среднее значения величины $e_{train}(\tau, \tau_n)$, а также максимальное и среднее значения величины $e_{test}(\tau, \tau_n)$. Как видно, округление до 3 и 5 цифр после запятой не изменило качество векторного представления текстов, а округление до 1 цифры слегка его ухудшило.

Отметим, что результаты округления других типов целевых векторных представлений для набора данных *twitter*, а также целевых векторных представлений для набора данных *movies* дали такие же результаты.

Заключение

Был представлен подход к формированию векторного представления текстов, согласованного с заданным векторным представлением слов. Качество векторного представления текстов определяется точностью решения задачи классификации текстов, которую можно достигнуть, используя это векторное представление.

В основу подхода была положена идея трансформации векторного представления слов, согласованного с заданным векторным представлением текстов. Подобная трансформация включает изменение модели и размерности векторного представления. Ее целью может служить желание использовать векторные представления, более адекватно описывающие предметную область, а также желание уменьшить их размерность, которая напрямую влияет на затраты по их хранению и обработке. Рассматриваемые трансформации векторных представлений реализуются в виде решения задачи восстановления многомерной регрессии.

Полученные в ходе проведения экспериментов результаты показали эффективность предложенного подхода. Качество построенных целевых векторных представлений текстов оказалось не хуже, а в большинстве случаев лучше исходных.

Был получен ряд экспериментальных результатов, касающихся точности построенных регрессий векторных представлений. Некоторые из них оказались неожиданными и требующими дальнейшего осмысления. Например, оказалось, что точность регрессии не зависит от размерности исходного векторного представления, а определяется только типом и размерностью целевого векторного представления.

Относительно невысокая точность регрессий навела на мысль, что координаты векторных представлений могут быть округлены. Это не приводит к ухудшению качества векторных представлений с точки зрения решения задачи классификации, однако позволяет значительно уменьшить их размерность.

Полученные результаты свидетельствуют о том, что методы решения задачи восстановления многомерной регрессии могут успешно использоваться для формирования векторных представлений текстов с улучшенными характеристиками. В настоящей работе предполагалось, что множество текстов фиксировано. Дальнейшее развитие предложенного подхода будет связано со случаем, когда исходное множество текстов может быть расширено. Таким образом, будет рассматриваться задача интерполяции векторного представления текстов.

Список литературы

1. **Mikolov T., Chen K., Corrado G., Dean J.** Efficient Estimation of Word Representations in Vector Space // Computing Research Repository (CoRR) — 2013. P. 1—12. URL: https://arxiv.org/abs/1301.3781 (дата обращения 28.02.2019).

2. **Mikolov T., Sutskever I., Chen K., Corrado G., Dean J.** Distributed representations of words and phrases and their compositionality // Proceedings of the 26th International Conference on Neural Information Processing Systems. 2013. Vol. 2. P. 3111—3119.

3. **Pennington J., Socher R., Manning C. D.** GloVe: Global Vectors for Word Representation // Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP). 2014. P. 1532—1544.

4. **Sebastiani F.** Machine learning in automated text categorization // ACM Computing Surveys. 2002. Vol. 34, N 1. P. 1—47.
5. **Kim Y.** Convolutional Neural Networks for Sentence Classification // Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP). 2014. P. 1746—1751.
6. **Le Q., Mikolov T.** Distributed Representations of Sentences and Documents // Proceedings of the 31st International Conference on Machine Learning. 2014. Vol. 32, N. 2. P. 1188—1196.
7. **Mu J., Bhat S., Viswanath P.** All-but-the-Top: Simple and Effective Post-processing for Word Representations // Computing Research Repository (CoRR). 2018. P. 1—25. URL: https://arxiv.org/abs/1702.01417 (дата обращения 28.02.2019).
8. **Bishop C. M.** Pattern Recognition and Machine Learning. Springer Science + Business Media LLC, 2006. 738 p.
9. **Harris Z.** Distributional structure // Word. 1954. Vol. 10, N. 23. P. 146—162.
10. **Rong X.** word2vec Parameter Learning Explained // Computing Research Repository (CoRR). 2016. P. 1—21. URL: https://arxiv.org/abs/1411.2738 (дата обращения 28.02.2019).

On Changing the Dimension of the Document Embeddings

A. S. Shundeev, alex.shundeev@gmail.com, Lomonosov Moscow State University, Moscow, 119192, Russian Federation

Corresponding author:

Shundeev Aleksander S., Leading Researcher, Lomonosov Moscow State University, Moscow, 119192, Russian Federation
E-mail: alex.shundeev@gmail.com

Received on March 03, 2019
Accepted on April 04, 2019

Currently, data mining is the basis for building a wide range of information systems. A modern and rapidly developing approach to the analysis of textual data is the construction and use of the word and document embeddings. Such embeddings were originally applied for the word similarity task and the word analogies task. However, they turned out to be in demand also in the text classification task. From this point of view, the word and document embeddings are investigated in this paper.

An approach based on the word embeddings transformations is described. In these transformations the model and dimension of the word embeddings are changed. The document embeddings may be associated with the word embeddings. In this case, the transformations considered can be extended to the document embeddings. For this purpose, multidimensional regression methods are used.

To confirm the proposed approach, experiments on two test datasets were performed. The first data set contained movie reviews related to one of six genres. The second data set contained twitter messages, each of which was negative or positive. The initial Doc2Vec (DBOW) document embeddings of dimensions 50, 100, 200, 300 were built. Also the Word2Vec (CBOW, SG) and GloVe word embeddings of dimensions 50, 100, 200 were built. The experiments performed on these datasets showed the following result. The document embeddings constructed using the proposed method may have a smaller dimension. Moreover, their use in the considered text classification tasks in most cases gives a more accurate result than when using the original document embeddings.

Keywords: *word embeddings, document embeddings, text classification, regression*

For citation:

Shundeev A. S. On Changing the Dimension of the Document Embeddings, *Programmnaya Ingeneria*, 2019, vol. 10, no. 6, pp. 265—273

DOI: 10.17587/prin.10.265-273

References

1. **Mikolov T., Chen K., Corrado G., Dean J.** Efficient Estimation of Word Representations in Vector, *Computing Research Repository (CoRR)*, 2013, pp. 1—12, available at: https://arxiv.org/abs/1301.3781.
2. **Mikolov T., Sutskever I., Chen K., Corrado G., Dean J.** Distributed representations of words and phrases and their compositionality, *Proceedings of the 26th International Conference on Neural Information Processing Systems*, 2013, vol. 2, pp. 3111—3119.
3. **Pennington J., Socher R., Manning C. D.** GloVe: Global Vectors for Word Representation, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532—1544.
4. **Sebastiani F.** Machine learning in automated text categorization, *ACM Computing Surveys*, 2002, vol. 34, no. 1, pp. 1—47.
5. **Kim Y.** Convolutional Neural Networks for Sentence Classification, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1746—1751.
6. **Le Q., Mikolov T.** Distributed Representations of Sentences and Documents, *Proceedings of the 31st International Conference on Machine Learning*, 2014, vol. 32, no. 2, pp. 1188—1196.
7. **Mu J., Bhat S., Viswanath P.** All-but-the-Top: Simple and Effective Post-processing for Word Representations, *Computing Research Repository (CoRR)*, 2018, pp. 1—25, available at: https://arxiv.org/abs/1702.01417.
8. **Bishop C. M.** *Pattern Recognition and Machine Learning*, Springer, Science + Business Media LLC, 2006, 738 p.
9. **Harris Z.** Distributional structure, *Word*, 1954, vol. 10, no. 23, pp. 146—162.
10. **Rong X.** word2vec Parameter Learning Explained, *Computing Research Repository (CoRR)*, 2016, pp. 1—21, available at: https://arxiv.org/abs/1411.2738.

УДК 004.4’2 DOI: 10.17587/prin.10.274-280

К. В. Рочев, канд. экон. наук, доц. кафедры, e-mail: k@rochev.ru,
Ухтинский государственный технический университет

Анализ быстродействия строковых операций языка C# на разных платформах

Настоящая работа посвящена анализу результатов измерений быстродействия операций и часто используемых функций языка C# в разных окружениях, таких как WPF, Windows Forms, Unity. Более подробно анализируется быстродействие некоторых функций работы со строками. Рассмотрены несколько версий инструментальных средств разработки и запуска кода, включая Mono, Core и традиционный .NET Framework, чтобы показать, есть ли разница в скорости выполнения тех или иных функций.

Ключевые слова: *быстродействие, оптимизация кода, производительность, C#, WPF, Windows Forms, DOT.NET Framework, Unity, Core*

Введение

Вычислительные возможности современных ЭВМ в настоящее время достаточно велики. Тем не менее ресурсы, которые необходимы для функционирования современных крупных, в том числе важных программных продуктов, постоянно растут. Для увеличения производительности такого программного обеспечения используются различные методы. Наиболее распространенными из них являются:

1) оптимизация кода средствами разработки, применяемая на разных фазах компиляции [1];

2) механизмы распараллеливания задач для их выполнения на нескольких ядрах вычислительной системы одновременно [2];

3) профилирование быстродействия программного кода (скорости выполнения) на основе таких видов инструментальных средств, как (*a*) встроенные в средства разработки, (*b*) поставляющиеся отдельно или (*c*) собственной реализации под конкретную задачу [3];

4) написание эффективных алгоритмов, правильное использование структур данных и функций над ними, которое зачастую требует наибольших знаний и предварительных исследований.

Настоящая статья посвящена методам оптимизации программного обеспечения, обозначенным в пп. 3 и 4 представленного списка. Анализируется быстродействие часто используемых функций, связанных с обработкой строковых данных. Анализ проводится на примере языка C# в разных окружениях, таких как WPF (Windows Presentation Foundation), Windows Forms, Unity. Рассмотрены несколько версий средств разработки и запуска кода (далее — фреймворк) и видов выполняемого проекта, чтобы оценить различие в скорости выполнения тех или иных функций. Конечной целью является более эффективное, с учетом таких оценок, построение алгоритмов на участках кода с большой частотой вызовов.

