Математические структуры и моделирование
2015. № 1(33). С. 70–82

УДК 004.93'11

ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ В ОБНАРУЖЕНИИ ЛИЦ НА ИЗОБРАЖЕНИИ

П.В. Прохоров
к.т.н., e-mail: pavelvpster@gmail.com

Омский государственный университет им. Ф.М. Достоевского

Аннотация. В статье рассмотрен алгоритм обнаружения лиц на изображении, задействующий параллельные вычисления на языке Java. С использованием библиотеки Akka реализуется модель MapReduce. В статье приводится описание выбранного алгоритма (AdaBoost), архитектура классической и параллельной реализаций алгоритма, результаты вычислительных экспериментов. Делается вывод о возможности использовании ресурсов многоядерных процессоров.

Ключевые слова: распознавание лиц, параллельная обработка, модель MapReduce.

Введение

Задача обнаружения лица (лиц) на изображении востребована в современной технике. Достаточно упомянуть, что большинство выпускаемых цифровых камер могут наводить резкость по лицу в кадре. Обнаружение лиц используется во многих интерактивных и развлекательных технологиях.

Ранние алгоритмы обнаружения лица на изображении использовали методы статистической обработки точек набора изображений [1–3]. Поясним. Изображение размером 100x100 точек с 1 байтом яркости на точку можно считать точкой в 2560000-мерном пространстве. Все возможные изображения лиц занимают только часть этого пространства. Чтобы ответить на вопрос, содержит ли предъявленное изображение лицо, необходимо оценить, насколько это изображение «близко» к известным нам изображениям лиц. Сравнение будет более информативным, если из исходного пространства выделить (методами статистики) пространство меньшей размерности, в котором изображения лиц и изображения, где лиц нет, отличаются наиболее сильно. В качестве методов статистической обработки используются Метод главных компонент (Principal Component Analysis, PCA) и Линейный дискриминантный анализ (Linear Discriminant Analysis, LDA).

Также существовали алгоритмы, основанные на геометрической модели лица [4], [5]. Используя оптимизацию параметров модели, можно добиться максимального соответствия между моделью и наблюдаемым изображением лица: выполнить адаптацию к позе человека. Однако чаще эти методы применялись

для локализации на изображении заранее известных объектов и, например, выявления и отслеживания позы или мимики.

Наиболее прогрессивный подход был предложен в 2001 году авторами Paul Viola и Michael Jones в своей статье "Rapid Object Detection using a Boosted Cascade of Simple Features" [6]. Алгоритм получил название AdaBoost.

1. Алгоритм обнаружения лиц

Для обнаружения лиц (объектов) они предложили использовать набор двумерных фильтров Хаара, организованных в каскад Хаара. Примеры фильтров Хаара показаны на рисунке 1.

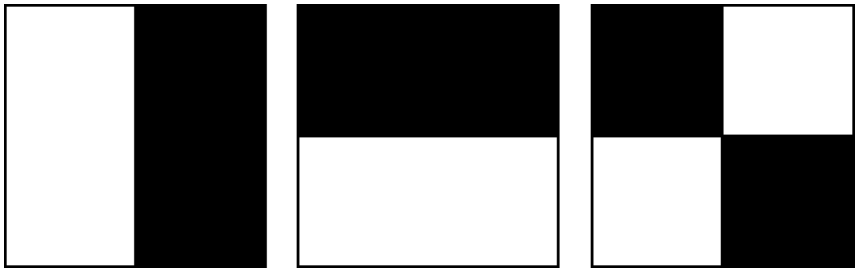

Рис. 1. Примеры фильтров Хаара

При применении к изображению (должно быть «в оттенках серого», grayscale) откликом фильтра считается разность сумм интенсивностей точек, расположенных «под» светлыми и тёмными участками фильтра.

Если применить любой из показанных фильтров Хаара к сплошной поверхности (белой, чёрной или серой), отклик будет нулевым (см. рисунок 2, фильтр 1). Если же изображение содержит характерные границы, соответствующий фильтр даст существенный отклик (см. рисунок 2, фильтр 2).

