
**ПАРАЛЛЕЛЬНОЕ И РАСПРЕДЕЛЕННОЕ
ПРОГРАММИРОВАНИЕ**

УДК 004.021,004.042

БЫСТРЫЙ АЛГОРИТМ ПОИСКА РАСПРЕДЕЛЕННЫХ РАССЕЙВАТЕЛЕЙ SQUEESAR В ЗАДАЧЕ ПОСТРОЕНИЯ СКОРОСТЕЙ СМЕЩЕНИЙ ЗЕМНОЙ ПОВЕРХНОСТИ

© 2021 г. С. Е. Попов^{a,*}, В. П. Потапов^{a,**}

^a *Федеральное государственное бюджетное научное учреждение
“Федеральный исследовательский центр информационных и вычислительных технологий”
630090 Новосибирск, пр. Академика Лаврентьева, 6, Россия*

**E-mail: popov@ict.sbras.ru,*

***E-mail: vadimptpv@gmail.com*

Поступила в редакцию 23.12.2020 г.

После доработки 27.04.2021 г.

Принята к публикации 15.06.2021 г.

В статье описывается программная реализация быстрого алгоритма поиска распределенных рассеивателей для задачи построения скоростей смещений земной поверхности на базе платформы Apache Spark. Рассматривается полная схема расчета скоростей смещений методом постоянных рассеивателей (PS). Предложенный алгоритм интегрируется в схему после этапа совмещения с субпиксельной точностью стека изображений временной серии радарных снимков космического аппарата Sentinel-1. Поиск распределенных рассеивателей происходит независимо в окнах сдвига по всей площади снимка. Наличие последних определяется путем предположения о гомогенности пар выборок в окне, составленных из векторов комплексных значений пикселей в каждом из N изображений. Данное предположение вытекает из выполнимости критерия Колмогорова–Смирнова для каждой из пар. Для оценки значений фаз гомогенных пикселей решается задача максимизации. Показано, что предложенный алгоритм не является итерационным и может быть реализован в парадигме параллельных вычислений. Применяемая платформа Apache Spark позволила распределенно обрабатывать массивы стека радарных данных (от 60 изображений) в памяти на большом количестве физических узлов в сетевой среде. При этом, время поиска распределенных рассеивателей удалось снизить в среднем в 10 раз по сравнению с однопроцессорной реализацией алгоритма. Приведены сравнительные результаты тестирования вычислительной системы на демонстрационном кластере. Алгоритм реализован на языке программирования Python с подробным описанием объектов и методов алгоритма.

DOI: 10.31857/S0132347421060066

1. ВВЕДЕНИЕ

Основная ценность космической информации, поступающей при мониторинге земной поверхности, зачастую, заключается в возможности ее оперативного анализа. Поэтому, активное развитие методов дифференциальной интерферометрии и средств дистанционного зондирования кроме совершенствования аппаратной части спутников, также требует создания проблемно-ориентированных алгоритмов для автоматизированной и оперативной обработки большого объема радарных данных. Для стандартного в радарной интерферометрии аналитического аппарата DInSAR [1] для оценки скоростей смещений земной поверхности широкое распространение получил метод постоянных отражателей (Persistent Scatterer, PS) [2]. PS направлен на идентификацию когерентных радиолокационных целей, де-

монстрирующих высокую фазовую стабильность в течение всего периода наблюдения. Эти цели (пиксели изображения), на которые лишь незначительно влияет временная и геометрическая декорреляция, часто соответствуют точечным рассеивателям и обычно характеризуются высокими значениями отражательной способности [3]. В работах [4, 5 и др.] показано, что эти пиксели также соответствуют пикселям изображения, принадлежащим областям умеренной когерентности в некоторых интерферометрических парах доступного набора данных. Здесь многие соседние пиксели имеют одинаковые значения отражательной способности, поскольку они принадлежат к одному объекту. Эти цели, называемые распределенными рассеивателями (Distributed Scatterers (DS)), обычно соответствуют участкам небольших строений, необрабатываемых земель с короткой растительно-

стью, промышленному мусору, металлическим завалам или пустынным территориям. Хотя средняя временная когерентность этих естественных радиолокационных целей обычно низкая из-за явлений как временной, так и геометрической декорреляции, количество пикселей с одинаковым статистическим поведением может быть достаточно большим, чтобы некоторые из них могли превысить порог когерентности и стать постоянными рассеивателями.

В работах [4–6] описывается решение актуальной задачи слияния данных для правильного объединения рассеивателей PS и DS для увеличения плотности точек измерения. Это позволяет увеличить пространственную плотность точек областей, характеризующимися DS при сохранении высококачественной информации, полученной с помощью метода PS по детерминированным целям. Данные пространственно усредняются по статистически однородным областям, увеличивая отношение сигнал/шум (SNR), без ущерба для идентификации когерентных точечных рассеивателей. Данный алгоритм назван SqueeSAR [7, 8]. Он позволяет проводить поиск и обработку точек распределённых рассеивателей, интегрируя их в общую схему расчета скоростей смещений методом PS.

Тщательный анализ алгоритма SqueeSAR выявил места, критически влияющие на его производительность. Весь алгоритм строится на переборе начальных данных. На каждом шаге выполняются нетривиальные преобразования данных. Так, например, ощутимо сложными в плане вычислительных затрат оказались этапы поиска смежных точек в окне сдвига и решение задачи максимизации при оценке реальных значений интерферометрических фаз [9]. Кроме того, при уменьшении размеров окна, увеличивалось общее количество шагов проходки по всей площади снимка.

Для устранения выявленных проблем вычислительного характера было предложено использование платформы массово-параллельных вычислений Apache Spark [10]. Apache Spark – это фреймворк с открытым исходным кодом для распределённой пакетной и потоковой обработки неструктурированных и слабоструктурированных данных, входящий в экосистему проектов Hadoop. В отличие от классического обработчика ядра Apache Hadoop с двухуровневой концепцией MapReduce на базе дискового хранилища, Spark использует специализированные примитивы (Resilient Distributed Data) для рекуррентной обработки в оперативной памяти. Благодаря этому появляется возможность многократного доступа к загруженным в память радарным данным с каждого узла кластера. Это позволяет логически раз-

делять стек снимков на подобласти и проводить вычисления независимо.

Целью данной работы является разработка высокопроизводительного алгоритма поиска распределённых рассеивателей для математической модели SqueeSAR.

Для выполнения поставленной цели предлагается решение следующих задач:

1. Разработка программного алгоритма на основе математической модели SqueeSAR на базе открытых библиотек с возможностью его интеграции в схему расчета скоростей смещений методом постоянных отражателей.

2. Адаптация представленного алгоритма для запуска его в среде массово-параллельного исполнения заданий.

2. МАТЕМАТИЧЕСКАЯ МОДЕЛЬ АЛГОРИТМА SqueeSAR

Алгоритм SqueeSAR базируется на идее поиска статистически однородных пикселей, в результате чего выявляются подмножества точечных, распределённых рассеивателей, и выполняется пространственно-адаптивная фильтрация последних. Вследствие чего резко возрастает суммарный набор постоянных отражателей, что позволяет повысить эффективность решения задач дифференциальной радарной интерферометрии [9]. Алгоритм состоит из следующих этапов [11]:

1. Определение статистически однородных точек (Statistically Homogeneous Pixels – SHP) по всей площади стека интерферометрических изображений.

2. Формирование наборов распределённых рассеивателей путем фильтрации по количественному порогу SHP-пикселей.

3. Поиск для каждого DS-набора уточненных (оценка) значений интерферометрической фазы SHP-пикселей, используя специальную матрицу когерентности, и решение на ее основе задачи максимизации.

4. Фильтрация полученных уточненных интерферометрических фаз пикселей DS-набора на основе оценки параметра γ_{PTA} .

