Zeszyty Naukowe WWSI, No 15, Vol. 10, 2016, s. 79-121

Porównanie wydajności i produktywności algorytmu tworzenia drzew decyzyjnych zaimplementowanego w środowiskach SPARK oraz GASPI

Roman Wyrzykowski, Tomasz Karoń*

Politechnika Częstochowska,
Instytut Informatyki Teoretycznej i Stosowanej

Abstrakt

W pracy zbadano wydajność i produktywność programistyczną wykorzystania chmur obliczeniowych oraz dwu odmiennych środowisk programistycznych, a mianowicie SPARK i GASPI, do równoległej implementacji algorytmów eksplorujących duże zbiory danych na przykładzie algorytmu ID3 tworzenia drzew decyzyjnych. Implementacje uruchomiono na platformie Google Compute Engine.

Słowa kluczowe **–** chmury obliczeniowe, tworzenie drzew decyzyjnych, algorytm ID3, środowiska GASPI i SPARK, porównanie, Google Compute Engine

1. Wprowadzenie

Chmury obliczeniowe są jednym z popularnych modeli przetwarzania danych. Wykorzystywana jest w nich idea dostarczania funkcjonalności rozumianej jako połączenie oprogramowania oraz koniecznej infrastruktury. W ten sposób użytkownik nie będąc właścicielem sprzętu ani oprogramowania ma możliwość korzystania z tych zasobów. Warto zauważyć, że sama chmura obliczeniowa jest usługą dostępną

* E-mail: tkaron@icis.pcz.pl

Zgłoszono do druku 24 października 2016 r.

„na żądanie”. W efekcie daje to możliwość szybkiej zmiany charakterystyki potrzebnych zasobów. W ten sposób przejawia się zasadnicza zaleta chmur obliczeniowych, czyli ich skalowalność i elastyczność. Rozróżnia się chmury prywatne, publiczne oraz łączące elementy dwu poprzednich, czyli hybrydowe. Ze znanych chmur publicznych wymienić można: Amazon Web Service, Google Cloud Platform, Microsoft Azure, Cloud OVH [1].

Istnieje wiele różnych modeli chmury obliczeniowej. W celu realizacji badań wykorzystano model IaaS czyli możliwość pracy na infrastrukturze informatycznej dostarczonej przez platformę chmury obliczeniowej [1].

Różnorodność dostępnych usług przetwarzania danych w chmurach to rozmaitość parametrów i rozwiązań. Oprócz parametrów technicznych należy także brać pod uwagę koszty wynajmowania mocy obliczeniowych. Ze względów ekonomicznych, celem realizacji badań, zdecydowano się skorzystać z możliwości pracy na infrastrukturze informatycznej dostarczonej przez platformę Google Compute Engine [2].

Na dostarczonej infrastrukturze można uruchomić maszyny wirtualne z różnorodnymi środowiskami programistycznymi. Jednym z ciekawszych jest środowisko Spark, oferujące duże możliwości wspomagania zrównoleglania obliczeń z jednoczesną odpornością na uszkodzenia i skalowalnością [3]. Innym środowiskiem jest interfejs programistyczny GASPI, korzystający z modelu PGAS [4]. Rozwiązanie to pozwala na wykorzystywanie mechanizmu RDMA, zwiększającego wydajność dostępu do danych. Podobnie jak środowisko Spark, GASPI jest oferuje odporność na uszkodzenia i dużą skalowalność [5].

Jednym z najważniejszych zastosowań chmur obliczeniowych jest eksploracja dużych zbiorów danych (ang. *Big Data*) [6]. Zbiory te, oprócz dużej objętości (ang. *Volume*) samych danych, charakteryzują się także dużą ich różnorodnością (ang. *Variety*) oraz zmiennością (ang. *Velocity*). Bardzo ważną cechą dużych zbiorów danych jest także ich wiarygodność (ang. *Veracity*). Warto zwrócić uwagę na angielskie nazwy cech dużych zbiorów danych, od których wzięła się nazwa modelu 4V opisującego te zbiory.

Zbiory charakteryzujące się wcześniej wymienionymi cechami trudno przetwarzać tradycyjnymi metodami, mówi się raczej o ich eksploracji lub pozyskiwaniu wiedzy (ang. data mining). Stosuje się w tym celu różnorodne metody szczegółowo opisane w pracach [7]–[9].

Jedną z podstawowych metod pozyskiwania wiedzy z dużych zbiorów danych jest tworzenie drzew decyzyjnych. Istnieje wiele różnych algorytmów tworzących

drzewa decyzyjne [10]–[12]. Do celów badania zdecydowano się wykorzystać algorytm ID3, kierując się głównie jego popularnością oraz prostotą [13]. Inne algorytmy pominięto ze względu na ich złożoność implementacyjną.

Celem niniejszego artykułu jest zbadanie zarówno wydajności, mierzonej czasem wykonania, jak i produktywności programistycznej wykorzystania chmur obliczeniowych oraz różnych środowisk programistycznych do równoległej implementacji algorytmów eksplorujących duże zbiory danych na przykładzie algorytmu ID3.

Należy tu zauważyć, że produktywność programistyczna w pracy [14] zdefiniowana została jako współczynnik obejmujący zarówno rezultat pracy programisty, czyli poprawnie działający program równoległy, jak i koszt stworzenia tego programu, który w sposób bardzo ogólny można przybliżyć liczbą linii jego kodu. Zakładając więc, że stworzony program jest poprawny, produktywność programistyczna jest odwrotnie proporcjonalna do liczby linii jego kodu. Wynika stąd zatem, że każde zmniejszenie liczby linii kodu, na przykład poprzez użycie zaawansowanych środowisk programistycznych, może przyczynić się do podniesienia produktywności programistycznej.

Trzeba podkreślić, że sama implementacja algorytmów przetwarzających duże zbiory danych, w szczególności w celu zbudowania drzew decyzyjnych, nie jest zagadnieniem nowym. Zostało ono opisane w pracach: [7]–[11], [13], [15]. Jednak prowadzone dotychczas badania obejmowały głównie problematykę wydajności zastosowanych algorytmów oraz podejść wykorzystanych do ich implementacji. Nie były natomiast badane łącznie zagadnienia produktywności programistycznej oraz wydajności w środowisku chmur obliczeniowych.

Algorytm ID3 może być zrównoleglony na kilka sposobów [15]. Na tym etapie badań zdecydowano się na realizację badania z wykorzystaniem najprostszego podejścia bazującego na synchronicznej konstrukcji drzewa decyzji. Założono jednocześnie, że wspomniane już zalety środowisk programistycznych oraz skalowalność chmury obliczeniowej pozwolą na zniwelowanie wad tego podejścia [10], [15]. W efekcie podczas badania samej implementacji, dokonano również analizy przydatności rozwiązania komercyjnego, jakim jest Google Compute Engine, w zakresie zwiększania produktywności programistycznej i wydajności równoległych implementacji algorytmów przetwarzających duże zbiory danych.

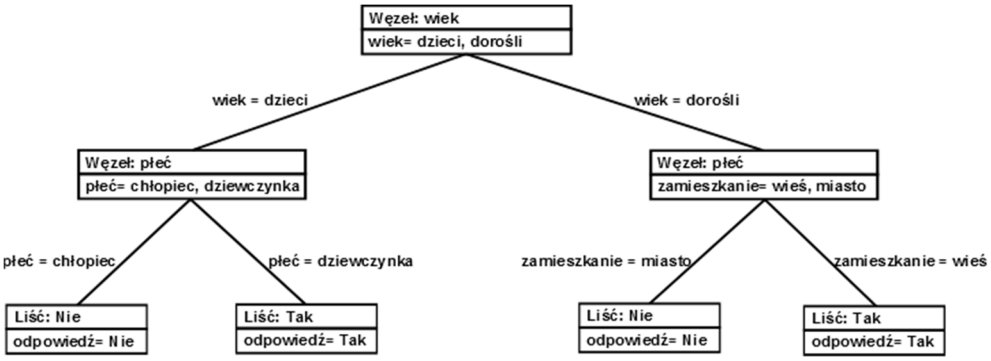
W sekcji 2 niniejszego artykuł opisano algorytm tworzenia drzew decyzyjnych ze zwróceniem uwagi na złożoność czasową tworzenia drzew decyzyjnych. Sekcja 3 i 4 zawiera przegląd możliwości oferowanych odpowiednio przez środowisko Spark i GASPI. Opis implementacji algorytmu ID3 przedstawiono w sekcjach 5 i 6. W sekcji 7 zaprezentowano szczegóły związane z środowiskiem uruchomieniowym. Dyskusję otrzymanych wyników przeprowadzono w sekcjach: 8, 9 i 10, przy czym

w sekcji 10 dokonano porównania wydajności obu implementacji. Wnioski i propozycję dalszych prac przedstawiono w sekcji 11.

2. Algorytm tworzenia drzew decyzyjnych ID3

Istnieje wiele metod eksploracji danych zawartych w dużych zbiorach danych. Jedną z nich jest klasyfikacja danych. Danymi wejściowymi w tej metodzie są krotki (ang. *tuple*) składające się z listy atrybutów opisowych oraz atrybutu decyzyjnego. W trakcie przeglądania danych treningowych, tworzony jest klasyfikator, który pozwala dokonać ustalenia atrybutu decyzyjnego dla nieznanego wcześniej przypadku na podstawie jego atrybutów opisowych. Metoda klasyfikacji danych jest więc procesem składającym się z dwu etapów: etapu budowania klasyfikatora na podstawie przykładowego, treningowego zbioru krotek, czyli powiązanych ze sobą atrybutów opisowych i decyzyjnego oraz etapu wykorzystywania zbudowanego modelu do ustalenia atrybutu decyzyjnego dla nowego, wcześniej nieznanego przypadku opisanego atrybutami opisowymi [8], [9], [7].

Jedną z technik klasyfikacji danych jest budowa klasyfikatora opisanego za pomocą drzewa decyzyjnego, czyli zbioru węzłów decyzyjnych połączonych gałęziami z liśćmi. Gałęzie biegną w dół od korzenia i kończą się na liściach, przy czym pomiędzy liściem a korzeniem może istnieć wiele węzłów decyzyjnych. Rozgałęzienia w drzewie tworzone są na podstawie wartości atrybutu decyzyjnego, tak więc w liściach zgrupowane są krotki o takiej samej wartości atrybutu decyzyjnego. Wartym zauważenia, jest fakt, że z punktu widzenia uczenia maszynowego, tworzenie drzewa decyzyjnego jest uczeniem nadzorowanym, gdyż algorytmowi należy dostarczyć zbiór krotek treningowych [8], [9], [7].



Rysunek 1. Przykładowe drzewo decyzyjne

Przykładowe drzewo decyzyjne przedstawiono na rysunku 1. Atrybutem decyzyjnym w tym drzewie jest odpowiedź, która może przyjmować wartości Tak i Nie. Atrybuty opisowe to: wiek, płeć i zamieszkanie Węzłami decyzyjnymi są: wiek będący jednocześnie korzeniem, płeć oraz zamieszkanie.

Budowa drzewa decyzyjnego jest kończona w momencie, gdy nie można już utworzyć żadnego nowego węzła decyzyjnego, czyli gdy krotki danych wejściowych umieszczone w liściach są jednorodne pod względem atrybutu decyzyjnego. Zdarzają się jednak sytuacje, gdy mimo różnorodnej zawartości atrybutu decyzyjnego, nie można utworzyć nowego węzła, gdyż brakuje atrybutu względem, którego można dokonać podziału [8]. Przykład takiej sytuacji zaprezentowano na rysunku 2.

Nr krotki	Wiek	Płeć	Zamieszkanie	Odpowiedź
5	dzieci	chłopiec	wieś	Tak
14	dzieci	chłopiec	wieś	Nie
22	dzieci	chłopiec	wieś	Tak

Rysunek 2. Przykład krotek umieszczonych w liściu drzewa decyzyjnego mimo różnych wartości atrybutu decyzyjnego

Istnieje wiele algorytmów budujących klasyfikatory drzewiaste na przykład: CART, ID3, C4.5, CAL5, SPRINT oraz inne [15]. Jednym z prostszych algorytmów jest ID3 [13]. Algorytm ten wykorzystuje ideę zysku informacji lub redukcji entropii do podjęcia decyzji w zakresie wyboru atrybutu względem, którego nastąpi podział danych wejściowych.

Zakładając, że zbiór danych wejściowych T ma K wartości i każdej wartości przypisano prawdopodobieństwo p_i, gdzie $i \in \langle 1, K \rangle$, opisujące możliwość jej wystąpienia w zbiorze T, to entropię zbioru T opisać można wzorem [13]:

$$H(T) = -\sum_{i=1}^{K} p_i \cdot log_2(p_i) \tag{1}$$

Niech S jest podziałem, który dzieli zbiór T na K podzbiorów T_j, gdzie $j \in \langle 1, K \rangle$. Ilość informacji potrzebnej do opisania przynależności elementu do podzbioru można obliczyć jako ważoną sumę entropii dla każdego z podzbiorów z osobna, co daje wzór [13]:

$$H_S(T) = -\sum_{j=1}^{K} P_j \cdot H\left(T_j\right) \tag{2}$$

gdzie P_j opisuje procent krotek w j podzbiorze. Zysk informacji lub redukcję entropii można wyznaczyć z formuły [13]:

$$zysk(S) = H(T) - H_S(T) \quad (3)$$

Na podstawie wybranego podziału budowany jest węzeł. Ważną cechą algorytmu ID3 jest jego rekurencyjne wywoływanie dla wszystkich podzbiorów. Zakończenie algorytmu następuje gdy nie istnieje możliwość utworzenia dalszych węzłów decyzyjnych.

Przyjmuje się, że tworzenie drzew decyzyjnych ma złożoność czasową określoną formułą [11]:

$$O(dfNlogN) \quad (4)$$

gdzie d oznacza liczbę węzłów w drzewie, f jest liczbą atrybutów a N jest ilością krotek wejściowych.

3. Środowisko Apache Spark – wprowadzenie

Istnieje wiele modeli programowania współbieżnego przetwarzających duże zbiory danych i korzystających z rozwiązań cloud computing. Jednym z popularniejszych jest model MapReduce. Główną zaletą tego modelu jest umożliwienie łatwego rozproszenia operacji celem ich współbieżnego wykonania [16].

Implementacją modelu MapReduce, udostępnioną na zasadach otwartego oprogramowania, jest projekt Apache Hadoop zarządzany przez Apache Foundation przy wsparciu takich firm jak Google Inc. i Yahoo!. Ważnym elementem Apache Hadoop jest Hadoop Distributed File System (HDFS), czyli rozproszony system plików pozwalający na przechowywanie danych w miejscu dostępnym dla wszystkich węzłów roboczych wchodzących w skład klastra. Jest to także przyczyna problemów z wydajnością Apache Hadoop, który bardzo intensywnie korzysta z możliwości zapisywania danych w HDFS [17].

