

POLITECHNIKA CZĘSTOCHOWSKA
WYDZIAŁ INFORMATYKI I SZTUCZNEJ INTELIGENCJI
KATEDRA INFORMATYKI

ROZPRAWA DOKTORSKA

Metody i algorytmy adaptacji obliczeń uczenia maszynowego i aplikacji numerycznych do wysokowydajnych systemów komputerowych

mgr inż. Paweł Bratek

Praca wykonana pod kierunkiem:
dr hab. inż. Łukasza Szustaka, prof. PCz

Częstochowa, 2025

Spis treści

Streszczenie w języku polskim	5
Streszczenie w języku angielskim	8
1. Wstęp	10
1.1 Tło badań	10
1.2 Teza i cel pracy	14
1.3 Struktura pracy	15
2. Metoda redukcji zużycia energii elektrycznej dla aplikacji zrównoleglanych w modelu równoległości danych	18
2.1 Aplikacje zrównoleglane w modelu równoległości danych	18
2.1.1 Problem trójwymiarowej dyfuzji	18
2.1.2 MPDATA – wielowymiarowy dodatnio określony algorytm adwekcji	20
2.2 Motywacja do badań i sformułowanie problemu badawczego	21
2.3 Koncepcja opracowanego rozwiązania	27
2.4 Badanie wpływu opracowanej metody na zużycie energii elektrycznej . . .	28
2.5 Algorytmy wyboru heterogenicznej konfiguracji częstotliwości	32
2.5.1 Algorytm bazujący na pomiarze energii elektrycznej	33
2.5.2 Algorytm bazujący na pomiarze czasu	35
2.6 Badanie efektywności opracowanych algorytmów	36
2.7 Ocena kosztu zastosowania opracowanej metody	39
2.8 Weryfikacja opracowanej metody z wykorzystaniem platformy AMD	40
3. Adaptacja realizacji zapytań zliczających do wysokowydajnych systemów komputerowych	43
3.1 Zapytania zliczające w uczeniu maszynowym	43
3.2 Motywacja do badań	48
3.3 SABNAtk: biblioteka open-source do efektywnej realizacji zapytań zliczających	49
3.3.1 Interfejs programistyczny	50
3.3.2 Strategia Bitmap	52

3.3.3	Strategia Radix	56
3.3.4	Realizacja przykładowego zapytania zliczającego	59
3.3.5	Koncepcja zrównoleglenia realizacji zapytań zliczających	61
3.4	Metodologia ewaluacji	63
3.4.1	Platformy obliczeniowe	63
3.4.2	Przygotowanie zbiorów danych	64
3.4.3	Opracowanie benchmarku losowych zapytań	66
4.	Metody poprawy wydajności strategii Bitmap	69
4.1	Metoda redukcji liczby przecięć bitmap (<i>SNIJK</i>)	69
4.2	Metoda redukcji liczby odwiedzanych węzłów drzewa DFS (<i>SNIJ</i>)	73
4.3	Kod źródłowy opracowanych metod	78
4.4	Badanie wpływu opracowanych metod na wydajność zapytań zliczających	78
5.	Metoda automatycznego wyboru strategii	84
5.1	Analiza wydajności strategii realizacji zapytań zliczających	84
5.2	Koncepcja opracowanego rozwiązania	91
5.3	Oszacowanie złożoności strategii realizacji zapytań zliczających	92
5.4	Teoretyczne modele kosztu realizacji zapytania zliczającego	99
5.5	Eksperymentalna weryfikacja modeli kosztu realizacji zapytania zliczającego	103
5.6	Adaptacja opracowanych modeli kosztu realizacji zapytania zliczającego	104
5.7	Algorytm aktualizacji modeli	106
5.8	Algorytm przetwarzania strumienia zapytań przy użyciu opracowanej metody	109
5.9	Kod źródłowy opracowanej metody	114
5.10	Badanie wpływu opracowanej metody na wydajność zapytań zliczających	117
6.	Równoległa realizacja metody automatycznego wyboru strategii	128
6.1	Koncepcja opracowanego rozwiązania	128
6.2	Kod źródłowy opracowanej metody	130
6.3	Badanie wpływu opracowanej metody na wydajność zapytań zliczających	133
7.	Metoda równoległej realizacji zapytań zliczających uwzględniająca architekturę systemów ccNUMA	136
7.1	Koncepcja opracowanego rozwiązania	136
7.2	Kod źródłowy opracowanej metody	140

7.3	Badanie wpływu opracowanej metody na wydajność zapytań zliczających .	141
8.	Podsumowanie oraz kierunki dalszych badań	143
8.1	Wnioski końcowe	143
8.2	Kierunki dalszych prac	148
Aneks		150
Spis rysunków		172
Spis tabel		175
Spis listingów		177
Bibliografia		178

Metody i algorytmy adaptacji obliczeń uczenia maszynowego i aplikacji numerycznych do wysokowydajnych systemów komputerowych

Streszczenie

Praca dotyczy poszukiwania metod poprawy efektywności obliczeniowej i energetycznej uczenia maszynowego oraz aplikacji numerycznych w środowisku wysokowydajnych systemów komputerowych. W pracy uzyskano następujące oryginalne wyniki w tym zakresie:

1. Opracowano metodę redukcji zużycia energii elektrycznej dedykowaną dla aplikacji zrównoleglanych w modelu równoległości danych. Głównym celem opracowanej metody jest złagodzenie skutków nierównomiernej dystrybucji obliczeń pomiędzy dostępne wątki lub rdzenie procesorów, powstającej na skutek zrównoleglania tego typu aplikacji w oparciu o powszechnie stosowane podejścia. Istotą metody jest zarządzanie częstotliwością odpowiednio dobranych grup rdzeni, co pozwala na lepsze dopasowanie parametrów pracy procesora do rzeczywistych obciążeń. Opracowana metoda adresuje problem nierównomiernej dystrybucji obliczeń, pozwalając na istotną redukcję zużycia energii bez jednoczesnej utraty wydajności. Proponowane podejście przewyższa efektywnością standardowo stosowane techniki, takie jak DVFS, zapewniając lepsze dopasowanie do charakterystyki danej aplikacji.
2. Zaproponowano algorytmy wyboru heterogenicznej konfiguracji częstotliwości, umożliwiające praktyczne wdrożenie opracowanej metody redukcji zużycia energii w aplikacjach zrównoleglanych w modelu równoległości danych. Opracowane algorytmy istotnie ograniczają liczbę potrzebnych konfiguracji testowych, przyspieszając proces poszukiwania optymalnej konfiguracji częstotliwości. Pierwszy z algorytmów podejmuje decyzje na podstawie pomiarów zużycia energii, natomiast drugi eliminuje potrzebę takich pomiarów, opierając się jedynie na analizie czasu wykonania aplikacji. Takie podejście sprawia, że opracowana metoda redukcji zużycia energii może być z powodzeniem wdrożona nawet w środowiskach, w których pomiar energii jest utrudniony a nawet niemożliwy.
3. Zaprojektowano oraz zaimplementowano dwie metody poprawy wydajności strategii realizacji zapytań zliczających opartej o reprezentowanie stanów poszczególnych zmiennych zapytania za pomocą bitmap oraz algorytm przeszukiwania

w głąb (DFS, Depth First Search). Pierwsza z zaproponowanych metod ukierunkowana jest na redukcję liczby wykonywanych przecięć bitmap (operacji AND), natomiast druga zmierza do minimalizacji liczby odwiedzanych węzłów drzewa. Opracowane metody zostały wdrożone przez autora niniejszej pracy w bibliotecę SABNAtk, będącej narzędziem służącym do efektywnej realizacji zapytań zliczających.

4. Opracowano metodę automatycznego wyboru strategii realizacji zapytań zliczających, której celem jest skrócenie czasu obsługi strumienia zapytań. Istota podejścia polega na połączeniu zalet trzech komplementarnych strategii — BV, RAD i CT. Metoda wykorzystuje regresję online jako mechanizm decyzyjny, który dla każdego zapytania w strumieniu wybiera strategię o najniższym estymowanym czasie wykonania. Do oryginalnych osiągnięć związanych z opracowaniem tej metody należą:
 - (a) Sformułowanie teoretycznych modeli kosztu realizacji zapytania zliczającego dla trzech różnych strategii, będące efektem dogłębnej analizy poszczególnych podejść. Opracowane modele estymują koszt wykonania zapytania na podstawie właściwości samego zapytania jak i rozmiaru zbioru danych.
 - (b) Opracowanie efektywnego algorytmu aktualizacji modeli kosztu realizacji zapytania zliczającego. Rozwiązanie to bazuje na algorytmach inkrementalnych, umożliwiając efektywną aktualizację statystyk modeli regresji na podstawie pojedynczych nowych obserwacji, bez konieczności gromadzenia dotychczasowych danych i bez narastania kosztu obliczeniowego.
 - (c) Zaprojektowanie algorytmu przetwarzania strumienia zapytań, który definiuje pełny przepływ działania metody — od inicjalizacji i uczenia modeli, przez adaptacyjny wybór strategii, po obsługę zapytań przekraczających ograniczenia pamięciowe, zapewniając spójne i praktyczne ujęcie opracowanego podejścia.
 - (d) Wdrożenie opracowanej metody w oficjalnej wersji biblioteki SABNAtk, co umożliwia jej praktyczne wykorzystanie i udostępnienie szerszemu gronu użytkowników.
5. Opracowano i zaimplementowano w pełni bezpieczną dla przetwarzania wielowątkowego (thread-safe) wersję metody automatycznego wyboru strategii realizacji zapytań zliczających. Wdrożone podejście minimalizuje ilość operacji zapisu współdzielonego oraz wykorzystuje mechanizm muteksów, skutecznie zapobiegając sytuacjom wyścigu pomiędzy wątkami podczas równoległego przetwarzania. Taka

implementacja umożliwia zastosowanie modelu równoległości zadań (task parallelism), w którym zapytania zliczające traktowane są jako niezależne zadania realizowane równolegle na wielu rdzeniach procesora, przy użyciu dowolnego narzędzia (np. OpenMP, Intel TBB).

6. Zaproponowano metodę równoległej realizacji zapytań zliczających, dedykowaną architekturom nowej generacji typu ccNUMA (cache-coherent Non-Uniform Memory Access). Systemy te składają się z wielu domen NUMA i charakteryzują się zróżnicowanymi opóźnieniami dostępu do pamięci w zależności od fizycznej lokalizacji danych względem rdzenia wykonawczego. Współczesne wysokowydajne systemy komputerowe oferują do ośmiu domen NUMA (2×4) w serwerach dwuprocessorowych opartych na procesorach AMD [7]. Opracowane rozwiązanie jest niezależne od wybranej strategii realizacji zapytań zliczających i może być stosowane z dowolnym podejściem, w szczególności z metodą automatycznego wyboru strategii realizacji zapytań zliczających.

Methods and Algorithms for Adapting Machine Learning and Numerical Computations to High-Performance Computing Systems

Summary

This dissertation concerns the search for methods to improve the computational and energy efficiency of machine learning and numerical computations, achieved in the environment of high-performance computing (HPC) systems. The following original contributions have been achieved:

1. a method was developed to reduce energy consumption in applications parallelized under the data parallelism model. The primary goal of this method is to mitigate the effects of load imbalance among threads or processor cores, which typically arises from applying widely used data-parallel approaches. The method is based on managing the frequency of carefully selected groups of cores, enabling a better alignment of processor operating parameters with actual workloads. This approach effectively addresses load imbalance, achieving significant energy savings without performance loss. The proposed solution outperforms commonly used techniques such as Dynamic Voltage and Frequency Scaling (DVFS), offering better adaptation to application-specific characteristics.
2. Algorithms for selecting heterogeneous frequency configurations were proposed, providing a practical implementation of the developed energy reduction method for data-parallel applications. These algorithms significantly reduce the number of required test configurations, accelerating the search for the optimal frequency setup. The first algorithm makes decisions based on energy consumption measurements, while the second eliminates the need for such measurements, relying solely on execution time analysis. This enables practical deployment even in environments where energy measurement is difficult or even impossible.
3. Two methods were designed and implemented to improve the performance of counting queries execution strategies based on representing query variable states with bitmaps and using a Depth-First Search (DFS) traversal algorithm. The first method focuses on reducing the number of bitmap intersections (AND operations), while the second minimizes the number of visited tree nodes. Both methods were implemented in the SABNAtk library, a tool for efficient execution of counting queries.

4. a method for automatic selection of counting queries execution strategies was developed, with the goal of reducing query stream processing time. The approach combines the strengths of three complementary strategies — BV, RAD, and CT — and uses online regression as a decision-making mechanism to select, for each query in the stream, the strategy with the lowest estimated execution time. The original contributions include:
 - (a) Formulation of theoretical cost models for counting query execution under the three strategies, derived from in-depth analysis of their operational principles. These models estimate query execution cost based on query properties and dataset size.
 - (b) Development of an efficient algorithm for incrementally updating the cost models, enabling effective model refinement using new observations without storing past data and increasing computational overhead.
 - (c) Design of a query stream processing algorithm that defines the complete workflow of the method — from model initialization and training, through adaptive strategy selection, to handling memory-bound queries — ensuring consistency and practicality of the approach.
 - (d) Implementation of the method in the official SABNAtk library, enabling its use in real-world scenarios and providing access for a broader community of users.
5. a fully thread-safe implementation of the automatic counting queries execution strategy selection method was designed and implemented. The solution minimizes the number of write operations and employs mutex mechanism to eliminate race conditions during parallel processing. This implementation enables task parallelism, where counting queries are treated as independent tasks executed concurrently across multiple processor cores using any framework (e.g., OpenMP, Intel TBB).
6. a method for parallel execution of counting queries tailored to new ccNUMA (cache-coherent Non-Uniform Memory Access) architectures was proposed. Such systems consist of multiple NUMA domains and exhibit heterogeneous memory access latencies depending on the physical location of data relative to the executing core. Modern high-performance computing systems provide up to eight NUMA domains (2×4) in dual-socket servers based on AMD processors [7]. The proposed method is independent of the specific counting queries execution strategy and can be applied universally, specifically in combination with the automatic strategy selection method.

1. Wstęp

1.1. Tło badań

Intensywny rozwój wysokowydajnych systemów komputerowych (HPC, ang. High Performance Computing) przekłada się na rosnące możliwości rozwiązywania złożonych zagadnień naukowych i inżynierskich. Postęp w dziedzinie HPC w połączeniu z zaawansowanymi algorytmami sztucznej inteligencji, umożliwia rozwiązywanie problemów naukowych o skali i złożoności, które jeszcze niedawno wydawały się niemożliwe do osiągnięcia. Przykładem mogą być rozwiązania z zakresu dużych modeli językowych (LLM, ang. Large Language Model), takie jak ChatGPT, notujące ostatnimi czasy rekordowy wzrost popularności. Narzędzie te stały się istotnym sposobem wyszukiwania informacji w internecie, zmniejszając tym samym ruch odnotowywany bezpośrednio na stronach internetowych [48, 62, 74]. Dynamiczny rozwój dużych modeli językowych nie byłby możliwy bez wysokowydajnych systemów komputerowych, umożliwiających trenowanie modeli, zawierających nawet biliony parametrów [42].

Wykorzystanie pełnego potencjału wysokowydajnych systemów komputerowych nie jest zazwyczaj możliwe bez uwzględnienia specyfiki docelowej platformy obliczeniowej. Tradycyjne algorytmy koncentrują się przede wszystkim na matematycznym aspekcie rozwiązania problemu, przyjmując zwykle sekwencyjny model obliczeń. Takie podejście prowadzi jednak do niedostatecznego wykorzystania możliwości współczesnych architektur wielordzeniowych, oferujących szerokie wsparcie dla przetwarzania równoległego [27, 67, 71]. W literaturze podkreśla się, że osiągnięcie wysokiej wydajności wymaga nie tylko doboru odpowiednich technik równoległych, ale także świadomej adaptacji algorytmów do cech danej architektury, takich jak organizacja jednostek obliczeniowych, hierarchia pamięci czy charakterystyki energetyczne [37, 68, 69, 73].

W niniejszej rozprawie doktorskiej szczególną uwagę poświęcono zagadnieniu adaptacji obliczeń uczenia maszynowego do wysokowydajnych systemów komputerowych. Główny nacisk położono na problematykę zapytań zliczających, które stanowią fundamentalną operację wielu algorytmów uczenia maszynowego. Operacje te polegają na obliczaniu liczby rekordów w zbiorze danych, spełniających określone warunki.

Problem zapytań zliczających pojawia się za każdym razem, gdy konieczne jest oszacowanie rozkładu prawdopodobieństwa, dlatego ma on szerokie zastosowania praktyczne. Przykładowo, zapytania zliczające odgrywają istotną rolę w funkcji oceny wykorzystywanej podczas uczenia struktur sieci bayesowskich [34] oraz w analizie reguł asocjacyjnych, gdzie służą do wyznaczania miar wsparcia i ufności [5]. Do innych zastosowań na-

leży zaliczyć metody klasyfikacji [50], uczenie głębokie [59] oraz systemy wyszukiwania informacji [51].

Wraz ze wzrostem rozmiaru danych analizowanych przez systemy uczenia maszynowego pojawia się wyraźna potrzeba opracowywania efektywnych strategii zliczania. Proces realizacji zapytań zliczających bywa jednak powszechnie traktowany jako procedura typu „black-box” i implementowany przy użyciu prostych, lecz nie zawsze optymalnych metod, takich jak tabele kontyngencji. Znaczenie zapytań zliczających podkreśla fakt, iż w wielu rzeczywistych zastosowaniach mogą one odpowiadać nawet za ponad 90% całkowitego czasu wykonania aplikacji [30]. Z tego względu poprawa ich wydajności stanowi istotny obszar badań, mający potencjał do znaczącego przyspieszenia działania wielu systemów uczenia maszynowego.

Inspiracją do badań nad adaptacją obliczeń uczenia maszynowego do wysokowydajnych systemów komputerowych, opisanych w niniejszej pracy, było narzędzie SABNAtk [58] – nowatorskie podejście do problemu zliczania, zaproponowane w pracy [30]. Poza intuicyjnym interfejsem programistycznym, SABNAtk oferuje skalowalne strategie realizacji zapytań zliczających. Jak pokazano w cytowanej pracy, strategie te znacząco przewyższają pod względem wydajności obliczeniowej oraz pamięciowej powszechnie stosowane podejścia, takie jak alternujące drzewa decyzyjne czy tablice mieszające.

Bezpośrednim efektem przeprowadzonych badań w zakresie adaptacji obliczeń uczenia maszynowego do wysokowydajnych systemów komputerowych jest opublikowanie przez autora niniejszej rozprawy doktorskiej prac [12, 15, 16, 17].

Skala i tempo rozwoju wysokowydajnych systemów komputerowych znajdują swoje odzwierciedlenie w regularnie publikowanych zestawieniach superkomputerów. Lista TOP500 [77] to ranking najbardziej wydajnych systemów, aktualizowany co sześć miesięcy nieprzerwanie od 1993 roku. Klasyfikuje on 500 najszybszych superkomputerów na podstawie liczby operacji zmiennoprzecinkowych wykonywanych w ciągu sekundy (flops). Liderem najnowszej, 65. edycji opublikowanej w czerwcu 2025 roku jest El Capitan – system składający się z ponad 11 milionów rdzeni, osiągający wydajność na poziomie 1.7 eksaflopa ($1 \text{ eksa} = 10^{18}$). Podium uzupełniają systemy Frontier oraz Aurora, będące obecnie jedynymi oprócz El Capitana superkomputerami z wydajnością powyżej jednego eksaflopa. Frontier, zajmujący w najnowszym rankingu drugie miejsce, zapisał się wyraźnie w historii rozwoju superkomputerów, gdy w 2022 roku jako pierwszy na świecie przekroczył barierę jednego eksaflopa. Wyścig w kierunku wydajności liczonej w eksaflopach rozpoczął się z kolei w 2008 roku, gdy superkomputer Roadrunner [28] jako pierwszy na świecie przekroczył granicę 1 petaflopa ($1 \text{ peta} = 10^{15}$), co odpowiada jednej tysięcznej eksaflopa.

Oprócz dążenia do osiągnięcia coraz lepszych wyników wydajnościowych, w ostatnich latach istotnym czynnikiem w rozwoju wysokowydajnych systemów komputerowych stała

się również efektywność energetyczna [40]. Potwierdzeniem tego trendu jest fakt, że od 2013 roku równoległe z listą TOP500 publikowany jest ranking Green500 [25], klasyfikujący superkomputery na podstawie liczby operacji zmiennoprzecinkowych wykonywanych na sekundę na każdy zużywany wat energii. Liderem najnowszej, 25. edycji opublikowanej w czerwcu 2025 roku jest superkomputer JEDI, osiągający wynik 72.7 gigaflopa na wat. Dla porównania, wydajność energetyczna superkomputera CINECA, zwycięzcy pierwszej edycji rankingu Green500 z 2013 roku, była ponad 22 razy niższa i wynosiła około 3,2 gigaflopa na wat. Poprawa wydajności energetycznej widoczna jest również wśród zwycięzców zestawienia TOP500. Podczas gdy lider rankingu z czerwca 2025 roku – El Capitan osiągnął 60.3 gigaflopa na wat, wyniki uzyskane przez Fugaku oraz Tainhe-2A, liderów zestawień odpowiednio z 2020 i 2015 roku, wyniosły 14.7 oraz 1.9 gigaflopa na wat.

Spółeczność naukowa podejmuje próbę sprostania wyzwaniu poprawy wydajności efektywności energetycznej na różnych poziomach, zarówno sprzętowym, jak i programowym [23]. Dynamiczne skalowanie napięcia i częstotliwości (DVFS) jest powszechnie akceptowaną techniką z poziomu rozwiązań sprzętowych [52]. Podejście to znane jest jako efektywny sposób oszczędzania energii w przypadku kodów ograniczonych przepustowością pamięci [26]. Zastosowanie DVFS umożliwia obniżenie napięcia i częstotliwości kosztem potencjalnie dłuższego czasu wykonania [18]. Dla wielordzeniowych procesorów z pamięcią współdzieloną [35] wyróżnia się różne warianty metody DVFS, pozwalające zmieniać częstotliwość: (i) całego układu procesora, (ii) poszczególnych domen DVFS, oraz (iii) indywidualnych rdzeni. Przykładowo, zarządzanie częstotliwością na poziomie pojedynczego rdzenia w przypadku procesorów firmy Intel możliwe jest począwszy od architektury Haswell i polega na ustalaniu minimalnej i maksymalnej częstotliwości danego rdzenia [75]. Podczas gdy pierwszy wariant metody DVFS odpowiada jednorodnemu (homogenicznemu) skalowaniu napięcia i częstotliwości w całym procesorze, dwa pozostałe odnoszą się do skalowania niejednorodnego (heterogenicznego).

Zrównoleglanie w modelu równoległości danych [60] stanowi popularną strategię dekompozycji obliczeń, w której domena danych aplikacji dzielona jest na tyle części, ile wątków wykonujących obliczenia. W podejściu tym każdy wątek wykonuje ten sam rodzaj operacji na różnych fragmentach danych. Z punktu widzenia kodu źródłowego, aplikacje zrównoleglane w tym modelu zawierają zazwyczaj ciało pętli, będące wielokrotnie wykonywane przez każdy fragment domeny obliczeniowej. W niniejszej pracy rozpatrywane są dwie aplikacje numeryczne należące do tej grupy. Pierwszą z nich jest problem trójwymiarowej dyfuzji, stanowiący klasyczne zagadnienie modelowania procesów transportowych w fizyce i inżynierii. Aplikacja ta, dzięki regularnej strukturze obliczeń dobrze wpisuje się w model równoległości danych. Drugą aplikacją jest MPDATA (ang. Multidimensional Positive Definite Advection Transport Algorithm) – szeroko stosowany

algorytm adwekcji, wykorzystywany m.in. w numerycznych modelach prognozowania pogody i klimatu. Obie aplikacje należą do klasy aplikacji ograniczonych przepustowości pamięci, co czyni je interesującymi przypadkami testowymi w badaniach nad efektywnością energetyczną.

Aplikacje zrównoleglane w modelu równoległości danych często charakteryzują się podobnym obciążeniem poszczególnych rdzeni lub wątków [3]. Z tego względu stanowią one powszechny przedmiot zastosowania homogenicznego wariantu metody DVFS. W literaturze dostępnych jest wiele przykładów zastosowania tego podejścia w celu redukcji zużycia energii z akceptowalną stratą wydajności [19, 54, 55, 73, 83].

Z kolei heterogeniczny wariant metody DVFS znajduje zastosowanie głównie w aplikacjach nieregularnych, w których różne rdzenie wykonują fragmenty kodu o odmiennych charakterystykach [3]. W takich przypadkach heterogeniczne dostosowanie częstotliwości umożliwia poprawę efektywności energetycznej dzięki precyzyjniejszej kontroli nad zachowaniem poszczególnych wątków [4, 45, 46, 82].

Dotychczas jednak brak jest badań, w których heterogeniczne podejście do DVFS zostałoby zastosowane w odniesieniu do aplikacji zrównoleglanych w modelu równoległości danych na platformach homogenicznych. Zagadnienie to zostanie omówione w niniejszej rozprawie doktorskiej w ramach części poświęconej adaptacji obliczeń numerycznych do wysokowydajnych systemów komputerowych. Bezpośrednim efektem przeprowadzonych badań w tym zakresie jest opublikowanie przez autora niniejszej rozprawy doktorskiej prac [13, 14].

1.2. Teza i cel pracy

Teza pracy brzmi:

Przeprowadzone badania wskazują na możliwość opracowania metod i algorytmów prowadzących do poprawy efektywności obliczeniowej i energetycznej uczenia maszynowego oraz aplikacji numerycznych w środowisku wysokowydajnych systemów komputerowych.

Celem pracy jest:

Opracowanie i eksperymentalna weryfikacja metod oraz algorytmów poprawiających efektywność obliczeniową i energetyczną uczenia maszynowego oraz aplikacji numerycznych w środowisku wysokowydajnych systemów komputerowych.

Do celów szczegółowych należą:

1. Opracowanie i eksperymentalna weryfikacja metody redukcji zużycia energii elektrycznej dla aplikacji zrównoleglanych w modelu równoległości danych oraz opracowanie algorytmów doboru konfiguracji częstotliwości umożliwiających praktyczne zastosowanie tej metody
2. Opracowanie i zbadanie efektywności metod poprawy wydajności strategii realizacji zapytań zliczających opartej o reprezentowanie stanów poszczególnych zmienionych zapytania za pomocą bitmap oraz algorytm przeszukiwania w głąb (DFS)
3. Sformułowanie teoretycznych modeli kosztu dla różnych strategii realizacji zapytań zliczających oraz opracowanie algorytmu ich adaptacyjnej aktualizacji
4. Implementacja metody doboru optymalnej strategii realizacji zapytań zliczających, bazującej na teoretycznych modelach kosztu oraz ocena jej skuteczności na przykładzie aplikacji uczenia maszynowego
5. Adaptacja opracowanej metody doboru optymalnej strategii realizacji zapytań zliczających do środowiska wielowątkowego
6. Opracowanie metody realizacji zapytań zliczających uwzględniającej architekturę systemów ccNUMA oraz ocena jej wpływu na wydajność aplikacji uczenia maszynowego

1.3. Struktura pracy

Praca składa się z ośmiu rozdziałów oraz aneksu, zawierającego istotniejsze fragmenty kodu źródłowego, dotyczącego opracowanych rozwiązań. Na końcu zamieszczono również spisy rysunków, tabel i listingów oraz bibliografię.

Rozdział 1. stanowi wstęp do pracy. Nakreślono w nim tło badań, sformułowano tezę badawczą, określono cel główny i cele szczegółowe, a także przedstawiono strukturę rozprawy.

Rozdział 2. poświęcony jest problematyce adaptacji obliczeń numerycznych do wysokowydajnych systemów komputerowych. Przedstawiono w nim opracowaną metodę redukcji zużycia energii elektrycznej w aplikacjach równoległych opartych na modelu równoległości danych. W pierwszej kolejności omówiono dwie aplikacje numeryczne, które posłużyły jako przypadki testowe w przeprowadzonych badaniach. Następnie, po przedstawieniu motywacji do badań i sformułowaniu problemu badawczego, zaprezentowano koncepcję proponowanego rozwiązania, polegającego na zarządzaniu częstotliwością odpowiednio zdefiniowanych grup rdzeni. W dalszej części rozdziału zawarto wyniki obszernej serii testów, mających na celu ocenę możliwych do uzyskania oszczędności energii. Istotnym elementem jest również opis dwóch opracowanych algorytmów, umożliwiających wyznaczenie optymalnej konfiguracji częstotliwości procesora w oparciu o pomiary zużycia energii lub czasu wykonania. Rozdział kończy dodatkowa weryfikacja metody przeprowadzona na platformie firmy AMD, oferującej mniejsze możliwości w zakresie zarządzania częstotliwością, niż docelowa platforma obliczeniowa firmy Intel.

Rozdział 3. stanowi wprowadzenie do problematyki adaptacji obliczeń uczenia maszynowego do wysokowydajnych systemów komputerowych. W pierwszej części przedstawiono formalne podstawy zapytań zliczających wraz z przykładami, a także omówiono ich zastosowania w uczeniu maszynowym oraz motywację do podjęcia badań w tym obszarze. Kolejna część poświęcona jest bibliotece SABNAtk, stanowiącej przykład narzędzia umożliwiającego efektywną realizację tego typu zapytań w aplikacjach uczenia maszynowego. Rozdział zamyka opis przyjętej metodologii ewaluacji, która stanowi punkt odniesienia dla wszystkich badań prezentowanych w dalszych częściach pracy (rozdziały 4–7).

Rozdział 4. przedstawia dwie opracowane metody poprawy wydajności realizacji zapytań zliczających, dedykowane dla strategii dostępnej w narzędziu SABNAtk. Strategia ta opiera się na reprezentowaniu stanów poszczególnych zmiennych zapytania za pomocą bitmap oraz na algorytmie przeszukiwania w głąb (DFS, Depth First Search). Pierwsza z zaproponowanych metod ukierunkowana jest na redukcję liczby wykonywanych przecięć bitmap (operacji AND), natomiast druga zmierza do minimalizacji liczby odwiedzanych węzłów drzewa. Rozdział kończy prezentacja wyników badań oceniają-

cych wpływ opracowanych metod na wydajność realizacji zapytań zliczających, przeprowadzonych z wykorzystaniem benchmarku losowych zapytań na trzech różnych platformach obliczeniowych.

Rozdział 5. dotyczy autorskiej metody automatycznego wyboru strategii realizacji zapytań zliczających. Celem tej metody jest skrócenie czasu realizacji strumienia zapytań poprzez indywidualny dobór optymalnej strategii do realizacji każdego z nich. W rozdziale tym w sposób szczegółowy omówiono motywację podjęcia problemu, koncepcję rozwiązania a także kolejne etapy jego rozwoju. Istotny element stanowi prezentacja modeli kosztu realizacji zapytania zliczającego, opracowanych przez autora w oparciu o dogłębną analizę matematyczną. Ważne miejsce zajmują także sekcje opisujące opracowane algorytmy aktualizacji modeli oraz przetwarzania strumienia zapytań. W rozdziale tym sporą uwagę poświęcono również procesowi wdrożenia opracowanego podejścia w praktyce, jako niezależnej strategii biblioteki SABNAtk. Rozdział zamyka sekcja poświęcona zbadaniu wpływu opracowanej metody na wydajność zapytań zliczających. W badaniach tych wdrożone rozwiązanie zostało porównane z pięcioma innymi strategiami, zarówno w benchmarku losowych zapytań, jak i w rzeczywistej aplikacji uczenia sieci Bayesowskich.

Rozdział 6. poświęcony jest opracowanej koncepcji, umożliwiającej wielowątkową realizację metody automatycznego wyboru strategii przedstawionej w rozdziale 5. Implementacja ta pozwala na zastosowanie modelu równoległości zadań (task parallelism), w którym poszczególne zapytania zliczające stanowią niezależne zadania wykonywane równolegle na wielu rdzeniach procesora. W rozdziale omówiono fragmenty pierwotnej wersji metody, mogące prowadzić do sytuacji wyścigu pomiędzy wątkami (race condition) podczas przetwarzania wielowątkowego oraz przedstawiono sposób rozwiązania tego problemu przy użyciu mechanizmu muteksów. Rozdział kończy prezentacja wyników badań wpływu opracowanej metody na wydajność zapytań zliczających, przeprowadzonych na dwóch platformach obliczeniowych oraz rzeczywistej aplikacji uczenia maszynowego zrównoleglonej przy użyciu biblioteki Intel TBB (Threading Building Blocks).

Rozdział 7. przedstawia metodę równoległej realizacji zapytań zliczających, dedykowaną architekturom nowej generacji typu ccNUMA (cache-coherent Non-Uniform Memory Access). Systemy te składają się z wielu domen NUMA i charakteryzują się zróżnicowanymi opóźnieniami dostępu do pamięci w zależności od fizycznej lokalizacji danych względem rdzenia wykonawczego. Zaproponowane rozwiązanie jest niezależne od wybranej strategii realizacji zapytań zliczających i może być stosowane z dowolnym podejściem, w szczególności z metodą automatycznego wyboru strategii, opisaną w poprzednich rozdziałach. Skuteczność opracowanego rozwiązania oceniono na przykładzie dwóch platform obliczeniowych, porównując je ze standardową techniką zrównoleglania opartą na bibliotece OpenMP.

Podsumowanie pracy znajduje się w **rozdziale 8.**, gdzie zawarto wnioski końcowe a także przedstawiono kierunki dalszych badań.

2. Metoda redukcji zużycia energii elektrycznej dla aplikacji zrównoleglanych w modelu równoległości danych

Niniejszy rozdział przedstawia opracowaną metodę redukcji zużycia energii elektrycznej dedykowaną dla aplikacji zrównoleglanych w modelu równoległości danych. Głównym celem opracowanej metody jest złagodzenie skutków nierównomiernej dystrybucji obliczeń pomiędzy dostępne wątki lub rdzenie procesorów, powstającej na skutek zrównoleglania tego typu aplikacji w oparciu o powszechnie stosowane podejścia. Opracowana metoda adresuje ten problem, pozwalając na istotną redukcję zużycia energii bez jednoczesnej utraty wydajności. Wyniki badań przedstawione w niniejszym rozdziale zostały zaprezentowane podczas konferencji Euro-Par 2021 [14] (140 pkt wg wykazu MNiSW) a także opublikowane w czasopiśmie Journal of Parallel and Distributed Computing [13] (100 pkt wg wykazu MNiSW).

2.1. Aplikacje zrównoleglane w modelu równoległości danych

W tej sekcji przedstawiono dwa przykłady aplikacji zrównoleglanych w modelu równoległości danych. Stanowią one punkt wyjścia do dalszych rozważań, których celem jest prezentacja opracowanej metody redukcji zużycia energii, opartej o heterogeniczne skalowanie częstotliwości odpowiednio wybranych grup rdzeni.

2.1.1. Problem trójwymiarowej dyfuzji

Dyfuzja jest zjawiskiem polegającym na samoczynnym przemieszczaniu się cząsteczek substancji w ośrodku, co prowadzi do ich równomiernego rozproszenia. Proces ten zachodzi nieustannie w przyrodzie i odgrywa istotną rolę w wielu zjawiskach fizycznych, chemicznych oraz biologicznych. Na szybkość dyfuzji wpływają głównie stan skupienia ośrodka oraz jego temperatura – intensywność dyfuzji rośnie wraz ze wzrostem temperatury, natomiast maleje przy zwiększeniu koncentracji cząsteczek. Największe tempo rozprzestrzeniania obserwuje się w gazach o wysokiej temperaturze, natomiast najwolniejsze w ciałach stałych w niskiej temperaturze. W wyniku dyfuzji dochodzi do stopniowego wyrównania stężeń substancji w danym ośrodku. Mechanizm ten wynika z losowego ruchu cząsteczek, będącego efektem ich wzajemnych zderzeń. Chaotyczny cha-

rakter tych ruchów został zaobserwowany w 1827 roku przez szkockiego biologa Roberta Browna, a samo zjawisko nazwano ruchem Browna [10].

Proces dyfuzji opisywany jest [21] następującym równaniem różniczkowym cząstkowym, będącym konsekwencją prawa Ficka oraz zasady zachowania masy:

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} + \frac{\partial^2 U}{\partial z^2} \quad (2.1)$$

Funkcja $U = U(x, y, z, t)$ z równania (2.1) opisuje stężenie danej substancji w punkcie (x, y, z) w chwili t . Przedstawione równanie może być również wykorzystywane do analizy innych zjawisk fizycznych. Przy spełnieniu odpowiednich założeń może ono opisywać proces przewodnictwa cieplnego, w którym funkcja U reprezentuje temperaturę. Zastosowanie metody różnic skończonych [85] do równania różniczkowego cząstkowego, danego wzorem 2.1, prowadzi do następującego schematu numerycznego:

$$U_{i,j,k}^{h+1} = \Delta t \left[(U_{i-1,j,k}^h - 2U_{i,j,k}^h + U_{i+1,j,k}^h) / \Delta x^2 + (U_{i,j-1,k}^h - 2U_{i,j,k}^h + U_{i,j+1,k}^h) / \Delta y^2 + (U_{i,j,k-1}^h - 2U_{i,j,k}^h + U_{i,j,k+1}^h) / \Delta z^2 \right] + U_{i,j,k}^h \quad (2.2)$$

W ramach omawianej aplikacji, rozważane są problemy trójwymiarowej dyfuzji zdefiniowane dla prostopadłościennej domeny o rozmiarach $X \times Y \times Z$, w której kolejne punkty siatki indeksowane są odpowiednio przez zmienne i , j oraz k . Zastosowana metoda jawna wymaga spełnienia określonych warunków dotyczących dyskretyzacji. W celu zapewnienia stabilności numerycznej algorytmu, kroki czasowe muszą być wystarczająco krótkie w porównaniu do kroków przestrzennych. W związku z tym wykonywane symulacje są zazwyczaj długotrwałe i obejmują wiele tysięcy kroków czasowych [11, 14].

Na listingu 2.1 zaprezentowano główny fragment kodu aplikacji, tzw. jądro obliczeniowe. Zostało ono zaimplementowane w sposób równoległy przy użyciu standardu OpenMP [44]. Schemat zrównoleglania oparty jest o równoległość danych wzdłuż wymiaru i i polega na podziale danych pomiędzy wątki przy pomocy dyrektywy `#pragma omp for`. Następnie wykorzystana jest wektoryzacja obliczeń wzdłuż wymiaru k z użyciem dyrektywy `#pragma vector`.

```

1 #pragma omp for
2 for(int i=0; i<X; i++)
3   for(int j=0; j<Y; j++)
4     #pragma omp simd
5     for(int k=0; k<Z; k++)
6       v(i,j,k) = u(i,j,k);
7       v(i,j,k) += xS * (u(i-1,j,k) - 2*u(i,j,k) + u(i+1,j,k));
8       v(i,j,k) += yS * (u(i,j-1,k) - 2*u(i,j,k) + u(i,j+1,k));
```

```
9      v(i,j,k) += zS * (u(i,j,k-1) - 2*u(i,j,k) + u(i,j,k+1));
```

Listing 2.1. Zrównoleglenie jądra obliczeniowego aplikacji trójwymiarowej dyfuzji przy użyciu biblioteki OpenMP

W prezentowanym kodzie, głównym ograniczeniem wydajności jest przepustowość pamięci operacyjnej. Problem ten staje się „wąskim gardłem” szczególnie w przypadku analizowania problemów o dużych rozmiarach domen, które skutkują znacznym przekroczeniem dostępnych zasobów pamięci podręcznej [13].

2.1.2. MPDATA – wielowymiarowy dodatnio określony algorytm adwekcji

Aplikacja MPDATA implementuje ogólne podejście do modelowania szerokiego zakresu złożonych przepływów geofizycznych w skali od mikro- do planetarnej [56, 66, 70]. Do przykładów jej zastosowania należą prognozowanie pogody, symulacje przepływów miejskich, turbulencji oraz prądów oceanicznych. MPDATA należy do klasy metod numerycznych stosowanych w symulacji przepływów cieczy, które bazują na własnościach schematów różnicowych zachowujących znak. Schemat ten jest głównie wykorzystywany do rozwiązywania problemów adwekcji w kolejnych krokach czasowych, co pozwala zaliczyć aplikację MPDATA do grupy algorytmów krokowych w czasie. Schemat numeryczny MPDATA został szczegółowo opisany w [63].

W każdym kroku aplikacji wykonywany jest zestaw 17 jąder obliczeniowych, które są wzajemnie od siebie zależne. Wyniki wcześniejszych jąder zazwyczaj stanowią dane wejściowe dla kolejnych. Wszystkie jądra wykonywane są sekwencyjnie, jedno po drugim, z równoległym przetwarzaniem każdego jądra przy użyciu OpenMP. Podobnie jak w przypadku problemu trójwymiarowej dyfuzji, MPDATA umożliwia zastosowanie jednowymiarowej dekompozycji, w której równoległość uzyskuje się poprzez podział iteracji zewnętrznej pętli każdego jądra pomiędzy rdzenie za pomocą dyrektywy `#pragma omp for`. Podobnie jak w przypadku omówionej aplikacji trójwymiarowej dyfuzji, wewnętrzna pętla wektoryzowana jest przy użyciu dyrektywy `#pragma omp simd`. Jak pokazano w pracach [67, 72, 73], jednowymiarowa dekompozycja umożliwia bardziej efektywne wykorzystanie pamięci w architekturach współdzielonej pamięci typu ccNUMA niż dekompozycja dwu- lub trójwymiarowa.

```
1  /*...*/
2  //Kernel 4
3  #pragma omp for
4  for(int i=0; i<X; i++)
5      for(int j=0; j<Y; j++)
```

```

6      #pragma omp simd
7      for(int k=0; k<Z; k++)
8          x[i,j,k] = XIn[i,j,k] - (((F1[i+1,j,k] - F1[i,j,k]) + (F2[i,j+1,k]
9              -F2[i,j,k]) + (F3[i,j,k+1] - F3[i,j,k])) / H[i,j,k]);
10     /*...*/

```

Listing 2.2. Zrównoleglenie jądra obliczeniowego aplikacji MPDATA przy użyciu biblioteki OpenMP

MPDATA jest zwykle wykorzystywana w długotrwałych symulacjach, realizujących tysiące kroków czasowych. Jeden krok operuje na pięciu macierzach (tablicach) wejściowych i zwraca pojedynczą tablicę wyjściową wykorzystywaną w kolejnym kroku. Jak wykazano w pracach [68, 69], MPDATA jest aplikacją ograniczoną przez przepustowość pamięci.

2.2. Motywacja do badań i sformułowanie problemu badawczego

Strategia zrównoleglenia przedstawiona podczas omawiania obu aplikacji (listingi 2.1 oraz 2.2) stanowi punkt wyjścia do przeprowadzonych badań. W obu przypadkach, wykorzystując bibliotekę OpenMP, iteracje pierwszej pętli (wymiar i) są rozdzielane pomiędzy wątki utworzonego regionu równoległego. W efekcie równomierny rozkład obciążenia występuje wyłącznie w sytuacji, gdy liczba iteracji X dzieli się bez reszty przez liczbę logicznych wątków danej platformy obliczeniowej, co w praktyce dzieje się bardzo rzadko. W konsekwencji przyjęta strategia zrównoleglenia w większości przypadków prowadzi do nierównomiernego podziału pracy pomiędzy wątki oraz wynikających z tego różnic w czasie ich pracy.

Przed przejściem do opisu przeprowadzonych badań, w pierwszej kolejności zdefiniowane zostaną dwa parametry, mające na celu sformalizowanie oraz opisanie skali zaobserwowanej dysproporcji. Pierwszym z parametrów jest procent niezbalansowania obciążenia (load imbalance percentage, LIP), definiowany jako stosunek liczby wątków bardziej obciążonych pracą do całkowitej liczby wątków, oznaczonej symbolem LC (logical cores):

$$LIP = \frac{X \bmod LC}{LC} \cdot 100\%. \quad (2.3)$$

Drugim parametrem jest współczynnik R , określający dysproporcję przydzielonych iteracji pomiędzy wątkami. Zakładając, że $X > LC$, bardziej obciążone wątki wykonują R razy więcej iteracji niż pozostałe wątki, gdzie parametr ten wyrażony jest jako:

$$R = \frac{\lceil X/LC \rceil}{\lceil X/LC \rceil - 1}. \quad (2.4)$$

Aby wyjaśnić interpretację zdefiniowanych parametrów, przedstawiony zostanie przykład obliczania ich wartości. Niech dana będzie platforma obliczeniowa wyposażona w $LC = 64$ logiczne wątki oraz domena obliczeniowa, w której liczba iteracji zrównoleglanej pętli wynosi $X = 100$. Zastosowanie przedstawionej strategii zrównoleglenia prowadzi wtedy do problemu nierównego podziału pracy pomiędzy wątki, którego skala może zostać opisana za pomocą parametrów LIP oraz R . Dla przedstawionej konfiguracji wartość parametru LIP wynosi:

$$LIP = \frac{100 \bmod 64}{64} \cdot 100\% = \frac{36}{64} \cdot 100\% = 56,25\%. \quad (2.5)$$

Oznacza to, że ponad połowa wątków wykonuje więcej iteracji niż pozostałe.

Z kolei współczynnik dysproporcji obciążenia R może zostać obliczony jako:

$$R = \frac{\lceil 100/64 \rceil}{\lceil 100/64 \rceil - 1} = \frac{2}{1} = 2. \quad (2.6)$$

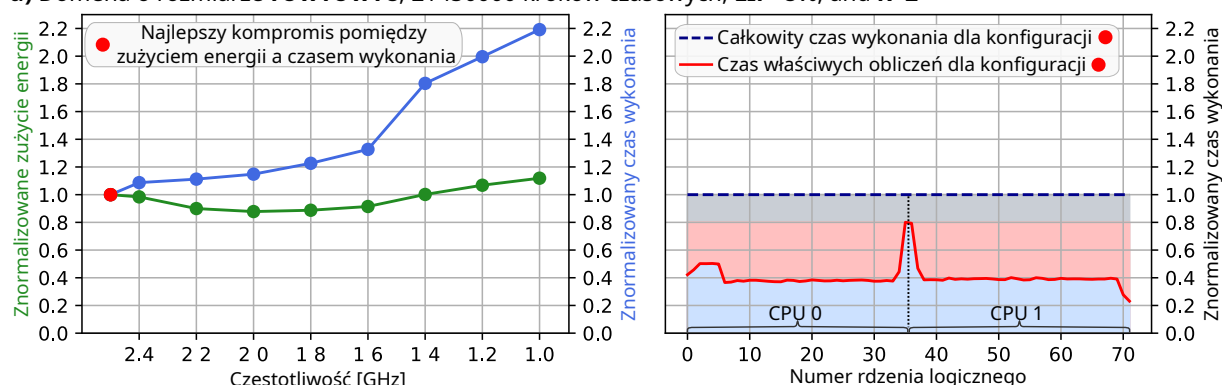
Uzyskana wartość dla tego parametru oznacza, iż najbardziej obciążone wątki wykonują dwukrotnie więcej iteracji niż pozostałe.

Kolejnym etapem prac było przeprowadzenie serii eksperymentów, mających na celu ocenę skuteczności powszechnie stosowanej, homogenicznej metody DFVS w kontekście zaobserwowanej dysproporcji obciążenia. W tym celu obie opisane aplikacje przetestowano z wykorzystaniem platformy obliczeniowej wyposażonej w dwa 18-rdzeniowe procesory Intel Xeon Gold 6240 (Cascade Lake), oferujące łącznie 72 wątki logiczne. Każdy z procesorów wyposażony jest w 24.75 MB pamięci podręcznej L3. Parametry techniczne platformy umożliwiają ustawienie minimalnej częstotliwości taktowania na poziomie 1.0 GHz z możliwością jej zwiększania co 0.1 GHz, aż do osiągnięcia maksymalnej częstotliwości Turbo Boost wynoszącej 2.5 GHz.

