УДК 004.434:004.75 https://doi.org/10.15407/pp2021.01.049

В.О. Ларін, О.І. Провотар

ПОРІВНЯННЯ ЕФЕКТИВНОСТІ ПІДХОДІВ MAP-REDUCE І АКТОРНОЇ МОДЕЛІ ПРИ РОЗВ'ЯЗАННІ ЗАДАЧ ІЗ ВИСОКОЮ ЗВ'ЯЗНІСТЮ ВХІДНИХ ДАНИХ НА ПРИКЛАДІ ЗАДАЧІ ОПТИМІЗАЦІЇ РОЮ ЧАСТОК

Показані приклади класу розподілених паралельних задач зі зв'язаними компонентами вхідних даних в рамках моделі Map-Reduce. Виконано порівняння ефективності подібної задачі на прикладі задачі рою часток в рамках моделі Map-Reduce (на основі фреймворку Spark) і акторної моделі з підтримкою спільної пам'яті (на основі Strumok DSL). Оцінено перспективи використання гібридної акторної моделі для інших подібних задач.

Ключові слова: акторна модель, Map-Reduce, оптимізація методом рою частинок, паралельні системи із загальною пам'яттю.

Вступ

В даний час модель Map-Reduce, набула широкого поширення в різних областях – бізнес аналітика, розподілені обчислення, обробка даних для прикладних задач (медицина, комерція, фінанси і т. д.). В основі цієї моделі лежить принцип паралельної обробки і агрегації лінійних масивів даних. Для обробки масиву даних необхідно задати дві функції: map, яка для кожного вхідного елемента співставить від 0 до n вихідних пар ключ-значення, і reduce, яка агрегує всі значення, що відповідають одному ключу [1–3]. При цьому допускається що дана модель – це універсальний примітив паралельних обчислень [4]. Альтернативний підхід – це модель акторів, винайдена в 1973 році, представляючи математичну модель, яка трактує поняття «актор» як універсальний примітив для паралельних обчислень: у відповідь на повідомлення, які він отримує, актор може змінити свій локальний стан, створити нові актори, і відправити повідомлення тим акторам яких він знає [5]. Дана модель залишалася в обмеженому застосуванні до появи фреймворків Akka [6], Akka .NET, Orleans, які значно спростили її практичне використання. При більшій гнучкості моделі акторів (контроль життєвого циклу кожного актора; маршрутизації повідомлень; контрольований супервізор для нештатних ситуацій) залишається питання – наскільки доцільно застосування набагато більш комплексного підходу при вирішенні прикладних задач. Далі розглядатимемо обмеження моделі map-reduce і представлені рішення як для моделі акторів (на основі раніше запропонованої предметно-орієнтованої мови Strumok [7]), так і у вигляді ациклічно направленого графа з серій викликів map-reduce використовуючи фреймворк Apache Spark на прикладі задачі оптимізації методом рою часток.

Задачі зі зв'язаними компонентами вхідних даних і визначення їх в термінах Map-Reduce і акторів

Структура підходу map-reduce передбачає наявність всіх вхідних параметрів перед викликом безпосередньо map-reduce алгоритму. Але не для кожного завдання таке визначення всіх вхідних елементів є можливим. Припустимо, у нас є задача, задана в такий спосіб:

$$S = F(V), V = \{v_i\}_{(i=1)}^{N}.$$

В даному вираженні V вектор великої розмірності $\mathcal{R}^N$. Нехай $O(f) \gg O(F)$, тобто основна обчислювальна складність полягає у знаходженні V, знаючи який пошук $S = F(V)$ не складає великої обчислювальної складності. Масив компонент V

представлений функціонально залежними елементами, які можна обчислити ітераційно:

$$v_i = \{a_{(i,j)}\}_{(j=0)}^{M}, a_{(i,k+1)} = f(a_{(i,k)}),$$

$$a_{(i,0)} = a_i, \text{ де } f: \mathcal{R} \to \mathcal{R}.$$

Така ітерація не може бути представлена одним викликом map-reduce, так як кожен наступний елемент ітерації залежить від попереднього, що в свою чергу порушує головну вимогу операції map – незалежність вхідних елементів один від одного. Але водночас, ми можемо її реалізувати як серію викликів окремих map-reduce підзадач.