Innym bardzo ważnym elementem Apache Hadoop jest Apache Hadoop YARN (ang. Yet Another Resource Negotiator). YARN jest systemem planowania obciążeń, który zarządza realizacją zadań użytkowników. Separacja zarządzania zasobami od modelu programistycznego zlecającego zadania, dała możliwość współkorzystania z klastra obliczeniowego przez inne środowiska programistyczne [18]. Jako przykłady można wymienić [19]: Dryad, Apache Giraph, Apache Spark, Apache Storm, Apache Tez.

Interesującym rozwiązaniem rozbudowującym model MapReduce przy jednoczesnym zwiększeniu wydajności jest Apache Spark [3]. Jest to środowisko, udostępnione na zasadach wolnego oprogramowania, do realizacji obliczeń w klastrach

obliczeniowych. Dostarcza ono przyjaznego użytkownikowi, wysokopoziomowego interfejsu programistycznego dostępnego w: Scali, Javie i Pythonie.

Główną abstrakcją udostępnianą przez Apache Spark, jest Resilient Distributed Datasets (RDD) [20]. Jest to zbiór obiektów dostępny w trybie tylko do odczytu. Elementy tego zbioru są przechowywane w węzłach roboczych wchodzących w skład klastra obliczeniowego dokładnie na tej maszynie, na której realizowane są na nich obliczenia. Zapobiega to zbędnemu przesyłaniu danych poprzez sieć. Należy tu podkreślić, że istnieje możliwość odtworzenia RDD na podstawie przechowywanych informacji dotyczących sposobu ich utworzenia. Czyni to RDD odpornymi na uszkodzenia węzłów. Część informacji przechowywana na uszkodzonym węźle, jest bowiem odtwarzana na innym, sprawnym węźle. Przyśpieszeniu obliczeń służy też buforowanie RDD w pamięci węzłów. Przyśpiesza to dostęp do danych przechowywanych w RDD w przypadku ponownego użycia [20].

RDD mogą być tworzone na kilka sposobów [21]. Pierwszym z nich jest utworzenie RDD z pliku, który wczytywany jest bezpośrednio z systemu plików, na przykład z HDFS, choć Apache Spark umożliwia także wczytanie pliku, z systemu plików węzłów klastra. Drugą możliwością utworzenia RDD jest zrównoleglenie innych obiektów na przykład tablic. Trzecia możliwość polega na transformacji jednego RDD w inny. Przykładem takiej operacji jest „map" znana z modelu z MapReduce. Apache Spark dostarcza jednak znacznie więcej takich operacji. Ich dokładny opis znajduje się w pozycjach [22], [23]. Ostatnią, czwartą możliwością jest zmiana stanu istniejącego RDD za pomocą dwu akcji: *persist* i *save*. Akcja *persist* wskazuje, że RDD powinno być przechowywane w pamięci. Jest to tylko podpowiedź, gdyż w klastrze może brakować pamięci do przechowywania całego RDD. Akcja *save* powoduje zapisanie zawartości RDD do pliku.

W Apache Spark istnieją także dwa ograniczone rodzaje zmiennych współdzielonych. Pierwszym rodzajem są zmienne rozgłoszeniowe. Zapis do tych zmiennych jest możliwy tylko w momencie ich tworzenia. Później możliwy jest już tylko ich odczyt. Ich charakterystyczną cechą jest dostępność w każdym węźle roboczym. Mogą być używane do rozsyłania danych nawet o dużych rozmiarach [21]. Drugi typ to akumulatory, czyli zmienne posiadające tylko metody dodawania i sprawdzania ich zawartości. Można ich użyć w charakterze kolektora informacji, co daje możliwość tworzenia dzienników opisujących przebieg wykonywania operacji transformacji RDD [21].

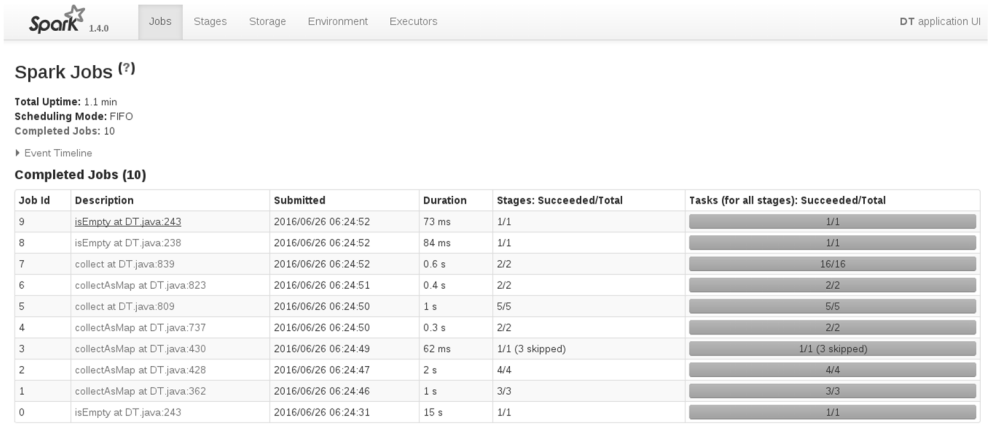
Zadaniem programisty jest stworzenie tak zwanego drivera, w którym opisuje się ciąg przekształceń RDD, a więc stworzenie zbioru operacji typu: tworzenie RDD, przekształcanie RDD, gromadzenie lub zapisywanie danych. Wartym podkreślenia

jest fakt, że driver może być pisany z wykorzystaniem jednego z trzech wspomnianych wcześniej języków programowania: Scali, Javy lub Pythona. Dostępne są także instrukcje warunkowe czy też pętli. Pozwala to na stworzenie iteracyjnych i rozgałęzionych algorytmów transformacji RDD. Trzeba także nadmienić, że Apache Spark pozwala także na pracę z pairRDD, czyli RDD zawierającymi krotki wartości. Daje to możliwość przetwarzania danych typu klucz – tablica wartości [21]–[23].

Uruchomienie drivera polega na jego analizie przez interpreter Apache Spark, który tworzy grafy skierowane, acykliczne (DAG) reprezentujące wyżej wymienione operacje [22], [24], [25]. Następnie tworzony jest plan wykonania, w ramach którego wykonywane są różnego rodzaju optymalizacje, takie jak tworzenie potoków operacji oraz ich łączenie w etapy (ang. *stages*) bazując na potrzebie ewentualnej reorganizacji danych [22], [24].

W trakcie powyższych działań operacje transformacji RDD są przygotowywane do współbieżnej realizacji na węzłach roboczych poprzez stworzenie z nich zadań (ang. *task*) [22], [24]. Kolejnym etapem realizacji programu jest nadzorowanie wykonywania zadań przez programy wykonawcze (ang. *executor*) pracujące pod kontrolą menadżera YARN. Odbywa się to poprzez zlecanie prac (ang. *jobs*) zawierających jeden lub więcej etapów [22].

Wygenerowane DAG oraz realizowane przez programy prace można obserwować za pomocą wbudowanego interfejsu Spark Web UI. Jest to interaktywna witryna internetowa pozwalająca na szczegółową analizę zleconych etapów. Na rysunku 3 zaprezentowano zawartość Spark Web UI pokazującego wykaz realizowanych prac (ang. *jobs*) w ramach realizowanej aplikacji.



Rysunek 3. Zawartość Spark Web UI pokazująca wykaz realizowanych prac

Korzystając z górnego menu można uzyskać dostęp do szczegółowych informacji o:

- etapach prac (ang. *stages*),
- sposobach przechowywania RDD (ang. *storage*),
- konfiguracji środowiska uruchomieniowego oraz węzłach roboczych.

Widoczna poniżej menu tabela prezentuje dane odnoszące się do całej aplikacji i zawiera informacje o: kolejnych numerach zadań, ich nazwie, znaczniku czasowym pokazującym czas zlecenia poszczególnych zadań, czas trwania tych zadań, liczbie zrealizowanych etapów w ramach zadania z podaniem informacji o etapach zakończonych sukcesem. Należy tu podkreślić, że nazwa pracy, wyprowadzona z nazwy operacji transformującej RDD, jest także linkiem prowadzącym do części witryny prezentującej szczegółowe informacje o niej tj. wspomniany już DAG oraz szczegółowe informacje o etapach prac (ang. *stages*).

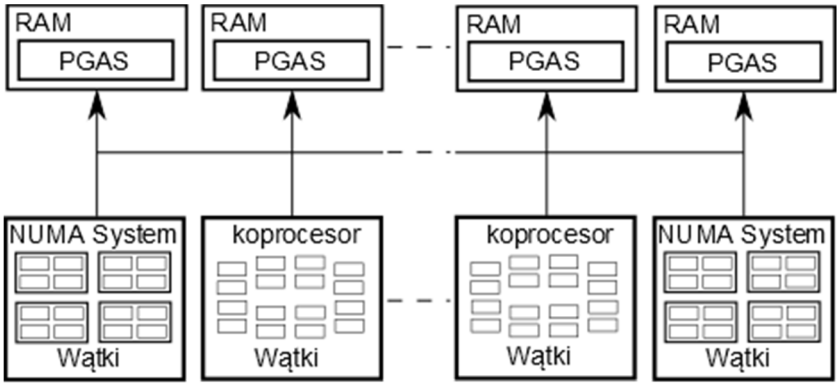
4. Wprowadzenie do środowiska GASPI

Innym modelem programowania współbieżnego, który może zostać wykorzystany do przetwarzania dużych zbiorów danych, jest model Podzielonej Globalnej Przestrzeni Adresowej (Partitioned Global Address Space, PGAS) [4]. Model ten oferuje globalną przestrzeń pamięci utworzoną z pamięci wszystkich procesorów wchodzących w skład chmury obliczeniowej. Pozwala to na dostęp przez dowolny procesor do dowolnego elementu tej przestrzeni. Istnieje jednak powiązanie pomiędzy procesorem a podprzestrzenią pamięci globalnej, którą udostępnia. Dzięki temu można zarządzać umieszczaniem danych w pamięci w taki sposób, aby znajdowały się fizycznie jak najbliżej procesora, który z nich korzysta. Dodatkowo pobieranie danych odbywa się asynchronicznie z wykorzystaniem komunikacji jednostronnej. Rozwiązanie takie pozwala na wykorzystanie sprzętowych mechanizmów usprawniających dostęp do danych w modelu PGAS takich jak Zdalny Bezpośredni Dostęp do Pamięci (ang. Remote Direct Memory Access, RDMA) [4], [26]. Efektem tego jest minimalizacja czasu dostępu do danych.

Dostępnych jest wiele języków programowania, które wykorzystują model PGAS: Co-Array Fortran (CAF) Unified Parallel C (UPC), Chapel, Titanium, X10 i inne wskazane w pracy [4]. Istnieją jednak rozwiązania, które uzupełniają istniejące języki programowania o interfejs, który pozwala na wykorzystanie koncepcji PGAS w tworzonym programie.

Jednym z nich jest interfejs programistyczny globalnej przestrzeni adresowej (ang. Global Address Space Programming Interface, GASPI) [27]. Ważną możliwością dostarczaną przez GASPI jest możliwość mapowania pamięci heterogenicznej takiej jak pamięć główna GPGPU lub Xeon Phi na wskazany segment PGAS [28].

Inną ważną cechą GASPI jest odporność na błędy w dostępie do segmentów pamięci, realizowana w formie mechanizmu przekroczenia czasu dostępu. Rozwiązanie to pozwala wykrywanie błędów dostępu do pamięci i odpowiednie reagowanie na nie. Pozwala to także na zwiększanie lub zmniejszanie ilości węzłów obliczeniowych w trakcie realizacji programu [28]. Obraz architektury pamięci widziany poprzez interfejs programistyczny globalnej przestrzeni adresowej zaprezentowano na rysunku 4. Przestrzeń pamięci utworzona jest tam z pamięci udostępnionej przez system zbudowany w architekturze NUMA oraz z pamięci dostępnej w dołączonych do systemu koprocesorach lub procesorach GPGPU.



Rysunek 4. Architektura pamięci udostępniana przez GASPI

Zrównoleglenie programu napisanego z użyciem GASPI odbywa się z wykorzystaniem technik SPMD (ang. *Single Program Multiple Data*) lub MPMD (ang. *Multiple Program Multiple Data*). Uruchomienie programu lub programów GASPI powoduje stworzenie procesów, którym przypisywane są unikalne numery zwane rank. Numery te wykorzystywane są także do tworzenia grup procesów z użyciem procedur gaspi_group_create, gaspi_group_add, gaspi_group_commit. Tak stworzone grupy procesów mogą być użyte do realizacji komunikacji jednostronnej (ang. *one-sided communication*) lub komunikacji zbiorowej (ang. *collective communication*) [5].

Komunikacja jednostronna to podstawowy mechanizm komunikacji dostarczany przez GASPI. W celu realizacji komunikacji jednostronnej wewnątrz grup tworzy się segmenty z użyciem funkcji gaspi_segment_create, a następnie wykorzystując procedurę gaspi_segment_ptr pozyskuje wskaźniki do segmentów lokalnego i zdalnego. Segment lokalny, to ten który dostępny jest w procesie do zapisywania danych, a segment zdalny to ten który dostępny jest w procesie do odczytu danych.

Należy tu podkreślić, że na poziomie GASPI jest to podział umowny. Rolą programisty jest wskazanie jednego z segmentów jako lokalnego, czyli do zapisu danych, a drugiego jako zdalnego – do odczytu. Wywołanie funkcji gaspi_write prowadzi do zapisania danych w zdalnej lokacji, czyli w segmencie, który ma mieć dostęp do tych danych. Jak już wspomniano, jest to komunikacja jednostronna, co oznacza, że odbywa się ona z wykorzystaniem mechanizmu RDMA. Nie blokuje ona wykonywania programu, co pozwala na realizację fazy komunikacji równolegle z realizacją samego programu.

Funkcja gaspi_write może zwrócić trzy wartości: GASPI_SUCCESS, GASPI_ERROR oraz GASPI_TIMEOUT. Ostatnia ze zwracanych wartości oznacza, że przekroczono czas przesyłania danych do sprzętu, co może się zdarzyć gdy inny proces próbuje zapisać dane w tym samym segmencie.

Warto w tym miejscu zauważyć, że wartość GASPI_SUCCESS oznacza tylko, że przesyłane dane dotarły do warstwy sieciowej, natomiast nie jest to potwierdzenie, że dotarły do zdalnej lokacji. Pozwala to na zapisywanie kolejnych danych do segmentu podczas trwania transferu poprzednich danych. Brak potwierdzenia dotarcia na miejsce przeznaczenia oznacza, że należy użyć dodatkowej funkcji pozwalającej na pozyskanie takiego potwierdzenia.

