UNIWERSYTET EKONOMICZNY W KATOWICACH

KIERUNEK INFORMATYKA

JOANNA PAWŁOWSKA
149061

ANALIZA WYDAJNOŚCI RÓWNOLEGŁYCH IMPLEMENTACJI ALGORYTMU ID3

PERFORMANCE ANALYSIS OF PARALLEL IMPLEMENTATIONS OF THE ID3 ALGORITHM

Praca magisterska napisana
pod kierunkiem dr hab. Jana Kozaka, prof. UE

KATOWICE 2023

OŚWIADCZENIE PROMOTORA

Oświadczam, że niniejsza praca została przygotowana pod moim kierunkiem i stwierdzam, że spełnia wymogi stawiane pracom dyplomowym.

Pracę akceptuję

....
(data) (podpis promotora)

.....
Imię i nazwisko

Katowice, dnia .....

.....
Kierunek

.....
Nr albumu

OŚWIADCZENIE

Mając świadomość odpowiedzialności prawnej oświadczam, że złożona praca magisterska pt.

……..……..……...

.....……

została napisana przeze mnie samodzielnie.

Równocześnie oświadczam, że praca ta nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych oraz dóbr osobistych chronionych prawem.

Ponadto praca nie zawiera informacji i danych uzyskanych w sposób niedozwolony i nie była wcześniej przedmiotem innych procedur związanych z uzyskaniem dyplomów lub tytułów zawodowych uczelni wyższej.

Wyrażam zgodę na nieodpłatne udostępnienie mojej pracy w celu oceny jej oryginalności przez Jednolity System Antyplagiatowy prowadzony przez ministra właściwego ds. szkolnictwa wyższego oraz przechowywania jej w Ogólnopolskim Repozytorium Prac Dyplomowych oraz wewnętrznej bazie prac dyplomowych Uniwersytetu Ekonomicznego w Katowicach. Oświadczam, że poinformowano mnie o zasadach dotyczących oceny oryginalności pracy dyplomowej przez Jednolity System Antyplagiatowy.

Oświadczam także, że ostateczna wersja pracy przesłana przeze mnie drogą elektroniczną jest zgodna z plikiem poddanym ocenie w Jednolitym Systemie Antyplagiatowym.

Jednocześnie oświadczam, że jest mi znany przepis art. 233 § 1 Kodeksu karnego określający odpowiedzialność za składanie fałszywych zeznań.

.....
(podpis składającego oświadczenie)

Spis treści

Wstęp

Równoległość jest techniką wykorzystywaną w wielu obszarach informatyki, dzięki której możliwe jest szybsze przetwarzanie danych oraz bardziej efektywne wykorzystanie zasobów komputera. Stosowanie współbieżności w uczeniu maszynowym miało swoje początki już w latach 80-tych XX wieku [1]. Od tego czasu nastąpiło jednak wiele zmian zarówno technologicznych jak i w dziedzinie uczenia maszynowego. Rozwój informatyki sprzętowej zapewnia większe możliwości w kontekście dostępnej mocy obliczeniowej. Z kolei postęp w obszarze uczenia maszynowego doprowadził do opracowania coraz bardziej skomplikowanych algorytmów, przetwarzających stale rosnące zbiory danych. Wszystkie te czynniki sprawiają, że zagadnienie współbieżności w uczeniu maszynowym pozostaje ciągle istotne i aktualne.

Celem pracy jest zaproponowanie różnych implementacji algorytmu ID3 oraz analiza ich wydajności. Punktem wyjściowym do porównania będzie implementacja sekwencyjna, w odniesieniu do której, porównane zostaną dwie implementacje równoległe. Oczekiwanym rezultatem pracy jest przygotowanie dwóch równoległych wersji algorytmu, które w założeniu powinny okazać się bardziej efektywne od wersji sekwencyjnej.

Praca składa się ze wstępu, pięciu rozdziałów oraz podsumowania. Pierwsze trzy rozdziały wchodzą w skład teoretycznej części pracy. Rozdział czwarty oraz piąty stanowią praktyczną część pracy.

Pierwszy rozdział poświęcony jest matematycznemu opisowi drzew decyzyjnych. Przedstawiony został w nim schemat konstrukcji drzew decyzyjnych oraz różne algorytmy służące do ich budowy. Rozdział drugi skupiony jest na zagadnieniach związanych z przetwarzaniem równoległym. Zawiera on opis rodzajów procesów, sposoby dekompozycji problemów obliczeniowych a także opis wzorców programowania równoległego. Rozdział trzeci prezentuje przegląd literatury na temat równoległych konstrukcji drzew decyzyjnych. W rozdziale czwartym przedstawione zostały zaimplementowane wersje algorytmu ID3, które umożliwiają budowanie drzew decyzyjnych w sposób sekwencyjny oraz równoległy. W ostatnim rozdziale omówiona została analiza wydajności zaimplementowanych wersji algorytmu. Wyniki przeprowadzonych testów zaprezentowane zostały w formie graficznej. Dodatkowo opisane zostały zaistniałe problemy implementacyjne.

1. Drzewa decyzyjne

Drzewo decyzyjne (ang. decision tree) to rodzaj algorytmu uczenia maszynowego wykorzystywanego do klasyfikacji danych. Na podstawie danych wejściowych budowany jest model, który pozwala na podejmowanie decyzji. Drzewo decyzyjne przedstawiane jest graficznie jako struktura zbudowana z węzłów, gałęzi i liści. Każdy węzeł reprezentuje test, na podstawie którego algorytm podejmuje decyzję. Gałęzie symbolizują możliwe wyniki testu. Ostateczne decyzje, które mogą zostać podjęte przez algorytm, reprezentowane są przez liście.

Według matematycznej definicji drzewo decyzyjne to acykliczny graf skierowany. Gałęzie odpowiadają krawędziom grafu. Wierzchołki grafu nazywane są węzłami. Węzły, które nie mają żadnych potomków, określane są liśćmi, natomiast węzeł nieposiadający rodzica nazywany jest korzeniem [2].

1.1. Definicje

Rekord (inaczej nazywany krotką, wierszem lub obiektem) jest wektorem wartości atrybutów. Zbiór $n \in \mathbb{N}$ atrybutów wejściowych oznaczany jest przez $A = \{a_1, ..., a_i, ..., a_n\}$. Atrybut a_i przyjmuje wartości, których zbiór oznaczany jest przez $dom(a_i) = \{v_{i,1}, v_{i,2}, ..., v_{i,|dom(a_i)|}\}$, gdzie $|dom(a_i)|$ oznacza moc zbioru wartości atrybutu a_i. Atrybut wyjściowy nazywany jest decyzją i oznaczany jest przez y. Możliwe wartości decyzji $dom(y) = \{c_1, ..., c_{|dom(y)|}\}$ nazywane są klasami decyzyjnymi. Przestrzeń rekordów wyznaczona jest jako iloczyn kartezjański wszystkich zbiorów atrybutów wejściowych $X = dom(a_1) \times ... \times dom(a_i) \times ... \times dom(a_n)$ oraz zbioru atrybutów wyjściowych $dom(y)$ i oznaczana jest literą $U = X \times dom(y)$ [3].

Zbiór danych S to zbiór m par takich, że $S = (\langle x_1, y_1 \rangle, ..., \langle x_m, y_m \rangle)$, gdzie: $m \in \mathbb{N}$, $x_q \in X$, $y_q \in dom(y)$. Zbiór S graficznie przedstawiany jest jako tabela i nazywany jest tabelą decyzyjną. Rekordy tworzą wiersze tabeli a kolumny grupują wartości atrybutów [3].

Tabela 1: Tabela decyzyjna dla $m = 5$, $|dom(y)| = 3$, $|dom(a_1)| = 2$, $|dom(a_2)| = 2$, $|dom(a_3)| = 4$ i $n = 3$.

a_1	a_2	a_3	y
$v_{1,1}$	$v_{2,1}$	$v_{3,1}$	c_1
$v_{1,2}$	$v_{2,1}$	$v_{3,2}$	c_3
$v_{1,1}$	$v_{2,1}$	$v_{3,1}$	c_1
$v_{1,2}$	$v_{2,1}$	$v_{3,3}$	c_3
$v_{1,2}$	$v_{2,2}$	$v_{3,4}$	c_2

Zazwyczaj zakłada się, że rekordy, które należą do zbioru S generowane są losowo zgodnie z pewnym nieznanym, wspólnym rozkładem prawdopodobieństwa D nad przestrzenią U. Selekcja (σ) względem atrybutów przedstawiona jest za pomocą notacji używanej w algebrze. Przykładowo wybranie z tabeli 1 rekordów, które dla atrybutu a_3 przyjmują wartość $v_{3,1}$ opisywane jest wyrażaniem $\sigma_{a_3=v_{3,1}}S$.

Testem t nazywany jest warunek dla podziału danych. W celu wyznaczenia najlepszego testu wykorzystywane są różne kryteria podziału. Przy użyciu testów na atrybutach konstruowane jest drzewo decyzyjne DT. Klasyfikator, który został stworzony ze zbioru danych S oznaczany jest jako $DT(S)$. Korzystając z klasyfikatora $DT(S)$ możliwe jest wyznaczenie predykcji $DT(S)(x_q)$ dla wybranego elementu $x_q \in X$. Rozmiar drzewa decyzyjnego $DT(S)$ oraz dokładność predykcji $DT(S)(x_q)$ w dużym stopniu zależy od wielkości zbioru S. Jeśli zbiór danych jest zbyt mały, to dokładność predykcji będzie niska.

Błąd klasyfikacji drzewa $DT(S)$ jest prawdopodobieństwem błędnej predykcji obiektu wybranego zgodnie z rozkładem D. Błąd ε zdefiniowany jest następująco (w przypadku atrybutów ciągłych znak sumy zastępowany jest całką):

$$\varepsilon(DT(S), D) = \sum_{\langle x,y\rangle \in U} D(x,y) \cdot L(y, DT(S)(x)), \tag{1}$$

$$L(y, DT(S)(x)) = \begin{cases} 1, & DT(S)(x) = y, \\ 0, & DT(S)(x) \neq y. \end{cases} \tag{2}$$

Dokładność klasyfikacji obliczana jest jako $1 - \varepsilon$. Rzeczywista wartość błędu ε jest jednak rzadko wyznaczana, ponieważ najczęściej rozkład D nie jest znany. W zamian, jako oszacowanie błędu klasyfikacji, korzysta się z błędu $\hat{\varepsilon}$ obliczanego tylko na zbiorze danych. Dokładność klasyfikacji wyznaczana jest wtedy jako $\frac{\hat{\varepsilon}}{|S|}$.

$$\hat{\varepsilon}(DT(S), S) = \sum_{\langle x,y\rangle \in S} L(y, DT(S)(x)), \tag{3}$$

Wykorzystywanie błędu $\hat{\varepsilon}$ wyliczanego na podstawie całego zbioru danych S zazwyczaj daje jednak zbyt optymistycznie oszacowanie błędu ε. Z tego powodu zbiór danych dzieli się na zbiór treningowy oraz testowy. Zbiór treningowy jest większy i na jego podstawie buduje się klasyfikator. Zbiór testowy wykorzystywany jest do wyliczania $\hat{\varepsilon}$, który zwykle zapewnia lepsze oszacowanie błędu ε [3].

Nadmierne dopasowanie (ang. overfitting) to sytuacja, w której utworzony klasyfikator jest zbyt dobrze dopasowany do danych treningowych. Wystąpienie nadmiernego dopasowania sprawia, że klasyfikator dobrze sprawdza się dla danych treningowych, jednak zmniejsza się jego zdolność generalizacji. Oznacza to, że dla elementu $x_q \in X$, mniejsze jest prawdopodobieństwo na otrzymanie poprawnej predykcji $DT(S)(x_q)$. W przypadku drzew decyzyjnych nadmierne dopasowanie występuje najczęściej, gdy drzewo ma zbyt wiele węzłów w stosunku do ilości dostępnych danych treningowych [4].

W celu uniknięcia lub minimalizacji zjawiska nadmiernego dopasowania stosuje się technikę przycinania drzewa (ang. pruning). Polega ona na odpowiednim upraszczaniu drzewa poprzez zmniejszanie jego rozmiaru. W drzewie decyzyjnym wycina się wybrane fragmenty (poddrzewa), których znaczenie jest niewielkie podczas przeprowadzania predykcji obiektów. Poddrzewo, które zostało wybrane do wycięcia, najczęściej zastępuje się liściem z etykietą klasy, która najczęściej występuje w wycinanym podzbiorze. Przekształcenia drzewa mogą pogorszyć dokładność klasyfikacji na zbiorze danych treningowych, jednak często skutkują dokładniejszymi predykcjami na obiektach spoza zbioru treningowego [4].

1.2. Schemat konstrukcji

Proces budowania drzewa oparty jest na wielokrotnym podziale danych. Pierwszy krok polega na przypisaniu korzeniowi wszystkich obiektów ze zbioru treningowego S. Następnie, stosując kryterium podziału zależne od użytego algorytmu, wyznaczany jest atrybut wejściowy względem którego zostanie wykonany podział obiektów. Najlepszym podziałem jest ten, który najmniej różnicuje obiekty ze względu na ich klasę decyzyjną. Gdy wybrany zostanie atrybut, wykonywany jest test, który przydziela obiekty nowym węzłom potomnym. Po dopasowaniu obiektów do nowych węzłów, proces podziału jest powtarzany według takiej samej zasady. Podział wykonywany jest tak długo, aż osiągnięte zostanie kryterium stopu.