Для измерения производительности участков кода автором реализована компактная библиотека классов, которая подключается к разным средам выполнения [4].

Методы профилирования

Основные функциональные возможности библиотеки, реализованной для измерения скорости выполнения (быстродействия) приложений:

1) поток, постоянно выполняющий переданную в библиотеку для анализа быстродействия функцию из ее делегата (ссылки на функцию), что позволяет частично обойти оптимизацию повторяющихся операций механизмами .NET Framework;

2) функция измерения быстродействия кода, принимающая делегат — указатель на участок изучаемого кода, эта функция измеряет число выполнений этого кода потоком в течение 1 мс;

3) функция подсчета, которая накапливает результаты одинаковых измерений для последующего устранения пиковых результатов, усреднения, вычисления медианного значения;

4) механизм минимизации вызова сборщика мусора и подсчета частоты его вызовов во время тестов.

Приложение с простейшим интерфейсом позволяет скопировать результаты измерений в CSV-формате в программе Excel и там их обработать (рис. 1). Аналогичные формы или прямая запись в CSV-файлы реализованы и для других типов проектов (Unity, ASP.NET Core, Widows Forms).

Профилирование программного кода на предмет скорости его выполнения реализуется по следующему алгоритму.

1. Проводится несколько измерений, например, десять. В каждом из них изучаемый участок кода запускается в бесконечном цикле в течение определенного времени, в данном исследовании — по 1 мс.

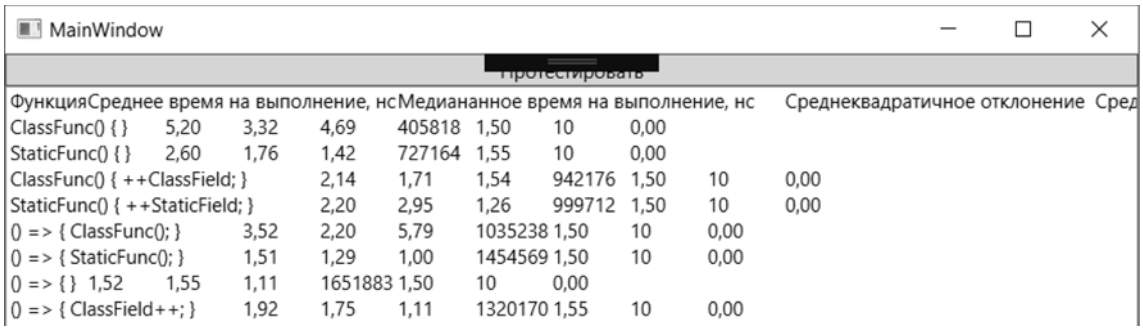

Рис. 1. Результат измерения в простом окне WPF для последующего копирования в Excel

Подсчитывается число запусков за эту миллисекунду и путем деления времени тестирования на это число определяется время выполнения изучаемого участка кода. Как следует из данных рис. 1, а также подтверждается данными табл. 1, время выполнения теста примерно на 0,5 мс больше установленного минимального времени вследствие ожидания переключения потока.

2. Рассчитывается среднее и медианное время выполнения кода по всем измерениям. При этом далее для анализа использовано медианное значение, так как на него не влияют так называемые "выбросы" — экстремальные значения, например, при первом измерении или при переключении процессора на другие потоки.

3. Дополнительно вычисляется среднее квадратичное отклонение, чтобы можно было оценить разброс результатов измерений. Подсчитывается число срабатываний сборщика мусора, чтобы отследить и нивелировать его влияние на результаты измерений.

4. Все эти данные сохраняются в CSV-формате (см. рис. 1) для дальнейшей обработки, например, в программе Excel.

Результаты измерения быстродействия по рассмотренным группам операций оценивались относительно WPF по формуле

$$X = \frac{\sum S_t \cdot 2}{\sum S_t + S_{WPF}}, \qquad (1)$$

где X — относительное быстродействие; S_t — результат на рассматриваемой платформе; S_{WPF} — результат на WPF.

Результаты измерений

Измерение проводили на компьютере ASUS X556UQ: i7-7500U, 2.7 GHz, 20 Гбайт ОЗУ, Windows 10×64. Для оценки быстродействия реализованного тестового окружения (насколько велики накладные расходы на вызов функций по ссылкам) были выбраны такие операции, как обращение к функциям, к полям и свойствам класса (табл. 1); а именно:

- () => { } — выполнение пустой функции;
- () => { ++localInt; } — прибавление единицы к переменной, расположенной на стэке функции, из которой вызывается запуск измерений;
- () => { ++ClassIntField; } — прибавление единицы к полю класса;
- () => { ++StaticIntField; } — прибавление единицы к статическому полю класса;
- () => { ++ClassIntProperty; } — прибавление единицы к статическому свойству класса (свойства C# отличаются от полей тем, что могут содержать обращения к функциям во время вызова);
- () => { ++StaticIntProperty; } — прибавление единицы к статическому свойству класса;
- StaticFunc() { } — вызов статической функции напрямую.

По результатам измерений (табл. 1 и 2) быстродействие тестовой инфраструктуры примерно сопо-

Таблица 1

Результаты тестовых измерений на примере проекта WPF в Release-режиме (десять тестов при среднем числе сборки мусора на тест 0)

Функция	Среднее время на выполнение, нс	Медианное время на выполнение, нс	Среднее квадратичное отклонение	Среднее число запусков за тест	Среднее время теста, мс
() => { }	0,61	0,54	0,43	3 381 908	1,60
() => { ++localInt; }	1,7	2,58	1,18	2 409 092	1,47
() => { ++ClassIntField; }	1,02	0,33	1,10	3 631 160	1,49
() => { ++StaticIntField; }	1,58	1,46	0,99	2 119 760	1,50
() => { ++ClassIntProperty; }	1,45	1,30	0,89	1 863 567	1,55
() => { ++StaticIntProperty; }	1,98	1,02	1,92	1 944 740	1,46
StaticFunc() { }	1,22	1,12	0,48	1 268 418	1,45

Таблица 2

Результаты тестовых измерений в среде WPF в разных режимах выполнения

Функция	WPF Debug с отладчиком, нс	WPF Debug без отладчика, нс	WPF Release с отладчиком, нс	WPF Release без отладчика, нс
() => { }	4,08	3,15	3,07	1,64
() => { ++LocalInt; }	**16,20**	4,28	2,12	1,51
() => { ++ClassIntField; }	5,95	3,77	2,33	1,55
() => { ++StaticIntField; }	4,52	3,09	2,16	1,36
() => { ++ClassIntProperty; }	**20,18**	**7,80**	1,98	1,61
() => { ++StaticIntProperty; }	**11,46**	**7,47**	1,80	1,49

ставимо с обращением к переменной или к пустой функции. Таким образом, можно считать, что тестовая инфраструктура не оказывает существенного воздействия на результаты измерений и может быть применена для дальнейшего профилирования.

Измерения скорости выполнения операций проводили в различных режимах сборки. К их числу относится фаза отладка в Debug-режиме, а также фаза выпуска приложения в режиме Release. Это важно, поскольку во второй фазе компиляции при создании Release-приложения используются дополнительные механизмы оптимизации [3]. Приведем результаты измерений в разных режимах выполнения сборки (см. табл. 2).

Как можно заметить, элементарные операции довольно существенно оптимизируются при переводе проекта в фазу выпуска — Release. Устраняются лишние сложности вызова свойств классов, и они начинают работать также быстро, как обычные поля, удаляются вызовы пустых функций. В табл. 2 видно, что в фазе отладки обращения к свойствам класса выполняются в 4—10 раз медленнее, чем в фазе выпуска, также это можно заметить и при обращении к локальной переменной, что обусловлено обращением к последней из функции-делегата. Расхождения в результатах измерений быстродействия кода становятся обусловлены в большей степени случайными флуктуациями. При этом оптимизатор настолько хорош, что можно заметить постепенное ускорение быстродействия для похожих операций, несмотря на их вызов на основе применения делегатов.

Далее, для сопоставления результатов оценки быстродействия кода на разных фреймворках, рассмотрим наиболее актуальный для них режим запуска — Release. В процессе исследования быстродействия базовых операций на разных фреймворках были проанализированы такие возможности, как обращения к полям, к свойствам классов и математические операции — арифметические операции, разные способы округления, тригонометрические функции, возведение в степень, логарифм и др. [4]. Средства WPF и Windows Forms показывают примерно одинаковые результаты. В среднем по рассмотренным операциям Windows Forms медленнее на 6 %, что вполне может быть обусловлено погрешностями измерений. Причина в том, что обе платформы реализованы на классическом .NET. Заметим, что в Debug-режиме разница между ними более существенная. В то же время на Unity некоторые операции выполняются с существенной разницей в скорости ввиду того, что основаны на MONO-Framework (Unity, в среднем, медленнее на 220 %). В .NET Core-реализации заметна не менее ощутимая разница в быстродействии как в большую, так и в меньшую сторону по разным операциям (в среднем по рассмотренным функциям на 77 % медленнее). Однако следует отметить, что выборка отдельных функций не является достаточно презентабельной и не дает оснований судить о производительности того или иного фреймворка в целом.

Далее перейдем непосредственно к оценке быстродействия при работе со строками.

Результаты оценки быстродействия при работе со строками

В табл. 3—6 представлены результаты измерений быстродействия выполнения строковых операций. Рассмотрим результаты измерений в Release-режиме для WPF, Windows Forms, Unity и ASP.NET Core. Рассмотрим также отношение результатов оценок быстродействия для кода Unity и Core применительно к WPF с учетом формулы (1).

