Autorzy:
Jakub Sejdak
Maciej Dobrowolski

Data:
04.06.2012

Projekt ostateczny – infrastruktura map-reduce

Treść zadania

Przedmiotem zadania jest opracowanie systemu komunikacji między-procesowej dla zadań typu map-reduce. Typowo obliczenia w tym modelu odbywają się w systemie rozproszonym z komunikacją poprzez pliki (DFS – rozproszony system plików), na potrzeby tego projektu przyjmiemy jednak „obliczenia" zlecane procesom wykonującym się na jednej maszynie (wykorzystujemy potencjalną wielo-procesorowość maszyny, przy czym zakładamy, że system operacyjny optymalnie wykorzysta procesory, tj. nie zajmujemy się przydziałem zadań do fizycznych procesorów).

System powinien realizować następujące funkcje:

- Tworzenie i zarządzanie pulą procesów obliczeniowych (i procesem nadzorcy)
- Przydzielanie procesom roboczym zadań do fazy „map", tj. przekazanie danych, zlecenie zadania dla wskazanej porcji danych, przygotowanie do odbioru danych wynikowych z fazy „map"
- Przydzielanie procesom roboczym zadań do fazy „reduce", tj. przekazanie danych, zlecenie zadania dla wskazanej porcji danych, przygotowanie do odbioru danych wynikowych z fazy „reduce"

Przedmiotem projektu jest przede wszystkim stworzenie infrastruktury komunikacyjnej opisanej wyżej, należy też przygotować prosty program testujący, nie musi on realizować algorytmu map-reduce, ale powinien funkcjonować w zakresie komunikacji, przekazywania danych i synchronizacji podobnie do programów map-reduce.

Zrealizować komunikację przy pomocy: potoków nienazwanych (pipe)

Dodatkowe założenia i wymagania:

- W ramach testów należy zadbać o to, aby komunikacja i synchronizacja procesów była w czytelny sposób widoczna (odpowiednie logi, itp.)

Interpretacja treści zadania

Powinniśmy zbudować infrastrukturę, która będzie frameworkiem dla programistów chcących skorzystać z dobrodziejstwa algorytmu map-reduce. Infrastruktura powinna korzystać z komunikacji między-procesowej – potoków nienazwanych, tak aby było to niewidoczne (przezroczyste) dla programistów korzystających z niej.

W skład systemu wchodzą dwa typy procesów – wykonujący zlecone mu zadania „worker" i zarządzający wszystkimi procesami programu – „master". Proces zarządzający, oprócz tworzenia procesów roboczych, powinien także organizować im pracę, czyli przekazać dane, zlecić pracę do wykonania i odebrać jej wyniki.

Ponadto powinniśmy umieścić w programie funkcje logujące pracę, jeśli program zostanie wywołany z odpowiednim przełącznikiem.

Opis funkcjonalny (black-box)

System realizuje algorytm MapReduce pozwalając przy tym użytkownikowi na zdefiniowanie własnych funkcji:

- Odczytywania danych (dla różnych źródeł danych takich jak baza danych, plik tekstowy, itp.)
- Map
- Reduce

Po ustaleniu tych funkcji/metod użytkownik może uruchomić mechanizm MapReduce.

Personalizacja działania systemu jest rozbudowana o ustalenie ilości procesów, które wykonują zadania map i reduce (oddzielnie).

Dodatkowo użytkownik może ustalić logowanie, zarówno do plików jak i na konsolę, a także pozostawić pliki tymczasowe będące produktem działania procesów *map* – na przykład w celu ich późniejszej analizy.

W celach dostosowania zaimplementowanego przez nas mechanizmu do przetwarzanych danych użytkownik może ustalić specjalny ciąg znaków będący ogranicznikiem przy wysyłaniu ciągów danych, które nie mają z góry znanej długości.