1.2.1. Testy atrybutów

Reguły definiujące podział w drzewach decyzyjnych są najczęściej jednowymiarowe. Oznacza to, że test dokonywany jest na podstawie tylko jednego atrybutu. Podziały wielowymiarowe są spotykane zdecydowanie rzadziej ze względu na wysoką złożoność obliczeniową [5]. Testy dzieli się w zależności od rodzaju atrybutów. Wyróżniane są dwa różne podziały dla atrybutów typu dyskretnego i dwa kolejne dla atrybutów ciągłych:

- dla atrybutów dyskretnych:
  - podział oparty na wartościach atrybutu:

$$t(x) = a_i(x),$$

gdzie:
$x \in X$,

  - podział oparty na równości:

$$t(x) = \begin{cases} 1, & \text{gdy } a_i(x) = v_{i,j}, \\ 0, & \text{w przeciwnym wypadku,} \end{cases}$$

gdzie:
$v_{ij} \in dom(a_i)$,

- dla atrybutów ciągłych:
  - podział oparty na nierównościach:

$$t(x) = \begin{cases} 1, & \text{gdy } a_i(x) \leqslant p, \\ 0, & \text{w przeciwnym wypadku,} \end{cases}$$

gdzie:
$p \in dom(a_i)$ – wartość progowa,

  - podział oparty na przedziałach:

$$t(x) = \begin{cases} 1, & \text{gdy } a_i(x) \in I_1, \\ 2, & \text{gdy } a_i(x) \in I_2, \\ 3, & \text{gdy } a_i(x) \in I_3, \\ \vdots & \\ k, & \text{gdy } a_i(x) \in I_l, \end{cases}$$

gdzie:
$l \in \mathbb{N}$,
$I_1, I_2, I_3, ..., I_l \subset dom(a_i)$,
$j \neq k \implies I_j \cap I_k = \varnothing$.

1.2.2. Kryteria stopu

Faza rozbudowy drzewa trwa do momentu, gdy któryś z warunków stopu zostanie spełniony. Wyróżnia się następujące kryteria zatrzymania konstruowania drzewa decyzyjnego:

- pusty zbiór treningowy,
- jednorodność obiektów - wszystkie rekordy mają taką samą wartość y (należą do tej samej klasy decyzyjnej),
- drzewo osiągnęło maksymalną wysokość,
- brak możliwość odnalezienia testu, który pozwoliłby na dokonanie podziału [3].

1.2.3. Ocena jakości

Dwie składowe, które wpływają na ocenę jakości Q drzewa decyzyjnego $DT(S)$, to dokładność klasyfikacji oraz wielkość drzewa decyzyjnego. Wielkość drzewa decyzyjnego wyznaczana jest na różne sposoby. Miarą może być wysokość drzewa, liczba węzłów lub liczba liści. We wszystkich przypadkach występuje jednak taka sama zasada – im mniejsze drzewo, które umożliwia przeprowadzanie poprawnych predykcji, tym wyższa jest jego jakość. Jakość drzewa decyzyjnego może oceniona być za pomocą (4):

$$Q(DT, S) = \alpha \cdot \hat{\varepsilon}(DT(S), S) + \beta \cdot h(DT), \tag{4}$$

gdzie:
α, $\beta \in \mathbb{R}$,
h – wysokość drzewa decyzyjnego.

1.3. Algorytmy konstrukcji drzew decyzyjnych

Algorytm ID3 jest uważany za jeden z prostszych algorytmów konstrukcji drzew decyzyjnych. Jako kryterium podziału wykorzystany jest zysk informacji $InformationGain$, który obliczany jest przy wykorzystaniu entropii H:

$$H(y, S) = -\sum_{c_j \in dom(y)} \frac{\sigma_{y=c_j} S}{|S|} \cdot log_2 \frac{\sigma_{y=c_j} S}{|S|}, \tag{5}$$

$$InformationGain(a_i, S) = H(y, S) - \sum_{v_{i,j} \in dom(a_i)} \frac{\sigma_{a_i=v_{i,j}} S}{|S|} \cdot H(y, \sigma_{a_i=v_{i,j}} S). \tag{6}$$

Atrybut a_i dla którego zysk informacji jest najwyższy wybierany jest do wyznaczenia podziału. Budowanie drzewa kończy się, gdy wszystkie obiekty w węźle mają taką samą wartość klasy lub gdy najlepszy obliczony zysk informacji nie jest większy od zera [2].

Algorytm stosowany jest dla atrybutów dyskretnych. Jeśli atrybuty są ciągłe, wówczas muszą zostać przekształcone. ID3 nie sprawdza się w przypadku, gdy zbiór treningowy składa się z rekordów, którym brakuje wartości pojedynczych atrybutów. ID3 jest algorytmem zachłannym, dlatego nie daje gwarancji odnalezienia rozwiązania optymalnego, ponieważ może utknąć w optimum lokalnym [3].

Algorytm C4.5 jest ulepszoną wersją ID3 zaproponowaną przez tego samego autora. Jako kryterium podziału wykorzystany został współczynnik względnego zysku informacji *GainRatio*, który zdefiniowany jest następująco:

$$GainRatio(a_i, S) = \frac{InformationGain(a_i, S)}{H(a_i, S)}, \tag{7}$$

W przeciwieństwie do algorytmu ID3, algorytm C4.5 może przetwarzać zarówno atrybuty dyskretne jak i ciągłe. Obsługa atrybutów ciągłych uzyskiwana jest poprzez podział wartości atrybutu na dwa podzbiory zgodnie z najlepszym znalezionym progiem. Dodatkowo jest on dostosowany do obsługi rekordów z brakującymi wartościami atrybutów. Kolejnym usprawnieniem jest zastosowanie procedury przycinania. Pozwala ona na usuwanie gałęzi wraz z połączonymi węzłami potomnymi, które nie przyczyniają się do poprawienia dokładności klasyfikatora i zastępowanie ich jednym węzłem liścia. Komercyjnie wykorzystuje się **algorytm C5.0**, który jest zaktualizowaną wersją algorytmu C4.5. Jest on zoptymalizowany pod kątem czasu obliczeń i wykorzystanej pamięci [3].

Algorytm CART pozwala na konstruowanie drzew opartych są na binarnym podziale atrybutów. Kryterium podziału wykorzystuje indeks *Gini*, który mierzy rozbieżności między rozkładami prawdopodobieństwa wartości atrybutów. Kryterium oceny wyboru atrybutu a_i zdefiniowane jest jako *GiniGain* [3]:

$$Gini(y, S) = 1 - \sum_{c_j \in dom(y)} \left(\frac{|\sigma_{y=c_j} S|}{|S|} \right)^2, \tag{8}$$

$$GiniGain(a_i, S) = Gini(y, S) - \sum_{v_{i,j} \in dom(a_i)} \frac{|\sigma_{a_i=v_{i,j}} S|}{|S|} \cdot Gini(y, \sigma_{a_i=v_{i,j}} S) \tag{9}$$

Podobnie jak w algorytmie C4.5 dopuszczalne są zarówno atrybuty dyskretne jak i ciągłe.

Algorytm CHAID (ang. Chi-squared Automatic Interaction Detectorest) to jeden z najstarszych algorytmów budowy drzew decyzyjnych, który można wykorzystać zarówno dla atrybutów ciągłych jak i dyskretnych. Drzewa stworzone przy wykorzystaniu algorytmu CHAID są drzewami niebinarnymi.

Dla każdego atrybutu a_i znajdywana jest para wartości $\langle v_{ij}, v_{ik} \rangle$, która jest najmniej różna w odniesieniu do decyzji. Różnica jest mierzona poprzez wykonanie testu statystycznego, którego wybór zależy od rodzaju atrybutu wyjściowego. Gdy jest on typu dyskretnego kryterium podziału opiera się na teście *Chi*-kwadrat, w przeciwnym wypadku wykorzystuje się test F.

Dla każdej pary wartości sprawdza się, czy wynik testu statystycznego jest większy od progu scalania. Jeśli wynik testu jest większy to wartości są łączone, a następnie ponownie wykonuje się proces wyboru pary. Proces powtarza się do momentu, gdy nie będzie możliwe odnalezienie żadnej znaczącej pary. Następnie wybierany jest najlepszy atrybut wejściowy, który zostanie użyty do podziału bieżącego węzła, tak aby każdy węzeł potomny składał się z grupy jednorodnych wartości wybranego atrybutu [3].

Algorytm CHAID obsługuje brakujące wartości, które traktuje jako jedną kategorie. W algorytmie nie stosuje się przycinania.

2. Programowanie równoległe

Rozdział ma na celu uporządkowanie istotnych pojęć związanych z tematem programowania równoległego. Opisane zostały podstawowe zagadnienia, różnice między programowaniem współbieżnym i równoległym, rodzaje dekompozycji zadań, a także wzorce, stosowane w równoległych implementacjach algorytmów.

2.1. Podstawowe pojęcia

Przetwarzanie równoległe jest tematem omawianym w ramach dziedziny systemów operacyjnych. Mimo, że celem pracy nie jest analiza zagadnień specyficznych dla systemów operacyjnych, poniżej przestawione zostały pojęcia, których zrozumienie jest konieczne do tworzenia równoległych implementacji algorytmów.

Procesor to jednostka sprzętowa, która pobiera dane z pamięci operacyjnej, interpretuje je i wykonuje. Pojęcie procesor używane jest w dwóch kontekstach. Pierwszy, zgodny z podaną definicją, używany jest głównie w elektronice. Drugie znaczenie procesora wykorzystywane jest częściej w programowaniu. Wówczas pojęcie procesor jest synonimem pojęcia rdzeń.

Rdzeniem fizycznym (ang. core) określany jest fizyczny element procesora, który pozwala na wykonywanie obliczeń. W uproszczeniu, im więcej rdzeni posiada procesor, tym więcej obliczeń może on wykonać w jednostce czasu, co definiowane jest jako moc obliczeniowa. Jest to jeden z czynników branych pod uwagę przy ocenie wydajności procesora. Obecnie używa się głównie procesorów wielordzeniowych.

Rdzeń logiczny (ang. logical core, thread) przygotowuje dane wykorzystywane podczas obliczeń wykonywanych przez rdzeń fizyczny. Do niedawna na jeden rdzeń fizyczny przypadał jeden rdzeń logiczny. Obecnie najczęściej każdemu rdzeniowi fizycznemu przypisuje się dwa rdzenie logiczne. Rdzenie logiczne uzależnione są od rdzenia fizycznego, do którego są przypisane, natomiast nie są zależne od operacji rdzeni logicznych przypisanych do innego rdzenia fizycznego.

Rozkazem nazywane jest pojedyncze polecenie, zapisane jest w postaci liczb binarnych, które wykonywane jest przez procesor.

Instrukcja definiowana jest jako bardziej złożone zadanie, które składa się ze zbioru rozkazów. Instrukcje mogą być niskopoziomowe (napisane w np. języku *Asembler*) lub wysokopoziomowe (napisane w językach np. *C* i *Java*). Instrukcje wysokopoziomowe tłumaczone są na kilka instrukcji niskopoziomowych, natomiast instrukcje niskopoziomowe tłumaczone są na zbiór rozkazów. Zbiór rozkazów może być dzielony na podzbiory w celu uruchomienia

każdego podzbioru na innym procesorze.

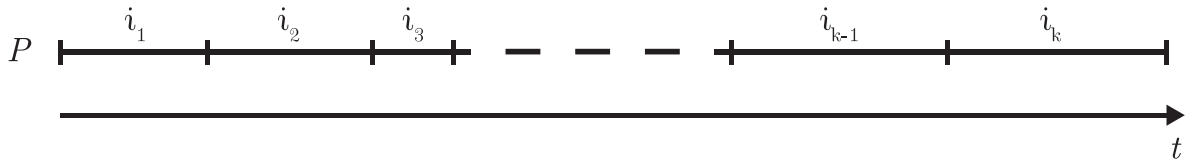
Program jest zbiorem instrukcji, który pozwala na rozwiązanie pewnego problemu obliczeniowego.

Proces najprościej definiowany jest jako program, który jest w trakcie wykonywania. Pod pojęciem procesu zawiera się jednak wiele mechanizmów, przy pomocy których system operacyjny zarządza wykonywaniem programu. System operacyjny przydziela każdemu nowemu procesowi zasoby m.in. odrębny obszar pamięci operacyjnej, nadaje unikatowy numer PID (ang. process identifier), kontroluje stan procesu oraz zarządza plikami, z których korzysta proces [6].

Wątek (ang. thread) jest częścią procesu. Jest to niezależny strumień instrukcji, który uruchamiany jest przez system operacyjny. Na jeden proces najczęściej składa się wiele strumieni instrukcji. Instrukcje składające się na wątek są wykonywane sekwencyjnie. Wszystkie wątki istniejące w ramach jednego procesu współdzielą przestrzeń adresową – mają dostęp do pamięci wspólnej. Z tego powodu komunikacja między wątkami należącymi do jednego procesu jest łatwa i nie wymaga wsparcia systemu operacyjnego. Przekazanie danych polega na podaniu jedynie wskaźnika do miejsca w pamięci. Programiści często definiują wątek nieco inaczej. Wątek traktowany jest jako nieblokująca metoda, która wykonywana jest niezależnie od procesu, który ją uruchomił [6]. Określenie wątek jest również używane wymiennie z określeniem rdzeń logiczny. Wątek w znaczeniu rdzenia logicznego nie jest jednak równoznaczny przedstawionej definicji wątku.

2.2. Rodzaje procesów

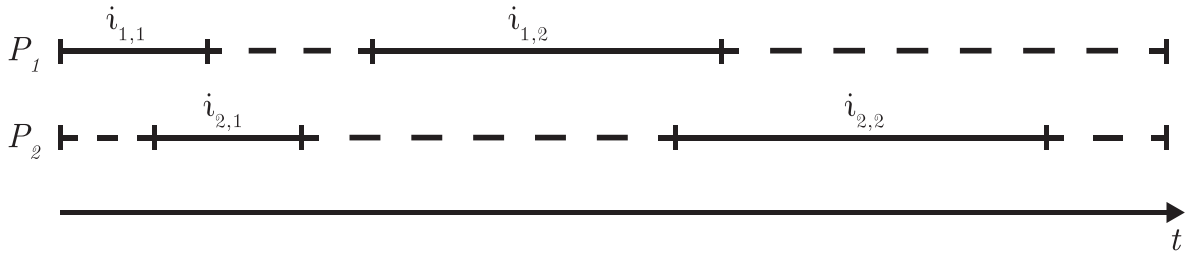
Proces sekwencyjny P charakteryzuje się tym, że każda kolejna instrukcja i wykonywana jest dopiero wtedy, gdy zakończy się wykonywanie poprzedniej. Kolejność wykonywania instrukcji jest jednoznacznie określona, dlatego proces sekwencyjny określany jest jako pojedynczy ciąg instrukcji [7]. W przypadku procesu sekwencyjnego wątek nie jest częścią procesu, natomiast jest z nim utożsamiany [8]. Schemat przykładowego procesu sekwencyjnego przedstawiony jest na rysunku 1.