В табл. 3 представлены результаты измерений быстродействия функций преобразования целых чисел (10.ToString()), чисел с плавающей (10f.ToString()) и фиксированной (10m.ToString()) запятой, а также функций получения даты и перевода ее в строку разными способами (DateTime.Now.ToString()).

Можно заметить, что для дробных чисел быстрее работает преобразование в строку в классическом DOT.NET, а для целочисленных — в DOT.NET Core. Реализация Mono в Unity работает с той же скоростью, что и классическая, за исключением операции свободного форматирования ToString("##.###"), которая в Unity медленнее в 4 раза по сравнению с WPF. Форматирование даты быстрее работает в DOT.NET Core и медленнее в Mono. В целом получение и форматирование даты уступает преобразованию числа в строку лишь в 3...9 раз.

На рис. 2 эти результаты измерений представлены в виде диаграммы, из которой наглядно видна разница в производительности одного и того же кода в разных фреймворках.

Таблица 3

Результаты измерений быстродействия преобразования в строку

Функция	WPF, нс	Windows Forms, нс	Unity, нс	Core, нс	Unity/WPF, %	Core/WPF, %
() => ClassStr = DateTime.Now.ToString("F")	932,8	920,8	1822,7	572,9	132	76
() => ClassStr = DateTime.Now.ToString()	871,9	874,9	1597,3	523,1	129	75
() => ClassStr = DateTime.Now.ToString("yyyy.MM.dd")	666,2	696,7	1308,6	354,5	133	69
() => 10.1234567890.ToString("##.###")	285,5	237,3	1035,0	598,0	157	135
() => 10.1234567890d.ToString()	221,1	223,8	213,8	585,5	98	145
() => 10.1234567890.ToString("F2")	215,2	201,3	220,2	553,5	101	144
() => 10d.ToString()	210,8	199,5	173,0	285,5	90	115
() => 10.1234567890f.ToString()	209,3	190,3	193,9	408,5	96	132
() => 10f.ToString()	201,8	194,6	162,0	234,4	89	107
() => 100000.ToString()	131,0	114,6	140,5	59,6	103	63
() => 10m.ToString()	129,8	103,0	202,8	81,2	122	77
() => 0.ToString()	123,2	95,6	57,4	57,3	64	63
() => 10.ToString()	100,3	86,4	127,5	53,1	112	69

Таблица 4

Результаты измерений быстродействия различных способов конкатенации строк

Функция	WPF, нс	Windows Forms, нс	Unity, нс	Core, нс	Unity/WPF, %	Core/WPF, %
() => ClassStr = $"S1{++ClassField} S2{++Class ... Field}S6"	1449,1	1323,3	1580,0	1935,7	104	114
() => ClassStr = "S1" + ++Class ... Field + "S6"	1323,9	1227,8	1607,2	1728,8	110	113
() => ClassStr += "S1"	564,9	550,1	915,0	1771,5	124	152
() => ClassStr = String.Format("S1{0}S2", ++ClassField)	420,3	405,0	435,1	424,3	102	100
() => ClassStr = $"S1{++ClassField}S2"	311,1	290,5	364,1	450,1	108	118
() => ClassStr = "S1" + ++ClassField + "S2"	265,1	240,2	327,3	347,4	110	113
() => ClassStr = String.Join(", ", StrArr10)	182,0	183,3	292,8	125,4	123	82
() => ClassStringBuilder.Append("S1")	6,6	6,5	9,4	6,4	118	99

Далее рассмотрим операции сложения строк.

В табл. 4 представлены результаты профилирования следующих функций над строками:

1) конкатенация — сложение строк с помощью оператора "+", например, сложение двух постоянных строк с меняющимся целым значением: () => ClassStr = "S1" + ++ClassField + "S2" (здесь можно отметить, что половину времени занимает преобразование числа в строку); () => ClassStr = "S1" + ++Class... Field + "S6" — аналогично для сложения строки из шести постоянных и пяти меняющихся частей;

2) интерполяция — особая форма записи строк в языке C# в формате $"текст {ВставляемоеЗначение} текст" — аналогично для двух и пяти переменных, вставляемых в строку;

3) String.Format — специальная функция, форматирующая строку по заданному шаблону, — "предшественник" интерполяции;

4) накопительное сложение строк с помощью конкатенации и специального класса обработки строк StringBuilder — здесь строки складываются в одну в течение всего времени измерения (несколько десятков тысяч раз): () => ClassStr += "S1" и () => ClassStringBuilder.Append("S1").

Операции сложения строк в целом довольно тяжеловесны, также как и преобразование чисел в строки. Как можно заметить, функция String.Format уступает по скорости интерполяции строк примерно на 15 %. Интерполяция, в свою очередь, уступает конкатенации примерно на столько же. Функция String.Join, естественно, опережает обе из перечисленных функций (см. табл. 4 и рис. 3).

Добавление текста в строку через класс StringBuilder работает в тестовых условиях примерно в 100 раз быстрее, чем через оператор +=. Соответственно, при увеличении размеров складываемых строк этот отрыв будет еще больше, ввиду чего StringBuilder часто указывают как рекомендуемый способ работы с большими строковыми данными, так как он не создает лишних копий текстовых данных, а оперирует с массивом символов напрямую.

Таблица 5

Результаты измерений быстродействия сравнения строк и поиска подстроки

Функция	WPF, нс	Windows Forms, нс	Unity, нс	Core, нс	Unity/WPF, %	Core/WPF, %
() => ClassBool = Str100.Contains('-')	663,4	718,2	1049,0	925,3	123	116
() => ClassBool = Str10.Contains('-')	101,9	116,7	155,8	126,4	121	111
() => ClassBool = Str1.Contains("123")	75,3	66,5	34,4	19,7	63	41
() => ClassBool = Str10.Contains("321")	62,3	55,3	38,9	28,8	77	63
() => ClassBool = Str10.Contains("123")	59,4	53,6	39,8	20,7	80	52
() => ClassBool = Str100.Contains("123")	56,9	58,3	39,8	21,4	82	55
() => ClassBool = Str10.Contains('1')	41,0	39,2	58,5	33,6	118	90
() => ClassBool = Str100.Contains('1')	39,9	39,6	56,5	34,2	117	92
() => ClassBool = Str1 == "123"	3,2	3,5	7,5	4,6	141	118
() => ClassBool = Str10.Equals("123")	3,0	3,2	3,4	4,5	106	121
() => ClassBool = String.IsNullOrEmpty(Str10)	1,4	2,3	4,8	2,6	154	129
() => ClassBool = String.IsNullOrWhiteSpace(Str10)	1,1	1,6	9,4	6,7	178	171

Таблица 6

Результаты измерений некоторых других строковых операций

Функция	WPF, нс	Windows Forms, нс	Unity, нс	Core, нс	Unity/WPF, %	Core/WPF, %
() => ClassStr = Str100.Substring(5)	82,5	84,1	144,0	32,5	127	57
() => ClassStr = Str10.Substring(5)	18,4	18,4	63,8	15,7	155	92
() => ClassStr = Str10.Trim()	12,3	13,8	20,5	11,1	125	95
() => ClassStr = Str100.Trim()	10,6	9,5	18,5	12,3	127	107
() => ClassStr = "S1"	4,3	4,0	3,2	4,1	86	98
() => ClassStr = "S1" + "S2"	3,9	2,9	2,3	4,4	74	105

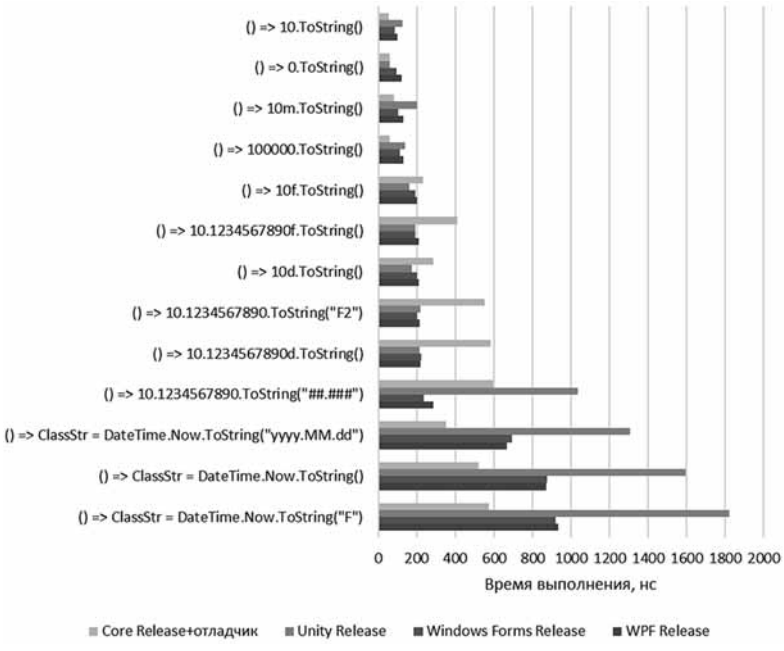

Рис. 2. Результаты измерений быстродействия преобразования в строку

Рассмотрим следующие функции: Contains — поиск вхождения подстроки в строку, операции сравнения "==" и Equals, функции проверки строки на незаполненность IsNullOrEmpty и IsNullOrWhiteSpace. Для измерения быстродействия поиска и преобразования строк будем присваивать результат поиска переменной класса логического типа и используем строки с разной длиной: Str1 = "1", Str10 = "1234567890", Str100 = "1234567890123...90" — до длины в 100 символов (см. табл. 5 и 6).

