КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Києво-Могилянська академія

Факультет Інформатики

Курсова робота

на тему:

Реалізація PageRank алгоритму використовуючи розподілені обчислення

студента III курсу

Спеціальності Комп'ютерні Науки

Власенка Павла

Науковий керівник:

старший викладач, Борозенний

Сергій Олександрович

Завідувач кафедри

мультимежійних систем

доцент, кандидат наук. Жежерун

Олександр Петрович

Київ — 2022

Зміст

Розділ 1

Вступ

1.1 Анотація

У цій роботі буде розглянуто імплементацію алгоритму PageRank використовуючи підхід MapReduce, який дозволить розполалелити алгоритм щоб він працював одночасно на кількох комп'ютерах. Також буде опис досвіду створення кластеру з кількох комп'ютерів для виконання цього алгоритму.

1.2 Вступ

PageRank - алгоритм, створений корпорацією Google для однойменного пошукового рушія. Він є важливою частиною пошуку, адже завдяки йому визначається важливість і популярність кожного сайту, що проіндексований в рушії. Алгоритм працює на підрахунку посилань на певний сайт з інших сайтів. Хоча й оригінальну примітивну версію алгоритму легко написати в кілька стрічок коду, але враховуючи кількість сторінок в інтернеті, запускати алгоритм на одному комп'ютері, а тим паче, на одному потоці, абсолютно непрактично Тому у цій статті буде розглянуто розподілену модель обчислень, яка дозволяє реалізувати заданий

алгоритм щоб він працював паралельно на кількох комп'ютерах. Для цього буде використано підхід MapReduce, якому буде посвячений розділ 3 цієї роботи. Також у цьому розділі буде опис практичних проблем, що виникли при запуску алгоритму. Для реалізації MapReduce підходу буде використана акторна модель обчислень, якій буде посвячений розділ 2.

У останньому розділі 4 буде як детальний опис роботи PageRank алгоритму, так і його конкретна реалізація, яка використовує модель, побудовану у розділі 3. Також буде описаний досвід запуску алгоритму на кластері з кількох комп'ютерів

Розділ 2

Модель акторів

2.1 Фундаментальні концепти

Акторна модель за своїми концептами дуже нагадує об'єктно орієнтовану модель, імплементовану в мові Smalltalk Так, основою модеді є актори, які працюють незалежно один від одного і можуть:

- Відсилати повідомлення іншими акторам
- Створювати нові актори
- Мати різну поведінку, яку детермінують отримані актором повідомлення, тобто актори мають певний стейт

Завдяки тому що актори взаємодіють між собою лише через передачу повідомлень, зовсім неважливо щоб вони находились на спільному комп'ютері Актори можуть знаходитись на абсолютно різних комп'ютерах, а відсилати повідомлення один одному через мережу Це і є відмінною рисою акторної моделі, завдяки якій їх часто використовують для розподілених обчислень

2.2 Akka.NET

2.2.1 Akka.NET - вступ

Так як весь код для цієї роботи буде написаний на мові програмування C, то і бібліотеки будуть використані ті що створені для цієї мови

Akka.NET - імплементація акторної моделі для мов програмування C та F. Вона є портом бібліотеки Akka, яка була створена для мов на JVM, в першу чергу для Scala та Java. Akka, у свою чергу, була натхненна мовою програмування Erlang, у якій актори взагалі є First Class Sitizen.

Далі розглянемо основні класи та концепти, використані у Akka.NET

2.2.2 Akka.NET - актори

Щоб визначити актор, ми будемо використовувати клас `ReceiveActor` Він є наслідником від величезної ієрархії класів - більш низькорівневих акторів. Але детально вдивлятись у архітектуру бібліотеки у цій роботі ми не будемо, лише глянемо на те, що нам потрібно знати для реалізації MapReduce