5. Замена значений первоначальных фаз в интерферометрическом стеке их уточненными значениями. Модифицированные изображения далее участвуют в процессе обработки методом постоянных рассеивателей [2].

Будем рассматривать стек из N совмещенных с субпиксельной точностью радарных изображений длиной временной серии (от 60 снимков). Здесь субпиксельная точность означает, что любая произвольно взятая точка на мастер-изображении будет иметь абсолютно одинаковые географические координаты и на $N-1$ подчиненных

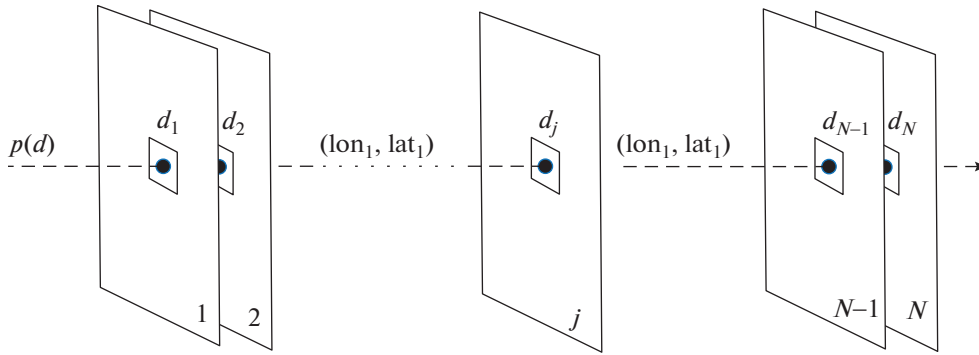


Рис. 1. Стек из N совмещенных с субпиксельной точностью изображений.

изображениях (рис. 1). Обозначим W и H ширину и высоту изображений в пикселях, соответственно.

Делается предположение, что данные радарных изображений и рассчитываемые на их основе геофизические параметры являются общими для каждой статистически однородной выборки близлежащих пикселей. В соответствии с этим предположением однородность (гомогенность) пикселей оценивается в пределах заданного окна размером $m \times n$ (обычно 21×15 пикселей). Сдвиг окна происходит по ширине и высоте изображения с шагом равным ширине и высоте окна.

На каждом шаге выбирается центральный пиксель p_0 в окне сдвига. Так как на изображениях каждый пиксель характеризуется комплексным представлением (каналами I и Q , для данных Sentinel-1), то можно сформировать вектор комплексных чисел d в стеке из N изображений (1)

$$d_k = [d_{k,1}, d_{k,2} \dots d_{k,N}]^T, \quad (1)$$

где $d_{k,j} = a + ib$, $k = 0 \dots m \times n$, $j = 1 \dots N$, $a \in I$, $b \in Q$ – комплексное представление значения пикселя в соответствующем j -м изображении в стеке для k -го пикселя, T – оператор транспонирования.

Два пикселя p_0 и p_k в окне будем считать однородными если для них выполняется тест Колмогорова–Смирнова. Каждая пара пикселей (p_0, p_k) рассматривается как две независимые выборки $|p_0(d_j)|$ и $|p_k(d_j)|$, где $k = 1 \dots m \times n$, $|p_k(d_j)| = [|d_{k,1}|, |d_{k,2}|, \dots |d_{k,N}|]^T$, $|\cdot|$ – модуль комплексного числа. Здесь тест Колмогорова–Смирнова выполняется следующим образом:

1. Выдвигаются две гипотезы: H_0 – различия между двумя выборками недостоверны, H_1 – противоположная (различия между выборками достоверны), соответственно для заданного уровня значимости α .

2. Строится частотное распределение каждой выборки.

3. Вычисляются относительные частоты, равные частному от деления частот на объем выборки, для каждой из имеющихся выборок.

4. Определяется модуль разности соответствующих относительных частот.

5. Определяется наибольший модуль D_{max} .

6. Вычисляется эмпирическое значение критерия λ_{emp} .

$$\lambda_{emp} = D_{max} \sqrt{\frac{N}{2}}.$$

7. Определяется критическое значение критерия для выбранного уровня значимости $\alpha = 0.05$.

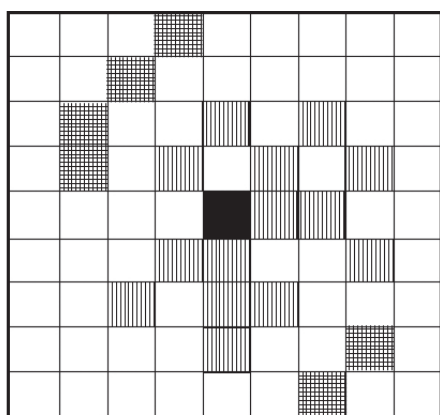
8. Если эмпирическое значение критерия меньше критического, то нулевая гипотеза принимается, и выборки по рассмотренному признаку не отличаются существенно, т.е. пара пикселей считается однородной.

В получившемся наборе пикселей, прошедших тест, отбрасываются те пиксели, которые не являются смежными с центральным (p_0) либо напрямую, либо через соседние (рис. 2).

Оставшиеся пиксели (обозначим их количество параметром N_{ds}) образуют DS-набор. Если параметр N_{ds} меньше порогового значения ($N_p = 20$), то такой набор не принимается. Таким образом, при проходе всего снимка образуются наборы распределенных рассеивателей. В каждом корректном DS-наборе (2) для центрального пикселя p_0 выполняется процедура уточнения значения его “свернутой” интерферометрической фазы в каждом из N изображений стека.

$$DS = \{p_i(d)\}, \quad i = 1 \dots N_{ds}, \quad N_{ds} > N_p \quad (2)$$

Значения интерферометрических фаз пикселей в DS-наборе могут быть статистически охарактеризованы с использованием матрицы когерентности, которая определяется следующим образом (3):






 Пиксели не смежные с центральным
 Пиксели смежные с центральным
 Центральный пиксель

Рис. 2. Вариант расположения пикселей в наборе DS. Белые пиксели не являются гомогенными с центральным, заштрихованные пиксели прошли тест Колмогорова–Смирного и являются гомогенными с центральным.

$$T = \frac{1}{N_{ds}} \sum_{i=1}^{N_{ds}} \hat{p}_i \hat{p}_i^+, \quad (3)$$

где

+ – операция эрмитового сопряжения вектора \hat{p}_i ,

$$\hat{p}_i = p_i(d) / \sqrt{E[|p_i(d)|^2]},$$

$$E[|p_i(d)|^2] = \frac{1}{N} \sum_{j=1}^N |d_{ij}|^2,$$

$|d_{ij}|$ – модуль комплексного числа, T – матрица комплексных чисел размерностью $N \times N$.

Ключевой проблемой является нахождение вектора $\theta = [\theta_1, \theta_2 \dots \theta_N]^T$, который бы служил оценкой полученных значений фаз φ для недиагональных элементов в матрице T . Для этого представим матрицу T в параметрическом виде, заменив диагональные элементы на 1 (4):

$$T = \begin{bmatrix} 1 & \gamma_{1,2} e^{j\varphi_{1,2}} & \dots & \gamma_{1,N} e^{j\varphi_{1,N}} \\ \gamma_{2,1} e^{j\varphi_{2,1}} & 1 & \dots & \gamma_{2,N} e^{j\varphi_{2,N}} \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_{N,1} e^{j\varphi_{N,1}} & \gamma_{N,2} e^{j\varphi_{N,2}} & \dots & 1 \end{bmatrix} \quad (4)$$

При этом значения фаз $\varphi_{n,m} = -\varphi_{m,n}$, $n,m = 1 \dots N$, а матрица $|T|$ будет представлена элементами $\gamma_{m,n}$.