Поиск в строках осуществляется в целом быстрее, чем операции по их модификации и примерно с одной скоростью на разных платформах. К положительным моментам следует отнести тот факт, что проверки на null проходят весьма оперативно, так как эти операции в практике программирования применяются весьма часто. Хотя необходимо отметить, что в Unity IsNullOrWhiteSpace по непонятным пока причинам такая проверка работает в 9 раз медленнее. Функция Equals выполняется немного быстрее, чем оператор сравнения. Хотя можно было бы

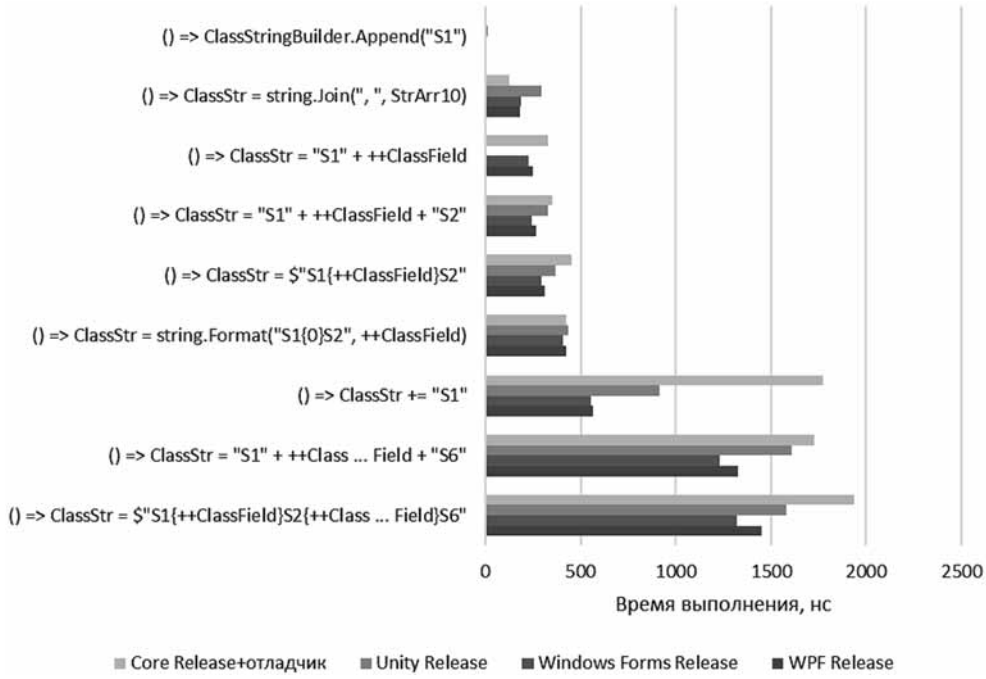

Рис. 3. Результаты измерений быстродействия различных способов конкатенации строк

предположить, что оптимизатор второй фазы должен был бы убрать эти различия.

Рассмотрим также результаты по следующим отдельным операциям (см. табл. 6):

- присвоение (() => ClassStr = "S1", время этого измерения можно вычесть из результатов всех остальных для увеличения точности и разброса в оценке разницы скорости выполнения, так как присвоение применяется во всех измерениях с целью избежать оптимизаций со стороны компилятора);
- получение подстроки (Substring);
- обрезка строк (Trim).

Операции сложения константных строк оптимизируются всеми компиляторами. Операции получения подстроки и обрезки выполняются довольно быстро. При этом получение подстроки для тестового случая в Unity медленнее в 3 раза, чем в других версиях фреймворка.

Заключение

При сравнении результатов быстродействия кода WPF и WindowsForms в Release-режиме получено, что средняя разница быстродействия операций по разным группам составляет до 10 %. Это может быть обусловлено погрешностями измерений. В целом все операции выполняются примерно с одинаковой скоростью, что неудивительно, так как используется единый фреймворк.

Что касается сравнения с Unity и .NET Core, то фреймворки уже различны, поэтому и отличия в быстродействии операций более существенные. Вызовы пустых функций и обращения к переменным в Unity выполняются в среднем на 43 % медленнее (за счет обращения к свойствам). Первоначальные измерения показывали, что работа со строками в Unity происходит медленнее, чем в WPF в 100 раз (или на 98 % по формуле (1)). Однако после минимизации вызовов сборщика мусора этот результат существенно улучшился. Тем не менее при относительно долгом функционировании сборка мусора в любом случае внесет свой вклад в быстродействие реальной программы.

В .NET Core базовые операции также, в среднем, на 43 % медленнее. Что касается кода выполнения строковых операций, то здесь среднее быстродействие такое же, что и в WPF. Однако быстродействие выполнения отдельной функции различается до 2...3 раз (наибольшие различия: IsNullOrWhiteSpace в WPF быстрее в 6 раз, а Contains в 3 раза медленнее).

Таким образом, можно заметить, что при близких результатах измерений по большинству рассмотренных операций даже в родственных средах разработки в отдельных случаях есть принципиальные различия в быстродействии часто используемых операций. Такие различия следует учитывать при написании программного обеспечения.

Список литературы

1. **Четверина О. А.** Повышение производительности кода при однофазной компиляции // Программирование. 2016. № 1. С. 51—59.

2. **Joisha P. G., Schreiber R. S., Banerjee P., Boehm H.-J., Chakrabarti D. R.** On a technique for transparently empowering classical compiler optimizations on multithreaded code // ACM Transactions on Programming Languages and Systems. 2012. Vol. 34, N. 2. P. 9:1—9:42.

3. **Дараган Е. И.** Система анализа производительности программного кода // Известия Тульского государственного университета. Технические науки. 2013. № 9—2. С. 89—94.

4. **Рочев К. В.** Анализ быстродействия типовых операций языка C# на платформах DOT.NET и Mono // Информационные технологии в управлении и экономике. 2019. № 1. С. 7—19.

Analysis of Performance of String Operations in C# on Different Platforms

K. V. Rochev, k@rochev.ru, Ukhta State Technical University, Ukhta, 169300, Russian Federation

Corresponding author:

Rochev Konstantin V., Associate Professor, Ukhta State Technical University, Ukhta, 169300, Russian Federation, E-mail: k@rochev.ru

Received on December 22, 2018
Accepted on April 04, 2019

This publication analyzes the results of performance measurements of operations and commonly used C# language functions in different environments, such as WPF, Windows Forms, Unity. The performance of some string functions is analyzed in more detail. Several versions of the code development and execution tools, including Mono, Core, and the traditional.NET Framework, are reviewed to see if there is a difference in the speed at which functions are executed. To measure the performance of code sections, the author has implemented a compact class library that connects to different execution environments. Measuring the execution speed of the operations was conducted in various modes of Assembly. These include the debug phase in DEBUG mode and the release phase of the application in RELEASE mode.

When comparing the results of WPF and Windows Forms code performance in release mode, it was found that the average difference in the performance of operations for different groups can be caused by measurement errors, which is not surprising, since a single framework is used. As for the comparison of WPF with Unity and .Net Core, the differences in the speed of operations are more significant. Calls to empty functions and variable calls in Unity are somewhat slower. And working with strings in Unity is even more inferior to the traditional version of the framework. In. Net Core, the average performance is the same as in WPF. However, the performance of single functions differs significantly.

It is shown that with close measurement results for most of the considered operations, even in related development environments in some cases there are fundamental differences in the performance of frequently used operations.

Keywords: *performance, code optimization, performance, C#, WPF, Windows Forms, DOT.NET Framework, Unity, Core*

For citation:

Rochev K. V. Analysis of Performance of String Operations in C# on Different Platforms, *Programmnaya Ingeneria,* 2019, vol. 10, no. 6, pp. 274—280.

DOI: 10.17587/prin.10.274-280

References

1. **Chetverina O. A.** Povyshenie proizvoditel'nosti koda pri odnofaznoj kompilyacii **(**Improving code performance during single-phase compilation), *Programmirovanie*, 2016, no. 1, pp. 51—59 (in Russian).

2. **Joisha P. G., Schreiber R. S., Banerjee P., Boehm H.-J., Chakrabarti D. R.** On a technique for transparently empowering classical compiler optimizations on multithreaded code, *ACM Transactions on Programming Languages and Systems*, 2012, vol. 34, no. 2, pp. 9:1—9:42.

3. **Daragan E. I.** Sistema analiza proizvoditel'nosti programmnogo koda (System performance analysis software code), *Izvestiya Tul'skogo gosudarstvennogo universiteta. Tekhnicheskie nauki*, 2013, no. 9—2, pp. 89—94 (in Russian).

4. **Rochev K. V.** Analiz bystrodejstviya tipovyh operacij yazyka C# na platformah DOT.NET i Mono (Analysis of the performance of typical operations of the language C# on platform DOT.NET and Mono), *Informacionnye tekhnologii v upravlenii i ehkonomike*, 2019, no. 1, pp. 7—19 (in Russian).

УДК 004.4; 004.67 DOI: 10.17587/prin.10.281-288

Д. Н. Кобзаренко[1, 2], д-р техн. наук, зав. лабораторией, e-mail: kobzarenko_dm@mail.ru,
А. М. Камилова[1], вед. специалист, e-mail: anna702@mail.ru,
Б. И. Шихсаидов[3], канд. техн. наук, проф., декан, e-mail: sbi707@yandex.ru,
[1] Институт проблем геотермии Дагестанского научного центра Российской академии наук, г. Махачкала,
[2] Дагестанский государственный университет народного хозяйства, г. Махачкала,
[3] Дагестанский государственный аграрный университет им. М. М. Джамбулатова, г. Махачкала

Средства автоматизации процесса построения гистограмм частотного распределения по результатам непрерывного вейвлет-преобразования с помощью функции morlet

Рассмотрен подход к автоматизации процесса построения гистограмм частотного распределения по результатам непрерывного вейвлет-преобразования с использованием функции morlet. Для этого разработано программное обеспечение, которое позволяет детализировать результаты частотно-временного анализа на основе вейвлет-преобразования применительно к метеорологическим временным рядам. Отличительной особенностью этой программы является то, что она позволяет компоновать и сопоставлять гистограммы частотного распределения из разных временных периодов и разных источников данных.