Приблизно так має виглядати актор:

```
1 public class MyActor: ReceiveActor
2 {
3   private readonly ILoggingAdapter log = Context.
      GetLogger();
4
5   public MyActor()
6   {
7     Receive<string>(message => {
8       log.Info("Received String message: {0}",
          message);
```

```
9         Sender.Tell(message);
10      });
11      Receive<SomeMessage>(message => {...});
12    }
13  }
```

Метод `Receive<TMessage>(Action<TMessage> handler)` у конструкторі дозволяє встановити хендлер повідомлень, які надходять з інших акторів У нього передається лямбдя-функція, яка і містить сам код хендлінгу Receive - generic метод, тому ми можемо мати різну логіку для різних типів повідомлень. Це ми будемо часто використовувати, так як будемо задвати повідомлення за допомогою простих record-ів

Ось певні приклади повідомлень:

```
1     public record Map(ISourceRef<int> DocumentPaths
        );
2     public record Reduce(ISourceRef<byte[]>
        CompressedPairs);
3     public record Collect;
4     public record Done;
```

Akka.NET - IActorRef

`IActorRef` - посилання на певний актор. Може бути як актором на тій самій машині, так і бути актором десь у мережі. Наприклад, Sender у прикладі вище є інстансом `IActorRef` Найважливішим методом цього інтерфейсу є `Tell(object message)`, який відправляє повідомлення актору, на який цей `IActorRef` посилається.

2.2.3 Akka.NET - Streams

Так як одна нода може обробляти величезну кількість данних, їх не можна просто запихнути в одне повідомлення, стиснути і відправити по мережі. Крім того що це просто не дуже оптимізовано, це банально неможливо, адже пам'ять у комп'ютера обмежена. Розробники Akka про це подбали, реалізувавши потокивий підхід передачі данних, код якого знаходиться у просторі імен `Akka.Streams`.

Поток представляє з себе пайплайн акторів такого виду:

- `Source<TOut>` - джерело потоку, має хендлити повідомлення `RequestElem`, відправляючи на них повідомлення - елемент потоку.
- `Flow<TIn, TOut>` - має з'єднати `Source` з `Sink`, який як відправляє `RequestElem` до Source, так і хендлить `RequestElem` що прийшли від Sink. Має ключову логіку back-pressure, регулюючи скільки елементів генерує `Source`, скільки треба закешувати і скільки треба відправити `Sink`. У розділі MapReduce буде описана кастомна імплементація цього актору, необхідна для отпимізації алгоритму.
- `Sink<TIn>` - відправляє повідомлення `RequestElem` до Flow, щоб отримати потокові дані.

Back-pressure - основна фіча Akka.Streams, завдяки якій `Source` генерує стільки елементів потоку, скільки необхідно для `Sink`

Розділ 3

MapReduce

3.1 MapReduce - загальний опис підходу

MapReduce - програмна модель і підхід, розроблений для проведення розподіленої паралельної обробки величезної кількості данних, з використанням кластерів. Як і PageRank, був розроблений компанією Google.

Програма, реалізована з MapReduce, має три головні частини: `map`, `reduce` та `shuffle` Алгоритм був натхненним однойменними функціями з функційоного програмування.

Так, `map` - функція що приймає аргументом список, функцію що робить відображення з одного елементу списку в інший, а вертає список, для кожного елементу якого було виконано це відображення. Її сигнатуру можна описати так:

```
1 map :: [a] -> (a -> b) -> b
2 map [1,2,3,4,5] (+1) = [2,3,4,5,6]
```

У свою чергу `reduce`, а також як її ще часто називають - `fold`, `accumulate`, `aggregate` - це функція, що приймає список, початковий елемент, і аккумулюючу функцію. Її опис складніше за сигнатуру, а найлегше її зрозуміти побачивши приклади:

```
1 fold :: (b -> a -> b) -> b -> [a] -> b
2 fold (+) 0 [1,2,3,4,5] = 0 + 1 + 2 + 3 + 4 + 5
3 fold (+) 10 [1,2,3,4,5] = 10 + 1 + 2 + 3 + 4 + 5
4 fold (*) 10 [1,2,3,4,5] = 10 * 5!
```

І так, три основні кроки MapReduce алгортму:

- Map - кожна нода бере певну кількість данних, розраховану саме на неї і виконує до них функцію `map`. Дані, оброблені `map` функцією вертаються у вигляді пари ключ-значення і записуються у тимчасове сховище.