Rysunek 1: Proces sekwencyjny. Źródło: Opracowanie własne na podstawie [7]

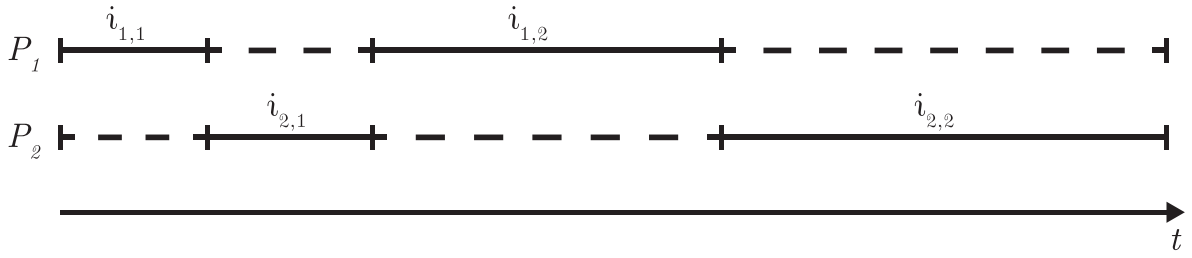
Procesy sekwencyjne, które zachodzą na siebie w czasie, określane są jako **proces współbieżny**. Innymi słowami proces współbieżny to proces, który składa się z wielu strumieni instrukcji. Instrukcje należące do jednego wątku, wykonywane są, zanim ukończone zostanie wykonywanie wszystkich instrukcji tworzących drugi, wcześniej uruchomiony wątek.

Dane są dwa procesy sekwencyjne P_1 i P_2, instrukcje $i_{1,1}, i_{1,2} \in P_1$ oraz $i_{2,1}, i_{2,2} \in P_2$. Rysunek 2 przedstawia jedną z wielu możliwych realizacji procesu P_1 oraz P_2.



Rysunek 2: Proces współbieżny. Źródło: Opracowanie własne na podstawie [7]

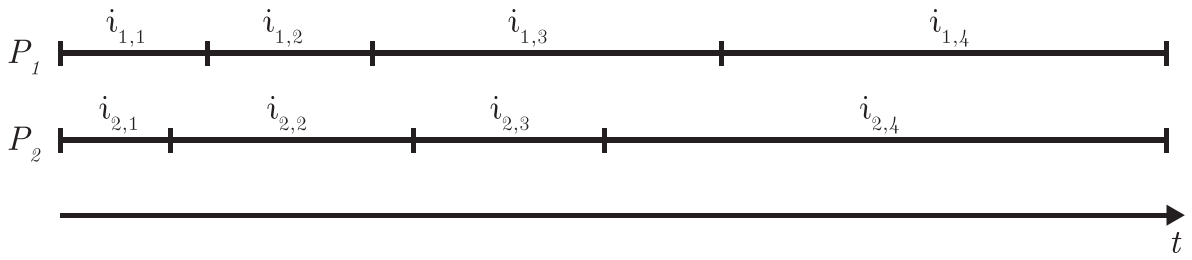
Procesy wykonywane w przeplocie są procesami współbieżnymi, w których wątki uruchamiane są naprzemiennie. Podczas wykonywania procesu P_1 następuje wstrzymanie procesu P_2. Gdy przerywane jest działanie procesu P_1 to przez pewien czas wykonywany jest proces P_2. Metoda przeplotu pozwala na zastosowanie współbieżności w procesorach jednordzeniowych. Rysunek 3 prezentuje schemat procesów wykonywanych w przeplocie.



Rysunek 3: Procesy wykonywane metodą przeplotu.
Źródło: Opracowanie własne na podstawie [7]

Proces równoległy jest szczególnym rodzajem procesu współbieżnego, w którym wątki uruchamiane są jednocześnie. Jeśli wątki mają zostać rozdzielone między różne rdzenie procesora, wówczas konieczne jest wykorzystanie procesora z kilkoma rdzeniami. Inną możliwością jest zastosowanie architektury rozproszonej. W takiej sytuacji wątki dzielone są między zbiór

komputerów, które połączone są ze sobą siecią. Proces równoległy nazywany jest wtedy rozproszonym. Na rysunku 4 przedstawione zostały procesy równoległe.



Rysunek 4: Procesy równoległe. Źródło: Opracowanie własne na podstawie [7]

2.3. Rodzaje dekompozycji problemów obliczeniowych

W celu równoległego wykonywania programu istotne jest zaprojektowanie podziału zadań obliczeniowych. Podział problemu na zadania nazywany jest dekompozycją. Wyróżniane są cztery rodzaje dekompozycji [7].

Dekompozycja danych to jeden z najczęściej wykorzystywanych rodzajów dekompozycji. Swoje zastosowanie znajduje szczególnie w przypadkach, gdzie przetwarzane są bardzo duże ilości danych. Dekompozycja danych dzieli się na dekompozycję danych wejściowych i wyjściowych. Pierwsza z nich polega na podziale danych wejściowych na względnie równe części, które przetwarzane są w ramach osobnych zadań. Najczęściej zadania polegają na wykonaniu dokładnie takiego samego rodzaju obliczeń. Taki rodzaj dekompozycji charakteryzuje się tym, że po zakończeniu zadań, konieczne jest ich zsumowanie.

Dekompozycja danych wyjściowych jest możliwa, gdy elementy danych wyjściowych mogą zostać wyznaczone niezależnie od siebie. Wówczas każdemu zadaniu przydzielone zostają te dane wejściowe, które konieczne są do otrzymania poszczególnych elementów danych wyjściowych. Wadą takiego podejścia jest stosunkowo niski stopień współbieżności [7].

Podejście, w którym zrównoleglanie osiągane jest poprzez zastosowanie dekompozycji danych, określany jest pojęciem *równoległość danych*.

Dekompozycja funkcjonalna polega na wyodrębnieniu obliczeń, których wykonanie konieczne jest do rozwiązania problemu. Obliczenia dzielone są na grupy, które formowane są w funkcje. Zadania funkcji różnią się od siebie i najczęściej przetwarzają różne rodzaje danych [7]. Zastosowanie dekompozycji funkcjonalnej oznacza wykorzystanie *równoległości zadań*.

Dekompozycja rekursywna stosowana jest przy rozwiązywaniu problemów metodą

„dziel i zwyciężaj". Problem dzielony jest na mniejsze podproblemy, które są od siebie niezależne. Każdy podproblem jest mniejszym przypadkiem pierwotnego problemu. Podział wykonywany jest tak długo, aż podproblemy stają się trywialne do rozwiązania. Następnie wszystkie rozwiązania scalane są w jedno, które jest ostatecznym rozwiązaniem [7].

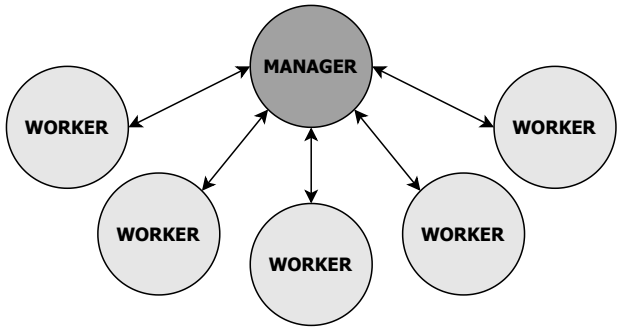
Dekompozycja eksploracyjna używana jest wtedy, gdy zadanie obliczeniowe polega na przeszukiwaniu przestrzeni rozwiązań. Przestrzeń dzielona jest na części, które eksplorowane są równolegle przez odrębne zadania. Jeśli rozwiązanie zostanie znalezione w którejś części przestrzeni, wówczas wykonywanie pozostałych zadań jest przerywane [7].

2.4. Wzorce programowania równoległego

Istnieje wiele wzorców programowania równoległego. Podczas implementacji algorytmu równoległego ciężko jest dobrać jeden, najlepiej pasujący wzorzec. Z tego powodu wzorce traktowane są raczej jako ogólne wskazówki, które mogą być przydatne podczas projektowania algorytmu. Najczęściej programy tworzone są zgodnie z kilkoma wzorcami, które łączone są ze sobą w celu stworzenia implementacji dopasowanej do konkretnego przypadku. W tym rozdziale opisane zostały wybrane wzorce programowania równoległego.

2.4.1. Wzorzec Manager-Worker

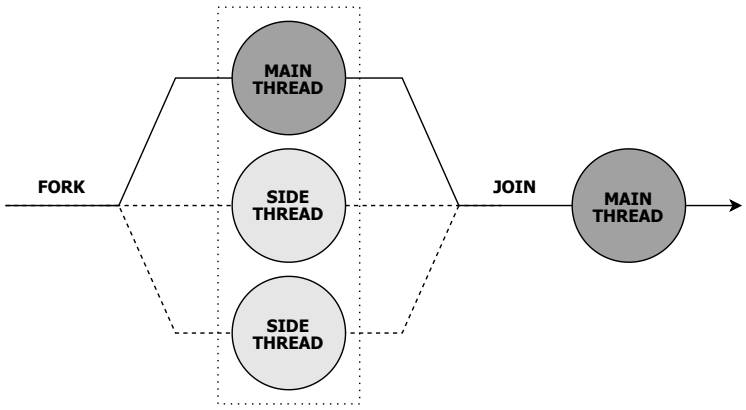
Wzorzec *Manager-Worker* zaprezentowany jest na rysunku 5. Wątek, który nazywany jest zarządcą (ang. manager) definiuje zadania i rozdziela je pomiędzy wykonawców (ang. worker). Gdy wykonawca zrealizuje zadanie, przesyła otrzymane wyniki zarządcy, którego zadaniem jest scalić wszystkie zgromadzone wyniki w jedno rozwiązanie problemu [9]. Wzorzec nie sprawdza się w problemach o dużym rozdrobnieniu zadań. Wówczas dochodzi do sytuacji, w której wykonawcy realizują poszczególne zadania szybciej, niż zarządca jest w stanie je generować i rozdzielać [7].



Rysunek 5: Wzorzec *Manager-Worker*. Źródło: Opracowanie własne.

2.4.2. Wzorzec Fork-Join

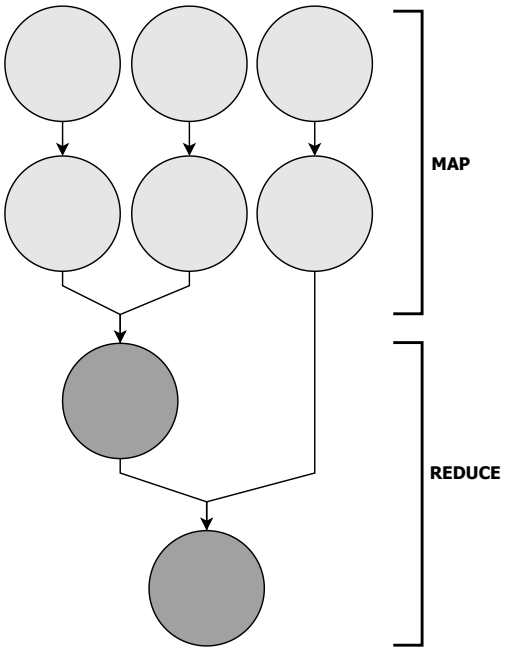
Wzorzec *Fork-Join* to jeden z najczęściej stosowanych wzorców. Wykonywanie programu rozpoczyna się w ramach jeden wątku głównego. W momencie, gdy w kodzie programu pojawia się instrukcja, która wymaga równoległego przetwarzania, tworzone są dodatkowe wątki poboczne. Sytuacja ta zaprezentowana jest na rysunku 6. Dopóki wszystkie wątki poboczne nie zakończą równoległej pracy i nie zostaną zniszczone, wątek główny nie może wznowić wykonywania sekwencyjnej części kodu. Etap „fork" polega na ustaleniu argumentów, które następnie otrzymuje każdy wątek. Etap „join" łączy wyniki po zakończeniu pracy wszystkich wątków równoległych. Etapy „fork" i „join" mogę wykonywane być dowolną liczbę razy [10].



Rysunek 6: Wzorzec *Fork-Join*. Źródło: Opracowanie własne.

2.4.3. Wzorzec Map-Reduce

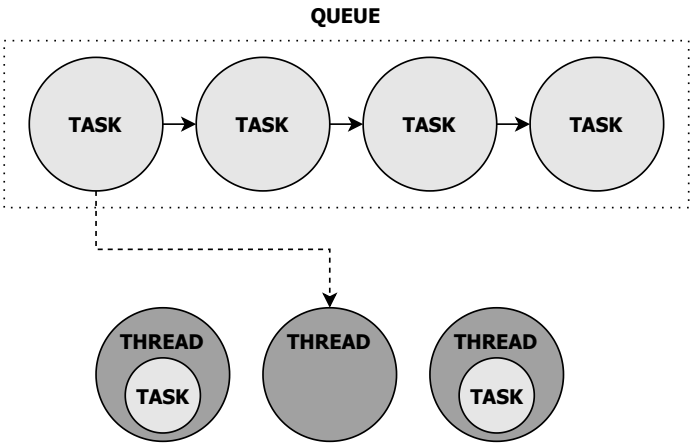
Wzorzec *Map-Reduce* jest podobny do wzorca *Fork-Join* 6. Dane wejściowe są przetwarzane równolegle przez wiele wątków. Następnie wszystkie uzyskane wyniki są łączone, aż do momentu uzyskania jednego rozwiązania. W działaniu obydwa wzorce są prawie identycznie, jednak wywodzą się z różnych pomysłów. Idea mapowania pochodzi z technik stosowanych w funkcjonalnych językach programowania. Kilka mapowań może być połączonych w łańcuchy składające się na większe funkcje. Etapy „map" i „reduce" są od siebie bardziej niezależne niż etapy „fork" i „join". Mapowanie może występować bez redukcji, a redukcja bez mapowania [10]. Graficzne przedstawienie wzorca *Map-Reduce* zaprezentowane jest na rysunku 7.