Ключевые слова: *частотно-временной анализ, программное обеспечение, гистограмма, непрерывное вейвлет-преобразование, вейвлет-функция morlet, метеорологический временной ряд*

Введение

Непрерывное вейвлет-преобразование одномерного временного ряда состоит в его разложении по базису, сконструированному из обладающей определенными свойствами функции — вейвлета, путем масштабных изменений и переносов. Каждая из функций этого базиса характеризует как определенную пространственную (временную) частоту, так и ее локализацию в физическом пространстве (времени) [1].

Результатом непрерывного вейвлет-преобразования произвольного временного ряда $f(t)$ будет функция $W(a, t)$, которая зависит от двух переменных: от времени t и от масштаба a. Значения $W(a, t)$ вычисляют следующим образом.

- Вейвлет Ψ растягивается в a раз по горизонтали и в $1/a$ раз по вертикали.
- Вейвлет Ψ сдвигается в определенную временную точку t. Полученный вейвлет обозначим как $\Psi_{a, t}$.
- "Усредняется" значение сигнала в окрестности точки t с помощью $\Psi_{a, t}$.

Схематически процесс вычислений показан на рис. 1. На этом рисунке изображен график функции $f(t)$. Столбиками изображены графики вейвлет-функций $\Psi_{a, t}$ при разных значениях a. Выделенные участки графика исходной функции поточечно умножаются на значения растянутой и смещенной относительно временной оси вейвлет-функции, потом все это суммируется. Такие действия выполняются для всех пар (a, t).

Формула вейвлет-преобразования имеет следующий вид:

$$W(a, t) = \frac{1}{\sqrt{a}} \int_{-\infty}^{\infty} f(t)\psi\left[\frac{t-x}{a}\right]dt, \qquad (1)$$

где $W(a, t)$ — коэффициенты вейвлет-преобразования; $f(t)$ — исследуемая функция; t — время;

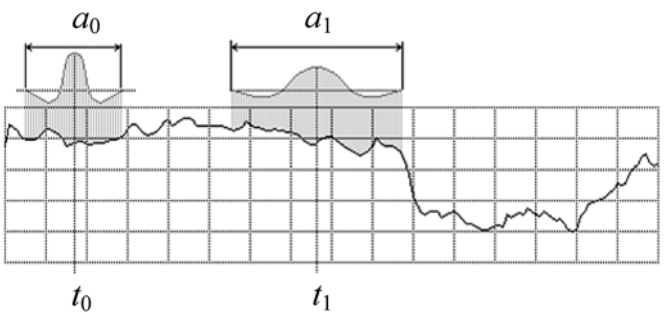

Рис. 1. Схема процесса вейвлет-преобразования

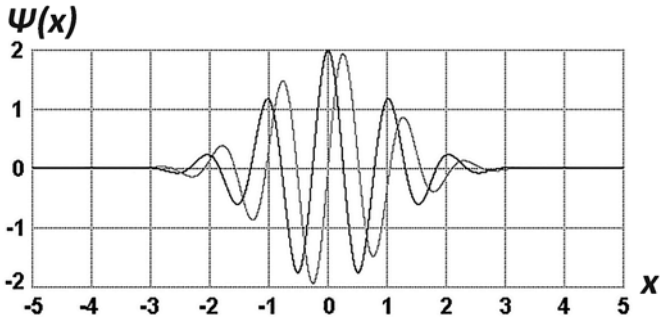

Рис. 2. Комплексная вейвлет-функция morlet: вещественная (черная кривая) и мнимая (серая кривая) части

ψ — вейвлет; a — величина масштаба; x — параметр сдвига по оси t.

Выбор анализирующего вейвлета определяется тем обстоятельством, какую информацию необходимо извлечь из сигнала. Каждый вейвлет имеет характерные особенности во временном и в частотном пространствах. Поэтому иногда с помощью разных вейвлетов можно полнее выявить и подчеркнуть те или иные свойства анализируемого сигнала. В настоящей работе рассматривается вейвлет-функция morlet (рис. 2).

Вейвлет morlet комплексный, он хорошо приспособлен для анализа сигналов, для которых важен принцип причинности. Этот вейвлет сохраняет направление времени и не создает паразитной интерференции между прошлым и будущим. При использовании такого анализирующего вейвлета в результате вейвлет-преобразования получаются два двумерных массива значений модуля и фазы. В контексте настоящей статьи представляет интерес значение модуля, которое позволяет показать наличие периодичностей во временном ряде и их динамику и вычисляется по формуле

$$W(a, t) = \sqrt{W_{\mathrm{Re}}(a, t)^2 + W_{\mathrm{Im}}(a, t)^2}, \qquad (2)$$

где $W_{\mathrm{Re}}(a, t)$ — вейвлет-преобразование функцией действительной части комплексного вейвлета morlet; $W_{\mathrm{Im}}(a, t)$ — вейвлет-преобразование функцией мнимой части комплексного вейвлета morlet.

Рис. 3 демонстрирует результаты вычисления модуля вейвлет-преобразования вейвлетом morlet для модельной функции. На рис. 3, *а* показан переход от синусоиды с периодом 0,5 с к синусоиде с постепенным уменьшением периода; на рис. 3, *б* — переход от синусоиды с постепенным уменьшением периода к сумме двух гармоник с периодами 0,5 и 0,02 с.

Исследования по применению непрерывного вейвлет-преобразования для анализа временных метеорологических рядов известны как в отечественных [2, 3], так и в зарубежных [4] работах. В отечественных работах непрерывное вейвлет-преобразование рассматривается как инструмент прогнозирования и восстановления пропущенных данных.

С позиции исследований [5] авторов настоящей статьи в области возобновляемых ветроэнергетических энергоресурсов, инструментальные средства непрерывного вейвлет-преобразования позволяют выполнять пространственно-временной анализ закономерностей во временных рядах, которые представляют скорости и направления ветра в регионе. Анализ выполняется как по пространству (сопоставление данных разных территорий для одного временного периода), так и по времени (сопоставление различных временных периодов для одной территории).

В работе [5] показано, что на основе частотно-временного анализа временных рядов — скоростей ветра с помощью непрерывного вейвлет-преобразования вейвлет-функцией morlet можно сделать обобщенные выводы о пространственных и временных закономерностях. Для это используется визуализация матрицы коэффициентов $W(a, t)$, где по оси x откладывается время t, а по оси y — откладывается периодичность a.

Визуализация матрицы $W(a, t)$ в виде двумерной картины цветовых оттенков (рис. 3) дает первичную информацию о частотной динамике входного сигнала. Однако такое представление полученных результатов непрерывного вейвлет-преобразования не совсем полное и может быть расширено за счет инструментальных средств построения гистограмм частотного распределения. Процесс построения гистограмм частотного распределения на основе матриц $W(a, t)$ может быть автоматизирован созданием дополнительного программного обеспечения.

Постановка задачи

Целью разработки, которая рассматривается в настоящей статье, является создание средств автоматизации процесса построения гистограмм частотного распределения по результатам непрерывного вейвлет-преобразования с использованием функции morlet. Такие средства призваны обеспечить возможность количественно оценить и сопоставить результаты непрерывного вейвлет-преобразования с помощью функции morlet для метеорологических временных рядов.

В задачи разработки целевых средств автоматизации входят обеспечение возможностей:

— загрузки двумерных массивов $W(a, t)$ — результатов непрерывного вейвлет-преобразования функцией morlet;

— задания диапазона интервала периодичностей;

— компоновки данных для графика на основе различных источников и различных временных периодов;

— одновременного вывода несколько гистограмм в рамках одного графика;

— сохранения полученных результатов.

Перед тем как определить входные данные и параметры решения поставленной задачи, опишем исходные данные, с которыми ведется работа. Они представляют собой метеорологические временные ряды формата время-значение (в нашем случае значения — это скорости и направления ветра с восемью измерениями в сутках: 00:00, 03:00, 06:00, 09:00, 12:00, 15:00, 18:00, 21:00). Начало временного ряда соответствует нулевому отсчету. Временная шкала