- Shuffle - кожна reduce нода обробляє певний набір данних за певним ключем. Тобто усі результати з ключем `A` з усіх map нод мають йти на одну й ту саму reduce ноду. Саме це розподілення і має виконуватись `Shuffle` кроком.

- Reduce - нода, яка приймає купу результатів на один ключ і „редюсить„ їх до одного результату.

Ці кроки можна описати такими сигнатурами:

```
1 map :: [a] -> (a -> (key, [value])) -> [(key, [
    value])]
2 shuffle :: [(key, [value])] -> [(key, reduceNodeId,
    [value])]
3 reduce :: [[(key, [value])]] -> [(key, [endvalue])]
```

Map і Shuffle мають бути на одній ноді, Shuffle має знати адреси усіх Reduce нод, так і те, які саме ключі вони приймають. Хоча для ще більшої паралелізації Reduce крок краще виконувати на окремих від Map

нодах, у нашій імплементації на кожній Map ноді буде виконуватись і Reduce крок, націлений на певний діапазон ключей. Причиною такого рішення є недостатня кількість комп'ютерів у автора статті.

Існує багато реалізацій підходу MapReduce. До найпопулярніших можна віднести Apache Hadoop та Apache Spark (Apache Spark має значно більше можливостей розподілених обчислень ніж MapReduce, але MapReduce реалізується на ньому в кілька стрічок). Але в рамках цієї роботи буде написана власна реалізація підходу, яка використовуватиме акторну модель.

3.2 MapReduce - базовий опис імплементації

Так як ми використовуємо акторну модель, то в нас буде три види акторів:

- MasterActor - актор, що регулюватиме роботу усього кластеру. Буде одним єдиним свого типу на весь кластер.
- MapActor - актор що виконуватиме Map та Shuffle функціонал. Їх буде стільки, скільки комп'ютерів у кластері.
- ReduceActor - актор, що виконуватиме Reduce крок. Аналогічно до MapActor, їх буде стільки ж скільки комп'ютерів у кластері (хоча в теорії можна створити скільки завгодно, головне реалізувати Shuffle крок так, щоб він правильно розподіляв данні по Reduce акторам)

3.3 MapReduce - MasterActor

MasterActor - єдиний актор, адресу якого має мати кожна інша нода. Кожен MapActor на початку своєї роботи має відправити майстер ноді

повідомлення виду

```
1     public record InitMapRequest(int Percentage);
```

Де Percentage - відсоток навантаження від усього кластеру, який на себе візьме ця нода. Аналогічно Reduce ноди відправляють повідомлення

```
1     public record InitReduceRequest(int Percentage)
        ;
```

Відповідно, у майстер акторі є такі хендлери:

```
1  Receive<InitMapRequest>(map =>
2  {
3      _mappers.Add((Sender, map.Percentage));
4      Console.WriteLine("Connected map node {Sender.
         Path.ToSerializationFormat()}");
5      _mapperCounter.Increment();
6  });
7  Receive<InitReduceRequest>(reduce =>
8  {
9      _reducers.Add((Sender, reduce.Percentage));
10     Console.WriteLine("Connected reduce node {
         Sender.Path.ToSerializationFormat()}");
11     _reduceCounter.Increment();
12 });
```

Цей код додає у внутрішній стан майстер актору посилання на усі мап та редюс актори в кластері. Коли усі актори відправлять повідомлення InitMapRequest/InitReduceRequest, майстер актор запускає код, який передає усім мап акторам посилання на усі Reduce актори. Повідомлення виглядає приблизно так:

```
1    public record ReducePath(IActorRef ActorPath,
       int Start, int End);
```

Де ActorPath - посилання на Reduce актор, int Start/End - діапазон значень ключів, за які відопідає цей Reduce актор.