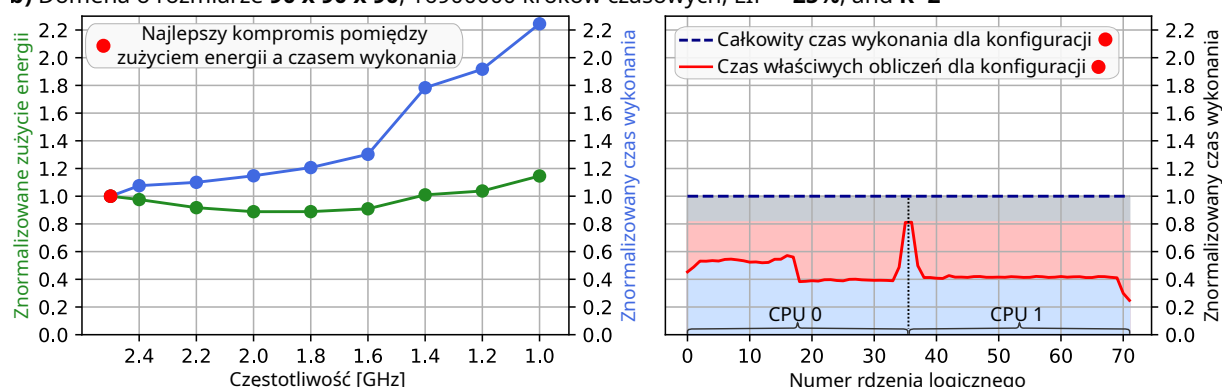
Pomiar zużycia energii realizowano z wykorzystaniem miernika Yokogawa WT310 [73], umożliwiającego pomiar całkowitej energii pobieranej przez serwer obliczeniowy. Urządzenie to dostarcza zasilanie do serwera wykonującego obliczenia, umożliwiając rejestrację całkowitego zużycia energii, co zapewnia maksymalną precyzję i wiarygodność uzyskanych wyników. W celu zwiększenia rzetelności pomiarów zagwarantowano, że symulacje dla obu aplikacji trwały odpowiednio długo. Liczbę kroków czasowych dla każdego rozmiaru domeny dostosowano tak, aby czas wykonania wynosił co najmniej 700 sekund. Pozwoliło to na utrzymanie współczynnika zmienności [24] na poziomie nieprzekraczającym 2.5% dla pomiarów zużycia energii oraz 1.5% dla pomiarów czasu wykonania. Dodatkowo, eksperymenty przeprowadzono w klimatyzowanej serwerowni o stabilnej temperaturze, przy zapewnieniu wyłączności dostępu do wykorzystywanego serwera.

Rysunek 2.1 przedstawia podsumowanie testów dla aplikacji trójwymiarowej dyfuzji.

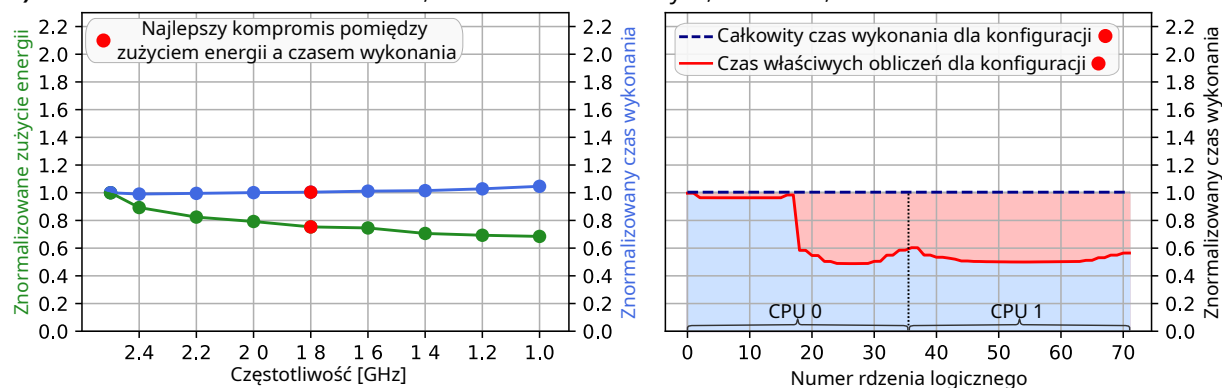
a) Domena o rozmiarze **78 x 78 x 78**, 21450000 kroków czasowych, **LIP=8%**, and **R=2**



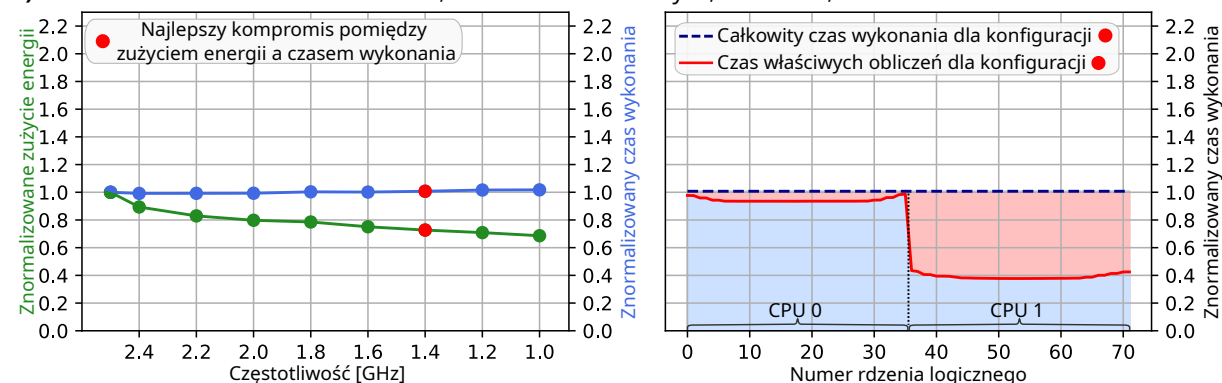
b) Domena o rozmiarze **90 x 90 x 90**, 16900000 kroków czasowych, **LIP = 25%**, and **R=2**



c) Domena o rozmiarze **90 x 794 x 794**, 56160 kroków czasowych, **LIP = 25%**, and **R=2**



d) Domena o rozmiarze **108 x 794 x 794**, 3300 kroków czasowych, **LIP = 50%**, and **R=2**



Rysunek 2.1. Czas realizacji oraz zużycie energii elektrycznej uzyskane dla aplikacji trójwymiarowej dyfuzji podczas stosowania homogenicznej metody DFVS

We wszystkich czterech prezentowanych przykładach (a – d) wartość parametru R wynosi 2, natomiast parametr LIP zmienia się od 8 do 50%.

Lewa część rysunku 2.1 ilustruje zależność pomiędzy czasem wykonania a zużyciem energii elektrycznej przy zastosowaniu jednorodnego skalowania napięcia i częstotliwości (DVFS) dla wszystkich rdzeni. W przeprowadzonych testach założono dopuszczalny poziom strat wydajności nieprzekraczający 2%. Jak pokazano na rysunkach 2.1a i 2.1b, najlepszy kompromis między wydajnością a zużyciem energii (oznaczony czerwonymi punktami) uzyskiwany jest przy najwyższej dostępnej częstotliwości taktowania procesorów. Nawet nieznaczne obniżenie częstotliwości prowadzi do redukcji zużycia energii, jednak jednocześnie powoduje przekroczenie dopuszczalnego progu strat wydajności. Sytuacja ta wynika z faktu, iż stosunkowo niewielki rozmiar domeny obliczeniowej pozwala na to, aby wszystkie dane aplikacji mieściły się w pamięci podręcznej, dzięki czemu jednostki obliczeniowe nie muszą oczekiwać na dostęp do pamięci operacyjnej.

Zupełnie odmienną charakterystykę czasu wykonania i zużycia energii elektrycznej przedstawiają rysunki 2.1c i 2.1d, dotyczące większych rozmiarów domen obliczeniowych. Dla wszystkich rozważanych częstotliwości czas wykonania pozostaje niemal niezmienny.

W tym przypadku częstotliwość procesora nie stanowi wąskiego gardła i jej zmniejszenie nie wpływa wyraźnie na wydajność obliczeń. Dominujące ograniczenie stanowi tutaj przepustowość pamięci operacyjnej.

Największe oszczędności energetyczne uzyskane dzięki homogenicznej metodzie DVFS wynoszą około 30%, przy stratach wydajności nieprzekraczających 1–2%. Przeprowadzone eksperymenty wskazują jednak na potencjał modyfikacji klasycznego podejścia do metody DVFS, zakładającej jednorodne skalowanie częstotliwości dla wszystkich rdzeni. W celu zilustrowania tej obserwacji, prawa część rysunków 2.1a — 2.1d prezentuje rozkład czasu pracy poszczególnych wątków dla różnych rozmiarów domen oraz zróżnicowanego obciążenia, określanego przez wartość parametrów R oraz LIP .

Wykresy te przedstawiają czas wykonania mierzony dla każdego wątku OpenMP, podzielony na czas obliczeń (oznaczony błękitnym tłem) oraz czas synchronizacji. W ramach synchronizacji wyróżniono dwa etapy: (i) oczekiwanie na osiągnięcie bariery przez wszystkie wątki (czerwone tło) oraz (ii) etap opuszczania bariery, czyli moment, w którym wszystkie oczekujące wątki zostają zwolnione (szare tło).

Etap opuszczania bariery dla małych rozmiarów domen (2.1a, 2.1b) jest relatywnie kosztowny, natomiast dla większych domen (2.1c, 2.1d) staje się praktycznie pomijalny. Wynika to z faktu, iż koszt ten zależy od liczby punktów synchronizacji, która w analizowanych przypadkach rośnie liniowo wraz z liczbą kroków czasowych. Dla pojedynczego punktu synchronizacji czas ten zależy głównie od implementacji OpenMP oraz charakterystyki sprzętowej platformy.

Analizując przedstawione czasy obliczeń (niebieskie obszary) widoczne jest ich istotne

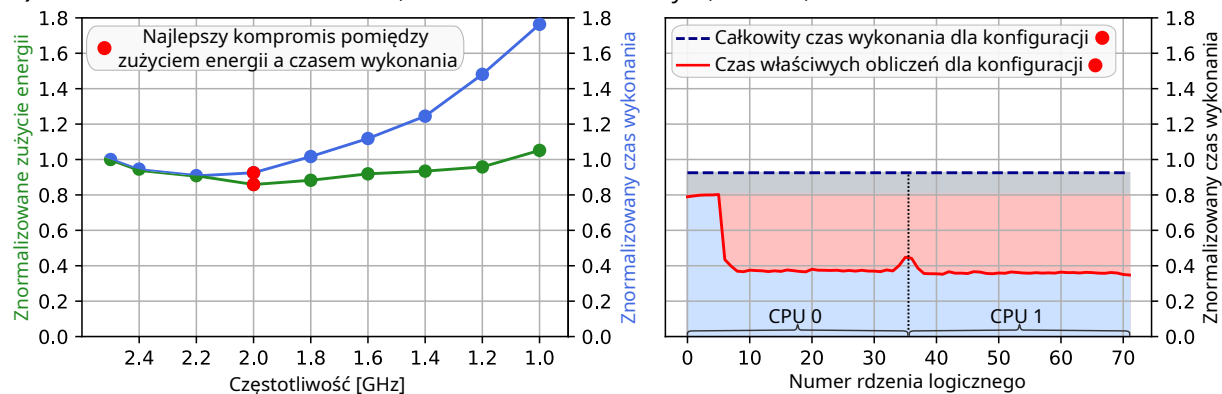
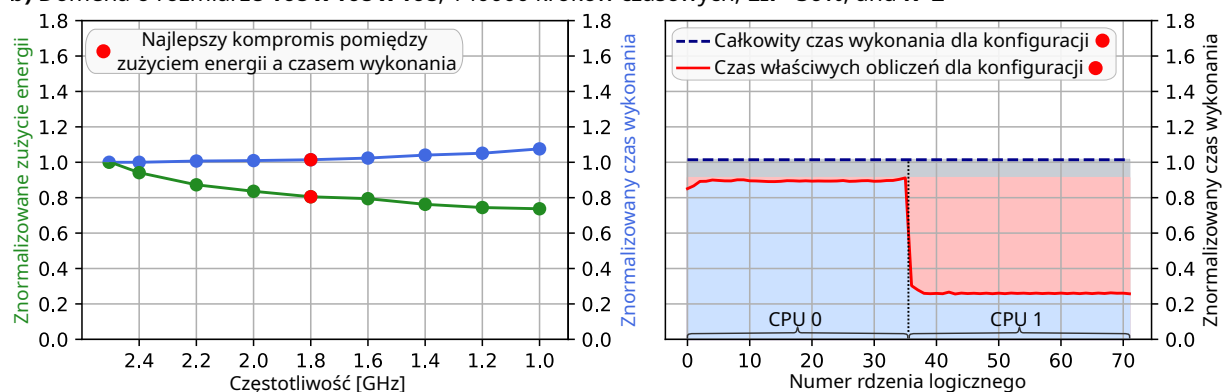
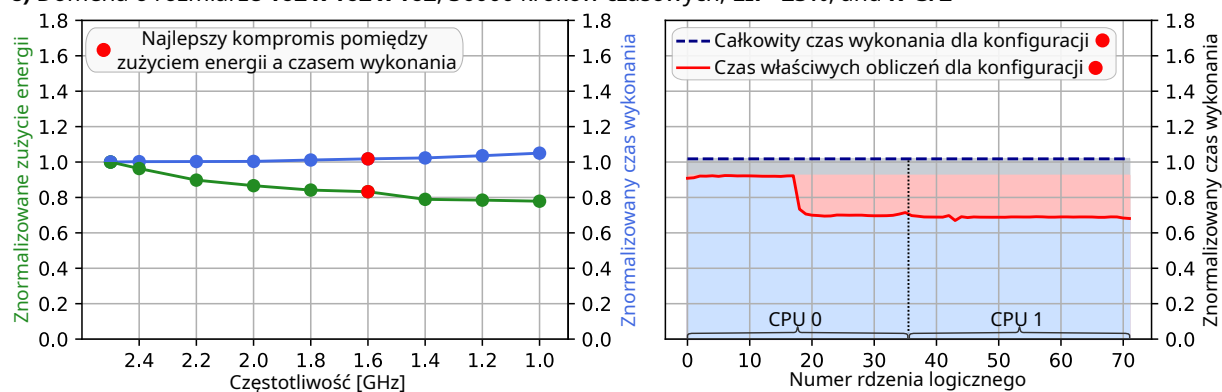
zróznicowanie pomiędzy wątkami. Niejednorodność ta wynika z nierównomiernego rozkładu obciążenia pomiędzy rdzeniami, opisanego parametrami R oraz LIP .

W przedstawionych rysunkach można również dostrzec negatywny wpływ wymiany danych między procesorami na czas pracy wątków. Z prawej części rysunków 2.1a i 2.1b wynika, że wątki wymagające wymiany danych między gniazdami procesorów (CPU0: 34 i 35 oraz CPU1: 36 i 37) wykazują najdłuższy czas obliczeń, mimo tego, iż wykonują one dwukrotnie mniejszą liczbę iteracji niż bardziej obciążone wątki. Taka sytuacja wynika z faktu, iż w tych przypadkach dane aplikacji mieszczą się w całości w pamięci podręcznej, a głównym ograniczeniem wydajności stają się opóźnienia wynikające z między-gniazdowej wymiany danych. Dla większych rozmiarów domen (2.1c, 2.1d) opóźnienia te są pomijalne, gdyż dominującym czynnikiem ograniczającym staje się przepustowość pamięci operacyjnej.

Analogiczny zestaw eksperymentów przeprowadzono dla aplikacji MPDATA w celu potwierdzenia uzyskanych obserwacji. Rysunki 2.2a – 2.2c przedstawiają wyniki dla trzech różnych rozmiarów domen, odpowiadających różnym poziomom niezrównoważenia obciążenia, określonym przez parametr LIP . W dwóch pierwszych analizowanych przypadkach wartość współczynnika R wynosi 2, natomiast w trzecim jest ona równa $3/2$. Podobnie jak w przypadku aplikacji trójwymiarowej dyfuzji, lewe części rysunków 2.2a – 2.2c przedstawiają czas wykonania oraz zużycie energii przez algorytm MPDATA w zależności od częstotliwości ustawionej dla wszystkich rdzeni. W odróżnieniu od aplikacji trójwymiarowej dyfuzji, we wszystkich trzech rozpatrywanych przypadkach możliwe jest obniżenie częstotliwości bez gwałtownego wzrostu czasu wykonania. Wynika to z faktu, iż aplikacja MPDATA charakteryzuje się znacznie większym zapotrzebowaniem na pamięć. W efekcie, wąskim gardłem jest nie częstotliwość taktowania procesora, lecz czas dostępu do pamięci operacyjnej.

Analiza prawych części rysunku 2.2 ponownie wskazuje na zróznicowanie rzeczywistego czasu obliczeń pomiędzy wątkami. Możliwe jest wyróżnienie grupy wątków, dla których czas wykonania jest znacznie dłuższy, co wynika z niezbalansowanego obciążenia, określonego przez parametry R oraz LIP . W porównaniu do aplikacji trójwymiarowej dyfuzji, główną różnicą jest mniejsze znaczenie między-gniazdowej wymiany danych dla rozbieżności czasów wątków. Wynika to z faktu, iż ilość pamięci związana z rozpatrywanymi domenami obliczeniowymi znacząco przekracza dostępną pojemność pamięci podręcznej. Warto zauważyć, iż choć znalezienie rozmiarów domen, mieszczących się w pełni w pamięci podręcznej jest możliwe, to z punktu widzenia praktycznych zastosowań przypadki te mają ograniczone znaczenie.

Podsumowując obserwacje dotyczące obu przypadków użycia, homogeniczna wersja metody DVFS umożliwia wybór kompromisu pomiędzy wydajnością a zużyciem energii, uwzględniając łączny czas wykonania, odzwierciedlający maksymalny czas obliczeń

a) Domena o rozmiarze **78 x 78 x 78**, 1000000 kroków czasowych, **LIP=8%**, and **R=2**b) Domena o rozmiarze **108 x 108 x 108**, 140000 kroków czasowych, **LIP=50%**, and **R=2**c) Domena o rozmiarze **162 x 162 x 162**, 36000 kroków czasowych, **LIP=25%**, and **R=3/2**

Rysunek 2.2. Czas realizacji oraz zużycie energii elektrycznej uzyskane dla aplikacji MPDATA podczas stosowania homogenicznej metody DFVS

wśród wszystkich wątków. Należy jednak zauważyć, że wątki przetwarzające mniejszą liczbę iteracji pętli marnują energię, oczekując w punktach synchronizacji na zakończenie pracy przez bardziej obciążone wątki. Znaczenie tego mankamentu nasila się wraz ze stale rosnącą liczbą rdzeni oferowanych przez nowoczesne architektury wielordzeniowe.

2.3. Koncepcja opracowanego rozwiązania

Analiza przedstawiona w poprzedniej sekcji pozwala na dostrzeżenie potencjału dostosowywania częstotliwości taktowania oraz napięcia zasilającego do poszczególnych grup rdzeni w celu zaadresowanie problemu nierównomiernego obciążenia. Opracowane rozwiązanie wyróżnia dwa scenariusze heterogenicznego podejścia DVFS, w których zarządzanie częstotliwością taktowania dotyczy odpowiednio zdefiniowanych grup rdzeni. Główna idea polega na indywidualnym dostosowywaniu częstotliwości zegara dla odpowiednio wyselekcjonowanych grup rdzeni, różniących się rzeczywistym czasem obliczeń. Pierwszy ze scenariuszy nosi nazwę *in-cache* i dotyczy przypadków charakteryzujących się mniejszym rozmiarem problemu (rysunki 2.1a i 2.1b). Scenariusz ten odnosi się do sytuacji, w których czas rzeczywistych obliczeń ograniczany jest przez narzut związany z wymianą danych pomiędzy procesorami. W tym przypadku wyróżniono trzy grupy rdzeni: (i) rdzenie obciążone komunikacją między procesorami, (ii) rdzenie wykonujące większą liczbę iteracji zrównoleglanej pętli oraz (iii) pozostałe rdzenie wykonujące mniej iteracji zrównoleglanej pętli. Liczebność drugiej i trzeciej grupy zależy od parametru *LIP*, będącego rezultatem rozmiaru domeny obliczeniowej oraz liczby rdzeni platformy obliczeniowej. Strategia zakłada przypisanie najwyższej częstotliwości pierwszej grupie oraz sukcesywne obniżanie częstotliwości dla pozostałych grup w celu znalezienia optymalnej konfiguracji. Przeprowadzone badania wykazały, że obniżanie częstotliwości taktowania rdzeni obciążonych komunikacją między procesorami nie jest konieczne. Zastosowanie maksymalnej dostępnej częstotliwości dla tych rdzeni powinno poprawić efektywność operacji ładowania i zapisu danych, co umożliwia osiągnięcie optymalnych rezultatów zarówno pod względem wydajności, jak i zużycia energii.

Drugi scenariusz nosi nazwę *out-of-cache* i odnosi się do przypadków większych domen obliczeniowych, w których ograniczenia pamięci operacyjnej prowadzą do spadku wydajności (Rys. 2.1c, 2.1d oraz Rys. 2.2). W tym scenariuszu wyodrębnia się dwie grupy rdzeni: rdzenie wykonujące więcej (i) oraz mniej (ii) iteracji pętli. W rezultacie, wyższy priorytet — a co za tym idzie, wyższe napięcie i częstotliwość — przypisuje się pierwszej grupie rdzeni.

2.4. Badanie wpływu opracowanej metody na zużycie energii elektrycznej

Aby ocenić wpływ opracowanej metody na zużycie energii elektrycznej, dla obu aplikacji przeprowadzono serię eksperymentów w stylu „brute force”. Podejście to polegało na testowaniu wszystkich możliwych konfiguracji częstotliwości taktowania dla zdefiniowanych grup rdzeni. Każda z analizowanych częstotliwości dla grupy rdzeni z większym obciążeniem, utworzyła wiele konfiguracji testowych z częstotliwościami grupy z mniejszym obciążeniem, które powstały poprzez stopniowe zmniejszanie częstotliwości co 0.2 GHz aż do wartości minimalnej. Przykładowo, podczas testowania częstotliwości 2.0 GHz dla rdzeni z większym obciążeniem, częstotliwość grupy rdzeni z mniejszym obciążeniem zmieniała się w zakresie 2.0, 1.8, ..., 1.0 GHz. Tabela 2.1 przedstawia pełny zestaw testowanych konfiguracji. Ponadto, w przypadkach dotyczących scenariusza *in-cache*, częstotliwość rdzeni granicznych, niezależnie od przynależności do wyszczególnionych grup, ustawiana była na maksymalną możliwą, wynoszącą 2.5 GHz.

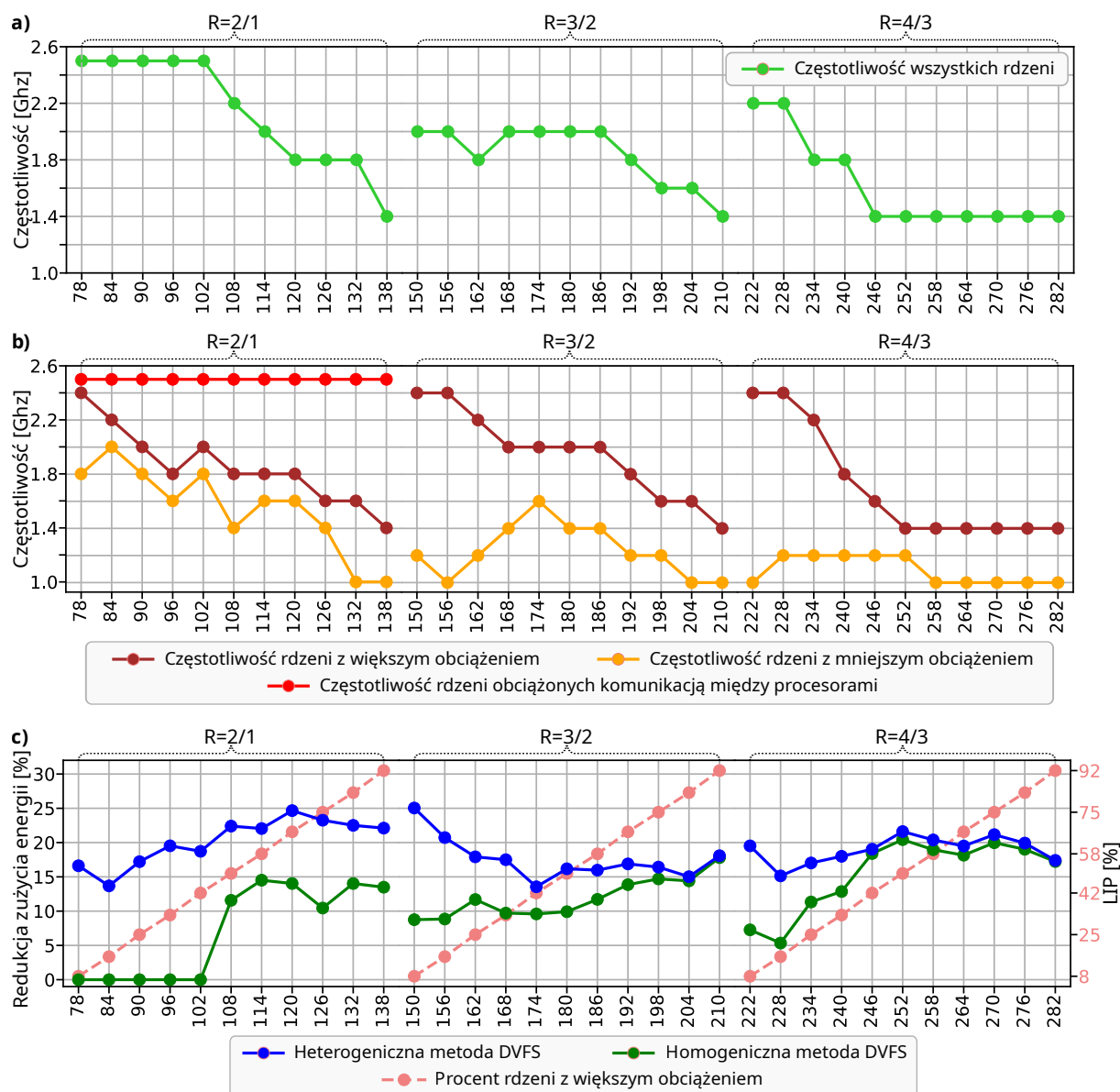
Grupa rdzeni z większym obciążeniem	Grupa rdzeni z mniejszym obciążeniem
2.5	2.5, 2.4, 2.2, 2.0, 1.8, 1.6, 1.4, 1.2, 1.0
2.4	2.4, 2.2, 2.0, 1.8, 1.6, 1.4, 1.2, 1.0
2.2	2.2, 2.0, 1.8, 1.6, 1.4, 1.2, 1.0
2.0	2.0, 1.8, 1.6, 1.4, 1.2, 1.0
1.8	1.8, 1.6, 1.4, 1.2, 1.0
1.6	1.6, 1.4, 1.2, 1.0
1.4	1.4, 1.2, 1.0
1.2	1.2, 1.0
1.0	1.0

Tabela 2.1: Wykorzystane konfiguracje częstotliwości (w GHz) podczas testowania opracowanej metody

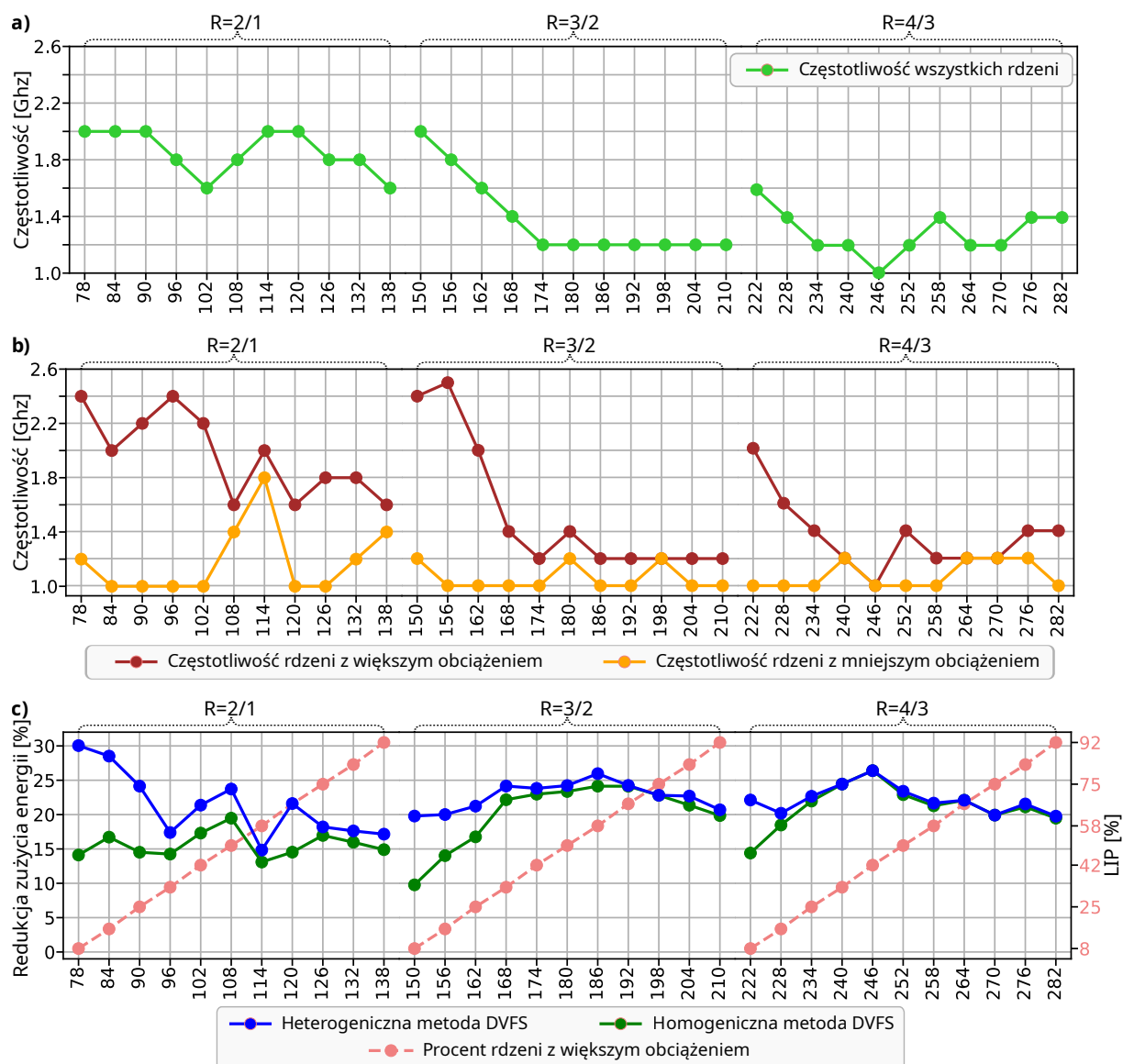
Celem badań było porównanie oszczędności energetycznych oferowanych przez opracowaną metodę heterogenicznego zarządzania częstotliwością z powszechnie stosowanym podejściem homogenicznym, zakładającym pracę wszystkich rdzeni z tą samą częstotliwością. Podsumowanie przeprowadzonych testów dla obu aplikacji trójwymiarowej dyfuzji oraz aplikacji MPDATA przedstawiono odpowiednio na rysunkach 2.3 oraz 2.4. Do badań w obu przypadkach wybrano zestaw tych samych 33 domen obliczeniowych, charakteryzujących się różnym rozmiarem. W rezultacie tworzą one kompleksowy zestaw testów, uwzględniający różne wartości stosunku $R \in \{2, \frac{3}{2}, \frac{4}{3}\}$ oraz parametr LIP zmieniający się dla każdego stosunku R w zakresie od 8 do 92%.

W obu rysunkach, górny wykres (a) dotyczy podejścia homogenicznego i przedstawia częstotliwości rdzeni, dla których odnotowano najniższe zużycie energii, przy założeniu nieprzekroczenia 2% spadku wydajności w porównaniu do częstotliwości bazowej, wynoszącej 2.5 GHz. Analogiczne założenia przyjęto dla środkowych wykresów (b), na których prezentowane są optymalne konfiguracje częstotliwości dla podejścia heterogenicznego. Dolne wykresy (c) porównują natomiast oba podejścia pod względem uzyskanych oszczędności energetycznych.

W przypadku aplikacji trójwymiarowej dyfuzji (rysunek 2.3) dla domen o rozmiarze nieprzekraczającym $138 \times 138 \times 138$, wyodrębniono trzy grupy częstotliwości taktowania (scenariusz *in-cache*), natomiast dla większych domen wystarczające okazały się dwie grupy (scenariusz *out-of-cache*). Najlepsze uzyskane wyniki za pomocą podejścia heterogenicznego wynoszą około 25% i zostały odnotowane dla domeny o rozmiarze $150 \times 150 \times 150$. Dla porównania, wynik w tej samej domenie uzyskany przez podejście homogeniczne wynosi jedynie 9%. Należy zauważyć, że opracowane podejście heterogeniczne zapewniło lepsze wyniki energetyczne we wszystkich przeprowadzonych testach.



Rysunek 2.3. Porównanie redukcji zużycia energii elektrycznej uzyskanego za pomocą homogenicznego oraz heterogenicznego wariantu metody DVFS zastosowanych dla domen aplikacji trójwymiarowej dyfuzji o różnym rozmiarze



Rysunek 2.4. Porównanie redukcji zużycia energii elektrycznej uzyskanego za pomocą homogenicznego oraz heterogenicznego wariantu metody DVFS zastosowanych dla domen aplikacji MPDATA o różnym rozmiarze

Największą poprawę efektywności energetycznej względem rozwiązania homogenicznego zaobserwowano dla dziedziny o rozmiarze $96 \times 96 \times 96$. W tym przypadku odnotowano przewagę rzędu 20 punktów procentowych względem tradycyjnego skalowania częstotliwości, które nie przyniosło żadnych korzyści energetycznych.

W przypadku aplikacji MPDATA (rysunek 2.4) wszystkie analizowane domeny nie mieszczą się w pamięci podręcznej. W związku z tym najlepszym rozwiązaniem okazało się zastosowanie scenariusza *out-of-cache*, polegającego na wyszczególnieniu jedynie dwóch grup rdzeni. Najbardziej wyraźna różnica pomiędzy podejściem heterogenicznym a homogenicznym widoczna jest w przypadkach, gdy parametr $R = 2$. Dla dziedziny o rozmiarze $78 \times 78 \times 78$ podejście heterogeniczne pozwala na redukcję zużycia energii o 30%, podczas gdy podejście homogeniczne zapewnia oszczędność rzędu 14%. Podsumowując, w przypadku aplikacji MPDATA, w 28 z 33 analizowanych przypadków możliwe było wyznaczenie konfiguracji heterogenicznej, która pozwoliła na zmniejszenie zużycia energii względem wyniku uzyskanego przy optymalnej konfiguracji homogenicznej. Ponadto, warto zauważyć, że w 21 z 33 testowanych przypadków rdzeniom z niższym obciążeniem przypisano najniższą możliwą częstotliwość danej platformy obliczeniowej, równą 1.0 GHz. Tym samym, potencjalne oszczędności energetyczne mogłyby być jeszcze większe, gdyby dalsze obniżanie częstotliwości było możliwe.

2.5. Algorytmy wyboru heterogenicznej konfiguracji częstotliwości

Przedstawiona w poprzedniej sekcji metoda przeszukiwania w stylu „brute force” umożliwiła odnalezienie heterogenicznych kombinacji częstotliwości, zapewniających minimalne zużycie energii bez pogorszenia wydajności. Podejście te wiąże się jednak z wysokim kosztem obliczeniowym, czyniącym je niepraktycznym sposobem wyznaczania optymalnego ustawienia częstotliwości.

Aby ocenić koszt strategii „brute force”, w pierwszej kolejności wprowadzone zostaną oznaczenia, formalizujące schemat postępowania. Przyjmuje się, że testy rozpoczynane są od częstotliwości zegara równej f_{max} . Następnie częstotliwość jest zmniejszana w zakresie od f_{start} do f_{stop} , co f_{step} . Zbiór testowanych częstotliwości wygląda więc następująco:

$$\mathbf{F} = \{f_{max}, f_{start}, \dots, f_{stop}\}, \quad (2.7)$$

a liczba jego elementów, może zostać wyznaczona za pomocą wzoru:

$$N = |\mathbf{F}| = \frac{f_{start} - f_{stop}}{f_{step}} + 2. \quad (2.8)$$

Zgodnie z opisanym podejściem „brute force”, każda częstotliwość f_i ze zbioru $\mathbf{F}$ łączona jest z podzbiorem $\mathbf{F}$, zawierającym elementy f_j spełniające warunek $f_j \leq f_i$. W rezultacie, liczba testowanych konfiguracji TC wyrażona jest następująco:

$$TC = \frac{N(N+1)}{2} \quad (2.9)$$

Przykładowo, strategia „brute force” dla częstotliwości f_{max} , f_{start} , f_{stop} oraz f_{step} równych odpowiednio 2.5, 2.4, 1.0 oraz 0.2 GHz, wymaga przetestowania $TC = 45$ konfiguracji częstotliwości. Dla każdej z tych konfiguracji przeprowadzane są testy zużycia energii, których czas trwania powinien być wystarczająco długi, aby zapewnić stabilność pomiarów. Z tego względu znalezienie optymalnej konfiguracji częstotliwości tym podejściem może okazać się wysoce czasochłonne.

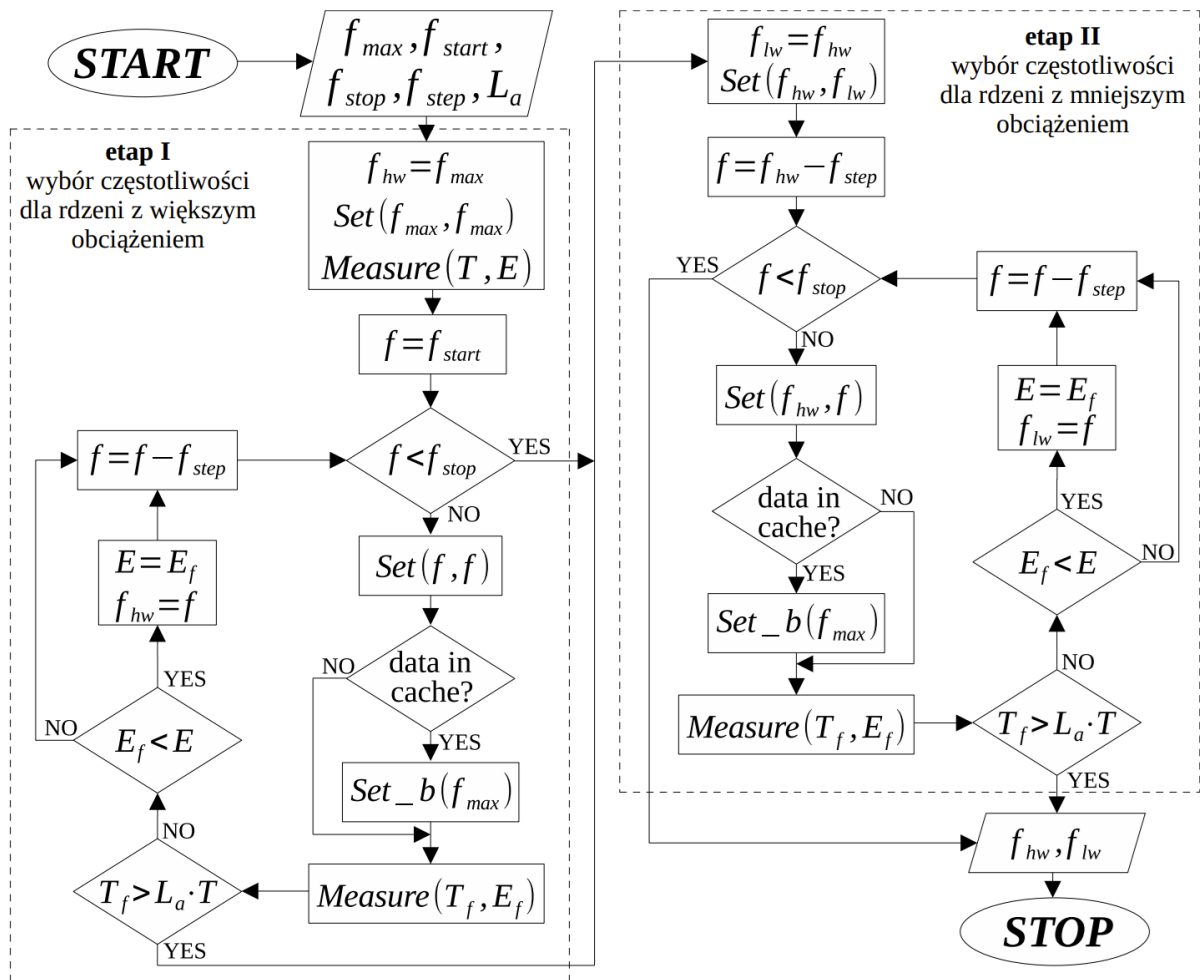
W niniejszej sekcji przedstawione zostaną dwie wersje algorytmu, służącego do wyboru optymalnej heterogenicznej konfiguracji DVFS. Zaproponowane algorytmy pozwalają na istotne ograniczenie liczby testowanych konfiguracji, a co za tym idzie przyspieszenie procesu odnajdywania optymalnych konfiguracji częstotliwości.

2.5.1. Algorytm bazujący na pomiarze energii elektrycznej

W pierwszej wersji algorytmu decyzje podejmowane są na podstawie pomiarów zużycia energii elektrycznej. Opracowany algorytm przedstawiono na rysunku 2.5. Składa się on z dwóch etapów, których celem jest wyznaczenie optymalnych częstotliwości f_{hw} oraz f_{lw} dla rdzeni odpowiednio o wyższym i niższym poziomie obciążenia. Parametrem wejściowym algorytmu jest wartość L_a , określająca dopuszczalny spadek wydajności. Przykładowo, wartość $L_a = 1.02$ oznacza, że częstotliwość może być obniżana do momentu, w którym czas wykonania zwiększy się maksymalnie o 2%.

Wybór konkretnej konfiguracji częstotliwości realizowany jest za pomocą procedury $Set(f_1, f_2)$, która ustawia częstotliwość f_1 dla grupy rdzeni o wyższym obciążeniu oraz częstotliwość f_2 dla grupy rdzeni o niższym obciążeniu. Zarządzanie częstotliwością rdzeni odpowiedzialnych za komunikację między procesorami odbywa się za pomocą procedury $Set_b(f)$. Procedura $Measure(T, E)$ służy natomiast to pomiaru czasu wykonania T oraz zużycia energii E dla aktualnej konfiguracji częstotliwości. W celu zapewnienia dokładności wyników, wszystkie realizowane pomiary mogą zostać powtórzone m razy, a obowiązująca wartość przedstawiona jako mediana uzyskanych pomiarów (w badaniach przyjęto $m = 5$).

Przedstawiony algorytm sprawdza również, czy wszystkie dane mieszczą się w pamięci podręcznej w celu wyboru pomiędzy scenariuszem *in-cache* lub *out-of-cache*. Spełnienie tego warunku oznacza, iż rdzenie odpowiedzialne za komunikację między procesorami charakteryzują się najdłuższym czasem rzeczywistych obliczeń (rysunek 2.1a i 2.1b).



Rysunek 2.5. Schemat blokowy algorytmu wyboru heterogenicznej konfiguracji częstotliwości w oparciu o pomiary energii elektrycznej

W takim przypadku najbardziej efektywną strategią jest przypisanie tym rdzeniom maksymalnej możliwej częstotliwości pracy, co umożliwia zminimalizowanie narzutów czasowych, wynikających z wymiany danych.

2.5.2. Algorytm bazujący na pomiarze czasu

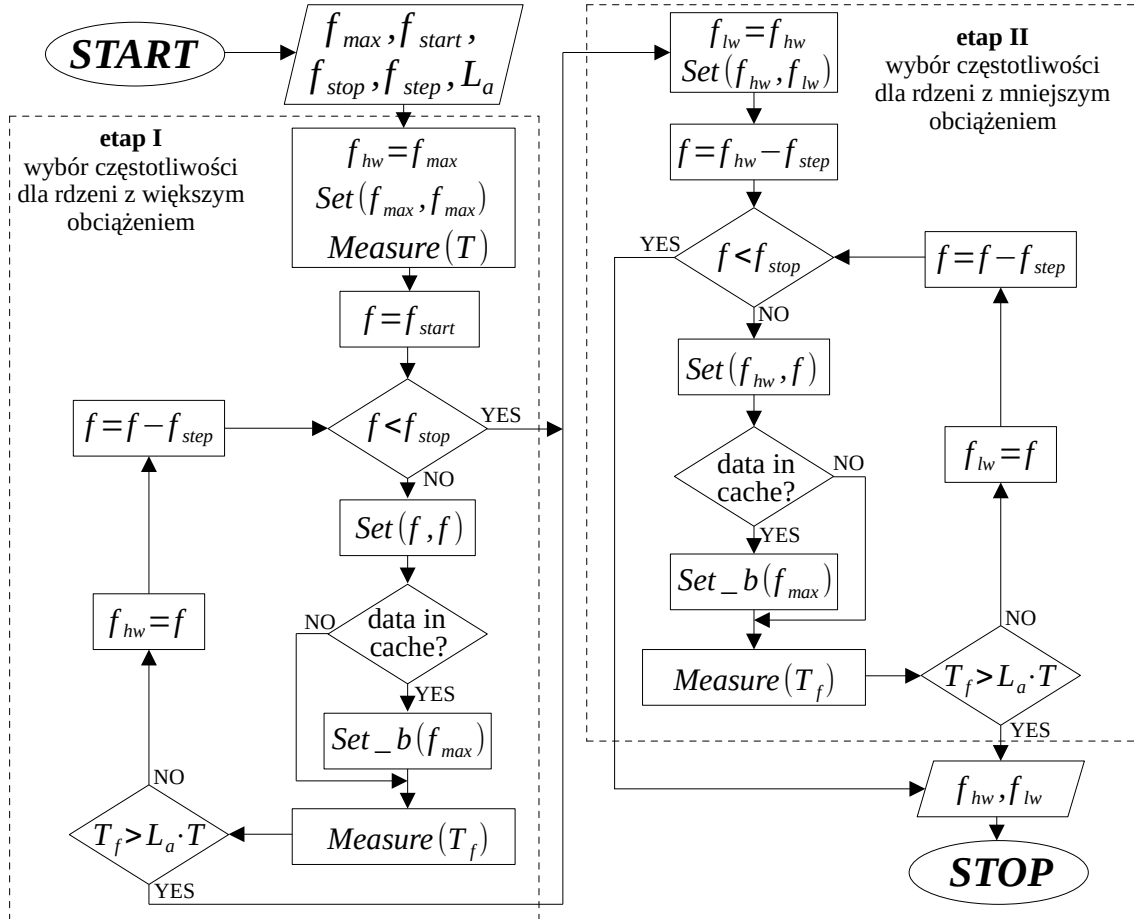
Implementacja algorytmu przedstawionego w poprzedniej sekcji wymaga wykorzystania narzędzi, umożliwiających wykonywanie pomiarów energii elektrycznej. Monitorowanie zużycia energii jest powszechnie dostępne zarówno z wykorzystaniem technik sprzętowych, jak i programowych [73]. W badaniach przedstawionych w niniejszym rozdziale zastosowano podejście sprzętowe, wykorzystując cyfrowy miernik mocy Yokogawa WT310, pozwalający na monitorowanie zużycia energii przez całą platformę obliczeniową. Alternatywnym rozwiązaniem może być interfejs Intel Running Average Power Limit (RAPL) [31], będący przykładem rozwiązania programowego. W przeciwieństwie do mierników cyfrowych, RAPL umożliwia odczyty energii dla różnych komponentów platformy obliczeniowej, w tym dla pakietów CPU oraz pamięci DRAM.