Таким чином, в разі якщо обчислення f може бути представлено як незалежне обчислення кожної компоненти вектора $a_{(i,k)}$ з простору $\mathcal{R}^{\mathcal{M}}$, то задача знаходження S у цілому зводиться до ациклічного направленого графу (дерева) глибини N, де елемент буде являти собою задачу map-reduce вигляду $a_{(i,k+1)} = f(a_{(i,k)})$, $\forall i = 1..N$ для вершин із ступенем 1, і $S = F(V)$ для крайньої вершини.

Задача оптимізації рою часток (Particle Swarm Optimization) як приклад класу задач з пов'язаними компонентами вихідних даних

Метод рою часток (МРЧ) – метод чисельної оптимізації, для використання якого не потрібно знати точного градієнта функції що оптимізується [8–10]. МРЧ спочатку призначався для імітації соціальної поведінки [11, 12]. Алгоритм був спрощений, і було помічено, що він придатний для виконання оптимізації. МРЧ оптимізує функцію, підтримуючи популяцію можливих рішень, які називаються частками, і переміщує ці частинки в просторі рішень згідно простої формули. Переміщення підкоряються принципу найкращого знайденого в цьому просторі положення, яке постійно змінюється при знаходженні частками більш вигідних положень. Розглянемо структуру обчислень в цьому алгоритмі. Ітераційно оновлюється набір часток – «рій», які поступово наближаються до екстремумів досліджуваної функції рухаючись з певною швидкістю щодо своїх минулих позицій. Неважко помітити, що така структура співвідноситься з обчисленням масиву залежних елементів і поданням його як ациклічного дерева обчислень у моделі map-reduce. Але, додатково до цього «рій» частинок вносить ще один елемент оптимізації – швидкість руху частинок, а таким чином і кількість ітерацій, залежить від поточного кращого знайденого екстремуму. Це, в свою чергу, створює додатковий вид залежності між вхідними даними (в рамках ітерацій). Спробуємо представити даний параметр апроксимовано нашій моделі. Припустимо наявність зовнішнього для системи параметра L, який дозволяє знайти відразу наступний елемент ітерації:

$$a_{(i,k+2)} = f(a_{(i,k)}), a_{(i,k)} < \text{L}.$$

З цього співвідношення видно, що при хорошій оцінці глобального екстремуму L, рішення може бути знайдено при дворазовому скороченні кількості ітерацій. Таким чином для отримання оптимальної паралельної ефективності методу важливо якомога швидше оновлювати значення L. В реальних задачах рою частинок прискорення залежить від значення параметрів функції швидкості частинки і впливає на стабільність алгоритму. Розглянемо додавання параметра L у систему ациклічного графа map-reduce і акторної моделі. В процесі ітерацій рою часток кожне обчислення $a_{(i,k)}$ дозволяє оновити оцінку L. Оцінимо час синхронізації параметра в різних моделях.

У разі акторної моделі, параметр L може бути негайно оновлений для всіх акторів, які займаються ітеруванням часток. Розглянемо конфігурацію з одним обчислювальним вузлом. Витрати складуть час на доставку й обробку повідомлення будь-кому акторам. Цей час можна оцінити як час обробки одного значення $a_{(i,k)}$ кожним актором. При додаванні спільної пам'яті (Strumok) час оновлення стає

дуже малим і можна говорити про негайне оновлення, так як інші актори будуть безпечно звертатися до області загальної пам'яті, значення в якій можна оновити за кілька тактів процесорного часу.

У разі розподіленої конфігурації час оновлення складатиме часу затримки всередині кластера, і механізм спільної пам'яті дозволить зменшити кількість трафіку що пересилається, і прискорити оновлення в рамках кожного вузла кластера.