У відповідь, кожен мап актор відправляє пусте повідомлення ReceivedReducePath, що сповіщає майстер актор що цей мап актор отримав інформацію про Reduce актор. Коли усі мап актори дізнаються про усі Reduce актори, майстер актор відправляє мап акторам сигнал-повідомлення, щоб вони почали роботу. Після цього майстер актор не потрібен, він виконав свою задачу - сповістив усі актори в системі про інші актори.

3.4 MapReduce - MapActor

Як було описано вище, цей актор хендлить повідомлення ReducePath, отримаючи шлях до Reduce акторів і зберігаючи їх у своєму стейті. Але основний функціонал і логіка виконується при обробці повідомлення Map. Ми не будемо реалізовувати узагальнене рішення будь якої MapReduce проблеми, а тому наведемо псевдокод того, що виконується у цьому хендлері:

```
1  int cpuCount = Environment.ProcessorCount * 2;
2  int offs = docPaths.Count / cpuCount;
3  int remains = docPaths.Count;
4  var tasks = new Task[cpuCount];
5  for (int i = 0; i < cpuCount; i++)
6  {
7      var i1 = i;
8      var maxVal = offs * i1 + offs;
```

```
9     tasks[i] = Task.Run(async () =>
10    {
11        for (int j = offs * i1; j < maxVal; j++)
12        {
13            Interlocked.Decrement(ref remains);
14            log.Info("Indexing {toParse[docPaths[j
                  ]]} - {docPaths[j]}, {remains}
                  remains");
15            if (remains <= 0)
16                break;
17            var file = await File.ReadAllTextAsync(
                  filesPath + toParse[docPaths[j]]);
18            ...
19            reducers[x].BlockingCollection.Add(
                  result for specific reducer);
20        }
21    });
22 }
23
24 foreach (var reducer in _reducers)
25    await reducer.StartConsuming();
26 await Task.WhenAll(tasks);
27 log.Info("All tasks are done");
28 foreach (var reducer in _reducers)
29    reducer.BlockingCollection.CompleteAdding();
30 server.Tell(new Done());
```

Тут відносно багато логіки. Перше що можна побачити, це паралелізація використовуючи вдвічі більше потоків ніж є на ноді. Так, це не

ідіоматичне рішення для акторної моделі обчислень, але нам акторна модель в першу чергу потрібна для розподілених обчислень, а не паралельних на одному ком'ютері. Немає сенсу ускладнювати код додаючи більше акторів, якщо можна запустити код на одному акторі в кілька потоків, ніяк при цьому не змінюючи архітектуру.

У читача може виникнути питання, де виконується відправка даних reduce акторам. Для цього потрібно спочатку зрозуміти що таке `BlockingCollection<T>`

BlockingCollection

BlockingCollection - колекція, яка необхідна для реалізації BackPressure. Вона дозволяє паралельно читати, видаляти і додавати елементи. Основні її риси:

- У конструктор як аргумент передається максимальна кількість елементів у колекції
- Метод `Add(T item)` якщо елементів у колекції забагато, замість `OverflowException`, блокує поточний поток, поки кількість елементів у колекції не зменшиться
- Цикл `foreach(var elem in collection.GetConsumingEnumerable())` буде видаляти при кожній ітерації елемент з колекції, а якщо вона пуста - блокувати поток, а не виходити з циклу.
- Щоб завершити foreach, треба викликати `collection.CompleteAdding()`

Ми будемо її використовувати, щоб одні потоки додавали результати Map кроку у колекцію, а інші - зчитували їх і передавали reduce акторам. І в кінці, коли MapActor завершить свою роботу, викликається `BlockingCollection.CompleteAdding()`, щоб завершити додавання елементів.

3.4.1 Передача даних reduce акторам

Проблема передачі даних

Ця частина імплементація найбільш критична з точки зору швидкодії, адже вимагає передачі величезної кількості даних через мережу. У автора виникло багато проблем з нею, далі буде описано які саме проблеми виникли і як їх прийшлось отпимізовувати.