Należy jednak zauważyć, że oba podejścia wiążą się z istotnymi ograniczeniami. W przypadku rozwiązań sprzętowych konieczne jest fizyczne posiadanie odpowiedniego urządzenia pomiarowego oraz jego bezpośrednie podłączenie do platformy obliczeniowej. W praktyce, zwłaszcza w przypadku korzystania z zasobów obliczeniowych należących do zewnętrznych instytucji lub centrów danych, dostęp do infrastruktury fizycznej może być znacznie utrudniony lub wręcz niemożliwy, co uniemożliwia zastosowanie tego podejścia. Przechodząc do ograniczeń podejścia programowego, należy zauważyć, iż narzędzia oparte na RAPL nie zawsze są dostępne na platformach z procesorami firmy AMD, co skutkuje brakiem możliwości przeprowadzania pomiarów energii. Dodatkowo, standardowe narzędzia wykorzystujące RAPL (takie jak PAPI [76] czy LIKWID [78]) niekiedy nie wspierają poprawnie najnowszych architektur sprzętowych, co utrudnia ich zastosowanie.

W celu zwiększenia możliwości zastosowania opracowanej heterogenicznej strategii DVFS, zaproponowano rozwiązanie niewymagające pomiaru zużycia energii. Zamiast tego, decyzje podejmowane są na podstawie czasu wykonania aplikacji przy zadanej częstotliwości. Podejście to znacząco rozszerza zakres zastosowań opracowanej metody, ponieważ pomiary czasu nie wymagają zazwyczaj dodatkowego nakładu pracy i są łatwe do wdrożenia nawet dla użytkowników nieposiadających zaawansowanej wiedzy technicznej.

Rysunek 2.6 przedstawia opracowany algorytm bazujący na pomiarach czasu wykonania. Jego koncepcja jest zbliżona do algorytmu wykorzystującego pomiary zużycia energii. Parametry wejściowe oraz procedury odpowiedzialne za ustawianie częstotliwości dla grup rdzeni pozostają niezmienione. Modyfikacji poddano jedynie procedurę *Measure*,

która w nowej wersji przyjmuje wyłącznie parametr związany z czasem wykonania. Kluczową różnicą jest rezygnacja z analizy zużycia energii przy kolejnych uruchomieniach aplikacji. Na każdym etapie działania algorytmu wybierana jest najniższa częstotliwość, która nie skutkuje przekroczeniem zadanego limitu czasu wykonania.



Rysunek 2.6. Schemat blokowy algorytmu wyboru heterogenicznej konfiguracji częstotliwości w oparciu o pomiary czasu trwania aplikacji

2.6. Badanie efektywności opracowanych algorytmów

Opracowane algorytmy zostały poddane badaniom mającym na celu określenie ich skuteczności na przykładzie obu rozważanych aplikacji. Celem analiz było porównanie rezultatów trzech podejść: (i) BF – metody „brute force” będącej punktem odniesienia jako strategia zapewniająca rozwiązanie optymalne, (ii) AE – algorytmu bazującego na pomiarach zużycia energii, oraz (iii) AT – algorytmu opartego na pomiarach czasu wykonania. Dla każdego z podejść określono liczbę testowanych konfiguracji (TC), niezbędną do uzyskania wyniku końcowego przez daną strategię, a także poziom osiągniętej redukcji zużycia energii. Dodatkowo, dla obu opracowanych algorytmów wyznaczono

efektywność, zdefiniowaną jako stosunek (wyrażony w procentach) redukcji zużycia energii osiągniętej przez dany algorytm do wartości uzyskanej metodą „brute force”. Porównanie przeprowadzono dla dziewięciu rozmiarów dziedziny, odpowiadających różnym wartościom parametrów R i LIP .

		$R = 2$			$R = 3/2$			$R = 4/3$		
LIP [%]		8	25	50	8	25	50	8	25	50
Rozmiar domeny		78	90	108	150	162	180	222	234	252
BF	f_{hw} [GHz]	2.4	2.0	1.8	2.4	2.2	2.0	2.4	2.2	1.4
	f_{lw} [GHz]	1.8	1.8	1.4	1.2	1.2	1.4	1.0	1.2	1.2
	f_{bs} [GHz]	2.5	2.5	2.5	–	–	–	–	–	–
	TC	45	45	45	45	45	45	45	45	45
	Redukcja energii [%]	16.6	17.1	22.9	25.0	17.9	16.2	19.5	17.0	21.6
AE	f_{hw} [GHz]	2.4	2.0	1.6	2.0	2.2	2.0	2.2	1.8	1.4
	f_{lw} [GHz]	1.8	1.8	1.2	1.0	1.2	1.4	1.2	1.4	1.2
	f_{bs} [GHz]	2.5	2.5	2.5	–	–	–	–	–	–
	TC	8	7	11	10	10	9	10	9	10
	Redukcja energii [%]	16.6	17.1	22.4	24.0	17.9	16.2	18.3	16.0	21.6
Efektywność [%]		100	100	97.9	95.8	100	100	93.9	93.8	100
AT	f_{hw} [GHz]	2.4	2.0	1.4	2.0	2.2	2.0	2.2	1.8	1.4
	f_{lw} [GHz]	1.6	1.8	1.2	1.0	1.2	1.4	1.2	1.4	1.0
	f_{bs} [GHz]	2.5	2.5	2.5	–	–	–	–	–	–
	TC	8	7	10	10	10	9	10	9	10
	Redukcja energii [%]	15.9	17.1	20.7	24.0	17.9	16.2	18.3	16.0	21.0
Efektywność [%]		95.7	100	90.4	95.8	100	100	93.9	93.8	97.1

Tabela 2.2: Porównanie podejścia „brute force” z opracowanymi algorytmami (AE – bazujący na pomiarach energii, AT – bazujący na pomiarach czasu) dla trójwymiarowej aplikacji dyfuzji i sześciennych domen obliczeniowych ($X = Y = Z$)

Tabela 2.2 przedstawia wyniki uzyskane dla aplikacji trójwymiarowej dyfuzji. Naturalnym następstwem testowania wszystkich możliwych konfiguracji jest to, że strategia BF zawsze zapewnia najlepsze rezultaty w zakresie redukcji zużycia energii. Oba opracowane algorytmy osiągają jednak co najmniej 90-procentową efektywność we wszystkich testach. Co więcej, algorytm oparty na pomiarach energii (AE) w ponad połowie przypadków osiąga 100-procentową skuteczność a jego odpowiednik oparty na czasie wykonania (AT) osiąga ten wynik w trzech z dziewięciu analizowanych przypadków.

Przykładowo, w przypadku domeny o rozmiarze $78 \times 78 \times 78$, optymalne rozwiązanie polega na ustawieniu częstotliwości dla grup rdzeni o większym i mniejszym obciążeniu odpowiednio na $f_{hw} = 2.4$ GHz oraz $f_{lw} = 1.8$ GHz. Jednocześnie rdzenie odpowiedzialne za komunikację między procesorami powinny pracować z maksymalną dostępną

częstotliwością $f_{bs} = 2.5$ GHz. Takie ustawienie pozwoliło na zmniejszenie zużycia energii o 16.6%, przy jednoczesnym nieprzekroczeniu progu 2-procentowego wzrostu czasu wykonania względem wyniku uzyskanego dla homogenicznej częstotliwości 2.5 GHz. W omawianej domenie, efektywność strategii AE wyniosła 100%, co oznacza, że zarekomendowała ona taki sam zestaw częstotliwości jak metoda BF. Dla porównania, różnica w algorytmie AT polegała na propozycji ustawienia częstotliwości dla rdzeni o mniejszym obciążeniu na poziomie 1.6 GHz, co w rezultacie pozwoliło na redukcję zużycia energii o 15.9%, odpowiadającą efektywności równej 95.7%.

Wyniki uzyskane dla aplikacji trójwymiarowej dyfuzji wskazują, iż opracowane algorytmy w znaczący sposób redukują koszt wyboru heterogenicznych konfiguracji DVFS. Przykładowo, podczas gdy metoda „brute force” wymagała każdorazowego przetestowania 45 konfiguracji częstotliwości, oba algorytmy potrzebowały jedynie od 7 do 11 konfiguracji.

		$R = 2$			$R = 3/2$			$R = 4/3$		
LIP [%]		8	25	50	8	25	50	8	25	50
Domain size		78	90	108	150	162	180	222	234	252
BF	f_{hw} [GHz]	2.4	2.2	1.6	2.4	2.0	1.4	2.0	1.4	1.4
	f_{lw} [GHz]	1.2	1.0	1.4	1.2	1.0	1.2	1.0	1.0	1.0
	TC	45	45	45	45	45	45	45	45	45
	Redukcja energii [%]	30.0	24.1	23.7	19.8	21.2	24.2	22.1	22.7	23.4
AE	f_{hw} [GHz]	2.0	2.0	2.0	2.2	2.0	1.4	2.0	1.6	1.8
	f_{lw} [GHz]	1.2	1.2	1.2	1.4	1.0	1.2	1.0	1.0	1.0
	TC	10	10	10	9	10	10	10	10	10
	Redukcja energii [%]	28.7	24.0	23.1	18.9	21.2	24.2	22.1	21.5	21.1
	Efektywność [%]	95.5	99.2	97.5	95.7	100	100	100	95.0	90.3
AT	f_{hw} [GHz]	2.0	2.0	2.0	2.2	2.0	1.4	2.0	1.6	1.8
	f_{lw} [GHz]	1.2	1.0	1.2	1.4	1.0	1.0	1.0	1.0	1.0
	TC	10	10	10	9	10	10	10	10	10
	Redukcja energii [%]	28.7	22.1	23.1	18.9	21.2	23.4	22.1	21.5	21.1
	Efektywność [%]	95.5	91.5	97.5	95.7	100	96.6	100	95.0	90.3

Tabela 2.3: Porównanie podejścia „brute force” z opracowanymi algorytmami (AE – bazujący na pomiarach energii, AT – bazujący na pomiarach czasu) dla aplikacji MPDATA i sześciennych domen obliczeniowych ($X = Y = Z$)

Analogiczne wnioski pozostają aktualne dla drugiego przypadku użycia – aplikacji MPDATA. Z tabeli 2.3 wynika, że we wszystkich testach efektywność obu algorytmów przekraczała 90%. Uzyskanie rozwiązania wymagało sprawdzenia od 9 do 10 konfiguracji testowych, w porównaniu do stałej liczby 45 konfiguracji, wymaganych przez podejście „brute force”.

Najniższą efektywność zaobserwowano dla domeny o rozmiarze $252 \times 252 \times 252$. Ustawienie częstotliwości na poziomie 1.8 GHz i 1.0 GHz pozwoliło na redukcję zużycia energii o 21.1%, co odpowiada efektywności równej 90.3%. W tym przypadku strategia BF zapewniła lepszy rezultat, obniżając częstotliwość rdzeni o większym obciążeniu do 1.4 GHz.

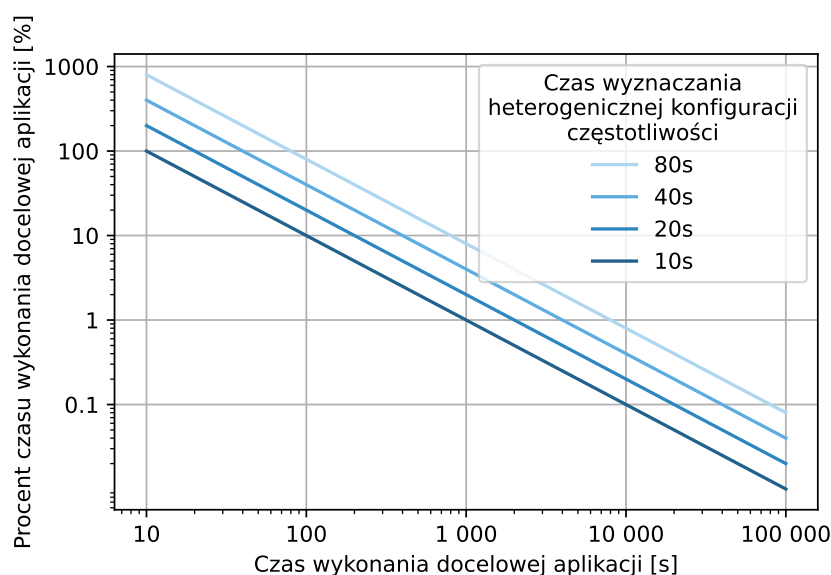
W przypadku dużych domen ($R = 4/3$ w tabeli 2.3) częstotliwość dla grupy rdzeni o mniejszym obciążeniu wyniosła 1.0 GHz w obu algorytmach. Sugeruje to, że wyniki redukcji zużycia energii mogłyby być jeszcze korzystniejsze, gdyby platforma obliczeniowa umożliwiała dalsze obniżanie częstotliwości taktowania procesorów.

2.7. Ocena kosztu zastosowania opracowanej metody

Opracowane algorytmy wyznaczania heterogenicznej konfiguracji częstotliwości zakładają, że proces poszukiwania odbywa się w trybie offline, tj. przed właściwym, docelowym uruchomieniem aplikacji. Podejście to wiąże się więc z dodatkowym narzutem czasowym, którego wielkość jest ściśle zależna od konkretnego przypadku.

Tabele 2.2 oraz 2.3, zamieszczone w poprzedniej sekcji, przedstawiają zachowanie algorytmów, wywnioskowane na podstawie eksperymentów, w których aplikacje były uruchamiane na kilkaset sekund dla każdej analizowanej konfiguracji częstotliwości.

W praktyce, czas trwania testów może być jednak znacznie krótszy. Przeprowadzone badania dla obu aplikacji pokazały, że symulacje trwające około 10 sekund zapewniają wystarczającą dokładność pomiarów czasu i zużycia energii. Należy zauważyć, że zarządzanie czasem działania analizowanych aplikacji jest stosunkowo proste, ponieważ jest on liniowo zależny od liczby kroków czasowych. Liczba ta na potrzeby testów może być ustalona dowolnie, ponieważ nie wpływa na rozkład obciążenia opisany przez parametry R i LIP . Uwzględniając stosunkowo niewielką liczbę niezbędnych konfiguracji testowych ($TC \leq 11$ w przeprowadzonych eksperymentach), całkowity koszt opracowanej strategii można oszacować na poziomie kilkudziesięciu sekund. Rysunek 2.7 ilustruje wpływ tego narzutu na ogólną wydajność w zależności od zakładanej długości trwania docelowej aplikacji.



Rysunek 2.7. Szacowany koszt poszukiwania heterogenicznej konfiguracji częstotliwości w oparciu o opracowane algorytmy

W przypadkach, gdy przewidziane jest jednorazowe, krótkie uruchomienie aplikacji, narzut związany z doбором heterogenicznej konfiguracji częstotliwości może okazać się zbyt wysoki. W wielu praktycznych scenariuszach analizowanych aplikacji obliczenia trwają jednak wiele godzin, co powoduje, że narzut ten staje się pomijalny. Przykładowo, zakładając że całkowity czas trwania docelowej aplikacji wyniesie przynajmniej dwie godziny, narzut związany z zastosowaniem opracowanej metody nie przekroczy 1%. Ponadto, konfiguracje częstotliwości uzyskane za pomocą opracowanych algorytmów mogą być wykorzystywane wielokrotnie. Przykładem może być dziedzina numerycznej prognozy pogody, w której symulacje dla ustalonego rozmiaru domeny i określonej architektury sprzętowej są powtarzane nawet tysiące razy przy użyciu algorytmu MPDATA.

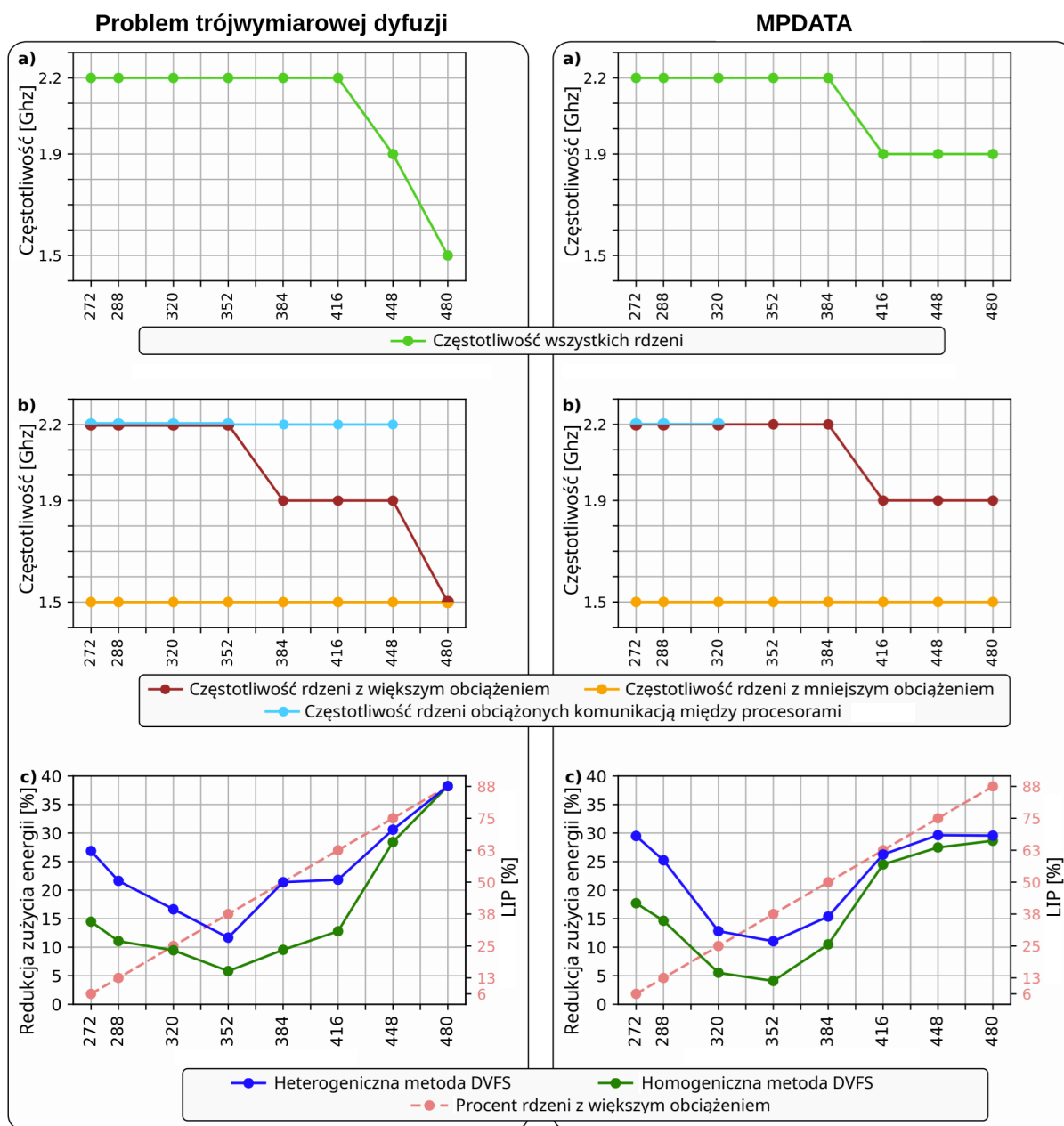
2.8. Weryfikacja opracowanej metody z wykorzystaniem platformy AMD

Opracowana metoda została również przetestowana na platformie obliczeniowej wyposażonej w dwa procesory AMD EPYC 7773X (Milan-X) [6]. Każdy z nich zawiera 64 rdzenie zorganizowane w osiem jednostek Core Complex Die oraz oferuje łącznie 768 MB pamięci podręcznej trzeciego poziomu (L3). W rezultacie cała platforma obliczeniowa udostępnia 256 rdzeni logicznych oraz ponad 1.5 GB pamięci podręcznej ostatniego poziomu.

Dostępność tak dużej liczby rdzeni w platformie opartej na procesorach AMD EPYC

umożliwia przeprowadzenie ewaluacji opracowanego podejścia dla większych rozmiarów domen obliczeniowych. Podczas gdy w przypadku architektury Intel Cascade Lake wartość $R = 2$ odpowiada rozmiarom domen z zakresu (72, 144), dla AMD EPYC analogiczna wartość obejmuje zakres (256, 512). Utrudnienie stanowi fakt, że procesory AMD Milan-X, w porównaniu do układów firmy Intel, charakteryzują się ograniczonym zakresem zarządzania częstotliwością. Dostępne są jedynie cztery poziomy częstotliwości: 1.5 GHz, 1.9 GHz, 2.2 GHz oraz tryb turbo z maksymalnym taktowaniem do 3.5 GHz. Eksperymenty przeprowadzono w sposób analogiczny do przedstawionej w sekcji 2.4 metody „brute force”. Dla obu analizowanych aplikacji wybrano ten sam zestaw ośmiu rozmiarów domen, charakteryzujących się współczynnikiem $R = 2$ oraz parametrem LIP , zmieniającym się w zakresie od 6% do 88%. Rysunek 2.8 przedstawia podsumowanie przeprowadzonych testów dla obu analizowanych aplikacji.

Założenia eksperymentalne były tożsame z przyjętymi podczas badań wykonanych na platformie Intel. Poszukiwano konfiguracji częstotliwości minimalizujących zużycie energii przy jednoczesnym nieprzekroczeniu 2-procentowego spadku wydajności względem konfiguracji z pełną częstotliwością turbo. Górne i środkowe wykresy ilustrują najlepsze konfiguracje częstotliwości odpowiednio dla podejścia homogenicznego i heterogenicznego, natomiast dolne wykresy przedstawiają porównanie tych podejść pod kątem uzyskanych oszczędności energetycznych. Jak wynika z rysunku 2.8, opracowane podejście heterogeniczne zapewnia lepsze wyniki niż strategia homogeniczna. W obu aplikacjach obserwuje się podobną tendencję – największe korzyści z heterogenicznego DVFS uzyskiwane są dla najniższych wartości parametru LIP . Różnica w oszczędnościach energetycznych pomiędzy wariantem homogenicznym i heterogenicznym sięga 13 punktów procentowych dla aplikacji trójwymiarowej dyfuzji oraz 12 punktów procentowych dla aplikacji MPDATA.



Rysunek 2.8. Porównanie redukcji zużycia energii elektrycznej uzyskanego za pomocą homogenicznego oraz heterogenicznego wariantu metody DVFS na platformie obliczeniowej firmy AMD

3. Adaptacja realizacji zapytań zliczających do wysokowydajnych systemów komputerowych

Niniejszy rozdział stanowi wprowadzenie do tematyki adaptacji realizacji zapytań zliczających do wysokowydajnych systemów komputerowych. W pierwszej kolejności przedstawiano podstawowe informacje dotyczące zapytań zliczających oraz ich zastosowań w uczeniu maszynowym. Następnie omówiono motywację do podjęcia badań w tym obszarze. Kolejna sekcja dotyczy biblioteki SABNAtk, stanowiącej przykład narzędzia do efektywnej realizacji zapytań zliczających. Ostatnia sekcja poświęcona jest przyjętej metodologii ewaluacji, wspólnej dla wszystkich przeprowadzonych badań. Stanowi ona wprowadzenie do dalszych rozdziałów (4 – 7), w których zaprezentowano metody oraz algorytmy opracowane przez autora w ramach omawianej tematyki.

3.1. Zapytania zliczające w uczeniu maszynowym

Zapytania zliczające są podstawową operacją wykorzystywaną przez algorytmy uczenia maszynowego. Ich zadaniem jest zliczanie rekordów danych pasujących do określonych konfiguracji wybranych zmiennych. Problem ten pojawia się za każdym razem, gdy należy oszacować rozkład prawdopodobieństwa i dlatego znajduje zastosowanie w wielu obszarach. Przykładowo, podczas uczenia struktury sieci Bayesowskiej [34] na podstawie danych, zapytania zliczające są niezbędne w funkcji oceny (scoring function), która mierzy, jak dobrze dana struktura sieci wyjaśnia obserwowane dane. Innym przykładem może być eksploracja reguł asocjacyjnych [5], w której zapytania zliczające dotyczą danych binarnych, reprezentujących transakcje i umożliwiają ocenę wsparcia oraz pewności dla danej reguły asocjacyjnej. Inne kluczowe obszary zastosowań obejmują klasyfikację [32, 50], głębokie uczenie [38, 59] i wyszukiwanie informacji [51].

Aby w pełni przybliżyć tematykę zapytań zliczających, należy najpierw wprowadzić pewne formalne założenia oraz definicje. Niech dany będzie zbiór n zmiennych losowych o charakterze kategoriowym $\mathcal{X} = X_1, X_2, \dots, X_n$, w którym każda zmienna X_i może przyjmować różną liczbę stanów. Ponadto, niech $D = [D_1, D_2, \dots, D_m]$ będzie kompletną bazą danych, zawierającą m obserwacji zmiennych z $\mathcal{X}$, w której każdy wiersz D_i przechowuje informacje o zaobserwowanych stanach wszystkich n zmiennych z $\mathcal{X}$. Na rysunku 3.1 zaprezentowany jest przykład takiej bazy, składającej się z trzech zmiennych oraz siedmiu obserwacji.

	X_1	X_2	X_3
D_1	2	0	1
D_2	0	1	0
D_3	1	0	1
D_4	2	0	1
D_5	1	1	0
D_6	2	0	1
D_7	0	2	1

Rysunek 3.1. Przykład prostej bazy danych

Dla danej bazy D oraz zestawu zmiennych $X_i, X_j, \dots \subseteq \mathcal{X}$, zapytanie zliczające $\text{COUNT}((X_i = x_i) \wedge (X_j = x_j) \wedge \dots)$ zwraca liczbę przypadków z bazy danych D , które są zgodne z daną konfiguracją $[x_i, x_j, \dots]$ zmiennych $[X_i, X_j, \dots]$. Na przykład, odpowiedzią na zapytanie $\text{COUNT}((X_1 = 2) \wedge (X_2 = 0) \wedge (X_3 = 1))$ zrealizowane w ramach bazy danych przedstawionej na rysunku 3.1 jest 3, ponieważ istnieją trzy rekordy spełniające warunek zapytania. Jednym z podstawowych przykładów zastosowań takich zapytań jest estymacja prawdopodobieństw wspólnych, takich jak $p(X_i = x_i, X_j = x_j)$ poprzez obliczenie

$$\frac{\text{COUNT}((X_i = x_i) \wedge (X_j = x_j))}{m}. \quad (3.1)$$

Operacja ta stanowi podstawę metod takich jak wydobywanie reguł asocjacyjnych oraz uczenie probabilistycznych modeli graficznych.

W wielu praktycznych sytuacjach, proste zapytania zliczające COUNT są wykonywane w partiach, składających się z kolejnych zapytań dotyczących tego samego zestawu zmiennych. Taką partię zapytań można nazwać zapytaniami o wspólnym kontekście i traktować jako jedno (bardziej złożone) zapytanie $\text{COUNT}(X_i, X_j, \dots)$, w którym zadaniem jest uzyskanie niezerowych odpowiedzi dla zapytań COUNT , utworzonych dla wszystkich możliwych konfiguracji zmiennych wchodzących w skład zapytania. Na przykład, wynikiem wykonania zapytania $\text{COUNT}(X_2, X_3)$ jest następująca lista:

$$\left[\left((0, 1), 4 \right), \left((1, 0), 2 \right), \left((2, 1), 1 \right) \right], \quad (3.2)$$

w której każdy element, to para zapisana w następującej postaci:

$$\left((x_i, x_j), \text{COUNT}((X_2 = x_i) \wedge (X_3 = x_j)) \right). \quad (3.3)$$

Przykładowo, pierwszym elementem ww. listy jest para $((0, 1), 4)$, ponieważ w bazie istnieją cztery obserwacje, spełniające warunek $(X_2 = 0) \wedge (X_3 = 1)$. Są nimi D_1 , D_3 , D_4 oraz D_6 .

Oznaczając przez $Pa(X_i)$ podzbiór $\mathcal{X} - X_i$, nazywany również zbiorem rodziców, możliwe jest sformułowanie zapytania typu **COUNT**($X_i \mid Pa(X_i)$). Jego celem jest uszczegółowienie uzyskiwanych odpowiedzi na zapytanie **COUNT**($Pa(X_i)$) w zależności od możliwych stanów danej zmiennej X_i . Rysunek 3.2 pokazuje odpowiedzi na przykładowe zapytanie **COUNT**($X_1 \mid \{X_2, X_3\}$) w formie tabeli kontyngencji zawierającej jedynie niezerowe odpowiedzi.

		$Pa\{X_1\} = \{X_2, X_3\}$					
		00	01	10	11	20	21
X_1	0			1			1
	1		1	1			
	2		3				

Rysunek 3.2. Tabela kontyngencji dla zapytania zliczającego **COUNT**($X_1 \mid \{X_2, X_3\}$)

Przed przejściem do praktycznych przykładów zastosowań tego rodzaju zapytań zliczających, konieczne jest wprowadzenie pewnych oznaczeń. Niech N_{ij} będzie liczbą instancji z bazy danych D , dla których zmienne ze zbioru $Pa(X_i)$ znajdują się w konfiguracji j . Z kolei niech N_{ijk} uściśla definicję N_{ij} dla konkretnej konfiguracji zmiennej X_i , tzn. określa liczbę instancji z bazy danych D , dla których zmienne ze zbioru $Pa(X_i)$ znajdują się w konfiguracji j oraz dodatkowo zmienna X_i przyjmuje stan k .

Przykłady wprowadzonych oznaczeń prezentowane są na rysunku 3.3. Przedstawia on omawianą już tabelę kontyngencji, w której dodatkowo ponumerowano możliwe stany zmiennej X_1 oraz możliwe konfiguracje zmiennych wchodzących w skład zbioru $Pa(X_1)$.

$Pa\{X_1\} = \{X_2, X_3\}$

$j=0 \quad j=1 \quad j=2 \quad j=3 \quad j=4 \quad j=5$

		00	01	10	11	20	21	
$k=0$	0		1	1			1	→ $N_{150}=1$
$k=1$	1		1	1				
$k=2$	2		3					

$N_{11}=4$

Rysunek 3.3. Interpretacja liczb N_{ij} oraz N_{ijk} na przykładzie tabeli kontyngencji

Można zauważyć, że wartości N_{ijk} to nic innego, jak pojedyncze liczby z tabeli, czyli odpowiedzi na zapytania typu $\mathbf{COUNT}(X_i \mid Pa(X_i))$. Rozpatrując konkretną wartość, np. N_{150} , jest ona równa 1, ponieważ w bazie danych istnieje dokładnie jeden rekord, dla którego zmienna $X_i = X_1$ przyjmuje stan $k = 0$, gdy zmienne ze zbioru $Pa(X_1)$ znajdują się w konfiguracji "21" oznaczonej jako $j = 5$.

Przechodząc do N_{ij} można zaobserwować, że dla ustalonych i oraz j , zachodzi następująca własność:

$$N_{ij} = \sum_k N_{ijk}, \quad (3.4)$$

tzn. wartość N_{ij} jest sumą wszystkich wartości N_{ijk} , dla danych i oraz j . Wszystkie niezerowe wartości N_{ij} dla omawianej tabeli kontyngencji to:

$$N_{11} = N_{110} + N_{111} + N_{112} = 0 + 1 + 3 = 4, \quad (3.5)$$

$$N_{12} = N_{120} + N_{121} + N_{122} = 1 + 1 + 0 = 2, \quad (3.6)$$

$$N_{15} = N_{150} + N_{151} + N_{152} = 1 + 0 + 0 = 1. \quad (3.7)$$

Warto zauważyć, że wartości N_{ij} są odpowiedziami na wprowadzone wcześniej zapytania typu $\mathbf{COUNT}(X_i, X_j, \dots)$. Przykładowo, stosując wprowadzoną notację do rozpatrywanego wcześniej zapytania $\mathbf{COUNT}(X_2, X_3)$, jego odpowiedzi (oznaczone wzorem 3.2) mogą zostać również przedstawione w następujący sposób:

$$\left[\left(j = 1, N_{11} = 4 \right), \left(j = 2, N_{12} = 2 \right), \left(j = 5, N_{15} = 1 \right) \right]. \quad (3.8)$$

W celu lepszego przybliżenia tematyki zapytań zliczającej, przedstawione zostaną praktyczne przykłady ich zastosowań. Pierwszy przykład polega na obliczeniu metryki $BDeu$ (Bayesian Dirichlet equivalent uniform), która jest powszechnie stosowana w uczeniu sieci Bayesowskich [22]:

$$BDeu(X_i|Pa(X_i); \alpha) = \sum_{j=1}^{q_i} \left[\log \left(\frac{\Gamma\left(\frac{\alpha}{q_i}\right)}{\Gamma\left(N_{ij} + \frac{\alpha}{q_i}\right)} \right) + \sum_{k=1}^{r_i} \log \left(\frac{\Gamma\left(N_{ijk} + \frac{\alpha}{r_i \cdot q_i}\right)}{\Gamma\left(\frac{\alpha}{r_i \cdot q_i}\right)} \right) \right] \quad (3.9)$$

W powyższym równaniu α jest hiperparametrem a $\Gamma(\cdot)$ to funkcja Gamma. $Pa(X_i)$ oznacza zbiór zmiennych objaśniających dla X_i a zmienna j iteruje przez wszystkie możliwe stany zmiennych w $Pa(X_i)$, których liczba wynosi w sumie

$$q_i = \prod_{X_j \in Pa(X_i)} r_j, \quad (3.10)$$

gdzie r_j , to liczba stanów przyjmowanych przez zmienną X_j .

Aby obliczyć $BDeu$, należy dysponować wartościami N_{ij} oraz N_{ijk} , co przekłada się na wiele zapytań zliczających na tym samym zbiorze zmiennych $Pa(X_i) \cup X_i$, testujących różne konfiguracje ich stanów. Co więcej, istotne są tylko zapytania, które zwracają niezerowe wartości N_{ijk} , ponieważ tylko takie wartości mają wkład w końcową sumę. Warto również zauważyć, że niezerowe wartości N_{ijk} implikują niezerowe N_{ij} .

Innym przykładem może być obliczanie zysku informacyjnego (information gain), który jest kryterium stosowanym w wielu algorytmach budowy drzew decyzyjnych (np. ID3, C4.5 oraz ich uogólnieniach, takich jak lasy losowe) do wyboru atrybutu rozgałęzienia [49]:

$$IG(X_i, Pa(X_i)) = H(X_i) - \sum_{j=1}^{q_i} \left(\frac{N_{ij}}{m} \cdot \sum_{k=1}^{r_i} \frac{N_{ijk}}{N_{ij}} \log \left(\frac{N_{ij}}{N_{ijk}} \right) \right). \quad (3.11)$$

W powyższym równaniu X_i jest zmienną klasyfikacyjną, $Pa(X_i)$ to atrybut rozważany jako kandydat do rozgałęzienia drzewa, a $H(X_i)$ to entropia zmiennej X_i , którą można obliczyć jako:

$$H(X_i) = - \sum_{k=1}^{r_i} \frac{N_{i \cdot k}}{m} \log \left(\frac{N_{i \cdot k}}{m} \right), \quad (3.12)$$

gdzie $N_{i \cdot k}$ to liczba instancji w D , w których zmienna X_i znajduje się w stanie k . Podobnie jak w pierwszym przykładzie, aby obliczyć IG , należy wykonać wiele zapytań zliczających o wspólnym kontekście, a następnie zagregować je przez sumowanie.

3.2. Motywacja do badań

Istotność zapytań zliczających wynika z faktu, że w wielu praktycznych przypadkach proces ich realizacji zajmuje nawet ponad 90% całkowitego czasu trwania danej aplikacji [30]. Jednocześnie okazuje się, że często ich znaczenie jest pomijane poprzez implementację za pomocą prostych, ale niekoniecznie wydajnych strategii, np. wspomnianych tabel kontyngencji. Uwzględniając zauważoną wcześniej mnogość obszarów, w których zapytania zliczające stanowią integralną część realizacji danego zadania, można dojść do wniosku, że poprawa wydajności wykonywania zapytań zliczających jest istotnym celem, którego osiągnięcie może w bezpośredni sposób przełożyć się na poprawę wydajności wielu aplikacji.

Jak już wspomniano, skorzystanie z tabeli kontyngencji to jedna z najprostszych metod na zrealizowanie zapytania zliczającego ze wspólnym kontekstem (**COUNT**). Polega ona na przeszukaniu bazy danych D w celu utworzenia tabeli reprezentującej wszystkie możliwe konfiguracje zmiennych wchodzących w skład zapytania. Strategia ta okazuje się jednak skuteczna jedynie dla stosunkowo małych zapytań. Powodem jest to, że liczba kombinacji niezbędnych do przejrzania rośnie w tempie wykładniczym w zależności od liczby zmiennych oraz stanów, które mogą one przyjmować. W efekcie konstruowana tabela szybko osiąga ogromne rozmiary, co bardzo często prowadzi do przekroczenia możliwości pamięciowych nawet najlepszych serwerów obliczeniowych [17]. Przykładowo, zakładając, że każda z rozpatrywanych zmiennych zapytania **COUNT** może przyjmować trzy stany, zapytanie składające się z pięciu zmiennych wymaga przeanalizowania 243 (3^5) kombinacji. Rozpatrując dziesięć zmiennych będzie to już jednak prawie 60 tysięcy konfiguracji a przy 30 zmiennych ponad 200 ... bilionów. Zakładając że każda konfiguracja zajmuje jedynie cztery bajty, przechowywanie 200 bilionów konfiguracji wiązałoby się z wykorzystaniem około 750 terabajtów pamięci.

Problemem tablic kontyngencji jest również to, że tworzona tablica najczęściej jest bardzo rzadka, ponieważ ogromna część kombinacji jest nigdy nieobserwowana. Rozwiązaniem mogłoby więc być skorzystanie ze słownika, który rozwiązuje problem rzadkości, poprzez przechowywanie informacji tylko o konfiguracjach, które faktycznie wystąpiły. To z kolei, w zależności od zastosowanej implementacji słownika, wprowadza jednak dodatkowe koszty obliczeniowe oraz pamięciowe. W przypadku tablicy haszującej koszt ten wynika bezpośrednio z operacji haszowania. Z kolei w przypadku słownika opartego na drzewie wyszukiwania jest to koszt, związany z przeszukiwaniem nieciągłych obszarów pamięci.

Kolejnym podejściem do realizacji zapytań zliczających jest wykorzystanie jednego z wielu dostępnych wariantów indeksu ADTree (alternating decision tree) [8, 41, 79, 80], opartego na drzewach decyzyjnych. Metoda ta polega na wstępnym przeanalizowaniu całej

bazy D w celu obliczenia liczby wystąpień wszystkich znalezionych konfiguracji. Wyniki tego procesu są następnie wykorzystywane do obsługi kolejnych zapytań. Wadą tego rozwiązania jest jednak konieczność przeprowadzenia wstępnego, czasochłonnego indeksowania danych, które w wielu przypadkach przekracza akceptowalne ramy czasowe [30]. Wynika to z faktu, że pomimo różnych optymalizacji, złożoność pamięciowa ADTree wzrasta wykładniczo wraz ze wzrostem liczby rozpatrywanych zmiennych w bazie danych. Ponadto, ADtree przechowuje wszystkie konfiguracje bez selekcji, co często skutkuje gromadzeniem danych, które nigdy nie zostaną użyte. Dodatkowo, mimo tego, że omawiane podejście pozwala uniknąć pełnego tworzenia (często ogromnej) tabeli kontyngencji, nadal konieczne jest dynamiczne wygenerowanie jej fragmentu dla konkretnego zapytania.

3.3. SABNAtk: biblioteka open-source do efektywnej realizacji zapytań zliczających

Fakt, że w wielu praktycznych przypadkach realizacja zapytań zliczających zajmuje ponad 90% całkowitego czasu trwania danej aplikacji oraz wspomniane ograniczenia powszechnie stosowanych metod skłaniają do poszukiwania nowych, bardziej wydanych algorytmów. Oprócz złożoności obliczeniowej, istotnym aspektem jest również złożoność pamięciowa, ponieważ wiele algorytmów uczenia maszynowego sama w sobie cechuje się dużym zapotrzebowaniem na pamięć [84]. Zagadnienia wymagające realizacji zapytań zliczających ze wspólnym kontekstem, takie jak omówione w równaniach (3.9) i (3.11), występują zazwyczaj w połączeniu z pewnym algorytmem przeszukiwania, który analizuje różne warianty wyboru X_i i $Pa(X_i)$. W związku z tym, intensywne wykorzystywanie zasobów pamięciowych do realizacji zapytań zliczających, może znacząco wpłynąć na możliwości samego algorytmu.

Punktem wyjścia do przeprowadzonych badań była biblioteka open-source SABNAtk [58], służąca do efektywnej realizacji zapytań zliczających. Została ona zaproponowana [30] w 2018 roku jako wysoce konkurencyjna alternatywa dla powszechnie stosowanych podejść, omówionych w poprzedniej sekcji. SABNAtk napisana jest w języku C++, jednocześnie oferując powiązania (bindings) dla języka Python. Swoją skuteczność zawdzięcza przede wszystkim nowym strategiom realizacji zapytań zliczających: Bitmap i Radix, opartym na efektywnych pamięciowo i obliczeniowo algorytmach przetwarzania danych. Istotny jest również wygodny i intuicyjny interfejs programistyczny, pozwalający na przedstawienie danego problemu obliczeniowego w sposób umożliwiający strumieniowe agregowanie wyników zapytań zliczających. Niewątpliwą zaletą biblioteki jest również to, że jest ona zaimplementowana w sposób wspierający równoległą realizację zapytań zliczających, co zostanie szerzej omówione w sekcji 3.3.5.

3.3.1. Interfejs programistyczny

W typowych aplikacjach uczenia maszynowego wyniki zapytań zliczających o wspólnym kontekście są iteracyjnie agregowane za pomocą operatora, spełniającego warunki łączności i przemienności, np. sumy lub iloczynu. Spełnianie warunków łączności i przemienności stanowi istotną zaletę z punktu widzenia efektywności obliczeniowej, ponieważ umożliwia zastosowanie różnych optymalizacji, np. przetwarzania poza kolejnością. Choć założenia dotyczące łączności i przemienności mogą wydawać się mocno ograniczające, w rzeczywistości obejmują zaskakująco szeroki wachlarz zastosowań w uczeniu maszynowym, takich jak klasyfikatory, drzewa decyzyjne, regresja, ekstrakcja cech a także różne warianty probabilistycznych modeli graficznych. Wynika to z faktu, że tego typu aplikacje w dużej mierze opierają się na estymacji prawdopodobieństw warunkowych i łącznych, które to z kolei bazują na operacjach sumowania i mnożenia. Przestrzeganie praw łączności i przemienności umożliwia strumieniowe podejście do problemu zapytań zliczających. Zamiast najpierw zbierać wszystkie zliczenia, a dopiero potem przeprowadzać agregację, można utworzyć strumień zliczeń odpowiadający wszystkim unikalnym i istotnym konfiguracjom X_i i $Pa(X_i)$ znalezionym w bazie danych D . Pozwala to na bezpośrednie wykonywanie agregacji na strumieniu, co w rezultacie zmniejsza zużycie pamięci, wynikające z przechowywania wyników zliczeń. Kolejną zaletą tego podejścia, wykorzystywaną przez SABNAtk, jest możliwość zaprojektowania interfejsu programistycznego, który całkowicie ukrywa strategię generowania strumienia przed użytkownikiem końcowym. Zadaniem użytkownika jest jedynie określenie sposobu wykorzystania zliczeń. W tym celu, agregacja zliczeń została zamknięta w mechanizmie funkcji zwrotnej, która automatycznie przetwarza nowe wartości pojawiające się w strumieniu. Za każdym razem, gdy pojawiają się nowe, niezerowe zliczenia N_{ij} i N_{ijk} , są one przekazywane jako argumenty do obiektu funkcyjnego zaimplementowanego przez użytkownika końcowego [30].

```

1  class BDeu {
2  public:
3      explicit BDeu(double alpha = 1.0) : alpha_(alpha) { }
4
5      void init(int ri, int qi) {
6          score_ = 0.0;
7          aq_ = alpha_ / qi;
8          lnaq_ = lgamma(aq_);
9          arq_ = alpha_ / (ri * qi);
10         lnarq_ = lgamma(arq_);
11     }
12
13     void finalize(int qi) { }
```

```
14
15 void operator()(int Nij) {
16     score_ += (lnaq_ - lgamma(Nij + aq_));
17 }
18
19 void operator()(int Nijk, int Nij) {
20     score_ += (lgamma(Nijk + arq_) - lnarq_);
21 }
22
23 double score() const { return score_; }
24
25 private:
26     double score_ = 0.0;
27     double alpha_ = 1.0;
28     double aq_ = 1.0;
29     double arq_ = 1.0;
30     double lnaq_ = 1.0;
31     double lnarq_ = 1.0;
32 }; // BDeu
```

Listing 3.1. Przykładowy obiekt funkcyjny C++ implementujący operator BDeu dany równaniem (3.9). Źródło: [30]

Aby lepiej zobrazować zasadę działania, listing 3.1 przedstawia przykład obiektu funkcyjnego w C++ odpowiadającego równaniu (3.9). W uproszczeniu zliczenia N_{ijk} i N_{ij} są przekazywane do kodu użytkownika za pośrednictwem operatorów wywołania funkcji (listing 3.1, linia 15 i 19). Operatory te wykonują wymagane przetwarzanie pośrednie, a następnie agregują wynik w wewnętrznym stanie obiektu. Stan ten można później odczytać (listing 3.1, linia 23), aby uzyskać ostateczny wynik agregacji. Jak wcześniej zauważono, z perspektywy użytkownika końcowego operator wywołania funkcji pełni rolę interfejsu, który jest automatycznie wywoływany przez strategię (Bitmap lub Radix) odpowiedzialną za generowanie i przetwarzanie wszystkich unikalnych konfiguracji zmiennych. Dzięki temu interfejs stanowi wygodną formę enkapsulacji logiki. Użytkownik definiujący obiekt funkcyjny (np. implementujący funkcję oceny w uczeniu sieci Bayesowskiej) może skoncentrować się wyłącznie na opisie obliczeń, czyli wysokopoziomowej logice i jej poprawności, nie przejmując się złożonością niskopoziomowych operacji, takich jak enumeracja zliczeń. Dodatkowo obiekty funkcyjne, choć działają jak funkcje, posiadają wewnętrzny stan, co czyni je wygodnym mechanizmem do wyrażenia nawet najbardziej wymagających obliczeń.

3.3.2. Strategia Bitmap

Rozpatrując dane zapytanie zliczające $\text{COUNT}(X_i \mid Pa(X_i))$, celem jest wyznaczenie niezerowych wartości N_{ijk} oraz N_{ij} dla wszystkich konfiguracji j oraz k z bazy danych D i przekazanie ich do obiektu funkcyjnego w celu wykonania agregacji.

Koncepcja strategii Bitmap polega na przedstawieniu obserwacji każdej zmiennej X_i z bazy danych D za pomocą r_i bitmap, gdzie r_i oznacza liczbę różnych stanów, jakie dana zmienna przyjmuje. Każda bitmapa składa się z m bitów, co odpowiada liczbie obserwacji w bazie danych. Wartości bitów w bitmapie sygnalizują, czy zmienna X_i przyjmuje dany stan w konkretnej obserwacji (wartość 1 oznacza wystąpienie danego stanu, a 0 jego brak).