Рассматривается метод максимального правдоподобия (ML estimator (MLE)). Построение корректной формы MLE проводится путем анализа статистических свойств матрицы когерентности (4), используя комплексное распределение Уишарта. На основе центральной предельной теоремы, делается предположение, что нормализованный вектор радарных данных \hat{p}_i соответствует комплексному многомерному нормальному распределению с нулевым средним и дисперсионной матрицей Σ [12–14]. Предполагается, что матрица T логически может быть представлена комплексным распределением Уишарта с N_{ds} – степенями свободы в виде $T \sim W_c(N, N_{ds}, \Sigma)$, где Σ для пикселя $p_0(d)$ может быть определена с использованием уточненных значений когерентности и фазы как (5)

$$\Sigma = \Theta Y \Theta^H, \quad (5)$$

$$\text{где } \Theta = \begin{bmatrix} e^{j\theta_1} & 0 & \dots & 0 \\ 0 & e^{j\theta_2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & e^{j\theta_N} \end{bmatrix}, Y = \begin{bmatrix} 1 & \gamma_{1,2} & \dots & \gamma_{1,N} \\ \gamma_{2,1} & 1 & \dots & \gamma_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_{N,1} & \gamma_{N,2} & \dots & 1 \end{bmatrix},$$

$\theta = [\theta_1, \theta_2 \dots \theta_N]^T$ – вектор оптимальных (уточненных) фаз для вектора $\varphi = [\varphi_1, \varphi_2 \dots \varphi_N]$. При вычисленной матрице T корректная форма MLE для Σ может быть получена путем максимизации абсолютного значения логарифма следующей функции плотности вероятности [12]:

$$\begin{aligned} \hat{\Sigma} &= \arg \max_{\Sigma} \{\ln[p(T | \Sigma)]\} = \\ &= \arg \max_{\Sigma} \{-\text{trace}(N_{ds} \Sigma^{-1} T) - N_{ds} \ln(\text{Det}(\Sigma))\} = \quad (6) \\ &= \arg \max_{Y, \Theta} \{-\text{trace}(\Theta Y^{-1} \Theta^H T) - \ln(\text{Det}(\Sigma))\} \end{aligned}$$

Для оценки Θ в (6) требуется значения γ матрицы Y . Не теряя общности, их можно заменить значениями γ из $|T|$ [12]. Следовательно, (6) примет следующий вид:

$$\begin{aligned} \Theta_{ML} &= \arg \max_{\Theta} \{-\text{trace}(\Theta |T|^{-1} \Theta^H T) - \ln(\text{Det}(|T|))\} \\ &= \arg \max_{\Theta} \{-\text{trace}(\Theta |T|^{-1} \Theta^H T)\} = \quad (7) \\ &= \arg \max_{\Theta} \{\Lambda^H (-|T|^{-1} \circ T) \Lambda\}, \end{aligned}$$

где $\Lambda = [e^{j\theta_1}, e^{j\theta_2} \dots e^{j\theta_N}]^T$, знак \circ – произведение Адамара.

Для удобства программной реализации преобразуем (7) в следующий вид, используя тригонометрическую форму комплексного числа:

Таблица 1. Начальные переменные

Переменная	Тип	Размерность	Описание
img_width, img_height, N	int	1	Ширина, высота изображений, их количество в серии
i, q	Array, float32	width*height, N	Массивы значений пикселей для полос Q и I для каждого изображения в серии, соответственно
Np	int		Пороговое значение для количества пикселей в DS-наборе
shift_win_widht, shift_win_height	int	1	Ширина и высота окна сдвига
num_slices	int	1	Количество разделов (partitions) во вновь создаваемом RDD [10]

$$\Theta_{ML} = \arg \max_{\Theta} \{ \text{sum}((-|T|^{-1} \circ |T|) \circ \Phi \circ (\Lambda \Lambda^H)^T) \} = \arg \max_{\Theta} \{ F_{\max} \}, \quad (8)$$

$$F_{\max} = \sum_{m=1}^N \sum_{n>m}^N \hat{\gamma}_{m,n} \cos(\varphi_{m,n} - \theta_m - \theta_n),$$

где $\hat{\gamma}_{m,n}$ – элемент матрицы $-|T|^{-1} \circ |T|$ в строке m и столбце n, соответственно. sum^* – оператор суммирования.

Решение (8) можно проводить различными методами. В данной работе будем использовать итерационный метод численной оптимизации BFGS (Broyden, Fletcher, Goldfarb, Shanno). После того, как будет найден вектор оптимальных решений $\theta = [\theta_1, \theta_2 \dots \theta_N]^T$, необходимо оценить качество полученной оптимизации. Для этого используется следующий фильтр:

$$\gamma_{PTA} = \frac{2}{N^2 - N} \text{Re} \sum_{n=1}^N \sum_{k=n+1}^N e^{i\varphi_{n,k}} e^{-i(\theta_n - \theta_k)}, \quad (9)$$

где оператор Re^* – взятие вещественной части комплексного числа, $\varphi_{n,k}$ – значения фаз комплексных чисел матрицы T . Вектор оптимальных решений θ принимается если значение $\gamma_{PTA} \geq 0.5$ [15]. Значения начальных фаз пикселей каждого из изображений в стеке заменяются соответствующими значениями $\theta = [\theta_1, \theta_2 \dots \theta_N]^T$ пикселей из DS-наборов. Далее, продолжается стандартная пре- и постобработка для метода постоянных отражателей [2] с измененными интерферометрическими изображениями.

3. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ

Программная реализация алгоритма построена на базе языка Python, параллельные вычисления – на базе Apache Spark API For Python [10]. Параллелизация строится на том факте, что каждое окно является независимой сущностью. Вычисления над входными данными не используют результаты аналогичных вычислений. Значения

переменных и объектов внутри окна не влияют на выполнение вычислений в других окнах сдвига. Количество окон определяется только размером исходных изображений в стеке. Совмещение всех изображений в стеке с субпиксельной точностью гарантирует равный размер и количество окон. Таким образом, можно утверждать, что представленный алгоритм не является итерационным, и может быть реализован в парадигме параллельных вычислений.

Исходный программный код находится в открытом доступе по адресу <https://bitbucket.org/ogidog/pysqueesar/src/master/>.

Для удобства работы координаты пикселей изображений будем представлять в одномерном виде, т.е. пиксель с координатами $(x, y) \leftrightarrow xy = yW + x$. $x \in [0, W-1]$, $y \in [0, H-1]$, W и H – ширина и высота изображений в пикселях, соответственно (переменные: width и height).

Создаются следующие программные переменные (табл. 1):

Формируется массив, содержащий координаты центральных пикселей `central_pixels`, тип `int`, заполняем массивы `i, q`. Данная процедура называется `get_input_data()` (см. исходный код). На основе данного массива происходит полный расчет распределенных отражателей и их уточненных значений фаз согласно алгоритму, представленному в разделе “Математическая модель алгоритма SqueeSAR”.

Для подключения, управления и инициализации вычислений на кластере применяется технология контекстного запуска независимых разделяемых исполнителей (Executors) Spark Context (объект `SparkContext` из библиотеки `pyspark`). `SparkContext` предоставляет методы соединения с кластером Spark и может использоваться для создания RDD и широковещательных переменных в кластере (рис. 3). Настройка контекста происходит непосредственно в коде и в командной строке запуска программы-драйвера (см. исходный код, файл `main_parallel.py`)

```

1  #####
2  # Spark config and init
3  #####
4
5  conf = SparkConf()
6  conf.setMaster("yarn-client")
7  conf.setAppName("PySqueeSar")
8  sc = SparkContext(conf=conf)
9  num_slices = 30
10
11  i_broadcast = sc.broadcast(i)
12  q_broadcast = sc.broadcast(q)
13  img_height_broadcast = sc.broadcast(lines)
14  img_width_broadcast = sc.broadcast(samples)
15  N_broadcast = sc.broadcast(N)
16  Np_broadcast = sc.broadcast(Np)
17
18  #####
19  # Partitioning central_pixels
20  #####
21
22  central_pixels_slices = np.array_split(central_pixels, num_slices)
23  central_pixels_slices_broadcast = sc.broadcast(central_pixels_slices)
24
25  #####
26  # Start calc proc
27  #####
28
29  theta_entire = sc.parallelize(['np.arange(num_slices)'], numSlices=num_slices)\
30  .map(processing_ds).reduce(lambda x, y: x + y)

```

Рис. 3. Фрагмент кода инициализации и запуска расчетного задания на кластере Apache Spark/Yarn.