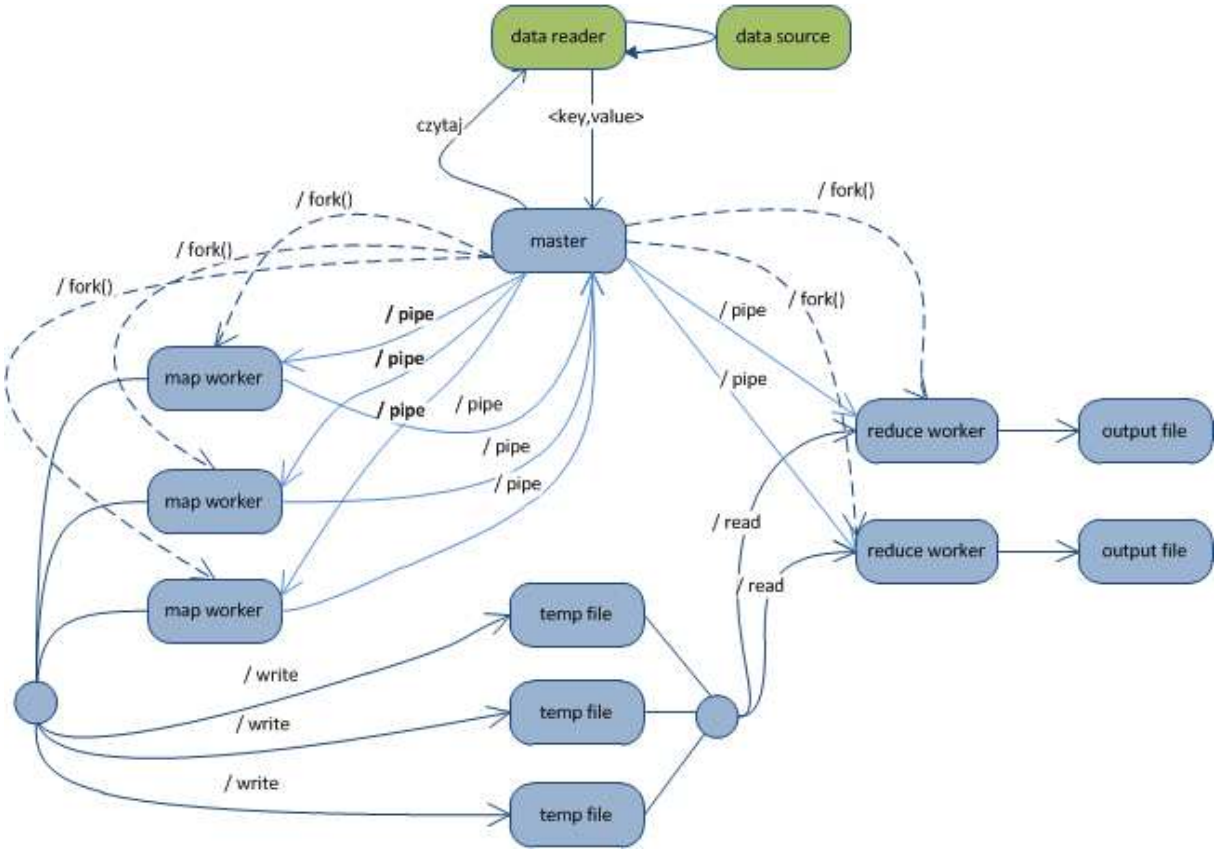
Każdy proces *reduce* wyprodukuje jeden plik wyjściowy (zgodnie z algorytmem MapReduce) o nazwie *ReduceOutput.x*, gdzie x jest kolejnym numerem procesu.

Podział na moduły i strukturę komunikacji między nimi

Nasz framework, oprócz **modułu inicjującego**, który powinien zostać dostarczony przez użytkownika składa się z kilku klas o ściśle określonych funkcjach:

- **MapReduce** – klasa zawierająca logikę działania algorytmu MapReduce, tworzy procesy i zarządza nimi, organizuje także komunikację poprzez potoki nienazwane (pipe).
- **Logger** – klasa wydzielająca moduł logowania (zbierania informacji), realizuje wszystkie funkcjonalności związane z wyświetlaniem komunikatów na konsolę, bądź też zapisywaniem ich do plików.
- **ProcessInfo** – klasa przechowująca informacje o procesach, zapamiętuje najważniejsze dane takie jak numer procesu (PID), numer workera oraz uchwyty do komunikacji poprzez potoki nienazwane.
- **AbstractMapWorker** – abstrakcyjna klasa służąca do ułatwienia użytkownikowi zaimplementowania metody mapującej.
- **AbstractReduceWorker** – abstrakcyjna klasa służąca do ułatwienia użytkownikowi zaimplementowania metody redukującej.