У моделі map / reduce, в свою чергу, кожен елемент map повинен зберігати всі дані необхідні для обчислення, таким чином оновлення параметра L можливо тільки між ітераціями, і, крім того, в розподіленої конфігурації час затримки кластера також накладається на швидкість оновлення, тому для подальшого порівняння цей параметр можна опустити, при цьому потенційно акторна модель дозволяє максимально оптимізувати кластерний трафік.

Загальний час оновлення необхідний системі на map-reduce можна оцінити в залежності від співвідношення кількості пар паралельних вузлів / процесорів до кількості часток в ітерації. Якщо співвідношення наближається до рівності (тобто при невеликій кількості часток у популяції) час оновлення map reduce стає відповідним з акторною моделлю. Водночас, при великій кількості часток, раннє оновлення L при перших обчисленнях $a_{(i,k)}$ не відбудеться до повного завершення ітерації, що, по суті, робить оновлення параметра неефективним для більшості часток в ітерації, і призводить до втрати прискорення, яке досягається оновленням L (дворазове в апроксимованому випадку).

Реалізація задачі рою частинок використовуючи підходи map-reduce (Spark) і акторної моделі (Strumok / Akka)

Тепер розглянемо практичну реалізацію даної задачі в рамках акторної моделі. Залежно від величини $\mathcal{N}$ можна делегувати по одному актору на кожен елемент v_i, або ж у разі великих значень $\mathcal{N}$ кожен актор може обробляти певний сегмент значень $v_i..v_{(i+l)}$ заданий при конфігуруванні акторної моделі. У першому (більш тривіальному) випадку логіку акторів можна визначити наступним псевдокодом:

```
Message "Update":
 If k == K then send "Compute
S" to root with Ai and i
 Else Ai = f(Ai); k = k + 1;
send "Update" to self
```

Даний псевдокод дозволяє відразу звернути увагу що проміжне значення $a_{(i,k)}$ зберігається всередині актора, завдяки чому для виконання кожній ітерації досить лише порожнього повідомлення-тригера, пересилати додаткові дані, в тому числі між кластером (в разі розподіленої архітектури) не потрібно. Крім того, кожен актор оновлюється асинхронно, додатковий бар'єр на очікування однієї повної ітерації V_k не потрібен. У той же час в парадигмі Map-Reduce, запуск кожної ітерації вимагає копіювання вхідних даних, ініціалізації завдання на виконавчих вузлах (executors), а також блокування сесії до повного завершення ітерації.

При використанні предметно орієнтованої мови Strumok завдання оптимізації методом рою часток можна представити таким чином:

```
package strumok.pso;

import
strumok.pso.TestFunction;

actor Particle {
  actorref GMinimum handler

  local double minimum
  local double x
  local double value
  local double velocity

  message update() {
    var approxG = ~handler.g
    if
(approxG<=handler.threshold)
    {
      return
    }
```

```
          var rp = Random.value
          var rg = Random.value
          velocity = velocity * 0.3 +
0.7 * rp * (minimum - value) + 1.3 *
rg * (approxG - value)

          x += velocity

          value = TestFunction(x)

          if (value < minimum) {
              minimum  = value
          }

          if (value < approxG) {

handler.updateMinimum(value)
          }
          update()
        }
      }
      actor GMinimum {
        actorref Swarm swarm

        shared double g
        shared double threshold

        message updateMinimum(double
newValue)
        {
          if (newValue < g) {
            g = newValue

            if (g <= threshold) {
               swarm.complete(g)
            }
          }
        }
      }

      actor Swarm {
          @AkkaRoutedPool(strategy =
RoutingStrategy.RoundRobin, size =
1000)
          actorref
Selection<Particle> swarm

          actorref GMinimum handler

          Swarm() {
            handler = new GMinimum()
            broadcast
swarm.initialize(handler)
            broadcast swarm.update()
          }

          message complete(double
result) {
            println("Result is " -
result);
          }
      }
```

Даний лістинг визначає 3 класи акторів: Swarm – керуючий, GlobalMinimum – зберігає поточний глобальний екстремум (параметр L у моделі), Particle – клас обчислювача в рамках ітерації. Частки розміщуються в акторному пулі (розміром 1000), який ініціалізується початковими значеннями часток, а потім оновлює їх ітераційно до досягнення зазначеного threshold (оцінки глобального мінімуму).