Для эффективного предоставления каждому узлу копии большого входного набора данных применяются широковещательные переменные (broadcast). Они позволяют хранить данные только для чтения в кэше на каждой машине, а не отправлять их копию вместе с задачами. Spark автоматически передает данные, необходимые для задач на каждом этапе, если эти данные являются глобальными для функции, которая выполняется на map-стадии (рис. 3). Общие данные кэшируются в сериализованной форме и десериализуются перед запуском каждого расчетного задания.

Для создания broadcast-переменных используется метод объекта SparkContext.broadcast() (рис. 3), переменные i_broadcast, q_broadcast, Np_broadcast, N_broadcast, img_width_broadcast, img_height_broadcast для соответствующих переменных из табл. 1). Данные, хранящиеся в перечисленных переменных, являются общими и неизменяемыми на протяжении всех преобразований в алгоритме.

Массив координат центральных пикселей central_pixels разбивается на разделы (slices). Для этого используется функция numpy.array_split() (рис. 3), и создается broadcast-переменная central_pixels_slices_broadcast размерностью (num_slices, len(central_pixels)// num_slices).

Для работы с данными в среде Apache Spark в параллельном режиме создаются специальные объекты RDD (Resilient Distributed Dataset). RDD – это отказоустойчивый набор элементов, с которыми можно работать параллельно [10]. Формирование RDD происходит путем логического распараллеливания входной коллекции/массива данных в управляющей программе-драйвере при помощи метода SparkContext.parallelize() (рис. 3). При этом на получившихся RDD-наборах возможно использование классических функций map-трансформаций, которые имплементируют алгоритм построения DS-наборов и поиска оптимальных решений (метод processing_ds(), рис. 3,4).

В качестве входного параметра в метод parallelize() передается массив, содержащий индексы разделов (slices) переменной central_pixels_slices_broadcast. Все разделы обрабатываются расчетными заданиями на map-стадии параллельно. В то время как отдельно взятое расчетное задание обрабатывает свой конкретный набор центральных пикселей в разделе последовательно на выделенном заданию ядре, используя метод processing_ds().

Ниже представлена последовательность действий и их описание для процедуры processing_ds.

```

def processing_ds(slice_number):
1   central_pixels_slice = central_pixels_slices_broadcast
2   .value[slice_number]
3   theta_slice = []
4   for central_pixel in central_pixels_slice:
5       ds_pixels = np.array(get_ds_within_window(central_pixel))
6       if len(ds_pixels) >= Np_broadcast.value:
7           T = np.zeros((N_broadcast.value, N_broadcast.value))
8           for ds_pixel in ds_pixels:
9               norm_complex_ds_pixel, norm_hermitian_complex_ds_pixel \
10                  = get_norm_complex_sn(ds_pixel)
11              T = T + norm_complex_ds_pixel * norm_hermitian_complex_ds_pixel
12              T = T / len(ds_pixels)
13              theta_initial = np.arctan2(q[central_pixel],
14                                         i[central_pixel]).astype("float32")
15              gamma = -linalg.pinv(np.abs(T)) * np.abs(T)
16              sol2 = optimize.minimize(fun=objective, x0=theta_initial,
17                                     args=(np.angle(T), gamma, N),
18                                     method='L-BFGS-B',
19                                     jac=jacobian,
20                                     bounds=optimize.Bounds(-np.pi, np.pi),
21                                     options={'ftol': 1e-3})
22
23              theta = sol2.x
24              test_result, test_value = pta_test(theta, np.angle(T), N)
25              if test_result:
26                  theta_slice.append([theta, ds_pixels])
27              else:
28                  theta_slice.append([])
29          else:
30              theta_slice.append([])
return (theta_slice)

```

Рис. 4. Фрагмент кода метода processing_ds.

1. По индексу раздела из broadcast-переменной central_pixels_slices_broadcast получаем список central_pixels_slice (тип List), содержащий координаты центральных пикселей окон сдвига (mxn) для данного раздела (рис. 4, строка 1). Координаты записаны в формате $xy = yW + x$, как упоминалось выше.

2. Для каждого пикселя в текущем разделе получаем массив (DS-набор, тип List, рис. 4, строка 5), в котором хранятся индексы тех точек серии изображений, которые могут считаться кандидатами в распределенные рассеиватели (2), метод get_ds_within_window() (см. исходный код, файл main_parallel.py). В данном методе для каждого пикселя-кандидата выполняется тест Колмогорова–Смирнова, описанный в разделе “Математическая модель алгоритма SqueeSAR”.

Используется вспомогательный метод get_sn_vec() (рис. 5) для получения относительных частот по каждому пикселю в окне. В данном методе происходит обращение к broadcast-переменным i_broadcast и q_broadcast (рис. 5, строки 1,2), расчет амплитуд, их частотного распределения и кумулятивного накопления (рис. 5, строки 5–9).

Далее, в методе get_ds_within_window() для каждого пикселя в окне рассчитывается модуль

разности соответствующих относительных частот и его наибольшее значение Dmax (рис. 6, строка 13), вычисляется эмпирическое значение критерия lambda_emp (рис. 6, строка 14). Если эмпирическое значение критерия lambda_emp меньше критического lambda_crit, то индекс текущего пикселя добавляется в массив window_ds_pixels (рис. 6, строка 15, 16). После того как найдены все пиксели-кандидаты в распределенные отражатели в окне сдвига необходимо отсеять те, которые не являются смежными с центральным.

3. Алгоритм поиска смежных пикселей строится на базе k-мерного дерева, используя объект scipy.spatial.cKDTree, которому передается массив window_ds_pixels (рис. 7, строка 21). Метод query_ball_point() объекта cKDTree (рис. 7, строка 32), позволяет находить ближайшие соседние точки к заданной с координатами (x,y), находящиеся на определенном расстоянии в декартовой системе координат. Данный метод возвращает список координат (x,y) всех соседних элементов, принадлежащих k-мерному дереву. Примем расстояние (радиус) D равным $\sqrt{2}$, т.е. значения координат (x, y) смежных пикселей отличаются ровно на 1.

Создается массив window_neighbors, тип List, с начальным элементом – индексом центральной

```

def get_sn_vec(pixel):
1   pixel_i = i_broadcast.value[pixel]
2   pixel_q = q_broadcast.value[pixel]
3   amplitude_vec = []
4   for i in [*range(N_broadcast.value)]:
5       amplitude_vec.append(np.sqrt(pixel_i[i] ** 2 + pixel_q[i] ** 2))
6
7   sort_ampl = np.sort(amplitude_vec)
8   hist, bin_edges = np.histogram(sort_ampl, bins=8)
9   sn_vec_p = np.cumsum(hist / N_broadcast.value)
10
11  return sn_vec_p
    
```

Рис. 5. Метод get_sn_vec().

```

def get_ds_within_window(central_pixel):
1   lambda_crit = 0.75
2
3   sn_vec_p0 = get_sn_vec(central_pixel)
4   center_y, center_x = central_pixel // img_width_broadcast.value, \
5       central_pixel % img_width_broadcast.value
6
7   window_ds_pixels = []
8   for i in range(-6, 7):
9       for j in range(-8, 9):
10          pixel_pk = (center_y + i) * img_width_broadcast.value
11             + (center_x + j)
12          sn_vec_pk = get_sn_vec(pixel_pk)
13          Dmax = np.max(np.abs(np.array(sn_vec_p0) - np.array(sn_vec_pk)))
14          lambda_emp = Dmax * np.sqrt(N_broadcast.value / 2)
15          if lambda_emp < lambda_crit:
16              window_ds_pixels.append([center_y + i, center_x + j])
17
18          .....
19          .....
    
```

Рис. 6. Фрагмент метода get_ds_within_window().