Rysunek 3.4 przedstawia bitmapy dla trzech zmiennych X_1, X_2, X_3 z przykładowej bazy danych D , prezentowanej wcześniej na rysunku 3.1. W przypadku zmiennej X_1 utworzono trzy bitmapy, odpowiadające trzem różnym stanom (0, 1, 2), które zmienna ta przyjmuje w bazie D . Przykładowo, bitmapa dla stanu 0 zmiennej X_1 zawiera wartość 1 dla obserwacji D_2 oraz D_7 , ponieważ w tych właśnie obserwacjach zmienna X_1 przyjmowała stan 0.

X_1	0	1	2
D_1	0	0	1
D_2	1	0	0
D_3	0	1	0
D_4	0	0	1
D_5	0	1	0
D_6	0	0	1
D_7	1	0	0

X_2	0	1	2
D_1	1	0	0
D_2	0	1	0
D_3	1	0	0
D_4	1	0	0
D_5	0	1	0
D_6	1	0	0
D_7	0	0	1

X_3	0	1
D_1	0	1
D_2	1	0
D_3	0	1
D_4	0	1
D_5	1	0
D_6	0	1
D_7	0	1

Rysunek 3.4. Bitmapy zmiennych X_1, X_2 oraz X_3 z bazy danych D przedstawionej na rysunku 3.1

Po utworzeniu bitmap, realizacja danego zapytania zliczającego polega na wykonaniu serii logicznych operacji AND dla odpowiednich bitmap oraz zliczaniu niezerowych bitów w wynikowych bitmapach. W tym celu, pierwszym krokiem jest utworzenie drzewa składającego się L warstw, gdzie L to liczba zmiennych wchodzących w skład zapytania. Następnie, algorytm analizuje węzły tego drzewa metodą przeszukiwania w głąb (ang. Depth First Search).

Proces ten przedstawiony został w Algorytmie 1 wraz z przykładem z rysunku 3.5, wizualizującym realizację zapytania $\text{COUNT}(X_1 \mid \{X_2, X_3\})$ dla omawianej przykładowej bazy danych D . Do argumentów algorytmu, poza zmiennymi wchodzącymi w skład zapytania, należą obiekt funkcyjny F , służący do zgłaszania wyznaczonych wartości N_{ijk} i $N_{i,j}$ oraz bitmapa b o długości m - bitów, początkowo wypełniona samymi jedynkami. Jest ona związana z rekurencyjną naturą algorytmu, a jej znaczenie zostanie omówione w dalszej części tej sekcji.

Pierwsze warstwy drzewa tworzone są przez zmienne wchodzące w skład zbioru $Pa(X_i)$, natomiast ostatnia warstwa odpowiada stanom zmiennej X_i . Algorytm działa w sposób rekurencyjny, rozpoczynając od węzła będącego korzeniem. Jeżeli dany węzeł nie jest liściem ($|Pa| \neq 0$), algorytm przechodzi w głąb drzewa (algorytm 1, linia 9-13) i dla rozpatrywanego węzła wykonywane są logiczne operacje AND (w algorytmie oznaczone jako $\cap$, z racji stosowania notacji dla zbiorów) pomiędzy wspomnianą bitmapą b a wszystkimi bitmapami danego węzła. Bitmapa b umożliwia zapamiętywanie wyniku łącznej operacji AND wykonanej dla węzłów z wszystkich dotychczasowo odwiedzonych poziomów drzewa. Przechodzenie w głąb jest kontynuowane poprzez rekurencyjne wywołanie algorytmu (algorytm 1, linia 13) tylko wtedy, gdy wynikowa bitmapa nie składa się z samych bitów o wartości 0, co oznacza odkrycie niezerowych zliczeń dla danej konfiguracji zmiennych, znajdujących się w kolejnych poziomach drzewa. Po dotarciu do dolnej warstwy drzewa, czyli do węzłów będących liśćmi (algorytm 1, linie 1-7), następuje obliczenie oraz zgłoszenie wartości N_{ijk} i N_{ij} poprzez wywołanie operatora $()$ przekazanego obiektu funkcyjnego F .

Algorytm 1 $\text{QUERY}(X_i, Pa, F, b)$

```

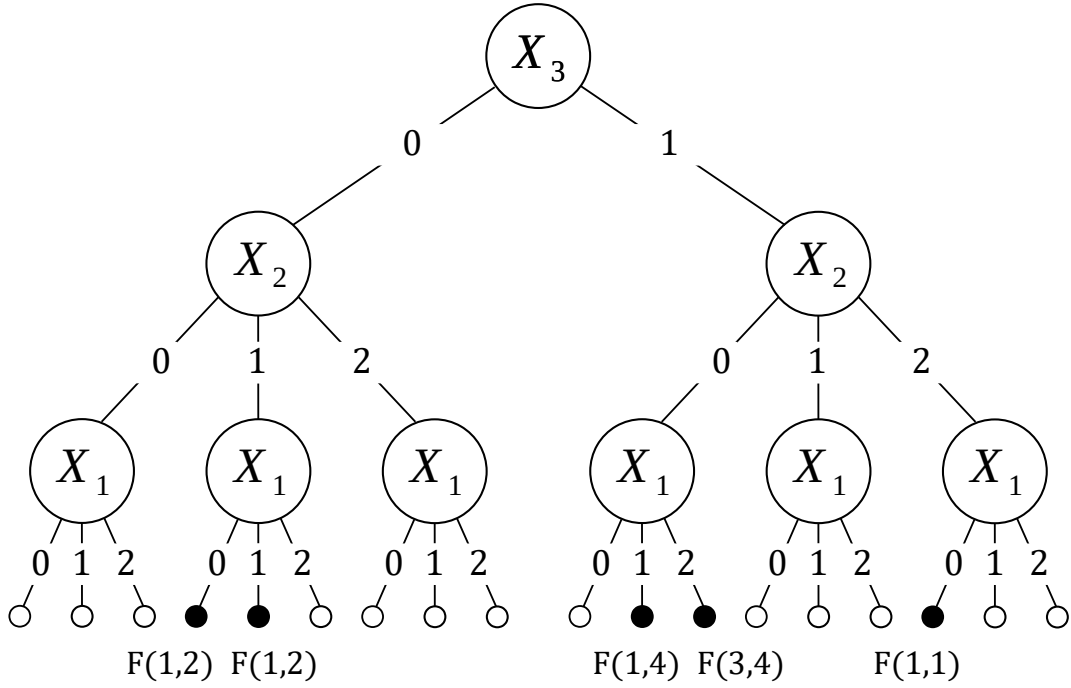
1: if  $|Pa| = 0$  then
2:    $N_{ij} \leftarrow |b|$ 
3:   for  $v \in [0, \dots, r_i - 1]$  do
4:      $b_v \leftarrow \{p \mid D_i[p] = v\}$ 
5:      $N_{ijk} \leftarrow |b \cap b_v|$ 
6:     if  $N_{ijk} > 0$  then
7:        $F(N_{ijk}, N_{ij}) \triangleleft \text{emit new configuration}$ 
8:   else
9:      $X_h \leftarrow \text{HEAD}(Pa)$ 
10:    for  $v \in [0, \dots, r_h - 1]$  do
11:       $b_v \leftarrow \{p \mid D_h[p] = v\}$ 
12:      if  $|b \cap b_v| > 0$  then
13:         $\text{QUERY}(X_i, \text{TAIL}(Pa), F, b \cap b_v)$ 

```

Rysunek 3.5 przedstawia wizualizację drzewa, związanego z realizacją zapytania zliczającego $\text{COUNT}(X_1 \mid \{X_2, X_3\})$. Przedstawione drzewo składa się z trzech poziomów. Dwa pierwsze poziomy utworzone są przez zmienne wchodzące w skład zbioru $Pa(X_1) = \{X_2, X_3\}$. Kolejność zmiennych w drzewie wynika z optymalizacji wdrożonej

w SABNAtk. Polega ona na posortowaniu zmiennych ze zbioru Pa względem ich entropii, co pozwala na ogół zmniejszyć liczbę węzłów, odwiedzanych podczas przetwarzania zapytania. W pewnym uproszczeniu może być to rozumiane jako umieszczenie w górnych warstwach drzewa węzłów, które mają najmniejszą liczbę stanów. Warto podkreślić, że kolejność przetwarzania zmiennych ze zbioru Pa nie ma wpływu na wyniki $F(N_{ijk}, N_{ij})$, które generowane są sukcesywnie podczas odwiedzania węzłów związanych ze zmienną X_i , znajdujących się w ostatniej warstwie drzewa.

Pod rysunkiem 3.5 zamieszczono fragment przebiegu algorytmu dotyczący generowania wartości $F(1, 4)$ oraz $F(3, 4)$.



Rysunek 3.5. Przykład wykonania zapytania zliczającego $\text{COUNT}(X_1 \mid \{X_2, X_3\})$ z użyciem strategii Bitmap

Analiza przebiegu algorytmu w części drzewa $X_3 = 1 \rightarrow X_2 = 0 \rightarrow X_1 = \{0, 1, 2\}$

$\vdots$

$b_1 \leftarrow 1011011$ (bitmapa stanu 1 dla zmiennej X_3)

$b \cap b_1 = 1011011$ ($b = 1111111$)

$|b \cap b_1| = 5 > 0 \rightarrow \text{QUERY}(X_1, X_2, F, 1011011)$

$b_0 \leftarrow 1011010$

$b \cap b_0 = 1011010$

$|b \cap b_0| = 4 > 0 \rightarrow \text{QUERY}(X_1, \emptyset, F, 1011010)$

$N_{ij} \leftarrow |1011010| = 4$

$b_0 \leftarrow 0100100$

$N_{ijk} \leftarrow |b \cap b_0| = 0$

$b_1 \leftarrow 0010100$

$N_{ijk} \leftarrow |b \cap b_1| = 1$

F(1, 4)

$b_2 \leftarrow 1001010$

$N_{ijk} \leftarrow |b \cap b_2| = 3$

F(3, 4)

$\vdots$

3.3.3. Strategia Radix

Oprócz omówionej w poprzedniej części strategii Bitmap, SABNAtk oferuje również alternatywną strategię, nazwaną Radix. Jej zastosowanie staje się szczególnie korzystne w sytuacjach, gdy przetwarzane są zbiory danych o bardzo dużej liczbie obserwacji (np. setki tysięcy) lub zapytania obejmujące dużą liczbę zmiennych (np. kilkadziesiąt). W takich przypadkach strategia Bitmap może tracić na wydajności ze względu na dynamiczne rozrastanie się drzewa przeszukiwań DFS. W efekcie przewaga wynikająca z szybkich operacji bitowych zostaje zniwelowana przez niekorzystne asymptotyczne zachowanie algorytmu.

Strategia Radix opiera się na klasycznym algorytmie sortowania pozycyjnego (ang. radix sort). Polega ona na rekurencyjnym podziale obserwacji w bazie danych D w taki sposób, aby każda pojedyncza partycja na danym poziomie zawierała wszystkie obserwacje, odpowiadające określonej konfiguracji zmiennych zapytania zliczającego. Szczegółowy sposób postępowania został przedstawiony w ramach algorytmów 2 i 3.

Algorytm 2 QUERY(X_i, Pa, F)

```

1:  $B \leftarrow [[1, \dots, m]]$ 
2: if  $|Pa| \neq 0$  then
3:    $B \leftarrow \text{BUCKETS}(\text{HEAD}(Pa), \text{TAIL}(Pa), \text{HEAD}(B))$ 
4: for  $b \in B$  do
5:    $N_{ij} \leftarrow |b|$ 
6:   if  $N_{ij} > 0$  then
7:      $B' \leftarrow \text{BUCKETS}(X_i, [], b)$ 
8:     for  $b' \in B'$  do
9:        $N_{ijk} \leftarrow |b'|$ 
10:    if  $N_{ijk} > 0$  then
11:       $F(N_{ijk}, N_{ij}) \triangleleft \text{emit new configuration}$ 
```

Algorytm 3 BUCKETS(X_p, Pa, b)

```

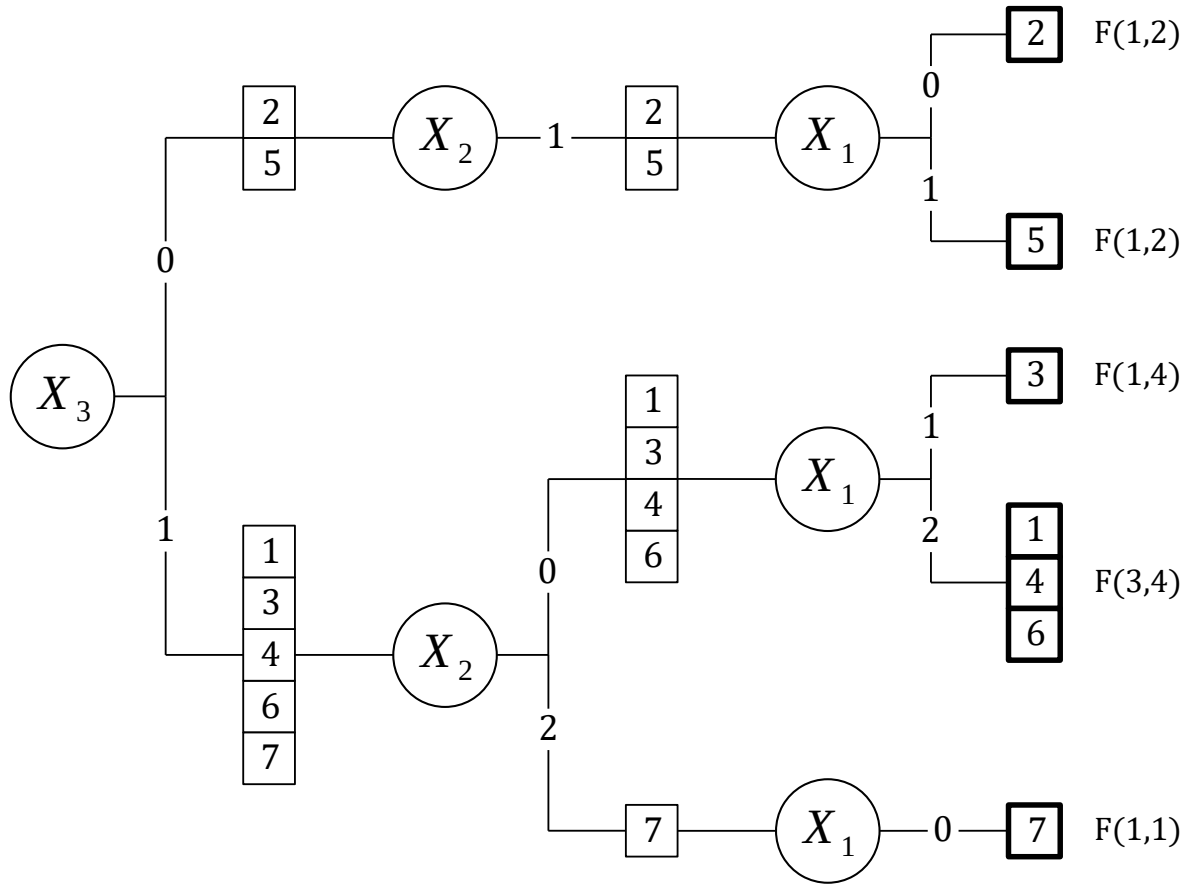
1:  $B' \leftarrow []$ 
2: for  $q \in [1, \dots, |b|]$  do
3:    $x_p \leftarrow D_{b[q]}[p]$ 
4:    $B'[x_p].\text{APPEND}(b[q])$ 
5: if  $Pa = []$  then
6:   return  $B'$ 
7:  $B'' \leftarrow []$ 
8: for  $b' \in B'$  do
9:    $B''.\text{APPEND}(\text{BUCKETS}(\text{HEAD}(Pa), \text{TAIL}(Pa), b'))$ 
10: return  $B''$ 
```

Algorytm realizuje podejście zgodne z metodą Most Significant Digit (MSD) radix, gdzie najbardziej znaczącymi „cyframi” są stany poszczególnych zmiennych ze zbioru

Pa , natomiast najmniej znaczącą cyfrą są stany zmiennej X_i . Początkowo wszystkie obserwacje traktowane są jako jedna partycja, a ich indeksy są przechowywane w liście B (algorytm 2, linia 1). Następnie wykonywana jest procedura BUCKETS, która partycjonuje obserwacje w zależności od stanu zmiennych w Pa (linia 3). Liczba nowo utworzonych partycji na każdym poziomie jest proporcjonalna do liczby przyjmowanych stanów przez analizowaną zmienną.

Po zakończeniu tworzenia partycji względem zmiennych w Pa , algorytm przeprowadza końcowe partycjonowanie względem zmiennej X_i (algorytm 2, linia 7). Następnie dla każdej partycji o niezerowej liczebności zgłaszane są wyniki przez wywołanie obiektu funkcyjnego F z wartościami N_{ij} oraz N_{ijk} .

Wizualizacja realizacji zapytania $COUNT(X_1 \mid \{X_2, X_3\})$ z wykorzystaniem strategii Radix dla omawianej przykładowej bazy danych D , została przedstawiona na rysunku 3.6. W przeciwieństwie do strategii Bitmap, kolejność przetwarzania zmiennych ze zbioru Pa nie jest istotna. Powodem jest to, że identyfikacja pustych partycji nie wiąże się z dodatkowymi kosztami obliczeniowymi.

Rysunek 3.6. Przykład wykonania zapytania zliczającego $\text{COUNT}(X_1 \mid \{X_2, X_3\})$ strategią Radix

3.3.4. Realizacja przykładowego zapytania zliczającego

W tej sekcji przedstawione zostanie wykorzystanie biblioteki SABNAtk do realizacji przykładowego zapytania zliczającego. W tym celu w dalszym ciągu rozważana jest baza danych D , wprowadzona w sekcji 3.1 oraz zapytanie zliczające $\text{COUNT}(X_1 \mid \{X_2, X_3\})$. Listing 3.2 przedstawia kod programu napisanego w C++, służący do realizacji wyżej wymienionego zapytania. Pierwszym krokiem jest wczytanie bazy danych. W przedstawionym kodzie, baza danych wczytywana jest w linii 5. jako wektor liczb typu `uint8_t`. Warto zauważyć, że jest to jedynie prosty przykład i w rzeczywistych zastosowaniach baza danych składa się ze znacznie większej liczby zmiennych oraz obserwacji, dlatego zwykle wczytywana jest z pliku za pomocą gotowych funkcji dostarczonych przez bibliotekę.

```

1  struct example {
2      void init(int ri, int qi) {}
3      void finalize(int qi) {}
4
5      void operator()(int Nij) {
6          std::cout << "Nij=" << Nij << std::endl;
7      }
8
9      void operator()(int Nijk, int Nij) {
10         std::cout << "Nijk=" << Nijk << std::endl;
11     }
12 };
13
14 int main(int argc, char* argv[]) {
15     std::vector<uint8_t> D{ 2, 0, 1, 2, 1, 2, 0, \
16                           0, 1, 0, 0, 1, 0, 2, \
17                           1, 0, 1, 1, 0, 1, 1};
18
19     auto counter = sabnatk::create_counter<1, sabnatk::BV>(3, 7, std::begin(D));
20     using set_type = decltype(counter)::set_type;
21
22     auto xi = set_empty<set_type>();
23     xi = set_add(xi, 0); // X1
24
25     auto pa = set_empty<set_type>();
26     pa = set_add(pa, 1); // X2
27     pa = set_add(pa, 2); // X3
28
29     std::vector<example> F(1);
30
31     // COUNT(X1 | X2, X3)

```

```

32     counter.apply(xi, pa, F);
33
34     return 0;
35 }

```

Listing 3.2. Wykorzystanie biblioteki SABNAtk do realizacji zapytania zliczającego **COUNT**($X_1 \mid \{X_2, X_3\}$)

Koncepcja wykorzystana w SABNAtk polega na utworzeniu tzw. silnika do realizacji zapytań zliczających. Silnikiem nazywany jest obiekt utworzony za pomocą funkcji `create_counter`. Podczas jego tworzenia należy wskazać strategię, jaka będzie wykorzystywana do realizacji zapytań zliczających, liczbę zmiennych oraz obserwacji a także iterator, wskazujący na pierwszy element wczytywanej bazy danych. Zadaniem funkcji tworzącej silnik jest przede wszystkim przeanalizowanie danych z bazy D oraz utworzenie na jej podstawie struktur, które będą niezbędne dla danej strategii (np. bitmap stanów wszystkich zmiennych w przypadku wyboru strategii Bitmap). W przedstawionym przykładzie podczas tworzenia silnika (listing 3.2, linia 19) wskazano strategię Bitmap (BV) oraz iterator do bazy danych D , składającej się z trzech zmiennych, z których każda obserwowana była siedmiokrotnie.

Następnym krokiem jest zdefiniowanie zmiennych wchodzących w skład zapytania. W tym celu wykorzystane zostały funkcje `set_empty` oraz `set_add`, które są dostarczone przez bibliotekę. Pozwalają one na zdefiniowanie pustego zbioru a następnie dodanie do niego wybranych zmiennych. Przedstawiony przykład dotyczy omawianego zapytania $\text{COUNT}(X_1 \mid \{X_2, X_3\})$, dlatego `xi` to jednoelementowy zbiór, którego elementem jest zmienna X_1 , natomiast `pa` to zbiór, składający się ze zmiennych X_2 oraz X_3 .

Po zdefiniowaniu zmiennych wchodzących w skład zapytania, możliwa jest już jego właściwa realizacja. W tym celu należy skorzystać z metody `apply`, dostępnej dla obiektu, będącego silnikiem realizacji zapytań. Metoda `apply` jest wielokrotnie przeciążona, zapewniając różne warianty argumentów, pozwalające na realizację licznych przypadków użycia. W przedstawionym przykładzie jej argumentami są zbiory, dotyczące zmiennych wchodzących w skład zapytania oraz obiekt funkcyjny, do którego przekazywane będą liczby N_{ij} oraz N_{ijk} , będące wynikami realizacji danego zapytania. Koncepcja ta została już przedstawiona szerzej w sekcji 3.3.1 wraz z przykładem z listingu 3.1. W analizowanym obecnie przykładzie, zaprezentowanym na listingu 3.2, rolę obiektu funkcyjnego pełni obiekt `F`, będący instancją przykładowej struktury o nazwie `example`, zdefiniowanej na początku prezentowanego kodu. Dla prostoty przykładu, jej implementacja została ograniczona do minimum, tzn. do wypisania na ekran znalezionych wartości N_{ij} oraz N_{ijk} . W tym miejscu należy również zauważyć, że F to tak naprawdę jednoelementowy wektor struktur `example`. Wynika to z tego, że w ogólnym przypadku zbiór

$\mathbf{x}_i$ może zawierać więcej elementów a dla każdej zmiennej wchodzącej w skład $\mathbf{x}_i$ niezbędny jest osobny obiekt funkcyjny.

Wyniki realizacji zapytania $\text{COUNT}(X_1 \mid \{X_2, X_3\})$ (listing 3.2, linia 32) zostały przedstawione na rysunku 3.7. Są one oczywiście tożsame z wynikami tego zapytania przedstawionymi wcześniej w tabeli kontyngencji zamieszczonej na rysunku 3.2.

```

Ni j=4
Ni jk=3
Ni jk=1
Ni j=1
Ni jk=1
Ni j=2
Ni jk=1
Ni jk=1

```

Rysunek 3.7. Wyniki realizacji zapytania zliczającego $\text{COUNT}(X_1 \mid \{X_2, X_3\})$ przy użyciu SABNAtk (listing 3.2)

3.3.5. Koncepcja zrównoleglenia realizacji zapytań zliczających

W poprzedniej sekcji przedstawiono kod źródłowy (listing 3.2) ilustrujący sposób użycia biblioteki SABNAtk do realizacji przykładowego zapytania zliczającego. Celem tego przykładu było przybliżenie podstawowych koncepcji oraz interfejsu oferowanego przez bibliotekę. W rzeczywistych zastosowaniach, takich jak np. proces uczenia sieci Bayesowskich, konieczne jest jednak wykonanie znacznie większej liczby zapytań, często rzędu setek tysięcy a nawet milionów. Realizacja tak dużej liczby zapytań wiąże się często z wysokimi wymaganiami obliczeniowymi, dlatego kluczowe staje się poszukiwanie sposobów na zwiększenie efektywności przetwarzania zapytań zliczających. Jednym z naturalnych kierunków optymalizacji jest wykorzystanie potencjału nowoczesnych wielordzeniowych architektur komputerowych, oferujących znaczące możliwości równoległego przetwarzania danych.

Istotnym powodem, dla którego wybrano bibliotekę SABNAtk do badań jest to, że została ona zaimplementowana w sposób wspierający równoległą realizację zapytań zliczających. Fakt ten, w połączeniu z oferowanymi przez nią efektywnymi pamięciowo i obliczeniowo strategiami realizacji zapytań zliczających, sprawia, że stanowi ona interesujący przedmiot badań nad adaptacją realizacji zapytań zliczających do wysokowydajnych systemów komputerowych.

Aby wyjaśnić, dlaczego SABNAtk wspiera równoległą realizację zapytań zliczających, konieczne jest bardziej szczegółowe przedstawienie stosowanego w niej podejścia. Omawiana biblioteka oferuje możliwość wyboru strategii do realizacji zapytań zliczających

(np. Bitmapy lub Radix), zaimplementowanych z użyciem języka C++ jako osobne klasy. Wybór ten odbywa się poprzez wywołanie funkcji `create_counter` z odpowiednimi argumentami, co zostało omówione w poprzedniej sekcji (Listing 3.2, linia 19). Zwracany przez tę funkcję obiekt (nazywany silnikiem) przechowuje dane wejściowe w sposób zgodny z daną strategią oraz udostępnia metodę `apply`, służącą do wykonywania zapytań zliczających. Kluczowym aspektem architektury SABNAtk, istotnym z punktu widzenia równoległości, jest bezstanowy charakter strategii, co oznacza to, że metoda `apply` nie modyfikuje wewnętrznych danych silnika, a jedynie odczytuje informacje niezbędne do realizacji zapytania. Dzięki temu wiele wątków może jednocześnie korzystać z tego samego obiektu strategii bez ryzyka naruszenia integralności danych. Taka implementacja umożliwia naturalne zastosowanie modelu przetwarzania task-parallel, w którym poszczególne wywołania metody `apply` stanowią niezależne zadania, gotowe do równoległego wykonania na wielu rdzeniach procesora.

Co szczególnie istotne, architektura SABNAtk zapewnia dużą elastyczność. Bezstanowość strategii oraz fakt, że metoda `apply` nie wprowadza żadnych modyfikacji danych, pozwalają użytkownikowi samodzielnie decydować o sposobie organizacji przetwarzania równoległego. Innymi słowy, biblioteka nie narzuca konkretnego modelu wykonania, lecz umożliwia jego swobodną implementację. Możliwe jest więc dostosowanie aplikacji do różnych scenariuszy użycia — od prostych pojedynczych zapytań (Listing 3.2), przez równoległą realizację z góry znanego strumienia zapytań, aż po rozbudowane przypadki wymagające dynamicznego generowania zapytań w czasie wykonywania programu.

Przykładem takiego scenariusza użycia, wykorzystanym podczas testowania metod opracowanych przez autora niniejszej rozprawy, jest równoległa realizacja strumienia zapytań z zastosowaniem biblioteki OpenMP. Kod źródłowy dotyczący tego scenariusza został przedstawiony na Listingu 3.3. Polega on na dynamicznym rozdzieleniu zapytań zliczających poprzez dostępne wątki, co pozwala na efektywne zrównoważenie obciążenia i pełne wykorzystanie zasobów obliczeniowych systemu. W przedstawionym fragmencie kodu zmienna Q to wektor zawierający qs zapytań zliczających $\text{COUNT}(X_i \mid Pa(X_i))$. Każde zapytanie reprezentowane jest jako para, w której pierwszy element stanowi zmienna X_i , natomiast drugi — zbiór zmiennych $Pa(X_i)$. Zmienna eng to silnik, czyli obiekt udostępniający metodę `apply` do realizacji zapytań zliczających. Sposób jego tworzenia został szczegółowo omówiony w sekcji 3.3.4. Obiekt F pełni natomiast rolę funkcji zwrotnej, wykorzystywanej do zgłaszania wyników — jego opis znajduje się w sekcji 3.3.1.

```
1 #pragma omp parallel for shared(Q, eng), firstprivate(F) schedule(dynamic)
2 for (int i = 0; i < qs; ++i)
3     eng.apply(Q[i].first, Q[i].second, F);
```

Listing 3.3. Zrównoleglenie realizacji strumienia zapytań zliczających z wykorzystaniem biblioteki OpenMP

3.4. Metodologia ewaluacji

Niniejsza sekcja przedstawia metodologię ewaluacji metod oraz algorytmów opracowanych w ramach badań, związanych z tematyką zapytań zliczających. Opisane zostały wykorzystywane platformy obliczeniowe, sposób przygotowania zbiorów danych oraz proces opracowania benchmarku zapytań wykorzystywanego wielokrotnie podczas eksperymentów. Elementy te stanowią część wspólną przeprowadzonych badań. Zostały zebrane w niniejszej sekcji, aby można było odwoływać się do nich podczas omawiania wyników eksperymentów dotyczących rozwiązań opracowanych przez autora, przedstawionych w rozdziałach 4 – 7. Zakres tych rozdziałów obejmuje:

- rozdział 4 – metody poprawy wydajności strategii Bitmap,
- rozdział 5 – metodę automatycznego wyboru strategii,
- rozdział 6 – równoległą realizację metody automatycznego wyboru strategii,
- rozdział 7 – metodę równoległej realizacji zapytań zliczających uwzględniającą architekturę systemów ccNUMA.

Wszystkie opisane rozwiązania zostały zaprezentowane podczas recenzowanych, wysoko punktowanych konferencji naukowych (140 pkt wg wykazu MNiSW) oraz opublikowane w materiałach pokonferencyjnych. Szczegółowe informacje na ten temat zawarto w podsumowaniu pracy, przedstawionym w rozdziale 8.

3.4.1. Platformy obliczeniowe

Opracowane metody oraz algorytmy były testowane z wykorzystaniem różnych platform obliczeniowych, przy czym dostęp do poszczególnych systemów zmieniał się w trakcie prowadzonych eksperymentów. W celu zapewnienia rzetelności i porównywalności wyników, w niniejszej pracy uwzględniono wyłącznie te platformy, dla których rezultaty zostały opublikowane w recenzowanych publikacjach, przedstawionych w rozdziale 8.

Tabela 3.1 przedstawia szczegółową charakterystykę wykorzystanych platform, obejmującą liczbę rdzeni obliczeniowych, częstotliwość taktowania, dostępną pamięć operacyjną oraz liczbę domen NUMA. W badaniach wykorzystano trzy architektury serwerowe: Intel Cascade Lake, AMD Rome oraz AMD Milan. Platforma Intel Cascade Lake była dostępna bez ograniczeń czasowych, natomiast dostęp do platform AMD Rome oraz AMD Milan był możliwy dzięki grantowi obliczeniowemu i realizowany z wykorzystaniem systemu kolejkowego z limitem czasu jednorazowej sesji wynoszącym cztery godziny. W celu zapewnienia porównywalnych wyników, podczas wszystkich eksperymentów wyłączono wielowątkowość współbieżną (Simultaneous Multithreading, SMT), wykorzystując wyłącznie fizyczne rdzenie obliczeniowe.

Platforma obliczeniowa	Procesor	Rdzenie	Pamięć	Domeny NUMA
Intel Cascade Lake	2 × Intel Xeon Gold 6240	2 × 18	192 GiB DDR4-2933	2 × 1
AMD Rome	2 × AMD EPYC 7742	2 × 64	512 GiB DDR4-3200	2 × 8
AMD Milan	2 × AMD EPYC 7763	2 × 64	512 GiB DDR4-3200	2 × 8

Tabela 3.1: Specyfikacja platform obliczeniowych wykorzystanych podczas ewaluacji opracowanych metod i algorytmów w ramach tematyki zapytań zliczających

3.4.2. Przygotowanie zbiorów danych

Do testów przygotowano zestaw zbiorów danych, powszechnie wykorzystywanych w uczeniu maszynowym [9]. Zbiory te zostały wygenerowane przez autora niniejszej pracy na podstawie pakietu `bnlearn` — narzędzia języka R, przeznaczonego do uczenia i wnioskowania w sieciach bayesowskich. Szczegóły dotyczące wygenerowanych zbiorów danych przedstawiono w tabeli 3.2. Poza liczbą zmiennych (n), dla każdego zbioru danych podano także informacje o zakresie oraz średniej liczbie stanów przyjmowanych przez jego zmienne.

Zbiór danych	n	Zakres r_i	Średnia r_i
Sachs	11	3-3	3.00
Child	20	2-6	3.00
Insurance	27	2-5	3.30
Water	33	1-4	2.47
Mildew	35	3-100	16.77
Alarm	37	2-4	2.84
Barley	48	2-67	8.77
Hailfinder	56	2-11	3.98
Win95pts	76	1-2	2.00
Pathfinder	109	2-63	4.09

Tabela 3.2: Zbiory danych wykorzystywane podczas testowania opracowanych rozwiązań

Każdy zbiór danych zapisano w osobnym pliku CSV. Wiersze tych plików odpowiadają poszczególnym zmiennym losowym, natomiast kolumny kolejnym obserwacjom. Wszystkie zmienne mają charakter kategoriyczny i każda z nich może przyjmować różną liczbę stanów, numerowanych od 0. Przykładowo, plik CSV odpowiadający zbiorowi danych Child zawiera 20 wierszy, z których każdy reprezentuje ciąg obserwacji jednej zmiennej. Rysunek 3.8 przedstawia fragment pliku CSV odpowiadającego temu zbiorowi danych.

```

1  0 0 0 1 1 0 0 0 1 0 1 0 0 0 0 1 1 0 0 0 0 1 0 0 0
2  0 1 1 2 0 2 1 2 0 1 2 1 0 0 1 2 1 0 1 0 1 1 1 1 1
3  2 2 2 2 0 2 0 2 0 2 2 2 2 2 2 0 2 2 2 0 2 0 2 2 2
4  3 0 0 3 3 3 1 3 3 0 0 0 1 3 1 3 0 0 0 3 0 0 0 0 0
5  1 1 0 1 1 0 2 0 1 0 0 0 1 0 2 1 0 0 0 1 0 0 0 0 0
6  0 0 1 0 0 1 0 0 0 1 1 0 0 0 0 0 1 1 1 0 0 1 1 0 1
7  0 1 2 0 0 0 2 0 0 0 0 2 0 0 0 0 0 0 0 0 1 0 2 0 0
8  0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 1 0 0 0 0 1
9  5 0 2 5 5 2 3 5 5 2 2 2 5 5 3 5 2 2 2 5 0 2 2 0 2
10 4 3 2 2 3 2 0 0 1 3 2 0 2 4 3 4 3 4 2 3 2 2 4 3 3

```

Rysunek 3.8. Fragment pliku przedstawiający 25 obserwacji dziesięciu pierwszych zmiennych losowych zbioru danych Child

Ponadto, aby umożliwić szczegółową weryfikację opracowanych metod, dla każdego zbioru danych przygotowano kilka wersji różniących się liczbą obserwacji, czyli kolumn. Powodem jest to, że liczba obserwacji zmiennych wchodzących w skład zapytania zliczającego jest jednym z kluczowych czynników wpływających na jego czas odpowiedzi. Przykładowo, podczas gdy dane zapytanie zliczające realizowane dla bazy danych składającej się z tysiąca obserwacji może zostać przetworzone w kilka mikrosekund, te same zapytanie dla bazy zawierającej milion obserwacji może już zająć nawet kilka sekund. Eksperymenty potwierdzające tę zależność, wraz ze szczegółową analizą złożoności strategii przetwarzania zapytań zliczających, zostaną przedstawione w sekcjach 5.1 oraz 5.3.

W związku z tym, w celu kompleksowego przetestowania skuteczności metod, dla każdego zbioru danych z tabeli 3.2 przygotowano pięć wersji, zawierających kolejno tysiąc (1K), dziesięć tysięcy (10K), sto tysięcy (100K), milion (1M) oraz dziesięć milionów (10M) obserwacji.

3.4.3. Opracowanie benchmarku losowych zapytań

Aby umożliwić weryfikację skuteczności opracowywanych metod, opracowano benchmark polegający na realizacji strumienia zapytań o losowym rozmiarze. W tym celu dla każdego zbioru danych z tabeli 3.2 wygenerowano dziesięć milionów (10M) zapytań postaci $\text{COUNT}(X_i \mid Pa(X_i))$. Każde pojedyncze zapytanie zostało wygenerowane za pomocą tej samej procedury. Najpierw losowano rozmiar zbioru $Pa(X_i)$ jako liczbę z zakresu $[1, \dots, n - 1]$, gdzie każda wartość miała jednakowe prawdopodobieństwo wystąpienia. Następnie wybierano $|Pa(X_i)| + 1$ zmiennych losując bez zwracania z danego zbioru danych a wylosowane zmienne zostawały przypisane do $Pa(X_i)$ oraz X_i .

Wygenerowane zapytania dla każdego zbioru danych zapisano w osobnych plikach z rozszerzeniem .dat. Każde zapytanie strumienia stanowi osobny wiersz pliku. Format zapytań obejmuje trzy informacje rozdzielone pionową kreską. W pierwszej kolejności podawana jest liczba zmiennych wchodzących w skład zbioru rodziców (pa_size). Następnie wymieniane są zmienne tego zbioru, a dokładniej numery wierszy pliku CSV przechodzącego obserwacje tych zmiennych. Ostatnim elementem jest numer wiersza odpowiadający zmiennej X_i . Na rysunku 3.9 przedstawiono fragment pliku z zapytaniami wygenerowanymi dla zbioru danych Child.

```

1  pa_size = 12 | 0 2 4 5 7 10 13 15 16 17 18 19 |8 |
2  pa_size = 1 | 8 |1 |
3  pa_size = 17 | 1 2 3 4 5 6 7 9 10 11 13 14 15 16 17 18 19 |8 |
4  pa_size = 2 | 1 3 |0 |
5  pa_size = 6 | 1 3 4 9 12 19 |17 |
6  pa_size = 11 | 0 1 4 5 9 11 13 15 16 17 19 |12 |
7  pa_size = 8 | 2 3 4 10 14 15 16 17 |11 |
8  pa_size = 10 | 0 1 2 3 4 5 7 11 15 16 |8 |
9  pa_size = 10 | 0 1 2 4 8 9 13 16 17 19 |3 |
10 pa_size = 0 | |13 |

```

Rysunek 3.9. Fragment pliku przedstawiający dziesięć pierwszych losowych zapytań dla zbioru danych Child

Każdy wygenerowany plik zawiera dziesięć milionów zapytań. Podczas testowania opracowanych metod nie zawsze jednak wykorzystywana jest całość dostępnych zapytań. Ich liczba jest dobierana w zależności od potrzeby, co pozwala na dostosowanie czasu trwania obliczeń. Kwestia ta zostanie omówiona w kolejnych częściach pracy.

Listing 3.4 przedstawia przykład wykorzystania opracowanego benchmarku losowych zapytań w praktyce. W pierwszej kolejności następuje wczytanie z plików zbioru danych D (Listing 5.1, linie 18-23) oraz strumienia zapytań Q (Listing 5.1, linie 25-29). Formaty tych danych muszą spełniać warunki określone w poprzedniej oraz niniejszej sekcji. Następnie tworzony jest obiekt funkcyjny F , służący do zgłaszania wyników zrealizowanych zapytań. Koncepcja tego podejścia została szerzej omówiona w sekcji 3.3.1. Po utworzeniu silnika związanego z wybraną strategią (w przykładzie wybrano metodę Bitmap), następuje realizacja strumienia zapytań poprzez wywołanie metody `apply` dla każdego zapytania ze strumienia Q (listing 3.4, linia 35).

```

1  #include <iostream>
2  #include <vector>
3  #include <sabnatk.hpp>
4  #include <estimator.hpp>
5  #include "csv.hpp"
6  #include "read_queries.hpp"
7
8  int main(int argc, char* argv[]) {
9      if (argc != 3) {
10         std::cout << "usage: random_query csv_file dat_file" << std::endl;
11         return -1;
12     }
13
14     const int N = 1;
15     using set_type = uint_type<N>;
16     using data_type = uint8_t;
17
18     std::vector<data_type> D;
19     auto [res, n, m] = read_csv(argv[1], D);
20     if (res == false) {
21         std::cout << "error: unable to read input data" << std::endl;
22         return -1;
23     }
24
25     std::vector<std::pair<set_type, set_type>> Q;
26     if (!read_queries(argv[2], Q)) {
27         std::cout << "error: unable to read queries" << std::endl;
28         return -1;
29     }
30
31     std::vector<estimator> F(1);
32
33     auto eng = sabnatk::create_counter<N, sabnatk::BV>(n, m, std::begin(D));
34

```

```
35     for (const auto& q : Q) eng.apply(q.first, q.second, F);
36
37     return 0;
38 }
```

Listing 3.4. Realizacja strumienia zapytań Q na zbiorze danych D z wykorzystaniem biblioteki SABNAtk

4. Metody poprawy wydajności strategii Bitmap

Jak już wcześniej wspomniano, biblioteka SABNAtk stanowiła punkt wyjścia do prowadzonych badań, dotyczących efektywnej realizacji zapytań zliczających. Badania rozpoczęto od analizy strategii realizacji zapytań zliczających, oferowanych przez SABNAtk. Przeprowadzona analiza obejmowała zapoznanie się z algorytmami wykorzystywanymi przez strategię oraz ich otwartym kodem źródłowym.

Szczególną uwagę poświęcono strategii Bitmap, której szczegóły zostały przedstawione w sekcji 3.3.2. Efektem dokonanej analizy było zaproponowanie dwóch metod dla strategii Bitmap, pozwalających na przyspieszenie czasu realizacji zapytań zliczających.

Opracowane metody spotkały się z aprobatą twórców biblioteki a autor niniejszej rozprawy doktorskiej uzyskał możliwość zaimplementowania ich w oficjalnej wersji biblioteki. Ponadto, zostały one zaprezentowane podczas konferencji Euro-Par 2025 oraz opublikowane w materiałach pokonferencyjnych [15].

4.1. Metoda redukcji liczby przecięć bitmap (*SNIJK*)

Pierwsza z opracowanych metod pozwala na przyspieszenie realizacji zapytań zliczających poprzez zredukowanie liczby wykonywanych przecięć bitmap. Metoda ta skupia się na ostatniej warstwie tworzonego drzewa DFS, w której następuje zgłaszanie liczb N_{ij} oraz N_{ijk} . Warstwa ta zawiera największą liczbę węzłów, co wynika z hierarchicznej struktury drzewa DFS. To właśnie w tej warstwie drzewa realizowanych jest najwięcej przecięć bitmap, co czyni ją najbardziej kosztowną pod względem wykonywanych obliczeń.

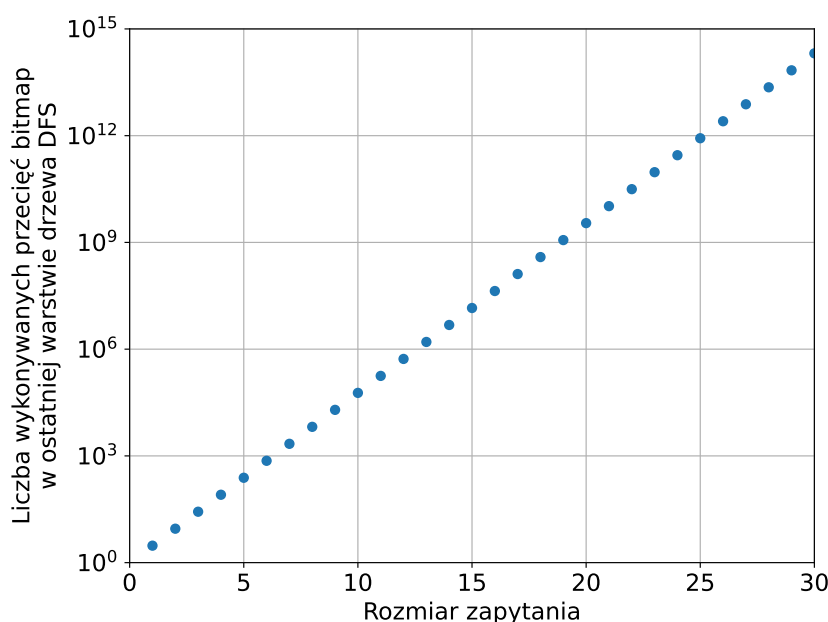
Przykład prostego drzewa DFS pokazany został na rysunku 3.5 w sekcji 3.3.2, gdzie omówiono szczegóły strategii bitmap. Ostatnia warstwa prezentowanego tam drzewa składa się z sześciu węzłów, z których każdy posiada trzy stany. W rezultacie w ostatniej warstwie drzewa wykonanych zostanie 18 przecięć bitmap, z których pięć zakończy się zgłoszeniem wyników poprzez wywołanie obiektu funkcyjnego F .

Należy zauważyć, że omawiane drzewo DFS dotyczy niewielkiego zapytania, składającego się z zaledwie trzech zmiennych. W rzeczywistych zastosowaniach realizowane zapytania zliczające mogą jednak obejmować znacznie większą liczbę zmiennych, co prowadzi do dynamicznego rozrastania się tworzonego drzewa. To z kolei w bezpośredni sposób przekłada się na dużą liczbę przecięć bitmap wykonywanych w ostatniej warstwie drzewa DFS.

Liczbę tych przecięć można obliczyć w prosty sposób, jako iloczyn stanów wszystkich zmiennych, wchodzących w skład zapytania. Dla omówionego przykładu iloczyn ten wyniesie 18 ($2 \cdot 3 \cdot 3$). Warto zauważyć, że oznacza to, iż liczba wykonywanych przecięć w ostatniej bitmap w ostatniej warstwie drzewa DFS rośnie w sposób wykładniczy w zależności od liczby zmiennych wchodzących w skład zapytania. Oznaczając przez n liczbę zmiennych zapytania oraz zakładając, że każda zmienna zapytania może przyjmować tę samą liczbę k stanów, liczba wykonywanych przecięć w ostatniej warstwie drzewa DFS może zostać obliczona jako k^n .

Rysunek 4.1 prezentuje tempo wzrostu liczby wykonywanych przecięć bitmap w ostatniej warstwie drzewa DFS w zależności od liczby zmiennych wchodzących w skład zapytania. Przyjęto tutaj, że każda ze zmiennych ma dokładnie trzy stany. Oś y została natomiast przedstawiona w skali logarytmicznej, aby możliwe było zaprezentowanie wartości rosnących w tempie wykładniczym.

Z rysunku wynika, że przy powyższym założeniu, zapytanie składające się z siedmiu zmiennych będzie wymagało wykonania ponad tysiąca przecięć bitmap w ostatniej warstwie drzewa DFS ($3^7 = 2187$). Warto odnotować, że liczba miliona przecięć zostanie szybko przekroczona - już dla zapytania składającego się z 13 zmiennych. Z kolei realizacja zapytania składającego się z 26 zmiennych skutkuje wykonaniem ponad biliona przecięć w ostatniej warstwie drzewa DFS.