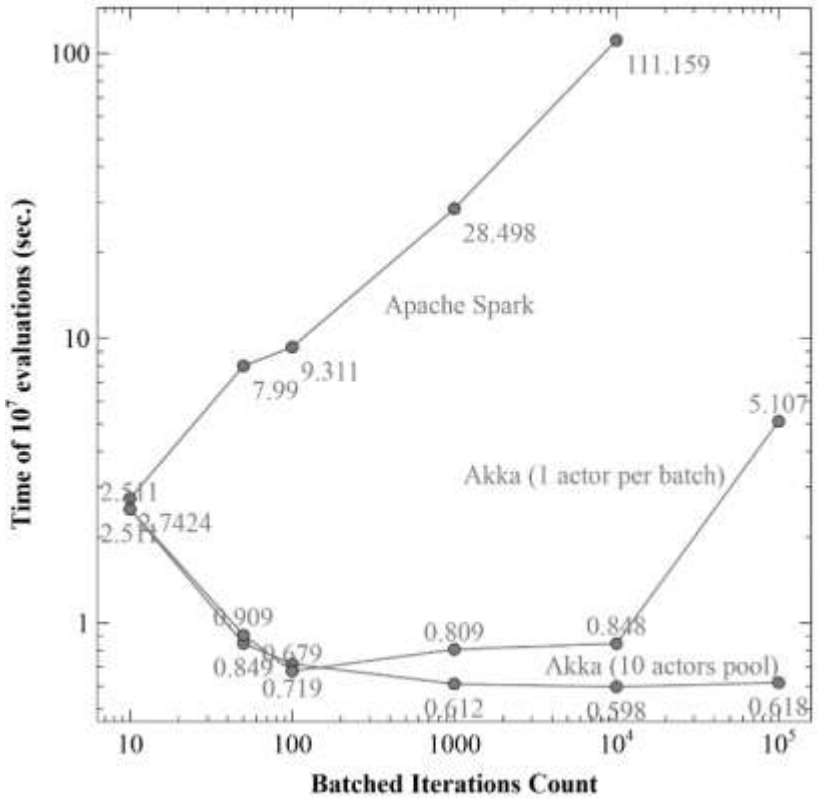
```
val minX = 0
val maxX = 10
val amount = 1000000
val startTime = System.currentTimeMillis()
var particles =
sc.parallelize(generateParticles(minX,
     maxX, amount)).repartition(8)
val globalMinimumValue: Double = parti
   cles.map(_.localMinimum).min()
val globalMinimum = new Minimum
   sc.register(globalMinimum, "minimum")
   globalMinimum.add(globalMinimumValue)

def iterate() = {
 val g = sc.broadcast(globalMinimumValue)
 particles = particles.map(p => {
  val rp = r.nextDouble()
  val rg = r.nextDouble()
  var velocity = p.velocity * 0.7 + 0.7 * rp *
(p.localMinimum - p.current) + 0.3 * rg * (g.value
- p.current)

  //handle out of bounds..
  if (p.x < minX && velocity < 0 || p.x > maxX
&& velocity > 0) {
```

```
    velocity = -velocity
   }

   val x = p.x + velocity
   val current = f(x)
   val minimum = if (current < p.localMinimum)
current
    else p.localMinimum
   if (current < g.value)
globalMinimum.add(current)
   Particle(x, velocity, current, minimum)
  }).localCheckpoint()

  particles.foreach(_ => {})
  globalMinimum.value
}
```

Лістинг вище визначає реалізацію ітерації методу рою часток для map-reduce фреймворка Apache Spark. Дана ітерація визначає глобальний лічильник globalMinimum, який оновлюється між ітераціями, і ітерацію map-reduce, де в map частини оновлюється значення частки, а reduce частина опущена (в реальному алгоритмі може бути вибір максимуму, реалізація якого тривіальна і може бути опущена в тестовій реалізації). Отриманий проміжний екстремум повертається назовні для повторних ітерацій.