точки. Он будет содержать все индексы смежных пикселей, после завершения работы метода. Так как метод `query_ball_point()` принимает координаты точек в формате (x,y), то на каждом шаге итерации будем преобразовывать индексы элементов массива `window_ds_pixels` в такой формат (рис. 7, строки 28, 29).

Создается временный массив `tmp_point_neighbors` (рис. 7, строка 26), который на каждой итерации для каждой точки из `window_ds_pixels` будет заполняться смежными с ней. Создается массив `point_neighbors`. Он будет содержать индексы точек, которых еще нет в основном массиве `window_neighbors`, и, соответственно, дополнять основной массив ими.

На первой итерации выполняется метод `query_ball_point()` для центрального пикселя в массиве `point_neighbors`. Получается модифицированный массив `tmp_point_neighbors`, содержащий не только координаты центральной точки, но и координаты смежных пикселей, находящихся на расстоянии

$\sqrt{2}$ от нее (рис. 7, строки 30–33). Ищется различие массива `tmp_point_neighbors` с основным `window_neighbors` и записывается в массив `point_neighbors` (рис. 7, строки 35–38). Соответственно, расширяется массив `window_neighbors` (рис. 7, строки 39).

Следующая итерация начинается с проверки длины массива `point_neighbors`. Если длина массива больше 0, т.е. на предыдущей итерации были найдены смежные пиксели, отличные от уже содержащихся в массиве `window_neighbors` по значениям индексов, то выше описанная процедура повторяется. Другими словами, область поиска в окне сдвига расширяется на расстояние $\sqrt{2}$ от центральной (рис. 8).

В конечном итоге данный итерационный метод позволит перебрать каждый пиксель-кандидат из массива `window_ds_pixels`, достигнув самых отдаленных пикселей от центрального. То есть, метод `numpy.setdiff1d()` (рис. 7, строка 35) вернет пустой массив `point_neighbors`, так как вновь найденные смежные точки уже будут содержаться в


```

def get_ds_within_window(central_pixel):
18     .....
19     .....
20
21     window_neighbors = [center_y * img_width_broadcast.value + center_x]
22
23     tree = cKDTree(np.array(window_ds_pixels))
24     point_neighbors = [center_y * img_width_broadcast.value + center_x]
25     while len(point_neighbors) > 0:
26         tmp_point_neighbors = np.array([[center_y, center_x]])
27         for point in point_neighbors:
28             point = [point // img_width_broadcast.value,
29                     point % img_width_broadcast.value]
30             tmp_point_neighbors = np.concatenate(
31                 (tmp_point_neighbors, np.array(tree.data[
32                     tree.query_ball_point(point, r=np.sqrt(2))
33                 ])).astype('int32'))
34
35     point_neighbors = np.setdiff1d(
36         [tmp_point_neighbor[0] * img_width_broadcast.value
37          + tmp_point_neighbor[1]
38          for tmp_point_neighbor in tmp_point_neighbors], window_neighbors)
39     window_neighbors.extend(point_neighbors)
40
41     return window_neighbors

```

Рис. 7. Фрагмент метода `get_ds_within_window()`.

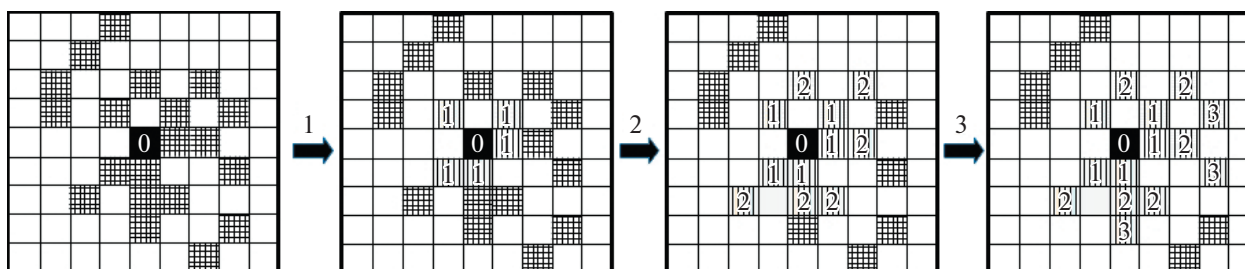


Рис. 8. Схем поиска смежных пикселей. Цифрой обозначен номер итерации и, соответственно, пиксели, найденные как смежные с центральным на текущей итерации.

основном массиве `window_neighbors`. Итерации прекращаются.

4. Если количество смежных пикселей в сформированном массиве `ds_pixels` (рис. 4, строка 5) больше заданной переменной `Np_broadcast` (не менее 20), выполняем расчет уточненных значений вектора фаз θ .

Рассчитываются нормированные комплексные значения (3) (\hat{p}_i и \hat{p}_i^+) каждого пикселя из массива `ds_pixels`, метод `get_norm_complex_sn()` (см. исходный код, файл `main_parallel.py`). Рассчитывается матрица когеренции T (3) (рис. 4, строки 11–12, тип `complex`). Согласно математической модели алгоритма задается начальное значение для θ , переменная `theta_initial` (рис. 4, строки 13).

5. Для оценки значений интерферометрических фаз каждого центрального пикселя из `central_pixels_slices_broadcast` для соответствующего DS-набора `ds_pixels` используется объект `scipy.optimize`. Объект `optimize` не содержит методов, позволяющих решать задачу максимизации. Поэтому применяется метод `minimize()` (рис. 4, строки 16–21) для решения задач минимизации. Другими словами, мы минимизируем функцию $-F_{max}$ из (8) (параметр `fun`) для всех θ (θ), при заданном ограничении $-\pi \leq \theta \leq \pi$ (параметр `bounds`). Параметр `method` задает тип решателя для проблемы минимизации. В данном случае модификация метода BFGS с возможностью задания нижней и верхней границ поиска решения. Параметр `x0` — вектор начального приближения, равен `theta_initial`. В предположении того, что изначальные фазы уже являются оптимальными для достаточно

```
def pta_test(theta, phase, N):
    s = 0
    for n in range(N):
        for k in range(n + 1, N):
            s = s + np.exp(1j * phase[n, k]) * \
                np.exp(-1j * (theta[n] - theta[k]))
        test_value = 2 * np.real(s) / (N ** 2 - N)
        if test_value > 0.5:
            return True, test_value
        else:
            return False, test_value
```

Рис. 9. Метод pta_test().

большого значения отношения сигнал/шум в стеке интерферометрических изображениях [1, 15]. Параметром `jac_fun` задается якобиан (метод `jacobian()`, см. исходный код) для функции параметра `fun` (метод `objective()`, см. исходный код). Данные методы принимают на вход искомое значение (приближение на каждой итерации метода `minimize()`), а на выходе возвращают вычисленное значение согласно выражению внутри этих методов. То есть, это значение функции $-F_{max}$ (8) и ее якобиана (см. исходный код), вычисленные при заданных параметрах (`args`) равных `numpy.angle(T)` и `gamma` (рис. 4, строки 15, 16) и векторе искомых значений `theta` на каждой итерации.

6. Качество полученной оптимизации `theta` оценивается на основе специального теста (метод `pta_test()`) (рис. 9).