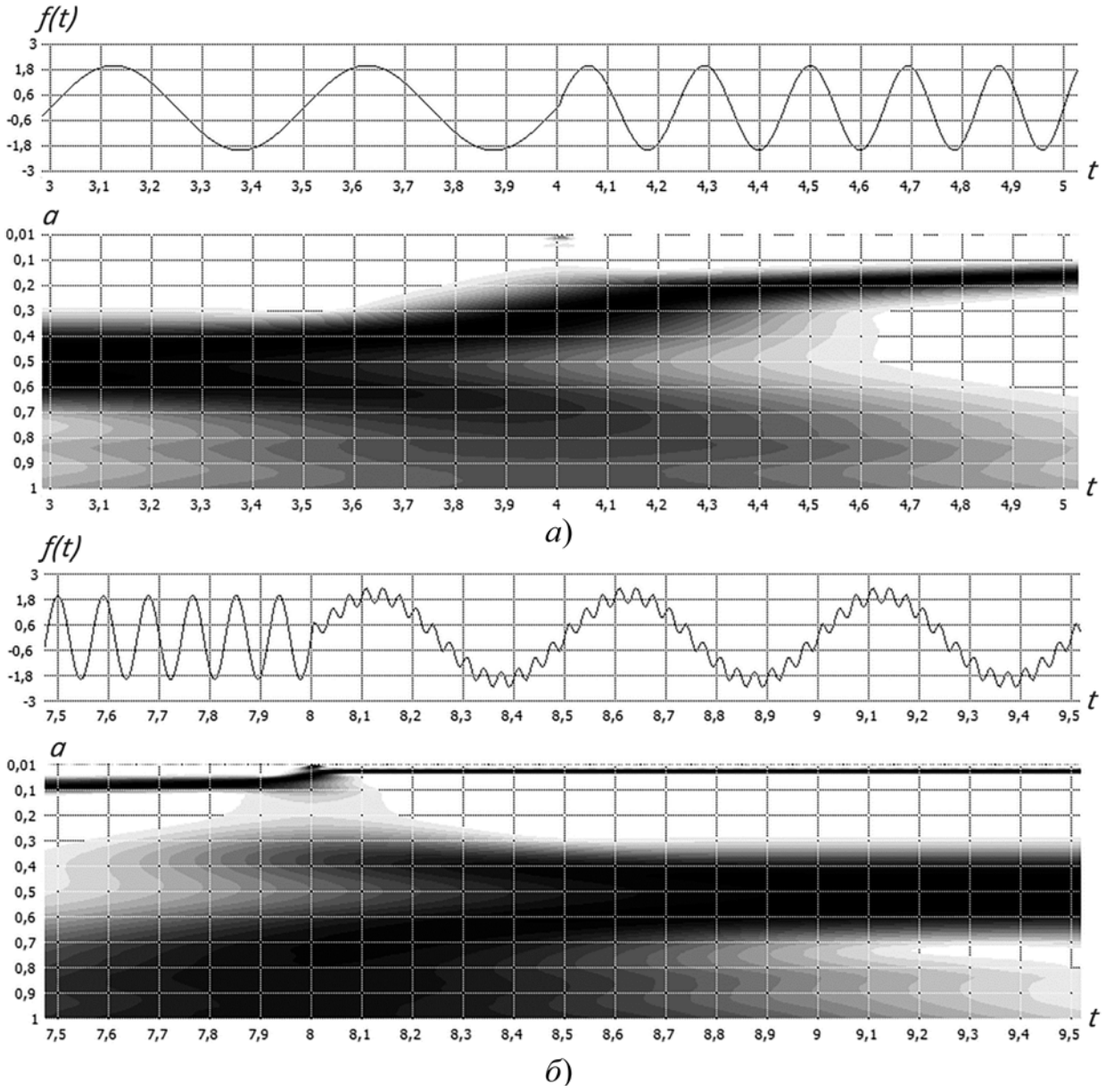

Рис. 3. Фрагменты картины коэффициентов модуля вейвлет-преобразования функцией morlet для модельного сигнала

задается в сутках. Шаг между значениями принимается равным 0,125 суток.

Определим входные данные и параметры. Входные данные представляют собой множество матриц $W(a, t)$ — результатов вейвлет-преобразования с использованием функции morlet. Каждой матрице ставятся в соответствие следующие параметры:

1) a_{min} — минимальное значение периода/частоты;
2) a_{max} — максимальное значение периода/частоты;
3) a_{step} — шаг между значениями периода/частоты;
4) t_{min} — начальный отсчет временного интервала;
5) t_{max} — конечный отсчет временного интервала;
6) t_{step} — шаг между значениями временного интервала;
7) название временного ряда (например, "скорость ветра");
8) название источника данных временного ряда (например, название метеостанции);
9) число измерений в сутках.

Для того чтобы идентифицировать во временной шкале интервалы, соответствующие определенному календарному месяцу, вводится массив идентификаторов типа "номер месяца" — "номер года" — "временной отсчет начала интервала в сутках".

Для исследований метеорологического временного ряда представляет интерес идентифицировать временные отсчеты по времени суток (дневное и ночное). Для этого вводятся параметры — массивы временных отсчетов дневного и ночного времени суток. Например, для случая, когда имеется восемь измерений в сутках, получаются следующие временные отсчеты по кругу от 0 до 7: 0 — 00:00 (ночь);

1 — 03:00 (ночь); 2 — 06:00 (ночь); 3 — 09:00 (день); 4 — 12:00 (день); 5 — 15:00 (день); 6 — 18:00 (день); 7 — 21:00 (ночь).

Определим последовательность действий (алгоритм) решения задачи.

Шаг 1. Задание пути к каталогу входных данных.

Шаг 2. Считывание матриц $W(a, t)$ и разбивка их в иерархическую структуру данных в оперативной памяти (описывается ниже) по интервалам.

Шаг 3. Задание диапазона периодичностей.

Шаг 4. Выборка данных для расчета гистограммы из опций: источник, год, месяц, время суток.

Шаг 5. Расчет гистограммы и вывод результата.

Шаг 6. Добавление гистограмм по шагам 4, 5.

Формат входных данных

Разработанные форматы входных данных представляют собой пары файлов с одинаковым именем и расширениями .WTR и .TXT, которые собираются в рамках одного каталога.

В файле с расширением .WTR содержится матрица вейвлет-преобразования вейвлетом morlet $W(a, t)$. Файл .WTR имеет двоичный формат в виде массива восьмибайтовых вещественных чисел. Структура файла приведена в таблице.

Структура формата двоичного файла .WTR

Название поля	Размер, байт	Описание
Start_A	8	Начальное значение периода
Finish_A	8	Конечное значение периода
Count_A	8	Размер матрицы по масштабам
Step_A	8	Шаг между значениями периодов
Start_Time	8	Начальное значение времени
Finish_Time	8	Конечное значение времени
Count_Time	8	Размер матрицы по времени
Step_Time	8	Шаг между значениями времени
Data	8·Count_A× ×Count_Time	Значения элементов матрицы $W(a, t)$

В текстовом файле .TXT содержится информация о наименовании исходного сигнала, о наименовании источника, о числе измерений в сутках, о порядковых номерах измерений в дневное и ночное время суток, а также об интервалах отсчетов с указанием номера месяца и номера года. Для этого файла определен формат с указанием последовательности параметров.

Входные данные считаются корректными только при наличии двух файлов — двоичного .WRT и описательного .TXT.

Структура данных в оперативной памяти

В рамках решения поставленной задачи разработана иерархическая структура данных в оперативной памяти, в которую помещается информация после предварительной обработки исходных пар файлов .WRT и .TXT. Данные хранятся блоками (Б) (рис. 4), доступ к которым осуществляется по следующим четырем атрибутам: тип временного ряда (ВР); источник данных (И); год (Г); месяц (М).

Такая структура данных представляет удобную форму для написания программного кода расчета гистограмм на основе выборки. Реализация представленной на рис. 4 структуры данных выполнена путем создания трех объектов (далее по тексту — классов). Первый класс управляет непосредственно блоком данных в рамках одного месяца, второй класс работает с массивом блоков данных в рамках одной пары файлов .WTR и .TXT, третий класс инкапсулирует в себе полное управление структурой данных как единым целым.

Реализация функциональной части приложения

Разработка приложения, решающего поставленную задачу, проводилась в среде визуального программирования Embarcadero RAD Studio. Функциональная часть приложения реализована на основе объектно-ориентированного подхода и состоит из трех классов.

Класс TMonthData определен как потомок TPersistent, что позволяет использовать данный тип в качестве описания составного свойства другого объекта. В задачи данного класса входят: хранение части матрицы вейвлет-преобразования $W(a, t)$ в рамках одного календарного месяца; обеспечение доступа к элементам матрицы; импорт данных из исходной матрицы $W(a, t)$. Основными параметрами создания класса TMonthData являются: номер года, номер месяца, число измерений в сутки, размер матрицы по частоте. Данные параметры определены в конструкторе класса и задаются при создании его экземпляра.

Класс TWTData также определен как потомок TPersistent, поскольку его экземпляр используется в другом классе как составное свойство. Класс реализует задачу чтения данных и их структурирование на основе файлов .WTR и .TXT. В структуру полей данного класса входит массив экземпляров класса TMonthData.

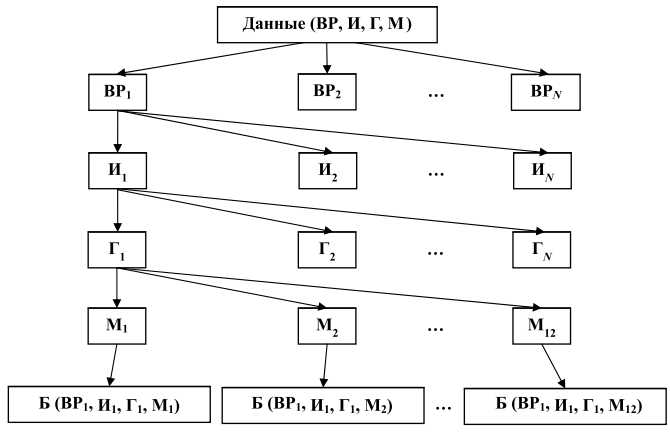

Рис. 4. Иерархическая структура данных в оперативной памяти

Класс TWTDataAnalysis инкапсулирует всю информацию, получаемую из рабочего каталога. В задачи класса входят чтение данных из текущего рабочего каталога и их структурирование в оперативной памяти. Экземпляр этого класса взаимодействует с интерфейсными элементами приложения. Данный класс предоставляет полный перечень свойств и методов, с помощью которых осуществляется доступ к структурированным данным, полученным из текущего рабочего каталога. В классе определены два метода. Метод Init с параметром DirName (имя каталога) запускает на выполнение процесс чтения данных рабочего каталога. Метод SetSignal с параметром Index загружает данные, связанные с типом исходного временного ряда. В единицу времени могут быть загружены данные только одного типа, например, "скорость ветра".

Реализация интерфейсной части приложения

Интерфейсная часть приложения состоит из главного окна (рис. 5) и двух диалоговых окон: окна задания диапазона масштабов и окна визуализации графиков на весь экран. Главное окно разделено на три части:

- задание рабочего каталога с выбором диапазона масштабов;
- панель выборки данных;
- панель визуализации результатов.

На панели выборки данных пользователь выбирает временной ряд, с которым предстоит работа, и компонует выборку для построения гистограммы. Выборка состоит из опций "источники данных", "годы", "месяцы", "время суток".

В начале работы необходимо указать каталог, в котором имеются файлы входных данных, через нажатие на кнопку "Задать рабочий каталог". Текущий каталог запоминается в файле инициализации приложения.

После выбора временного ряда по нажатию опции "Задать диапазон масштабов" в диалоговом окне задается диапазон масштабов (рис. 6).