Ostatnim elementem nie-klasą jest funkcja odczytująca dane **DataReader** – służy do czytania danych ze zdefiniowanego przez użytkownika źródła danych (może być to np. baza danych, plik tekstowy).



1. Proces *master* tworzy ustaloną liczbę **2xM+R** par sprzężonych deskryptorów plików – wskazujących na i-node potoku oraz **M** procesów, które będą pracowały w roli *map worker'a* wykonując funkcję *map,* a także **R** *reduce workerów* wykonujących funkcję *reduce*.
2. Proces master odpytuje funkcję DataReader o kolejne dane z plików wejściowych i po otrzymaniu ich przesyła parę <klucz, wartość> potokiem nienazwanym do procesu map. Odpytywanie jest powtarzane do momentu, kiedy funkcja DataReader zwróci puste wartości, co jest znakiem zakończenia czytania danych.
3. Procesy potomne – *map worker* wykonują funkcję *map* na otrzymanych parach <klucz, wartość>, wynik każdego wywołania funkcji jest buforowany, po zakończeniu pracy – zapisywany na dysku do unikalnego pliku tymczasowego. Procesy *map worker* wysyłają nazwę pliku tymczasowego do procesu *master* poprzez potok nienazwany.
4. Proces *master* odbiera nazwy plików tymczasowych, przegląda je szukając kluczy zwróconych przez funkcję map. Generuje posortowaną listę unikatowych kluczy, która posłuży do przydziału pracy dla procesów *reduce worker*.
5. Klucze są przydzielane do procesów *reduce worker*, każdy proces dostaje potokiem nienazwanym listę kluczy, którymi powinien się zająć oraz listę plików tymczasowych zwróconych przez procesy *map worker*.

6. Procesy *reduce worker* po otrzymaniu niezbędnych danych (lista kluczy oraz pliki tymczasowe) przystępują do zebrania danych z plików, posortowania ich oraz grupowania po wspólnym kluczu.
7. Produktem poprzedniego kroku będzie lista <klucz, lista(wartość0, wartość1, …)>, której każdy element posłuży jako argument wejściowy dla funkcji *reduce*.
8. Wywoływana jest funkcja *reduce*, której produktem jest lista wartości (w większości zastosowań będzie to lista jednoelementowa).
9. Każdy wynik wywołania *reduce* dla unikatowego klucza jest zapisywany do pliku o nazwie *ReduceOutput.x*, gdzie x jest kolejnym numerem procesu.
10. System kończy pracę.

Jak widać z powyższego rysunku i opisu – komunikację potokami nienazwanymi zastosowaliśmy w trzech sytuacjach:

- do przekazywania danych procesom *map worker*
- do odbierania danych od procesów *map worker*
- do wysyłania danych (lista kluczy i nazw plików tymczasowych) do procesów *reduce worker*

Opis najważniejszych rozwiązań funkcjonalnych wraz z uzasadnieniem (opis protokołów, kluczowych funkcji).

Ważne decyzje projektowe

Kluczową decyzją było ograniczenie funkcjonalności programu do przyjmowania danych tylko typu <string klucz, string wartość>. Decyzja była pokierowana głównie czasochłonnością i złożonością rozwiązania problemu dla dowolnej klasy - należało by wtedy zmusić użytkownika do zaimplementowania dodatkowych funkcji, służących do porównywania obiektów zadanej klasy, oraz ich serializacji. Należałoby się wtedy skupić na implementacji szablonowego rozwiązania, zamiast na działaniu komunikacji poprzez potoki nienazwane.