Оцінка впливу зв'язності компонентів вхідних даних на ефективність Map-Reduce в порівнянні з акторним підходом

Для кожної версії реалізації задачі була проведена оцінка ефективності роботи. Зважаючи на велику чутливість алгоритму МРЧ до параметрів і недетермінованого характеру алгоритму в цілому, за основу оцінки було взято час обчислення 10 мільйонів часток при збільшенні сумарної кількості ітерацій. Більш складні задачі вимагають більшої кількості ітерацій для вирішення, таким чином ефективність на великих кількостях ітерацій дозволяє оцінити продуктивність системи в реальних умовах. З характеристиками стендової машини: Intel Core i7 3770K 3.5GHz, 16GB 2133 MHz RAM, 128 GB SSD на десяти тестових прогонах отримані наступні усереднені результати (шкали логарифмічні) (рисунок).



Рисунок

Як видно з графіку, Apache Spark дуже важко працює з довгими ациклічними графами виконання (на значеннях ітерацій в 1000 і вище, прискорення акторної моделі стає більш ніж на 2 порядки). Акторна модель реалізована на Strumok / Akka спочатку локально прискорюється (заповнюючи вільні обчислювальні потоки тестової машини), а після цього зберігає продуктивність і при високих розмірах задачі, а при використанні оптимізованої версії (коли кожен актор зберігає відразу кілька часток для поновлення) швидкість роботи зберігається стабільною при будь-якому зростанні ітерацій, які були в рамках експерименту.

Висновки

Перетворення над масивами незалежних наборів даних добре вирішуються в підході map-reduce, але при цьому при появі залежності між вхідними даними альтернативи, такі як модель акторів, а також модель акторів із спільною пам'яттю дозволяють отримувати вкрай істотні прирости в ефективності паралельних обчислень. Клас задач з пов'язаними вхідними даними добре представлений задачами глобальної оптимізації. Поведінка і ефективність такого класу задач були розглянуті на прикладі задачі оптимізації методом рою часток. На окремих конфігураціях прискорення може досягати десятків і сотень разів, що доводить значимість альтернативних підходів для подання паралельних обчислень. Крім того, залишається важливим вивчення залежностей в даних для кроку reduce, що так само може виявити клас задач, які ефективно вирішуються в рамках акторної моделі або подібних альтернатив, але при цьому помітно втрачають в ефективності в рамках map-reduce парадигми.

Література

1. Борис Т.В., Алєксєєв М.О. Порівняльний аналіз технології паралельного обчислення великих масивів даних MapReduce. *Proceedings of Second International Conference "Cluster Computing"*. Lviv , 2013. С. 54–57.
2. Гладкий М.В. Модель распределенных вычислений MapReduse. Труды БГТУ. 2016. Т. № 6. С. 194–198.
3. MapReduce – краткое руководство: веб-сайт. URL: https://coderlessons.com/tutorials/bolshie-dannye-i-analitika/izuchit-kartu-umenshit/mapreduce-kratkoe-rukovodstvo (дата звернення 15 07 2020).
4. Mateu Zaharia. An Architecture for Fast and General Data Processing on Large Clusters. Association for Computing Machinery and Morgan & Claypool. 2016. P. 89.
5. Глибовець М.М., Зінчук С.О. Використання моделі акторів для реалізації розподілених обчислень. *Системні дослідження та інформаційні технології*. 2015. № 2. С. 16–25.
6. Akka: веб-сайт. URL: https://akka.io/ (дата звернення 19.07.2020).
7. Ларін В.О., Бантиш О.В., Галкін О.В., Провотар О.І. Предметно-ориентированный язык Strumok для описания акторных систем с общей памятью. *Кибернетика и системный анализ*. 2018. Т. 54. С. 170–180.
8. Карпенко А.П, Селиверстов Е.Ю. Обзор методов роя частиц для задачи глобальной оптимизации (Particle Swarm Optimization). Наука и образование: электрон. науч.-тех. изд. URL: http://www.technomag.edu.ru (дата звернення 04 08 2020).
9. Курейчик В.В., Запорожец Д.Ю. Роевой алгоритм в задачах оптимизации. *Известия Южного федерального университета.* Технические науки. 2010 р. № 7. С. 28–32.
10. Kennedy J., Eberhart R. Particle Swarm Optimization. *Proceedings of the IEEE International Conference on Neural.* NJ : IEEE Press, 1995. P. 1942–1948.
11. Олійник О.О., Субботін С.О. Оптимізація на основі колективного інтелекту рою часток з керуванням зміною їхньої швидкості. *Радіотехніка. Інформатика. Управління*. 2009. № 2. С. 96–101.
12. Hoorfar A. Evolutionary programming in electromagnetic optimization: a review. *IEEE Trans. Antennas Propag*. 2007. Vol. 55. Iss. 3. P. 523–537.