Rysunek 4.1. Liczba wykonywanych przecięć bitmap w ostatniej warstwie drzewa DFS w zależności od liczby zmiennych wchodzących w skład zapytania przy założeniu, że każda zmienna przyjmuje trzy stany

Przeprowadzona analiza algorytmu bitmap pozwoliła na zaobserwowanie, że wiele z re-

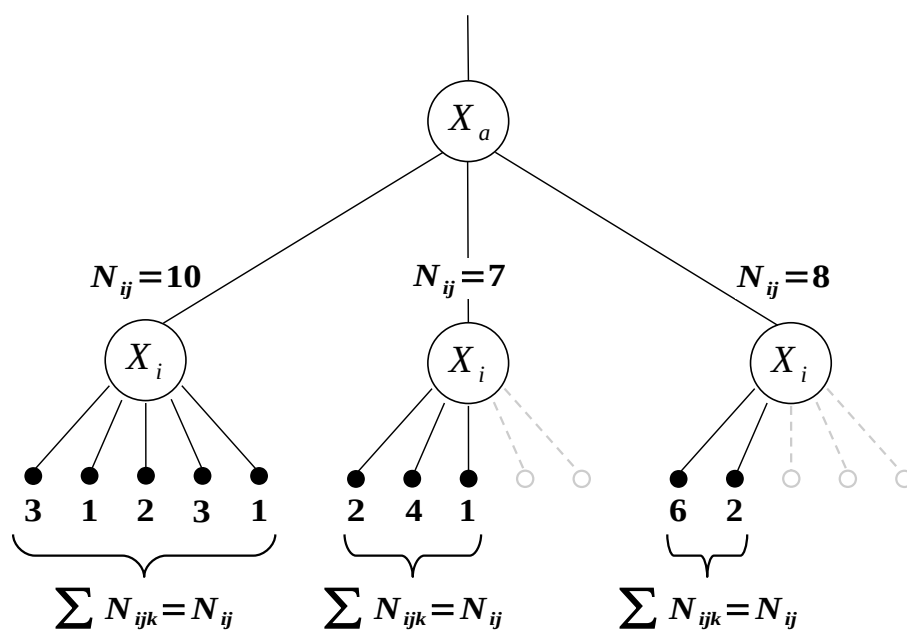
alizowanych przecięć może zostać pominięte. Istotą tej obserwacji jest wykorzystanie sformułowanego wcześniej równania 3.4, mówiącego o tym, że dla danej kombinacji zmiennych wchodzących w skład równania, liczba N_{ij} równa jest sumie liczb N_{ijk} .

W praktyce strategii bitmap, oznacza to, że liczby N_{ijk} zgłaszane dla danego wężła sumują się zawsze do odpowiadającej mu liczby N_{ij} . W konsekwencji, analizowanie sumy zgłoszonych już dla danego wężła wartości N_{ijk} może pozwolić na ograniczenie liczby wykonywanych przecięć bitmap. Jeżeli bowiem suma zgłoszonych liczb N_{ijk} dla danego wężła jest już równa odpowiadającej mu liczbie N_{ij} , to wykonywanie dalszych przecięć bitmap, związanych z kolejnymi stanami, może zostać pominięte, ponieważ będą one prowadzić tylko do zerowych wartości N_{ijk} .

Przykłady mające na celu zaprezentowanie sytuacji, w których opisana metoda znajduje zastosowanie zostały pokazane na rysunku 4.2. Przedstawia on fragment przykładowego drzewa DFS, a dokładniej jeden z węzłów (X_a) przedostatniej warstwy drzewa oraz odpowiadającą mu ostatnią warstwę, składającą się z trzech węzłów X_i .

Pierwszy z przedstawionych przypadków ($N_{ij} = 10$) zakłada negatywny scenariusz, w którym metoda nie przyniesie żadnych efektów. Wynika to z tego, że w przedstawionej sytuacji konieczne okazuje się przeanalizowanie wszystkich pięciu bitmap zmiennej X_i , ponieważ dopiero wtedy uzyskane zostaną wszystkie niezerowe wartości N_{ijk} (3, 1, 2, 3, 1), składające się na liczbę $N_{ij} = 10$.

Bardziej korzystna sytuacja z punktu widzenia zastosowania opracowanej metody została przedstawiona na drugim z przedstawionych przypadków ($N_{ij} = 7$). W tym przypadku po wykonaniu przecięć dla trzech pierwszych bitmap okazuje się, że suma zgłoszonych liczb N_{ijk} (2, 4, 1) jest już równa odpowiadającej im liczbie N_{ij} . Oznacza to, że wykonywanie przecięć związanych z dwiema ostatnimi bitmapami może zostać pominięte, ponieważ będzie skutkowało liczbami N_{ijk} równymi 0. Analogiczną sytuację można zaobserwować w przypadku ostatniego wężła ($N_{ij} = 8$), gdzie analiza dwóch pierwszych bitmap pozwoli na zgłoszenie wszystkich niezerowych wartości N_{ijk} (6, 2). Po zostale trzy bitmapy mogą zostać pominięte, ponieważ ich przecięcia będą skutkowały zerowymi wartościami N_{ijk} .



Rysunek 4.2. Redukcja liczby przecięć bitmap w przykładowym fragmencie drzewa DFS uzyskana dzięki zastosowaniu opracowanej metody *SNIJK*

Algorytm 4 QUERY(X_i, Pa, F, b)

```

1: if  $|Pa| = 0$  then
2:    $N_{ij} \leftarrow |b|$ 
3:    $S_{ijk} \leftarrow 0$ 
4:   for  $v \in [0, \dots, r_i - 1]$  do
5:      $b_v \leftarrow \{p \mid D_i[p] = v\}$ 
6:      $N_{ijk} \leftarrow |b \cap b_v|$ 
7:     if  $N_{ijk} > 0$  then
8:        $F(N_{ijk}, N_{ij}) \triangleleft \text{emit new configuration}$ 
9:        $S_{ijk} \leftarrow S_{ijk} + N_{ijk}$ 
10:      if  $S_{ijk} = N_{ij}$  then
11:        break for
12:   else
13:      $X_h \leftarrow \text{HEAD}(Pa)$ 
14:     for  $v \in [0, \dots, r_h - 1]$  do
15:        $b_v \leftarrow \{p \mid D_h[p] = v\}$ 
16:       if  $|b \cap b_v| > 0$  then
17:         QUERY( $X_i, \text{TAIL}(Pa), F, b \cap b_v$ )

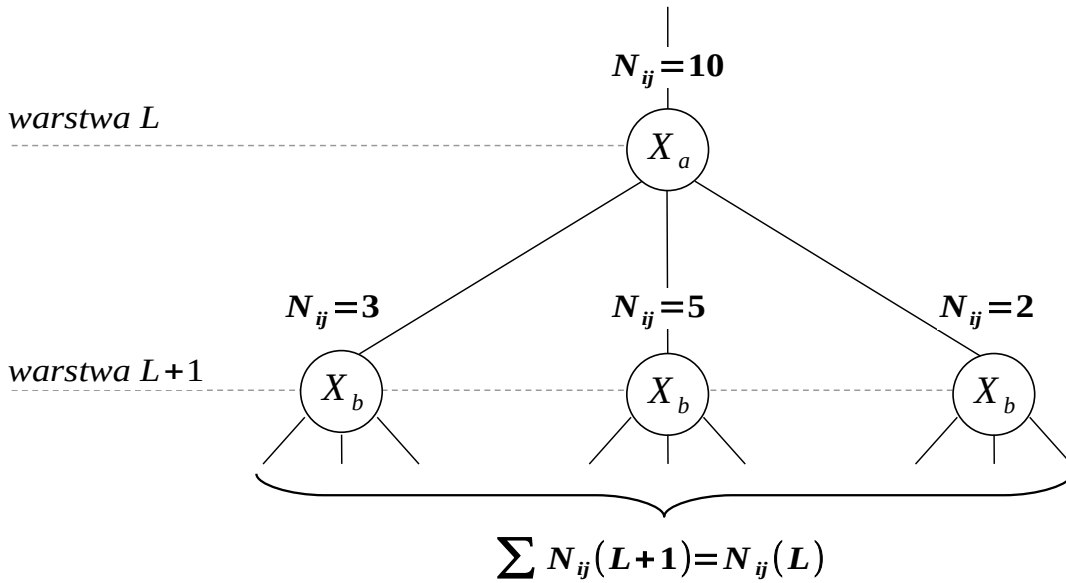
```

Opracowana metoda została przedstawiona w Algorytmie 4. Linie wyróżnione kolorem pokazują zmiany wprowadzone w porównaniu do oryginalnego Algorytmu 1, opisanego w sekcji 3.3.2. Metoda znajduje swoje zastosowanie w ostatniej warstwie drzewa DFS, czyli gdy spełniony jest warunek $|Pa| = 0$. W linii 3. zdefiniowano nową zmienną S_{ijk} i zainicjowano ją wartością 0. Zadaniem tej zmiennej jest kontrolowanie sumy zgłoszonych liczb N_{ijk} dla danego węzła. W algorytmie dla każdej bitmapy danego węzła (pętla **for**) wykonywane jest przecięcie z bitmapą b . W przypadku gdy wynikiem tego przecięcia jest bitmapa niezerowa ($N_{ijk} > 0$), następuje przekazanie wyników poprzez wywołanie obiektu funkcyjnego F . Dzięki opracowanej metodzie, zgłoszona liczba N_{ijk} zostaje dodana do zdefiniowanej wcześniej zmiennej S_{ijk} . Następnie następuje sprawdzenie, czy suma zgłoszonych N_{ijk} , oznaczona jak S_{ijk} osiągnęła już wymaganą wartość N_{ij} . Spełnienie tego warunku pozwala na przerwanie analizowania pozostałych bitmap i przejście do kolejnej części algorytmu.

4.2. Metoda redukcji liczby odwiedzanych węzłów drzewa DFS (SNIJ)

Druga z opracowanych metod pozwala na przyspieszenie realizacji zapytań zliczających poprzez zredukowanie liczby odwiedzanych węzłów drzewa DFS. Metoda ta stanowi pewną analogię do opisanej w poprzedniej sekcji metody *SNIIK*. Podczas gdy metoda *SNIIK* dotyczy jedynie ostatniej warstwy drzewa, metoda *SNIJ* uogólnia wykorzystaną tam zasadę i przenosi ją na wszystkie pozostałe warstwy drzewa.

Analiza działania algorytmu bitmap oraz struktury powstającego w jego wyniku drzewa DFS pozwoliła na ujawnienie właściwości, która może zostać wykorzystana do optymalizacji procesu przeszukiwania. Zaobserwowano, że dla każdego węzła znajdującego się w warstwie L , odpowiadająca mu liczba N_{ij} jest równa sumie liczb N_{ij} , przypisanych jego bezpośrednim potomkom z warstwy $L + 1$. Obserwacja ta została zwizualizowana na rysunku 4.3, przedstawiającym fragment drzewa DFS wygenerowanego podczas realizacji przykładowego zapytania zliczającego.



Rysunek 4.3. Fragment drzewa DFS pokazujący obserwację wykorzystywaną w metodzie SNIJ

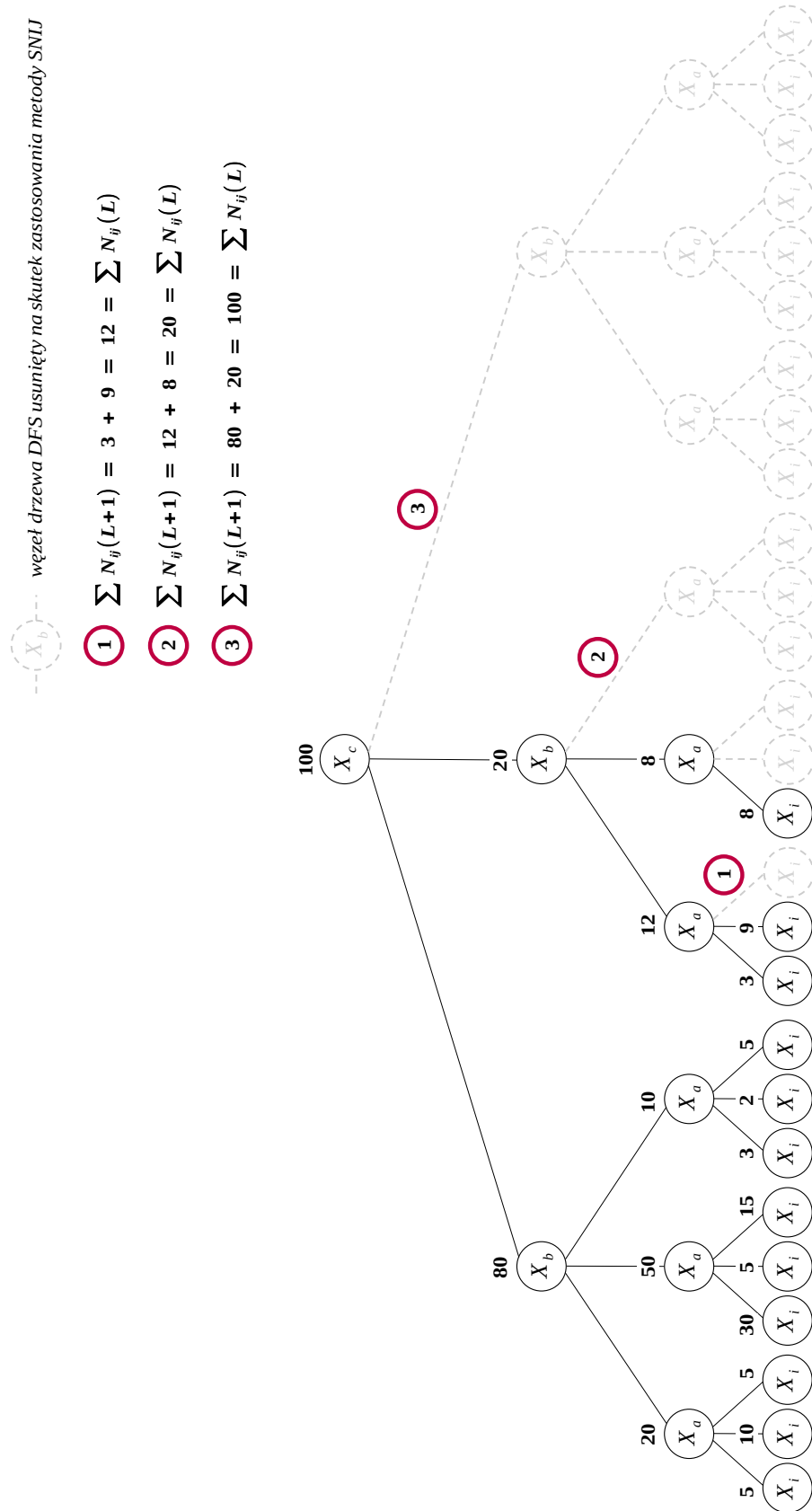
Wykorzystanie wspomnianej właściwości pozwala na wprowadzenie mechanizmu redukcji przestrzeni przeszukiwań (ang. pruning). Polega on na kontrolowaniu obliczonych liczb N_{ij} dla danej warstwy drzewa w celu ograniczenia liczby odwiedzanych węzłów drzewa DFS. W momencie, gdy suma zgłoszonych liczb N_{ij} dla warstwy $L+1$ jest równa liczbie N_{ij} odpowiadającej węzłowi z warstwy L , to analiza kolejnych węzłów z tej warstwy może zostać pominięta, ponieważ będzie prowadzić do zerowych wartości N_{ij} . Warto zauważyć, że usunięcie danego węzła z drzewa powoduje również usunięcie węzłów będących jego potomkami, czyli całego poddrzewa zakorzenionego w analizowanym węźle. Opracowana metoda może więc pozwolić na istotne zredukowanie liczby odwiedzanych węzłów, co będzie miało bezpośredni wpływ na czas realizacji zapytań zliczających.

Idea ta została zobrazowana na rysunku 4.4, który przedstawia pełne drzewo DFS powstałe podczas realizacji zliczającego $\text{COUNT}(X_i \mid Pa(X_i) = \{X_a, X_b, X_c\})$. W przedstawionym drzewie szarym kolorem zaznaczono węzły, które zostaną pominięte dzięki opracowanej metodzie. Nad każdym węzłem znajduje się związana z nim liczba N_{ij} .

Aby zapewnić czytelność rysunku, zdecydowano o pominięciu rozpisania stanów węzłów z ostatniej warstwy drzewa. Z punktu widzenia opisywanej metody *SNIJ* są one zbędne, w przeciwieństwie do metody *SNJK*, gdzie odgrywały kluczową rolę, co pokazano np. na rysunku 4.2.

Lewa część drzewa ($N_{ij} = 80$) przedstawia negatywny scenariusz z punktu widzenia możliwości zastosowania opisywanej metody. W tym przypadku każdorazowo konieczne okazuje się wykonanie przecięć bitmap dla wszystkich możliwych stanów zmiennych X_b oraz X_a . Przykładowo, znajdując się w węźle X_a , dla którego liczba $N_{ij} = 20$, niezbędne jest przeanalizowanie wszystkich 3 bitmap prowadzących do węzłów X_i , ponieważ dopiero wtedy suma odpowiadających im liczb N_{ij} (5, 10, 5) osiągnie wartość równą liczbie N_{ij} z węzła X_a . Analogiczną sytuację obserwujemy w dwóch pozostałych węzłach X_a , będących potomkami węzła X_b , dla którego $N_{ij} = 80$. W pierwszym z nich, aby uzyskać wymaganą liczbę $N_{ij} = 50$, należy rozpatrzyć 3 bitmapy prowadzące do węzłów X_i , dla których liczby N_{ij} to 30, 5 i 15. W drugim natomiast, liczbie $N_{ij} = 10$ dla węzła X_a odpowiadają liczby N_{ij} równe 3, 2 oraz 5, związane z węzłami X_i .

Korzystniejszą sytuację można zaobserwować w środkowej części drzewa ($N_{ij} = 20$). Analizując bitmapy węzła X_a , dla którego $N_{ij} = 12$, okazuje się, że wystarczające jest wykonanie przecięć dwóch pierwszych bitmap prowadzących do węzłów X_i , ponieważ suma odpowiadających im liczb N_{ij} będzie już równa liczbie N_{ij} z węzła X_a , znajdującego się w wyższej warstwie ($3 + 9 = 12$). W tym przypadku ostatnia bitmapa zmiennej X_a może zostać pominięta, ponieważ jej wykonanie poskutkowałoby liczbą $N_{ij} = 0$, niemającą praktycznego znaczenia. Analizując bitmapy kolejnego węzła X_a ($N_{ij} = 8$) okazuje się, że już pierwsza z nich skutkuje odkryciem wymaganej liczby $N_{ij} = 8$, dlatego wykonywanie przecięć dla dwóch pozostałych stanów zmiennej X_a , może zostać w tym przypadku pominięte, ponieważ tak samo jak poprzednio, doprowadziłoby do węzłów X_i , dla których liczby N_{ij} są równe 0. Środkowa część drzewa ($N_{ij} = 20$) pokazuje również, w jaki sposób metoda znajduje zastosowanie w dowolnych warstwach drzewa. Suma liczb N_{ij} dla dwóch pierwszych węzłów X_a ($12 + 8$) odpowiada liczbie N_{ij} z węzła X_b , znajdującego się w wyższej warstwie. Oznacza to, że analiza trzeciej bitmapy zmiennej X_b byłaby w tym przypadku bezcelowa, ponieważ poskutkowałaby liczbą $N_{ij} = 0$. Pozwoliło to więc, na usunięcie z rozpatrywanego drzewa węzła X_a wraz z całym poddrzewem zakorzenionym w tym węźle (3 węzły X_i). Analogicznie, zastosowanie opracowanej metody do wyższej warstwy drzewa pozwala na pominięcie całej prawej części drzewa DFS. Suma liczb N_{ij} z węzłów X_b odpowiada już liczbie N_{ij} węzła X_c ($80 + 20 = 100$). Możliwe jest więc usunięcie z drzewa węzła X_b , znajdującego się po prawej stronie wraz z całym poddrzewem zakorzenionym w tym węźle, a więc 3 węzłami X_a oraz 9 węzłami X_i .



Rysunek 4.4. Redukcja liczby węzłów w przykładowym drzewie DFS uzyskana dzięki zastosowaniu opracowanej metody SNIJ

Opracowana metoda została przedstawiona w Algorytmie 5. Linie wyróżnione kolorem niebieskim pokazują zmiany wprowadzone w porównaniu do oryginalnego Algorytmu 1, opisanego w sekcji 3.3.2. W przeciwieństwie do metody *SNIJK*, która miała zastosowanie tylko w ostatniej warstwie drzewa DFS, metoda *SNIJ* skupia się na wyższych warstwach drzewa (linie 13-22). Wywołując algorytm dla węzła X_i niebędącego liściem ($|Pa| \neq 0$), pierwszy z dodatkowych kroków polega na utworzeniu zmiennej S_{ij} . Zadaniem tej zmiennej będzie przechowywanie sumy liczb N_{ij} , związanych z bezpośrednimi potomkami (dziećmi) rozpatrywanego węzła X_i , czyli węzłami znajdującymi się w kolejnej warstwie drzewa DFS. Liczba węzłów w kolejnej warstwie wynika z liczby stanów przyjmowanych przez węzeł X_i , dla którego wywołano algorytm. Dla każdego węzła z kolejnej warstwy (pętla for) obliczana jest liczba N_{ij} . Następnie, jeżeli jej wartość jest dodatnia, to powiększa ona zdefiniowaną liczbę S_{ij} . Po rozpatrzeniu poddrzewa tworzonego przez analizowany węzeł kolejnej warstwy (linia 20), następuje sprawdzenie czy suma liczb N_{ij} (S_{ij}) z węzłów kolejnej warstwy jest równa liczbie N_{ij} ($|b|$), związanej z węzłem X_i , dla którego wywołano algorytm. Spełnienie tego warunku pozwala na przerwanie analizowania pozostałych dzieci węzła X_i , a tym samym usunięcie ich wraz z całymi poddrzewami zakorzenionymi w tych węzłach.

Algorytm 5 QUERY(X_i, Pa, F, b)

```

1: if  $|Pa| = 0$  then
2:    $N_{ij} \leftarrow |b|$ 
3:    $S_{ijk} \leftarrow 0$ 
4:   for  $v \in [0, \dots, r_i - 1]$  do
5:      $b_v \leftarrow \{p \mid D_i[p] = v\}$ 
6:      $N_{ijk} \leftarrow |b \cap b_v|$ 
7:     if  $N_{ijk} > 0$  then
8:        $F(N_{ijk}, N_{ij}) \triangleleft \text{emit new configuration}$ 
9:        $S_{ijk} \leftarrow S_{ijk} + N_{ijk}$ 
10:    if  $S_{ijk} = N_{ij}$  then
11:      break for
12: else
13:    $X_h \leftarrow \text{HEAD}(Pa)$ 
14:    $S_{ij} \leftarrow 0$ 
15:   for  $v \in [0, \dots, r_h - 1]$  do
16:      $b_v \leftarrow \{p \mid D_h[p] = v\}$ 
17:      $N_{ij} \leftarrow |b \cap b_v|$ 
18:     if  $N_{ij} > 0$  then
19:        $S_{ij} \leftarrow S_{ij} + N_{ij}$ 
20:     QUERY( $X_i, \text{TAIL}(Pa), F, b \cap b_v$ )
21:     if  $S_{ij} = |b|$  then
22:       break for

```

4.3. Kod źródłowy opracowanych metod

Metody przedstawione w sekcjach 4.1 oraz 4.2 zostały wdrożone przez autora niniejszej pracy w oficjalnej wersji biblioteki SABNAtk. Kod źródłowy przedstawiający wprowadzone zmiany dostępny jest na platformie Gitlab w repozytorium biblioteki SABNAtk [58]. Wprowadzone zmiany zostały również zamieszczone w aneksie niniejszej pracy na listingu 8.1.

4.4. Badanie wpływu opracowanych metod na wydajność zapytań zliczających

W ramach niniejszej sekcji zaprezentowane zostaną efekty opracowanych metod, przedstawionych w sekcjach 4.1 oraz 4.2. Celem przeprowadzonych testów było zbadanie wpływu opracowanych metod na czas realizacji zapytań zliczających. Ocena skuteczności oparta została na benchmarku polegającym na równoległej realizacji strumienia dużej liczby losowych zapytań dla różnych zbiorów danych. Szczegóły tego podejścia zostały przedstawione w sekcji 3.4.3.

Do badań wybrano osiem zbiorów danych z tabeli 3.2. Dla każdego zbioru rozpatrywano cztery wersje różniące się liczbą obserwacji w zakresie od 1K do 1M. Jak wspomniano wcześniej, liczba obserwacji w zbiorze danych ma istotny wpływ na czas realizacji zapytań zliczających — większa liczba obserwacji może znacząco wydłużyć czas ich przetwarzania. Z tego względu, wykonanie pełnego strumienia dziesięciu milionów zapytań dla każdej wersji zbioru danych okazało się zbyt czasochłonne. W związku z tym, aby zachować czas realizacji pojedynczego przypadku w rozsądnych granicach (nie dłużej niż kilka godzin), konieczne okazało się dostosowanie długości przetwarzanego strumienia w zależności od liczby obserwacji w zbiorze danych. Szczegółowy dobór liczby zapytań dla poszczególnych wersji zbiorów przedstawiono w tabeli 4.1.

Liczba obserwacji	Liczba zapytań
1K	10M
10K	1M
100K	100K
1M	1K

Tabela 4.1: Dobór liczby zapytań do liczby obserwacji zbioru danych podczas testowania opracowanych metod dla strategii Bitmap

Rozpatrzenie ośmiu zbiorów danych z tabeli 3.2 w czterech różnych rozmiarach, utworzyło łącznie 32 przypadki testowe. Podczas realizacji każdego z nich mierzono całkowity czas wykonania strumienia zapytań zliczających.

W każdym rozpatrywanym przypadku strumień zapytań został zrównoleglony z wykorzystaniem biblioteki OpenMP. Szczegóły techniczne tego podejścia, w tym fragment kodu źródłowego odpowiedzialny za równoległą realizację zapytań, zostały przedstawione w sekcji 3.3.5 oraz na listingu 3.3.

Zbiór danych (n)	Przyspieszenie			
	$m = 1K$	$m = 10K$	$m = 100K$	$m = 1M$
Sachs	1.17	1.21	1.19	1.17
Child	1.26	1.35	1.35	1.3
Insurance	1.24	1.3	1.33	1.3
Mildew	1.48	1.62	1.76	2.25
Alarm	1.34	1.31	1.29	1.26
Barley	1.44	1.54	1.56	1.6
Hailfinder	1.16	1.13	1.14	1.15
Pathfinder	1.53	1.57	1.66	1.53
Średnia	1.33	1.38	1.41	1.45

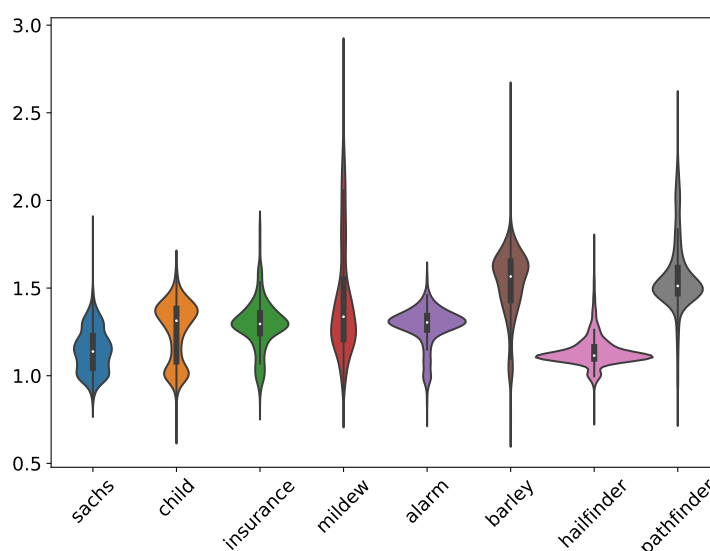
Tabela 4.2: Przyspieszenie uzyskane w benchmarku losowych zapytań dzięki opracowanym metodom *SNIJ* i *SNIJK* dla strategii Bitmap. Platforma obliczeniowa: Intel Cascade Lake

Pierwsze badania zostały przeprowadzone na platformie obliczeniowej Intel Cascade Lake, której specyfikację opisano w sekcji 3.4.1. Tabela 4.2 przedstawia przyspieszenie uzyskane, dzięki wprowadzeniu opracowanych metod *SNIJ* i *SNIJK* do strategii Bitmap. Należy zauważyć, że w każdym z testowanych przypadków skrócono czas realizacji całego strumienia zapytań zliczających. Uzyskane przyspieszenie różni się od zbioru danych. Opracowane metody odniosły najmniejszy efekt w zbiorze danych Hailfinder, przyspieszając realizację zadania od 1.13x do 1.16x. Z kolei najlepsze wyniki zostały uzyskane dla zbioru danych Mildew, w którym osiągnięte przyspieszenie waha się od 1.48x w przypadku zbioru danych zawierającego tysiąc obserwacji, do 2.25x dla zbioru danych składającego się z miliona obserwacji. Szczególnie dobre wyniki uzyskane dla tego zbioru należy wiązać ze stosunkowo sporą liczbą stanów, jakie mogą przyjmować zmienne tego zbioru danych (tabela 3.2). Duża liczba stanów przyjmowanych przez zmienną X_i skutkuje koniecznością wykonania dużej liczby przecięć w ostatniej warstwie drzewa DFS. Liczba tych przecięć jest jednak skutecznie zredukowana poprzez metodę *SNIJK*, która eliminuje wykonywanie tych przecięć, które nie niosą żadnego praktycznego znaczenia. Z drugiej strony, duża liczba stanów przyjmowanych przez zmienne ze zbioru $Pa(X_i)$ skutkuje dynamicznym rozrastaniem się drzewa DFS. Proces ten jest jednak ograniczony do minimum dzięki metodzie *SNIJ*, która usuwa z drzewa zbędne węzły wraz z całymi poddrzewami zakorzenionymi w tych węzłach, tak jak przedstawiono na rysunku 4.4.

Analizując średnie przyspieszenie uzyskane ze wszystkich zbiorów, można zauważyć, że rośnie ono wraz ze wzrostem liczby obserwacji. Podczas gdy dla zbiorów składających się z tysiąca obserwacji wyniosło ono 1.33x, w zbiorach zawierających milion obserwacji średni wynik to 1.45x. Trend ten wynika jednak przede wszystkim z rezultatów osiągniętych w zbiorach, których zmienne mogą przyjmować dużą liczbę stanów, takich jak Mildew oraz Barley. W pozostałych zbiorach danych osiągnięte przyspieszenie nie jest w tak dużym stopniu zależne od liczby obserwacji.

Aby lepiej zobrazować efekty opracowanych metod, oprócz określenia przyspieszenia czasu realizacji całego strumienia zapytań, przeanalizowano również przyspieszenia indywidualnych zapytań ze strumienia. Rysunek 4.5 przedstawia zestaw wykresów skrzypcowych (violin plot) dla testowanych zbiorów danych, składających się z dziesięciu tysięcy obserwacji. Źródłem danych tych wykresów są przyspieszenia uzyskane dla wszystkich stu tysięcy zapytań rozpatrywanych dla tych zbiorów danych. Wykres skrzypcowy łączy cechy wykresu pudełkowego (box plot) oraz wykresu gęstości (kernel density plot), umożliwiając jednocześnie przedstawienie podstawowych statystyk opisowych oraz kształtu rozkładu danych. Oprócz mediany (biały punkt) i kwartyliów wizualizuje on również zmienność danych oraz ich koncentrację w określonych przedziałach, co pozwala na bardziej kompleksową analizę.

Pomimo tego, że najlepsze rezultaty pod kątem średniego przyspieszenia w każdym analizowanym przypadku osiągnięto dla zbioru Mildew (tabela 4.2), z prezentowanych wykresów można odczytać, że analizując medianę wyników, lepsze rezultaty odnotowano w zbiorze Barley. Można również zauważyć, że dla wąskiej grupy zapytań ze zbiorów Mildew, Barley oraz Pathfinder, osiągnięte przyspieszenie jest niemalże 3-krotne. Z kolei wykres dla zbioru danych Hailfinder dostarcza informacji na temat dużego skupienia danych w okół mediany, potwierdzając najslabsze rezultaty osiągnięte dla tego zbioru danych.

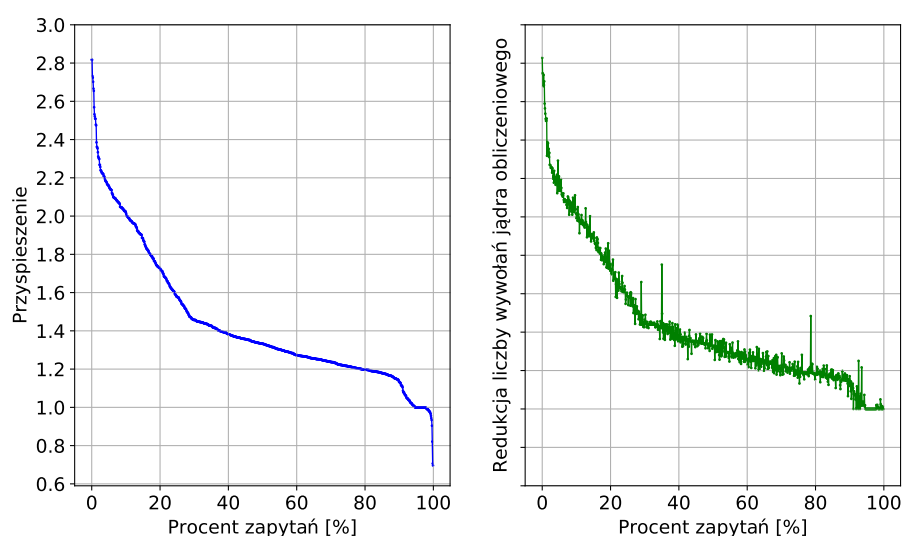


Rysunek 4.5. Wykresy skrzypcowe dla różnych zbiorów danych prezentujące przyspieszenie zapytań zliczających uzyskane dzięki opracowanym metodom dla strategii Bitmap

Celem dalszej analizy było zbadanie wpływu opracowanych metod na liczbę wykonywanych przecięć bitmap podczas realizacji zapytań zliczających. Operacja przecinania bitmap stanowi najbardziej czasochłonny fragment algorytmu, który może być nazwany jądrem obliczeniowym.

Aby ocenić wpływ metod, z otrzymanych wyników wybrano w sposób losowy tysiąc zapytań i przeanalizowano jak zmieniła się w nich liczba wykonanych przecięć bitmap. Dla każdego zapytania utworzono parę wartości: pierwsza oznaczała przyspieszenie czasu realizacji, a druga redukcję liczby wywołań jądra obliczeniowego. Redukcję tę, obliczono jako iloraz liczby wywołań jądra obliczeniowego w wersji oryginalnej do liczby wywołań w wersji z opracowanymi metodami. Otrzymane pary zostały następnie posortowane względem pierwszej wartości, czyli uzyskanego przyspieszenia.

Rysunek 4.6 przedstawia wyniki analizy dla zbioru danych Mildew, zawierającego dziesięć tysięcy zapytań. Lewy wykres ilustruje przyspieszenia czasu realizacji zapytań zliczających, natomiast prawy odpowiadającą im redukcję liczby wywołań jądra obliczeniowego. Taki sposób prezentacji danych dotyczących przyspieszenia, pozwala szczegółowo przeanalizować rozkład przyspieszenia w analizowanej grupie zapytań. Przykładowo, możliwe jest odczytanie informacji, iż dla około 40% zapytań uzyskane przyspieszenie przekracza 1.4x a dla około 10% procent jest nawet ponad dwukrotne. Prawy wykres pokazuje natomiast, że redukcja liczby wywołań jądra obliczeniowego przebiega zgodnie z rozkładem przyspieszeń. Oznacza to, że uzyskane przyspieszenie jest skutkiem zmniejszenia liczby operacji przecięć bitmap, osiągniętego dzięki opracowanym metodom.



Rysunek 4.6. Analiza przyspieszenia oraz redukcji liczby wywołań jądra obliczeniowego na przykładzie zbioru Mildew

Przedstawione dotychczas wyniki dotyczyły rezultatów uzyskanych dla procesorów Intel Cascade Lake. Opracowane metody zostały jednak również zweryfikowane z wykorzystaniem dwóch serwerów obliczeniowych, wyposażonych w procesory firmy AMD z serii Rome oraz Milan. Specyfikacja tych platform obliczeniowych została przedstawiona w sekcji 3.4.1.

Dla obu platform zrealizowano ten sam zestaw 32 przypadków testowych, obejmujących realizację strumieni losowych zapytań dla ośmiu różnych zestawów danych z tabeli 3.2, rozpatrywanych w czterech różnych konfiguracjach, scharakteryzowanych w tabeli 4.1. Uzyskane wyniki zostały zestawiono w tabelach 4.3 oraz 4.4, przedstawiających przyspieszenie osiągnięte dzięki wprowadzeniu opracowanych metod.

Zbiór danych (n)	Przyspieszenie			
	$m = 1K$	$m = 10K$	$m = 100K$	$m = 1M$
Sachs	1.09	1.21	1.21	1.2
Child	1.22	1.35	1.37	1.35
Insurance	1.18	1.29	1.34	1.35
Mildew	1.37	1.60	1.79	–
Alarm	1.20	1.29	1.29	1.38
Barley	1.31	1.54	1.59	1.84
Hailfinder	1.04	1.12	1.14	1.47
Pathfinder	1.33	1.54	1.65	–
Średnia	1.22	1.37	1.42	1.43

Tabela 4.3: Przyspieszenie uzyskane w benchmarku losowych zapytań dzięki opracowanym metodom *SNIJ* i *SNIJK* dla strategii Bitmap. Platforma obliczeniowa: AMD Rome

Zbiór danych (n)	Przyspieszenie			
	$m = 1K$	$m = 10K$	$m = 100K$	$m = 1M$
Sachs	1.10	1.31	1.34	1.28
Child	1.22	1.39	1.45	1.39
Insurance	1.17	1.32	1.40	1.36
Mildew	1.27	1.67	1.80	1.88
Alarm	1.20	1.31	1.34	1.24
Barley	1.35	1.55	1.60	1.66
Hailfinder	1.00	1.12	1.15	1.15
Pathfinder	1.35	1.53	1.65	–
Średnia	1.33	1.38	1.41	1.45

Tabela 4.4: Przyspieszenie uzyskane w benchmarku losowych zapytań dzięki opracowanym metodom *SNIJ* i *SNIJK* dla strategii Bitmap. Platforma obliczeniowa: AMD Milan

Brak niektórych rezultatów w tabelach wynika z przekroczenia w tych przypadkach maksymalnego dopuszczalnego czasu realizacji zadania. Dostęp do opisanych platform obliczeniowych AMD ograniczony był poprzez konieczność korzystania z systemu kolejkowego, w którym maksymalny przyznawany czas obliczeń wynosił cztery godziny. Otrzymane wyniki dla platform AMD są zbieżne z wynikami uzyskanymi dla platformy Intel. Podobnie jak w przypadku procesorów firmy Intel, najlepsze rezultaty zostały osiągnięte dla zbioru danych Mildew, w którym przyspieszenie znajduje się w przedziale 1.37 - 1.79x na platformie AMD Rome oraz 1.28 - 1.88x na platformie AMD Milan. W większości przypadków potwierdza się również trend wzrostu uzyskanego przyspieszenia wraz ze wzrostem liczby obserwacji danego zbioru danych.

5. Metoda automatycznego wyboru strategii

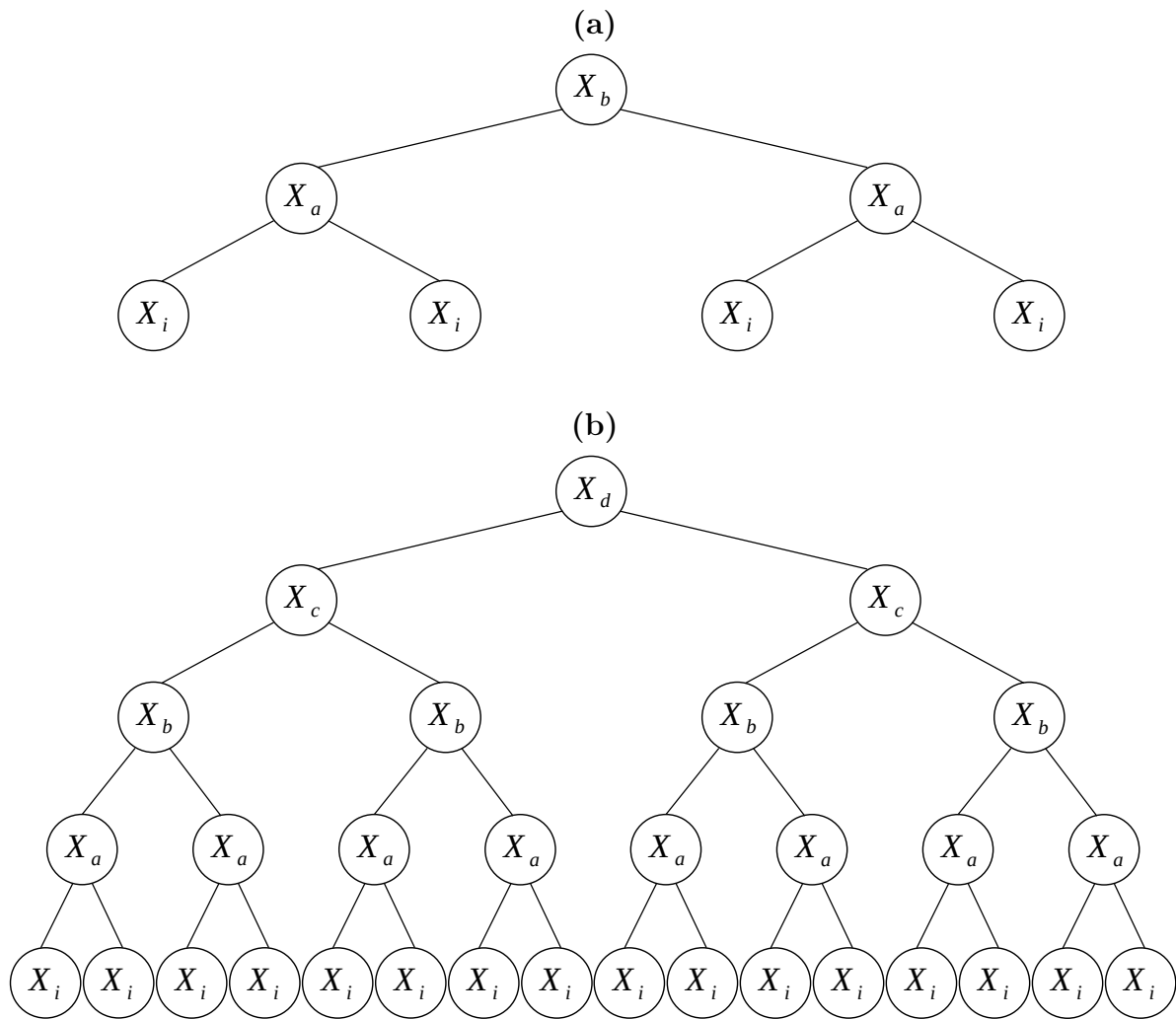
Niniejsza sekcja przedstawia opracowaną metodę automatycznego wyboru strategii realizacji zapytań zliczających. Celem tej metody jest skrócenie czasu realizacji strumienia zapytań poprzez indywidualny dobór optymalnej strategii do realizacji każdego zapytania. W rozdziale tym w sposób szczegółowy omówiono motywację podjęcia problemu, koncepcję rozwiązania, proces projektowania oraz uzyskane wyniki w zakresie wpływu opracowanej metody na wydajność realizacji zapytań zliczających. Idea omawianej metody została zaprezentowana podczas konferencji Euro-Par 2021 oraz opublikowana w materiałach pokonferencyjnych [16].

5.1. Analiza wydajności strategii realizacji zapytań zliczających

Po wdrożeniu metod opracowanych dla strategii Bitmap, kolejnym etapem prac była ocena efektywności wykonywania zapytań zliczających w kontekście obu strategii oferowanych przez bibliotekę SABNAtk. Głównym celem tego etapu było wskazanie obszarów, w których możliwa jest dalsza optymalizacja procesu realizacji zapytań zliczających.

Badania rozpoczęto od analizy wpływu różnych właściwości pojedynczego zapytania zliczającego na jego czas wykonania. Z racji tego, że wydajność zapytania zależy od wielu czynników, pierwszym krokiem było wyszczególnienie tych, które wywierają największy wpływ na teoretyczną wydajność zapytania zliczającego $\text{COUNT}(X_i \mid Pa(X_i))$. Przeprowadzona analiza pokazała, iż do najistotniejszych czynników wpływających na jego wydajność należą:

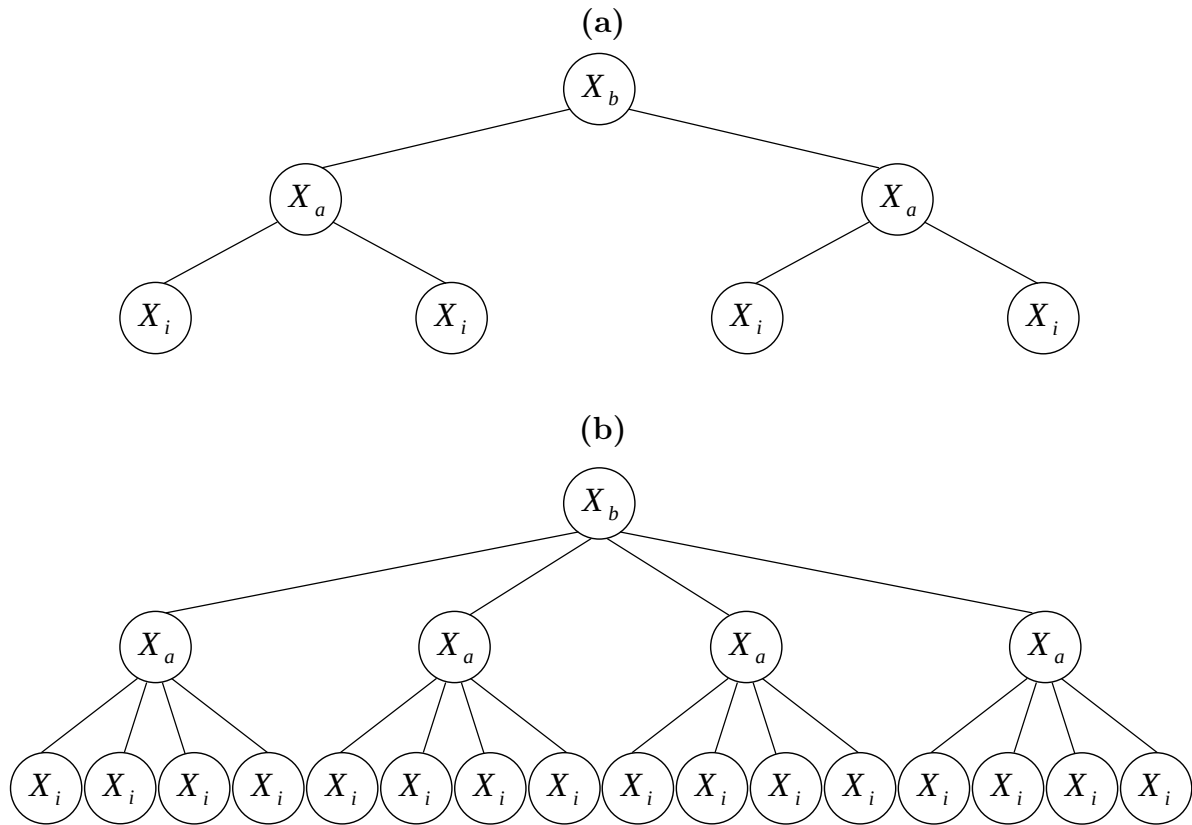
- rozmiar zapytania, czyli łączna liczba zmiennych uwzględnionych w zapytaniu,
- liczba możliwych stanów przyjmowanych przez wszystkie zmienne wchodzące w skład zapytania,
- liczba obserwacji m w analizowanym zbiorze danych D .