Taką funkcją jest gaspi_notify, która wysyła żądanie potwierdzenia odbioru danych. Wadą tego rozwiązania, jest korzystanie z tego samego mechanizmu co gaspi_write, tak więc zwrócenie GASPI_SUCCESS nie sygnalizuje, że potwierdzenie transmisji przybyło z miejsca przeznaczenia. Wynika stąd konieczność, synchronizacji operacji zapisu i odczytu danych tak, aby inne procesy wykorzystywały zapisywaną wartość, a nie wartość poprzednią. Rozwiązaniem tego problemu jest użycie funkcji gaspi_barrier, która działa w ten sposób, że blokuje wykonywanie programu we wszystkich procesach do momentu, aż wszystkie ją uruchomią [5].

Ważną grupą funkcji udostępnianych przez interfejs GASPI są funkcje realizujące operacje redukcji w formie komunikacji zbiorowej. Są to dwie funkcje gaspi_allreduce oraz gaspi_allreduce_user, które realizują dokładnie te same zadania co funkcja MPI_Allreduce [29]. Ich ważną cechą jest to, że realizacja tych funkcji synchronizuje procesy je wywołujące. W GASPI zdefiniowano kilka rodzajów operacji typów redukcji danych, takich jak: GASPI_OP_MIN, GASPI_OP_MAX, GASPI_OP_SUM operujących na zmiennych typu całkowitego i zmiennoprzecinkowego. Istnieje jednak możliwość zdefiniowania przez użytkownika własnego typu operacji [5].

5. Implementacje algorytmu ID3 w środowisku Spark

Implementacja algorytmu ID3 w frameworku Spark polega na opisaniu za pomocą modułu programowego zwanego driverem kolejnych przekształceń RDD oraz obliczeń wykonywanych na danych uzyskanych z RDD. Iteracyjność algorytmu ID3 powoduje, że zestawy niektórych przekształceń RDD będą wielokrotnie powtarzane. Wartą zauważenia jest także konieczność realizacji alternatywnych przekształceń RDD, uzależnionych od wartości danych pośrednich uzyskiwanych z RDD.

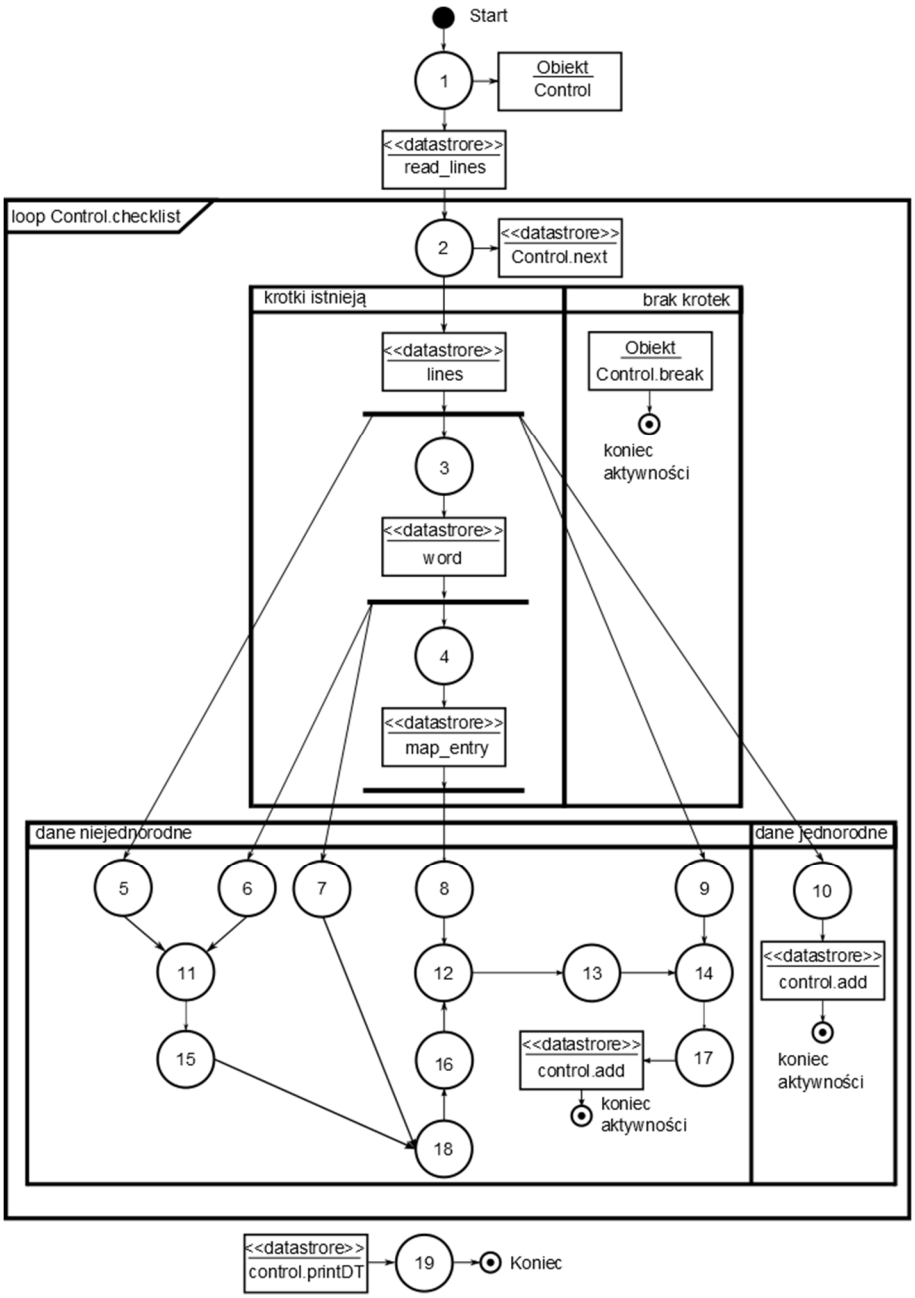
Powyższe uwagi prowadzą do zaproponowania formy opisu implementacji algorytmu ID3 w środowisku Spark będącej rozwiązaniem pośrednim pomiędzy diagramem czynności oferowanym przez język UML [30], a diagramem przepływu danych DFD [31]. Pośredniość wyraża się w zastosowaniu z jednej strony elementów procesu (transformacji) i przepływu danych pochodzących z DFD, a z drugiej strony magazynu danych, obiektu, węzłów: rozwidlenia, końcowego i początkowego oraz partycji aktywności pochodzących z diagramu czynności UML.

Elementy takie jak: proces, przepływ danych, magazyn danych, węzły rozwidlenia, końcowy i początkowy służą do zamodelowania przepływu danych. Elementów obiekt oraz partycja aktywności użyto do zamodelowania sterownia przepływem danych. Takie połączone zasady modelowania pozwalają w pełni zobrazować przebieg przekształceń RDD, opisanych w driverze.

Model przekształceń RDD prowadzący do realizacji algorytmu ID3 w Spark zobrazowano na rysunku 5. Ważnym obiektem pojawiającym się na tym rysunku jest obiekt Control. Pełni on funkcje gromadzenia i przechowywania danych związanych z tworzonym drzewem oraz udostępnia informacje pozwalające na kontrolowanie przepływu danych.

Ta różnorodność funkcji wyraża się na rysunku 5 poprzez umieszczenie obiektu w charakterze:

- magazynu danych, który pobiera dane za pomocą wywołania Control.add, zwraca informacje dla procesu filtrującego (2) wywołaniem Control.next lub zwraca całe drzewo decyzji za pomocą wywołania Control.printDT,
- kontrolera wykonywanej pętli za pomocą wywołania Control.checklist oraz Control.break.



Rysunek 5. Przepływ danych i sterowania w driverze środowiska Spark

Warto tu zauważyć, że wywołanie Control.checklist wewnętrznie korzysta ze zgromadzonych danych drzewa, celem sprawdzenia, czy utworzone liście drzewa mogą być przekształcone w węzły decyzyjne. Pociąga to wykonanie kolejnej iteracji pętli.

```
JavaPairRDD<String, Integer> decision =
  words.filter(new Function<Tuple2<String,Integer>,Boolean>() {
    public Boolean call(Tuple2<String,Integer> val) {
      if (val._2<1) return true;
      else return false;
    }
});

JavaPairRDD<String,Iterable<Integer>> counts_decision = decision.groupByKey();

JavaPairRDD<String,Integer> list_to_entropy =
counts_decision.flatMapToPair(
        new PairFlatMapFunction<Tuple2<String,Iterable<Integer>>,String,Integer>() {
           public Iterable<Tuple2<String,Integer>> call(Tuple2<String,Iterable<Integer>> tuple) {
               Integer count_true = 0;
               Integer count_false = 0;
                   for(Integer iter : tuple._2) {
                     if (iter==0) count_true++;
                     else count_false++;
                   }
                   List<Tuple2<String,Integer>> result = new ArrayList<Tuple2<String,Integer>>();
                   result.add(new Tuple2<String,Integer>("true",count_true));
                   result.add(new Tuple2<String,Integer>("false",count_false));
                   return result;
    }
});
```

Rysunek 6. Implementacja przekształceń filter, groupByKey, flatMapToPair

Algorytm rozpoczyna się procesem 1 polegającym na wczytaniu danych i utworzeniu pierwszego RDD o nazwie read_lines. Równocześnie tworzony jest obiekt Control sterujący przepływem danych oraz przechowujący elementy utworzonego drzewa danych.

Następnie realizowany jest proces 2 filtrujący dane z wykorzystaniem danych zawartych w obiekcie Control. Celem procesu jest utworzenie zbioru krotek odpowiadającego aktualnie analizowanemu liściowi drzewa decyzyjnego. Proces realizowany jest aż do momentu znalezienia liścia, dla którego zbiór krotek nie jest pusty. Brak takiego liścia oznacza, że proces budowania drzewa decyzji jest ukończony [8].

Jest to sprawdzane w następnym kroku, poprzez kontrolę czy magazyn danych, RDD o nazwie lines, zawierający produkt procesu filtrowania, jest pusty. Należy tu

zauważyć, że jest to potencjalnie kosztowny test, gdyż elementy RDD mogą być przechowywane na różnych węzłach roboczych.

Jeśli test wypadnie negatywnie realizowane jest przerwanie pętli za pomocą wywołania Control.break w partycji aktywności o nazwie: *brak krotek*. Po przerwaniu pętli następuje realizacja procesu 19 zapisującego utworzone drzewo decyzyjne do pliku tekstowego oraz zakończenie programu.

W przeciwnym wypadku realizowane są procesy partycji aktywności o nazwie: *krotki istnieją*. Pierwszym z nich jest proces 3 wykonujący transformację RDD o nazwie flatMapToPair [22], której celem jest pocięcie wczytanych linii danych na krotki.

Następnie realizowany jest zbiór transformacji procesu 4: filter, groupByKey, flatMapToPair, groupByKey, collectAsMap [22], [23] przekształcający krotki danych w magazyn danych map_entry. Kod trzech pierwszych transformacji to jest: filter, groupByKey, flatMapToPair został zawarty na rysunku 6.

Magazyn danych map_entry zawiera zliczone wartości atrybutu decyzyjnego. Jak wspomniano w sekcji 2 atrybut ten może przyjmować wartości Tak lub Nie. Gdy ilość którejkolwiek z wartości równa się zero, to zgodnie z wzorem (1) entropia również przyjmuje wartość zero. Oznacza to, że w liściu zgromadzono krotki danych jednorodne pod względem atrybutu decyzyjnego, a więc nie będzie on węzłem decyzyjnym. Z punktu widzenia opisywanego algorytmu, powoduje to realizację partycji aktywności o nazwie: *dane jednorodne*, czyli zapisanie liścia do drzewa za pomocą wywołania *Control.add*.

W przeciwnym przypadku należy ustalić, czy liść może być przekształcony w węzeł decyzyjny co realizowane jest w partycji aktywności: *dane niejednorodne.* Jest to jedna z najbardziej rozbudowanych partycji aktywności. Można w niej wyróżnić kilka ciągów procesów. Jednym z nich jest ciąg procesów 5, 6, 11 i 15, który docelowo oblicza entropię możliwych podziałów. Odbywa się to poprzez:

- zestaw przekształceń procesu 5: flatMapToPair, filter, flatMapToPair, groupByKey [22], [23], który prowadzi do przygotowania zestawienia krotek danych powiązanych w pary z wartościami zmiennej decyzyjnej,
- zestaw przekształceń procesu 6: flatMapToPair, groupByKey [22], [23], który prowadzi do zestawienia krotek danych, w sposób umożliwiający zliczenie krotek z tą samą wartością,
- przekształcenia cogroup [22], [23] oznaczonego jako proces 11, które z produktów procesu 5 i 6 tworzy jeden RDD zawierający wszystkie dane niezbędne do wyliczenia entropii podziału; przekształcenie to realizowane z wykorzystaniem następującego kodu: JavaPairRDD<String, Tuple2<Iterable<Iterable<Integer>>,Iterable<Iterable<Integer>>>> to_local_entropy = to_local_counts_decision.cogroup(counts_values);,

- przekształcenia flatMapToPair [22], [23] oznaczonego jako proces 15 prowadzącego do obliczenia entropii z wzoru (1) dla każdego z możliwych podziałów.

Kolejny ciąg procesów 18, 16, 12 i 8 prowadzi do wyznaczenia zysku dla każdego możliwego podziału. Realizacja tego zadania odbywa się poprzez:

- przekształcenie flatMapToPair [22], [23] procesu 18, które oblicza wartość $P_j H(T_j)$ z wzoru (2), dla każdego możliwego podziału,
- przekształcenia groupBykey [22], [23] w procesie 16 tworzącego RDD zawierającego wartości $P_j H\left(T_j\right)$ pogrupowane względem nazw atrybutów opisowych,
- zestawu obliczeń oznaczonych jako proces 8, które na podstawie zmiennej map_entropy wyznaczają wartość entropii przed podziałem,
- przekształcenia mapValues [22], [23] w procesie 12, które na podstawie wyniku obliczeń w procesie 8 oraz produktu procesu 16 dokonuje obliczenia zysku dla każdego z możliwych podziałów; kod tego przekształcenia zaprezentowano na rysunku 7.

```
final Broadcast<Double> entropy_br = ctx.broadcast(entropy_val);

JavaPairRDD<String,Double> gain = list_HST.mapValues(new Function<Iterable<Double>,Double>() {
  public Double call(Iterable<Double> val) {
    Double result=0.0;
    for(Double iter : val) {
      result=result+iter;
    }
    Double entropy_val=entropy_br.getValue();
    return entropy_val-result;
  }
```

Rysunek 7. Implementacja przekształcenie mapValues realizujące obliczenie zysku dla każdego z możliwych podziałów

Produkt procesu 12 jest następnie przekształcany poprzez zestaw transformacji procesu 13: mapToPair, SortByKey, collect [22], [23] celem wyznaczenia wartości maksymalnego zysku i ustalenia na tej podstawie jaki atrybut opisowy stworzy węzeł decyzyjny z liścia. Działanie to realizowane jest w ciągu procesów 9, 14 i 17 które korzystają z produktu procesu 13.