Użyte protokoły

Jedynym używanym przez nas protokołem jest zapis ciągów znaków, zapisujemy je w następującej konwencji:

SIZE_OF_STRING	STRING
[1 x size_t]	[SIZE_OF_STRING x char]

Kluczowe funkcje

void MapReduce::spawnWorkers():

Funkcja wysokiego poziomu do tworzenia workerów typu map oraz reduce. Dla każdego tworzonego procesu alokuje ona instancję klasy ProcessInfo przypisując jej numer workera oraz przekazuje ją do niskopoziomowej funkcji tworzącej procesy.

void MapReduce::run():

Jest to główna funkcja udostępniona użytkownikowi. Uruchamia ona cały framework i wykonuje następujący algorytm:

- wywołuje funkcje tworzące procesy potomne
- rozdziela pliki wejściowe na dane korzystając z dostarczonej przez użytkownika funkcji dataReaderFunc() i wysyła te dane procesom mapującym
- czeka, aż wszystkie MapWorkery zakończą działanie
- odbiera nazwy plików tymczasowych po fazie map
- odczytuje i sortuje unikalne nazwy kluczy dla fazy reduce
- przydziela klucze i wysyła nazwy plików tymczasowych do ReduceWorkerów
- czeka na zakończenie fazy reduce
- kasuje pliki tymczasowe (jeżeli taka opcja była ustawiona)

void MapReduce::writeTmpFile(FILE* tmpFile, pair<string, string> row):

Zapisuje wiersz danych do pliku tymczasowego zgodnie z formatem: sizeof(row.first), row.first, sizeof(row.second), row.second.

bool MapReduce::readTmpFile(FILE* tmpFile, string &k, string &v):

Odczytuje dane z pliku tymczasowego zgodnie z formatem przedstawionym w funkcji writeTmpFile().

void MapReduce::writeStringToPipe(int pipeDesc, string _str):

Zapisuje danych w postaci stringa do potoku nienazwanego zgodnie z formatem: sizeof(_str), str.

bool MapReduce::readStringFromPipe(int pipeDesc, string &_str):

Odczytuje string z potoku nienazwanego zgodnie z formatem przedstawionym w funkcji writeStringToPipe().

void MapReduce::runMap():

Implementuje „pętlę główną” procesów typu map wg algorytmu:

- odczytuje z potoku sekwencję danych przydzielonych danemu procesowi do momentu napotkania ustalonego wzorca znakowego (delimitera)
- wywołuje na odczytanych danych funkcję map()
- scala ze sobą wszystkie odczytane dane
- tworzy pliki tymczasowe wykorzystując standardową funkcję mkstemp()

- wysyła poprzez pipe do procesu nadzorcy nazwy utworzonych plików tymczasowych
- zapisuje wyniki fazy map do plików tymczasowych

void MapReduce::runReduce():

Implementuje „pętlę główną” procesów typu reduce wg algorytmu:

- odbiera przyznane procesowi klucze z potoku nienazwanego
- odbiera z potoku nazwy wszystkich plików tymczasowych powstałych w fazie mapowania
- tworzy pliki wyjściowe związane z danym procesem redukującym o nazwie ReduceOutput.<NUMER_REDUCE_WORKERA>
- szuka i odczytuje w otrzymanych plikach tymczasowych klucze i związane z nimi wartości które zostały przyznane danemu procesowi
- grupuje je i wywołuje funkcję reduce()
- zapisuje wyniki do plików wyjściowych

```
bool MapReduce::spawnMapWorker(ProcessInfo *info):
```

Niskopoziomowa funkcja do tworzenia pojedynczego MapWorkera. Tworzy dwie pary sprzężonych deskryptorów do komunikacji przez potok oraz zapisuje informacje o nich do obiektu typu ProcessInfo przekazanego od funkcji wysokiego poziomu. Następnie wywoływana jest funkcja fork. Funkcja odpowiada również za zamknięcie nieużywanych deskryptorów w części rodzica i potomka. Na koniec w części potomka wywoływana jest funkcja runMap(), a w części rodzica zapisywana jest informacja o powstałym potomku.

```
bool MapReduce::spawnReduceWorker(ProcessInfo *info):
```

Funkcja ta jest analogiczna do funkcji spawnreduceWorker z wyjątkiem wywołania funkcji runReduce() zamiast runMap().

```
void MapReduce::terminateWorkers():
```

Jest to awaryjna funkcja do kończenia pracy wszystkich procesów potomnych w przypadku błędu programu. Wysyła ona do wszystkich procesów potomnych sygnał SIGTERM.