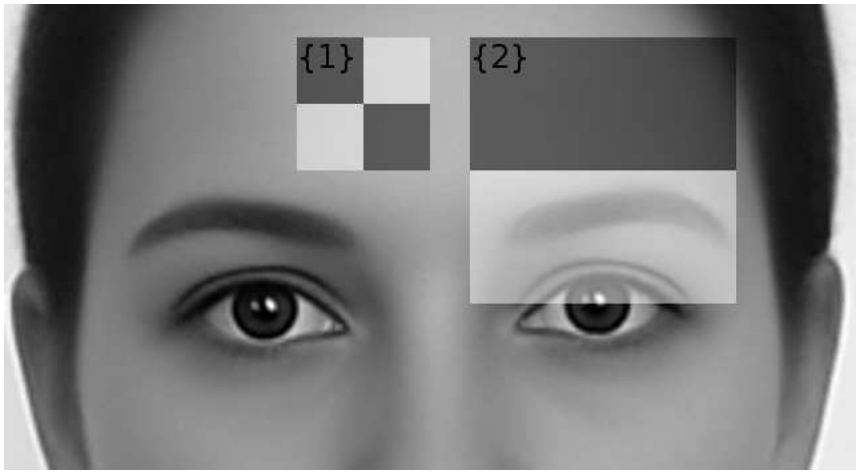

Рис. 2. Применение фильтра Хаара к сплошной поверхности

Конкретный фильтр Хаара, применённый к изображению в определённой точке, является слабым классификатором. Множество слабых классификаторов образуют сильный классификатор (каскад).

Задача классификатора: как можно раньше принять решение, что лица на изображении нет – это экономит вычислительные ресурсы. Поэтому сильный классификатор разбит на этапы. Если отклик этапа (определённого набора слабых классификаторов) превысил порог, обработка продолжается. В противном случае принимается решение, что лица нет. Окончательное решение может приниматься по очень большому (около 200) числу слабых классификаторов.

Использование интегрального изображения, интенсивность каждой точки которого равна сумме интенсивностей всех точек, лежащих выше и слева от неё, позволяет вычислить отклик фильтра Хаара в четыре сложения (вычитания) и одно умножение независимо от площади фильтра.

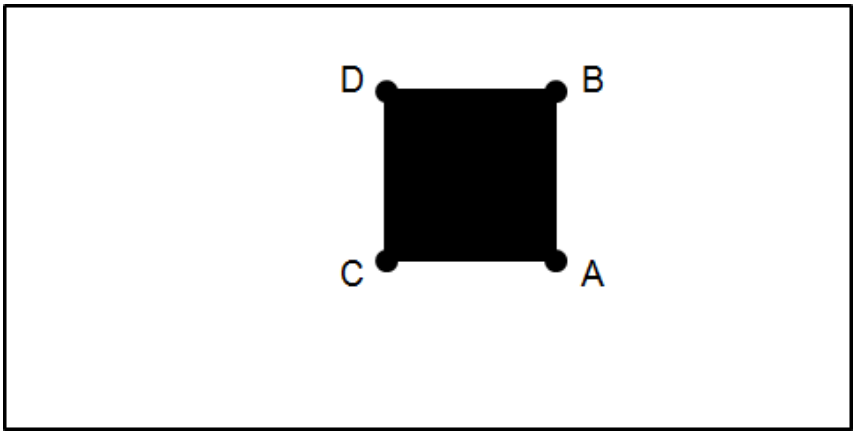

Рис. 3. Прямоугольный фильтр Хаара

Отклик прямоугольника фильтра Хаара, например, показанного на рисунке 3, вычисляется по формуле:

$$r = (A - B - C + D) * W,$$

где A – значение интенсивности в точке A интегрального изображения; B – значение интенсивности в точке B интегрального изображения; C – значение интенсивности в точке C интегрального изображения; D – значение интенсивности в точке D интегрального изображения; W – вес прямоугольника (>0 или <0).

Вес прямоугольников каждого фильтра выбирается так, чтобы вклад в отклик «положительных» и «отрицательных» прямоугольников был одинаковым.

Правильный тип каждого фильтра Хаара, его расположение, весовые коэффициенты и пороги формируются в результате обучения на множестве положительных (изображения, где есть лицо) и отрицательных (изображения, где лиц нет) примеров.

Высокая вычислительная эффективность алгоритма AdaBoost открыла дорогу самой технологии обнаружения лиц.

2. Работа алгоритма

Мы рассмотрели, как формируется отклик классификатора Хаара. Но изображение практически никогда не ограничивается лицом – есть фон. Чтобы найти лицо, необходимо «сканировать» изображение при помощи классификатора в разных масштабах и локализовать участки, где присутствует уверенный отклик.