Перша, найбільш тривіальна спроба - передавати дані одним великим повідомленням. Очевидно що вона працювала лише на дуже малих об'ємах даних і була зовсім непрактичо.

Друга спроба - використати `Akka.Streams`. У цій спробі автор намагався поточно передати елменти - пари типу `(key, value)`. Цей варіант спрацював і пам'яті вистачало, але виникла інша проблема - навантаження на мережу було 100%, тоді як процесор майже не навантажувався. Спочатку виникла підозра, що проблема не у алгоритмі, а у обладнанні - кластер працював по бездротовій мережі Wifi, а не напряму через кабель. Автор вирішив купити два кабелі Ethernet і це хоч і прискорило алгоритм, але процесор все ще простоював. Так з'явилось розуміння, що проблема не у обладнанні, а у коді.

Не важко помітити можливу оптимізацію у другій спробі: ми передаємо багато дублікатів ключів, тобто для `(keyA, [value1, value2, value3])`, будуть передані елементи `(keyA, value1)`, `(keyA, value2)`, `(keyA, value3)`. Це можна оптимізувати, передаючи одразу `(keyA, [value1, value2, value3]`. Але так виникає та ж проблема, що і є в першій спробі - якщо значень на один ключ буде надто багато, кидаються помики. Щоб це виправити, прийшлось написати власну реалізацію, через написання власного актору Flow, описаного у розділі Akka.Streams

Рішення через Flow актор

Так як основна проблема в тому, що різні елементи потоку мають різний розмір, нам потрібно зробити щоб усі елементи мали однаковий розмір. Тому елементами нашого потоку будуть масиви байт, кожен з яких міститиме 8192 байт. Для того, щоб перетворити різнорозміреві елементи в однакові 8 кілобайтні чанки, нам і знадобиться власний Flow.

Flow актор має хендлити два типи повідомлень: TIn, та TOut. Ось відповідно код кожного з цих хендлерів:

TOut:

```
1     SetHandler(chunker.Out, onPull: () =>
2         {
3             if (IsClosed(chunker.In))
4             {
5                 if (!_buffer.IsEmpty)
6                     PushSlice();
7                 else CompleteStage();
8             }
9             else
10            {
11                if (_buffer.Count < _chunker.
                      _chunkSize)
12                    Pull(chunker.In);
13                else PushSlice();
14            }
15        });
```

`_chunkSize` - розмір чанку, у нашому випадку 8192 байт.

TIn:

```
1 SetHandler(chunker.In, onPush: () =>
```

```
2         {
3             var element = Grab(chunker.In);
4             _buffer += element;
5             PushSlice();
6         }, onUpstreamFinish: () =>
7         {
8             if (_buffer.IsEmpty)
9                 CompleteStage();
10        });
```

`PushSlice()` - метод що відправляє у Sink поточний буфер, і очищує стан поточного буферу у Flow акторі.

`onUpstreamFinish` - метод, що викликається якщо у Source закінчились дані. У ньому викликається `PushSlice`, щоб примусово відправити останні дані до Sink.

Фінальний код, що перетворює BlockingCollection у оптимізований потік:

```
1 var sourceRef = await Source.FromEnumerator(
    BlockingCollection.GetConsumingEnumerable().
    GetEnumerator)
2     .Select(t => t.Concat(ByteString.FromString("\n
        ")))
3     .Via(new Chunker(8192))
4     .Select(t => t.ToArray())
5     .RunWith(StreamRefs.SourceRef<byte[]>(),
6         Materializer);
7 Ref.Tell(new Reduce(sourceRef));
```

Висновок

У результаті вийшло значно пришвидшити передачу даних до reduce акторів. В результаті навантаження на процесор виросло з 10% до 60%, а сам алгоритм значно пришвидився.