Rysunek 7: Wzorzec *Map-Reduce*. Źródło: Opracowanie własne.

2.4.4. Wzorzec Work Pool

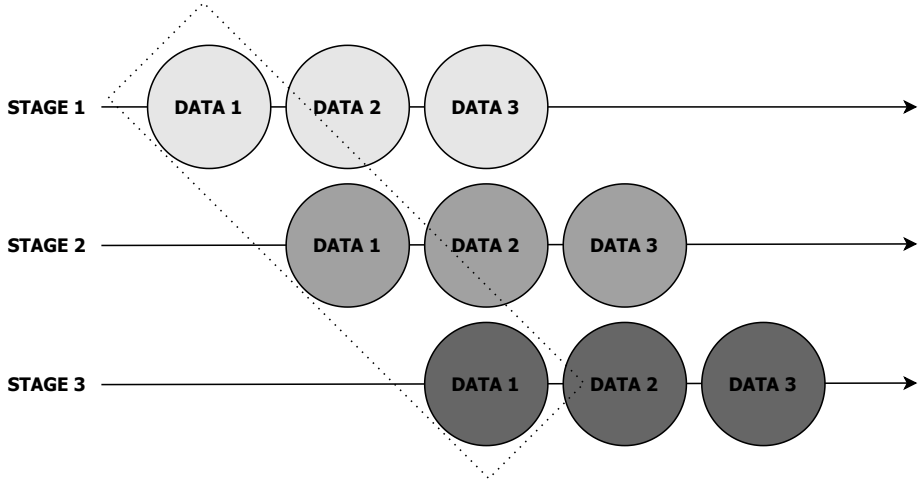
Wzorzec puli zadań *Work Pool* wykorzystywany jest w algorytmach, w których zadania generowane są dynamicznie lub gdy istotnie różnią się złożonością. Zadania przechowywane są w strukturze danych, która dostępna jest w pamięci współdzielonej [9]. Najczęściej wykorzystywane struktury to lista, kolejka priorytetowe czy tablica z haszowaniem. W momencie, gdy wątek zakończył obliczanie zadania, dostarczane jest mu kolejne zadanie przechowywane w strukturze [7]. Rysunek 8 prezentuje schemat z kolejką zdań i trzema wątkami.



Rysunek 8: Wzorzec *Work Pool*. Źródło: Opracowanie własne.

2.4.5. Wzorzec Pipeline

Wzorzec *Pipeline*, przedstawiony na rysunku 9, polega na potokowym przetwarzaniu danych. Nazywany jest również metodą producenta i konsumenta. Strumień danych przekazywany jest do kolejnych wątków, które modyfikują oryginalny strumień. Każdy z wątków wykonuje inny rodzaj obliczeń, które tworzą etapy potoku. Dane wyjściowe jednego etapu stają się danymi wejściowymi kolejnego etapu. Stosowanie wzorca jest efektywne, jeśli czasy realizacji poszczególnych etapów są podobne [7].



Rysunek 9: Wzorzec *Pipeline*. Źródło: Opracowanie własne.

3. Przegląd literatury

Rozdział trzeci zawiera przegląd publikacji naukowych. Tematem omawianych prac są różne metody programowania równoległego stosowane w algorytmach uczenia maszynowego.

3.1. Równoległa konstrukcja algorytmu C4.5

W artykule [11] zostało przedstawionych kilka strategii konstrukcji algorytmów drzew decyzyjnych, które oparte są o techniki takie jak: równoległość zadań, równoległość danych oraz równoległość hybrydowa. W pracy zaprezentowana została autorska implementacja równoległej konstrukcji drzewa decyzyjnego algorytmem C4.5. Na zakończenie autorzy przedstawili wyniki działania algorytmu i postawili wstępne wnioski dotyczące jego działania.

Artykuł rozpoczyna się od zaprezentowania trudności, które pokazują, jak złożonym zadaniem jest implementacja równoległych algorytmów do budowy drzew decyzyjnych. Wymienione zostają m.in. problemy z zastosowaniem statycznego, jak i dynamicznego przydzielania procesorów. Nieregularny kształt drzewa, który określany jest dopiero w momencie działania programu, jest dużą przeszkodą do stosowania statycznej alokacji. Takie podejście prowadzi najczęściej do nierównomiernego rozłożenia obciążenia. W przeciwnej sytuacji, gdy dane przetwarzane są przez dynamicznie przydzielane procesory, problemem staje się konieczność zaimplementowania przekazywania danych. Współdzielenie danych jest wymagane, ponieważ część danych związanych z rodzicami musi dostępna być również dla potomków.

Autorzy szczegółowo opisują różnice pomiędzy równoległością zadań, równoległością danych oraz równoległością hybrydową. Równoległość zadań określana jest jako dynamiczne rozdzielanie węzłów decyzyjnych między procesory, w celu kontynuowania ich rozbudowy. Wadą takiego podejścia jest konieczność replikacji całego zbioru treningowego lub, alternatywnie, zapewnienie częstej komunikacji pomiędzy procesorami. Równoległość danych przedstawiona jest jako wykonywanie tego samego zbioru instrukcji algorytmu przez wszystkie zaangażowane procesory. Zbiór treningowy zostaje podzielony pionowo lub poziomo. Podział poziomy (horyzontalny) polega na na takim podziale danych między procesory, że każdy z nich odpowiedzialny jest za inny zestaw przykładów ze zbioru treningowego. Podział pionowy (wertykalny) rozdziela przetwarzanie poszczególnych atrybutów między procesory.

Autorzy zwracają uwagę, że przetwarzanie z pionowym podziałem danych narażone jest na wystąpienie nierównowagi obciążenia. Równoległość hybrydowa scharakteryzowana jest jako połączenie równoległości zadań oraz danych. Dla węzłów, które muszą przetworzyć dużą liczbę przykładów, wykorzystywana jest równoległość danych. W ten sposób unika się

problemów związanych z nierównomiernym obciążeniem. W przypadku węzłów z przypisaną mniejszą liczbą przykładów czas, potrzebny do komunikacji może być większy niż czas potrzebny do przetwarzania przykładów. Zastosowanie równoległości zadań w takie sytuacji pozwala uniknąć dysproporcji.

W kolejnej części artykułu przestawiona została implementacja równoległej konstrukcji drzewa decyzyjnego. Program został stworzony do wykonywania w środowisku pamięci rozproszonej, w której każdy z procesorów ma własną pamięć prywatną. Autorzy zaproponowali takie podejście, ponieważ ma ono rozwiązywać dwa problemy wspominane na początku pracy: równoważenie obciążenia oraz konieczność przekazywania danych. Każdy z procesorów ma za zadanie tworzyć własne listy atrybutów i klas na podstawie przydzielonych podzbiorów przykładów. Wykorzystanie obydwu list jest kluczem do osiągnięcia efektywnego paralelizmu. Wpisy w liście klas zawierają etykietę klasy, indeks globalny przykładu w zbiorze treningowym oraz wskaźnik do węzła w drzewie, do którego należy dany przykład. Listy atrybutów również zawierają wpis dla każdego przykładu z atrybutem, jak również indeks wskazujący na odpowiadający wpis w liście klas. Każdy procesor znajduje własne, najlepsze podziały lokalnego zbioru dla każdego atrybutu. Następnie komunikuje się z pozostałymi procesorami, w celu ustalenia jednego, najlepszego podziału. Po podziale (utworzeniu węzła) następuje aktualizacja list atrybutów przez każdy procesor, dokonana poprzez rozdzielenie atrybutów w zależności od wartości wybranego atrybutu dzielącego.

Zaprezentowane przez autorów wyniki określone są jako wstępne i wymagające udoskonaleń. Autorzy zdecydowali się jednak na wykorzystanie ich do przewidzenia oczekiwanego zachowania algorytmu. Implementacja wykorzystuje takie same kryteria oceny jak stosowane w algorytmie C4.5, dlatego autorzy skupili się głównie na analizie czasu potrzebnego do zbudowania drzewa. Do wszystkich testów wykorzystany był zestaw danych syntetycznych Agrawal, w którym każdy przykład ma dziewięć atrybutów (pięć ciągłych i trzy dyskretne).

Z przedstawionych rezultatów testów wynika, że algorytm wykazał dobre wyniki przyspieszania. Twórcy artykułu przeprowadzili również testy mające na celu sprawdzenie skalowalności. Jak w pierwszym przypadku, testy wykazały, że algorytm osiąga dobre wyniki skalowania.

3.2. Zestawienie artykułów

W literaturze można znaleźć wiele artykułów na temat metod programowania równoległego, które wykorzystywane są do optymalizacji algorytmów konstruujących drzewa decyzyjne. W pracach przedstawione są implementacje różnych wzorców programowania równoległe-

go. W celu przybliżenia różnorodności oraz przedstawienia dokładniejszego obrazu na temat wykorzystywanych wzorców i metod, artykuły zestawione zostały w tabeli.

Tabela 2 zawiera odnośniki do tytułów artykułów, słowa kluczowe oraz krótki opis przybliżający tematykę poruszaną w każdym z artykułów.

Tabela 2: Zestawienie artykułów poruszających tematykę równoległości w algorytmach drzew decyzyjnych

Artykuł	**Słowa kluczowe**	**Opis**
[12]	równoległość wewnątrzwęzłowa, równoległość międzywęzłowa	Autorzy przedstawili oraz porównali wydajność czterech metod do równoległej implementacji algorytmu C4.5. W analizie uwzględniony został rodzaj danych wykorzystywanych do konstrukcji drzewa, który okazał się mieć duży wpływ na wyniki wydajności porównywanych metod. Wyszczególnione zostały trzy rodzaje techniki wewnątrzwęzłowej, które wykorzystują zrównoleglanie przetwarzania danych. Równolegle przetwarzane mogą być rekordy, atrybuty lub ich kombinacja (podejście hybrydowe). W technice międzywęzłowej równolegle przetwarzane są całe węzły – wszystkie operacje, które muszą zostać przeprowadzone do stworzenia węzła, wykonywane są przez jeden wątek.
[13]	węzły Hoeffdinga, architektura *Master-Slave*	Autorzy artykułu zdecydowali się na zrównoleglanie tylko fragmentów algorytmu tj. szukania najlepszego atrybuty do podziału węzła. Zadania przydzielane przez główny procesor (*master*) innym procesorom (*slave*) polegają na obliczenia zysku informacyjnego dla określonego atrybutu. Zastosowana architektura oraz zrównoleglanie tylko wyszukiwania atrybutów skutkuje jednak w niskiej skalowalności. Pomimo tego, testy wykazały, że przyspieszenie działania algorytmu jest dość efektywne. Dzięki zastosowaniu nierówności Hoeffdinga, do podziału drzewa nie muszą być używane wszystkie dane. Ma to dodatkowy wpływ na szybkość działania algorytmu.

[14]	równoległość w węźle, duże zbiory danych	W artykule przedstawiono algorytm ParDTLT. Algorytm jest równoległą wersją algorytmu DTLT (ang. Decision Trees from Large Training sets), który nie wymaga ładowania w całości wszystkich danych treningowych do pamięci komputera. ParDTLT oparty jest na idei sekwencyjnej budowy struktury drzewa oraz równoległym przetwarzaniu danych w każdym węźle. W danym czasie wszystkie procesory dostępne są dla jednego węzła. W węźle uprzywilejowanym tworzona jest kolejka atrybutów, dla których obliczany jest zrównoważony zysk informacji. Analiza kolejnych atrybutów przydzielane są procesorom do momentu, aż kolejka będzie pusta. Pozostałe węzły drzewa czekają, aż staną się węzłem uprzywilejowanym. Autorzy artykuły przeprowadzili testy algorytmu, które wykazały, że ParDTLT jest szybszy od algorytmów jak np. DTLT czy C4.5.
[15]	pamięć współdzielona, algorytm ID3	Modyfikacjom został poddany algorytm ID3. Przedstawiono dwie różne implementacje wykorzystujące równoległość. Pierwsza polega na stworzeniu tylu wątków, ile istnieje atrybutów, dla których obliczony musi zostać przyrost informacji. Gdy atrybut dzielący zostanie odnaleziony, tworzony jest węzeł. Następnie ponownie tworzone są kolejne wątki, które przetwarzają atrybuty nowo powstałych węzłów. Dopiero gdy wszystkie wątki zakończą pracę, z dostarczonych wyników składane jest drzewo. Różnica w drugiej implementacji polega na wykorzystaniu pamięci współdzielonej, dzięki czemu algorytm jest bardziej wydajny pamięciowo. Węzeł główny jest współdzielony, dlatego każdy nowy węzeł może być tworzony, gdy tylko atrybut dzielący został odnaleziony.