В части визуализации графиков гистограмм имеются три управляющие кнопки:

- "+" выполняет расчет и добавляет на экран график гистограммы на основе компоновки выборки данных;
- "Очистить" выполняет очистку графиков гистограмм;
- "Показать на весь экран" визуализирует графики гистограмм на весь экран монитора с возможностью сохранения результатов в растровом файле.

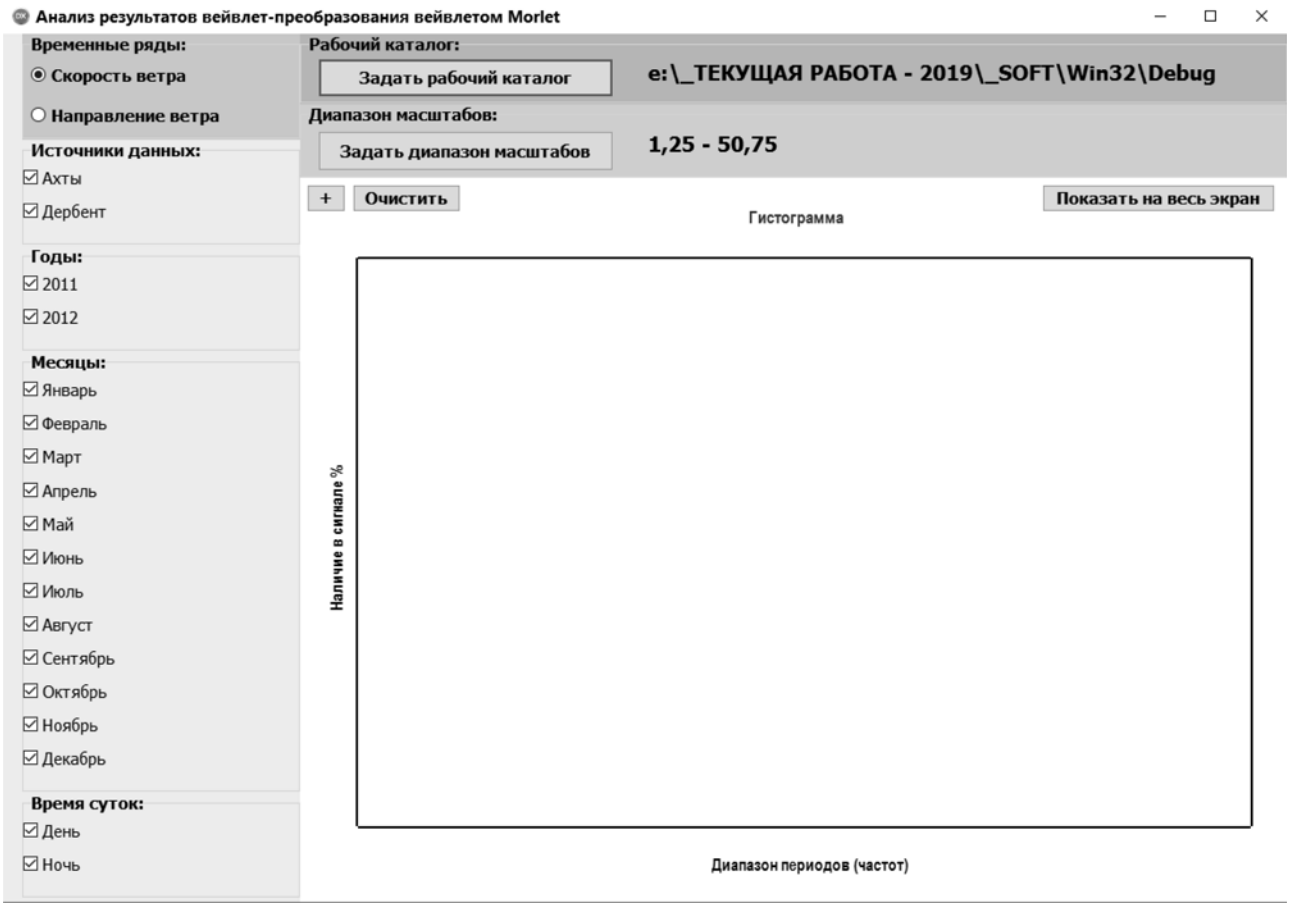

Рис. 5. Главное окно приложения

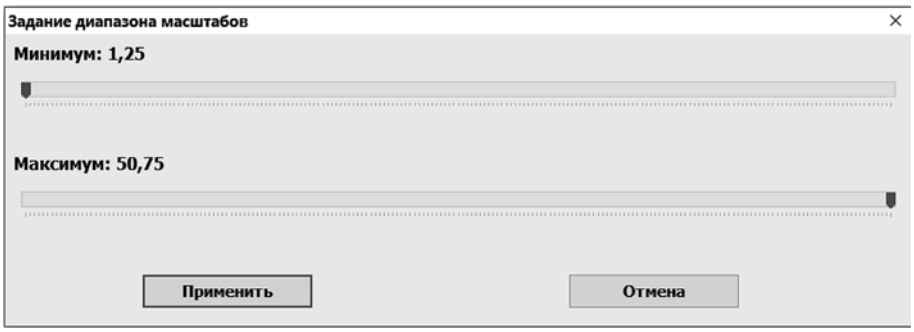

Рис. 6. Диалоговое окно задания диапазона масштабов

Результаты функционирования приложения и практическая значимость

Работа приложения протестирована в рамках текущих научных исследований временных рядов — скоростей ветра на примере данных метеостанций Приморского Дагестана ("Дербент", "Махачкала").

Для всех исследуемых временных рядов за период 2011—2017 гг. по данным метеостанций "Махачкала" и "Дербент" выполнено вейвлет-преобразование функцией morlet в диапазоне периодов от 0,5 до 50,5 суток с интервалом 0,25 суток. Полученные матрицы $W(a, t)$ сохранены в формате .WTR. К каждому файлу .WTR добавлены описательные файлы .TXT таким образом, что двухлетний временной период усекается до одного года по датам 01 июля — 30 июня (например, 01.07.2011—30.06.2012). Общий объем данных составляет около 110 Мбайт.

С помощью разработанного приложения получена информация, которая не была видна в явном виде при визуализации картин матриц $W(a, t)$, как показано на рис. 3. Рассмотрим некоторые примеры полученных результатов.

Рассмотрим гистограмму, в которую включены все имеющиеся данные по двум станциям за все временные периоды (рис. 7). Гистограмма показывает, что основным периодом изменчивости скорости ветра во всех рассматриваемых временных рядах Приморского Дагестана является период в одни сутки. Регулярность проявления периодичностей со значениями 1,5...3 суток в 3—4 раза меньше и уменьшается с ростом их значений.

Рассмотрим гистограммы в диапазоне периодов 0,75...7,25 дней по данным обработки временных рядов 2011—2017 гг. для метеостанций "Дербент" (рис. 8, см. третью сторону обложки) и "Махачкала" (рис. 9, см. третью сторону обложки) по временам года.

На рис. 8—9 наблюдается следующая закономерность относительно периодичности в одни сутки. Максимальный процент наличия периодичности в одни сутки характерен для летнего периода и составляет 80...90 %. Следом по наличию периодичности в одни сутки идет весенний период со значениями 65...75 %. В осенний период наличие в сигнале периодичности в одни сутки составляет 55...65 %. Минимальный процент наличия во временном ряде периодичности в одни сутки наблюдается в зимний период — 35...40 %.

Таким образом, показано, что динамика изменения скорости ветра имеет зависимость от времени года. И, если обратиться к рис. 10, можно говорить о периодической зависимости динамики скорости ветра по календарным месяцам.

На основе гистограмм, полученных по результатам вейвлет-преобразования вейвлетом morlet для временных рядов метеостанций "Дербент" и "Махачкала" за весь временной период исследования (2011—2017 гг.), в редакторе MS Excel построены графики процентного наличия во временном ряде периодичности в одни сутки

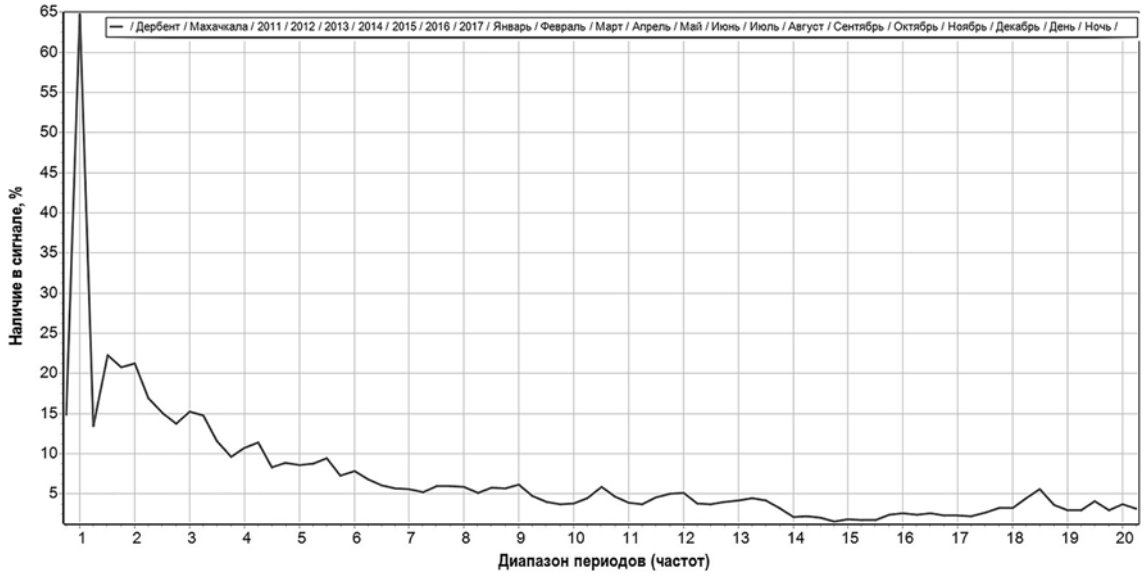

Рис. 7. Гистограмма с полной компоновкой по всему временному периоду данных метеостанций "Махачкала" и "Дербент"