Rysunek 5.1. Drzewa DFS powstałe podczas realizacji zapytań: (a) $\text{COUNT}(X_i \mid \{X_a, X_b\})$ oraz (b) $\text{COUNT}(X_i \mid \{X_a, X_b, X_c, X_d\})$, przy założeniu, że wszystkie zmienne wchodzące w skład zapytań są binarne

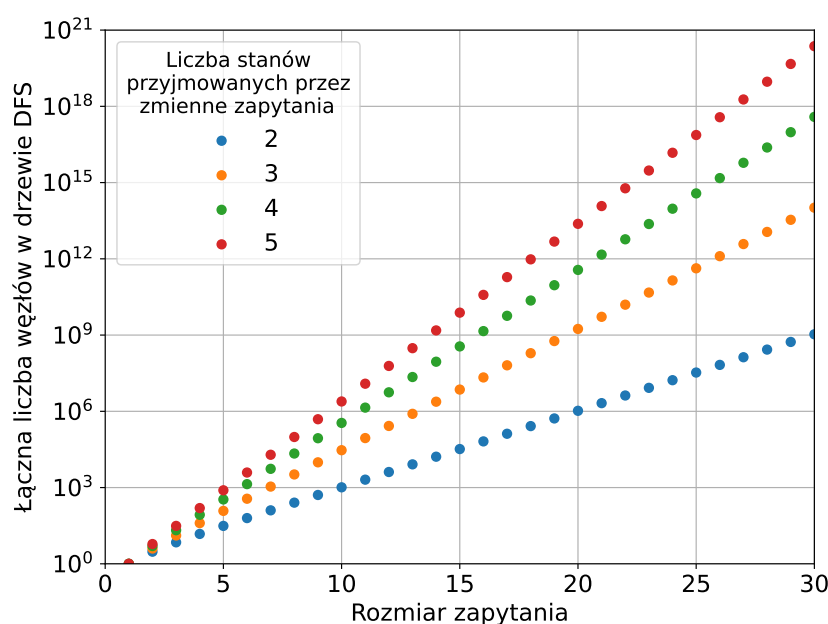
Praktyczne znaczenie wymienionych czynników dla wydajności zapytania zliczającego najlepiej zilustrować na przykładzie algorytmu strategii Bitmap, który został szczegółowo przeanalizowany we wcześniejszych sekcjach pracy. Rozpoczynając od liczby zmiennych wchodzących w skład zapytania, należy stwierdzić, że stanowi ona istotny parametr, ponieważ przekłada się bezpośrednio na liczbę warstw w budowanym drzewie DFS. Przykłady takich drzew zostały przedstawione na rysunku 5.1. Pierwsze z nich (a) składa się z trzech warstw, ponieważ powstało dla zapytania $\text{COUNT}(X_i \mid \{X_a, X_b\})$ dotyczącego łącznie trzech zmiennych. Drugie drzewo (b) liczy natomiast pięć warstw, ponieważ odpowiada zapytaniu $\text{COUNT}(X_i \mid \{X_a, X_b, X_c, X_d\})$ obejmującemu w sumie pięć zmiennych. Oznacza to, że każda zmienna wiąże się z osobną warstwą w strukturze drzewa DFS. Warto również zauważyć, że każda kolejna warstwa drzewa składa się z coraz większej liczby węzłów, co ma istotne znaczenie w kontekście złożoności obliczeniowej całego zapytania zliczającego.

Drugim z wymienionych parametrów jest liczba możliwych stanów przyjmowanych przez wszystkie zmienne wchodzące w skład zapytania. Parametr ten jest istotny, ponieważ znacząco wpływa na liczbę węzłów drzewa DFS powstającego podczas realizacji danego zapytania. Przykład takiej sytuacji został przedstawiony na rysunku 5.2. Oba zaprezentowane drzewa DFS dotyczą tego samego zapytania zliczającego $\text{COUNT}(X_i \mid \{X_a, X_b\})$. Różnica polega jednak na liczbie możliwych stanów przyjmowanych przez zmienne wchodzące w skład zapytania. W pierwszym prezentowanym przypadku (a) założono, że każda ze zmiennych X_i, X_a, X_b może przyjmować dwa stany. W rezultacie, utworzone drzewo DFS składa się łącznie z 7 węzłów. Z kolei w drugim przypadku (b) założono, że każda zmienna zapytania może przyjmować cztery stany. Efektem tego jest trzykrotny wzrost łącznej liczby węzłów ze wszystkich warstw drzewa.



Rysunek 5.2. Drzewa DFS powstałe podczas realizacji zapytania $\text{COUNT}(X_i \mid \{X_a, X_b\})$ przy założeniu, że wszystkie zmienne wchodzące w skład zapytania mogą przyjmować: (a) dwa stany, (b) cztery stany

Warto zauważyć, że dysproporcja ta naturalnie narasta wraz z dodawaniem kolejnych zmiennych, skutkujących kolejnymi warstwami drzewa DFS. Aby lepiej zobrazować tempo wzrostu łącznej liczby węzłów drzewa DFS w zależności od liczby stanów przyjmowanych przez zmienne zapytania, zostało ono przedstawione na rysunku 5.3. Rozpatrując drzewo utworzone dla zapytania obejmującego dziesięć zmiennych, z których

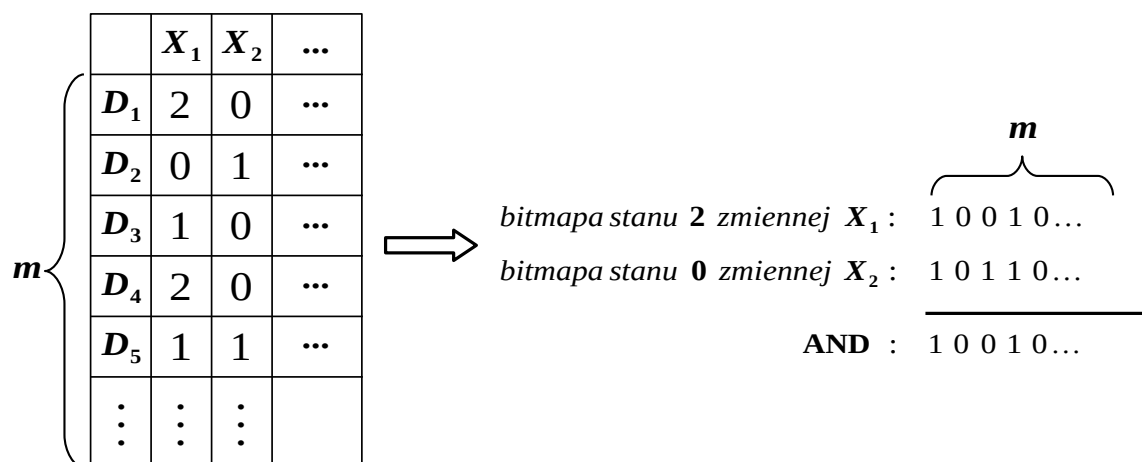


Rysunek 5.3. Łączna liczba węzłów w drzewie DFS w zależności od rozmiaru zapytania oraz liczby stanów przyjmowanych przez jego zmienne

każda może przyjmować dwa stany, łączna liczba węzłów wyniosłaby 1023. Dla porównania, te samo zapytanie zrealizowane dla czterostanowych zmiennych prowadziłoby do powstania drzewa zawierającego blisko 350 tysięcy węzłów. Uwzględniając fakt, że wartości na rysunku zostały przedstawione w skali logarytmicznej, możliwe jest zaobserwowanie, że zapytanie składające się z 30 zmiennych, z których każda może przyjmować cztery stany, generowałoby drzewo zawierające około miliarda razy ($10^{18}/10^9$) więcej węzłów, niż zapytanie o tej samej liczbie zmiennych binarnych. Należy jednak zaznaczyć, że rozważania te dotyczą nieoptymalizowanej wersji algorytmu strategii bitmap. W sekcji 4.2 przedstawiono opracowaną metodę, która pozwala na redukcję liczby odwiedzanych węzłów drzewa DFS. Nie zmniejsza to jednak istotności omawianego parametru w kontekście jego wpływu na wydajność zapytania zliczającego.

Ostatnim z wymienionych czynników decydujących o wydajności zapytania zliczającego jest liczba obserwacji w analizowanym zbiorze danych. Posługując się strategią bitmap jako przykładem ilustrującym znaczenie tego parametru, w pierwszej kolejności należy zauważyć, że liczba obserwacji nie wpływa w bezpośredni sposób na strukturę drzewa DFS. Ma ona jednak kluczowe znaczenia w kontekście czasu trwania podstawowej operacji wykonywanej wielokrotnie w ramach każdego węzła drzewa. Operacją tą jest iloczyn bitowy AND realizowany dla bitmap reprezentujących przyjmowanie poszczególnych stanów przez analizowane zmienne. Jak pokazano na rysunku 5.4, długość każdej bitmapy jest równa liczbie obserwacji w zbiorze danych, co sprawia, że większa liczba obserwacji oznacza dłuższe bitmapy, a tym samym większy koszt obliczeniowy operacji

AND.

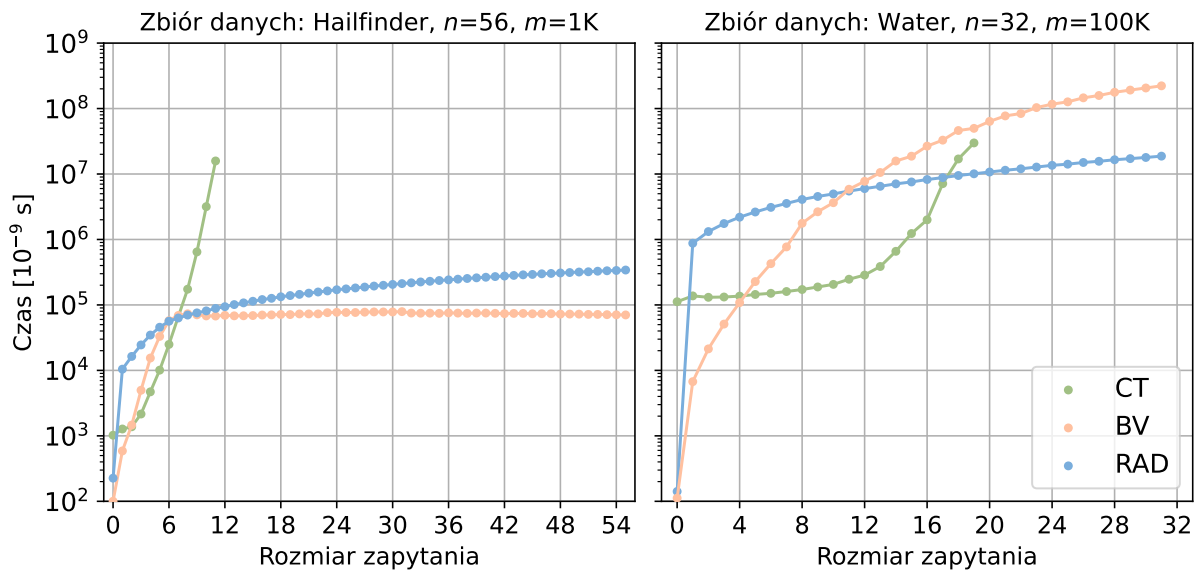


Rysunek 5.4. Operacja AND wykonana dla dwóch przykładowych bitmap bazy danych zawierającej m obserwacji

Po określeniu kluczowych czynników decydujących o teoretycznej wydajności pojedynczego zapytania zliczającego, kolejnym krokiem było wykonanie serii testów wydajnościowych. Testy te miały na celu zbadanie praktycznego wpływu wyszczególnionych czynników na wydajność poszczególnych strategii realizacji zapytań zliczających.

Dwie podstawowe strategie oferowane przez bibliotekę SABNAtk to strategia Bitmap oraz Radix, które zostały przedstawione w sekcjach 3.3.2 oraz 3.3.3. Jak pokazano na licznych przykładach praktycznych w pracy [30], strategie te wykazują znaczną przewagę nad podejściami opartymi o tablice haszujące (hash tables) oraz alternujące drzewa decyzyjne (ADTree), których koncepcje przybliżono w sekcji 3.2.

Wstępne eksperymenty przeprowadzone przez autora niniejszej pracy pokazały jednak, że w niektórych przypadkach strategie Bitmap oraz Radix mogą ustępować prostej strategii opartej o tabelę kontyngencji (contingency table), której istotę omówiono w rozdziale 3. Przypadki te dotyczą sytuacji, w których zbiór danych składa się z niewielkiej liczby zmiennych lub realizowane zapytania obejmują małą liczbę zapytań. Na podstawie tych wyników strategia tabel kontyngencji została oficjalnie włączona do zestawu oferowanych strategii, a dalsze testy wydajnościowe zostały przeprowadzone dla wszystkich trzech podejść: Bitmap (BV), Radix (RAD) oraz Tabel Kontyngencji (CT).



Rysunek 5.5. Porównanie wydajności strategii Tabeli Kontyngencji (CT), Bitmap (BV) oraz Radix (RAD) w zależności od rozmiaru zapytań zliczających oraz zbioru danych

Podsumowanie przeprowadzonych testów zostało przedstawione na rysunku 5.5, który stanowi syntetyczne ujęcie uzyskanych rezultatów, obrazując kluczowe obserwacje płynące z testów. Rysunek ten przedstawia czasy wykonania zapytań zliczających uzyskane dla każdej testowanej strategii na przykładzie dwóch zbiorów danych różniących się liczbą zmiennych oraz obserwacji. W obu przykładach oś odciętych reprezentuje rozmiar testowanych zapytań zliczających a każdy punkt wykresu przedstawia średni czas realizacji stu wygenerowanych zapytań charakteryzujących się tym samym rozmiarem, lecz losowo wybranymi zmiennymi. Obliczenia zostały wykonane na platformie obliczeniowej wyposażonej w dwa 18-rdzeniowe procesory Intel Xeon Gold 6240 oraz 192 GB pamięci RAM.

Porównując oba prezentowane przykłady (rysunek 5.5), kluczową obserwacją jest to, że wydajność poszczególnych strategii różni się i niemożliwe jest jednoznaczne wskazanie najlepszej z nich. Rozpoczynając analizę od lewej części rysunku 5.5, dotyczącej zbioru Hailfinder zawierającego tysiąc obserwacji, można dostrzec dynamiczny wzrost czasu realizacji zapytań strategią CT. Powodem tej sytuacji jest to, że rosnący rozmiar zapytań zliczających powoduje szybkie rozrastanie się wynikowej tabeli kontyngencji, generując coraz większe zapotrzebowanie obliczeniowe oraz pamięciowe. W rezultacie, możliwe okazało się jedynie zrealizowanie zapytań o rozmiarze mniejszym niż 12, ponieważ próba realizacji zapytań o większym rozmiarze kończyła się niepowodzeniem z powodu przekroczenia dostępnej pamięci systemu. Ponadto, w tym zbiorze danych strategia CT okazała się konkurencyjna względem pozostałych podejść jedynie dla zapytań składających się z nie więcej niż sześciu zmiennych. Zapytania o większym rozmiarze były domeną strategii BV, która wraz ze wzrostem rozmiaru zapytań powiększała swoją przewagę nad

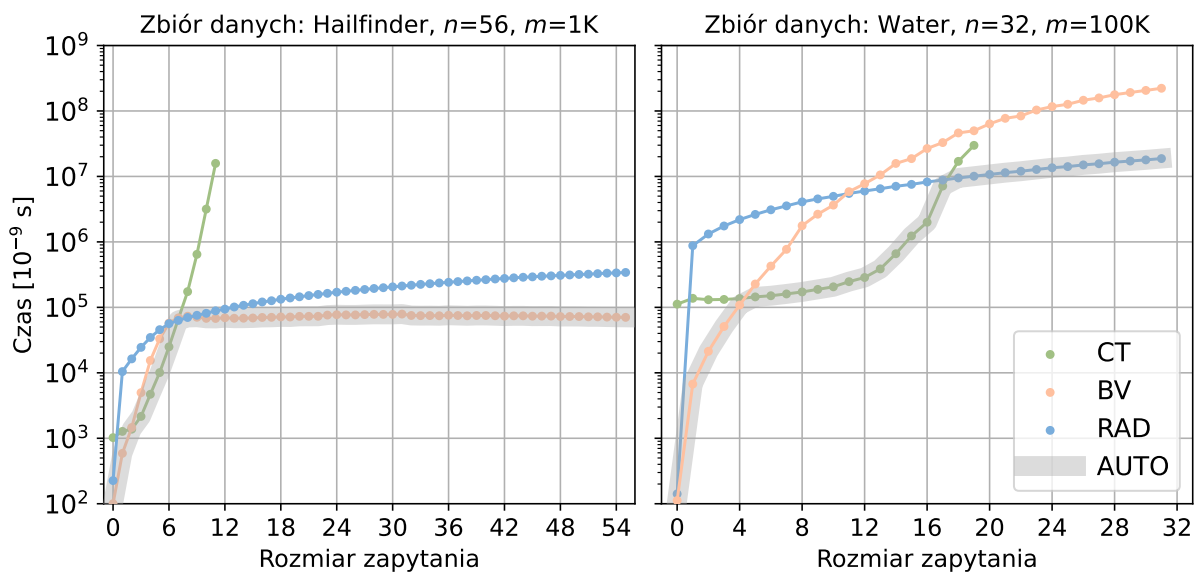
strategią RAD, zachowując niemal stały czas realizacji zapytań. Porównując czasy realizacji dla zapytań o największym rozmiarze, strategia BV okazała się około pięciu razy szybsza od podejścia RAD. Tak dobre wyniki uzyskane dla tej strategii wynikają z dwóch głównych czynników. Pierwszym z nich jest to, że niewielka liczba obserwacji (tysiąc) sprzyja efektywniejszemu wykorzystaniu pamięci podręcznej oraz przyspieszeniu obliczeń poprzez lepsze wykorzystanie zasobów obliczeniowych [30]. Jednocześnie, zwiększa ona prawdopodobieństwo, że wiele możliwych konfiguracji zmiennych nie występuje w zbiorze danych ani razu, co prowadzi do redukcji liczby węzłów w drzewie DFS. Węzły te mogą zostać pominięte dzięki opracowanej optymalizacji SNIJ, opisanej w sekcji 4.4.

Całkiem odmienna sytuacja występuje w przykładzie zamieszczonym po prawej stronie rysunku 5.5, dotyczącym zbioru danych Water, zawierającego sto tysięcy obserwacji. Strategia BV, która w poprzednio omówionym przykładzie była najlepszym podejściem dla zapytań zliczającym o największym rozmiarze, tutaj okazuje się być najlepszą jedynie dla zapytań najmniejszych, obejmujących maksymalnie cztery zmienne. Różnica w zachowaniu tej strategii wynika przede wszystkim ze znacznego wzrostu liczby obserwacji zbioru danych. Większa liczba obserwacji skutkuje dłuższymi bitmapami prowadząc do wydłużenia czasu podstawowej operacji tej strategii, jaką jest iloczyn bitowy AND. Ponadto, spora liczba obserwacji w zbiorze danych zwiększa szansę występowania różnych kombinacji zmiennych, a tym samym ogranicza możliwość redukcji liczby węzłów drzewa DFS. Z tego powodu strategia BV ustępuje podejściu CT, które w tym przypadku okazało się być najlepszą strategią dla zapytań, charakteryzujących się rozmiarem z przedziału [5, 17]. Można jednak zaobserwować, że o ile tempo wzrostu czasu realizacji zapytań składających się z kilku zmiennych strategią CT było niewielkie, to w przypadku zapytań o kilkunastu zmiennych czas realizacji zapytań rośnie znacznie szybciej. Ponadto, strategia CT okazała się możliwa jedynie dla zapytań o rozmiarze mniejszym niż 20, ponieważ podobnie jak poprzednio, próba realizacji zapytań o większym rozmiarze skutkowałą przekroczeniem dostępnych zasobów pamięciowych. Podejściem, które okazało się najlepszym wyborem dla zapytań o największym rozmiarze była strategia RAD, wyróżniająca się liniowym wzrostem czasu realizacji zapytań względem ich rozmiaru. Porównując czasy realizacji dla zapytań o największym rozmiarze, strategia RAD okazała się około dziesięć razy szybsza. Stanowi to całkowite przeciwieństwo przykładu prezentowanego po lewej stronie, gdzie największe zapytania realizowane były około pięciu razy szybciej z wykorzystaniem podejścia BV.

5.2. Koncepcja opracowanego rozwiązania

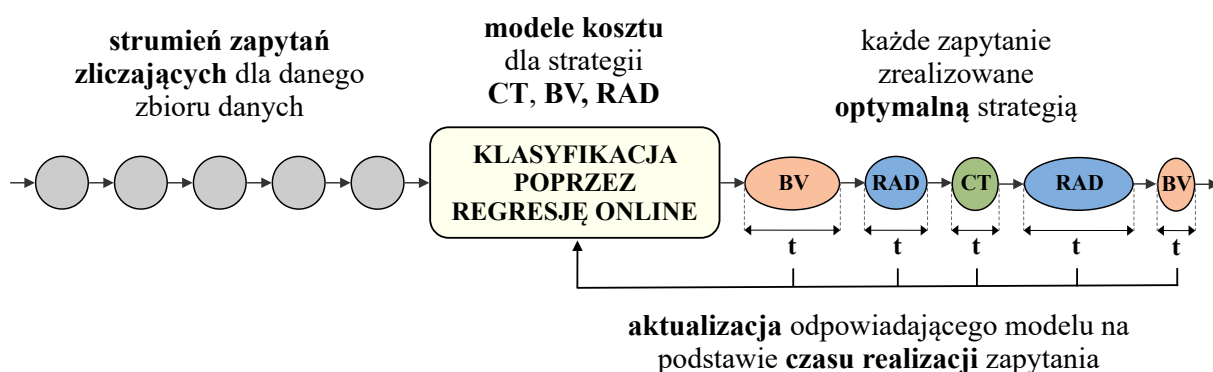
Głównym wnioskiem płynącym z przeprowadzonej analizy wydajności strategii realizacji zapytań zliczających jest to, że wydajność poszczególnych strategii zależy od wielu czynników i żadna z pojedynczych strategii nie dominuje nad pozostałymi. W efekcie niemożliwe jest wskazanie pojedynczej strategii, która okazałaby się najlepszą we wszystkich przypadkach. Jednocześnie, wybór odpowiedniego podejścia może skutkować znaczącym skróceniem czasu realizacji zapytania, ponieważ różnice pomiędzy poszczególnymi podejściami bywają ogromne. Znaczenie tego problemu podkreśla również fakt, że rzeczywiste aplikacje uczenia maszynowego wymagają często przetworzenia nawet milionów zapytań zliczających [30].

Przeprowadzana analiza poskutkowała pojawieniem się idei opracowania metody, pozwalającej na dobór odpowiedniej strategii realizacji do poszczególnych zapytań strumienia. Zadaniem tej metody jest estymacja kosztu realizacji danego zapytania każdą z trzech dostępnych strategii, w celu wybrania tej, która zapewni najkrótszy czas realizacji.



Rysunek 5.6. Potencjalny zysk z opracowania metody automatycznego wyboru strategii (AUTO) do realizacji zapytań zliczających

Rysunek 5.6 przedstawia potencjalne zyski wynikające z opracowania takiego rozwiązania na przykładzie dwóch przypadków omówionych w poprzedniej sekcji. Tak samo jak poprzednio, punkty w trzech różnych kolorach - zielonym, pomarańczowym oraz niebieskim - odpowiadają średnim czasom realizacji zapytań z wykorzystaniem trzech różnych strategii - Tabel Kontyngencji (CT), Bitmap (BV) oraz Radix (RAD). Kolorem szarym zaznaczono wybory jakich powinna dokonywać omawiana metoda.



Rysunek 5.7. Koncepcja opracowanej metody automatycznego wyboru strategii do realizacji zapytań zliczających

Koncepcja opracowanej metody automatycznego wyboru strategii realizacji zapytań zliczających została przedstawiona na rysunku 5.7. Metoda zakłada przetwarzanie strumienia zapytań o nieznanym wcześniej charakterystyce, realizowanych na określonym zbiorze danych. Kluczowym elementem podejścia jest opracowanie i wykorzystanie teoretycznych modeli kosztu dla każdej z rozważanych strategii realizacji zapytań. Decyzja o wyborze strategii jest podejmowana na podstawie klasyfikacji opartej na regresji online, tzn. umożliwiającej trenowanie modeli z jednoczesną obsługą strumienia zapytań. W ramach tej klasyfikacji szacowany jest koszt realizacji zapytania przez modele kosztu odpowiadające poszczególnym strategiom. Wybór optymalnej strategii następuje na podstawie porównania tych szacunków - wybrana zostaje strategia o najniższym estymowanym czasie wykonania.

Ponadto, czas realizacji każdego zapytania wykorzystywany jest do aktualizacji odpowiadającego mu modelu kosztu, co pozwala na adaptację systemu do zmieniających się warunków i charakterystyki zapytań. W efekcie każde zapytanie ze strumienia realizowane jest optymalną strategią, co przekłada się na skrócenie całkowitego czasu przetwarzania.

Szczegóły dotyczące opracowania poszczególnych elementów podejścia zostaną omówione w kolejnych sekcjach tego rozdziału.

5.3. Oszacowanie złożoności strategii realizacji zapytań zliczających

Po analizie wydajności strategii realizacji zapytań zliczających, kolejnym krokiem w kierunku opracowania metody automatycznego wyboru strategii było oszacowanie złożoności pamięciowej i obliczeniowej każdego z trzech indywidualnych podejść. W dalszej

części przyjęto wspólne założenia dotyczące charakterystyki zapytania oraz danych wejściowych:

1. rozważane zapytanie ma postać $\mathbf{COUNT}(X_i \mid Pa(X_i))$ i jest realizowane na zbiorze danych, zawierającym m obserwacji,
2. rozmiar zapytania to liczba zmiennych wchodzących w skład zbioru $Pa(X_i)$, oznaczona jako n ,
3. każda zmienna losowa może przyjmować dowolną liczbę stanów. Liczba możliwych stanów zmiennej X_i oznaczana jest przez r_i , natomiast q odpowiada liczbie możliwych konfiguracji zbioru rodziców $Pa(X_i)$ i jest równa iloczynowi liczby stanów zmiennych należących do tego zbioru.

Przedstawiona analiza uwzględnia zarówno złożoność w najgorszym przypadku (notacja $\mathbb{O}(\cdot)$), jak i złożoność oczekiwaną, zależną od charakterystyki danych wejściowych.

Złożoność strategii CT

Strategia CT oparta jest na bezpośrednim konstruowaniu i wykorzystaniu tabeli kontyngencji. Jest to podejście proste, lecz mało skalowalne, szczególnie dla zapytań zawierających dużą liczbę zmiennych.

Złożoność pamięciowa

Konieczne jest utworzenie pełnej tabeli kontyngencji, która przechowuje liczby wystąpień każdej możliwej kombinacji wartości zmiennej X_i i jej rodziców. Przykład takiej tabeli pokazano w sekcji 3.1. na rysunku 3.2.

Operacja ta wymaga pamięci rzędu

$$\mathbb{O}(r_i \times q), \tag{5.1}$$

co może być nieakceptowalne pamięciowo przy dużych wartościach r_i lub q . Potwierdzają to również eksperymenty przedstawione w sekcji 5.1, w których strategia CT przekraczała możliwości pamięciowe serwera obliczeniowego dla zapytań o kilkunastu zmiennych losowych. Należy dodać, że w przypadku tej strategii CT pesymistyczna odpowiada złożoności oczekiwanej.

Złożoność obliczeniowa

Obliczeniowo strategia obejmuje trzy główne etapy:

- obliczanie indeksów potrzebnych do zlokalizowania odpowiednich komórek tabeli — złożoność $\mathbb{O}(n \times m)$,
- zliczanie wystąpień poprzez iterację po danych — również $\mathbb{O}(n \times m)$,

- finalne przejście po tabeli w celu przekazania niezerowych wyników — $\mathbb{O}(r_i \times q)$.

Całkowita złożoność czasowa wynosi zatem

$$\mathbb{O}(\max(n \times m, r_i \times q)). \quad (5.2)$$

Złożoność strategii RAD

Strategia RAD opiera się na rekurencyjnym dzieleniu zbioru danych na partycje zgodnie z wartościami zmiennych warunkujących. Podejście to cechuje się najlepszą asymptotyczną złożonością wśród analizowanych metod. Zarówno złożoność pesymistyczna, jak i oczekiwana, są w tym przypadku jednakowe.

Złożoność pamięciowa

Wymagane jest jedynie przechowywanie pomocniczej struktury reprezentującej aktualny podział danych na partycje. Całkowita złożoność pamięciowa wynika z konieczności śledzenia przynależności indeksów do poszczególnych partycji i wynosi

$$\mathbb{O}(m). \quad (5.3)$$

Złożoność obliczeniowa

Dla każdej zmiennej $X_j \in Pa(X_i)$ przeprowadzana jest pełna analiza odpowiadającego jej wektora danych, co łącznie prowadzi do złożoności rzędu

$$\mathbb{O}(n \times m). \quad (5.4)$$

W praktyce strategia jest zaimplementowana w efektywny sposób poprzez organizację danych w formacie kolumnowym, a podziały realizowane są za pomocą kolejki typu FIFO (ang. First In, First Out). Kolejka ta przechowuje deskryptory partycji, czyli struktury zawierające informacje o zakresie i położeniu aktualnie przetwarzanych podzbiorów danych.

Złożoność strategii BV

Strategia BV wykorzystuje strukturę drzewa DFS oraz operacje przecięć wektorów bitowych w celu zliczania wystąpień. Charakteryzuje się potencjalnie bardzo dobrą efektywnością, w szczególności w zbiorach o stosunkowo niewielkiej liczbie obserwacji. Istotną cechą tej strategii jest silna zależność od danych wejściowych, mających bezpośredni wpływ na strukturę drzewa DFS. W rezultacie oczekiwana złożoność obliczeniowa różni się od złożoności pesymistycznej.

Złożoność pamięciowa

Dla każdej zmiennej wchodzącej w skład zapytania tworzony jest zbiór bitmap, których liczba odpowiada liczbie możliwych stanów danej zmiennej. Jak wyjaśniono w sekcji 5.1 oraz zilustrowano na rysunku 5.4, bitmapy rejestrują występowanie określonego stanu w poszczególnych obserwacjach, a ich długość jest równa m . Dodatkowo, podczas przeszukiwania drzewa DFS utrzymywany jest stos odpowiadający kolejnym warstwom drzewa, którego głębokość odpowiada liczbie zmiennych objętych zapytaniem. Złożoność pamięciowa strategii BV wynosi zatem

$$\mathbb{O}(n \times m). \quad (5.5)$$

Złożoność obliczeniowa

Pesymistyczna złożoność obliczeniowa tej strategii odpowiada sytuacji, w której algorytm przeszukuje całe drzewo DFS, wykonując w każdym węźle przecięcie wektorów bitowych o długości m . Koszt pojedynczej operacji wynosi $\mathbb{O}(m)$, natomiast liczba węzłów drzewa jest ograniczona przez liczbę wszystkich możliwych konfiguracji zbioru Pa , oznaczaną przez q , lub przez liczbę rzeczywistych obserwacji w danych, tj. m . Złożoność w wariancie pesymistycznym wynosi więc

$$\mathbb{O}(\max(q, m) \times m). \quad (5.6)$$

Należy jednak zaznaczyć, że oczekiwana złożoność obliczeniowa strategii BV różni się od złożoności pesymistycznej, co wynika z silnej zależności struktury drzewa DFS od danych wejściowych oraz z zastosowania opracowanych metod optymalizacyjnych. Metoda *SNIIK* opisana w sekcji 4.1 redukuje liczbę przecięć bitmap wykonywanych w ostatniej warstwie drzewa, natomiast metoda *SNIJ* przedstawiona w sekcji 4.2 pozwala na redukcję liczby węzłów analizowanych podczas przechodzenia przez drzewo DFS.

Oczekiwana złożoność obliczeniowa

Wyznaczanie oczekiwanej złożoności obliczeniowej strategii BV wymaga oszacowania liczby węzłów tworzących drzewo DFS. Oznaczając tę wartość przez X , oczekiwana złożoność strategii BV może zostać przedstawiona jako iloczyn oczekiwanej liczby węzłów w drzewie DFS oraz kosztu operacji przecinania bitmap:

$$\mathbb{E}(X \times m). \quad (5.7)$$

Aby wyznaczyć powyższą złożoność, w pierwszej kolejności niezbędne jest przyjęcie założenia o rozkładzie zmiennych losowych, wchodzących w skład analizowanego zapytania. Zakłada się, że każda ze zmiennych zapytania może przyjmować tę samą liczbę k stanów, oraz że prawdopodobieństwo wystąpienia każdego ze stanów jest jednakowe i wynosi $\frac{1}{k}$.

W następnym kroku, niech W_i będzie zmienną losową opisującą występowanie dowolnego węzła w rozpatrywanym drzewie DFS w następujący sposób:

$$W_i = \begin{cases} 1 & \text{gdy węzeł obecny w drzewie,} \\ 0 & \text{gdy węzeł usunięty z drzewa.} \end{cases} \quad (5.8)$$

Oczekiwana liczba węzłów X może zostać wtedy wyrażona jako wartość oczekiwana sumy wszystkich węzłów W_i drzewa DFS:

$$X = E \left(\sum_i W_i \right). \quad (5.9)$$

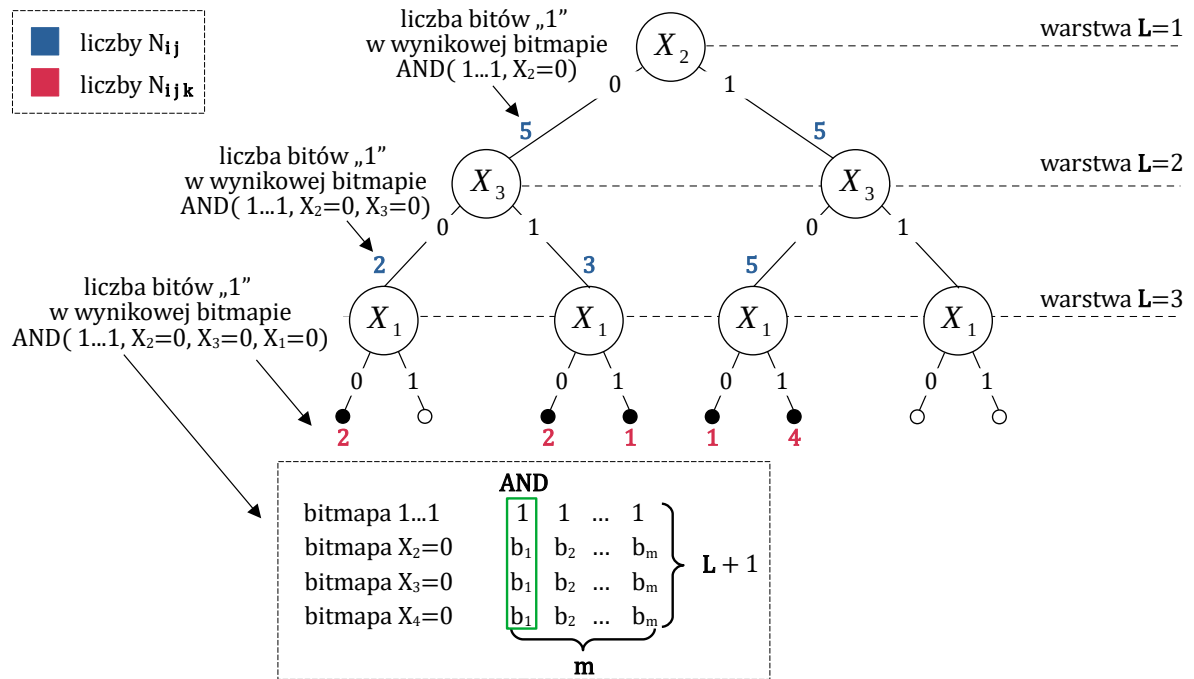
Z własności liniowości wartości oczekiwanej wynika, że wartość oczekiwana sumy zmiennych losowych jest równa sumie ich wartości oczekiwanych:

$$X = \sum_i E(W_i). \quad (5.10)$$

Wartość oczekiwana zmiennej W_i , zgodnie z definicją, równa jest sumie iloczynów wartości, które zmienna może przyjąć, oraz odpowiadających im prawdopodobieństw:

$$\begin{aligned} E(W_i) &= 0 \cdot P(W_i = 0) + 1 \cdot P(W_i = 1) \\ E(W_i) &= P(W_i = 1). \end{aligned} \quad (5.11)$$

Oznacza to, że zadanie wymaga wyznaczenia prawdopodobieństwa zdarzenia polegającego na tym, że dany węzeł nie zostanie usunięty z drzewa DFS. Należy zauważyć, że prawdopodobieństwo te zależne jest od warstwy w jakiej dany węzeł występuje w drzewie DFS. Uwzględniając, że usunięcie węzła z drzewa powoduje usunięcie poddrzewa zakorzenionego w tym węźle, węzły głębszych warstw będą cechować się większym prawdopodobieństwem usunięcia.



Rysunek 5.8. Drzewo DFS powstałe dla przykładowego zapytania zliczającego

Rysunek 5.8 przedstawia drzewo DFS powstałe podczas realizacji przykładowego zapytania zliczającego. Zgodnie z algorytmem strategii BV, omówionym szerzej w sekcji 3.3.2, analizując poszczególne stany węzła znajdującego się w warstwie L , każdorazowo dokonuje się przecięcia $L + 1$ bitmap. Każda z bitmap składa się z m bitów, co odpowiada liczbie obserwacji w rozpatrywanym zbiorze danych. Przecięcie bitmap polega na m -krotnym wykonaniu operacji AND dla poszczególnych $L + 1$ bitów.

Dany węzeł nie zostanie usunięty, jeżeli przynajmniej jedna z m wykonywanych operacji AND zwróci wynik równy 1. Rozpatrując pojedynczą operację AND, taka sytuacja ma miejsce tylko wtedy, gdy w każdym z $L + 1$ analizowanych bitów wystąpi 1. Oznaczając prawdopodobieństwo tego zdarzenia przez p oraz uwzględniając przyjęte założenie o rozkładzie zmiennych losowych, może być ono wyrażone za pomocą następującego wzoru:

$$p = \frac{1}{k^L}. \quad (5.12)$$

W powyższym wzorze potęga mianownika wynosi L , a nie $L + 1$, ponieważ pierwszy z analizowanych bitów zawsze pochodzi z bitmapy składającej się z samych jedynek, a więc jego wartość jest z góry określona.

Następnie, prawdopodobieństwo nieusunięcia węzła może zostać obliczone za pomocą schematu Bernoulliego [36]. Schemat ten opisuje ciąg niezależnych doświadczeń losowych, z których każde ma tylko dwa możliwe wyniki – sukces (z prawdopodobieństwem p) lub porażkę (z prawdopodobieństwem $1 - p$). W rozważanym przypadku, sukcesem będzie sytuacja polegająca na wystąpieniu samych jedynek wśród L analizowanych bi-

tów. Porażką będzie natomiast sytuacja przeciwna, czyli wystąpienie przynajmniej jednego zera.

Stosując schemat Bernoulliego, prawdopodobieństwo nieusunięcia węzła może zostać przedstawione jako zdarzenie polegające na otrzymaniu przynajmniej jednego sukcesu w m próbach:

$$P(W_i = 1) = P(S \geq 1). \quad (5.13)$$

Prawdopodobieństwo takiego zdarzenia można z kolei wyrazić za pomocą prawdopodobieństwa zdarzenia przeciwnego, czyli sytuacji niewystąpienia żadnego sukcesu podczas m prób:

$$\begin{aligned} P(S \geq 1) &= 1 - P(S = 0) \\ P(S \geq 1) &= 1 - (1 - p)^m. \end{aligned} \quad (5.14)$$

Uwzględniając wzór 5.12, otrzymuje się:

$$P(S \geq 1) = 1 - \left(1 - \frac{1}{k^L}\right)^m. \quad (5.15)$$

Zgodnie z równaniami 5.11 oraz 5.13, obliczona wartość odpowiada poszukiwanej wartości oczekiwanej zmiennej losowej W_i :

$$E(W_i) = P(W_i = 1) = P(S \geq 1) = 1 - \left(1 - \frac{1}{k^L}\right)^m. \quad (5.16)$$

Ostatnim etapem jest wyznaczenie liczby węzłów, jakie drzewo DFS zawierałoby w ogólnym przypadku, gdyby nie usunięto żadnego z węzłów. Biorąc pod uwagę, że na mocy przyjętego założenia, każda ze zmiennych może przyjmować k stanów, każdy węzeł niebędący liściem ma k potomków. Oznacza to, że liczba węzłów w warstwie L w ogólnym przypadku wynosi k^{L-1} . Przykład potwierdzający tę prawidłowość widoczny jest na rysunku 5.8. Każdy z węzłów prezentowanego tam drzewa ma $k = 2$ potomków. W rezultacie pierwsza warstwa drzewa zawiera jeden węzeł (2^0), druga dwa węzły (2^1), a trzecia cztery węzły (2^2). Całkowita liczba węzłów w drzewie DFS jest więc sumą liczby węzłów z kolejnych warstw:

$$\sum_{L=1}^{n+1} k^{L-1} = \sum_{L=0}^n k^L \quad (5.17)$$

Uwzględniając powyższe wyliczenia w równaniu 5.10, oczekiwana liczba węzłów w drzewie DFS może być wyrażona w następujący sposób:

$$X = \sum_i E(W_i) = \sum_{L=0}^n k^L \cdot \left(1 - \left(1 - \frac{1}{k^L}\right)^m\right). \quad (5.18)$$

Wobec tego, na bazie równania 5.7, oczekiwana złożoność strategii BV jest równa:

$$\mathbb{E}(X \times m) = \mathbb{E} \left(m \times \sum_{L=0}^n k^L \cdot \left(1 - \left(1 - \frac{1}{k^L} \right)^m \right) \right). \quad (5.19)$$

5.4. Teoretyczne modele kosztu realizacji zapytania zliczającego

Kolejnym etapem po oszacowaniu złożoności poszczególnych strategii realizacji zapytań zliczających było skonstruowanie modeli regresji dla każdej z nich. Opracowane modele mają postać funkcji wielu zmiennych i opisują koszt realizacji jednego zapytania, rozumiany jako czas jego wykonania. W dalszym ciągu rozpatrywane jest zapytanie **COUNT**($X_i \mid Pa(X_i)$) o rozmiarze $n = |Pa(X_i)|$, realizowane na zbiorze danych zawierającym m obserwacji. Uwzględnione zostaje również założenie wprowadzone podczas szacowania oczekiwanej złożoności obliczeniowej strategii BV, zgodnie z którym każda ze zmiennych wchodzących w skład zapytania przyjmuje k równo prawdopodobnych stanów.

Model kosztu realizacji zapytania zliczającego strategią CT

Na podstawie złożoności określonych wzorami 5.1 i 5.2, koszt realizacji zapytania zliczającego strategią CT może zostać przedstawiony za pomocą następującej funkcji regresji:

$$CT(n, m, k, \alpha) = \alpha_1 \cdot n \cdot m + \alpha_2 \cdot k^{\alpha_3 \cdot n} + \alpha_4. \quad (5.20)$$

Funkcja ta zależy od trzech zmiennych niezależnych: n , m , oraz k . Wektor α zawiera współczynniki regresji, czyli liczby rzeczywiste, które powinny zostać dopasowane podczas procesu uczenia modelu.

Model kosztu realizacji zapytania zliczającego strategią RAD

W przypadku strategii RAD, bazując na złożonościach określonych wzorami 5.3 i 5.4, koszt realizacji zapytania zliczającego modeluje następująca funkcja regresji:

$$RAD(n, m, \beta) = \beta_1 \cdot n \cdot m + \beta_2. \quad (5.21)$$

W przeciwieństwie do funkcji modelującej koszt strategii CT, ta funkcja zależy jedynie od dwóch zmiennych niezależnych: n oraz m . Podobnie jak poprzednio, wektor β zawiera współczynniki regresji, czyli liczby rzeczywiste, które powinny zostać dopasowane podczas procesu uczenia modelu.

Model kosztu realizacji zapytania zliczającego strategią BV

Zgodnie ze wzorem 5.19, oczekiwana złożoność obliczeniowa strategii BV wyraża się jako:

$$\mathbb{E} \left(m \times \sum_{L=0}^n k^L \cdot \left(1 - \left(1 - \frac{1}{k^L} \right)^m \right) \right). \quad (5.22)$$

Analizując składniki występujące we wzorze sumy można zauważyć, że wykładnicze zachowanie czynnika k^L , odpowiadającego liczbie węzłów w danej warstwie, jest ograniczane przez wyrażenie $1 - \left(1 - \frac{1}{k^L} \right)^m$, opisujące prawdopodobieństwo nieusunięcia węzła z warstwy L (wzór 5.16). Wyrażenie te może przyjmować wartości z przedziału $[0, 1]$ i stanowi istotny czynnik hamujący wzrost oczekiwanej złożoności. Rysunek 5.9 przedstawia zależność wartości omawianego wyrażenia od liczby warstw L dla dwóch różnych wartości parametru m . Dla każdego przypadku rozważono cztery krzywe odpowiadające różnym wartościom parametru k . Jak można zaobserwować, zwiększenie liczby możliwych stanów k prowadzi do obniżenia prawdopodobieństwa nieusunięcia węzła. Analogiczny efekt obserwuje się podczas zwiększenia liczby obserwacji m .