[16]	algorytm ID3, wzorzec *Map-Reduce*	W pracy został zaprezentowany równoległy algorytm do budowy drzewa decyzyjnego, który oparty jest na modelu *Map-Reduce*. W pierwszej części, przedstawione zostały dwie główne metody podziału danych – metoda dynamiczna i statyczna. Podobnie jak w [11], w ramach statycznej metody podziału danych wyróżniony został podział pionowy i poziomy. Autorzy proponują zmodyfikowaną implementację algorytmu ID3 z wykorzystaniem biblioteki *MapReduce*. Dane prezentowane są jako para (klucz, wartość). Etap „map" polega na pionowym i poziomym podziale danych oraz generowaniu par klucz-wartość. Etap „reduce" zawiera w sobie sumowanie częściowych wyników, odnalezieniu najlepszego atrybutu oraz przeprowadzaniu podziału. Omówiony na koniec przykład potwierdza, że tak skonstruowany algorytm jest w stanie w efektywny sposób przetwarzać duże ilości danych.

4. Realizacja praktyczna

Rozdział poświęcony jest opisowi różnych implementacji algorytmu ID3. W pracy nacisk położony jest na analizę zastosowania równoległości w konstrukcjach drzew decyzyjnych, dlatego wybrano tylko jeden algorytm, który został zaimplementowany na trzy różne sposoby. Pierwszy sposób to klasyczny, sekwencyjny algorytm ID3. Drugi sposób wykorzystuje równoległość do obliczania zrównoważonego przyrostu informacji dla poszczególnych atrybutów. Trzeci sposób opiera się na zastosowaniu równoległości podczas tworzenia całych węzłów drzewa decyzyjnego. W rozdziale została również przedstawiona instrukcja uruchomienia programu, który zawiera wszystkie trzy implementacje algorytmu.

4.1. Implementacja

Jako narzędzie został wybrany język programowania *Java*, dlatego implementacja opiera się na podejściu obiektowym. Język *Java* oferuje wiele możliwości programowania współbieżnego i równoległego. W terminologii *Java* pojęcie wątek nabiera kolejnego znaczenia. Programista ma możliwość tworzenia wielu wątków, które zarządzane są przez JVM (ang. Java Virtual Machine) w celu równoległego wykonywania kodu. Inne dostępne mechanizmy to synchronizacja, blokady (ang. locks), semafory, interfejsy *Callable* oraz *Future*, zmienne typu *atomic* czy framework *Executor* [17]. Podczas fazy projektowania została przeprowadzona analiza, które z mechanizmów będą konieczne i pomocne w implementacji. Ostatecznie w programie wykorzystane zostały głównie interfejsy *Callable*, *Future* oraz framework *Executor*.

Kod został podzielony na pakiety (ang. package), które grupują klasy zgodnie z ich przeznaczeniem oraz funkcjonalnością. Zostało wydzielonych pięć pakietów: `classifier`, `concurrent`, `table`, `tree` oraz `utils`. Poniżej omówione zostaną tylko wybrane klasy, które są kluczowe z punktu widzenia analizy równoległości oraz wydajności programu.

4.1.1. Współdzielone fragmenty kodu

Jedną z głównych klas jest klasa `DecisionTable`. Zawiera ona macierz `String[][] table`, która przetrzymuje rekordy ze zbioru danych, na podstawie których utworzone zostaje drzewo decyzyjne. W celu odnalezienia atrybutu dzielącego niezbędne jest wykonanie wielu obliczeń, które wymagają wstępnego uporządkowania danych np. odnalezienia indeksów w tabeli, w których występują takie same wartości dla wybranego atrybutu a_i. Dostęp do części z uporządkowanych danych potrzebny jest kilkukrotnie w różnych fragmentach algorytmu. Z tego powodu zdecydowano o utworzeniu metadanych. Koncepcja metadanych dotyczących zbioru

danych opiera się na jednokrotnym przetworzeniu całej tabeli z rekordami i zgromadzeniu wszystkich niezbędnych informacji, które w późniejszej fazie algorytmu będą koniecznie do ustalenia atrybutu dzielącego.

Do metadanych które zawiera klasa `DecisionTable` należą następujące pola: `Map<String, Integer> decisionsFrequencies` – mapa gromadzi informacje o częstości wystąpień unikalnych wartości atrybutu decyzyjnego oraz `AttributeValues[] attributesValues` – tablica przetrzymująca instancje klasy `AttributeValues`. Każdemu atrybutowi a_i (oprócz ostatniego, atrybutu decyzyjnego) odpowiada jedna instancja w tablicy `attributesValues` przetrzymywana na miejscu o $i-1$ indeksie, czyli takim samym jak indeks kolumny w tablicy `table` zawierającej wszystkie wartości atrybutu i. Na przykładzie tabeli 1 dla atrybutu a_1 zostałaby utworzona instancja klasy `AttributeValue`, która umieszczona byłaby w tablicy `attributesValues` na miejscu o indeksie 0.

Klasa `AttributeValues` rozszerza klasę typu `HashSet<AttributeValue>` a więc jest zbiorem obiektów typu `AttributeValue`. Zbiór zawiera tyle elementów, ile istnieje unikalnych wartości dla każdego atrybutu. Zadaniem klasy `AttributeValue`, której szkielet przestawiony jest na listingu 1, jest przetrzymywanie informacji powiązanych z jedną wartością atrybutu.

Listing 1: Skrócona implementacja klasy AttributeValue

```
public class AttributeValue {

    private final String value;
    private final Set<Integer> indices;
    private final Map<String, Integer>
                            decisionFrequenciesPerValue;

    public int getTotalFrequency() {...}

    public Collection<Integer> getFrequencies() {...}

    public AttributeValue(String value) {...}

    public void addIndex(int index) {...}

    public void addDecisionClass(String decisionClass) {...}
}
```

Pole `String value` przechowuje jedną, unikalną wartość atrybutu. Zbiór `Set<Integer> indices` zawiera indeksy, na których występują wartości równe polu `value`. Z kolei mapa `Map<String, Integer> decisionFrequenciesPerValue` jako klucz przyjmuje wartość atrybutu decyzyjnego, a jako wartość przechowuje częstość jego wystąpień, jednak tylko dla tych indeksów w tabeli `table`, które zawierają się w zbiorze `indices`.

Listing 2 przedstawia jak prezentowałyby się metadane dla atrybutu a_1 z tabeli 1. Atrybut a_1 posiada dwie unikalne wartości: $v_{1,1}$ oraz $v_{1,2}$. Skutkuje to stworzeniem dwóch instancji klasy `AttributeValue`.

Listing 2: Metadane na przykładzie atrybutu a_1 z tabeli 1

```
System.out.println(attributesValues[0].toString());
# output:
# [ AttributeValue( value=v11,
#                   indices=[0, 2],
#                   decisionFrequenciesPerValue={c1=2} ),
#   AttributeValue( value=v12,
#                   indices=[1, 3, 4],
#                   decisionFrequenciesPerValue={c3=2, c2=1} ) ]
```

Tak przygotowane metadane zapewniają dostęp do wszystkich potrzebnych wartości, aby obliczyć zrównoważony zysk informacji dla każdego atrybutu znajdującego się w tabeli `table`.

W pakiecie `tree` znajduje się kolejna istotna klasa `Node`. Klasa odwzorowuje węzeł drzewa decyzyjnego. Wartość etykiety węzła `String label` wskazuje numer atrybutu wybranego jako atrybut dzielący w węźle. Jeśli węzeł jest liściem, wówczas etykieta wskazuje na klasę decyzyjną. Etykieta gałęzi `String branchLabel` wskazuje na gałąź łączącą węzeł ze swoim rodzicem. Przechowuje więc ona jedną z wartości atrybutu dzielącego wybranego w węźle rodzica. Jeśli węzeł jest korzeniem, wówczas etykieta gałęzi równa się `null`. Lista węzłów `List<Node> children` przechowuje listę potomków węzła. Dzięki niej, możliwe jest poruszanie się po drzewie od korzenia w kierunku liści. Ostatnim polem zdefiniowanym w węźle jest tabela `DecisionTable decisionTable`. Podczas tworzenia węzła korzenia tabela decyzyjna przechowuje wszystkie rekordy, na podstawie których tworzone jest drzewo decyzyjne. Każdy podział drzewa skutkuje formowaniem się mniejszych tabel, które zawierają tylko część rekordów. W konsekwencji każdemu węzłowi przypisana jest tabela zawierająca pewien podzbiór rekordów. Podzbiór zawiera te rekordy, których wartość dla atrybutu dzielącego węzeł rodzica jest równa etykiecie gałęzi, łączącej potomka z rodzicem.

4.1.2. Wykonanie sekwencyjne

Wykorzystując specyfikę języka *Java* można potraktować sekwencyjną implementacje algorytmu jako bazę dla pozostałych dwóch, równoległych implementacji (mechanizm dziedziczenia).

Ujednoliconym warunkiem stopu tworzenia drzewa decyzyjnego jest sytuacja, gdy maksymalna wartość zrównoważonego przyrostu informacji w węźle jest równa 0. Listing 3 przedstawia metodę, która umożliwia rozbudowę drzewa decyzyjnego do momentu, gdy warunek stopu zostanie spełniony. Na listingu widać, że budowa drzewa odbywa się przy użyciu rekurencji `createTree(child)`. Obliczenia dla każdego atrybutu (od atrybutu o najmniejszym indeksie do atrybutu o najwyższym indeksie) jak i tworzenie węzłów potomków (rozbudowując drzewo od lewej strony do prawej) wykonywane są sekwencyjnie.

Listing 3: Rekurencyjne tworzenie drzewa decyzyjnego

```
protected static void createTree(Node node) {
    DecisionTable table = node.getDecisionTable();

    if (table.getMaxGainRatio() == 0) {
        node.setLabel(table.getDecisionClass());
            return;
    }

    node.setLabel(table.getAttributeToDivideBy());

    for (AttributeValue metadata :
                        table.getValuesOfDividingAttribute()) {
        Node child = new Node(
            metadata.getValue(),
            table.getSubTable(metadata.getIndices())
        );
        node.addChild(child);
        createTree(child);
    }
}
```

4.1.3. Równoległość na poziomie atrybutów

Pierwsza równoległa implementacja polega na podziale zadań obliczenia zrównoważonego przyrostu informacji pomiędzy oddzielne wątki. Jest to podejście które wykorzystuje równoległość danych. Dane dzielone są pionowo (wertykalnie).

Klasa `ParallelDecisionTable` rozszerza klasę bazową `DecisionTable`. Została w niej nadpisana tylko jedna metoda `int findAttributeToDivideBy()`, której celem jest odnalezienie atrybutu dzielącego. W metodzie tworzona jest lista zadań obliczenia zrównoważonego przyrostu informacji `List<ComputeGainRatioTask> taskList`. Zadanie tworzone jest osobno dla każdego atrybutu i zdefiniowane jest w klasie `ComputeGainRatioTask`, która implementuje interfejs `Callable`. Klasa w konstruktorze przyjmuje argumenty zawierające dane niezbędne do wykonania obliczeń: indeks atrybutu, dla którego definiowane jest zadanie, lista częstości wystąpień wartości klas decyzyjnych w tabeli decyzyjnej, liczba rekordów w tabeli decyzyjnej oraz zbiór obiektów typu `AttributeValue` tworzących metadane o przetwarzanym atrybucie. W klasie zdefiniowane są 4 prywatne metody, które służą obliczeniu wartości informacji `double getInfo()`, przyrostu informacji `double getGain()`, wyznaczeniu informacji o podziale `double getSplitInfo()` oraz obliczeniu zrównoważonego przyrostu informacji `getGainRatio()`. Jedna metoda publiczna `GainRatioResult call()`, której implementacja wymuszona jest przez interfejs `Callable`, umożliwia start wykonywania zadania przez wątek, któremu zostało ono przydzielone.

Klasa `ExecutorService` udostępnia metodę przyjmującą jako parametr listę zadań, które mają zostać wykonane równolegle. Podczas tworzenia instancji klasy `ExecutorService` należy zdefiniować liczbę wątków, które będą przetwarzać listę zadań. Implementacja opiera się na wzorcu *Work Pool* 2.4.4. Zadania z listy przekazywane są wolnym wątkom do momentu, aż wszystkie zadania zostaną rozdzielone i wykonane. Istotny jest fakt, iż dostęp do rezultatów jest dostępny dopiero wtedy, gdy wszystkie wątki zakończą prace. Taki mechanizm nazywany jest blokującym. W momencie, gdy wszystkie wyniki zgromadzone zostaną w formie listy, wybierany jest ten atrybut, dla którego obliczony zrównoważony przyrost informacji jest największy. W ten sposób odnajdywany jest atrybut dzielący.

4.1.4. Równoległość na poziomie węzłów

Różnica pomiędzy pierwszą a drugą równoległą implementacją, polega na zmianie definicji zadania równoległego. W tym przypadku, zadanie polega na stworzeniu jednego węzła. Zadanie zdefiniowane jest w klasie `CreateNodeTask`, która również implementuje interface `Callable`. Klasa zawiera tylko jedną metodę `Node call()`, jednak większość równole-

głej implementacji zawiera się w klasie `ParallelNodesDecisionTree`, która rozszerza klasę `DecisionTree`. W klasie tej została nadpisana jedna metoda `createTree(Node node)`. Warunek zatrzymania tworzenia drzewa decyzyjnego pozostaje bez zmian. Różnice w implementacji widoczne są podczas tworzenia węzłów potomnych.

Lista zawiera zadania utworzenia każdego z węzłów potomnych (bez kolejnego pokolenia potomków). Lista zadań tworzona jest oddzielnie dla każdego węzła rodzicielskiego.