Алгоритм обнаружения выглядит так:

```
haarDetectObjects(image, cascade, a < 1):
  objects = {}
  scale = min(image.width  / cascade.width,
              image.height / cascade.height)
  do
    point = detect(image, cascade, scale)
    if point != {}
      objects <- new object at point
  while scale > 1
  return objects

detect(image, cascade, scale):
  points = {}
  for y = 0..image.height
    for x = 0..image.width
      r = evaluateHaarCascade
            (image, cascade, x, y, scale)
      if r != 0
        points <- (x, y)
  return k-means-clustering(points)
```

Функция `haarDetectObjects()` перебирает масштабы. Параметрами функции является исходное изображение `image`, каскад Хаара `cascade` и шаг изменения масштаба `a`. Функция `detect()` «сканирует» изображение классификатором в заданном масштабе. Вычисление отклика инкапсулировано в функции `evaluateHaarCascade()`. Если в какой-либо точке получен отклик, координаты точки сохраняются во временном массиве points. Функция `k-means-clustering()` объединяет сохраненные точки в кластеры. Это необходимо, поскольку для одного лица классификатор даёт несколько откликов рядом; лиц на изображении также может быть больше одного. За истинные положения лиц принимаются точки с координатами наиболее близкими к центрам кластеров. Сами центры кластеров нельзя использовать в качестве положений лиц, поскольку это «виртуальные» точки, где классификатор отклика не дал.

3. Реализация каскада Хаара

Классы каскада Хаара показаны на рисунке 4.

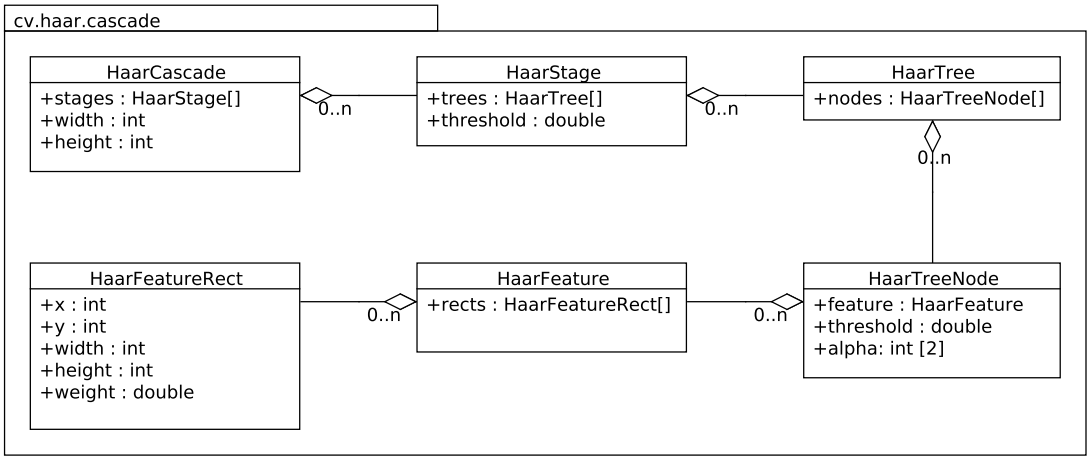

Рис. 4. Классы каскада Хаара

Каскад можно сохранить в файл и считать из файла в формате XML при помощи технологии JAXB. Функции чтения и записи каскада инкапсулированы во вспомогательные классы `HaarCascadeSaver` и `HaarCascadeLoader`.

Алгоритм вычисления отклика каскада Хаара:

```
evaluateHaarCascade(image, cascade, x, y, scale):
  integral = create integral image of image
  s = 0
  foreach stage in cascade.stages
    r = evaluateHaarStage(integral, stage)
    if r = 0 return 0
    s = s + 1
  return s

evaluateHaarStage(integral, stage):
  s = 0
  foreach tree in stage.trees
    s = s + evaluateTree(integral, tree)
  if s < stage.threshold
    s = 0
  return s

evaluateHaarTree(integral, tree):
  s = 0
  index = 0
  do
    node = tree.nodes[index]
    r = evaluateTreeNode(integral, node)
    if r < node.threshold
      index = node.left
```

```
      if index = -1
        r = node.alpha[0]
    else
      index = node.right
      if index = -1
        r = node.alpha[1]
  while index > 0
  return s

evaluateTreeNode(integral, node):
  r = evaluateFeature(integral, node.feature)
  return r

evaluateFeature(integral, feature):
  s = 0
  foreach r in feature.rects
    a = integral(r.x + r.width, r.y + r.height)
    b = integral(r.x + r.width, r.y)
    c = integral(r.x, r.y + r.height)
    d = integral(r.x, r.y)
    s = s + (a - b - c + d) * r.weight
  return s
```

4. Реализация алгоритма обнаружения лиц

Классы показаны на рисунке 5.