Jak wynika z rysunku 5.9, prawdopodobieństwo nieusunięcia węzłów z początkowych warstw jest bliskie 1. Wobec tego, w tych warstwach oczekiwana złożoność strategii BV może zostać ograniczona z góry w następujący sposób:

$$m \cdot \sum_{L=0}^n k^L \cdot \left(1 - \left(1 - \frac{1}{k^L} \right)^m \right) < m \cdot \sum_{L=0}^n k^L. \quad (5.23)$$

Wyrażenie po prawej stronie nierówności 5.23 stanowi sumę $n + 1$ początkowych wyrazów ciągu geometrycznego o ilorazie k^L . Możliwe jest więc jawne obliczenie tej sumy:

$$m \cdot \sum_{L=0}^n k^L \cdot \left(1 - \left(1 - \frac{1}{k^L} \right)^m \right) < m \cdot \sum_{L=0}^n k^L = m \cdot \frac{k^{n+1} - 1}{k - 1} \quad (5.24)$$

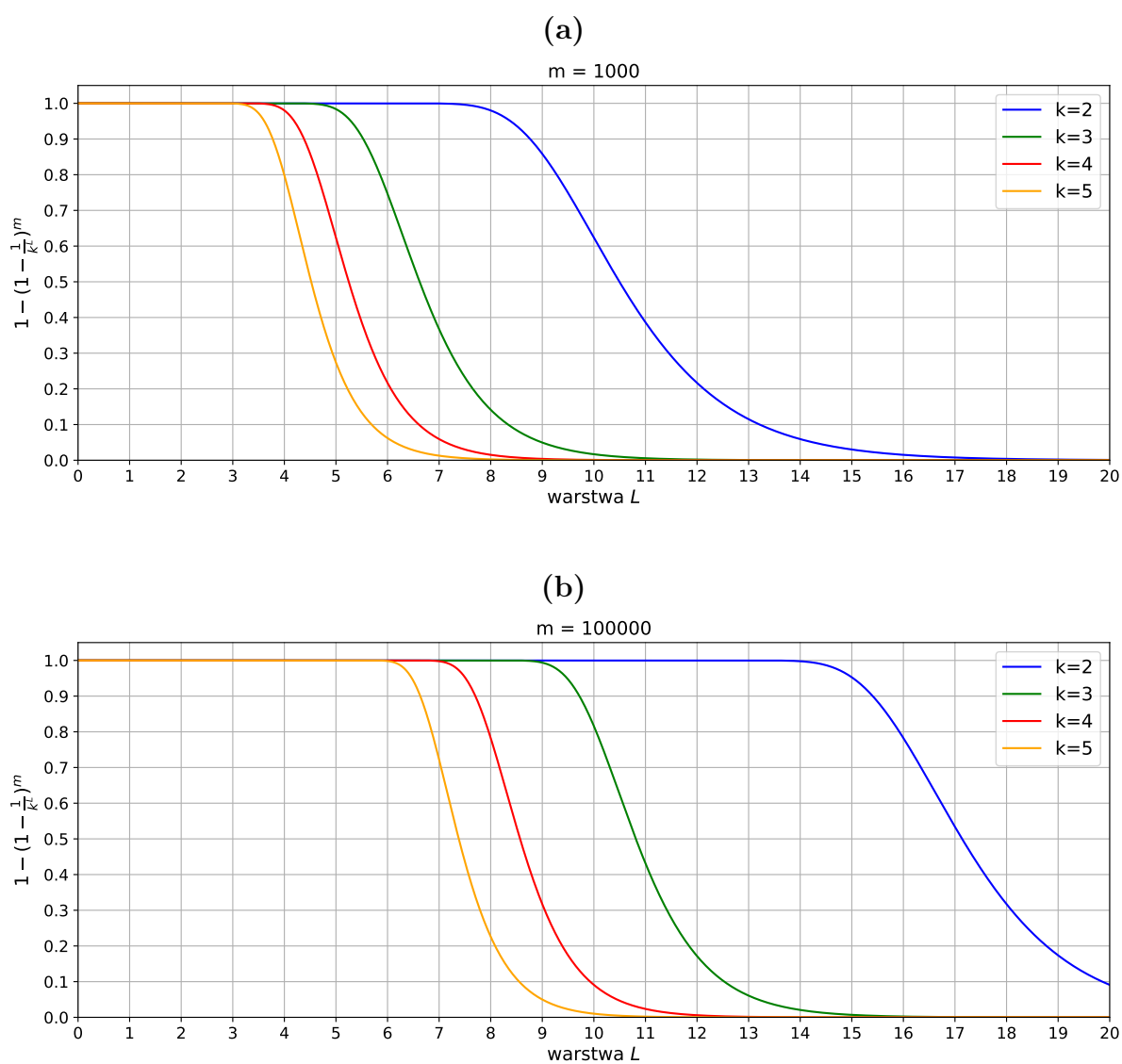
Oznacza to, że koszt strategii BV dla zapytań zliczających o niewielkim rozmiarze n można opisać następującą funkcją regresji:

$$BV(n, m, k, \gamma) = \gamma_1 \cdot m \cdot \frac{k^{\gamma_2 \cdot n + 1} - 1}{k - 1} \quad \text{dla } n \leq n_0, \quad (5.25)$$

gdzie γ to wektor współczynników regresji a n_0 to liczba rzeczywista, której sposób wyznaczenia zostanie przedstawiony w późniejszym etapie.

Rozpatrując przypadek większych zapytań ($n > n_0$), należy przeanalizować granicę wyrazu sumy opisującej oczekiwaną złożoność strategii BV, przy L dążącym do nieskończoności:

$$\lim_{L \rightarrow \infty} k^L \cdot \left(1 - \left(1 - \frac{1}{k^L} \right)^m \right). \quad (5.26)$$



Rysunek 5.9. Prawdopodobieństwo nieusunięcia węzła warstwy L dla różnych wartości parametru k oraz (a) $m = 1000$ i (b) $m = 100000$

Granica pierwszego czynnika powyższego wyrażenia wynosi nieskończoność:

$$\lim_{L \rightarrow \infty} k^L = \infty. \quad (5.27)$$

Analizując drugi czynnik, można dostrzec, że dla $L \rightarrow \infty$, jego wewnętrzny składnik dąży do 1:

$$\lim_{L \rightarrow \infty} \left(1 - \frac{1}{k^L}\right)^m = 1. \quad (5.28)$$

W związku z tym całe wyrażenie dąży do 0:

$$\lim_{L \rightarrow \infty} \left(1 - \left(1 - \frac{1}{k^L}\right)^m\right) = 0. \quad (5.29)$$

Oznacza to, że granica wyrażenia z równania 5.26 jest typu nieoznaczonego $\infty \cdot 0$, a więc w celu jej obliczenia w pierwszej kolejności należy dokonać pewnych przekształceń. Wyrażenie pod granicą wzoru 5.26 może zostać przekształcone do postaci ilorazu:

$$k^L \cdot \left(1 - \left(1 - \frac{1}{k^L}\right)^m\right) = \frac{1 - \left(1 - \frac{1}{k^L}\right)^m}{\frac{1}{k^L}}. \quad (5.30)$$

Stosując podstawienie

$$x = \frac{1}{k^L}, \quad (5.31)$$

otrzymuje się $x \rightarrow 0^+$ przy $L \rightarrow \infty$. Wówczas granica wyrażenia danego wzorem 5.30 może zostać zapisana jako:

$$\lim_{x \rightarrow 0^+} \frac{1 - (1 - x)^m}{x}. \quad (5.32)$$

W powyższym wyrażeniu, przy x dążącym do 0, licznik oraz mianownik również dążą do zera. Oznacza to, że granica jest typu $\frac{0}{0}$, co uzasadnia zastosowanie reguły de l'Hospitala, która głosi, że jeśli granica ilorazu dwóch funkcji ma postać $\frac{0}{0}$ (lub $\frac{\infty}{\infty}$), to można ją wyznaczyć jako granicę ilorazu ich pochodnych (o ile ta istnieje). Stosując tę zasadę otrzymuje się:

$$\lim_{x \rightarrow 0^+} \frac{1 - (1 - x)^m}{x} = \lim_{x \rightarrow 0^+} \frac{d}{dx} [1 - (1 - x)^m] \Big/ \frac{d}{dx} [x]. \quad (5.33)$$

Obliczenie pochodnych prowadzi do:

$$\lim_{x \rightarrow 0^+} \frac{m(1 - x)^{m-1}}{1} = m. \quad (5.34)$$

Uwzględniając dokonane podstawienie dane wzorem 5.31, ostatecznie otrzymano więc:

$$\lim_{L \rightarrow \infty} k^L \cdot \left(1 - \left(1 - \frac{1}{k^L}\right)^m\right) = m. \quad (5.35)$$

Oznacza to, że od pewnego momentu, kolejne wyrazy sumy:

$$\sum_{L=0}^n k^L \cdot \left(1 - \left(1 - \frac{1}{k^L}\right)^m\right) \quad (5.36)$$

dążą do m , a więc ich zachowanie jest liniowe względem rozmiaru zapytania, danego zmienną n . Wobec tego, uwzględniając równanie 5.22, koszt strategii BV dla zapytań zliczających o rozmiarze $n > n_0$ można opisać następującą funkcją regresji:

$$BV(n, m, k, \gamma) = BV(n_0, m, k, \gamma_1, \gamma_2) + \gamma_3 \cdot (n - n_0) \cdot m + \gamma_4 \quad \text{dla } n > n_0, \quad (5.37)$$

gdzie γ to wektor współczynników regresji a n_0 to liczba rzeczywista dana wzorem:

$$n_0 = \log_k(m). \quad (5.38)$$

Uwzględniając model regresji opracowany dla przypadku $n \leq n_0$ (równanie 5.25), funkcja regresji modelująca koszt strategii BV określona jest wzorem:

$$BV(n, m, k, \gamma) = \begin{cases} \gamma_1 \cdot m \cdot \frac{(k^{\gamma_2 \cdot n + 1} - 1)}{(k - 1)} & n \leq n_0 \\ BV(n_0, m, k, \gamma_1, \gamma_2) + \gamma_3 \cdot (n - n_0) \cdot m + \gamma_4 & n > n_0. \end{cases} \quad (5.39)$$

5.5. Eksperymentalna weryfikacja opracowanych modeli kosztu realizacji zapytania zliczającego

Zgodnie z iteracyjnym podejściem badawczym, opracowane modele kosztu realizacji zapytania zliczającego dla trzech strategii przetwarzania poddano wstępnej weryfikacji eksperymentalnej. Celem przeprowadzonych testów było określenie, czy proponowane podejście automatycznego wyboru strategii ma potencjał do poprawy efektywności realizacji strumienia zapytań zliczających.

W pierwszej kolejności wybrano zestaw kilku zbiorów danych różniących się liczbą zmiennych oraz obserwacji, wraz z odpowiadającymi im strumieniami tysiąca zapytań losowych. Szczegóły dotyczące zbiorów danych przedstawiono szerzej w sekcji 3.4.2. Następnie każdy ze strumieni przetworzono niezależnie każdą z trzech strategii (CT, RAD, BV), uzyskując rzeczywiste czasy realizacji zapytań dla każdego podejścia.

Na podstawie zebranych czasów realizacji przeprowadzono symulację działania prototypu metody automatycznego wyboru strategii. W tym celu opracowane modele regresji trzech strategii zaimplementowano w języku Python, z wykorzystaniem funkcji `curve_fit` z biblioteki `scipy`. Każdemu z modeli przypisano odpowiadającą mu funkcję kosztu, zgodnie z wcześniej wyprowadzonymi zależnościami określonymi równaniami

(5.20), (5.21) oraz (5.39).

Współczynniki każdego z modeli regresji zostały dopasowane w procesie uczenia na podstawie zebranych rzeczywistych czasów realizacji zapytań, co pozwoliło na odtworzenie przewidywanego zachowania każdej ze strategii. Następnie przeprowadzono symulację przetworzenia strumienia zapytań, zakładając, że dla każdego z nich wybierana jest ta strategia, dla której oszacowany koszt (czas realizacji) był najniższy. Warto zaznaczyć, że analiza miała charakter retrospektywny – zapytania nie były ponownie realizowane, a jedynie rekonstruowano decyzje, jakie zostałyby przez niego podjęte na podstawie wytrenowanych modeli regresyjnych.

Uzyskane wyniki porównano z podejściem losowym, polegającym na przypisywaniu każdemu zapytaniu ze strumienia jednej z trzech strategii w sposób losowy. Dla obu podejść wyznaczono łączny czas realizacji całego strumienia zapytań, a następnie obliczono przyspieszenie jako iloraz czasów wykonania w scenariuszu losowym i przy zastosowaniu prototypu metody automatycznego wyboru strategii.

Zestawienie wyników przedstawiono w tabeli 5.1. Jak pokazano w tabeli, opracowany prototyp metody automatycznego wyboru strategii umożliwił znaczną redukcję czasu realizacji zapytań, osiągając nawet kilkusetkrotne przyspieszenie w zależności od zbioru danych wejściowych. Wyniki te potwierdziły potencjał opracowywanego podejścia i stanowiły uzasadnienie dla jego dalszego rozwijania.

Zbiór danych (n)	Przyspieszenie		
	$m = 1K$	$m = 10K$	$m = 100K$
Child (20)	778.2	88.0	22.6
Insurance (27)	533.3	63.2	16.6
Mildew (35)	282.6	19.2	26.7
Alarm (37)	311.7	18.2	10.7
Barley (48)	145.8	15.2	17.0

Tabela 5.1: Przyspieszenie uzyskane dla prototypu metody automatycznego wyboru strategii do realizacji zapytań zliczających w porównaniu do losowego wyboru strategii

5.6. Adaptacja opracowanych modeli kosztu realizacji zapytania zliczającego

Wyniki eksperymentalne przedstawione w poprzedniej sekcji nie uwzględniają narzutu obliczeniowego, związanego z działaniem metody automatycznego wyboru strategii.

Należy zauważyć, że zarówno aktualizacja modeli regresji po każdym zapytaniu, jak i samo podejmowanie decyzji dotyczącej wyboru strategii przetwarzania, będą generowały dodatkowe koszty czasowe. W przypadku najkrótszych zapytań, których czas re-

alizacji mieści się w zakresie mikrosekund, nawet niewielki narzut może zniwelować potencjalne korzyści wynikające z poprawnego wyboru strategii. Z tego względu konieczne było dostosowanie opracowanych wcześniej modeli kosztu, tak aby mogły być stosowane w praktyce bez istotnego wpływu na ogólną wydajność systemu.

W szczególności przeanalizowano możliwość uproszczenia modeli określonych równaniami 5.20, 5.21 oraz 5.39. W pierwotnej postaci modele te wykorzystywały trzy zmienne niezależne: liczbę zmiennych w zapytaniu (n), liczbę obserwacji w zbiorze danych (m) oraz liczbę możliwych stanów przyjmowanych przez zmienne zapytania (k). W celu ograniczenia kosztów obliczeniowych podjęto decyzję o redukcji liczby zmiennych niezależnych i uwzględnieniu wyłącznie jednej z nich – liczby zmiennych w zapytaniu.

Redukcja ta była możliwa z uwagi na charakterystykę pozostałych dwóch parametrów. Liczba obserwacji w praktyce pozostaje stała dla danego zbioru danych i może być traktowana jako znana wartość konfiguracyjna, ustalana przed rozpoczęciem przetwarzania zapytań. W analogiczny sposób można wyeliminować ze zmiennych niezależnych liczbę możliwych stanów przyjmowanych przez zmienne. Wartość ta może zostać oszacowana przed przetwarzaniem zapytań, jako średnia liczba przyjmowanych stanów przypadająca na zmienną losową rozpatrywanego zbioru danych.

Wdrożone modele kosztu realizacji zapytania zliczającego strategiami CT, RAD, BV stanowią kombinację funkcji liniowych i wykładniczych. Określone są one następującymi równaniami:

$$C_{CT}(x, \alpha) = \begin{cases} \alpha_1 \cdot x + \alpha_2 & x \leq x_0 \\ \alpha_3 \cdot k^{\alpha_4 \cdot x} & x > x_0, \end{cases} \quad (5.40)$$

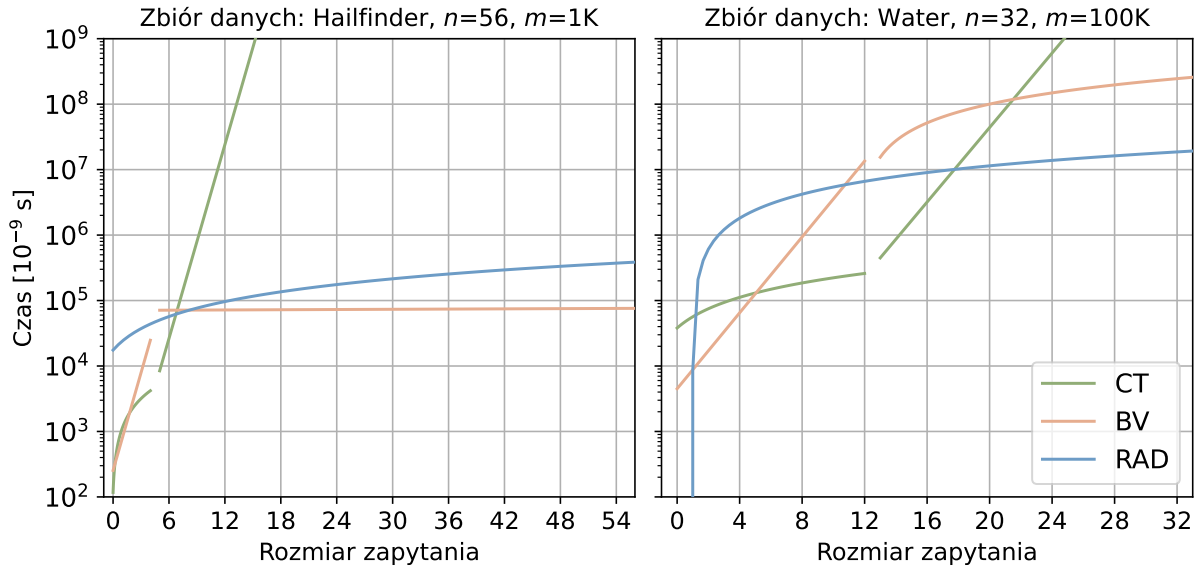
$$C_{RAD}(x, \beta) = \beta_1 \cdot x + \beta_2, \quad (5.41)$$

$$C_{BV}(x, \gamma) = \begin{cases} \gamma_1 \cdot k^{\gamma_2 \cdot x} & x \leq x_0 \\ \gamma_3 \cdot x + \gamma_4 & x > x_0, \end{cases} \quad (5.42)$$

W powyższych modelach x oznacza rozmiar zapytania zliczającego, czyli liczbę zmiennych wchodzących w jego skład. Wartość progowa $x_0 = \log_k(m)$ wyznacza punkt przełączenia pomiędzy dwiema postaciami funkcji kosztu dla strategii CT oraz BV. Stała k reprezentuje średnią liczbę stanów przyjmowanych przez zmienne rozpatrywanego zbioru danych D . Jego liczba obserwacji m nie jest bezpośrednio ujęta we wzorach, ponieważ jej wartość jest stała i będzie odzwierciedlona we współczynnikach regresji α , β oraz γ , których wartości wyznaczane będą w procesie dopasowania modeli do danych empirycznych.

Zaletą przedstawionych modeli jest ich uproszczona struktura, co przekłada się na niż-

szy koszt obliczeniowy. Ponadto, modele wykładnicze mogą zostać przekształcone do postaci liniowej przy użyciu odpowiednich transformacji. W rezultacie umożliwi to zastosowanie szybkich algorytmów inkrementacyjnych do bieżącej aktualizacji statystyk niezbędnych w procesie regresji, takich jak średnia, wariancja oraz kowariancja. Zagadnienie to zostanie szczegółowo omówione w kolejnej sekcji.



Rysunek 5.10. Przykład wykorzystania opracowanych modeli regresji do oszacowania kosztu realizacji zapytania zliczającego na podstawie jego rozmiaru

Rysunek 5.10 przedstawia przykładowe dopasowanie opracowanych modeli regresji dla dwóch przypadków testowych, omówionych wcześniej w analizie wydajności strategii (sekcja 5.1). Zestawienie z rysunkiem 5.5, gdzie pokazano rzeczywiste czasy realizacji zapytań, wskazuje, że finalne modele regresji dobrze odwzorowują rzeczywisty koszt wykonania zapytań zliczających.

Szczegółowa weryfikacja opracowanego podejścia zostanie przedstawiona w końcowych częściach niniejszego rozdziału. W pierwszej kolejności zostaną jednak omówione szczegóły algorytmów aktualizacji modeli oraz strategii przetwarzania strumienia zapytań.

5.7. Algorytm aktualizacji modeli

Zgodnie z koncepcją metody automatycznego wyboru strategii, przedstawioną w sekcji 5.2, po realizacji każdego zapytania ze strumienia dokonywana jest aktualizacja modelu kosztu strategii, która została użyta do jego wykonania. Aktualizacja ta polega na oszacowaniu nowych wartości współczynników regresji, których finalna postać określona jest wzorami 5.40–5.42. Opracowane modele kosztu przedstawione w tych równaniach mają postać liniowo-wykładniczą. Warto jednak zauważyć, że stosując odpowiednie przekształcenie, modele wykładnicze mogą zostać sprowadzone do postaci liniowej.

Przykładowo, rozpatrując część wykładniczą modelu dla strategii CT danego równaniem 5.40, obustronne zastosowanie logarytmu przy podstawie k prowadzi do następującej zależności:

$$\log_k(C_{CT}(x, \alpha)) = \log_k(\alpha_3 \cdot k^{\alpha_4 \cdot x}) \quad (5.43)$$

Korzystając z własności logarytmów, mówiącej o tym, że logarytm iloczynu jest równy sumie logarytmów jego czynników, otrzymuje się:

$$\log_k(C_{CT}(x, \alpha)) = \log_k(\alpha_3) + \log_k(k^{\alpha_4 \cdot x}) \quad (5.44)$$

Następnie, zgodnie z własnością logarytmów, wykładnik potęgi wewnątrz logarytmu może zostać przeniesiony przed logarytm:

$$\log_k(C_{CT}(x, \alpha)) = \log_k(\alpha_3) + \alpha_4 \cdot x \cdot \log_k k \quad (5.45)$$

Uwzględnienie, że $\log_k k = 1$, prowadzi bezpośrednio do zależności liniowej:

$$\log_k(C_{CT}(x, \alpha)) = \alpha_4 \cdot x + \log_k(\alpha_3) \quad (5.46)$$

Uzyskana zależność 5.46 pozwala na wykorzystanie metod regresji liniowej dla wszystkich modeli kosztu realizacji zapytania zliczającego, danych równaniami 5.40–5.42.

Regresja liniowa to model statystyczny opisujący zależność między zmienną objaśnianą x a zmienną objaśnianą y za pomocą funkcji liniowej postaci:

$$y = a x + b, \quad (5.47)$$

gdzie a oznacza współczynnik kierunkowy, natomiast b to wyraz wolny.

Współczynniki a i b wyznaczone są metodą najmniejszych kwadratów, polegającą na minimalizacji sumy kwadratów różnic między wartościami obserwowanymi y_i a wartościami przewidywanymi przez model:

$$S(a, b) = \sum_{i=1}^d (y_i - (a x_i + b))^2, \quad (5.48)$$

gdzie d to liczba wartości obserwowanych.

Warunki konieczne istnienia minimum lokalnego uzyskuje się poprzez wyzerowanie pochodnych cząstkowych funkcji kosztu względem a i b :

$$\frac{\partial S}{\partial a} = -2 \sum_{i=1}^d x_i (y_i - a x_i - b) = 0, \quad (5.49)$$

$$\frac{\partial S}{\partial b} = -2 \sum_{i=1}^d (y_i - a x_i - b) = 0. \quad (5.50)$$

Rozwiązując powyższy układ równań normalnych, otrzymuje się wzory na współczynniki regresji w postaci:

$$a = \frac{\text{cov}_{xy}}{\text{var}_x}, \quad (5.51)$$

$$b = \bar{y} - a \cdot \bar{x}, \quad (5.52)$$

gdzie $\bar{x}$ i $\bar{y}$ oznaczają średnie arytmetyczne odpowiednio zmiennych x i y , a cov_{xy} i var_x oznaczają kowariancję i wariancję.

Należy jednak zauważyć, że każdorazowa aktualizacja modelu na podstawie standardowych wzorów na średnią, wariancję oraz kowariancję wymaga przeliczenia wszystkich dotychczasowych wartości. W klasycznym podejściu średnia, wariancja oraz kowariancja wyznaczane są zgodnie z poniższymi wzorami:

$$\bar{x} = \frac{1}{d} \sum_{i=1}^d x_i, \quad \bar{y} = \frac{1}{d} \sum_{i=1}^d y_i, \quad (5.53)$$

$$\text{var}_x = \frac{1}{d} \sum_{i=1}^d (x_i - \bar{x})^2, \quad (5.54)$$

$$\text{cov}_{xy} = \frac{1}{d} \sum_{i=1}^d (x_i - \bar{x})(y_i - \bar{y}). \quad (5.55)$$

W konsekwencji liczba operacji potrzebnych do obliczenia współczynników rośnie liniowo wraz z liczbą zrealizowanych zapytań. Co więcej, konieczne jest przechowywanie pełnej historii obserwacji, co w przypadku długotrwałego działania systemu może prowadzić do istotnych kosztów pamięciowych.

Aby wyeliminować te ograniczenia, zastosowano algorytmy inkrementacyjne [47] do obliczania średnich, wariancji oraz kowariancji. Pozwalają one na bieżącą aktualizację statystyk regresji na podstawie pojedynczych nowych obserwacji, bez potrzeby przechowywania całego zbioru danych oraz bez narastającego kosztu obliczeniowego. Mechanizm ten stanowi podstawę efektywnej aktualizacji modeli kosztu dla każdej strategii realizacji zapytań.

Algorytm 6 UPDATEMODEL(x, y)

```

1:  $d+ = d + 1$ 

2:  $\Delta_x = x - \bar{x}$ 
3:  $\Delta_y = y - \bar{y}$ 

4:  $\bar{x} += \frac{\Delta_x}{d}$ 
5:  $\bar{y} += \frac{\Delta_y}{d}$ 

6:  $\text{var}_x += \frac{\Delta_x^2 \cdot (d-1)}{d^2} - \frac{\text{var}_x}{d}$ 
7:  $\text{cov}_{xy} += \frac{\Delta_x \cdot \Delta_y \cdot (d-1)}{d^2} - \frac{\text{cov}_{xy}}{d}$ 

```

Schemat inkrementacyjnej aktualizacji statystyk modelu przedstawiono w algorytmie 6. Jego celem jest bieżące wyznaczanie średnich $\bar{x}$, $\bar{y}$, wariancji var_x oraz kowariancji cov_{xy} na podstawie informacji pochodzących z nowo przetworzonego zapytania, reprezentowanego przez parę (x, y) , gdzie x oznacza rozmiar zapytania, a y jego czas realizacji. Początkowo, zakłada się, że wartości wszystkich statystyk są równe 0. W pierwszym kroku aktualizacji modelu inkrementowana jest liczba obserwacji d oraz obliczane są różnice pomiędzy wartościami x i y a ich średnimi. Na tej podstawie aktualizowane są odpowiednie statystyki zgodnie ze wzorami podanymi w liniach 4–7.

Zaletą tego podejścia jest istotna redukcja kosztów obliczeniowych w porównaniu do klasycznego sposobu wyznaczania średnich, wariancji i kowariancji, którego złożoność rośnie wraz z liczbą przetworzonych zapytań. Zastosowane algorytmy zapewniają natomiast stały, niski koszt aktualizacji modelu, polegający jedynie na wykonaniu kilku operacji arytmetycznych dla każdej nowej pary danych (x, y) . Ponadto, brak konieczności przechowywania historii przetworzonych zapytań przekłada się na znaczną oszczędność pamięci operacyjnej.

5.8. Algorytm przetwarzania strumienia zapytań przy użyciu opracowanej metody

Koncepcja przetwarzania strumienia zapytań w oparciu o opracowaną metodę automatycznego wyboru strategii została przedstawiona w algorytmie 7. Zakłada ona przetwarzanie strumienia zapytań Q kierowanego do zbioru danych D , zawierającego m obserwacji n zmiennych losowych. Dwa dodatkowe parametry wejściowe to: mem_{LIM} , określający maksymalną dopuszczalną ilość pamięci, jaką może zająć realizacja pojedynczego zapytania, oraz d_{REQ} , definiujący minimalną liczbę zrealizowanych zapytań potrzebną do uznania modelu za wytrenowany.

Algorytm 7 PROCESSQUERIES(Q, D, mem_{LIM}, d_{REQ})

```

1:  $x_0 \leftarrow \frac{1}{D.m} \sum_{i=1}^{D.n} r_i$ 
2:
3:  $U_L \leftarrow [CT_L, BV_L, RAD]$ 
4:  $U_R \leftarrow [CT_R, BV_R, RAD]$ 
5:  $T_L \leftarrow []$ 
6:  $T_R \leftarrow []$ 
7:
8: for  $q \in Q$  do
9:    $x \leftarrow q.SIZE()$ 
10:
11:   if  $x \leq x_0$  then
12:      $U \leftarrow U_L$ 
13:      $T \leftarrow T_L$ 
14:   else
15:      $U \leftarrow U_R$ 
16:      $T \leftarrow T_R$ 
17:
18:   if  $U \neq \emptyset$  then
19:      $S \leftarrow \text{ROUNDROBIN}(U)$ 
20:   else
21:      $S \leftarrow \text{OPTIMAL}(T)$ 
22:
23:   if  $S \in [CT_L, CT_R]$  and  $\text{MEMREQ}(S, q) > mem_{LIM}$  then
24:     if  $U \setminus S \neq \emptyset$  then
25:        $S \leftarrow \text{ROUNDROBIN}(U \setminus S)$ 
26:     else
27:        $S \leftarrow \text{OPTIMAL}(T \setminus S)$ 
28:
29:    $y \leftarrow S.APPLY(q)$ 
30:
31:    $S.UPDATEMODEL(x, y)$ 
32:
33:   if  $S \in U$  and  $S.d \geq d_{REQ}$  then
34:      $U.REMOVE(S)$ 
35:      $T.APPEND(S)$ 

```

Pierwszym krokiem strategii jest wyznaczenie wartości progowej x_0 , która stanowi punkt podziału przestrzeni zapytań ze względu na ich rozmiar. Wartość ta określana jest jako średnia liczba stanów zmiennych losowych w zbiorze danych D :

$$x_0 = \frac{1}{m} \sum_{i=1}^n r_i$$

Podział ten znajduje uzasadnienie w charakterze funkcji kosztu wykorzystywanych przez strategię CT oraz BV, które zgodnie ze wzorami 5.40 oraz 5.42 zmieniają swą postać w zależności od rozmiaru zapytania.

Następnie tworzony jest zestaw modeli kosztu realizacji zapytań dla trzech dostępnych strategii: CT, BV oraz RAD. Dla strategii CT i BV tworzone są po dwa niezależne modele: jeden dla zapytań, których rozmiar nie przekracza wartości x_0 oraz drugi dla zapytań większych niż x_0 . Modele te trafiają odpowiednio do zbiorów U_L oraz U_R , w których dolny indeks symbolizuje położenie względem punktu x_0 (L – left, R – right). Nie dotyczy to strategii RAD, dla której zgodnie ze wzorem 5.41 tworzony jest jeden wspólny model, niezależnie od rozmiaru zapytania. Model ten przypisywany jest do obu zbiorów, U_L i U_R .

W prezentowanym podejściu, początkowo wszystkie modele traktowane są jako niewytrenowane i przypisane do zbiorów U_L oraz U_R (ang. *Untrained*). Przetwarzanie kolejnych zapytań pozwala na aktualizację modeli na podstawie obserwowanych czasów realizacji. Gdy któryś z modeli osiągnie wymaganą liczbę danych uczących, zostaje przeniesiony do odpowiedniego zbioru T_L lub T_R (ang. *Trained*), zawierającego modele gotowe do podejmowania decyzji.

Każde zapytanie ze strumienia Q przetwarzane jest w analogiczny sposób. Najpierw ustalany jest jego rozmiar x . Na tej podstawie wybierany jest odpowiedni zestaw modeli: U_L i T_L w przypadku zapytań spełniających warunek $x \leq x_0$, lub U_R i T_R w przeciwnym przypadku.

Jeśli w wybranym zbiorze modeli znajdują się jeszcze modele niewytrenowane (czyli zbiór U nie jest pusty), strategia do realizacji danego zapytania wybierana jest w oparciu o algorytm ROUNDROBIN. Podejście te zapewnia, że każdy model otrzyma wystarczającą ilość danych, zanim zostanie wykorzystany do podejmowania decyzji.

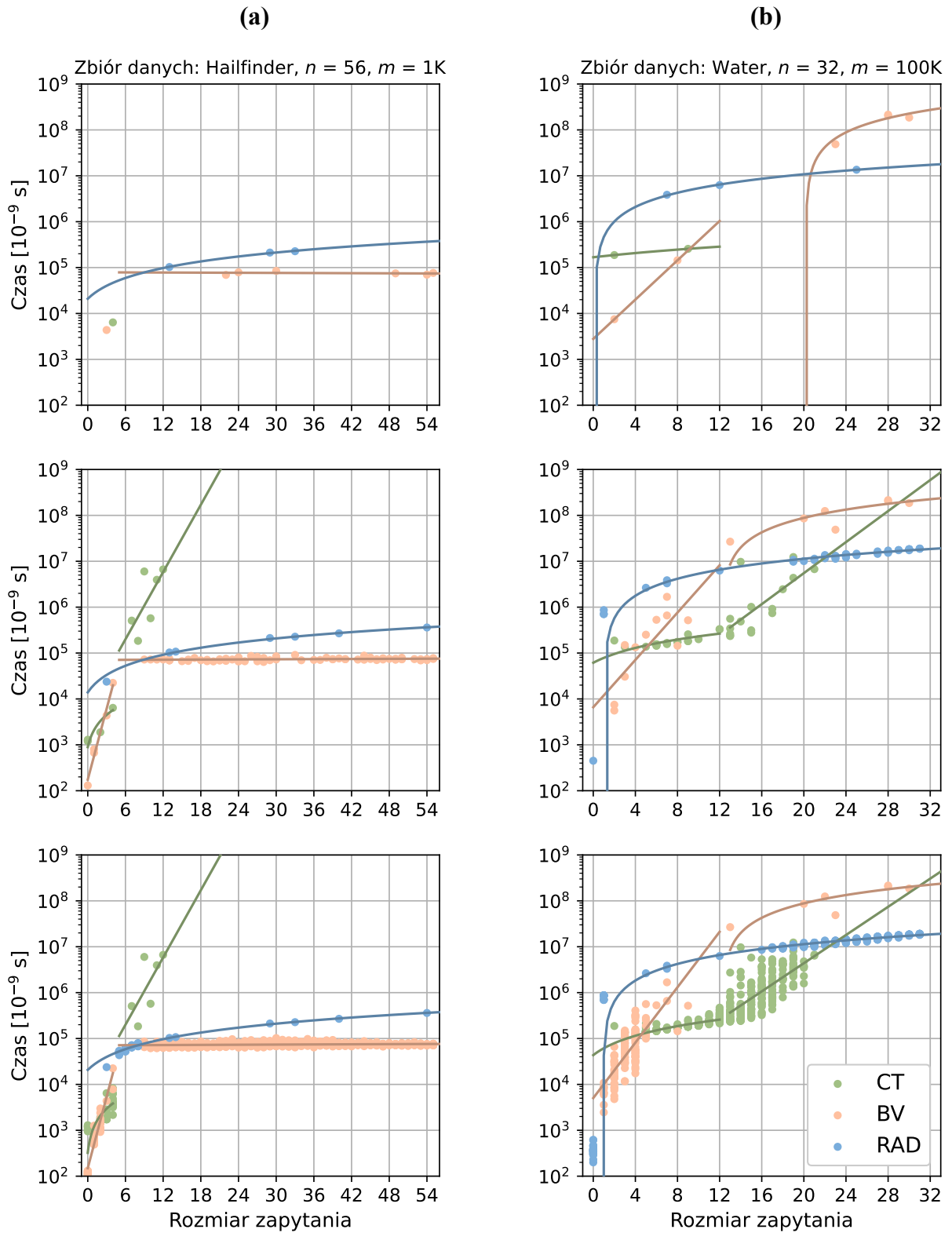
W przeciwnym razie, czyli gdy wszystkie modele są już wytrenowane, następuje estymacja czasu wykonania danego zapytania przez każdą z dostępnych strategii z wykorzystaniem modeli ze zbioru T , a następnie wybierana jest strategia o najniższym szacowanym koszcie.

Dla zapytań przydzielonych do strategii CT wykonywana jest dodatkowa kontrola ograniczeń pamięciowych. Jeżeli oszacowane zapotrzebowanie na pamięć przekracza wartość

mem_{LIM} , wówczas strategia CT zostaje wykluczona z obsługi danego zapytania, które zostaje przekierowane do innej strategii. Jeżeli istnieją inne niewytrenowane modele, to wybór strategii następuje w oparciu o podejście ROUNDROBIN. W przeciwnym razie wybierana jest strategia o najniższym estymowanym czasie realizacji z pośród wytrenowanych modeli. Zabieg ten zapobiega sytuacjom, w których strategia CT przekracza dostępne zasoby pamięciowe, co jak pokazano w sekcji 5.1 może wystąpić nawet przy relatywnie niewielkich zapytaniach.

Po wyborze strategii zapytanie jest wykonywane oraz mierzony jest czas jego realizacji y . Następnie uzyskana para (x, y) — rozmiar zapytania i czas jego wykonania — wykorzystywana jest do aktualizacji modelu odpowiadającego strategii S , przy użyciu algorytmu UPDATEMODEL, którego szczegóły przedstawiono w sekcji 5.7.

Jeżeli strategia, którą zrealizowano zapytanie związana jest z niewytrenowanym modelem, to po aktualizacji modelu sprawdzana jest liczba zapytań zrealizowanych przez daną strategię. W przypadku osiągnięcia wartości d_{REQ} , model odpowiadający danej strategii może zostać przeniesiony ze zbioru modeli niewytrenowanych U do zbioru modeli gotowych do podejmowania decyzji T .



Rysunek 5.11. Modele regresji dla dwóch zbiorów danych: (a) Hailfinder ($n = 56$, $m = 1K$) i b) Water ($n = 32$, $m = 100K$) po przetworzeniu pierwszych dziesięciu, stu i tysiąca zapytań ze strumienia (odpowiednio: pierwszy, drugi i trzeci wiersz każdej kolumny)

Rysunek 5.11 przedstawia dwa przykłady zastosowania przedstawionego algorytmu

$\text{PROCESS_QUERIES}(Q, D, \text{mem}_{\text{LIM}}, d_{\text{REQ}})$ w kontekście rzeczywistego przetwarzania strumienia zapytań zliczających z wykorzystaniem opracowanej metody automatycznego wyboru strategii. Prezentowane dane pochodzą z rzeczywistych eksperymentów - opracowana metoda została w pełni wdrożona przez autora niniejszej pracy w bibliotece SABNAtk. Szczegóły dotyczące implementacji zostaną przedstawione w kolejnej sekcji. W obu przykładach zilustrowanych na rysunku 5.11, strumień Q składa się z losowych zapytań wygenerowanych zgodnie z procedurą opisaną w sekcji 3.4.3. Argument D odnosi się do zbioru danych: Hailfinder ($n = 56$, $m = 1K$) w przypadku (a) oraz Water ($n = 32$, $m = 100K$) w przypadku (b). W obu przypadkach przyjęto jednakowe ograniczenie pamięciowe $\text{mem}_{\text{LIM}} = 32$ MiB, natomiast parametr d_{REQ} , określający minimalną liczbę obserwacji niezbędnych do uznania modelu za wytrenowany, ustawiono na wartość 3.

Na rysunku przedstawiono stan modeli regresji po przetworzeniu odpowiednio dziesięciu, stu oraz tysiąca zapytań zliczających (odpowiednio: pierwszy, drugi i trzeci wiersz w każdej kolumnie). Każdy punkt reprezentuje czas realizacji pojedynczego zapytania zliczającego. Na podstawie przedstawionych wyników można zauważyć, że opracowane modele kosztów realizacji zapytania szybko uzyskują poprawne oszacowanie parametrów regresji. Widoczne jest również, że wraz z postępem wykonywania strumienia, zapytania obsługiwane są przez strategię, które oferują najniższy koszt wykonania.

5.9. Kod źródłowy opracowanej metody

Opracowana metoda automatycznego wyboru strategii została w pełni wdrożona przez autora pracy w bibliotece SABNAtk. Kod źródłowy dostępny jest na platformie Gitlab [58]. Najistotniejsze fragmenty zostały również zawarte w aneksie.

```

1  #include <iostream>
2  #include <vector>
3  #include <sabnatk.hpp>
4  #include <estimator.hpp>
5  #include "csv.hpp"
6  #include "read_queries.hpp"
7
8  int main(int argc, char* argv[]) {
9      if (argc != 3) {
10         std::cout << "usage: random_query csv_file dat_file" << std::endl;
11         return -1;
12     }
13
14     const int N = 1;
15     using set_type = uint_type<N>;

```

```
16     using data_type = uint8_t;
17
18     std::vector<data_type> D;
19     auto [res, n, m] = read_csv(argv[1], D);
20     if (res == false) {
21         std::cout << "error: unable to read input data" << std::endl;
22         return -1;
23     }
24
25     std::vector<std::pair<set_type, set_type>> Q;
26     if (!read_queries(argv[2], Q)) {
27         std::cout << "error: unable to read queries" << std::endl;
28         return -1;
29     }
30
31     std::vector<estimator> F(1);
32
33     auto eng = sabnatk::create_counter<N, sabnatk::Auto>(n, m, std::begin(D));
34
35     for (const auto& q : Q) eng.apply(q.first, q.second, F);
36
37     return 0;
38 }
```

Listing 5.1. Realizacja strumienia zapytań Q na zbiorze danych D z wykorzystaniem opracowanej metody automatycznego wyboru strategii

Listing 5.1 przedstawia przykład wykorzystania opracowanej metody automatycznego wyboru strategii do realizacji strumienia zapytań zliczających. W pierwszej kolejności następuje wczytanie z plików zbioru danych D (Listing 5.1, linie 18-23) oraz strumienia zapytań Q (Listing 5.1, linie 25-29). Formaty tych danych muszą spełniać warunki określone odpowiednio w sekcjach 3.4.2 oraz 3.4.3. Następnie tworzony jest obiekt funkcyjny F , służący do zgłaszania wyników zrealizowanych zapytań. Koncepcja tego podejścia została szerzej omówiona w sekcji 3.3.1.

Opracowana metoda została zaimplementowana jako osobna strategia (klasa C++) o nazwie `AutoCounter`. Z racji znacznej objętości jej kod źródłowy został przedstawiony w aneksie na listingu 8.3. Implementacja jest w pełni zgodna z interfejsem wykorzystywanym przez bibliotekę `SABNatk`, polegającym na dostarczeniu obiektu (silnika), mającego zdolność realizacji zapytań za pomocą metody `apply`. Szczegóły dotyczące tego podejścia zostały opisane w sekcjach 3.3.4 oraz 3.3.5.

W przedstawionym listingu 5.1 utworzenie silnika związanego z metodą automatycznego wyboru strategii następuje poprzez wywołanie funkcji `create_counter` z odpowiednim

argumentem (`Auto`). Funkcja ta wywołuje z kolei konstruktor klasy `AutoCounter`, a dokładniej zaprzyjaźnioną funkcję:

```
create_AutoCounter(int n, int m, Iter it, int burn_in = -1,  
                  int CT_memory_limit = 32, int required_instances = 3)
```

Argumenty tej funkcji to:

- `n` – liczba zmiennych w zbiorze danych D .
- `m` – liczba obserwacji w zbiorze danych D .
- `it` – iterator wskazujący początek zbioru danych D .
- `burn_in` – argument związany z dodatkowo zaimplementowaną funkcjonalnością (nieujęta w algorytmie 7), polegającą na wstępnym trenowaniu modeli przed rozpoczęciem właściwego przetwarzania strumienia zapytań. Argument ten określa liczbę losowych zapytań, które silnik powinien zrealizować w celu wstępnego uczenia. Funkcjonalność ta może być przydatna w specyficznych przypadkach, np. gdy strumień zapytań składa się z zapytań o tym samym rozmiarze, co utrudnia wytrenowanie modeli podczas przetwarzania właściwego strumienia. Domyślna wartość (-1) powoduje wyłączenie funkcjonalności.
- `CT_memory_limit` – zapobiega przekroczeniu dostępnej pamięci przez strategię CT określając maksymalną dopuszczalną ilość pamięci (MiB) na realizację pojedynczego zapytania. Przekroczenie tej wartości skutkuje przekazaniem zapytania do realizacji najlepszą z pozostałych strategii. Domyślna wartość to 32 MiB.
- `required_instances` – definiuje minimalną liczbę zrealizowanych zapytań potrzebną do uznania modelu za wytrenowany. Domyślna wartość wynosi 3.

Zadaniem funkcji `create_AutoCounter` jest utworzenie trzech niezależnych silników dla strategii CT, RAD i BV oraz powiązanie ich z modelami kosztu realizacji zapytania, danymi wzorami 5.40-5.42. Etap ten odpowiada liniom 1-6 omówionego wcześniej algorytmu 7, przedstawiającego przetwarzanie strumienia zapytań z wykorzystaniem opracowanej metody.

Modele kosztu reprezentowane są jako obiekty klasy `LRmodel`, której kod źródłowy zamieszczono w aneksie na listingu 8.2. Tworząc model za pomocą konstruktora tej klasy należy wskazać, której strategii dotyczy (CT, RAD, BV) oraz jaki jest jego rodzaj - liniowy czy wykładniczy. Stan ten określa zmienna logiczna `is_linear`, w której prawda oznacza model liniowy a fałsz model wykładniczy. Do pól klasy należą między innymi statystyki wykorzystywane podczas aktualizacji modelu: średnie arytmetyczne zmiennych

x oraz y , wariancja zmiennej x oraz kowariancja zmiennych x i y . Wartości te aktualizowane są za pomocą metody `update`, będącej implementacją przedstawionego w sekcji 5.7 algorytmu 6. Ponadto, klasa zapewnia również metodę `evaluate` zwracającą wartość funkcji kosztu dla danego zapytania o rozmiarze x , uwzględniając odpowiednią transformację w przypadku modeli wykładniczych, opisaną w sekcji 5.7.

Po utworzeniu silnika, związanego z metodą automatycznego wyboru strategii, następuje realizacja strumienia zapytań poprzez wywołanie metody `apply` dla każdego zapytania ze strumienia Q (listing 5.1, linia 35). Każde wywołanie metody `apply` odpowiada realizacji linii 9-35 z algorytmu 7 opisanego w sekcji 5.8.

5.10. Badanie wpływu opracowanej metody na wydajność zapytań zliczających

Niniejsza sekcja przedstawia wyniki dotyczące wydajności opracowanej metody automatycznego wyboru strategii do realizacji zapytań zliczających, zaimplementowanej w bibliotece SABNAtk jako strategia o nazwie AC. Wydajność opracowanego podejścia została porównana z wydajnością strategii, na których ona bazuje, tj. CT, RAD oraz BV. Dodatkowo, w celu zapewnienia szerszego kontekstu porównawczego, do badań włączono również strategie powszechnie stosowane do realizacji zapytań zliczających, oparte na tablicach haszujących (hash tables) oraz alternujących drzewach decyzyjnych (ADT) [41]. Podejścia te zostały szerzej przedstawione w sekcji 3.2. W badaniach wykorzystano implementację tych strategii dostarczoną przez twórców biblioteki SABNAtk. Kod źródłowy dostępny jest pod następującymi adresami: HT – [2], ADT – [1].

Implementacja HT oparta jest na standardowej bibliotece C++. Tablica haszująca reprezentuje tabelę kontyngencji tworzoną poprzez bezpośrednie skanowanie bazy danych, w której klucze kodują przypisania zmiennych w $Pa(X_i)$, natomiast wartości przechowują wektory zliczeń odpowiadające przypisaniom zmiennej X_i . W celu maksymalizacji wykorzystania pamięci podręcznej strategia operuje na danych zapisanych w porządku wierszowym.

Z kolei implementacja podejścia ADT to autorskie rozwiązanie autorów biblioteki SABNAtk, zapewniające lepszą wydajność [30] niż inne dostępne implementacje, np. [64]. Z uwagi na sposób działania ADT, realizacja zapytań zliczających przy użyciu tej strategii wymaga wcześniejszego indeksowania danych wejściowych. We wszystkich wynikach badań dotyczących tej strategii koszt utworzenia indeksu został wliczony w całkowity koszt przetwarzania zapytań. Ponadto ADT umożliwia konfigurację rozmiaru list liści poprzez hiperparametr ℓ [41]. Przetestowano różne wartości parametru w celu osiągnięcia kompromisu między wydajnością zapytań a zużyciem pamięci, przy czym ostatecznie zastosowano $\ell = 16$ we wszystkich eksperymentach.