Listing 4: Lista zadań równoległego tworzenia węzłów

```
List<Callable<Node>> taskList = new ArrayList<>();

for (AttributeValue attributeValue :
                        table.getValuesOfDividingAttribute()) {
    CreateNodeTask createNodeTask = CreateNodeTask.builder()
            .decisionTable(table)
            .attributeValue(attributeValue)
            .build();

    taskList.add(createNodeTask);
}
```

Podobnie jak przy równoległym przetwarzaniu na poziomie atrybutów, został wykorzystany wzorzec Work-Pool. Mechanizm blokujący ponownie wstrzymuje przejście do kolejnej fazy algorytmu do momentu, gdy wszyscy potomkowie rodzica nie zostaną stworzeni i dodani do listy `children`. Po zakończeniu działania współbieżnej części kodu jako pierwszy zostaje wybrany potomek, który znajduje się na pierwszym miejscu listy `childern`. Na tym etapie rozwój drzewa przenoszony jest na niższy poziom. Potomek staje się rodzicem i procedura tworzenia listy zadań powtarza się. Drzewo rozwijane jest w lewo tak długo, aż któryś z węzłów potomnych stanie się liściem. Wówczas algorytm rekurencyjnie powraca do pominiętych węzłów w celu stworzenia ich potomków.

4.2. Uruchamianie programu

Program został napisany w języku *Java* w wydaniu 11, dlatego aby go uruchomić, konieczna jest instalacja środowiska *Java Runtime Environment (JRE)* w wersji 11 lub wyższej. Wszystkie pliki z kodem źródłowym zostały skompilowane i na ich podstawie stworzony został wykonywalny plik archiwum *Java JAR*.

Program został sparametryzowany tak, aby mógł zostać uruchomiony w każdym wariancie implementacji algorytmu ID3. Niezbędne jest również podanie ścieżki do pliku ze zbiorem danych. Argumenty pozwalają na ustawianie następujących wartości:

`--data-set` – ścieżka do pliku ze zbiorem danych,

`--delimiter` – separator wartości w zbiorze danych,

`--output-file` – ścieżka do pliku, gdzie zapisane zostaną dane o wydajności wykonania programu potrzebne do analizy,

`--threads` – liczbę użytych wątków,

`--test-ratio` – udział procentowy danych losowo wybieranych do zbioru testowego,

`--train-ratio` – udział procentowy danych losowo wybieranych do zbioru treningowego, wartość wraz z `--test-ratio` muszą być równe 1,

`--type` – wersja algorytmu.

Aby uruchomić program w wersji sekwencyjnej argument `--type` należy ustawić na wartość `SEQUENTIAL`. W celu zastosowania równoległości na poziomie atrybutów, użytkownik musi podać wartość `PARALLEL_ATTRIBUTES`. Ustawienie argumentu na wartość `PARALLEL_NODES` spowoduje uruchomienia algorytmu w wersji z równoległością węzłów.

Listing 5: Przykładowe uruchomienie programu

```
java -jar concurrent-id3.jar
    --data-set ./resources/data.csv\
    --delimiter ,\
    --output-file ./results.txt\
    --threads 5\
    --test-ratio 0.1\
    --train-ratio 0.9\
    --type PARALLEL_NODES
```

5. Analiza wyników

Rozdział piąty zawiera opis analizy algorytmów pod kątem ich wydajności. Poruszone zostały tematy związane z czasem trwania poszczególnych wykonań algorytmu, zużyciem pamięci RAM, pamięci przydzielonej procesowi i liczbę wykorzystanych wątków. Analiza oparta jest na danych zgromadzonych dzięki przeprowadzonym testom. Wyniki pomiarów zostały przedstawione w formie wykresów stworzonych przy użyciu języka *Python* oraz biblioteki *Plotnine*, która umożliwia korzystanie z pakietu *ggplot2*.

5.1. Środowisko testowe

Większość testów zostało przeprowadzonych na komputerze z 4 rdzeniowym procesorem Intel Core i7-7700HQ z 16GB pamięci RAM. Dodatkowo, do przeprowadzania jednego typu testów, użyto komputera z 4 rdzeniowym procesorem AMD Ryzen 5 2500U z 8 GB pamięci RAM.

Podczas przeprowadzania testów wydajnościowych istotne jest zachowanie podobnych warunków środowiska testowego. Nie jest możliwe całkowite wyłączenie innych aplikacji, ponieważ o działaniu wielu z nich decyduje system operacyjny. Jednakże w celu minimalizacji obciążenia procesora innymi procesami, wszystkie możliwe do zatrzymania programy zostały wyłączone.

Dodatkową techniką, która miała na celu zmniejszenie wpływu zewnętrznych czynników na rezultaty testów, było zastosowanie uśredniania wyników. Dla każdej konfiguracji parametrów program został uruchomiony kilkukrotnie. Zebrane wyniki zostały uśrednione, aby wygenerowane wykresy stały się przez to bardzo wiarygodne. Ręczne uruchamianie programu z wiersza poleceń jest mało optymalne, dlatego cały proces zbierania rezultatów testów został zautomatyzowany. W celu wielokrotnego uruchomienia programu dla różnych parametrów, zostały utworzone skrypty w języku *Bash*.

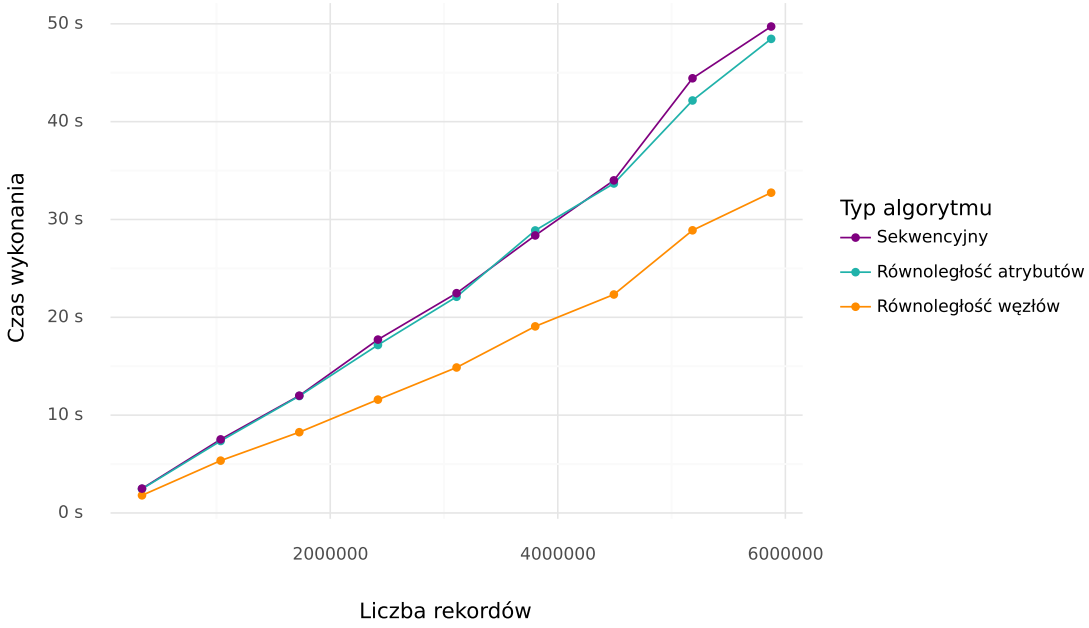
5.2. Kryteria testów wydajnościowych

W przypadku implementacji algorytmu ID3, głównym celem zastosowania przetwarzania wielowątkowego było skrócenie czasu budowy drzewa decyzyjnego. Analizowanie wydajności w pierwszej kolejności wymaga ustalenia, czy stworzony program jest w stanie dostarczać poprawne wyniki. W celu walidacji poprawności kluczowych elementów implementacji, program został przetestowany za pomocą testów jednostkowych. Klasy zawierające logikę, która wpły-

wa na strukturę drzewa, zostały przetestowane na danych, dla których znany był poprawny kształt drzewa. Dzięki uzyskaniu pozytywnych wyników testowania algorytmu pod względem poprawności, podczas analizy kluczowej dla tematu pracy, cała uwaga została skupiona już tylko na badaniu wydajności. W dalszej części rozdziału omówione zostały kryteria testów przeprowadzonych pod kątem wydajności poszczególnych implementacji algorytmu ID3.

5.2.1. Rozmiar danych

Czynnikiem, który ma istotny wypływ na czas konstruowania klasyfikatora jest rozmiar danych. Podczas implementowania algorytmów, często korzysta się z małych zbiorów danych, które ułatwiają testowanie ich poprawności. Dopiero w momencie, gdy algorytm zostanie uruchomiony dla rzeczywistych danych, istnieje możliwość sprawdzenia, czy program jest w stanie efektywnie przetwarzać dane i dostarczyć ostateczne rozwiązanie. Rysunek 10 przedstawia, jak zmienia się całkowity czas potrzebny na konstrukcje klasyfikatora, w zależności od liczby rekordów w zbiorze danych. Na wykresie ujęto trzy rodzaje algorytmu: wersja sekwencyjna, zrównoleglanie atrybutów oraz zrównoleglanie węzłów.



Rysunek 10: Zależność czasu od liczby rekordów.
Źródło: Opracowanie własne.

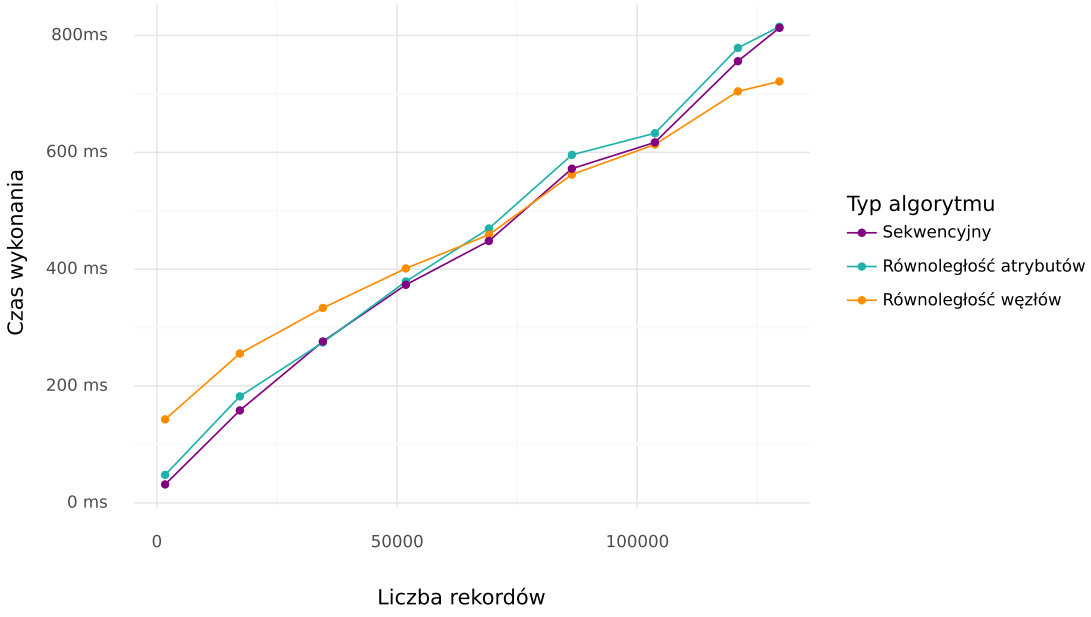
Przy niewielkiej liczbie rekordów czas trwania wszystkich trzech implementacji algorytmu ID3 jest prawie identyczny. Wraz z rosnącą liczbą rekordów, najlepsze rezultaty osiąga

algorytm, gdzie zastosowano równoległość węzłów. Zastosowanie algorytmu z równoległością atrybutów nie przynosi znaczących korzyści. Czas budowy klasyfikatora, bez względu na liczbę rekordów, praktycznie nie różni się od czasu osiąganego dla implementacji sekwencyjnej.

Po przeanalizowaniu rozwiązania z równoległym przetwarzaniem atrybutów, zostały wyciągnięte następujące wnioski. Implementacja jest mało efektywna bez względu na rodzaj i wielkość zbioru danych. Prawdopodobną przyczyną niskiej wydajności implementacji jest zastosowanie konceptu metadanych. Jak opisano w rozdziale 4.1.1, celem tworzenia metadanych było zredukowanie liczby powtarzanych obliczeń. Wiąże się to z tym, że najbardziej obciążające obliczenia dokonywane są w momencie tworzenia metadanych. Z tego powodu, obliczanie zrównoważonego przyrostu informacji dla każdego atrybutu jest działaniem mało złożonym obliczeniowo. Koszt powoływania do życia wątków oraz obsługi równoległości jest większy niż korzyść płynąca z równoległego przetwarzania atrybutów.

Inaczej sytuacja prezentuje się w przypadku równoległego przetwarzania węzłów. Tutaj równolegle tworzone są całe węzły. Operacje, które wykonywane są podczas tworzenia węzła to tworzenie metadanych jak i obliczanie zrównoważonych zysków informacji. Wspólnie, działania wymagają sporo mocy obliczeniowej i dlatego zastosowanie równoległości jest uzasadnione. Takie rozwiązanie sprawia, że implementacja jest skuteczna.

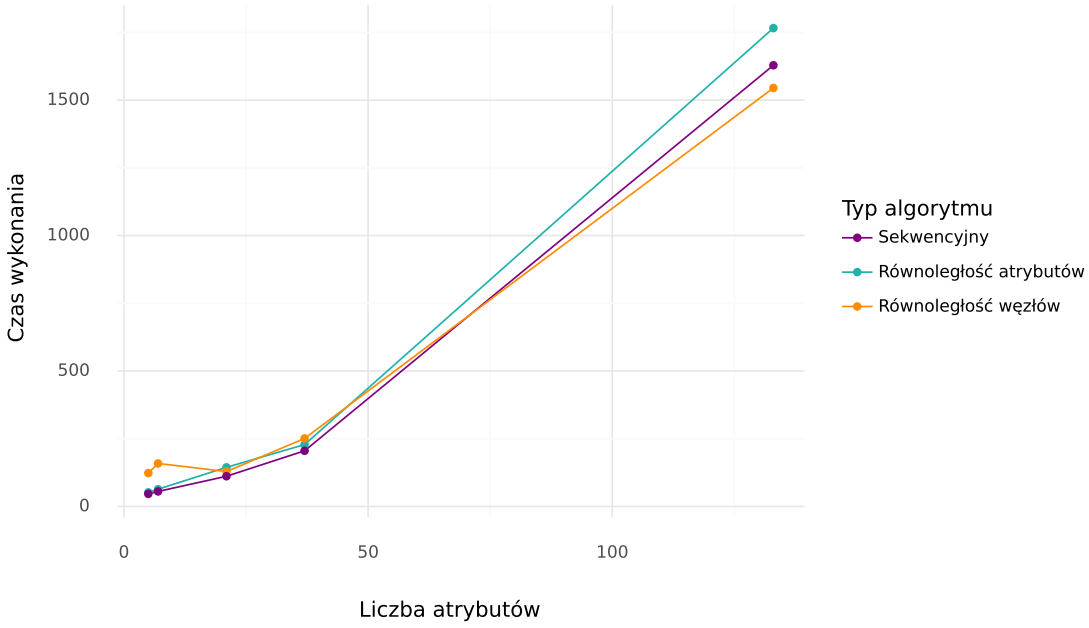
Sytuacja odwraca się w przypadku przetwarzania małych zbiorów danych. Niewielkie bloki danych nie wymagają dużej mocy obliczeniowej nawet w przypadku tworzenia metadanych.