7. Если массив `theta` прошел тест, то все его значения в текущем окне сдвига добавляются в специально созданный массив `theta_slice` (тип `List`) вместе с индексами смежных пикселей (массив `ds_pixels`). Если нет, то в массив `theta_slice` добавляется пустой массив, ровно как и в случае длины массива `ds_pixel` меньше чем заданное пороговое значение переменной `Np_broadcast` (рис. 4, строки 24–29).

Массив `theta_slice` содержит все результаты работы одного задания в среде Apache Spark, соответствующие текущему разделу.

Для объединения результатов в единый массив `theta_entire` используется метод объекта `rdd.reduce()` (рис. 3, строки 29, 30). Массив `theta_entire` сохраняется в бинарный файл в программе-драйвере (см. исходный код, файл `main_parallel.py`). В дальнейшем из этого файла считываются значения `theta` всех центральных пикселей и индексы смежных точек `ds_pixels`. Значения в каналах I и Q исходных снимков заменяются соответствующими расчетными значениями. При этом вещественная часть (полоса I) будет равна $\cos(\theta)$, а мнимая (полоса Q) – $\sin(\theta)$ (см. исходный код, файл `main.py`, метод `modify_images()`).

Ниже приведена диаграмма потоков данных (DFD) при выполнении программного кода алгоритма поиска распределенных рассеивателей (рис. 10).

4. ТЕСТ ПРОИЗВОДИТЕЛЬНОСТИ

Запуск программы-драйвера (см. исходный код, файл `main_parallel.py`), содержащего код (рис. 3) выполняется при помощи скрипта `spark-submit` [10] (рис. 11).

Параметр `spark.driver.memory` задает количество оперативной памяти, выделяемое для программы драйвера на управляющем узле Spark/Yarn (Master Node). Данный параметр влияет на объем предварительно подготавливаемых данных перед непосредственно их копированием на исполняющие узлы кластера. Устанавливать значение параметра необходимо с учетом суммарного объема всех переменных определяемых вне метода `map()`. Это переменные `i`, `q`, `lines`, `samples`, `N`, `Np`, `central_pixels`, `central_pixels_slices` (рис. 3).

Параметр `spark.executor.memory` задает количество оперативной памяти, выделяемое для каждого контейнера-исполнителя (Executor). При этом на одном узле-исполнителе (Worker Node) менеджером ресурсов Yarn может быть создано несколько контейнеров-исполнителей. Следовательно, общее количество выделяемой оперативной памяти на узле может быть увеличено на величину равной `spark.executor.memory * (кол-во контейнеров-исполнителей)`. Также, данный параметр должен коррелировать с количеством создаваемых `broadcast`-переменных, т.к. каждая переменная копируется для каждого контейнера-исполнителя [10]. Это переменные `i_broadcast`, `q_broadcast`, `img_height_broadcast`, `img_width_broadcast`, `N_broadcast`, `Np_broadcast`, `central_pixels_slices_broadcast` (рис. 3).

Параметр `spark.executor.instances` задает общее количество контейнеров-исполнителей. При выборе значения данного параметра необходимо учитывать общий объем оперативной памяти, необходимой для всех `broadcast`-переменных. Непосредственно каждый физический узел должен об-

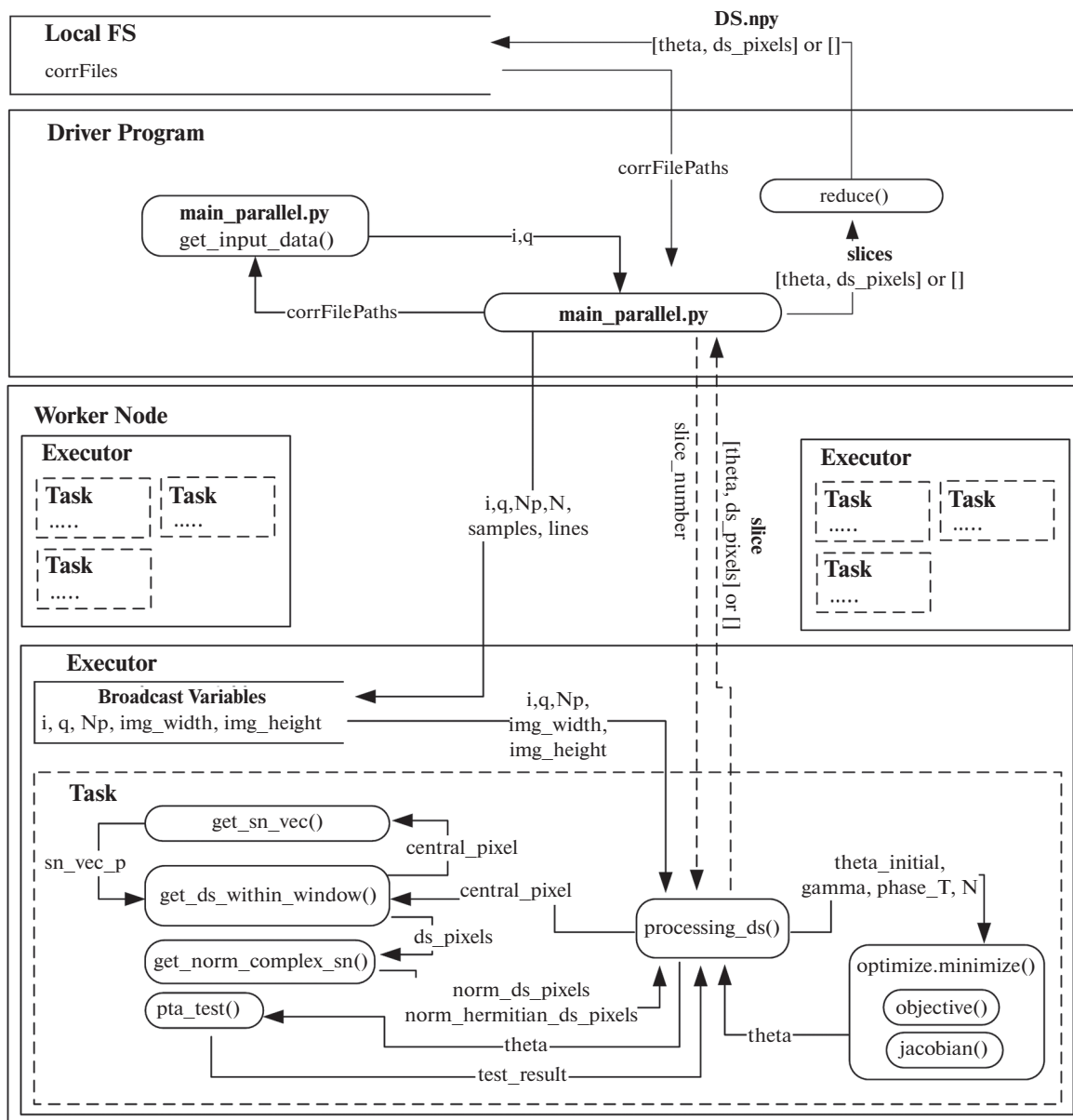


Рис. 10. DFD-диаграмма выполнения программного кода алгоритма.

```

1  spark-submit
2  --conf spark.driver.memory=8g
3  --conf spark.executor.memory=8g
4  --conf spark.executor.instances=6
5  --conf spark.executor.cores=5
6  --master yarn-client
7  main_parallel.py /path_to_input_dir /path_to_output_dir

```

Рис. 11. Команда запуска расчетного задания на кластере Apache Spark/Yarn.