Szczegółowy opis interfejsu użytkownika

System pozwala użytkownikowi na:

- Ustalenie funkcji czytania danych – tak, aby dane można było pobierać z dowolnego źródła (baza danych, plik tekstowy).

Ustalenie funkcji odbywa się poprzez wywołanie metody:
```
void MapReduce::setDataReader(pair<string, string> (*dataReaderFunc)()
```
która przyjmuje jako argument wskaźnik na funkcję zdefiniowaną przez użytkownika.

Funkcja powinna spełniać dodatkowy warunek (oprócz oczywistych wynikających z deklaracji):

- Będzie wywoływana wielokrotnie i powinna przy każdym wywołaniu zwracać kolejną porcję danych, która zostanie przekazana metodzie:
  `vector<pair<string,string> > MapWorker::map(string key, string value)`
  jako argument wywołania.

- Ustalenie funkcji *map.*

  Aby ustalić funkcję należy utworzyć klasę, która dziedziczy po AbstractMapWorker i implementuje metodę map:
  `vector<pair<string,string> > AbstractMapWorker::map(string key, string value)`

  Przekazanie metody do frameworka odbywa się poprzez wywołanie metody:
  `void MapReduce::setMap(AbstractMapWorker* mapWorker)`
  podając jako argument wskazanie na obiekt klasy dziedziczącej po AbstractMapWorker.

- Ustalenie funkcji *reduce.*

  Aby ustalić funkcję należy utworzyć klasę, która dziedziczy po AbstractReduceWorker i implementuje metodę reduce:
  `vector<string> AbstractReduceWorker::reduce(string value, list<string> values)`

  Przekazanie metody do frameworka odbywa się poprzez wywolanie metody:
  `void MapReduce::setReduce(AbstractReduceWorker* reduceWorker)`
  podając jako argument wskazanie na obiekt klasy dziedziczącej po AbstractReduceWorker.

- Ustalenie puli procesów roboczych – oddzielnie dla fazy *map* i *reduce*.

  Ustalenie liczby procesów, które będą obsługiwać dany problem odbywa się podczas tworzenia obiektu typu MapReduce, parametry podaje się jako argumenty konstruktora:
  `MapReduce framework(mapWorkers, reduceWorkers);`
  gdzie: *mapWorkers* to liczba procesów obsługujących zadanie map, a *reduceWorkers* to liczba procesów obsługujących zadanie reduce. Obydwie liczby powinny być większe od 0.

- Włączenie logowania oddzielnie dla logowania na ekran i do pliku.

  Logowanie na konsolę można ustalić poprzez metodę:
  `void MapReduce::consoleLogging(bool active);`
  dla argumentu **true** logowanie na ekran jest właczone, dla **false** – wyłączone.

  Podobna logika obowiązuje dla logowania do pliku, funkcjonalność ustalamy poprzez wywołanie metody:

```
void MapReduce::fileLogging(bool active);
```
Logi programu dostępne są w pliku: *mapreduce_log.txt*

- Włączenie/wyłączenie usuwania plików tymczasowych (produkt procesów *map worker*)

  Włączanie, lub też wyłączanie usuwania plików tymczasowych realizujemy poprzez wywołanie metody:
  ```
  void MapReduce::setRemoveTempFiles(bool active);
  ```
  Argument **true** (domyślnie ustawiony) dla usuwania plików, oraz **false** dla pozostawienia ich w folderze tmp/ - w celu późniejszej analizy działania algorytmu.

Postać plików konfiguracyjnych oraz logów

W naszym frameworku nie zastosowaliśmy żadnych plików konfiguracyjnych, ponieważ programista używając frameworka musi napisać swój kod i skompilować go wraz z wywołaniem naszego interfejsu, dlatego wszelka konfiguracja jest dostępna w postaci metod klasy MapReduce, które są opisane w opisie interfejsu użytkownika (punkt wyżej).