3.4.2 Reduce актор

У reduce акторі доволі простий код - ми просто зчитуємо дані з мап акторів, десеріалізуємо їх і записуємо у конкурентне key-value сховище:

```
1  await reduce.CompressedPairs.Source.Select(t =>
     ByteString.FromBytes(t))
2      .Via(Framing.Delimiter(ByteString.FromString("\
         n"), int.MaxValue, true)).Async()
3      .Select(t => DeCompress(t.ToArray()))
4      .RunForeach(pair =>
5      {
6          _log.Info("REDUCING {pair.key}, {remains}
             remains ...");
7          Interlocked.Decrement(ref remains);
8          foreach (var val in pair.values)
9          {
10             _dict.AddOrUpdate(val, (_, key) => new
                 () {key},
11                 (_, set, key) =>
12                 {
13                     Monitor.Enter(set);
14                     set.Add(key);
15                     Monitor.Exit(set);
16                     return set;
```

```
17                  }, pair.key);
18          }
19      }, materializer);
20  _log.Info("DONE Reducing for {address}...");
```

Розділ 4

PageRank

4.1 Загальний опис

PageRank - алгоритм, що визначає популярність веб сторінок, підраховуючи, скільки інших сторінок посилається на дану сторінку.

Його входом є список пар ключ-значення, де ключ - це певна сторінка, а значення - список посилань з цієї сторінки на інші, а також ймовірність що саме на сторінку з цим ключем перейде користувач.

Виходом ж є число від 0 до 1 до кожної сторінки - ймовірність що користувач перейде саме на цей сайт.

Алгоритм є ітеративним, виконується в кілька ітерацій до точності яка нам необхідна. На початку ймовірність для кожної сторінки задається як $\frac{1}{X}$, де `X` - кількість сторінок що обчислюються алгоритмом. При наступних ітераціях в якості ймовірностей використовується результат попередніх ітерацій.

Нова ймовірність для певного значення `a` обчислюється за наступною формулою:

$$\Pr(a) = \sum_{v \in B_v} \frac{\Pr(v)}{Ł(v)} \tag{4.1}$$

Де

- B_v - множина сторінок, що мають посилання на сторінку v
- Ł(v) - кількість посилань, що містить у собі сторінка v

Damping factor

Так як користувач клікатиме по вебсторінкам скінченну кількість разів, позначимо ймовірність, що користувач продовжить клікати, літерою d. Це значення часто називають damping factor. Найчастіше йому ставлять значення у 0.85. Він змінює формулу приблизно так:

$$\Pr(a) = \frac{1-d}{N} \sum_{v \in B_v} \frac{\Pr(v)}{Ł(v)} \tag{4.2}$$

Де N - кількість сторінок, що обчислює алгоритм.

У оригінальному папері по PageRank алгоритму натомість була така формула:

$$\Pr(a) = (1-d) + d \sum_{v \in B_v} \frac{\Pr(v)}{Ł(v)} \tag{4.3}$$

4.1.1 PageRank - простий приклад

Розглянемо як працює алгоритм на простому прикладі з 4 сторінок. Нахай в нас є сторінки A, B, C та D. І нехай вони ось так посилаються одна на одну:

- B - C, A
- C - A
- D - A, B, C

Тоді

$$\Pr(A) = \frac{Pr(B)}{2} + \frac{Pr(C)}{1} + \frac{Pr(D)}{3} = \frac{0.25}{2} + \frac{0.25}{1} + \frac{0.25}{3} = 0.125 + 0.25 + 0.083 = 0.458$$

Аналогічно робимо розрахунки і для інших сторінок. При першій ітерації використовуємо значення 0.25 для кожної сторінки, при наступних - ті, що вийшли у попередній ітерації.