W procesie 9 poprzez przekształcenie groupByKey oraz collectAsMap [22], [23] wyznacza się mapę wszystkich wartości krotek opisowych. Następnie produkt tego procesu wykorzystuje się w procesie 14, aby wybrać z niego możliwe wartości atrybutu opisowego, dającego największy zysk przy podziale. Proces 17 wykorzystuje uzyskaną

listę do stworzenia z niej RDD i zapisania w magazynie danych, przy użyciu wywołania control.add, opisu węzła decyzyjnego drzewa.

6. Realizacja w języku C++ z wykorzystaniem GASPI

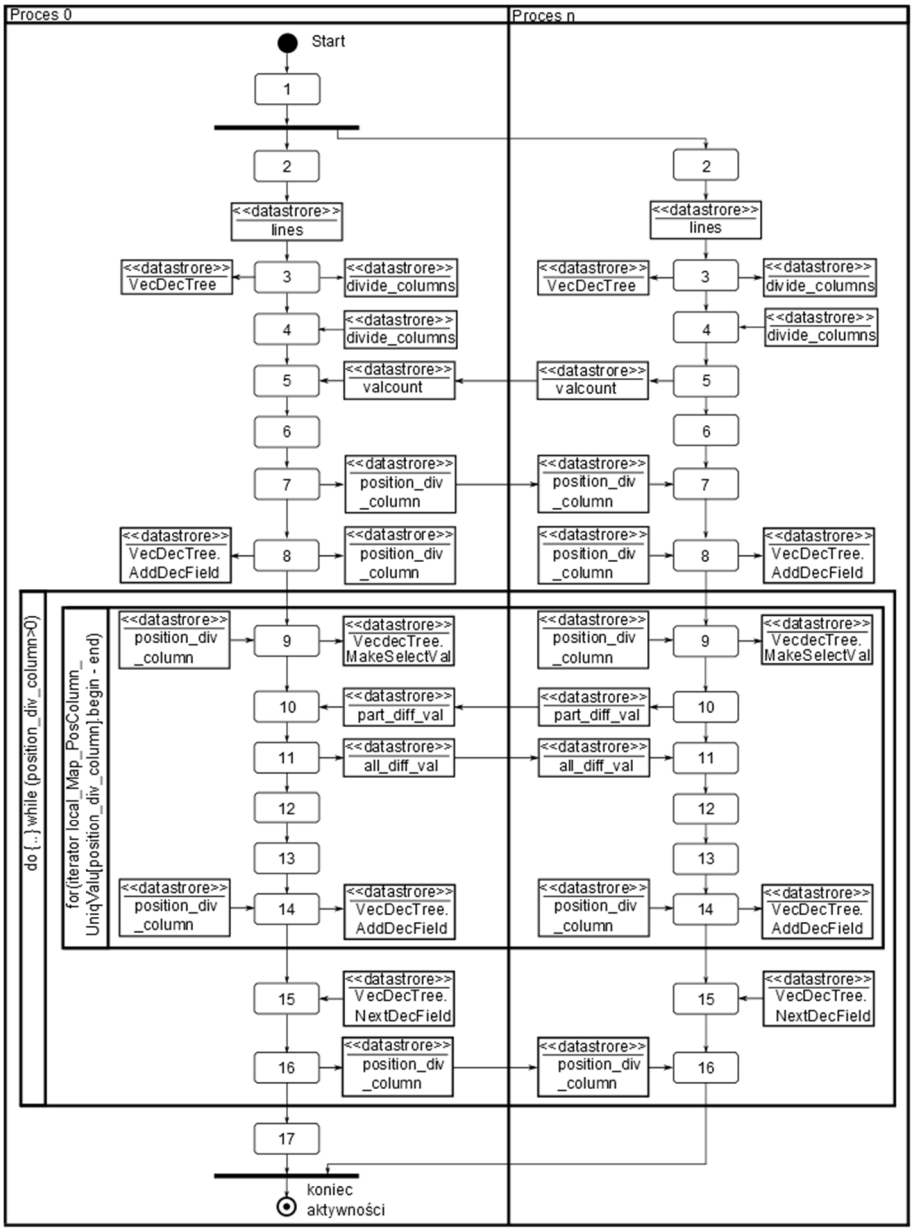
Zgodnie z pracami [10], [15] istnieje kilka strategii implementacji algorytmu tworzącego drzewa decyzyjne. Do celów implementacji algorytmu ID3 w GASPI wybrano strategię zakładającą podział zbioru krotek danych wejściowych pomiędzy wszystkie procesy. Zatem wykonują one swoje zadania dla części krotek tego zbioru, po czym zwracają wyniki cząstkowe, które agregowane są w procesie 0. Szczegółowy opis tych działań zostanie przedstawiony w dalszej części tej sekcji.

Należy zwrócić uwagę, że interfejs programistyczny globalnej przestrzeni adresowej (GASPI) jest zaimplementowany jako biblioteka uzupełniająca język C++ o interfejs modelu PGAS. Jest to rozwiązanie niskiego poziomu, które wymaga opracowania odpowiedniego zestawu klas obudowujących i ułatwiających jego wykorzystanie. Powoduje to, wzrost stopnia komplikacji implementacji algorytmu ID3.

Powyższe uwagi prowadzą do zastosowania opisu implementacji algorytmu ID3 wykorzystującego diagram czynności UML. Elementy: czynność, przepływ sterujący, węzły: początkowy, końcowy, rozwidlenia i scalenia służą do zamodelowania przepływu sterowania pomiędzy czynnościami i węzłami diagramu. Przepływ danych oraz magazyn danych użyte zostały do prezentacji przepływu danych pomiędzy rank, które zamodelowano z użyciem partycji aktywności.

Diagram czynności prowadzący do realizacji algorytmu ID3 w GASPI zobrazowano na rysunku 8. Na diagramie wyróżniono proces 0, od którego rozpoczyna się wykonywanie całego programu. Oznaczenie „proces n” symbolizuje natomiast każdy inny proces. Czynności i przepływy w partycji aktywności oznaczonej tym symbolem, są więc takie same dla wszystkich procesów, za wyjątkiem procesu 0, różnią się tylko przetwarzanymi danymi.

Ważnym magazynem danych, który pojawia się na rysunku 8 jest VecDecTree, który pełni funkcje przechowywania danych opisujących już utworzone drzewo decyzji. Jednak, jak widać na rysunku, oprócz nazwy magazynu używana jest także nazwa występująca po kropce. Konwencji tej użyto aby podkreślić fakt, iż magazyn danych VecDecTree jest w istocie obiektem. W efekcie można było zamodelować różne metody zwracania danych przechowywanych w obiekcie oraz metodę ich pobierania.



Rysunek 8. Diagram czynności prowadzący do implementacji algorytmu ID3 w GASPI

Program rozpoczyna się czynnością oznaczoną jako 1, która inicjuje środowisko GASPI i doprowadza do uruchomienia programu w innych węzłach. Na rysunku tę funkcję zamodelowano węzłem rozwidlenia. Należy zauważyć, że przepływ sterujący wychodzący z rozwidlenia i prowadzący do partycji aktywności oznaczonej jako proces n, symbolizuje wiele procesów, co oznacza, że pojedynczy przepływ sterowania na rysunku symbolizuje w rzeczywistości tyle przepływów, ile uruchomionych zostało procesów.

Następnie, w każdym procesie realizowana jest czynność oznaczona jako 2. Polega ona na odczycie danych wejściowych, przy czym zgodnie z przyjętą strategią implementacji algorytmu, każdy proces wczytuje swój zestaw krotek danych wejściowych. Wczytywane dane są przechowywane w magazynie danych lines, będącym wektorem map typów int i string. Zawierają one powiązanie pomiędzy nazwą atrybutu opisowego a jego wartością.

Taka struktura podyktowana została sposobem realizacji zadań, przez algorytm ID3, które sprowadzają się do zliczania występowania poszczególnych wartości atrybutu opisowego i atrybutu decyzyjnego w pojedynczej krotce, o czym będzie mowa w dalszej części sekcji. Pole int w opisywanej mapie odwzorowuje numer kolumny – atrybut opisowy lub atrybut decyzyjny. Wartości tych atrybutów przechowywane są w polu typu string.

Trzecia czynność w obu partycjach prowadzi do utworzenia wspomnianego już magazynu danych VecDecTree oraz magazyn danych divide_columns będącego wektorem złożonym ze struktur TDecisionPairStruct. Magazyn danych divide_columns służy do utworzenia „ścieżki dostępu" poprowadzonej od korzenia drzewa decyzji do analizowanego węzła. Informacja ta wykorzystywana jest później w trakcie selekcji krotek danych wejściowych przypisanych do analizowanego liścia drzewa decyzji, celem zliczenia wystąpienia poszczególnych wartości atrybutów opisowych i atrybutu decyzyjnego. Kolejną czwartą czynnością jest zliczenie wartości atrybutu decyzyjnego dla wszystkich atrybutów opisowych przypisanych do węzła drzewa wskazanego przy użyciu magazynu danych divide_columns. W tym kroku programu realizowane jest także zliczanie wystąpień wartości atrybutu decyzyjnego. Wyniki tego zliczania pozwalają na obliczenie wartości entropii przed podziałem zbioru krotek przypisanych do danego węzła. Należy tu zaznaczyć, że zliczanie odbywa się jednocześnie dla wszystkich atrybutów opisowych i atrybutu decyzyjnego w danej krotce. Takie rozwiązanie pozwala na jednokrotne przeglądnięcie zbioru krotek w danym procesie, co przy dużym rozmiarze danych zwiększa wydajność tego procesu.

```
vector<Telement> dectree::CountTheFileValues(head_val temp_head_val,
                         vector<TDecisionPairStruct> pos_div_cols) {
  vector<Telement> valcount;
  valcount=filein.searchvalcount(temp_head_val.pos,
     Map_PosColumn_UniqValu[temp_head_val.pos],pos_div_cols);
  string nrlist=filein.getnrselrows();
  int buff_nr_lenght=communication->GetMaxIntFromAll(nrlist.size()+1);
  list_of_all_lines=communication->GetStrFromAll(nrlist,buff_nr_lenght);
  int buff_lenght=communication->GetMaxIntFromAll(valcount.size());
  if (buff_lenght==0) buff_lenght=1;
  return communication->GetVectorFromAll(valcount,buff_lenght);
}
```

Rysunek 9. Implementacja metody CountTheFileValues

Samo zliczanie wystąpień wartości atrybutów opisowych i decyzyjnego wykonywane jest poprzez metodę CountTheFileValues, której tekst zaprezentowano na rysunku 9. Jej ważną częścią jest agregacja wyników cząstkowych uzyskanych z innych procesów. Na rysunku 8 zamodelowano to działanie za pomocą: czynności oznaczonej jako 5 czyli przepływu danych oraz magazynów danych valcount. W kodzie zadanie to realizowane jest przez metody obiektu communication.

Metoda GetStrFromAll pobiera od wszystkich procesów ciągi znaków, w których zapisano numery krotek przypisanych do danego węzła. Tekst tej metody podano poniżej:

```
string TCommunictaion::GetStrFromAll(string sdiffval,int buff_lenght) {
  string all_diff_val;
  TSharedMem<char> SharedMemChar;
  SharedMemChar.SetSegment(buff_lenght);
  if (SharedMemChar.GetiProc()==0) {
      SharedMemChar.WriteSegment();
      all_diff_val=sdiffval;
      for(gaspi_rank_t rank=1; rank < SharedMemChar.GetnProc(); ++rank) {
     char *rcv_array = SharedMemChar.GetRcvArray(rank);
     all_diff_val=all_diff_val+string(rcv_array);
      }
  } else {
    char *src_array = SharedMemChar.GetSrcArray();
    strncpy(src_array, sdiffval.c_str(),sdiffval.size()+1);
    SharedMemChar.WriteSegment();
  }
  SharedMemChar.EndSharedMem();
```

```
  return all_diff_val;
}
```

Kolejna metoda obiektu communication, GetMaxIntFromAll zwraca maksymalny rozmiar magazynu danych valcount występującego pośród wszystkich procesach. Jej tekst pokazano poniżej:

```
int TCommunictaion::GetMaxIntFromAll(int buf) {
  int buff_lenght=0;
  TSharedMem<int> SharedMem;
  SharedMem.SetGroup();
  if (SharedMem.GetiProc()==0) {
    buff_lenght=SharedMem.AllReduceMax(buf);
  } else {
    buff_lenght=SharedMem.AllReduceMax(buf);
  }
  SharedMem.EndSharedMem();
  return buff_lenght;
}
```

Rozmiar ten używany jest następnie do ustalenia wielkości segmentu danych, w którym zapisywane będą magazyny danych valcount utworzone w danym procesie i pobierane do procesu 0 metodą GetVectorFromAll, której kod zawarto jest następujący:

```
vector<Telement> TCommunictaion::GetVectorFromAll(vector<Telement> sdiffval,
  int buff_lenght) {
  vector<Telement> all_diff_val, temp_val;
  TSharedMem<Telement> SharedMemVec;
  SharedMemVec.SetSegment(buff_lenght);
  if (SharedMemVec.GetiProc()==0) {
      SharedMemVec.WriteSegment();
      all_diff_val.insert(all_diff_val.end(), sdiffval.begin(), sdiffval.end());
      temp_val.clear();
      for(gaspi_rank_t rank=1; rank < SharedMemVec.GetnProc(); ++rank) {
   Telement *rcv_array = SharedMemVec.GetRcvArray(rank);
   copy(&rcv_array[0], &rcv_array[buff_lenght], back_inserter(temp_val));
   all_diff_val.insert(all_diff_val.end(), temp_val.begin(), temp_val.end());
   temp_val.clear();
```

```
      }
  } else {
    Telement *src_array = SharedMemVec.GetSrcArray();
    copy(sdiffval.begin(), sdiffval.end(), src_array);
    SharedMemVec.WriteSegment();
  }
  SharedMemVec.EndSharedMem();
  return all_diff_val;
}
```

Analizując kod zawarty w tekstach metod GetVectorFromAll, GetStrFromAll i GetMaxIntFromAll zauważyć można, że ich kluczowym elementem jest klasa TSharedMem, będąca obudową procedur i funkcji środowiska GASPI. Jej szczegółowy opis zamieszczony jest na końcu tego rozdziału.