References

1. Boris T.V., Alekseev M.O., Comparative analysis of MapReduce - technology for parallel computation of large data sets. *Proceedings of the Second International Conference "Cluster Computing"*. Lviv, 2013. P. 54–57.
2. Gladkiy M.V., Model of distributed calculations MapReduse. *Proceedings of BSTU*. 2016. N 6. P. 194–198.
3. MapReduce – a quick guide: a website. URL: https://coderlessons.com/tutorials/bolshie-dannye-i-analitika/izuchit-kartu-umenshit/mapreduce-kratkoe-rukovodstvo (accessed 15 07 2020).
4. Matthew Zechariah, An Architecture for Fast and General Data Processing on Large Clusters. Association for Computing Machinery and Morgan & Claypool. 2016. P. 89.
5. Hlybovets M.M., Zinchuk S.O., Using the model of actors for the implementation of distributed computing. *System research and information technology*. 2015. N 2. P. 16–25.
6. Akka: website. URL: https://akka.io/ (accessed 19 July 2020).
7. Larin V.O., Bantysh O.V., Galkin O.V., Provotar O.I., Object-oriented Strumok language for describing actor systems with shared memory. *Cybernetics and systems analysis.* 2018. Vol. 54. P. 170–180.
8. Karpenko A.P. and Seliverstov E.Y. “Review of particle swarm methods for the global optimization problem (Particle Swarm Optimization),” Nauka i obrazovanie: digital scientific and technical ed. URL: http://www.technomag.edu.ru (access date 04 08 2020).
9. Kureychik V.V., Zaporozhets D.Yu. Swarm algorithm in optimization problems. *Izvestiya Yuzhnogo federalnogo universiteta*. Technical sciences. 2010. Vol. 7. P. 28–32.
10. Kennedy J., Eberhart R. Particle Swarm Optimization. *Proceedings of the IEEE International Conference on Neural*. NJ: IEEE Press, 1995. P. 1942–1948.
11. Oliynyk O.O., Subbotin S.O., Optimization on the basis of collective intelligence of a swarm of particles with control of change of their speed. *Radiotechnics. Informatics. Management*. 2009. Vol. 2. P. 96–101.
12. Hoorfar A. Evolutionary programming in electromagnetic optimization: a review. *IEEE Trans. Antenna Propag*. 2007. Vol. 55. Iss. 3. P. 523–537.

Одержано 24.02.2021

Про авторів:

Провотар Олександр Іванович,
доктор фізико-математичних наук,
професор, завідувач кафедри
інтелектуальних програмних систем.
Кількість наукових публікацій в
українських виданнях – 150.
Кількість наукових публікацій в
зарубіжних виданнях – 45.
Н-індекс – 5,
http://orcid.org/0000-0002-6556-3264,

Ларін Владислав Олегович,
аспірант.
Кількість наукових публікацій в
українських виданнях – 4.
Кількість наукових публікацій в
зарубіжних виданнях – 1.
Тел.: +380972712208.
E-mail: vlarinmain@gmail.com.

Місце роботи авторів:

Київський національний університет
імені Тараса Шевченка,
03187, Київ-187,
Проспект Академіка Глушкова, 4д.
Факультет комп’ютерних наук
та кібернетики.