4.2 PageRank - у MapReduce

Алгоритм ідеально підходить під MapReduce модель. Так, на Map кроці для кожного ключа ми збираємо усі доданки, за які відовідає ця Map нода, сумуємо їх і передаємо пару ключ-сума певному Reduce актору, що відповідає за цей ключ. Reduce актор у свою чергу просто сумує по ключу усі суми, що прийшли з Map акторів. Цей алгоритм виходить навіть простішим за реалізацію MapReduce, описану у розділі 3, адже у якості значення у нас виступає число, а не безліч значень, а тому ніякі оптимізації у вигляді розбивання потоку на чанки не потрібні, навантаження на мережу значно менші, як і вимоги до пам'яті.

В результаті ReduceActor виглядає так:

```
1 Receive<Reduce>(async reduce =>
2 {
3     var materializer = Context.System.Materializer
        ();
4     var address = Sender.Path.Address;
5     _log.Info("Reducing from {address}...");
6     await reduce.CompressedPairs.Source.Select(t =>
```

```
        ByteString.FromBytes(t))
7         .Via(Framing.Delimiter(ByteString.
            FromString("\n"), int.MaxValue, true)).
            Async()
8         .Select(t => DeCompress(t.ToArray()))
9         .RunForeach(pair =>
10        {
11            _log.Info("REDUCING {pair.docId}, {
                remains} remains ...");
12            Interlocked.Decrement(ref remains);
13                _dict.AddOrUpdate(pair.docId, (_,
                    docId) => pair.sum,
14                    (_, set, docId) => set + pair.
                        sum, pair.docId);
15        }, materializer);
16    _log.Info("DONE Reducing for {address}...");
17    Collect(id);
18 });
```

A MapActor так:

```
1  int cpuCount = Environment.ProcessorCount * 2;
2  int offs = docPaths.Count / cpuCount;
3  int remains = docPaths.Count;
4  var tasks = new Task[cpuCount];
5  for (int i = 0; i < cpuCount; i++)
6  {
7      var i1 = i;
8      var maxVal = offs * i1 + offs;
9      tasks[i] = Task.Run(async () =>
```

```
10      {
11          for (int j = offs * i1; j < maxVal; j++)
12          {
13              Interlocked.Decrement(ref remains);
14              log.Info("Indexing {toParse[docPaths[j
                    ]]} - {docPaths[j]}, {remains}
                    remains");
15              if (remains <= 0)
16                  break;
17              var file = await File.ReadAllTextAsync(
                    filesPath + toParse[docPaths[j]]);
18              var key = docPaths[j];
19              var values = file.SelectProbabilities()
                    ;
20              var reducer = GetReducerForKey(key);
21              reducer.BlockingCollection.Add((key,
                    values.Sum()));
22          }
23      });
24  }
25
26  foreach (var reducer in _reducers)
27      await reducer.StartConsuming();
28  await Task.WhenAll(tasks);
29  log.Info("All tasks are done");
30  foreach (var reducer in _reducers)
31      reducer.BlockingCollection.CompleteAdding();
32  server.Tell(new Done());
```

Нажаль, автор роботи не встиг реалізувати підтримку багатьох ітерацій. Алгоритм було протестовано на кластері з двох комп'ютерів, під'єднаних до локальної мережі через кабелі Ethernet. Один комп'ютер був потужнішим за інший, але завершили роботу вони майже одночасно. Через обмеження в часі у статті не буде надано детальних результатів запуску кластеру.

Розділ 5

Висновок

Ми детально розглянули імплементацію MapReduce підходу, проблеми які виникли під час реалізації, а також те, як можна оптимізувати мережовий код. Також розглянули алгоритм PageRank, хоча він і був скоріше прикладом того, як можна використати MapReduce підхід, ніж основною темою статті.

Код, використаний у роботі, буде наданий окремим архівом.

Бібліографія

[1] MapReduce *Simplified Data Processing on Large Clusters?* (https://web.archive.org/web/20171202123313/https://research.google.com/archive/mapreduce.html)

[2] PageRank *Bringing Order to the Web* (https://web.archive.org/web/20020506051802/http://www-diglib.stanford.edu/cgi-bin/WP/get/SIDL-WP-1997-0072?1)

[3] Akka.Net *Documentation* (https://getakka.net/articles/intro/what-is-akka.html)