Warto zwrócić uwagę, że ustalenie maksymalnego rozmiaru segmentu w metodzie GetMaxIntFromAll odbywa się bardzo podobnie jak w standardzie MPI [32] to znaczy z wykorzystaniem z funkcji gaspi_allreduce [27].

Cechą charakterystyczną implementacji tej metody jest jej podział instrukcją if na dwie części. Część wykonywana w procesie o numerze 0 rozpoczyna się bezpośrednio po wywołaniu instrukcji if. Jako pierwsza wywoływana jest metoda WriteSegment, która w tym przypadku realizuje, za pomocą funkcji gaspi_barier [5], potwierdzenie odebrania danych oraz synchronizację obu części implementacji w tym punkcie kodu. Następnie realizowane jest dodawanie do siebie odebranych ciągów znaków, które powoduje, że ciągi pochodzące z różnych procesów są scalane. Część wykonywana w innych procesach rozpoczyna się po słowie kluczowym else. Pierwszym zadaniem jest pobranie adresu źródłowego segmentu pamięci. Następnie kopiowana jest do niego zawartość przesyłanego ciągu. Zakończenie tej części odbywa się poprzez wywołanie metody WriteSegment.

Kolejną, szóstą czynnością, jest wyznaczanie numeru atrybutu opisowego, względem którego podział krotek przypisanych do danego węzła drzewa decyzyjnego da największy zysk. Wyznaczona wartość jest następnie zapisywana w magazynie danych position_div_column. Kompletną zawartością magazynu danych *valcount* dysponuje tylko proces 0, przez co, tylko w tym procesie istnieje możliwość wyznaczenia prawidłowej, z punktu widzenia algorytmu ID3, wartości zmiennej position_div_column. Można było, co prawda, przesłać kompletny magazyn valcount do wszystkich procesów, jednak ze względu na jego rozmiar byłoby to rozwiązanie nieefektywne.

Powyższe uwagi wyjaśniają konieczność przesłania wartości position_div_column z procesu 0 do wszystkich innych, co realizowane jest w czynności numer 7. Zadanie

to wykonywane jest w metodzie SendIntToAll, wspomnianego już obiektu communication. Jej implementacja jest tożsama z implementacją metody GetVectorFromAll, z tą tylko różnicą, że drugie wywołanie metody AllReduceMax, tej po słowie kluczowym else, ma podane jako argument 0. W efekcie jako maksymalna wartość, dostępna we wszystkich procesach, wybierana jest wartość podawana w pierwszym wywołaniu metody AllReduceMax.

Wyznaczony numer atrybutu opisowego, przechowywany w zmiennej position_div_column, jest w czynności numer 8, zapisywany w magazynie danych VecDecTree z wykorzystaniem metody AddDecField tworzącej jednocześnie liść drzewa decyzji. Jeśli wartość entropii krotek zgromadzonych w tworzonym liściu jest różna od zera, to metoda AddDecField ustawia flagę oznaczającą, że tworzony liść jest węzłem decyzyjnym. Tak oznaczony liść ma pustą listę związanych z nim węzłów drzewa. Metoda wywoływana jest we wszystkich procesach, co pozwala w późniejszym czasie na uniknięcie konieczności przesyłania danych synchronizujących czynności wykonywane w poszczególnych procesach.

W tym momencie rozpoczyna się iteracyjna część implementacji algorytmu ID3. Główną pętlą jest do..while, której dozorem jest wartość zmiennej position_div_column aktualizowana w każdej iteracji. Odbywa się to z wykorzystaniem magazynu danych VecDecTree i metody NextDecField. Metoda ta realizuje poszukiwanie w utworzonym już drzewie węzłów, które mają podniesioną flagę oznaczającą, że są węzłami decyzyjnymi ale jednocześnie mają pustą listę związanych z nim węzłów drzewa. Jeśli taki liść nie zostanie odnaleziony metoda zwraca wartość NO_DECISION_COLUMN, co wykorzystywane jest do przerwania pętli do..while. Taka realizacja iteracji pozwala na uniknięcie potencjalnie kosztownych i kłopotliwych do synchronizacji pomiędzy procesami, rekurencyjnych wywołań algorytmu ID3 dla każdego nowego liścia drzewa.

W pętli do..while realizowana jest także pętla for, której zadaniem jest wykonanie algorytmu ID3 dla wszystkich możliwych wartości atrybutu opisowego wskazanego zmienną position_div_column. Pierwszą czynnością realizowaną w pętli for jest czynność oznaczona numerem 9 tworząca magazyn danych divide_column, którego dane zawierają „ścieżkę dostępu" do wybranego węzła, zakończoną jedną z możliwych wartości danego atrybutu opisowego. Dla tak wskazanego węzła drzewa decyzji realizowane jest poszukiwanie unikalnych wartości dla wszystkich krotek przypisanych do niego. Działanie to realizowane jest we wszystkich procesach. Z tego powodu należy przesłać wyniki cząstkowe do procesu 0. Wykonywane jest to z użyciem metody GetVectorDiffFromAll, z wspomnianego już obiektu communication, co na rysunku 8 zamodelowano przepływem danych oraz magazynami danych part_diff_val. Implementacja tej metody jest analogiczna do implementacji metody GetStrFromAll z tą różnicą, że zamiast zawartości obiektu String przeszyłana jest zawartość magazynu danych part_diff_val.

Kolejną czynnością oznaczoną jako 12 jest usunięcie, z użyciem funkcji std::unique biblioteki standardowej języka C++, ewentualnych powtórzeń oraz rozesłanie kompletnego zestawu unikalnych wartości do wszystkich procesów metodą SendVectorTDiffToAll obiektu communication. Implementacja tej metody jest analogiczna jak metody GetVectorDiffFromAll z tą różnicą, że w procesie 0 realizowana jest operacja kopiowania zawartości przysłanego wektora a nie jej wstawiania. Należy tu zaznaczyć, że kompletna lista unikalnych wartości jest rozsyłana do wszystkich procesów celem ujednolicenia zwracanych później wyników zliczania.

Po zakończeniu pętli for realizowana jest czynność 15, w ramach której poszukuje się kolejnego liścia drzewa do analizy. Jest to wykonywane w ramach opisanej już wcześniej metody NextDecField obiektu VecDecTree.

Ostatnią czynnością realizowaną w pętli do..while jest czynność 16 rozsyłająca wyszukany numer kolumny, względem której podział krotek przypisanych do danego węzła da największy zysk. Realizacja pętli, a zarazem samego algorytmu kończy się w momencie gdy w zmiennej position_div_column ustawiana jest wartość NO_DECISON_COLUMN, co zgodnie z wcześniejszym opisem oznacza brak liści drzewa decyzji do analizy.

Ostatnią czynnością realizowaną przez opisywany program jest zapisanie stworzonego drzewa decyzji do pliku oraz zakończenie działania środowiska GASPI. Na rysunku 8 zadania te zamodelowano czynnością 18 oraz węzłem scalenia.

Jak wspomniano wcześniej, klasą obudowującą funkcje i procedury środowiska GASPI jest szablon klasy TSharedMem, którego parametrem jest typ tworzonego segmentu GASPI. Klasa szablonowa automatyzuje proces tworzenia segmentów oraz udostępnia metody pozwalające na zapisywanie lub pobieranie danych z utworzonych segmentów.

Konstruktor klasy szablonowej odpowiada za ustawienie wartości początkowych wewnętrznych pól klasy oraz za pobranie z wykorzystaniem funkcji gaspi_proc_rank i gaspi_proc_num [5] informacji odpowiednio o numerze procesu oraz o całkowitej liczbie procesów tworzących system. Metoda SetSegment odpowiada za utworzenie segmentu pamięci GASPI, a jej argumentem jest wielkość segmentu pamięci. Kod metody zaprezentowano na rysunku 10.

Charakterystyczne jest użycie makra SUCCESS_OR_DIE, którego zadaniem jest przerwanie realizacji programu oraz wyświetlenie informacji o przyczynie przerwania działania [33].

```
template<typename C>
void TSharedMem<C>::SetSegment(int buf_len) {
  buffor_lenght=buf_len;
  set_segment=true;
  segment_src_size = nProc * buffor_lenght * sizeof(C);
  segment_dst_size = nProc * buffor_lenght * sizeof(C);
  SUCCESS_OR_DIE(gaspi_segment_create(segment_src_id, segment_src_size,
           GASPI_GROUP_ALL, GASPI_BLOCK, GASPI_MEM_INITIALIZED));
  SUCCESS_OR_DIE(gaspi_segment_create(segment_dst_id,segment_dst_size,
           GASPI_GROUP_ALL, GASPI_BLOCK, GASPI_MEM_INITIALIZED));
  SUCCESS_OR_DIE(gaspi_segment_ptr(segment_src_id, &array_src));
  SUCCESS_OR_DIE(gaspi_segment_ptr(segment_dst_id, &array_dst));
  src_array = (C *)(array_src);
  rcv_array = (C *)(array_dst);
}
```

Rysunek 10. Implementacja metody SetSegment klasy szablonowej TSharedMem

Ważnymi metodami w implementacji klasy szablonowej TSharedMem jest para GetSrcArray i GetRcvArray. Obydwie metody zwracają adres segmentu pamięci GASPI odpowiednio źródłowego, czyli tego gdzie dane zapisujemy, oraz przeznaczenia, to znaczy tego, do którego dane zostaną wysłane. Ich użycie jest uwarunkowane procesem, w którym ich użyto i zależy od realizowanego zdania przesyłania danych. Teksty opisywanych metod zaprezentowano poniżej:

```
template<typename C> C * TSharedMem<C>::GetSrcArray() {
  return &src_array[buffor_lenght * iProc];
}

template<typename C> C * TSharedMem<C>::GetRcvArray(gaspi_rank_t rank) {
  return &rcv_array[buffor_lenght * rank];
}
```

Widać, że w celu uzyskania dostępu do właściwego segmentu globalnej przestrzeni adresowej wykorzystywania jest informacja o długości zarezerwowanego segmentu pamięci oraz o numerze procesu, który chce z niego skorzystać. Rozwiązanie takie ma zapewnić każdemu procesowi bufor służący do umieszczania danych, które są odbierane lub będą wysyłane.

Inną ważną metodą jest WriteSegment. Realizowane jest w niej przesłanie danych z segmentu pamięci należącego do lokalnego procesu, do segmentu pamięci należącego do procesu 0. Kod opisywanej metody zaprezentowano na rysunku 11.

```
template<typename C> void TSharedMem<C>::WriteSegment(void) {
  if (iProc==0) {
    SUCCESS_OR_DIE (gaspi_barrier(GASPI_GROUP_ALL, GASPI_BLOCK));
  } else {
    gaspi_offset_t loc_off=iProc * buffor_lenght * sizeof(C);
    gaspi_offset_t rem_off=iProc * buffor_lenght * sizeof(C);
    gaspi_rank_t rank = 0;
    wait_if_queue_full(queue_id,2);
     SUCCESS_OR_DIE(gaspi_write( segment_src_id, loc_off, rank, segment_dst_id, rem_off,
   buffor_lenght * sizeof(C), queue_id, GASPI_BLOCK));
    SUCCESS_OR_DIE(gaspi_barrier(GASPI_GROUP_ALL, GASPI_BLOCK));
  }
}
```

Rysunek 11. Implementacja metody WriteSegment klasy szablonowej TSharedMem

W implementacji klasy szablonowej TSharedMem zamieszczono także metodę korzystającą z komunikacji grupowej. Jest nią metoda AllReduceMax implementująca operację redukcji danych udostępnionych jako ich parametr, przy pomocy znajdowania ich wartości maksymalnej. Metoda ta stanowi obudowę dla funkcji gaspi_allreduce [5].

Metodą, która powinna być wykonywana celem zakończenia procesu komunikacji jest EndSharedMem. Realizuje ona zadanie usuwania utworzonych wcześniej segmentów pamięci oraz grup. Wywołuje w tym celu funkcje gaspi_segment_delete oraz gaspi_group_delete [5]. Nie zaimplementowano jej w formie destruktora, celem zachowania pełnej kontroli nad momentem jej wywołania.

7. Środowisko uruchomieniowe, dane wejściowe

W celu realizacji badań wykorzystano platformę Google Compute Engine udostępniającą maszyny wirtualne dysponujące od 1 do 32 procesorów wirtualnych, nazywanych vCPU, z maksymalną ilością pamięci do 6,5GB dla jednego vCPU [2]. Wykorzystano predefiniowane typy maszyn oznaczone jako: n1-standard-2 oraz n1-standard-4. Oferują one odpowiednio maszynę z dwoma vCPU i 7,5GB RAM oraz maszynę z czterema vCPU i 15GB RAM. Zgodnie z dokumentacją Google Cloud Platform seria maszyn wirtualnych n1 jest zaimplementowana jako jeden fizyczny procesor z technologią hyper–thread [34]. Maszyny te mogą być uruchomione z wykorzystaniem procesów: 2,6 GHz Intel Xeon E5, 2,5 GHz Intel Xeon E5 v2 lub 2,3 GHz Intel Xeon E5 v3, przy czym użytkownik nie ma możliwości zdecydowania, który z wymienionych procesorów zostanie użyty [34].

Badania zostały przeprowadzone na 2, 4, 6 opisanych powyżej typach maszyn wirtualnych. Wszystkie maszyny były połączone ze sobą siecią komputerową wykorzystującą protokół TCP/IP, przy czym brak informacji co do szczegółów technicznych tej sieci.

Oprócz dysków o pojemności 10GB, zawierających dane systemów operacyjnych, wykorzystywano także przestrzeń przechowywania danych nazywaną Cloud Storage Buckets, której typ określono jako Standard Storage [35]. Przestrzeń ta była dostępna dla wszystkich maszyn.

Na każdej maszynie został zainstalowany system operacyjny Debian Wheezy 7.9 oraz Hadoop 2.4.1 z wykorzystaniem obrazu dostępnego na platformie Google Cloud. Instalacja środowiska Hadoop była niezbędna celem umożliwienia pracy środowisku Spark wykorzystującego element składowy Hadoop – menadżer YARN. Podczas procesu konfiguracji określono także, która z maszyn pełnić będzie rolę węzła nadzorującego. Po zakończeniu procesu instalacji maszyn wirtualnych, zainstalowano środowisko Spark w wersji 1.4.0 przygotowane do współpracy z Hadoop 2.4 [36].

W trakcie procesu konfiguracji maszyn, na Cloud Storage Buckets został zainstalowany rozproszony system plików HDFS, dzięki któremu możliwe było udostępnienie przechowywania danych wejściowych oraz wyników obliczeń, dla wszystkich węzłów roboczych podczas korzystania ze środowiska Spark [17].