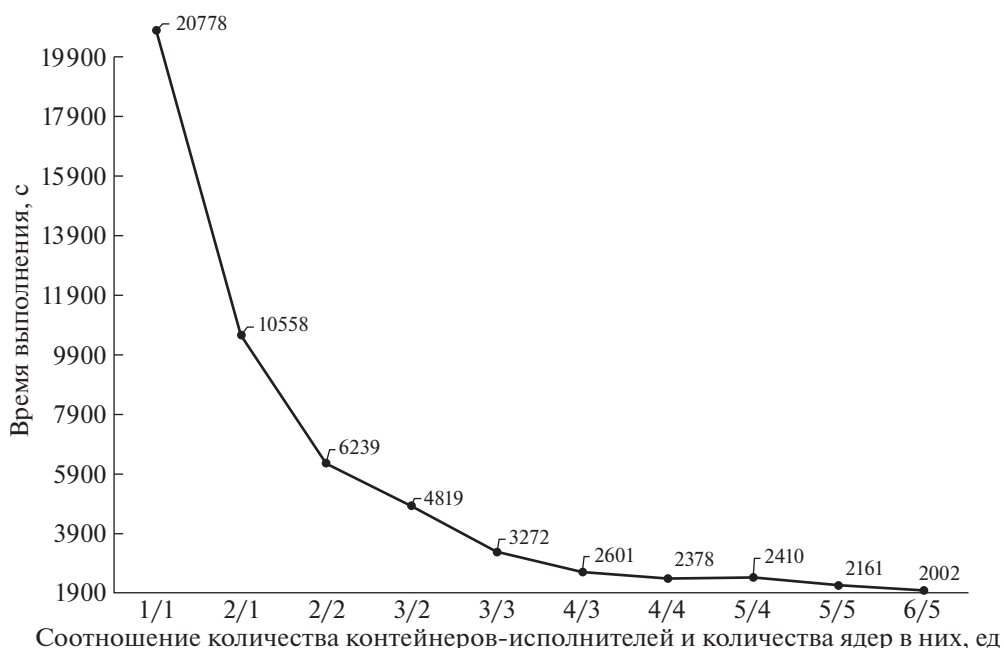


Рис. 12. График зависимости времени выполнения расчета от количества контейнеров-исполнителей и количества ядер процессора на один исполнитель для всей области снимка.

ладать памятью способной разместить объем равный `spark.executor.instances` *(суммарный размер в байтах `broadcast`-переменных).

Параметр `spark.executor.cores` задает количество ядер процессора на один исполнитель. При этом параметр неявно задает количество заданий `Task`, которое сможет выполнить исполнитель за один запуск. То есть в контексте алгоритма – это будет количество разделов (`slices`), для которых ведется расчет всех центральных точек в текущем разделе (рис. 10).

Тест производительности проводился для области снимка размером 6000×4000 пикселей, стек из 50 снимков ($N = 60$), на кластере, состоящем из 3 узлов со следующими аппаратными характеристиками: 3 сервера (AMD Ryzen 1700 (8 + 8 cores (Simultaneous Multi-Threading)) 3.2 GHz, 32 Gb RAM, 1 Gb/s скорость передачи данных между серверами). Один узел выступал в роли управляющего, два других – в роли узлов-исполнителей. Именно на них проводился расчет. В скрипте запуска `spark-submit` менялись значение параметров `spark.executor.instances` и `spark.executor.cores`. Остальные параметры были зафиксированы со значениями: `spark.driver.memory=8g` и `spark.executor.memory=8g` (рис. 11).

Ниже праведен график времени выполнения алгоритма с учетом вариативности вышеописанных параметров (рис. 12).

5. ПРИМЕНЕНИЕ АЛГОРИТМА SqueeSAR

Расчет скоростей смещений для тестовых радарных данных проводился в открытом пакете библиотек `StamPS` [16]. В стандартную схему пред- и постобработки был добавлен дополнительный шаг “`SqueeSAR`” (рис. 13), проводящий расчет `DS`-наборов и заменяющий исходные значения интерферометрических фаз их оптимальными значениями. Исходными данными для полного цикла расчета скоростей смещений методом постоянных рассеивателей (`PS`) послужили спутниковые радарные снимки аппарата `Sentinel-1B`. Для дифференциальной интерферометрии использовался канал типа `IW` (`Interferometric Wide swath`) с вертикальной поляризацией `VV`. Территория охвата – г. Норильск, и прилегающие территории (`ТЭЦ-3`), Россия. Данные были получены с открытого ресурса в сети Интернет “`Copernicus Open Access Hub`” (URL: <https://scihub.copernicus.eu/dhus/#/home>) Европейского космического агентства (URL: <https://sentinel.esa.int/web/sentinel/home>). Всего получено 60 снимков временной серии с февраля 2019 г. до апреля 2020 г. Модифицированная схема полного цикла расчета скоростей смещений представлена на рис. 13. Полное описание методов схемы приводится в [17, 18].

Рассматривалась географическая область, где 29 мая на территории `ТЭЦ-3` в Норильске произошла утечка из резервуара, в котором было около 21 тысячи тонн дизельного топлива. На рис. 14 приведены расчеты смещений для аварийных ба-

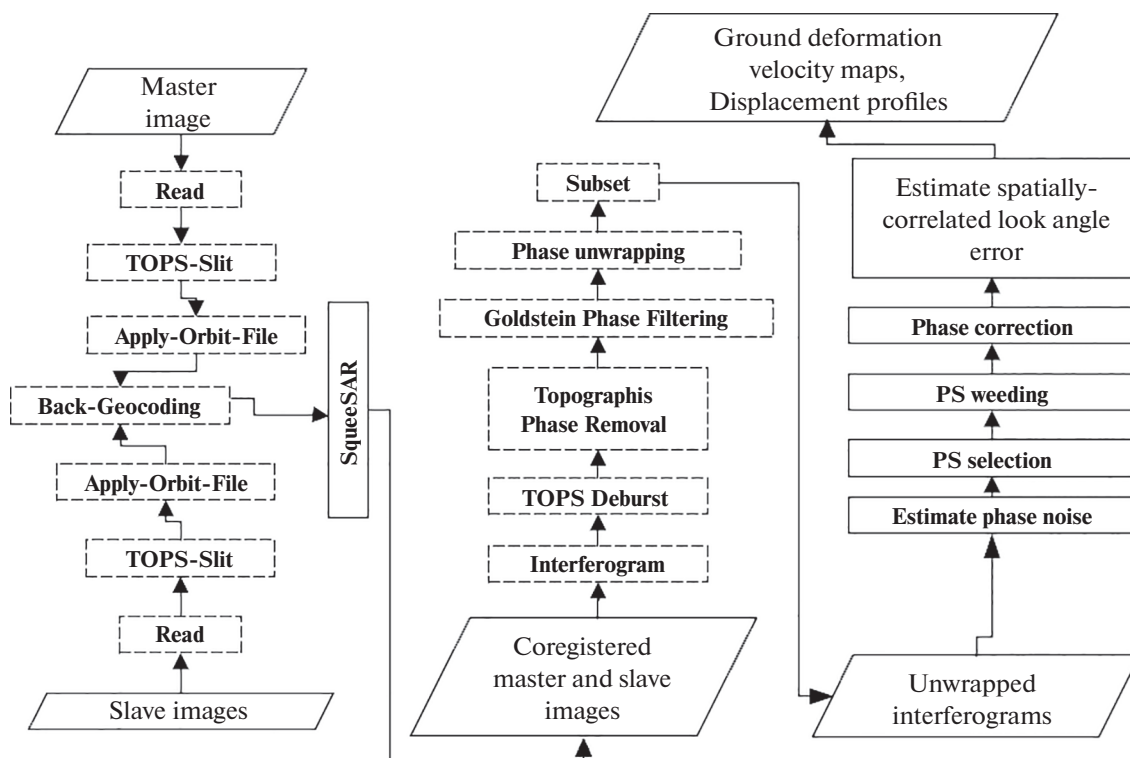


Рис. 13. Модифицированная схема полного цикла расчета скоростей смещений методом постоянных рассеивателей.