Основным является класс `HaarObjectDetector` – он реализует обнаружение объектов. Для обеспечения единообразного доступа клиентских классов к различным обработчикам предназначен интерфейс `ImageProcessor`. Он содержит методы передачи исходного изображения `setSourceImage()`, обработки этого изображения `process()` и получения обработанного изображения `getProcessedImage()`. Большую часть методов интерфейса `HaarObjectDetector`, как и другие обработчики изображений, делегирует абстрактному классу `AbstractImageProcessor`.

Получение отклика каскада Хаара в заданной точке изображения инкапсулировано во вспомогательном (содержит только статические методы) классе `HaarCascadeEvaluator`. Данные инкапсулированы в классе `HaarCascade`.

В результате обработки формируется список объектов класса `HaarObject`. Класс содержит, главным образом, координаты объекта на изображении.

Взаимодействие перечисленных классов показано на рисунке 6.

Перебор масштабов реализован непосредственно в методе `process()`. Метод `detect()` реализует обнаружение объектов в одном масштабе. В этом методе выполняется перебор координат `(x, y)`, получение отклика каскада Хаара, а также кластеризация полученных откликов в объекты.

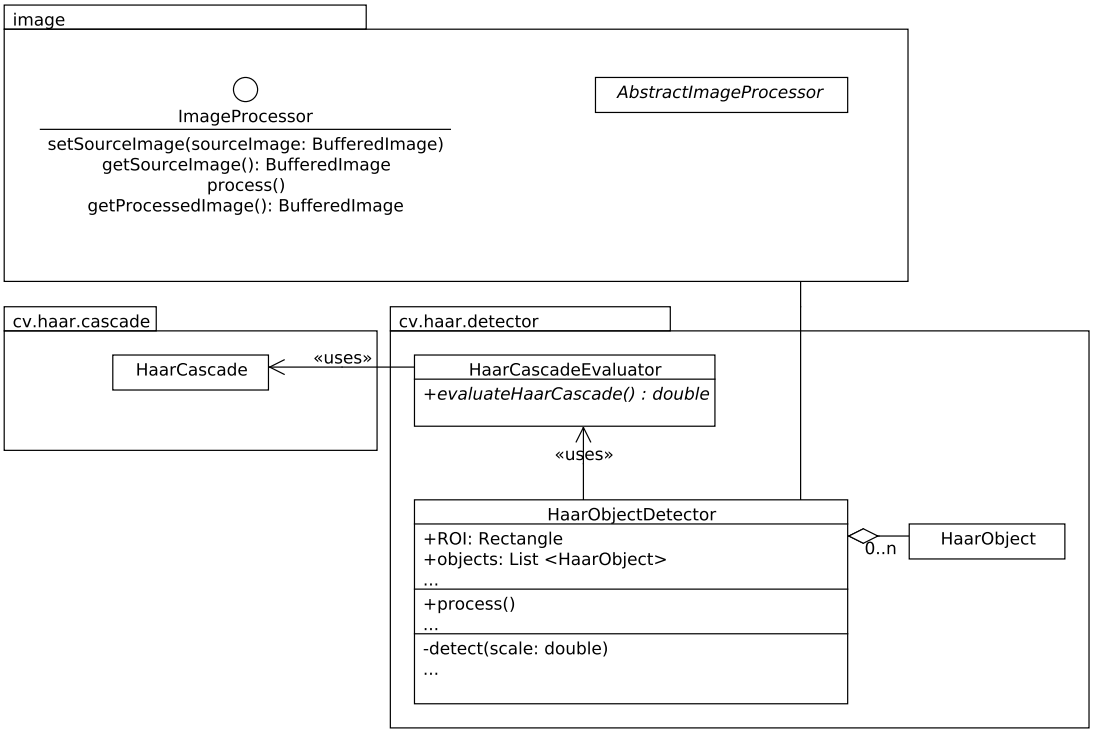

Рис. 5. Диаграмма классов

Классическая реализация не использует возможности многоядерных процессоров, что существенно ограничивает её эффективность. Типичный график загрузки процессора с несколькими (в данном случае двумя) ядрами показан на следующем рисунке. Как видно из рисунка 7, ядра работают по очереди.