Na tak skonfigurowanym środowisku zainstalowano dodatkowo środowisko kompilatora GCC 4.92 przygotowanego dla systemu Debian. Zostało ono wykorzystane do kompilacji biblioteki udostępniającej GASPI, a później także samego programu implementującego algorytm ID3 [37].

Środowisko GASPI nie miało możliwości dostępu do przestrzeni Cloud Storage Buckets. Zdecydowano się więc na zainstalowanie serwera protokołu NFS, którego zadaniem było udostępnienie jednego z katalogów, systemu plików węzła nadzorującego przechowującego dane wejściowe i wyniki obliczeń [38]. Serwer protokołu NFS działał tylko w trakcie pracy środowiska GASPI, na czas korzystania ze środowiska Spark był wyłączany.

Kompilacji programów dla środowiska Spark dokonano z użyciem narzędzia Apache Maven [39]. Generowanie kodu wynikowego dla środowiska GASPI odbywa się z załączoną opcją –O wymuszającą pełną optymalizację przez kompilator [40], załączona jest także opcja -std=c++11 wymuszająca wykorzystanie standardu ISO/IEC 14882-2011, potocznie nazywanego C++11 [41].

Jako danych wejściowych zawierających zbiór krotek treningowych użyto pliku tekstowego zawierającego 9000 wierszy, przy czym każdy z wierszy zawierał 10

kolumn danych. Pierwsza kolumna była kolumną zawierającą oznaczenia numerów kolejnych linii, kolejne zawierały wartości atrybutów opisowych i decyzyjnego.

Dane wejściowe zostały wygenerowane, w drodze wielokrotnego powtórzenia 20 krotek treningowych, co do których istniała pewność wygenerowania drzewa decyzji zawierającego więcej niż jeden poziom węzłów decyzyjnych.

Obie implementacje algorytmu ID3 uruchomiono dla:

- 900, 4200 i 9000 wierszy krotek danych wejściowych,
- 3,5 i 8 atrybutów opisowych.

Dało to 9 kombinacji krotek treningowych. Każda kombinacja była uruchomiona dla następujących konfiguracji: 2, 4 oraz 6 węzłach. Liczba vCPU zależała od uruchomienia na maszynach zawierających 2 lub 4 vCPU. Otrzymano więc siatkę pomiarową zawierającą 27 punktów pomiarowych.

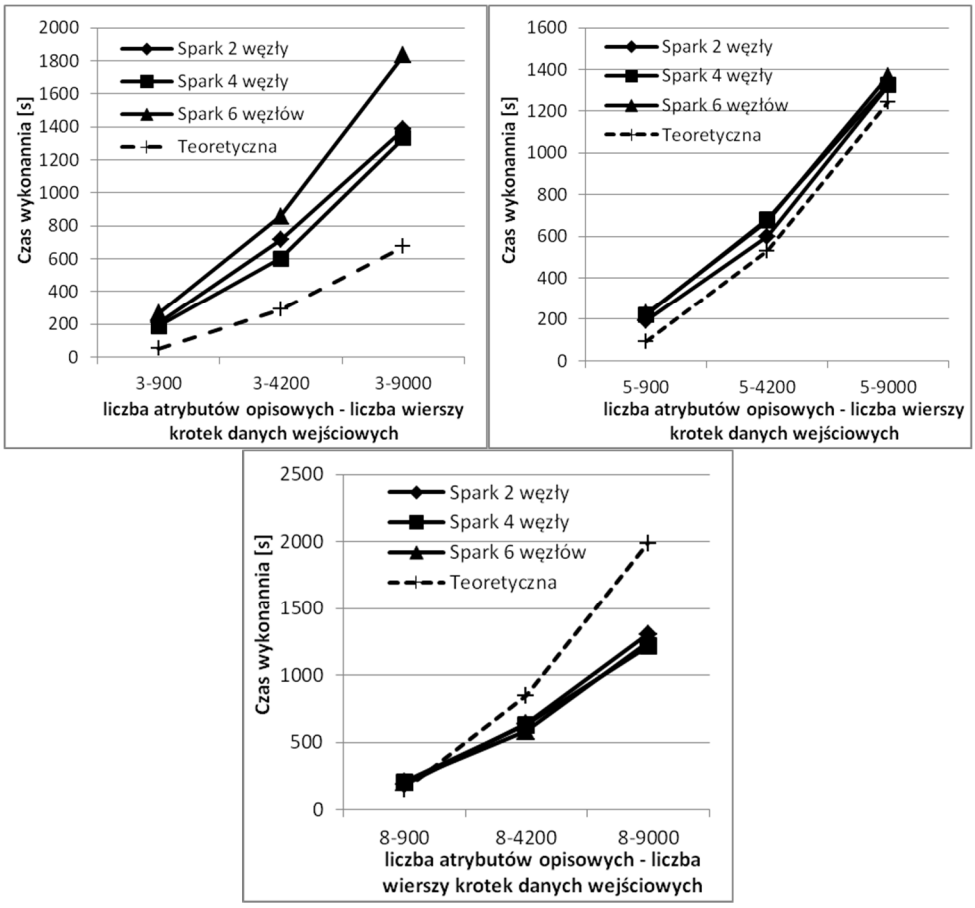
Należy tu zauważyć, że badaniu podlegał ten sam algorytm zaimplementowany w dwu różnych środowiskach programistycznych i przetwarzający te same dane wejściowe. W związku z powyższym przyjęto, że otrzymywane w wyniku działania programów drzewa decyzji, powinny być identyczne z drzewami otrzymanymi dla tych samych danych przetwarzanych w arkuszu kalkulacyjnym. Pomiary czasu wykonania programów, dla których otrzymano drzewa decyzyjne odmienne od tych z arkusza kalkulacyjnego były odrzucane.

Ze względu na uruchomianie programu w środowisku wielozadaniowym i wieloprocesorowym dla każdego punktu pomiarowego przeprowadzono 5 pomiarów czasów wykonania programu i jego części [42].

8. Wydajność i produktywność programistyczna algorytmu ID3 zaimplementowanego w środowisku Spark

Podczas wykonywania programu w środowisku Spark zapisywano tylko czasy wykonania pojedynczej iteracji drivera. Spowodowane to było znacznym spowolnieniem programu zapisującego większą ilość danych, dodatkowo wiele z tych danych jest dostępne z poziomu Spark Web UI.

Na rysunku 12 zamieszczono wykresy prezentujące czasy wykonania algorytmu ID3 zaimplementowanego w środowisku Spark. Analizując je można zauważyć, że wraz z zwiększaniem się rozmiaru zadania skraca się czas realizacji zdania, a zmiany ilości węzłów nie wpływają zbyt silnie na czas realizacji programu.



Rysunek 12. Czasy wykonania algorytmu ID3 w środowisku Spark dla różnych danych wejściowych

Celem optymalizacji implementacji algorytmu w środowisku Spark dokonano analizy danych udostępnianych przez wspomniany wcześniej serwis Spark Web UI.

Cennych informacji dostarczyła część serwisu opisująca bezpośrednio węzły obliczeniowe zaprezentowana na rysunku 13. Widać, że w obliczeniach uczestniczyły tylko dwa węzły robocze (ang. *executors*) oraz driver mimo, że system miał uruchomionych ich 6. System nie korzystał także z możliwości buforowania przekształceń niektórych RDD.

Executor ID	Address	RDD Blocks	Memory Used	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write
1	hadoop-w-cccc.c.decisiontree-966.internal:55836	0	0.0 B / 530.3 MB	0.0 B	0	0	198	198	27.2 s	15.6 MB	6.4 KB	24.8 KB
2	hadoop-w-ssun.c.decisiontree-966.internal:60104	0	0.0 B / 530.3 MB	0.0 B	1	0	160	161	19.4 s	11.6 MB	8.9 KB	15.4 KB
driver	10.240.40.27:40068	0	0.0 B / 265.4 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B

Rysunek 13. Wykaz informacji o węzłach roboczych realizujących obliczenia

Następną ważną wskazówkę dotyczącą możliwości optymalizacji można znaleźć na rysunku 14 prezentującym końcową część wykonywanego programu. Widać, że zdominowały ją zdania polegające na sprawdzaniu czy RDD jest pusty. Uwzględniając przepływ danych i sterowania w driverze środowiska Spark, zaprezentowane na rysunku 5, stwierdzić można, że działania te realizowane są w ramach procesu 2, który filtrujące dane z wykorzystaniem informacji zawartych w obiekcie Control.

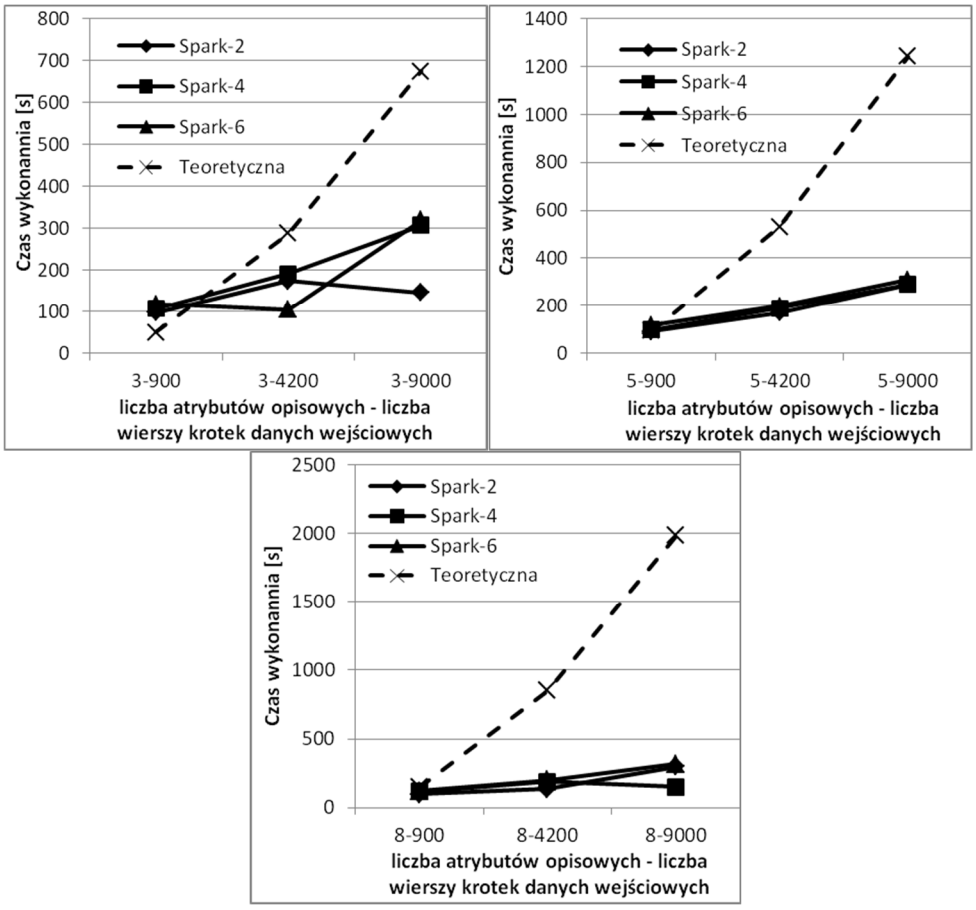
Jak już wspomniano, proces ten jest kosztowny ze względu na możliwość przechowywania elementów RDD na różnych węzłach roboczych. Potwierdza to także analiza czasów trwania poszczególnych testów, które przeciętnie trwają 0,1 s.

Działania optymalizacyjne przeprowadzono z wykorzystaniem pracy [43] i można je podzielić na dwie części: konfiguracyjną i zmianę kodu.

Część konfiguracyjna została przeprowadzona zgodnie z wnioskami zawartymi w pracy [43] i polegała na zmianach ilości: dostępnej pamięci, węzłów roboczych angażowanych do realizacji obliczeń oraz ilości wątków uruchamianych na poszczególnych węzłach. Zmian tych dokonano w pliku konfiguracyjnym środowiska Spark [44]. Zgodnie z pracą [43] ilość dostępnej pamięci została ustalona tak, aby prawie maksymalnie wykorzystać zasoby dostępne dla środowiska uruchomieniowego Spark. Liczbę wątków ustalono na dwa. Liczba węzłów, na których uruchamiane było zadanie, była ustawiana na 2, 4 i 6.

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total
847	isEmpty at DT.java:238	2016/06/26 21:57:04	96 ms	1/1
846	isEmpty at DT.java:238	2016/06/26 21:57:04	0.1 s	1/1
845	isEmpty at DT.java:238	2016/06/26 21:57:04	98 ms	1/1
844	isEmpty at DT.java:238	2016/06/26 21:57:03	95 ms	1/1

Rysunek 14. Wykaz realizowanych prac pod koniec programu



Rysunek 15. Czasy wykonania algorytmu ID3 w środowisku Spark po optymalizacji

Część dotycząca zmiany kodu poległa na jawnym określeniu buforowania w pamięci wyników przekształceń niektórych magazynów danych. Wykorzystano w tym celu metodę persist ze wskazaniem poziomu MEMORY_LEVEL [45].

Celem wskazania magazynów, których przekształcenia będą buforowane w pamięci wzięto pod uwagę:

- wcześniejsze uwagi dotyczące częstotliwości sprawdzania, czy określony RDD jest pusty,
- częstość wykorzystywania poszczególnych RDD określoną na podstawie rysunku 5.

W wyniku analizy zdecydowano się na zastosowanie buforowania wyników przekształceń dla magazynów: lines i word.

Po tak dokonanych zmianach ponownie przeprowadzono badanie czasu wykonania programu. Wyniki tego badania zamieszczono na rysunku 15. Widać, że czas trwania znacząco się skrócił. Analizując dane dostępne w serwisie Spark Web UI stwierdzić można, że jest to efektem buforowania w pamięci niektórych RDD.

Powoduje to skrócenie między innymi czasu sprawdzania czy są one puste, co zostało zaprezentowane na rysunku 16. Po optymalizacji czas trwania sprawdzenia to przeciętnie tylko 0,0018 s.