Rysunek 11: Zależność czasu od liczby rekordów przy małym obciążeniu procesora. Źródło: Opracowanie własne.

Rysunek 11 ponownie prezentuje zależność czasu wykonania od liczby rekordów, jednak na przykładzie zbiorów danych o małych rozmiarach. Liczba rekordów wacha się od 120 do 140000. Im mniej rekordów jest przetwarzanych, tym bardziej efektywny jest algorytm sekwencyjny.

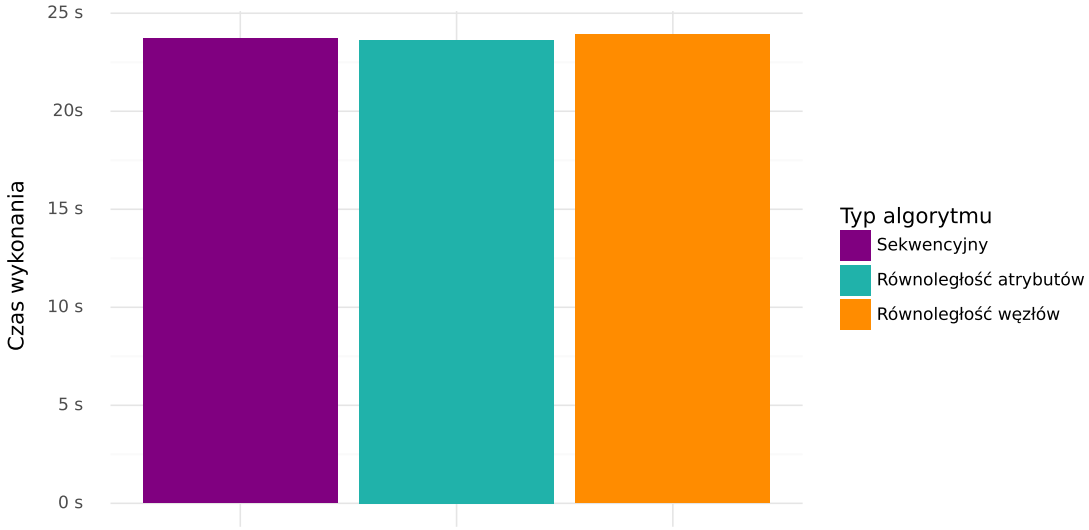
Kolejną składową wpływającą na wielkość zbioru danych jest liczba atrybutów. Im więcej atrybutów posiada rekord, tym więcej obliczeń zrównoważonego przyrostu informacji musi zostać wykonanych na poziomie każdego węzła. Rysunek 12 przedstawia zależność czasu wykonania algorytmu od rosnącej liczby atrybutów. Zaprezentowane wyniki zostały utworzone na podstawie zbiorów danych, z których każdy zawierał po 3000 rekordów.



Rysunek 12: Zależność czasu od liczby atrybutów.
Źródło: Opracowanie własne.

Podobnie jak w przypadku zwiększania liczby rekordów, dla niewielkiej liczby atrybutów, algorytm ze zrównolegleniem węzłów okazał się być mniej efektywny niż pozostałe. Moment wyrównania przypada na około 20 atrybutów. Przy stopniowym zwiększaniu liczby atrybutów widać, że zachodzą różnice w stosunku do zwiększania liczby rekordów. Na wykresie można zauważyć, że wydajność algorytmów równoległych nie ulega wyraźniej poprawie. Przy 130 atrybutach czas wykonania programu, gdzie równolegle przetwarzane są węzły jest niewiele lepszy od pozostałych. W celu wyciągnięcia wniosków, konieczne było sprawdzenie jak kształtuje się wykres dla większej liczby atrybutów. Znalezienie zbiorów danych nadających się do testowania algorytmu ID3, w których liczba atrybutów będzie wzrastała proporcjonal-

nie, nie jest łatwym zadaniem. Z tego powodu, wykres na rysunku 13 przedstawia zależności tylko dla jednego, największego pod względem liczby atrybutów zbioru danych, który udało się uzyskać do celów badania (13 200 atrybutów).

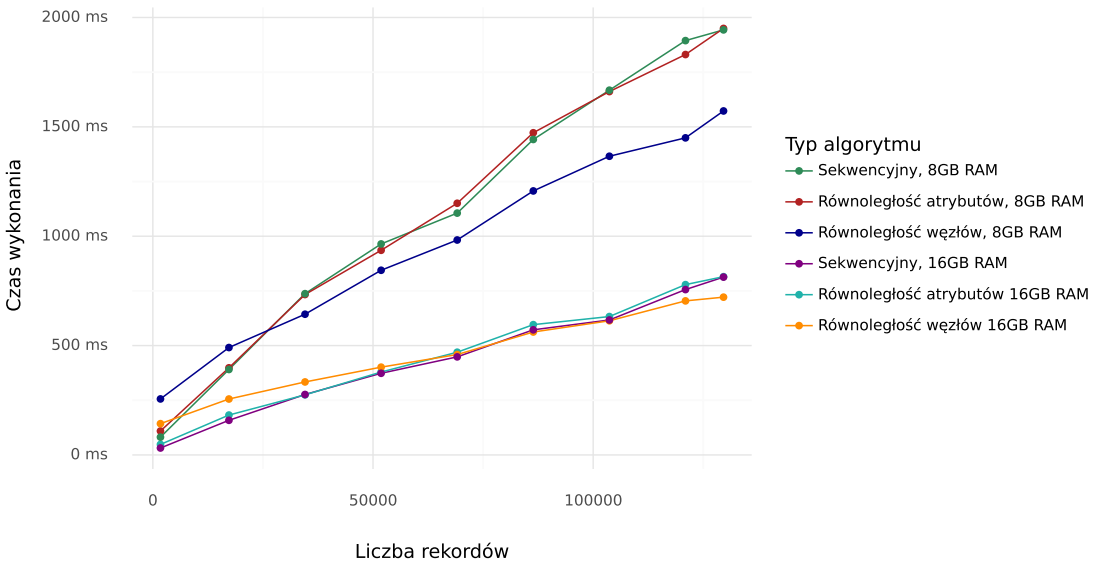


Rysunek 13: Czas wykonania programu dla danych o 13 200 atrybutach. Źródło: Opracowanie własne.

Wykres na rysunku 13 daje podstawy do stwierdzenia, że żadna ze stworzonych implementacji równoległych nie jest optymalna, gdy obciążenie procesora spowodowane jest dużą liczbą atrybutów.

5.2.2. Ilość pamięci RAM oraz pamięci Heap Space

Kolejnym czynnikiem ważnym podczas przeprowadzania analizy wydajności programu jest dostępna pamięć. Pamięć operacyjna RAM pełni funkcje pośrednika pomiędzy pamięcią tymczasową a dyskami twardymi. W pamięci tej przechowywane są m.in.: informacje na temat działających programów. Procesor wykorzystuje informacje przechowywane w pamięci RAM w celu zarządzania procesami. Większa ilość pamięci RAM korzystnie wpływa na szybkość i płynność działania programów [19]. Z powodu braku możliwości zwiększenia ilości pamięci RAM w komputerze, na którym przeprowadzane były testy, zdecydowano, że przeprowadzona zostanie serie testów na oddzielnym sprzęcie o innych parametrach. Program został uruchomiony dla jednakowego zbioru danych na dwóch różnych komputerach z dostępnymi 8GB oraz 16 GM pamięci RAM. Wykres 11 został rozszerzony o rezultaty zgromadzone podczas testów na komputerze z 8GB pamięci RAM. Wyniki zaprezentowane są na rysunku 14.



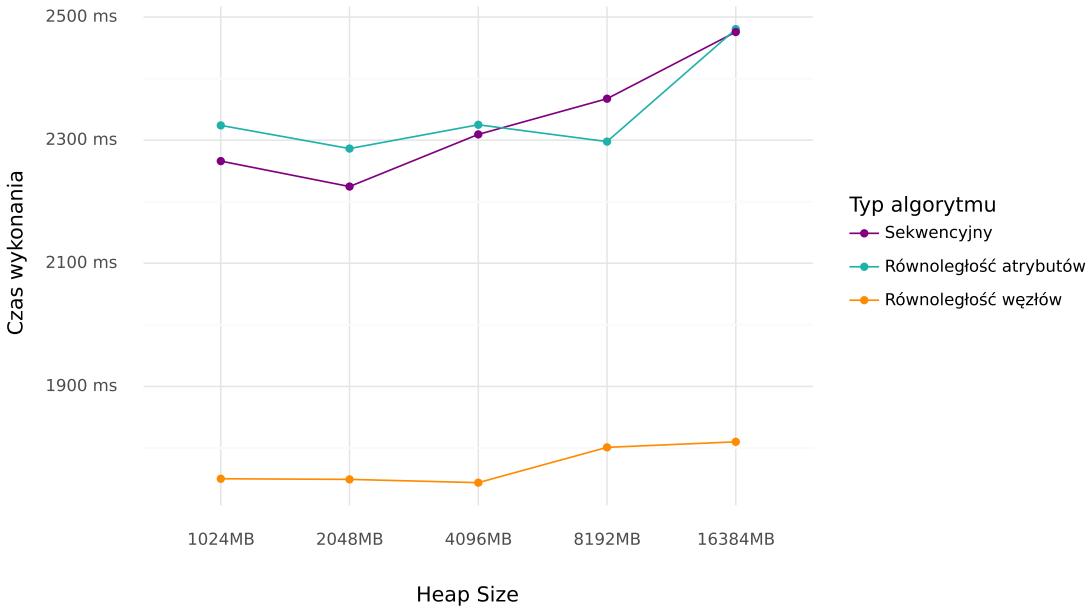
Rysunek 14: Zależność czasu od liczby rekordów i dostępnej pamięci RAM. Źródło: Opracowanie własne.

Mniejsza ilość pamięci RAM znacząco wpływa na czas potrzebny do stworzenia klasyfikatora. Prawie każde wykonanie algorytmu na komputerze z dostępnymi 16GB pamięci RAM było szybsze niż jakiekolwiek wykonanie algorytmu na komputerze z 8GB pamięci RAM.

Czasy wykonań poszczególnych implementacji algorytmu ID3, są bardziej zróżnicowane dla rezultatów osiągniętych na komputerze z 8GB pamięci. Nie jest to jednak dowód na to, że zrównoleglanie węzłów jest bardziej efektywne na komputerze o słabszych parametrach. Mniejsza ilość pamięci RAM skutkuje dłuższym czasem wykonania algorytmu. Jak pokazano w pierwszej części analizy, stosowanie równoległości jest skuteczniejsze, gdy procesor jest bardziej obciążony. Z tego powodu, wykres sprawia wrażenie, że algorytm wykonywany na słabszym sprzęcie działa bardziej efektywnie. Dla każdego komputera można ustalić moment, w którym stosowanie równoległości przynosi korzyści. Jest to moment przecięcia się lini odpowiadającej wykonaniu równoległemu z linią odpowiadającą wykonaniom sekwencyjnym. Dla słabszego komputera moment ten przypada na około 120 tys. rekordów. Dla mocniejszego komputera jest to około 5 tys. rekordów.

Kolejnym typem pamięci uwzględnionym podczas analizy wydajności jest pamięć nazywana stertą (ang. Heap Space). Jest to wydzielona część pamięci RAM, która przydzielona jest maszynie wirtualnej *Javy* w czasie jej startu. Podczas uruchomiania programu, napisanego w języku *Java*, istnieje możliwość konfiguracji ilości pamięci RAM przydzielonej maszynie wirtualnej. Parametr `-Xmx` umożliwia konfigurację maksymalnej wartości sterty (ang. Heap

Size). Przykładowo dodając parametr `-Xmx4096m` maksymalna wartość *Heap Size* wynosi 4096 megabajtów. Powiększanie *Heap Size* ma sens, jeśli do pamięci programu ładowane są bardzo duże ilości danych. Jeśli wartość pozostanie na niskim poziomie, wówczas pamięć zostaje przepełniona, w wyniku czego program zostaje przerwany z powodu błędu typu `java.lang.OutOfMemoryError` [20].



Rysunek 15: Zależność czasu od wartości *Heap Size*.
Źródło: Opracowanie własne.