5. Параллельная реализация

Применительно к языку программирования Java существует несколько подходов к реализации параллельной обработки:

– использование потоков (thread);

– actor model (часто, вместе с MapReduce);

– реализация специальных шаблонов, например, Disruptor.

Реализация обработки в отдельных потоках – наиболее простой подход. В рассматриваемом алгоритме присутствует перебор масштабов и координат. Можно было бы создать пул потоков; каждому потоку выделить для обработки часть изображения или один масштаб, а затем объединить полученные результаты. Однако это слишком похоже на actor model, чтобы делать все вручную.

Actor model [7] предполагает асинхронное взаимодействие универсальных примитивов распределённых вычислений посредством сообщений. Actor, получив сообщение, может выполнить расчёт, обновить своё состояние, передать сообщения другим actor-ам, создать новые actor-ы. Вместе actor-ы решают поставленную задачу не обязательно связанную с расчётами – это может быть

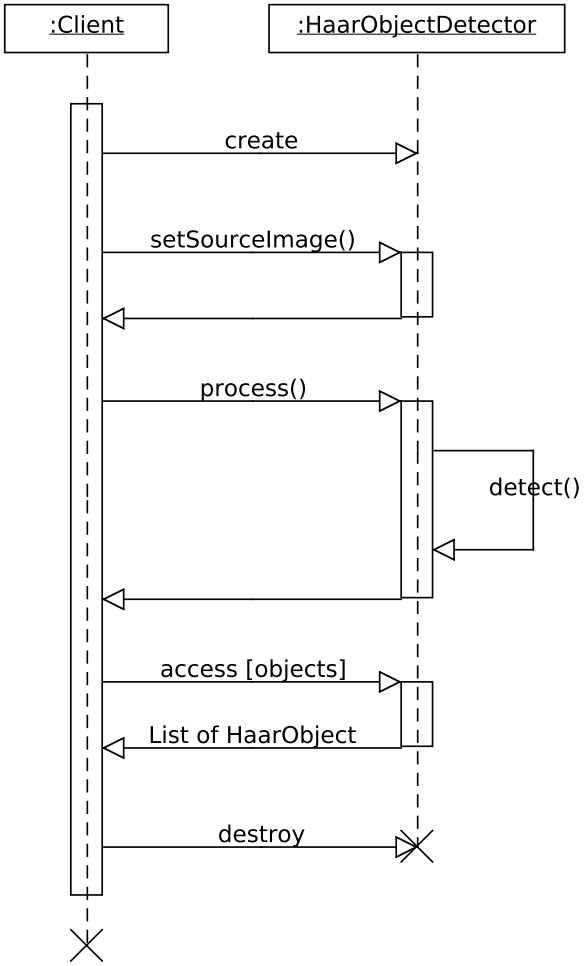

Рис. 6. Взаимодействие классов

Рис. 7. График загрузки процессора с двумя ядрами

преобразование данных, распределённое управление и т.д.

При решении вычислительных задач часто используется программная модель MapReduce [8]. Вычислительный процесс в соответствии с этой моделью разделен на два шага. На шаге map исходная задача делится на более мелкие

задачи и передаётся исполнителям. На шаге reduce полученные от исполнителей промежуточные результаты объединяются в окончательный ответ.

Модель MapReduce легко реализуется совместно с actor model. Исполнители – это actor-ы; исходная задача и промежуточные задачи – сообщения. Для решения задачи обнаружения лиц была выбрана одна из наиболее распространённых реализаций actor model для языков программирования Java и Scala – библиотека Akka [9].

Классы параллельной (concurrent) реализации показаны на рисунке 8:

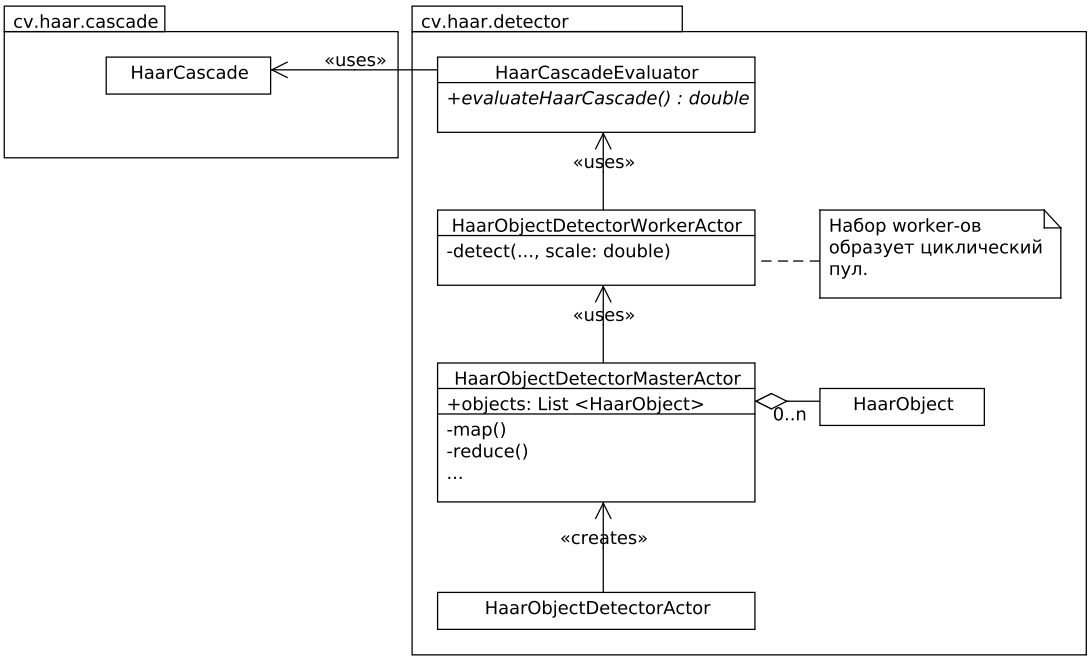

Рис. 8. Классы параллельной реализации

Основной actor – `HaarObjectDetectorActor`. Чтобы выполнить обнаружение лиц на изображении, клиент отправляет путь (или URL) к файлу изображения. Получив это сообщение, основной actor создает `HaarObjectDetectorMasterActor`. Это сделано, чтобы сохранить контекст задачи, т.к. клиентов может быть несколько. На шаге map новый `HaarObjectDetectorMasterActor` перебирает масштабы и формирует задачи исполнителям: выполнить обнаружение в заданном масштабе. Исполнителем является `HaarObjectDetectorWorkerActor`. Для обеспечения эффективности исполнители управляются маршрутизатором с round robin стратегией (свойство библиотеки). Количество исполнителей в эксперименте было равно 16 (задаётся в конфигурационном файле). Исполнитель выполняет перебор координат `(x, y)`, получение отклика каскада Хаара, а также кластеризацию полученных откликов в объекты. Получившиеся списки обнаруженных объектов отправляются обратно. На шаге reduce `HaarObjectDetectorMasterActor` формирует окончательный список обнаруженных объектов и отправляет его клиенту. Сам `HaarObjectDetectorMasterActor` завершает свою работу.

Это взаимодействие проиллюстрировано рисунком 9. При такой реализации ядра задействованы практически полностью (см. рисунок 10).