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total
1000	isEmpty at DT.java:243	2016/07/03 15:55:12	13 ms	1/1
999	isEmpty at DT.java:238	2016/07/03 15:55:12	16 ms	1/1
998	isEmpty at DT.java:238	2016/07/03 15:55:12	20 ms	1/1
997	isEmpty at DT.java:238	2016/07/03 15:55:12	18 ms	1/1
996	isEmpty at DT.java:238	2016/07/03 15:55:12	17 ms	1/1
995	isEmpty at DT.java:238	2016/07/03 15:55:12	20 ms	1/1
994	isEmpty at DT.java:238	2016/07/03 15:55:12	20 ms	1/1
993	isEmpty at DT.java:238	2016/07/03 15:55:12	19 ms	1/1
992	isEmpty at DT.java:238	2016/07/03 15:55:12	19 ms	1/1
991	isEmpty at DT.java:238	2016/07/03 15:55:12	19 ms	1/1
990	isEmpty at DT.java:238	2016/07/03 15:55:12	23 ms	1/1
989	isEmpty at DT.java:238	2016/07/03 15:55:12	20 ms	1/1
988	isEmpty at DT.java:238	2016/07/03 15:55:12	20 ms	1/1
987	isEmpty at DT.java:238	2016/07/03 15:55:12	19 ms	1/1

Rysunek 16. Wykaz realizowanych prac pod koniec programu – po wykonaniu optymalizacji

W tabeli 1 przedstawiono średnie czasy wykonywania implementacji algorytmu ID3 w środowisku Spark, dla programu przed i po optymalizacji, dla każdej kombinacji danych wejściowych przetwarzanych na różnej liczbie węzłów.

Porównując czasy wykonania obu wersji implementacji, można stwierdzić wielokrotne zmniejszenie się czasu wykonania programu zoptymalizowanego. Kształtuje się ono od 1,6 raza dla 8 atrybutów opisowych i 900 krotek, do 9 razy dla 3 atrybutów opisowych i 9000 krotek.

Innym wnioskiem, który można wysnuć na podstawie analizy danych zawartych w tabeli 1 jest także zwiększanie się czasów wykonywania programu wraz z wzrostem liczby węzłów, na których program jest wykonywany. Wyjątkami od tej reguły są realizacje programu na 4 węzłach dla niektórych kombinacji danych wejściowych, dla obu wersji programu. Oznacza to, że w danym środowisku uruchomieniowym

algorytm ten nie skaluje się wraz ze wzrostem liczby węzłów, a przyczyną tego jest wysoki koszt komunikacji.

Tabela 1. Czasy wykonywania implementacji algorytmu ID3 w środowisku Spark przed i po optymalizacji dla różnych kombinacji danych wejściowych i różnej ilości węzłów – węzły z 2 vCPU

Liczba:		Spark przed optymalizacją			Spark po optymalizacji		
atrybutów opisowych	**wierszy krotek**	**2 węzły**	**4 węzły**	**6 węzłów**	**2 węzły**	**4 węzły**	**6 węzłów**
3	**900**	206,24	187,49	266,49	97,66	108,24	118,42
	4200	713,81	601,45	856,54	174,86	191,91	105,85
	9000	1385,73	1331,74	1831,97	146,66	307,53	319,92
5	**900**	194,92	222,86	228,01	90,86	97,70	116,71
	4200	599,92	683,01	675,17	171,06	189,06	198,56
	9000	1320,11	1326,57	1366,62	289,51	288,94	308,37
8	**900**	185,49	208,19	205,04	97,41	116,88	122,12
	4200	638,42	633,84	587,57	136,56	188,92	198,98
	9000	1307,20	1216,47	1248,11	299,26	151,02	315,63

Zgodnie bowiem z wnioskami z pracy [15], badana implementacja algorytmu ID3 w środowisku Spark jest zbliżona do podejścia nazywanego synchroniczną konstrukcją drzewa (ang. *synchronous tree construction*), którego wadą jest wysoki koszt komunikacji.

W tabeli 2 przedstawiono średnie czasy wykonywania implementacji algorytmu ID3 w środowisku Spark, dla programu po optymalizacji uruchomionego na węzłach wyposażonych w 4 vCPU. Zaprezentowano czasy wykonywania dla każdej kombinacji danych wejściowych przetwarzanych na różnej liczbie węzłów.

Czasy wykonania są generalnie tylko nieznacznie krótsze od czasów uzyskanych na maszynach wyposażonych w 2 vCPU. Są jednak przypadki gdy czasy wykonania na 4 vCPU wydłużają się, nawet dwukrotnie, względem tych na 2 vCPU. Na przykład ma to miejsce dla następującej kombinacji danych wejściowych i konfiguracji: 3 atrybuty opisowe, 4200 wierszy krotek, 6 węzłów.

Dokonując analizy czasów wykonania programu dla wskazanych powyżej kombinacji danych wejściowych i konfiguracji, stwierdzić można ogólne wydłużenie

czasu wykonywania wszystkich prac (ang. jobs) oraz etapów (ang. stages) w programie. Uważamy, że mogło to mieć związek z okresowym zwiększeniem obciążenia chmury obliczeniowej.

Tabela 2. Czasy wykonywania implementacji algorytmu ID3 w środowisku SPARK dla różnych kombinacji danych wejściowych i różnej liczby węzłów – węzły z 4 vCPU

Liczba:		**Spark po optymalizacji**		
atrybutów opisowych	**wierszy krotek**	**2 węzły**	**4 węzły**	**6 węzłów**
3	**900**	97,25	106,68	130,05
	4200	170,49	187,52	207,29
	9000	220,14	286,69	316,60
5	**900**	90,66	104,47	117,64
	4200	164,78	180,83	192,48
	9000	264,61	285,45	281,82
8	**900**	92,37	108,28	119,73
	4200	165,23	191,10	197,59
	9000	271,16	284,88	301,53

Analizując kwestię produktywności programistycznej środowiska Spark, z wykorzystaniem definicji produktywności wskazanej we wstępie do artykułu, należy zwrócić uwagę na wielkość kodu oraz stopień jego skomplikowania. Kod modułu drivera dla środowiska Spark składał się 678 linii zapisanych w języku Java. Stopień jego złożoności w dużej mierze odzwierciedla rysunek 5 obrazujący model przekształceń RDD prowadzący do realizacji algorytmu ID3.

Należy tu nadmienić, że w celu przygotowania tego modelu należało opracować nowy sposób opisu uwzględniający specyfikę środowiska Spark. Jednak wysiłek ten opłacił się, gdyż przygotowany model w dużej mierze ułatwił dokonanie optymalizacji zwiększającej wydajność implementacji.

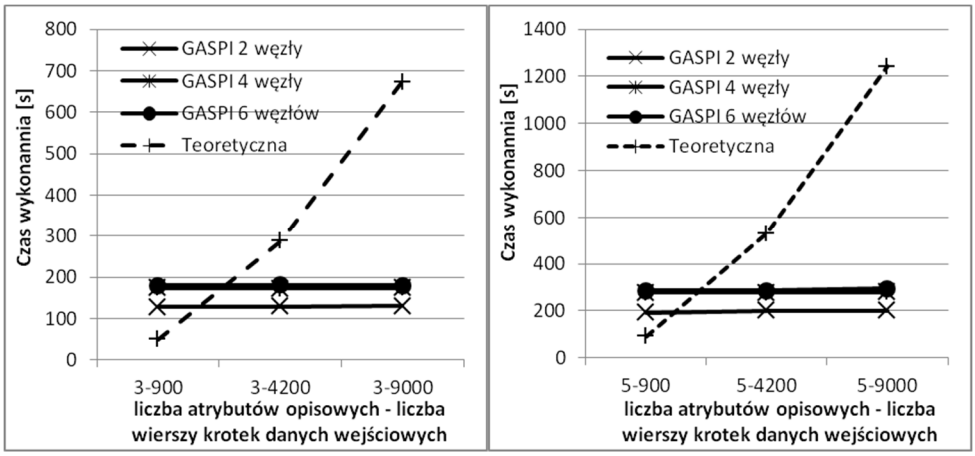
Kolejnym faktem, który ma znaczenie przy ocenie produktywności programistycznej jest to, że optymalizacja sprowadziła się do dodania łącznie trzech linii kodu i kilku zmian w konfiguracji środowiska programistycznego. Należy tu jednak pamiętać, że programista nie ma możliwości wpływu na sposób zrównoleglenia napisanego przez siebie drivera. Prowadzi nas to do stwierdzenia, że środowisko Spark charakteryzuje się dużą produktywnością programistyczną, choć ogranicza sposób implementacji algorytmu generowania drzew decyzyjnych.

9. Wydajność i produktywność programistyczna algorytmu ID3 zaimplementowanego w środowisku GASPI

Podczas wykonywania programu środowisku GASPI zapisywano między innymi czasy wykonania następujących części:

- operacji wczytywania krotek danych wejściowych z pliku wejściowego,
- pojedynczej iteracji implementacji algorytmu ID3 w środowisku GASPI, a w tym:
  - obliczeń związanych z wyznaczeniem maksymalnego zysku z podziału w aktualnie analizowanym liściu drzewa,
  - przesyłania danych pomiędzy procesami,
  - operacji zapisu wygenerowanego drzewa decyzji,
  - całego programu.

Na rysunku 17 zamieszczono wykresy prezentujące czasy wykonania algorytmu ID3 zaimplementowanego w środowisku GASPI. Na wszystkich wykresach zamieszczony został także wykres obrazujący teoretyczną złożoność tworzenia drzew decyzyjnych wyznaczony dla formuły (4). Pominięto wykres obrazujący czasy wykonania dla 8 atrybutów opisowych, którego przebieg jest analogiczny do wykresów zamieszczonych na rysunku 17.



Rysunek 17. Czasy wykonania algorytmu ID3 w środowisku GASPI

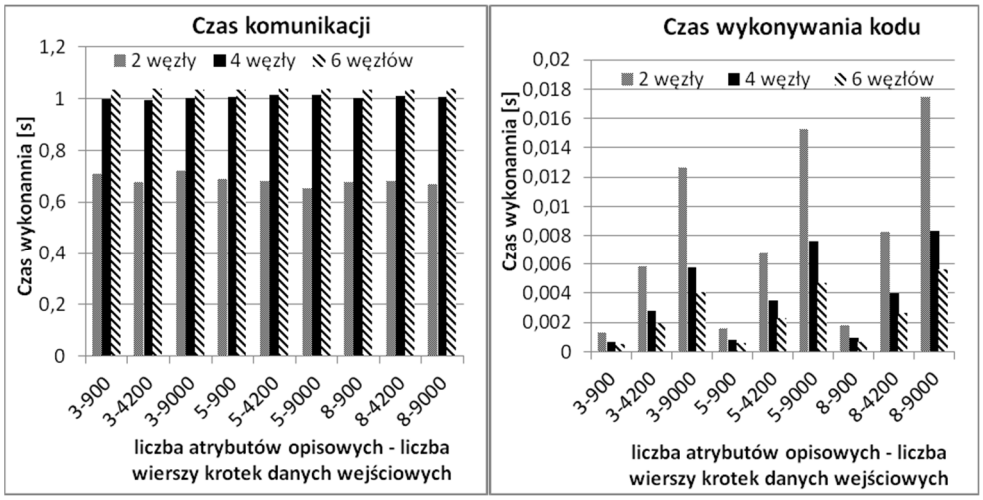
Jak już wcześniej wspomniano, czasy wykonywania programu w środowisku GASPI zwiększają się przy wzroście ilości węzłów wykorzystywanych do obliczeń oraz przy zwiększaniu ilości atrybutów opisowych. Są jednak prawie niezależne od

zmiany liczby wierszy krotek danych wejściowych. Cechą charakterystyczną jest natomiast dość mała zmiana czasu wykonania w odniesieniu do zmiany liczby wierszy krotek danych wejściowych.

Celem ustalenia przyczyny takich zmian czasu wykonania przeprowadzono analizę czasów wykonywania kodu oraz czasów komunikacji w metodzie CountTheFileValues, której tekst zaprezentowano na rysunku 9.

Wybrano do analizy tę właśnie metodę, gdyż zawiera ona wywołania dużej ilości metod komunikacyjnych oraz wywołanie metody, która iteracyjnie przegląda krotki danych wejściowych celem ich zliczenia. Jest to więc przykład zadania mieszanego: obliczeniowego o dość dużej złożoności i komunikacyjnego.

Wyniki analizy prezentuje rysunek 18.



Rysunek 18. Czas wykonywania obliczeń i komunikacji w metodzie CountTheFileValues

Wynikowe, średnie czasy wykonywania programu zaprezentowano w tabeli 3 dla każdej kombinacji danych wejściowych przetwarzanych na różnej liczbie węzłów.

Widać, że przeciętny czas komunikacji jest wielokrotnie większy od czasu wykonywania obliczeń, co więcej, inaczej niż czas wykonywania kodu, wzrasta on podczas zwiększania liczby węzłów realizujących zadanie. Bazując na wnioskach z pracy [15] uważamy, że jest to efektem sposobu implementacji algorytmu ID3 oraz właściwości wykorzystanej chmury obliczeniowej. Podobnie jak w środowisku Spark, tak i tu zastosowano podejście zbliżone do podejścia nazywanego synchroniczną konstrukcją drzewa (ang. *synchronous tree construction*), którego wadą jest

wysoki koszt komunikacji. Problem ten nie został zniwelowany mimo zastosowania komunikacji jednostronnej.

Tabela 3. Czasy wykonywania implementacji algorytmu ID3 w środowisku GASPI dla różnych kombinacji danych wejściowych i różnej liczby węzłów – węzły z 2 vCPU

Liczba:		**GASPI**		
atrybutów opisowych	**wierszy krotek**	**2 węzły**	**4 węzły**	**6 węzłów**
3	**900**	127,42	173,42	180,03
	4200	129,06	173,10	180,72
	9000	130,00	173,67	180,65
5	**900**	193,35	276,77	285,86
	4200	200,70	278,54	286,21
	9000	202,02	279,51	295,77
8	**900**	295,62	408,19	415,77
	4200	299,31	408,98	416,62
	9000	306,88	411,59	418,07

Przeprowadzono także próbę pomiaru czasów wykonania implementacji algorytmu ID3 na 2, 4 i 6 węzłach wyposażonych w 4 vCPU. Skończyły się one w większości wypadków błędem wykonania funkcji `gaspi_allreduce`. Bezbłędne uruchomienia programu uzyskiwano podczas pracy na 2 węzłach, jednak program generował niekompletne i niepoprawne wyniki.

Analizując opisywaną implementację z punktu widzenia jej produktywności programistycznej, zgodnie z definicją przyjętą we wstępie do niniejszego artykułu, należy zauważyć, że kod programu dla środowiska GASPI to 1380 linii języka C++.

Dokonując oceny złożoności tworzonego kodu, należy mieć na uwadze, iż środowisko GASPI jest zaimplementowane jako biblioteka uzupełniająca język C++ o interfejs modelu PGAS, co wymaga opracowania odpowiedniego zestawu klas obudowujących i ułatwiających jego wykorzystanie. Powoduje to wzrost stopnia komplikacji implementacji algorytmu ID3. Wpływa to więc negatywnie na produktywność programistyczną tego środowiska, w szczególności wobec konieczności dokonania zmian wynikających ze zmiany podejścia do implementacji. Jednak zasadniczą zaletą tego środowiska jest jego elastyczność, umożliwiająca dowolną implementację wybranego algorytmu analizującego duże zbiory danych. W szczególności, można zaimplementować podejście hybrydowe opisane w pracy [15], pozwalające na ominięcie wąskiego gardła związanego z komunikacją.