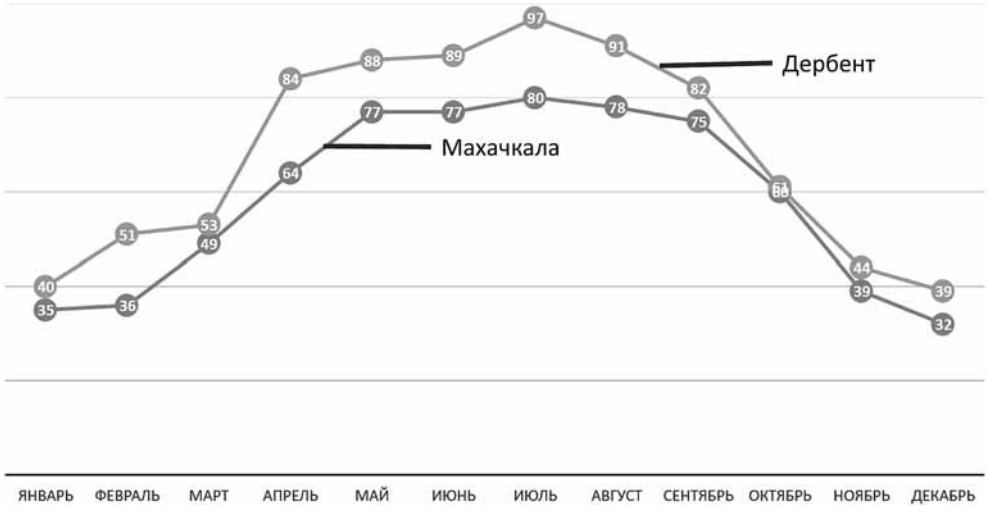

Рис. 10. Наличие периодичности изменения скорости ветра в одни сутки в % во временных рядах метеостанций "Дербент" и "Махачкала" за временной период 2011—2017 гг. по месяцам

в зависимости от календарного месяца (см. рис. 10). Рисунок демонстрирует четкую волну периодичности в рассматриваемом параметре по месяцам с максимальными значениями в июле и минимальными — в декабре. Для данных метеостанции "Дербент" все значения немного выше, чем для данных метеостанции "Махачкала".

Заключение

Разработаны средства автоматизации процесса построения гистограмм частотного распределения по результатам непрерывного вейвлет-преобразования с использованием функции morlet. Они реализованы в виде программного приложения для операционной системы Windows.

Основными характеристиками разработки являются:

- компоновка исходных данных в рамках одного рабочего каталога;
- гибкость в выборе параметров построения гистограмм, таких как тип временного ряда, источник данных, год и месяц, а также дневное и ночное время суток;
- выбор частотного диапазона;
- возможность сохранения результатов в растровое изображение.

Областью применения результатов разработки является исследование частотно-временных характеристик временных рядов, заданных с периодичностью в одно или несколько измерений в сутках, с помощью инструментальных средств непрерывного вейвлет-преобразования вейвлетом morlet.

Разработка позволяет извлекать информацию о частотном распределении, которая не видна в явном виде при визуализации картин матриц $W(a, t)$.

При необходимости программное обеспечение может быть модифицировано под соответствующие научные задачи, например, для другой вейвлет-функции или временных рядов другого масштаба.

Список литературы

1. **Левкович-Маслюк Л.** Дайджест вейвлет-анализа // Компьютерра. 1998. № 8. С. 31—37.

2. **Золотов С. Ю., Ипполитов И. И., Кабанов М. В., Логинов С. В.** Прогнозирование климатических характеристик с помощью метода вейвлет-преобразования // Оптика атмосферы и океана. 2005. Т. 18, № 4. С. 349—351.

3. **Золотов С. Ю., Ипполитов И. И., Логинов С. В.** Прогнозные оценки изменения температуры приземного воздуха с использованием метода вейвлет-преобразования // Оптика атмосферы и океана. 2009. Т. 22, № 5. С. 471—475.

4. **Chellali F., Khellaf A., Belouchrani A.** Wavelet spectral analysis of the temperature and wind speed data at Adrar, Algeria // Renewable Energy. 2010. Vol. 35. P. 1214—1219.

5. **Кобзаренко Д. Н., Камилова А. М., Газанова Н. Ш., Дадашев А. М.** Применение непрерывного вейвлет-преобразования в изучении временных рядов ветромониторинга на примере Дагестана // Информационные технологии. 2018. Т. 24, № 3. С. 201—208.

Automatic Tools for Construction Process of Frequency Distribution Histograms on the Results of Continuous Wavelet Transform by Morlet Function

D. N. Kobzarenko[1, 2], kobzarenko_dm@mail.ru, **A. M. Kamilova**[1], anna702@mail.ru,
B. I. Shikhsaidov[3], sbi707@yandex.ru,
[1] Institute for Geothermal Research of Dagestan Scientific Center of Russian Academy of Sciences, Makhachkala, 367030, Russian Federation,
[2] Dagestan State University of National Economy, Makhachkala, 367008, Russian Federation,
[3] Dagestan State Agrarian University M. M. Dzhambulatova, Makhachkala, 367032, Russian Federation

Corresponding author:

Kobzarenko Dmitry N. Head of Laboratory, Institute for Geothermal Research of Dagestan Scientific Center of Russian Academy of Sciences, Makhachkala, 367030, Russian Federation
E-mail: kobzarenko_dm@mail.ru

Received on April 05, 2019
Accepted on April 12, 2019

Visualization of the continuous wavelet transform matrix in the form of a two-dimensional picture with different color shades gives primary information about the frequency dynamics of the input signal. However, such a representation is not complete, and can be extended with the tool for constructing histograms of the frequency distribution. The process of constructing frequency distribution histograms based on wavelet transform matrices can be automated by creating additional software.

The development purpose is the creation of automation tools for constructing frequency distribution histograms based on the results of the continuous wavelet transform by the Morlet function. This will provide an opportunity to quantify and to compare the results of the continuous wavelet transform from the Morlet wavelet for meteorological time series.

Development tasks include:

— to provide the ability of loading large two-dimensional arrays of the continuous wavelet transform results;

— to provide the ability to set the periodicities interval range;

— to provide the ability to composite data from different sources and different time periods for the chart;

— to provide the ability to display multiple histograms within one chart;

— to provide the ability of saving the results.

The main characteristics of the development are:

— composition of the source data within one working directory;

— flexibility in the choice for histogram constructing parameters such as: type of time series, data source, year and month, day and night time;

— setting a frequency range;

— saving the results in a bitmap image.

The results can be applied in research for time-frequency characteristics of time series given at one or several measurements per day using the tool of continuous wavelet transform. The software allows to extract information about the frequency distribution, which is not visible in an explicit form when wavelet transform matrices are visualized. If necessary, it can be modified to solve some scientific tasks, for example, for another wavelet function or time series with a different scale.

Keywords: *time-frequency analysis, software, histogram, continuous wavelet transform, Morlet wavelet function, meteorological time series*

For citation:

Kobzarenko D. N., Kamilova A. M., Shikhsaidov B. I. Automatic Tools for Construction Process of Frequency Distribution Histograms on the Results of Continuous Wavelet Transform by Morlet Function, *Programmnaya Ingeneria,* 2019, vol. 10, no. 6, pp. 281—288.

DOI: 10.17587/prin.10.281-288

References

1. **Levkovich-Masljuk L.** Dajdzhest vejvlet-analiza (Digest of wavelet analysis), *Komp'juterra,* 1998, no. 8, pp. 31—37 (in Russian).

2. **Zolotov S. Yu., Ippolitov I. I., Kabanov M. V., Loginov S. V.** Prognozirovaniye klimaticheskikh kharakteristik s pomoshch'yu metoda veyvlet-preobrazovaniya (Predicting climatic characteristics using the wavelet transform method), *Optika atmosfery i okeana*, 2005, vol. 18, no. 4, pp. 349—351 (in Russian).

3. **Zolotov S. Ju., Ippolitov I. I., Loginov S. V.** Prognoznye ocenki izmenenija temperatury prizemnogo vozduha s ispol'zovaniem metoda vejvlet-preobrazovanija (Predicted estimates in surface air temperature changes using the wavelet transform method), *Optika atmosfery i okeana*, 2009, vol. 22, no. 5, pp. 471—475 (in Russian).

4. **Chellali F., Khellaf A., Belouchrani A.** Wavelet spectral analysis of the temperature and wind speed data at Adrar, Algeria, *Renewable Energy*, 2010, vol. 35, pp. 1214—1219.

5. **Kobzarenko D. N., Kamilova A. M., Gazanova N. Sh., Dadashev A. M.** Primenenie nepreryvnogo vejvlet-preobrazovanija v izuchenii vremennyh rjadov vetromonitoringa na primere Dagestana (Application of continuous wavelet transform in the study of time series of wind monitoring on the example of Dagestan), *Informacionnye tehnologii*, 2018, vol. 24, no. 3, pp. 201—208 (in Russian).

ООО "Издательство "Новые технологии". 107076, Москва, Стромынский пер., 4
Технический редактор *Е. М. Патрушева*. Корректор *Е. В. Комиссарова*

Сдано в набор 15.04.2019 г. Подписано в печать 21.05.2019 г. Формат 60×88 1/8. Заказ PI619
Цена свободная.

Оригинал-макет ООО "Адвансед солюшнз". Отпечатано в ООО "Адвансед солюшнз".
119071, г. Москва, Ленинский пр-т, д. 19, стр. 1. Сайт: **www.aov.ru**