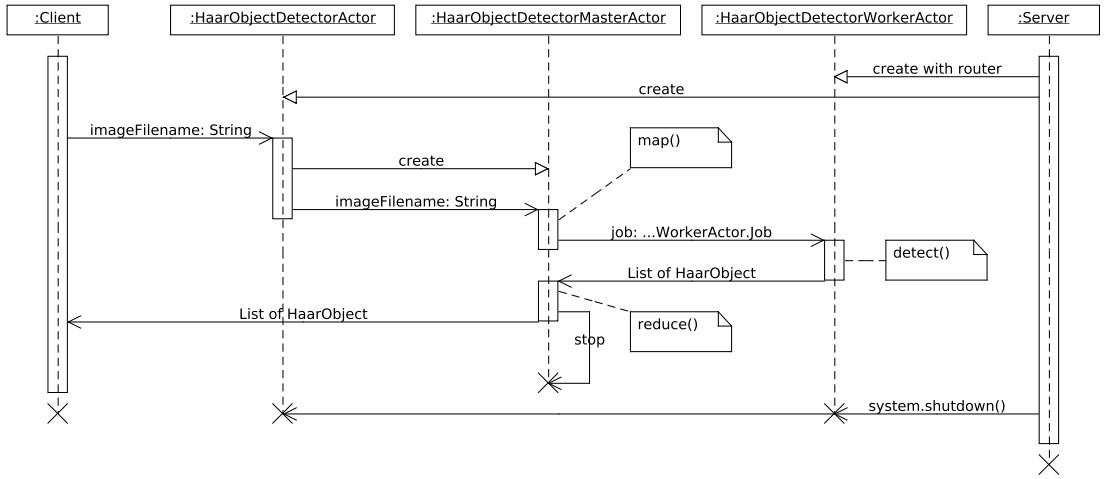

Рис. 9. Взаимодействие классов параллельной реализации

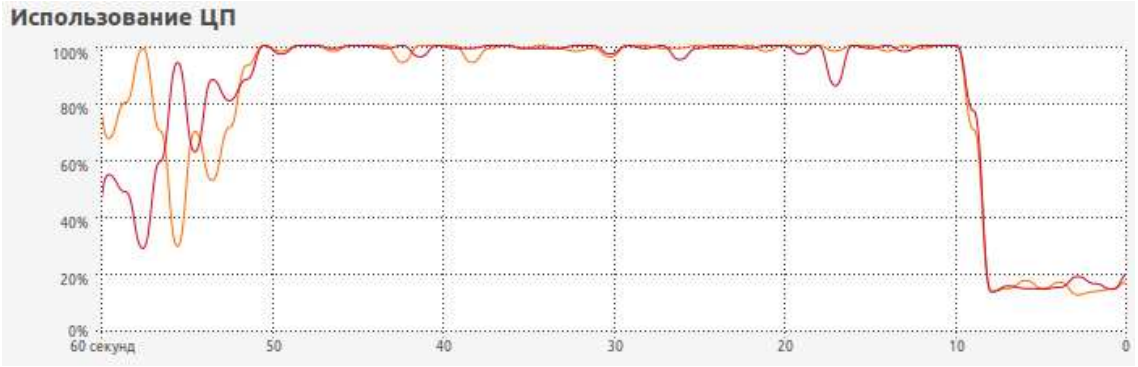

Рис. 10. График загрузки процессора с двумя ядрами при параллельной реализации

6. Результаты вычислительного эксперимента

Вычислительный эксперимент проводился на трёх персональных компьютерах и рабочей станции Dell. Также на каждом из персональных компьютеров запускалась виртуальная машина. Характеристики компьютеров приведены в таблице 1.

Среднее время выполнения (было сделано по 10 повторений) в миллисекундах показано на диаграмме (см. рисунок 11).

Разумеется, параллельная реализация работает быстрее классической. Интересно, что во всех экспериментах классическая реализация работала быстрее на виртуальной машине под управлением операционной системы Ubuntu, чем

Таблица 1. Характеристики компьютеров

Компьютер	Процессор / Количество ядер	Оперативная память	Операционная система
ПЭВМ № 1	Intel Core i7 4770K / 8	8 Гб	Windows 7, 64 бит
VM № 1	- / 2	2 Гб	Ubuntu 12.04, 64 бит
ПЭВМ № 2	Intel Core i5 3450 / 4	8 Гб	Windows 7, 64 бит
VM № 2	- / 2	2 Гб	Ubuntu 12.04, 64 бит
ПЭВМ № 3	Intel Core i5 520M / 4	8 Гб	Windows 7, 64 бит
VM № 3	- / 2	2 Гб	Ubuntu 12.04, 64 бит
ЭВМ Dell	Intel Xeon E5-2687W / 32	32 Гб	Windows 7, 64 бит