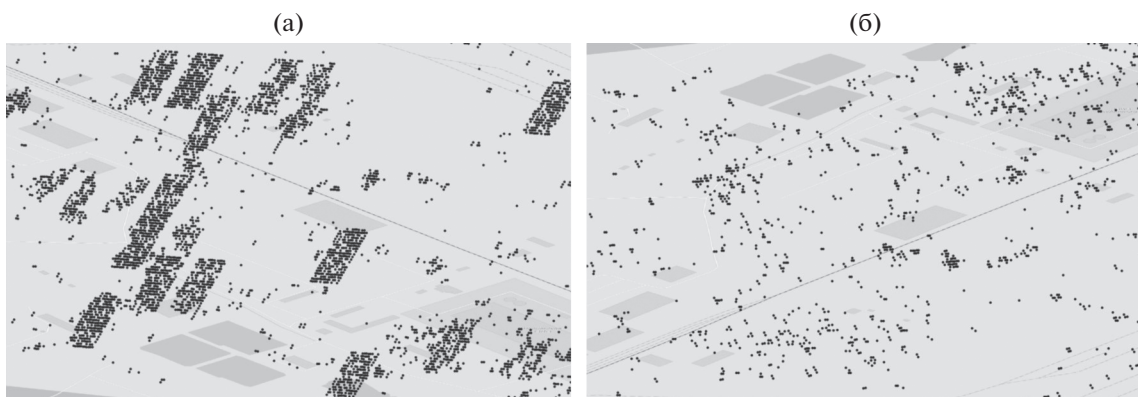


Рис. 14. Результат расчета полной схемы скоростей смещений с интегрированным алгоритмом SqueeSAR а) и без б).

ков. Показано облако точек (постоянных рассеивателей) с применением метода SqueeSAR и без него. Общее количество точек для расчетной области возросло в среднем до 130000 точек с интегрированным методом SqueeSAR, против 53000 в стандартном выполнении схемы.

Ниже приведен фрагмент области снимка для визуального сравнения облака постоянных рассеивателей с представленным методом и без него (рис. 14).

6. ЗАКЛЮЧЕНИЕ

Разработан быстрый алгоритм поиска распределенных отражателей для задачи построения скоростей смещений методом постоянных рассеивателей. Предложенный алгоритм позволяет увеличить количество искомым рассеивателей в два-три раза по сравнению с традиционным алгоритмом. Примеры расчетов на спутниковых радарных данных Sentinel-1A/B показывают увеличенную плотность и компактность точек расположения в местах возвышенностей и средних

построек (от 10 м). Это позволяет точнее строить карты цифровых моделей рельефа на базе временных серий снимков.

Показано, что предложенный алгоритм не является итерационным, и может быть реализован в парадигме параллельных вычислений. Применяемая платформа параллельной обработки Apache Spark позволила распределено обрабатывать массивы стека радарных данных (от 60 изображений) в памяти, на большом количестве физических узлов в сетевой среде. При этом время поиска распределенных рассеивателей удалось снизить в среднем в 10 раз по сравнению с однопроцессорной реализацией алгоритма.

Предложенный алгоритм и его параллельная имплементация позволят применять разработанные подходы в других задачах и типах спутниковых данных дистанционного зондирования земли из космоса.

7. БЛАГОДАРНОСТИ

Исследование выполнено при финансовой поддержке РФФИ и Кемеровской области в рамках научного проекта № 20-47-420002 р_а.

СПИСОК ЛИТЕРАТУРЫ

1. *Ferretti A., Prati C., Rocca F., Wasowski J.* Satellite interferometry for monitoring ground deformations in the urban environment // Proc. of the 10th Congress of the International Association for Engineering Geology and the Environment (IAEG). 2006. P. 1–4.
2. *Crosetto M., Monserrat O., Cuevas-González M., Devanthéry N., Crippa B.* Persistent Scatterer Interferometry: A review // ISPRS Journal of Photogrammetry and Remote Sensing. 2016. V. 115. P. 78–89.
3. *Perissin D., Ferretti A.* Urban target recognition by means of repeated spaceborne SAR images. IEEE Trans. Geosci. Remote Sens. 2007. V. 45. № 12. P. 4043–4058.
4. *Hooper A.A.* multi-temporal InSAR method incorporating both persistent scatterer and small baseline approaches // Geophysics Research Letter. 2008. V. 35. P. 302
5. *Rocca F.* Modeling interferogram stacks // IEEE Transactions on Geoscience and Remote Sensing. 2007. V. 45. № 10. P. 3289–3299.
6. *Zebker H.A., Shanker A.P.* Geodetic imaging with time series persistent scatterer InSAR // American Geophysical Union all Meeting Abstracts. San Francisco. CA. 2008.
7. *Ferretti A., Fumagalli A., Novali F., Prati C., Rocca F., Rucci A.* Exploitation of distributed scatterers in interferometric data stacks // International Geoscience Remote Sensing Symposium (IGARSS). Cape Town. South Africa. 2009.
8. *Ferretti A., Fumagalli A., Novali F., Prati C., Rocca F., Rucci A.* The second generation PSInSAR approach: SqueeSAR. International Workshop ERS SAR Interferometry (FRINGE). Frascati. Italy. 2009.
9. *Ferretti A., Fumagalli A., Novali F., Prati C., Rocca F., Rucci A.* A New Algorithm for Processing Interferometric Data-Stacks: SqueeSAR // IEEE Transactions on Geoscience and Remote Sensing. 2011. V. 49. P. 3460–3470.
10. Apache Spark Documentation // Apache Spark. <https://spark.apache.org/docs/latest/index.html> (дата обращения: 12.09.2020)
11. *Феоктистов А.А., Захаров А.И., Денисов П.В., Гусев М.А.* Исследование зависимости результатов обработки радиолокационных данных ДЗЗ от параметров обработки. Часть 4. Основные направления развития метода постоянных рассеивателей; ключевые моменты методов SQUEESAR И STAMPS // Журнал радиоэлектроники [электронный журнал]. 2017. № 7. <http://jre.cplire.ru/jre/jul17/5/text.pdf>
12. *Cao N., Lee H., Chul Jung H.* Mathematical Framework for Phase-Triangulation Algorithms in Distributed-Scatterer Interferometry // IEEE Geoscience and remote sensing letters. V. 12. № 9. P. 1838–1842.
13. *Guarnieri A.M., Tebaldini S.* On the exploitation of target statistics for SAR interferometry applications // IEEE Transactions on Geoscience and Remote Sensing. 2008. V. 46. № 11. P. 3436–3443.
14. *Bamler R., Hartl P.* Synthetic aperture radar interferometry // Inverse Problems. 1998. V. 14. № 4. P. R1–R54.
15. *Shamshiri R., Nahavandchi H., Motagh M., Hooper A.* Efficient Ground Surface Displacement Monitoring Using Sentinel-1 Data: Integrating Distributed Scatterers (DS) Identified Using Two-Sample t-Test with Persistent Scatterers (PS) // Remote Sensing. 2018. V. 10. № 5. P. 794–808.
16. STAMPS // A software package to extract ground displacements from time series of synthetic aperture radar (SAR) acquisitions. <https://homepages.see.leeds.ac.uk/~earahoo/stamps/> (дата обращения: 12.09.2020)
17. *Потанов В.П., Попов С.Е., Костылев М.А.* Информационно-вычислительная система массивно-параллельной обработки радарных данных в среде Apache Spark // Вычислительные технологии. 2018. Т. 23. № 4. С. 110–123.
18. *Попов С.Е., Замараев Р.Ю., Миков Л.С.* Массово-параллельный подход к обработке радарных данных // Современные проблемы дистанционного зондирования Земли из космоса. 2020. Т. 17. № 2. С. 49–61.