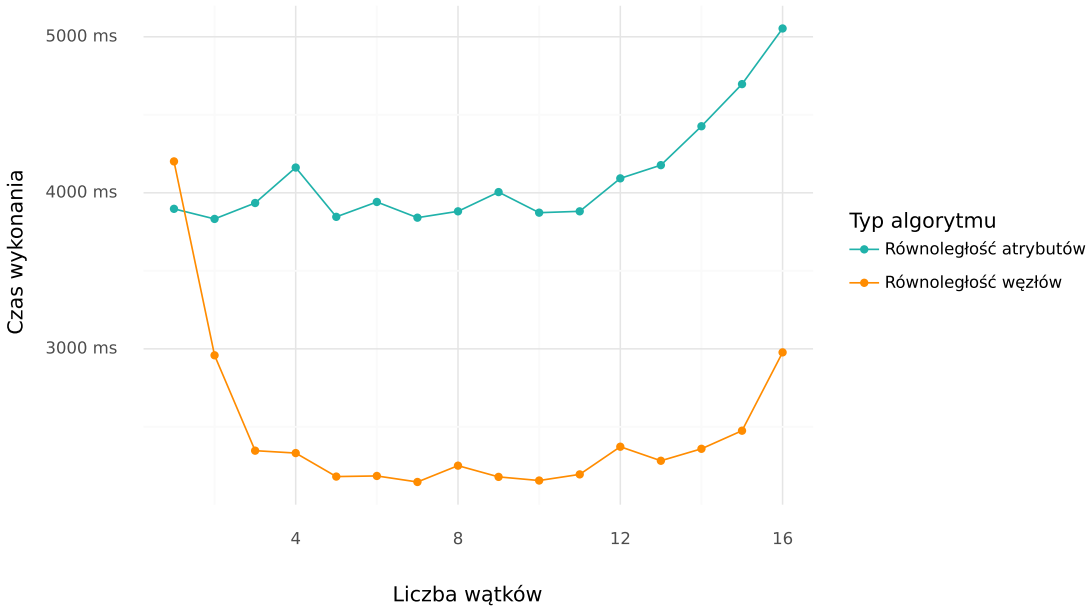
W oparciu o rysunek 15 można stwierdzić, że w odróżnieniu do pamięci RAM, zwiększanie wartości *Heap Size* nie wpływa bezpośrednio na poprawę wydajności którejkolwiek implementacji algorytmu. Pomiary zostały wykonane dla jednego zbioru danych, dla którego bez problemu można było uruchomić algorytm nawet przy niskiej wartości *Heap Size*. Wraz ze wzrostem wartości *Heap Size* nastąpiło nieznaczne pogorszenie efektywności działania programu. Może ono wynikać z tego, że uruchomianie maszyny wirtualnej *Javy* z większą ilością przydzielonej pamięci, jest bardziej czasochłonnym procesem.

5.2.3. Liczba wątków

Ostatnim przeanalizowanym zagadnieniem był wpływ liczby tworzonych wątków na czas trwania pracy programu. Podczas tej analizy, pojęcie wątek używane jest w kontekście wątków tworzonych w ramach maszyny wirtualnej *Javy*. Każdy wątek tworzony przez *JVM* związany jest z klasą `java.lang.Thread`. Tworzenie instancji wymienionej klasy rozpoczyna cykl życia

wątku. Dopuszczalne stany, w których może znajdować się wątek to m.in.: `RUNNABLE`, `BLOCKED` czy `WAITING`. Programista może zarządzać liczbą wątków poprzez jawne tworzenie instancji klasy `Thread` lub też może skorzystać z interfejsu `Executor`, w ramach którego tworzone są pule wątków [21].

Rysunek 16 przedstawia wyniki testów przeprowadzonych pod kątem zmiennej liczby przydzielanych wątków.



Rysunek 16: Zależność czasu od liczby przydzielonych wątków.
Źródło: Opracowanie własne.

Liczba wątków, którą należy użyć w programie *Java* zależy od wielu czynników. Największe znaczenie mają zasoby sprzętowe, charakter oraz obciążenie programu. Zaleca się stosowanie liczby wątków równej lub trochę większej od liczby dostępnych rdzeni logicznych [21]. Komputer, na którym przeprowadzano testy posiada 4 rdzenie fizyczne co w konsekwencji daje do dyspozycji 8 rdzeni logicznych. Na grafice 16 widać, że zbyt mała liczba wątków istotnie wydłuża czas pracy programu. Podobnie zbyt duża liczba wątków opóźnia moment zakończenia pracy programu. Przyczyną takiej tendencji, są ograniczone zasoby systemu operacyjnego, które umożliwiają efektywne przetwarzanie tylko ograniczonej liczby równoległych operacji. Powoływanie do życia nowego wątku jest kosztowną operacją, dlatego tworzenie wątków w zbyt dużej jak i zbyt małej liczbie, negatywnie wpływa na efektywność programu.

5.3. Zaistniałe problemy

Pierwotnie program został zaimplementowany z wykorzystaniem uproszczeń jakie oferuje język *Java* w wersji 11 jak np. strumienie (*Java Stream API*). Interfejs strumieni umożliwia tworzenie bardziej spójnego i czytelnego kodu. Korzystanie z udogodnień może wiązać się jednak ze skutkami ubocznymi. Obliczenia na dużych zbiorach danych znacznie obciążają procesor, co skutkuje zwiększeniem czasu potrzebnego na wykonanie programu. Z tego powodu, w celu poprawy wydajności wprowadzono poprawki takie jak zastąpienie list przez tablice (tam, gdzie to możliwe), a także rezygnacja z użycia strumieni i zamiana ich na pętle for.

Kolejnym problemem, który został zaobserwowany była gwałtownie rosnąca ilość wykorzystanego *Heap Space*. Język *Java* oferuje mechanizm usuwania nieużytków *Garbage Collection*. Oznacza to, że programista nie zarządza samodzielnie przydzielaniem jak i zwalnianiem pamięci. Za te czynności odpowiedzialny jest program nazywany *Garbage Collector*. Podstawowe zadania programu to alokacja pamięci oraz odzyskiwanie jej dzięki usuwaniu obiektów, do których nie istnieją już referencje. Pomimo tego, iż programista nie jest bezpośrednio odpowiedzialny za zarządzanie pamięcią, istnieje kilka typów błędów w implementacji, które mogą powodować wycieki pamięci (ang. Memory Leaks). Wycieki powodują przepełnienie pamięci, co skutkuje przerwaniem pracy programu. Dzięki użyciu narzędzi takich jak *Java VisualVM* możliwe jest generowanie wykresów wykorzystania stosu w trakcie działania programu. W celu rozwiązania problemu, podjęte zostały próby odnalezienia ewentualnych miejsc wycieków pamięci. Wykresy analizy wykonanej narzędziem *Java VisualVM* pokazały, że ilość zaalokowanej pamięci rosła wraz z czasem trwania programu, a *Garbage Collector* nie miał możliwości zwalniania jej dostatecznie szybko. Najprawdopodobniej kluczową rolę w zaistniałym problemie odgrywa wykorzystanie rekurencji, która nie jest optymalna pod względem zajmowanej pamięci. Dodatkowym czynnikiem może być konieczność ładowania do pamięci programu dużych zbiorów danych. Próbą zmniejszenia zużycia pamięci na stosie było zaimplementowanie współdzielenia danych. Zamiast duplikacji części rekordów, przekazywanych do węzłów potomnych, przekazywane były tylko numery indeksów, które dotyczyły wybranego węzła. Niestety zaimplementowane poprawki nie przyniosły pozytywnych rezultatów. Podejście dostarczania danych we fragmentach, które zaproponowane zostało w omówionym artykule [14] wydaje się być rozwiązaniem, które mogłoby pomóc pozbyć się problemu przeładowanej pamięci.

Zakończenie

Celem pracy było opracowanie dwóch równoległych implementacji algorytmu ID3 oraz porównanie ich wydajności na tle implementacji sekwencyjnej. Aby zapewnić punkt odniesienia niezbędny do wykonania analizy, jako pierwsza zaimplementowana została klasyczna, sekwencyjna wersja algorytmu. W zaproponowanych równoległych wersjach algorytmu wykorzystane zostały techniki zrównoleglające obliczenia na poziomie atrybutów oraz węzłów. Przyjęta została hipoteza, że implementacje równoległe powinny okazać się bardziej wydajne niż implementacja sekwencyjna.

Wszystkie trzy implementacje zostały porównane pod kątem prędkości wykonania. Przeanalizowane zostało jak zmieniał się czas wykonywania programu w zależności od zmieniającej się liczby atrybutów, rosnącej liczby rekordów, liczby przedzielonych wątków, dostępnej pamięci RAM oraz pamięci typu *Heap Space*. Przeprowadzone testy wykazały, że tylko jedna z równoległych wersji okazała się być bardziej wydajna niż wersja sekwencyjna. Zastosowanie zrównoleglenia na poziomie węzłów pozwoliło na szybszą konstrukcje drzew decyzyjnych, gdy przetwarzana była duża liczba rekordów. Z kolei implementacja, w której zastosowane zostało zrównoleglenie na poziomie atrybutów, nie przyniosła istotnych korzyści w kontekście czasu wykonywania programu. Przyczyną takich rezultatów prawdopodobnie jest wykorzystanie konceptu metadanych. Jego celem było zminimalizowanie liczby wykonywanych działań, które są najbardziej złożone obliczeniowo. Z tego powodu, w implementacji równoległego przetwarzania atrybutów, działania przetwarzane współbieżnie były niewystarczająco złożone obliczeniowo, aby wykorzystanie techniki równoległości przyniosło pozytywny efekt.

Istotną kwestią, która mogłaby zostać poprawiona w ramach dalszego rozszerzania pracy, jest ulepszenie implementacji zrównoleglającej węzły w zakresie zużycia pamięci *Heap Space*. W skrajnych przypadkach, gdy przetwarzana jest duża rekordów lub atrybutów, zbyt duże zużycie pamięci nie pozwala na pomyślne zakończenie konstrukcji drzewa decyzyjnego. Przykładem rozwiązania, które mogłoby zostać zrealizowane w ramach kontynuacji oraz optymalizacji pracy, to technika dostarczania danych we fragmentach, która została przedstawiona w przeanalizowanej literaturze.

Spis tabel

Spis rysunków

Spis listingów

Literatura

[1] David E. Rumelhart & James L. McClelland, PDP Research Group. Parallel Distributed Processing, Vol. 1. A Bradford Book, 1987.

[2] Kozak, J., & Juszczuk, P. (2016). Algorytmy do konstruowania drzew decyzyjnych w przewidywaniu skuteczności kampanii telemarketingowej banku. Studia Informatica Pomerania nr 1/2016 (39). doi: 10.18276/si.2016.39-05

[3] Lior Rokach, Oded Maimon. Data Mining with Decision Trees. Theory and Applications. World Scientific Publishing Company. Israel, 2014.

[4] Ashima Gambhir. Classification Problem in Data Mining Using Decision Trees. LAP LAMBERT Academic Publishing, 2019.

[5] Dariusz Majerek. Eksploracja danych. [Online] `https://dax44.github.io/datamining/drzewa-decyzyjne.html#w%C4%99z%C5%82y-i-ga%C5%82%C4%99zie`. Dostęp: 02.12.2022

[6] Andrzej Karbowski, Ewa Niewiadomska-Szynkiewicz. Programowanie równoległe i rozproszone. Oficyna Wydawnicza Politechniki Warszawskiej. Warszawa, 2009.

[7] Czech Zbigniew J. Wprowadzenie do obliczeń równoległych. Wydawnictwo Naukowe PWN. Wyd. 2, 2013.

[8] Krzysztof Banaś, Skrypt. Programowanie równoległe i rozproszone. Wydział Fizyki, Matematyki i Informatyki Politechniki Krakowskiej. Kraków, 2011.

[9] Grama, A., Anshul Gupta, A., George Karypis, G. & Kumar, V. Introduction to Parallel Computing. Addison-Wesley Professional, 2003.

[10] OpenCSF Project. [Online] `https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/ParallelDesign.html`. Dostęp: 12.12.2022

[11] Amado, N., Gama, J., & Silva, F. (2001). Parallel Implementation of Decision Tree Learning Algorithms. Lecture Notes in Computer Science, 6-13. doi:10.1007/3-540-45329-6_4

[12] Kubota, K., Nakase, A., Sakai, H., & Oyanagi, S. (2000). Parallelization of decision tree algorithm and its performance evaluation. Proceedings Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region. doi:10.1109/hpc.2000.843500

[13] Cal, P., & Woźniak, M. (2013). Parallel Hoeffding Decision Tree for Streaming Data. Advances in Intelligent Systems and Computing, 27-35. doi:10.1007/978-3-319-00551-5_4

[14] Rcega, A. F.-A., Suarez-Cansino, J., & Flores-Flores, L. G. (2013). A parallel algorithm to induce decision trees for large datasets. 2013 XXIV International Conference on Information, Communication and Automation Technologies (ICAT). doi:10.1109/icat.2013.6684045

[15] Maheshwari, S., Jatav, VK., & Meena, YK. (2011). Improved ID3 Decision Tree Generation using Shared-Memory and Multi-Threading Approach. 2011 International Conference on Education Technology and Computer (ICETC 2011). doi:10.13140/2.1.3216.2247

[16] Gu, Y., Shi, G., Cai, H., Chen, Y. & Sun, Y. (2013). Research of Parallel Decision Tree Algorithm Based on Mapreduce. Information Technology Journal, 12: 7345-7352. doi: 10.3923/itj.2013.7345.7352

[17] Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D. & Lea, D. Java Concurrency in Practice. Addison - Wesley, 2006.

[18] Lea, D. Concurrent Programming in Java: Design Principles and Pattern, 2nd Edition. Addision-Wesley, 2000.

[19] Herlihy, M., & Shavit, N. The Art of Multiprocessor Programming, Revised Reprint. Morgan Kaufmann, 2012.

[20] Maaike van Putten & Seán Kennedy. Java Memory Management. Packt Publishing, 2022.

[21] Oaks, S., & Wong, H. Java Threads: Understanding and Mastering Concurrent Programming. O'Reilly, 2004.

[22] Nelson, D. Data Visualization in Python. Stack Abuse, 2021.

[23] McKinney, W. Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython. O'Reilly, 2018.

[24] Blum, R., & Bresnahan, C. (Author). Linux Command Line and Shell Scripting Bible 4th Edition. Wiley, 2021.

[25] Newham C. Learning the bash Shell: Unix Shell Programming. O'Reilly, 2005.