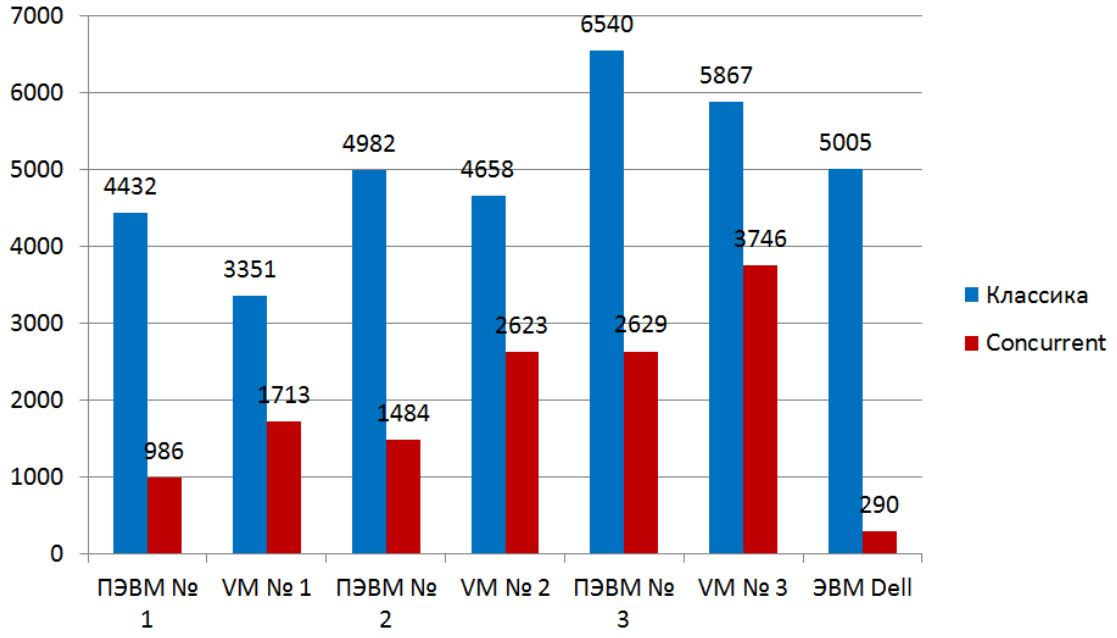

Рис. 11. Среднее время работы алгоритма

в базовой операционной системе Windows 7. Вероятно, это объясняется более эффективной реализацией виртуальной машины Java в Ubuntu.

В таблице 2 показана доля накладных расходов.

Выводы

Результаты вычислительных экспериментов, полученные автором, показывают, что применение шаблона MapReduce позволяет эффективно использовать ресурсы многоядерных процессоров при решении задачи обнаружения лиц на

Таблица 2. Накладные расходы

№ эксперимента	Ожидаемое ускорение	Полученное ускорение	Накладные расходы
1	8	4,5	44%
2	2	1,96	2%
3	4	3,36	16%
4	2	1,76	11%
5	4	2,49	38%
6	2	1,57	22%
7	32	17,25	46%

изображении.

В статье предложена программная архитектура для реализации поставленной задачи с использованием библиотеки Akka. Исходный код проекта доступен в сети Интернет [10].

В дальнейшем предполагается исследовать возможность работы предложенной архитектуры на компьютерном кластере, а также рассмотреть другие библиотеки, например, Apache Hadoop.

Литература

1. Turk M.A, Pentland A.P. Face recognition using eigenfaces // IEEE Conf. on Computer Vision and Pattern Recognition. 1991. P. 586–591.
2. Pentland A.P., Moghaddam B. Starner View-based and modular eigenspaces for face recognition // In Proc. IEEE Conf. Computer Vision and Pattern Recognition. 1994. P. 84–91.
3. Belhumeur P., Hespanha J., Kriegman D. Eigenfaces vs. Fisherfaces: recognition using class specific linear projection // IEEE Transactions on Pattern Analysis and Machine Intellegence. 1997. V. 19. P. 711–720.
4. Cootes T., Edwards G., Taylor C. Active appearance models // IEEE Transactions on Pattern Analysis and Machine Intellegence. 2001. V. 23(6). P. 681–685.
5. Matthews I., Baker S. Active appearance models revised // International Journal of Computer Vision. 2004. V. 60(2). P. 135–164.
6. Viola P., Jones M. Rapid object detection using a boosted cascade of simple features // In Proc. CVPR. 2001.
7. Actor Model URL: `http://en.wikipedia.org/wiki/Actor\_Model` (дата обращения: 01.05.2014).
8. MapReduce URL: `http://en.wikipedia.org/wiki/MapReduce` (дата обращения: 01.05.2014).
9. Akka URL: `http://akka.io/` (дата обращения: 01.05.2014)
10. Проект CV2 URL: `https://github.com/pavelvpster/CV2` (дата обращения: 01.05.2014).

CONCURRENT COMPUTER VISION

P.V. Prohorov
Ph.D. (Eng.), e-mail: pavelvpster@gmail.com

Omsk State University n.a. F.M. Dostoevskiy

Abstract. The article considers a face detection algorithm for still images and its implementation in Java language using parallel computing. MapReduce model is implemented by means of the Akka library. A description of the selected algorithm (AdaBoost), classical architecture and parallel implementations of it as well as the results of computational experiments are provided. Finally, a conclusion on benefits of using multi-core processors for face recognition is made.

Keywords: face recognition, parallel processing, MapReduce model.