Logi są zapisywane do pliku *mapreduce_log.txt* znajdującego się w katalogu z programem wykonywalnym, są one w postaci:

- Dla logów dotyczących konkretnego procesu:
  [PID xxxx] yyyy
- Dla logów ogólnych:
  yyyy

Gdzie xxxx – numer PID, a yyyy – treść wiadomości.

Dodatkowo udostępniliśmy funkcjonalność nieusuwania plików tymczasowych będących produktem działania procesów *map worker*. Pliki te zawierają sumaryczne wyniki działania funkcji *map* w obrębie danego procesu, mają postać binarną zapisaną w następujący sposób:

SIZE_OF_KEY	KEY	SIZE_OF_VAL	VALUE
[1 x size_t]	[SIZE_OF_KEY x char]	[1 x size_t]	[SIZE_OF_VAL x char]

Opis wykorzystanych narzędzi

- Środowisko uruchomieniowe: Gentoo/Ubuntu (różne wersje)
- Środowisko programistyczne: Eclipse 3.7.0
- Kompilator: gcc 4.6.1
- Debuger: gdb 7.3-2011-08 (przez Eclipse)
- Repozytorium: Git (googlecode)
- Testowanie aplikacji: testowanie powierzaliśmy sobie nawzajem, tak by nigdy nie testować kodu napisanego przez siebie.

Opis testów i wyników testowania

Testy, jak wspomnieliśmy wyżej – były wykonywane przez nas samych, tak aby nie testować kodu, który sami pisaliśmy.

Testy wydajnościowe

Dla przykładowego pliku tekstowego zawierającego treść sztuki „Shakespeare in Love”, ważącego 1000kB uruchomiliśmy program liczący wystąpienia wyrazów z różnymi parametrami (ilość procesów), oto wyniki czasowe jakie uzyskaliśmy:

Map(1)/Reduce(1)	Map(5)/Reduce(1)	Map(1)/Reduce(5)	Map(5)/Reduce(5)
42.16s	7.17s	41.01s	7.888s

Czynnikiem, którego nie wzięliśmy pod uwagę była różnorodność wyrazów (czym więcej unikatowych wyrazów, tym efektywniejsza wielo-procesowość fazy *reduce*), oraz rozłożenie wyrazów (mogło być tak, że proces *reduce* dostał do analizy tyle samo słów co inny, ale słowa te występowały znacznie częściej).

Wyniki potwierdziły nasze oczekiwania, dzielenie fazy *reduce* na kilka procesów, nie zawsze przynosi pozytywne efekty (w naszej implementacji), ponieważ oznacza to często konieczność przejrzenia plików tymczasowych więcej razy. Jednak zwiększenie liczby procesów *map* przynosi zauważalne skutki, wynikające z wieloprocesorowości maszyny na której testowaliśmy nasz framework, proces master zajmuje się czytaniem i przekazywaniem danych, a w międzyczasie procesy mapujące wykonują już pracę.

Testy poprawności działania

Po wielu testach doszliśmy do wniosku, że warto wprowadzić możliwość zmiany przez użytkownika znaku ograniczającego ciąg danych (delimiter) w potoku nienazwanym, czasami funkcja mapująca może wydzielić klucz, który jest taki sam jak nasz ogranicznik, dlatego przy implementacji programu warto zastanowić się, jak powinien ten ogranicznik wyglądać.

Program dobrze radzi sobie w plikami ważącymi nawet 100MB, dlatego zaniechaliśmy dalszych testów obciążeniowych.

Framework zabezpieczony jest przed podaniem błędnych danych, bądź też niepodaniem ich w ogóle. Zalogowany zostanie błąd, jeśli:

- Dane w potoku ulegną zniekształceniu (ich wielkość).
- Dane w pliku tymczasowym zostaną zmienione (struktura).
- Użytkownik poda niepoprawne ilości procesów.
- Użytkownik nie poda funkcji czytającej dane, lub obiektów klas dziedziczących po *AbstractReduceWorker* i *AbstractMapWorker.*
- Nastąpi inny błąd związany z niemożliwością otworzenia pliku, bądź utworzenia procesu.