10. Porównanie wydajności algorytmu ID3 zaimplementowanego w środowiskach Spark i GASPI

W tabeli 4 zaprezentowano łącznie czasy wykonania implementacji algorytmu ID3 dla obu środowisk na maszynach z 2 vCPU, przy czym dla środowiska Spark podano czasy wykonania po optymalizacji.

Tabela 4. Czasy wykonywania implementacji algorytmu ID3 w środowiskach GASPI i Spark (po optymalizacji) dla różnych kombinacji danych wejściowych i różnych liczb węzłów – węzły z 2 vCPU

Liczba:		**Spark po optymalizacji**			**GASPI**		
atrybutów opisowych	**wierszy krotek**	**2 węzły**	**4 węzły**	**6 węzłów**	**2 węzły**	**4 węzły**	**6 węzłów**
3	**900**	97,66	108,24	118,42	127,42	173,42	180,03
	4200	174,86	191,91	105,85	129,06	173,10	180,72
	9000	146,66	307,53	319,92	130,00	173,67	180,65
5	**900**	90,86	97,70	116,71	193,35	276,77	285,86
	4200	171,06	189,06	198,56	200,70	278,54	286,21
	9000	289,51	288,94	308,37	202,02	279,51	295,77
8	**900**	97,41	116,88	122,12	295,62	408,19	415,77
	4200	136,56	188,92	198,98	299,31	408,98	416,62
	9000	299,26	151,02	315,63	306,88	411,59	418,07

Porównując czasy wykonania obu implementacji stwierdzić, można iż mimo podobnego podejścia do implementacji algorytmu ID3, środowisko Spark było nieznacznie szybsze.

Uważamy, że było to związane ze sposobem zrównoleglenia ciągu przekształceń RDD będących w istocie rzeczy zbiorami rozproszonymi ulokowanymi na maszynach realizujących na nich obliczenia. W efekcie, mimo że implementacja algorytmu ID3 w środowisku Spark bazowała na podejściu znanym jako synchroniczna konstrukcją drzewa (ang. *synchronous tree construction*), to na skutek sposobu przekształceń RDD cały program realizowany był w sposób zbliżony do podejścia hybrydowego opisanego w pracy [15]. Pozwoliło to na częściowe ominięcie wąskiego gardła związanego z komunikacją.

Warto jednak zwrócić uwagę, że opisana powyżej sposób realizacji nie zawsze był skuteczny. Jest to widoczne przy wykonywaniu programu dla kombinacji danych

zawierających 9000 wierszy krotek na 6 węzłach. W tych bowiem przypadkach skuteczniejsza okazywał się jednak sposób wykorzystywany przez GASPI.

11. Wnioski i plan dalszych badań

W pracy zbadano wydajność i produktywność programistyczną wykorzystania chmur obliczeniowych oraz dwu środowisk programistycznych, a mianowicie SPARK i GASPI, do równoległej implementacji algorytmów eksplorujących duże zbiory danych na przykładzie algorytmu ID3 generowania drzew decyzyjnych. Wszystkie implementacje zostały uruchomione na platformie Google Compute Engine.

Po analizie uzyskanych czasów wykonania programu w środowisku Spark oraz danych dostarczonych przez serwis Spark Web UI stwierdzono, że możliwa jest optymalizacja tej implementacji. Polegała ona głównie na jawnym określeniu buforowania w pamięci wyników przekształceń niektórych magazynów danych. Dodatkowo optymalizowano także konfigurację uruchomieniową środowiska Spark. W trakcie optymalizacji bardzo przydatny okazał się diagram prezentujący sposób przekształcania RDD, stworzony z wykorzystaniem diagramu czynności języka UML oraz diagramu przepływu danych DFD, zaprezentowany na rysunku 5.

W środowisku GASPI stwierdzono, iż czasy komunikacji stanowią ważną składową czasu realizacji programu. Wynikało to zastosowania podejścia znanego jako synchroniczna konstrukcja drzewa (ang. *synchronous tree construction*) opisanego w pracy [15]. Znanym problemem tego podejścia jest duży koszt komunikacji. W środowisku GASPI objawiał się on podczas realizacji procesu synchronizacji asynchronicznej realizacji funkcji `gaspi_write`. Był on także przyczyną braku poprawnych wyników przy uruchomieniu kodu na węzłach wyposażonych w 4 vCPU. Obserwacja ta oraz wnioski z pracy [15] sugerują, że skuteczniejszą metodą realizacji implementacji algorytmu ID3 jest podejście hybrydowe.

Oprócz wydajności przetwarzania, w pracy starano się uwzględnić kwestię produktywności obu środowisk. Kod drivera dla środowiska Spark składał się mniejszej liczby linii kodu niż program dla środowiska GASPI. Widać więc, nawet nie uwzględniając stopnia złożoności obu środowisk, że większy nakład pracy konieczny był przy tworzeniu programu dla środowiska GASPI. Jednak to właśnie środowisko oferuje największą elastyczność pracy dającą możliwość dowolnej implementacji programu. Jest to jednak jednocześnie zaleta ale i wada. W przypadku bowiem konieczności zmiany podejścia do implementacji należy zmienić dużą ilość kodu, co wiąże się z dużym nakładem pracy. Ocena ta jednak powinna zostać uszczegółowiona z wykorzystaniem różnorodnych metryk opisujących tworzony kod. Sądzimy rów-

nież, że pod uwagę należy wziąć metodyki zwinne tworzenia kodu, które uzupełnione będą o wymaganie tworzenia kodu wydajnego. Uważamy bowiem, że iteracyjność tworzenia kodu wykorzystywana w tych metodykach, pozwoli w na wczesnym etapie zidentyfikować problem małej wydajności tworzonego kodu, co będzie miało pozytywny wpływ na produktywność programistyczną, w szczególności dla środowiska GASPI.

Analizując uzyskane czasy pracy programu stworzonego z wykorzystaniem środowiska GASPI stwierdzić można, iż platforma Google Compute Engine, będąca publiczną chmurą obliczeniową nie umożliwia pełnego wykorzystania możliwości tego środowiska. Wynika to z konieczności synchronizacji asynchronicznej realizacji komunikacji jednostronnej, która to synchronizacja jest wrażliwa na wszelkie opóźnienia wynikające z realizacji zdań innych użytkowników w publicznej chmurze obliczeniowej.

Dalsze badania powinny być więc realizowane zarówno w zamkniętych środowiskach gridowych, dających możliwość pełnej kontroli realizowanych zadań, jak i w różnorodnych środowiskach chmur obliczeniowych. W trakcie badań należy skorzystać zarówno, z opisanego w niniejszym artykule podejścia implementacyjnego, jak i wykorzystać wskazane już podejście hybrydowe. Taki sposób realizacji badań umożliwi ocenę wpływu różnorodnych właściwości chmur na wydajność wykonywanych programów. Tym sposobem możliwa będzie ocena przydatności chmur obliczeniowych i gridów do eksploracji dużych zbiorów danych.

Bibliografia

[1] Armbrust M. at el., *A view of cloud computing*, "Communications of the ACM", Vol. 53, No. 4

[2] *Compute Engine – IaaS | Google Cloud Platform*, https://cloud.google.com/compute/ [18.062016]

[3] Zaharia M.A., *An Architecture for and Fast and General Data Processing on Large Clusters*, Technical Report No. UCB/EECS-2014-12, Electrical Engineering and Computer Sciences University of California at Berkeley, 2014, http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-12.html

[4] Almasi G., *PGAS (Partitioned Global Address Space) Languages*, w: *Encyclopedia of Parallel Computing*, (ed.) Padua D., Springer, Ney York, 2011

[5] *GASPI: Global Address Space Programming Interface, Specification of a PGAS API for communication*, https://raw.githubusercontent.com/GASPI-Forum/GASPI-Forum.github.io/master/standards/GASPI-16.1.pdf

[6] McAfee A., Brynjolfsson E., Davenport T.H., Patil D.J., Barton D., *Big data. The management revolution*, "Harvard Bus Reviev" 2012, Vol. 90, No. 10
[7] Han J., Kamber M., Pei J., *Data mining: concepts and techniques*, Elsevier, Burlington, 2011
[8] Larose D.T., *Discovering knowledge in data: an introduction to data mining*. John Wiley & Sons, Hoboken, 2014
[9] Hand D.J., Mannila H., Smyth P., *Principles of data mining*, MIT Press, Cambridge, 2001
[10] Kufrin R., *Decision trees on parallel processors*, "Machine Intelligence and Pattern Recognition" 1997, Vol. 20
[11] Yıldız O.T., Dikmen O., *Parallel univariate decision trees*, "Pattern Recognition Letters" 2007, Vol. 28, No. 7
[12] Müller W., Wysotzki F., T*he decision-tree algorithm CAL5 based on a statistical approach to its splitting algorithm*, w: *Machine learning and statistics. The interface*, (eds.) Nakhaeizadeh G. Taylor C.C., Wiley, New York, 1997
[13] Quinlan J.R., *Induction of decision trees*, "Machine learning" 1986, Vol. 1, No. 1
[14] Richards J.T., Brezin J., Swart C.B., Halverson C.A., *Productivity in parallel programming: A decade of progress*, "Queue" 2014, Vol. 12, No. 9
[15] Srivastava A., Han E.-H., Kumar V., Singh V., *Parallel formulations of decision-tree classification algorithms*, w: *High Performance Data Mining*, Springer, Berlin–Heidelberg,1999
[16] Dean J., Ghemawat S., *MapReduce: simplified data processing on large clusters*, "Communications of the ACM" 2008, Vol. 51, No. 1
[17] Borthakur D., *HDFS architecture guide*, HADOOP APACHE PROJECT, 2008, http://hadoop. apache. org/common/docs/current/hdfs design. pdf,
[18] Vavilapalli V. K. at el., *Apache hadoop yarn: Yet another resource negotiator*, Proceedings of the 4th annual Symposium on Cloud Computing, 2013
[19] Landset S., Khoshgoftaar T.M., Richter A.N., Hasanin T., *A survey of open source tools for machine learning with big data in the Hadoop ecosystem*, "Journal of Big Data" 2015, Vol. 2, No. 1
[20] Zaharia M. at el., *Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing*, Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, 2012
[21] Zaharia M., Chowdhury M., Franklin M.J., Shenker S., Stoica I., Spark: *Cluster Computing with Working Sets*, "HotCloud" 2010, Vol. 10
[22] Karau H., Konwinski A., Wendell P., M. Zaharia, *Learning Spark: Lightning-Fast Big Data Analysis*, O'Reilly Media, Sebastopol, 2015

[23] Karau H., *Fast Data Processing With Spark*, Packt Publishing, Birmingham, 2013
[24] Davidson A., *A Deeper Understanding of Spark's Internals*, https://spark-summit.org/wp-content/uploads/2014/07/A-Deeper-Understanding-of-Spark-Internals-Aaron-Davidson.pdf [09.03.2016]
[25] Beena T.L.A, Amalarethinam D.G., *Analysis of Task Scheduling Algorithm in Fine-Grained and Course-Grained Dags in Cloud Environment*, "Intern. J. Fuzzy Mathematical Archive" 2015, Vol. 6, No. 2
[26] Yelick K. at el., *Productivity and performance using partitioned global address space languages*, Proceedings of the 2007 international workshop on Parallel symbolic computation, 2007
[27] *Forum of the PGAS-API GASPI*, http://www.gaspi.de/gaspi/ [29.032016]
[28] *Documentation of GPI-2. Implement the GASPI specification,* http://www.gpi-site.com/gpi2/docs/ [02.04.2016]
[29] *MPI: A Message-Passing Interface Standard*, Version 2.2, Message Passing Interface Forum, September 4, 2009, http://mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf
[30] *UML 2.5*, http://www.omg.org/spec/UML/2.5/PDF/ [17.02.2016]
[31] Yourdon E., *Modern structured analysis*, Vol. 191, Yourdon Press Englewood Cliffs, New York, 1989
[32] Snir M., *MPI--the Complete Reference: The MPI core*, Vol. 1. MIT Press, Cambridge, 1998
[33] *Tutorial GPI-2*, http://www.gpi-site.com/gpi2/tutorial/ [30.04.2016]
[34] *Machine Types | Compute Engine | Google Cloud Platform*, https://cloud.google.com/compute/docs/machine-types [1806.2016]
[35] *Storage Classes | Cloud Storage | Google Cloud Platform*, https://cloud.google.com/storage/docs/storage-classes [18.06.2016]
[36] *Downloads|Apache Spark*, http://spark.apache.org/downloads.html [18.06.2016]
[37] *GPI-2 | GPI-2 main website*, http://www.gpi-site.com/gpi2/ [18.06.2016]
[38] *RFC 3530 – Network File System (NFS) version 4 Protocol*, https://tools.ietf.org/html/rfc3530 [18-cze-2016]
[39] *Maven – Welcome to Apache Maven*, https://maven.apache.org/ [18.06.2016]
[40] *Optimize Options – Using the GNU Compiler Collection (GCC)*, https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html [18.06.2016]
[41] *ISO/IEC JTC1/SC22/WG21 – The C++ Standards Committee – ISOCPP*, http://www.open-std.org/JTC1/SC22/WG21/ [18.06.2016]

[42] Kwiatkowski J., *Parallel applications performance evaluation using the concept of granularity*, International Conference on Parallel Processing and Applied Mathematics, Warsaw, 2013
[43] Chiba T., Onodera T., *Workload Characterization and Optimization of TPC-H Queries on Apache Spark*, IBM Research Report RT0968, Tokyo, 2015
[44] *Configuration – Spark 1.4.0 Documentation*, http://spark.apache.org/docs/1.4.0/configuration.html#runtime-environment [27.06.2016]
[45] *Spark Programming Guide – Spark 1.6.1 Documentation*, http://spark.apache.org/docs/latest/programming-guide.html#rdd-persistence [27.06.2016]

Performance and productivity comparison of algorithm of decision tree generation implemented in SPARK and GASPI environments

Abstract

In this paper, the performance and programming productivity of cloud computing is explored for two different programming environments (SPARK and GASPI) applied to parallel implementation of big data problems. The ID3 algorithm of decision tree generation is selected as a test case. All the experiments are performed on the Google Compute Engine platform.

Keywords – Cloud computing, big data, decision tree generation, ID3 algorithm, SPARK and GASPI environments, comparison, Google Compute Engine