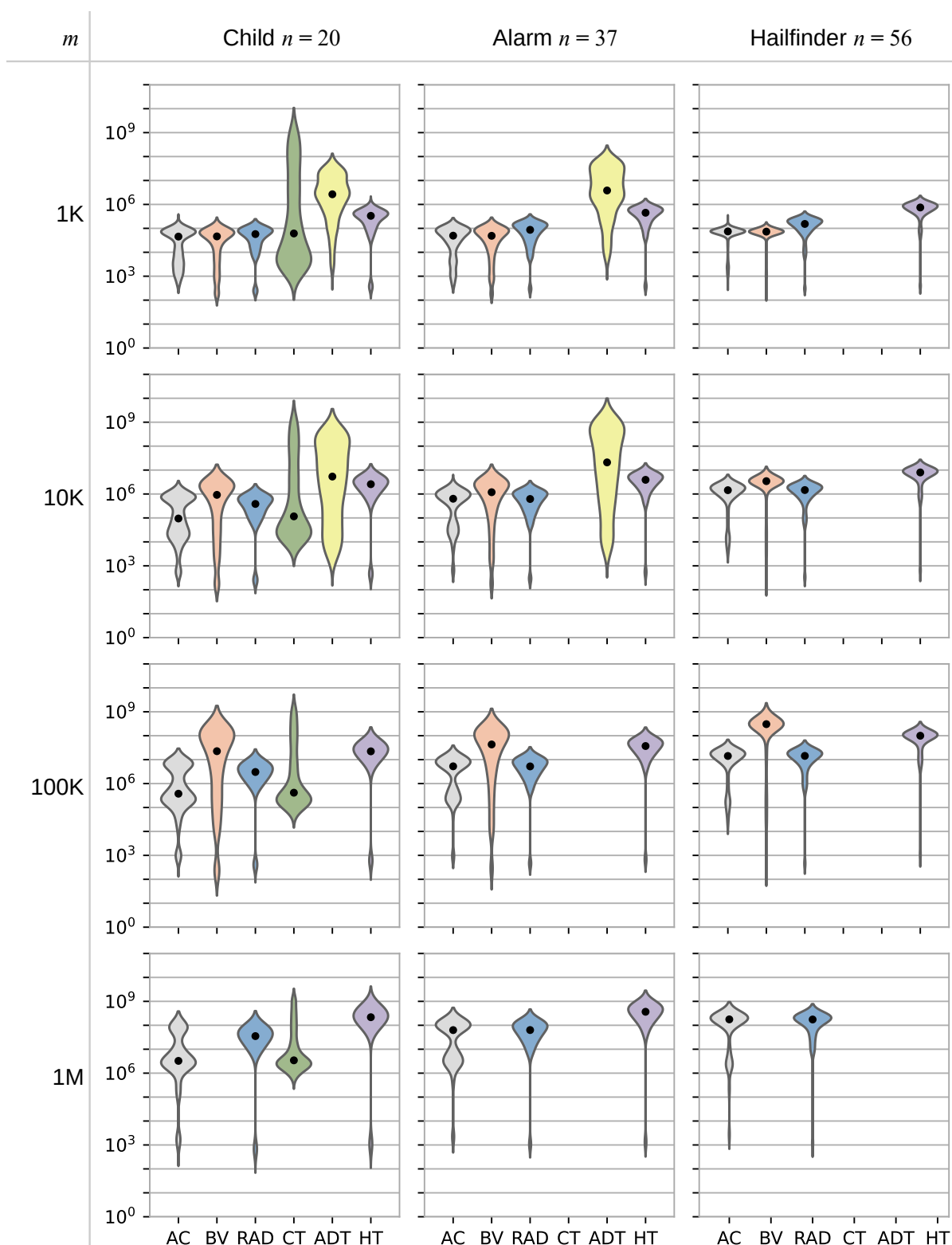
Eksperymenty przedstawione w niniejszej sekcji przeprowadzono na platformie obliczeniowej Intel Cascade Lake, której specyfikacja została przedstawiona w sekcji 3.4.1.

Benchmark losowych zapytań

W pierwszej serii eksperymentów zbadano, jak różne strategie reagują na strumień losowych zapytań. Celem testów było porównanie przeciętnej wydajności każdej ze strategii w sytuacji, gdy strumień zapytań składa się z zapytań o różnym rozmiarze. Koncepcja tego benchmarku oraz szczegóły dotyczące jego przygotowania zostały szerzej opisane w sekcji 3.4.3.

W badaniach rozpatrywano osiem różnych zbiorów danych, z których każdy występował w czterech wersjach różniących się liczbą obserwacji (1K, 10K, 100K, 1M). Szczegóły dotyczące przygotowania zbiorów danych zostały przedstawione w sekcji 3.4.2.

Dla każdego zbioru danych rozpatrywano strumień zawierający sto tysięcy zapytań zliczających. Ten sam strumień zapytań uruchomiono dla każdej z sześciu analizowanych strategii – BV, RAD, CT, ADT, HT oraz opracowanej metody automatycznego wyboru AC. Łącznie obejmuje to 192 przypadki testowe (8 zbiorów danych $\times$ 4 wersje $\times$ 6 strategii), dlatego aby utrzymać czas eksperymentów w rozsądnych ramach, zdecydowano o nałożeniu limitu 12 godzin na realizację pojedynczego przypadku testowego.



Rysunek 5.12. Wykresy skrzypcowe prezentujące rozkład czasów realizacji zapytań o losowym rozmiarze

Rysunek 5.12 prezentuje zestaw wykresów skrzypcowych dla trzech wybranych zbiorów danych. Pojedynczy wykres ilustruje rozkład czasów odpowiedzi (w skali logarytmicznej) dla próby tysiąca losowo wybranych zapytań strumienia, a czarną kropką zazna-

czono medianę.

Rozpoczynając analizę wyników od indywidualnych strategii, na których bazuje opracowana metoda automatycznego wyboru (BV, RAD, CT), można dostrzec, że strategia BV wyróżnia się w testach z tysiącem obserwacji, co jest rezultatem efektywnego wykorzystania pamięci podręcznej oraz przyspieszenia poprzez instrukcje wektorowe [30]. Dodatkowo, opracowane metody optymalizacji przedstawione w sekcjach 4.1 oraz 4.2 pozwalają na odcinanie znacznych części drzewa DFS, ponieważ wiele konfiguracji zmiennych nie występuje w małych zbiorach danych. Czynniki te tracą jednak na znaczeniu wraz ze wzrostem liczby obserwacji w bazie danych. Przykładowo, strategia BV nie była w stanie obsłużyć pełnego strumienia zapytań dla zbiorów danych z milionem instancji. W takich przypadkach lepiej sprawdzało się podejście RAD, będące jedyną strategią (poza AC), która przetworzyła pełny strumień zapytań we wszystkich przypadkach testowych. Strategia CT okazała się konkurencyjna jedynie w przypadkach dotyczących małej liczby zmiennych, co jest zgodne z oczekiwaniami, gdyż jak pokazano w sekcji 5.3, jej wymagania pamięciowe rosną wykładniczo względem rozmiaru zapytań.

Analizując pozostałe podejścia powszechnie wykorzystywane do realizacji zapytań zliczających [30], można zauważyć, że strategia ADT była w stanie przetworzyć jedynie cztery z prezentowanych na rysunku 5.12 przypadków testowych i nawet w tych sytuacjach była wolniejsza od pozostałych podejść. Z kolei strategia HT ukończyła większość przypadków testowych, a rozkład jej czasów odpowiedzi przypominał strategię RAD. Należy jednak zauważyć, że strategia RAD okazała się znacznie szybsza, co wynika z faktu, iż nie wymaga kosztownej operacji haszowania oraz zyskuje dzięki organizacji danych w sposób sprzyjający lepszemu wykorzystaniu pamięci podręcznej.

Opracowana metoda automatycznego wyboru strategii (AC) osiągnęła najlepsze wyniki we wszystkich przypadkach testowych. Fakt ten potwierdza słuszność opracowanej koncepcji, polegającej na selektywnym wyborze strategii do realizacji indywidualnych zapytań strumienia.

Szczegółowe wyniki eksperymentów przedstawiono w tabeli 5.2. Tabela ta pokazuje przyspieszenia uzyskane dzięki opracowanej metodzie AC względem innych strategii. Wskaźnik ten został obliczony jako stosunek całkowitego czasu realizacji strumienia zapytań daną strategią do czasu wymaganego przez AC.

Opracowane rozwiązanie wyraźnie przewyższa strategię ADT oraz HT. Przyspieszenie uzyskane względem strategii ADT wynosi zazwyczaj kilkaset razy (średnio $330\times$). Należy zaznaczyć, że w rzeczywistości jest ono jeszcze większe, ponieważ strategia ADT w większości przypadków przekroczyła przyjęty limit 12 godzin na przetworzenie strumienia zapytań, co uniemożliwia obliczenie przyspieszenia w tych przypadkach. Z kolei przyspieszenie względem strategii HT znajduje się w przedziale $5.45\times - 73.42\times$, ze średnią $13.3\times$.

Strategia	Zbiór danych (n)	Przyspieszenie			
		$m = 1K$	$m = 10K$	$m = 100K$	$m = 1M$
BV	Sachs (11)	1.18	5.71	23.59	42.42
	Child (20)	1.11	3.89	26.93	–
	Insurance (27)	1.07	3.26	21.14	–
	Mildew (35)	1.24	4.10	40.31	–
	Alarm (37)	1.05	2.49	13.93	–
	Barley (48)	1.09	2.67	23.98	–
	Hailfinder (56)	1.03	2.20	19.77	–
	Win95 (76)	1.04	1.30	6.62	–
RAD	Sachs (11)	1.97	4.37	9.40	11.37
	Child (20)	1.44	1.27	1.57	1.71
	Insurance (27)	1.51	1.07	1.19	1.25
	Mildew (35)	1.81	1.02	1.08	1.04
	Alarm (37)	1.83	1.06	1.09	1.19
	Barley (48)	2.06	1.02	1.01	1.05
	Hailfinder (56)	2.38	1.03	1.02	1.06
	Win95 (76)	2.77	1.11	1.06	1.08
CT	Sachs (11)	1.68	1.05	1.11	1.17
	Child (20)	4450.99	548.63	78.43	8.08
	Insurance (27)	*	*	*	*
	Mildew (35)	*	*	*	*
	Alarm (37)	*	*	*	*
	Barley (48)	*	*	*	*
	Hailfinder (56)	*	*	*	*
	Win95 (76)	*	*	*	*
ADT	Sachs (11)	26.11	24.79	10.66	1.78
	Child (20)	177.85	228.89	–	–
	Insurance (27)	397.18	480.35	–	–
	Mildew (35)	–	–	–	–
	Alarm (37)	282.79	360.53	–	–
	Barley (48)	–	–	–	–
	Hailfinder (56)	–	–	–	–
	Win95 (76)	878.78	–	–	–
HT	Sachs (11)	11.87	32.51	66.53	73.42
	Child (20)	7.30	7.90	10.85	11.24
	Insurance (27)	7.11	6.30	7.73	7.46
	Mildew (35)	9.21	6.46	8.44	–
	Alarm (37)	8.13	6.16	7.26	6.44
	Barley (48)	9.38	5.67	7.28	–
	Hailfinder (56)	10.62	5.45	6.80	–
	Win95 (76)	12.85	5.78	6.42	–

* oznacza przekroczenie dostępnej pamięci operacyjnej

– oznacza przekroczenie przyjętego limitu 12 godzin

Tabela 5.2: Przyspieszenie opracowanej metody automatycznego wyboru strategii w porównaniu do różnych podejść dla benchmarku strumienia stu tysięcy losowych zapytań

Jednocześnie opracowana metoda oferuje znaczące przyspieszenie względem strategii, spośród których dokonuje wyboru (tj. BV, RAD oraz CT). Przykładowo, w przypadku zbioru Child z milionem obserwacji podejście AC było $1.71\times$ szybsze od strategii RAD oraz $8.08\times$ szybsze od strategii CT. Oznacza to, że opracowana metoda skróciła czas wykonania o ponad 40% względem najlepszej możliwej indywidualnej strategii, co stanowi istotne usprawnienie w kontekście aplikacji uczenia maszynowego, które często przetwarzają nawet miliony zapytań zliczających [30].

Strategia	Zbiór danych (n)	Procent zrealizowanych zapytań			
		$m = 1K$	$m = 10K$	$m = 100K$	$m = 1M$
BV	Sachs (11)	45.5	27.5	18.4	27.4
	Child (20)	65.0	11.8	12.8	10.3
	Insurance (27)	77.9	3.7	3.7	3.8
	Mildew (35)	74.4	3.7	3.3	5.8
	Alarm (37)	86.4	3.8	1.1	4.2
	Barley (48)	80.5	0.0	0.0	2.7
	Hailfinder (56)	82.9	3.5	3.3	3.7
	Win95 (76)	96.1	37.8	1.6	1.4
RAD	Sachs (11)	5.7	3.4	5.2	9.1
	Child (20)	2.8	43.8	31.5	30.8
	Insurance (27)	3.4	66.8	57.8	57.9
	Mildew (35)	20.0	85.8	85.8	85.8
	Alarm (37)	2.6	72.3	72.1	65.3
	Barley (48)	9.1	87.4	84.3	84.2
	Hailfinder (56)	5.4	82.7	80.4	79.2
	Win95 (76)	0.0	53.0	77.5	76.3
CT	Sachs (11)	48.8	69.1	76.4	63.5
	Child (20)	32.2	44.4	55.7	58.9
	Insurance (27)	18.7	29.5	38.5	38.3
	Mildew (35)	5.6	10.5	10.9	8.4
	Alarm (37)	11.0	23.9	26.8	30.5
	Barley (48)	10.5	12.6	15.7	13.1
	Hailfinder (56)	11.7	13.8	16.3	17.1
	Win95 (76)	3.9	9.2	20.9	22.3

Tabela 5.3: Procent zrealizowanych zapytań przez strategie wybierane przez opracowaną metodę automatycznego wyboru

Tabela 5.3 przedstawia dodatkowe informacje na temat działania mechanizmu automatycznego wyboru strategii, prezentując procent zapytań zrealizowanych przez poszczególne strategie wybierane przez AC. Przykładowo, dla zbioru Sachs z tysiącem obserwacji strategia BV przetworzyła 45.5%, RAD 5.7%, a CT 48.8% wszystkich zapytań.

Analizując wyniki dla BV, można zauważyć, że strategia ta była dominującym wyborem w zbiorach danych o najmniejszej liczbie obserwacji (kolumna 1K w tabeli). Wraz

ze wzrostem liczby zmiennych oraz obserwacji dominującą strategią staje się RAD, która realizowała większość zapytań w zbiorach zawierających co najmniej tyle zmiennych co Insurance oraz posiadających co najmniej dziesięć tysięcy obserwacji. Z kolei podejście CT preferowane było w przypadkach o najmniejszej liczbie zmiennych, stając się najczęściej wybieraną strategią w zbiorze Sachs oraz w większości wariantów zbioru Child (dla $m \geq 10K$).

Warto jednak zauważyć, że pomimo tego, iż strategia CT nie była w stanie samodzielnie zrealizować wszystkich zapytań strumienia dla zbiorów o liczbie zmiennych przynajmniej takiej jak w zbiorze Insurance (co pokazano wcześniej w tabeli 5.2, opracowana metoda automatycznego wyboru AC pozwoliła jej na istotne przyczynienie się do przetwarzania zapytań. Przykładem może być zbiór Insurance, w którym jak zaprezentowano w tabeli 5.3, strategia CT zrealizowała 18.7 – 38.3 % wszystkich zapytań.

Uczenie sieci Bayesowskich

W drugiej serii eksperymentów zbadano jak opracowana metoda automatycznego wyboru strategii sprawdza się w rzeczywistych aplikacjach uczenia maszynowego. Zapytania zliczające są kluczowymi operacjami wykonywanymi podczas uczenia struktury sieci bayesowskich opartych na funkcji punktacji oraz w procesie odkrywania kocy Markova (Markov blanket discovery) [34], dlatego w badaniach skupiono się właśnie na tych problemach.

Oba zagadnienia wymagają wielokrotnego wyznaczania zbiorów rodziców $Pa(X_i)$ (parent set assignment) [33] i dla danej zmiennej $X_i \in \mathcal{X}$ mogą być rozwiązane w sposób dokładny poprzez przeszukiwanie kraty o n poziomach, utworzonej przez częściowy porządek zawierania zbiorów w zbiorze potęgowym utworzonym dla $\mathcal{X} \setminus X_i$. W rezultacie dla każdej zmiennej X_i wykonywane są zapytania postaci $Query(X_i, Pa, F)$, gdzie Pa iteruje po wszystkich możliwych podzbiorach zbioru potęgowego $\mathcal{X} \setminus X_i$, zaczynając od zbioru pustego. Z kolei obiekt funkcyjny F implementuje funkcję oceny, np. MDL [61] lub BDeu [20], która ocenia przypisanie zbioru Pa jako rodziców zmiennej X_i . Oznacza to że, na poziomie $i = 0, \dots, n - 1$ wykonywanych jest łącznie $\binom{n-1}{i}$ zapytań, co tworzy charakterystyczny wzorec zapytań, których rozmiar rośnie wraz z postępem obliczeń. Podobnie jak poprzednio, wydajność opracowanej metoda automatycznego wyboru strategii (AC) została porównana zarówno z wydajnością strategii, na których bazuje (BV, RAD, CT), jak i innych powszechnie stosowanych podejść do realizacji zapytań zliczających (ADT, HT). W tym celu wszystkie strategie zostały wdrożone w otwarto źródłowym solverze wyznaczania zbiorów rodziców, udostępnionym jako część pakietu SABNA [57]. Solver ten wykorzystuje funkcję oceny MDL oraz stosuje kilka optymalizacji, aby wyeliminować część zapytań na podstawie dotychczas uzyskanych wyników. Mimo tego,

eksplorowany obszar kombinatoryczny jest rozległy, dlatego problem charakteryzuje się istotnym zapotrzebowaniem pamięciowym, stanowiąc tym samym praktyczny punkt odniesienia dla oceny opracowanej metody. Tabela 5.4 pokazuje wybrane do badań zbiory danych oraz wynikową liczbę zapytań, jaka została zrealizowana w każdym przypadku.

Zbiór danych (n)	Liczba zrealizowanych zapytań (w milionach)		
	$m = 1K$	$m = 10K$	$m = 100K$
Child (20)	0.32	0.82	0.88
Insurance (27)	1.07	5.72	8.11
Mildew (35)	0.22	1.81	9.48
Alarm (37)	10.81	68.48	88.18
Barley (48)	0.65	5.49	44.63

Tabela 5.4: Całkowita liczba zapytań zrealizowanych przez solver wyznaczania zbioru rodziców (w milionach)

W przeprowadzonych eksperymentach, zamiast uwzględniania wszystkich możliwych rozmiarów zbiorów rodziców, jak wymaga tego solver dokładny, ograniczono rozmiar do $|Pa| \leq 6$, co umożliwiło realizację testów obliczeniowych w rozsądnych ramach czasowych. Warunek ten odpowiada heurystyce zakładającej, że żadna zmienna w końcowej sieci Bayesa nie posiada więcej niż sześciu rodziców. Podobnie jak w benchmarku losowych zapytań, czas rozwiązywania danego problemu ograniczono do 12 godzin.

Strategia	Zbiór danych (n)	Całkowity czas realizacji [s]		
		$m = 1K$	$m = 10K$	$m = 100K$
BV	Child (20)	2.61	102.76	1450.30
	Insurance (27)	6.54	576.22	13260.60
	Mildew (35)	3.60	812.42	–
	Alarm (37)	48.28	4405.59	–
	Barley (48)	13.35	2828.21	–
RAD	Child (20)	6.42	144.10	1441.73
	Insurance (27)	20.80	1013.54	13534.90
	Mildew (35)	5.23	386.59	19264.30
	Alarm (37)	191.42	11577.70	–
	Barley (48)	16.01	1104.07	–
CT	Child (20)	99.56	119.89	259.76
	Insurance (27)	*	*	*
	Mildew (35)	*	*	*
	Alarm (37)	*	*	*
	Barley (48)	*	*	*
ADT	Child (20)	148.36	302.01	577.03
	Insurance (27)	1609.36	2982.92	5081.62
	Mildew (35)	–	–	–
	Alarm (37)	–	–	–
	Barley (48)	–	–	–
HT	Child (20)	141.36	1172.60	11251.40
	Insurance (27)	1731.40	11988.40	–
	Mildew (35)	38058.90	–	–
	Alarm (37)	–	–	–
	Barley (48)	–	–	–
AC	Child (20)	1.79	22.28	164.15
	Insurance (27)	5.19	149.84	1540.73
	Mildew (35)	1.96	96.96	3642.08
	Alarm (37)	45.42	1630.83	16664.90
	Barley (48)	7.18	347.63	20476.20

* oznacza przekroczenie dostępnej pamięci operacyjnej

– oznacza przekroczenie przyjętego limitu 12 godzin

Tabela 5.5: Całkowity czas realizacji solvera wyznaczania zbioru rodziców (w sekundach)

Tabela 5.5 przedstawia podsumowanie eksperymentów, pokazując całkowity czas realizacji zadań dla wszystkich badanych strategii. Z prezentowanych danych wynika, że wydajność poszczególnych strategii zależy od rozmiaru bazy danych wejściowych. Rozpoczynając analizę od zbiorów danych zawierających tysiąc obserwacji ($m = 1K$), należy zauważyć, że strategia BV okazała się najlepsza wśród indywidualnych podejść

(tj. z wyłączeniem opracowanej metody AC). Wysoka efektywność BV wynika z tego, że niewielki rozmiar bazy danych sprzyja lokalności pamięci, a jednocześnie ogranicza rozmiar eksplorowanego drzewa DFS. Warto zauważyć, że podczas gdy podejścia BV oraz RAD zrealizowały niemal wszystkie przypadki testowe dla $m = 1K$ w czasie poniżej jednej minuty, strategie ADT oraz HT w wielu przypadkach nie ukończyły zadań nawet w czasie 12 godzin. Przykładem jest zbiór Mildew, który okazał się zbyt wymagający dla ADT, aby zmieścić się w założonym limicie czasu, a dla podejścia HT wymagał ponad dziesięciu godzin. Dla porównania BV oraz RAD realizują to samo zadanie odpowiednio w czasie trzech i pięciu sekund. Z kolei podejście CT przekracza dostępną pamięć dla zbiorów danych większych niż Child. Powodem tej sytuacji jest to, że pomimo przyjętego założenia o rozmiarze zapytań ($|Pa| \leq 6$), solver wymaga realizacji stosunkowo niewielkiej liczby zapytań o większym rozmiarze, które generowane są w początkowej fazie w celu obliczenia wykorzystywanych wewnętrznie ograniczeń.

Przechodząc do analizy wydajności indywidualnych podejść w zbiorach danych zawierających dziesięć tysięcy obserwacji ($m = 10K$), widoczne jest, że strategie BV oraz RAD przewyższają pozostałe podejścia. Strategia BV zapewniła najlepszy czas realizacji w trzech przypadkach (Child, Insur, Alarm), natomiast RAD dominował w dwóch pozostałych (Mild, Barley). Wyniki te potwierdzają, że w zależności od danych wejściowych oraz rodzaju zapytań preferowane mogą być różne strategie.

W przypadku zbiorów danych o rozmiarze stu tysięcy ($m = 100K$) każda z indywidualnych strategii napotkała trudności w realizacji wszystkich przypadków testowych w czasie 12 godzin. Najlepiej wypadło podejście RAD, będąc jedyną indywidualną strategią, która ukończyła zadanie w przynajmniej trzech z pięciu zbiorów danych.

Analizując dane uzyskane dla opracowanej metody automatycznego wyboru strategii, tabela 5.5 jednoznacznie pokazuje, że podejście AC było jedynym, które ukończyło testy w założonym limicie czasu we wszystkich zbiorach danych. Ponadto we wszystkich przypadkach testowych strategia ta okazała się najlepszym podejściem, oferując istotne przyspieszenie względem innych strategii realizacji zapytań. W porównaniu do ADT, AC działa od $3.3\times$ szybciej (Insur, 100K) do $310\times$ szybciej (Insur, 1K). Przyspieszenie względem HT jest jeszcze większe i wynosi od $68\times$ (Child, 100K) do $19418\times$ (Mild, 1K). Ponadto, opracowana metoda oferuje znaczną poprawę w stosunku do strategii, na których bazuje. Przykładowo, analizując wyniki dla zbioru Child w wariancie $m = 10K$, w którym najlepszą indywidualną strategią okazała się BV, strategia AC poprawia ten wynik $4.61\times$.

W celu podsumowania, w tabeli 5.6 przedstawiono, jaką część całkowitego czasu wykonania solver poświęca na obsługę zapytań zliczających. Jak pokazuje tabela, we wszystkich analizowanych przypadkach testowych, udział zapytań zliczających przekracza 95% całkowitego czasu realizacji. Wynik ten jednoznacznie pokazuje, że nawet niewiel-

kie usprawnienia w zakresie efektywności zapytań zliczających mogą przekładać się na istotne korzyści w ogólnej wydajności aplikacji uczenia maszynowego.

Zbiór danych (n)	Udział zapytań zliczających		
	$m = 1K$	$m = 10K$	$m = 100K$
Child (20)	95.0%	99.1%	99.1%
Insurance (27)	97.3%	99.5%	99.7%
Mildew (35)	96.3%	99.5%	99.8%
Alarm (37)	98.1%	99.5%	99.7%
Barley (48)	98.6%	99.7%	99.9%

Tabela 5.6: Udział zapytań zliczających w całkowitym czasie realizacji solvera wyznaczania zbioru rodziców

6. Równoległa realizacja metody automatycznego wyboru strategii

Przedstawione dotychczas wyniki dla opracowanej metody automatycznego wyboru strategii dotyczyły realizacji z wykorzystaniem pojedynczego wątku. W celu zapewnienia skalowalności i umożliwienia wdrożenia rozwiązania w wielu w rzeczywistych scenariuszach wymagających dużej mocy obliczeniowej, opracowano również wersję wspierającą równoczesne przetwarzanie wielowątkowe. Opracowana metoda została zaprezentowana podczas konferencji Euro-Par 2023 oraz opublikowana w materiałach pokonferencyjnych [17].

6.1. Koncepcja opracowanego rozwiązania

Jak przedstawiono w sekcji 3.3.5 dotyczącej koncepcji zrównoleglania realizacji zapytań zliczających, kluczowym aspektem architektury biblioteki SABNAtk jest bezstanowy charakter dostępnych w niej strategii przetwarzania (BV, RAD, CT). Realizacja pojedynczego zapytania odbywa się za pomocą metody `apply`, która nie modyfikuje wewnętrznych danych silnika, a jedynie odczytuje informacje niezbędne do realizacji zapytania. W konsekwencji wiele wątków może jednocześnie korzystać z tego samego obiektu strategii bez ryzyka naruszenia integralności danych. Taka implementacja umożliwia naturalne zastosowanie modelu równoległości zadań (task parallelism), w którym poszczególne wywołania metody `apply` stanowią niezależne zadania, gotowe do równoległego wykonania na wielu rdzeniach procesora.

Należy zauważyć, że opracowana metoda automatycznego wyboru strategii w dotychczas przedstawionej formie nie spełnia warunku bezstanowości. Jej koncepcja opiera się bowiem na dokonywaniu aktualizacji współczynników regresji obecnych w modelach czasu realizacji zapytań zliczających. W rezultacie, próba wielowątkowej realizacji zapytań prowadziłaby do powstania sytuacji wyścigu pomiędzy wątkami (race condition), co mogłoby skutkować niepoprawnym działaniem metody.

Aby zapobiec tej sytuacji, opracowana metoda automatycznego wyboru strategii została zaimplementowana w nowej wersji, zapewniającej w pełni bezpieczne wykonanie wielowątkowe. Cel ten został osiągnięty dzięki zastosowaniu mechanizmu muteksów [81]. Muteksy (ang. mutexes, mutual exclusions) stanowią narzędzie synchronizacji wykorzystywane w środowiskach wielowątkowych w celu zapewnienia, że w danym momencie tylko jeden wątek ma dostęp do zasobów wymagających ochrony. Dzięki ich zastosowaniu możliwe jest zapobieganie sytuacjom wyścigu poprzez blokowanie dostępu do współ-

dzielonego zasobu do czasu zakończenia jego modyfikacji przez dany wątek.

W języku C++ do synchronizacji dostępu do współdzielonych danych powszechnie stosowany jest mureks w połączeniu z klasą `std::scoped_lock` (dostępną od standardu C++17). Konstrukcja ta automatycznie blokuje mureks w momencie utworzenia obiektu `scoped_lock` oraz odblokuje go w momencie wyjścia z zakresu, w którym został zadeklarowany. Takie rozwiązanie znacząco upraszcza zarządzanie blokadami oraz eliminuje ryzyko przypadkowego niezwolnienia blokady w przypadku wystąpienia wyjątków. Listing 6.1 przedstawia najprostszy przykład wykorzystania mureksu wraz z klasą `scoped_lock`. Przedstawione podejście umożliwia bezpieczne wywołanie funkcji `increment` przez wiele wątków jednocześnie, ponieważ gwarantuje, że w danym momencie tylko jeden wątek uzyskuje dostęp do sekcji odpowiedzialnej za modyfikację wartości zmiennej. W momencie utworzenia obiektu klasy `std::scoped_lock` mureks zostaje zablokowany, a jego odblokowanie następuje automatycznie po wyjściu z zakresu funkcji.

```
1 #include <mutex>
2
3 std::mutex mtx;
4 int counter = 0;
5
6 void increment() {
7     std::scoped_lock lock(mtx);
8     ++counter;
9 }
```

Listing 6.1. Przykład wykorzystania mureksu i klasy `std::scoped_lock` do ochrony sekcji krytycznej przed sytuacją wyścigu

Analizując algorytm 7, przedstawiający strategię przetwarzania zapytań w oparciu o opracowaną metodę automatycznego wyboru strategii, należy zauważyć, że zawiera on fragmenty wymagające zabezpieczenia przed sytuacjami wyścigu. Przykładem takiej operacji jest aktualizacja modelu odpowiadającego strategii, którą zrealizowano dane zapytanie (algorytm 7, linia 31). Procedura ta polega na modyfikacji współczynników regresji w oparciu o algorytmy inkrementacyjne, co przedstawiono w algorytmie 6. Wielowątkowe wykonanie tej procedury dla danego modelu, bez wprowadzenia odpowiednich zabezpieczeń, mogłoby skutkować niepoprawną aktualizacją współczynników regresji, prowadząc do błędnego oszacowania kosztu realizacji strategii.

Dostosowania konieczne w celu poprawnego wykonania wielowątkowego wprowadzono także w procesie podejmowania decyzji o tym, czy strategia realizacji zapytania powinna zostać wybrana w oparciu o procedurę ROUNDROBIN, czy OPTIMAL (algorytm 7, linie 11–21). Adaptacji wymagała również procedura obsługująca przypadki, w których

wskazana do realizacji zapytania strategia CT nie mogła zostać zastosowana ze względu na przekroczenie dostępnej pamięci (algorytm 7, linie 23–27).

Szczegółowe aspekty implementacyjne omawianych zabezpieczeń zostaną przedstawione w kolejnej sekcji.

6.2. Kod źródłowy opracowanej metody

Opracowana metoda umożliwiająca wielowątkową realizację automatycznego wyboru strategii została w pełni wdrożona przez autora pracy w bibliotece SABNAtk. Pełny kod źródłowy przedstawiający opracowane rozwiązanie został zawarty w aneksie. W tej sekcji omówione zostaną najistotniejsze z wprowadzonych zmian względem przedstawionej wcześniej wersji, zakładającej jednowątkową realizację zapytań.

Jak omówiono w poprzedniej sekcji, jednym z głównych elementów wymagających dostosowania w celu umożliwienia wielowątkowej realizacji był obszar dotyczący aktualizacji modeli regresji. Wprowadzone zmiany zostały przedstawione na listingu 8.4 znajdującym się w aneksie pracy. Aby zapewnić poprawne działanie, należało zagwarantować, że dany model nie będzie aktualizowany w tym samym czasie przez więcej niż jeden wątek. Ponadto podczas aktualizacji modelu przez jeden z wątków, pozostałe wątki nie powinny w tym czasie odczytywać parametrów regresji w celu oszacowania kosztu zapytania w oparciu o dany model. Cel ten został osiągnięty poprzez utworzenie muteksu jako pola klasy `LRmodel.hpp` oraz blokowanie go za pomocą mechanizmu `scoped_lock` podczas aktualizacji współczynników oraz oceny modelu (metody (metody `update` oraz `evaluate`). Należy podkreślić, że wprowadzone zmiany nie generują istotnych kosztów czasowych podczas przetwarzania wielowątkowego. Algorytm aktualizacji oparty jest o szybkie algorytmy inkrementacyjne, co pozwala na minimalizację czasu trwania blokady nałożonej na dostęp do metody dla pozostałych wątków. Dodatkowo strategia przetwarzania (algorytm 7) zakłada utworzenie pięciu niezależnych modeli (`RAD`, `CTL`, `CTR`, `BVL`, `BVR`), co w znaczący sposób ogranicza ryzyko wstrzymywania pracy wielu wątków z powodu oczekiwania na zwolnienie blokady.

Oprócz zmian w zakresie aktualizacji modeli regresji wprowadzono również modyfikacje w samym algorytmie przetwarzania zapytań. Zmiany te zostały przedstawione na listingu 8.5 umieszczonym w aneksie pracy. Dotychczas algorytm `ROUNDROBIN` (metoda `m_round_robin`) oparty był na zmiennej `num_first_phase_choices`, która zliczała liczbę zapytań zrealizowanych z wykorzystaniem procedury `ROUNDROBIN`. Zmienna ta była polem klasy, a jej wartość była wielokrotnie aktualizowana, co stanowiło potencjalny problem podczas realizacji wielowątkowej. W celu zminimalizowania tego efektu metoda `m_round_robin` została zmodyfikowana poprzez zastąpienie zmiennej `num_first_phase_choices` dwoma oddzielnymi licznikami, zliczającymi zapytania re-

alizowane w trybie ROUNDROBIN osobno dla zapytań o rozmiarze $N \leq N_0$ oraz dla zapytań o rozmiarze $N > N_0$. Zarządzanie dostępem do tych liczników zostało zapewnione poprzez utworzenie dwóch niezależnych muteksów. W rezultacie proces podejmowania decyzji o tym, czy strategia realizacji zapytania powinna zostać wybrana w oparciu o procedurę ROUNDROBIN, czy OPTIMAL, odbywa się niezależnie dla zapytań z obu przedziałów rozmiaru. Takie podejście zmniejsza ryzyko wstrzymywania pracy wątków oczekujących na zwolnienie blokady. Należy również zauważyć, że ryzyko to jest i tak niewielkie, ponieważ średni czas realizacji zapytania jest znacznie dłuższy niż czas podejmowania decyzji o wyborze strategii.

Przebudowie uległ również fragment odpowiedzialny za obsługę przypadków, w których wskazana do realizacji zapytania strategia CT nie mogła zostać zastosowana z powodu przekroczenia dostępnej pamięci. Zadanie to realizowane jest przez metodę `m_CT_possible`, która podejmuje decyzję o możliwości realizacji zapytania $\text{COUNT}(X_i \mid Pa(X_i))$ z wykorzystaniem strategii CT na podstawie liczby możliwych stanów zmiennej X_i oraz liczby możliwych konfiguracji zmiennych Pa . W poprzedniej wersji metoda ta była bezargumentowa, a informacje o bieżącym zapytaniu pozyskiwała ze zmiennych `recent_pa_variables` oraz `recent_xi_variables`, będących polami klasy aktualizowanymi każdorazowo podczas realizacji zapytania. W celu umożliwienia realizacji wielowątkowej zdecydowano o usunięciu tych zmiennych jako pól klasy. W nowo opracowanej wersji metody wartości te są odczytywane i przekazywane do `m_CT_possible` w formie argumentów, nie pozostając w pamięci obiektu. Takie rozwiązanie eliminuje konieczność synchronizacji przy dostępie do danych związanych z bieżącym zapytaniem, a tym samym upraszcza obsługę wielowątkową.

Wprowadzone zmiany umożliwiają bezpieczną wielowątkową realizację zapytań z wykorzystaniem dowolnej metody zrównoleglania. Listing 6.2 przedstawia przykład wykorzystania standardu OpenMP, do zrównoleglenia realizacji strumienia zapytań Q na zbiorze danych D z wykorzystaniem opracowanej metody automatycznego wyboru strategii.

Przedstawiony kod stanowi analogię do listingu 5.1, omówionego w szczegółowy sposób w sekcji 5.9. Podobnie jak tam, w pierwszej kolejności następuje wczytanie z plików zbioru danych D (Listing 6.2, linie 19-24) oraz strumienia zapytań Q (Listing 6.2, linie 26-29). Następnie tworzony jest obiekt funkcyjny F , służący do zgłaszania wyników zrealizowanych zapytań. Utworzenie silnika związanego z metodą automatycznego wyboru strategii następuje poprzez wywołanie funkcji `create_counter` z odpowiednim argumentem (`Auto`). Jedyna różnica polega na tym, że realizacja strumienia zapytań zostaje zrównoleglona z wykorzystaniem standardu OpenMP (Listing 6.2, linia 36). Fragment ten odpowiada listingowi 3.3, omówionemu w szczegółowy sposób sekcji 3.3.5.

```

2  #include <vector>
3  #include <sabnatk.hpp>
4  #include <estimator.hpp>
5  #include "csv.hpp"
6  #include "read_queries.hpp"
7  #include <omp.h>
8
9  int main(int argc, char* argv[]) {
10     if (argc != 3) {
11         std::cout << "usage: random_query csv_file dat_file" << std::endl;
12         return -1;
13     }
14
15     const int N = 1;
16     using set_type = uint_type<N>;
17     using data_type = uint8_t;
18
19     std::vector<data_type> D;
20     auto [res, n, m] = read_csv(argv[1], D);
21     if (res == false) {
22         std::cout << "error: unable to read input data" << std::endl;
23         return -1;
24     }
25
26     std::vector<std::pair<set_type, set_type>> Q;
27     if (!read_queries(argv[2], Q)) {
28         std::cout << "error: unable to read queries" << std::endl;
29         return -1;
30     }
31
32     std::vector<estimator> F(1);
33
34     auto eng = sabnatk::create_counter<N, sabnatk::Auto>(n, m, std::begin(D));
35
36     #pragma omp parallel for shared(Q, eng), firstprivate(F) schedule(dynamic)
37     for (int i = 0; i < qs; ++i) eng.apply(Q[i].first, Q[i].second, F);
38
39     return 0;
40 }

```

Listing 6.2. Równoległa realizacja strumienia zapytań Q na zbiorze danych D z wykorzystaniem opracowanej metody automatycznego wyboru strategii oraz standardu OpenMP

6.3. Badanie wpływu opracowanej metody na wydajność zapytań zliczających

Podobnie jak w przypadku wcześniejszych badań, ocena opracowanej metody została przeprowadzona z wykorzystaniem rzeczywistej aplikacji uczenia maszynowego, związanej z uczeniem sieci bayesowskich. Jej istota została przedstawiona w sekcji 5.10, podczas omawiania wyników uzyskanych dla wersji metody zakładającej jednowątkową realizację zapytań.

W porównaniu do wcześniejszych eksperymentów, w niniejszych badaniach wykorzystano funkcjonalność solvera, umożliwiającą wielowątkowe przetwarzanie zapytań. Generowane przez solver zapytania są w tym przypadku automatycznie zrównoleglane w oparciu o bibliotekę Intel TBB (Threading Building Blocks) [29].

Do badań włączono również platformę obliczeniową firmy AMD. W rezultacie, opracowana metoda została zweryfikowana na przykładzie dwóch architektur - Intel Cascade Lake oraz AMD Milan, których specyfikację przedstawiono w sekcji 3.4.1.

Ze względu na ograniczenia systemu kolejkowego obowiązującego na platformie AMD Milan, limit czasu realizacji pojedynczego przypadku testowego został skrócony z 12 do czterech godzin, co odpowiada maksymalnemu dostępnemu limitowi na tej platformie.

W związku z tym, że badania dotyczą wersji zrównoleglonej, a zatem szybszej, ograniczenie to nie wpłynęło istotnie na zakres eksperymentów.

W przeprowadzonych eksperymentach zastosowano zbiory danych o liczności stu tysięcy obserwacji. W porównaniu z wynikami przedstawionymi w sekcji 5.10, jedyną istotną różnicą jest brak uwzględnienia zbioru Alarm. Jak przedstawiono w tabeli 5.4, przypadek ten generuje znacznie większą liczbę zapytań, co uniemożliwia jego realizację w przyjętym limicie czasu. Ponadto z badań wyłączono strategię ADT oraz HT, które jak pokazano we wcześniejszych eksperymentach, charakteryzują się niską wydajnością, a tym samym nie są w stanie ukończyć zadań w przewidzianym czasie.

Strategia	Zbiór danych (n)	Całkowity czas realizacji [s]	
		AMD Milan	Intel Cascade Lake
BV	Child (20)	167.13	838.00
	Insurance (27)	1210.42	6197.68
	Mildew (35)	1788.61	9275.16
	Barley (48)	7302.15	38489.10
RAD	Child (20)	135.97	290.42
	Insurance (27)	1461.15	3175.13
	Mildew (35)	382.83	860.44
	Barley (48)	1256.18	3091.74
CT	Child (20)	65.44	166.36
	Insurance (27)	*	*
	Mildew (35)	*	*
	Barley (48)	*	*
AC	Child (20)	11.20	74.32
	Insurance (27)	107.46	796.70
	Mildew (35)	46.95	248.45
	Barley (48)	202.47	973.32

* oznacza przekroczenie dostępnej pamięci operacyjnej

Tabela 6.1: Całkowity czas realizacji solvera wyznaczania zbioru rodziców (w sekundach)

Tabela 6.1 przedstawia podsumowanie eksperymentów, ukazując całkowity czas realizacji zadań dla wszystkich badanych strategii. Analiza wyników uzyskanych dla indywidualnych podejść (BV, RAD, CT) wskazuje, że ich wydajność w istotnym stopniu zależy od danych wejściowych. Przykładowo, w każdym z trzech zbiorów danych: Child, Insurance oraz Mildew, inna strategia okazywała się być najszybszą. Dla zbioru Child najlepsze wyniki uzyskano przy wykorzystaniu podejścia CT, dla Insurance była to strategia BV, natomiast dla Mildew – podejście RAD.

Warto podkreślić, że wydajność poszczególnych strategii zależy nawet od zastosowanej platformy obliczeniowej. Dla przykładu, w przypadku zbioru danych Insurance oraz platformy AMD najlepszą indywidualną strategią okazała się strategia BV, podczas gdy wyniki uzyskane dla tego samego zbioru danych na platformie Intel pokazują, że to strategia RAD była dwukrotnie szybsza od strategii BV.

Przechodząc do omówienia wyników uzyskanych dla opracowanej metody automatycznego wyboru strategii (AC), należy stwierdzić, że rozwiązuje ona wszystkie wskazane problemy, będąc najlepszym podejściem we wszystkich przypadkach testowych. Porównując wyniki z najlepszym możliwym indywidualnym podejściem (którego wybór jak pokazano nie jest trywialny), zastosowanie podejścia AC pozwoliło na uzyskanie skróce-

nie czasu realizacji od 5.85 do 11.26 razy na platformie AMD oraz od 2.29 do 3.99 razy na platformie Intel.

7. Metoda równoległej realizacji zapytań zliczających uwzględniająca architekturę systemów ccNUMA

Niniejsza sekcja przedstawia opracowaną metodę równoległej realizacji zapytań zliczających, uwzględniającą architekturę systemów obliczeniowych nowej generacji typu ccNUMA (cache-coherent Non-Uniform Memory Access). Systemy te składają się z wielu domen NUMA i charakteryzują się tym, że opóźnienia dostępu do pamięci zależą od fizycznej lokalizacji danych względem rdzenia wykonawczego. Opracowane rozwiązanie jest niezależne od wybranego podejścia do realizacji zapytań zliczających i może być stosowane z dowolnym z nich, w szczególności z opracowaną metodą automatycznego wyboru, przedstawioną w poprzednich rozdziałach. Opracowana metoda została zaprezentowana podczas konferencji Euro-Par 2025 oraz opublikowana w materiałach pokonferencyjnych [15].

7.1. Koncepcja opracowanego rozwiązania

Współczesne wysokowydajne systemy komputerowe oferują do ośmiu domen NUMA (2×4) w serwerach dwuprocesorowych opartych na procesorach AMD [7]. Zazwyczaj pojedyncza domena NUMA obejmuje grupę rdzeni procesora wraz z dedykowaną hierarchią pamięci podręcznej oraz kanałami pamięci, co umożliwia szybki, lokalny dostęp do przypisanej pamięci. W tego rodzaju systemach wydajność może ulec znacznemu obniżeniu w przypadku braku utrzymania lokalności danych. Z tego względu efektywne strategie zrównoleglenia muszą uwzględniać nie tylko zrównoważenie obciążenia obliczeniowego, ale również rozmieszczenie danych oraz przypisanie wątków do odpowiednich rdzeni, w celu minimalizacji kosztownych nielokalnych dostępu do pamięci [71].

Przeprowadzona analiza realizacji zapytań zliczających na systemach ccNUMA wykazała istnienie szeregu niejawnych ograniczeń utrudniających pełne wykorzystanie dostępnych domen NUMA. Podczas gdy podstawowe podejście do zrównoleglenia, zaprezentowane w Listingu 3.3 oraz 6.2, skutecznie równoważy obciążenie obliczeniowe, nie uwzględnia ono jednak topologii pamięci charakterystycznej dla systemów ccNUMA. Wraz ze wzrostem rozmiaru zbioru danych powyżej pojemności współdzielonych pamięci podręcznych procesora, dostęp do pamięci staje się dominującym czynnikiem ograniczającym wydajność. Analiza z wykorzystaniem narzędzia numastat [39] wykazała, że alokacja pamięci zachodzi głównie w obrębie jednej domeny NUMA. Wynika

to z polityki alokacji first-touch, która umieszcza dane w domenie NUMA najbliższej rdzeniowi procesora, na którym uruchomiony jest główny wątek OpenMP. W efekcie pozostałe domeny NUMA pozostają niewykorzystane, a ruch pamięciowy koncentruje się w jednym regionie, co ogranicza skalowalność obliczeń.

W celu rozwiązania wskazanego problemu zaprojektowano strategię wykonania strumienia zapytań dostosowaną do specyfiki architektur ccNUMA. Opracowane podejście, przedstawione w Listingu 7.1, wykorzystuje technikę zagnieżdżonego zrównoleglania (nested parallelism), umożliwiającą tworzenie regionów równoległych wewnątrz już istniejących, co pozwala na realizację wielopoziomowego zrównoleglania obliczeń. Pozwoliło to na poprawę lokalności danych poprzez organizację obliczeń w sposób odzwierciedlający hierarchiczną strukturę domen NUMA oraz rdzeni procesora.

```
1 processed_queries ← 0
2 #pragma omp parallel num_threads(NUMA_domains) {
3     eng ← create_counter(auto_scheduling, D)
4     omp_set_num_threads(cores_per_NUMA)
5     #pragma omp parallel shared(Q, eng, processed_queries) firstprivate(F) {
6         query_id ← 0
7         while (true) {
8             #pragma omp atomic capture
9             query_id = processed_queries++
10            if (query_id >= Q.size())
11                break
12            eng.apply(Q[query_id], F)
13        }
14    }
15 }
```

Listing 7.1. Koncepcja metody równoległej realizacji zapytań zliczających uwzględniająca architekturę systemów ccNUMA

W pierwszym etapie zrównoleglania (Listing 7.1, linia 2) tworzony jest region równoległy z liczbą wątków odpowiadającą liczbie domen NUMA w systemie docelowym. Każdy wątek przypisywany jest do odrębnej domeny NUMA przy wykorzystaniu mechanizmów zarządzania przypisaniem wątków w OpenMP. Następnie, w obrębie każdego wątku, tworzona jest osobna instancja silnika wykonującego zapytania, co wiąże proces alokacji pamięci z konkretną domeną NUMA.

W drugim etapie (Listing 7.1, linia 5) każdy wątek tworzy drugi poziom regionu równoległego, z liczbą wątków równą liczbie fizycznych rdzeni w odpowiadającej domenie NUMA. Proces przypisania wątków zagnieżdżonych do rdzeni procesora kontrolowany jest w sposób jawny za pomocą parametrów środowiskowych OpenMP, co zapewnia po-

prawne przypisanie wątków oraz zapobiega ich niepożądaney migracji pomiędzy domenami NUMA.

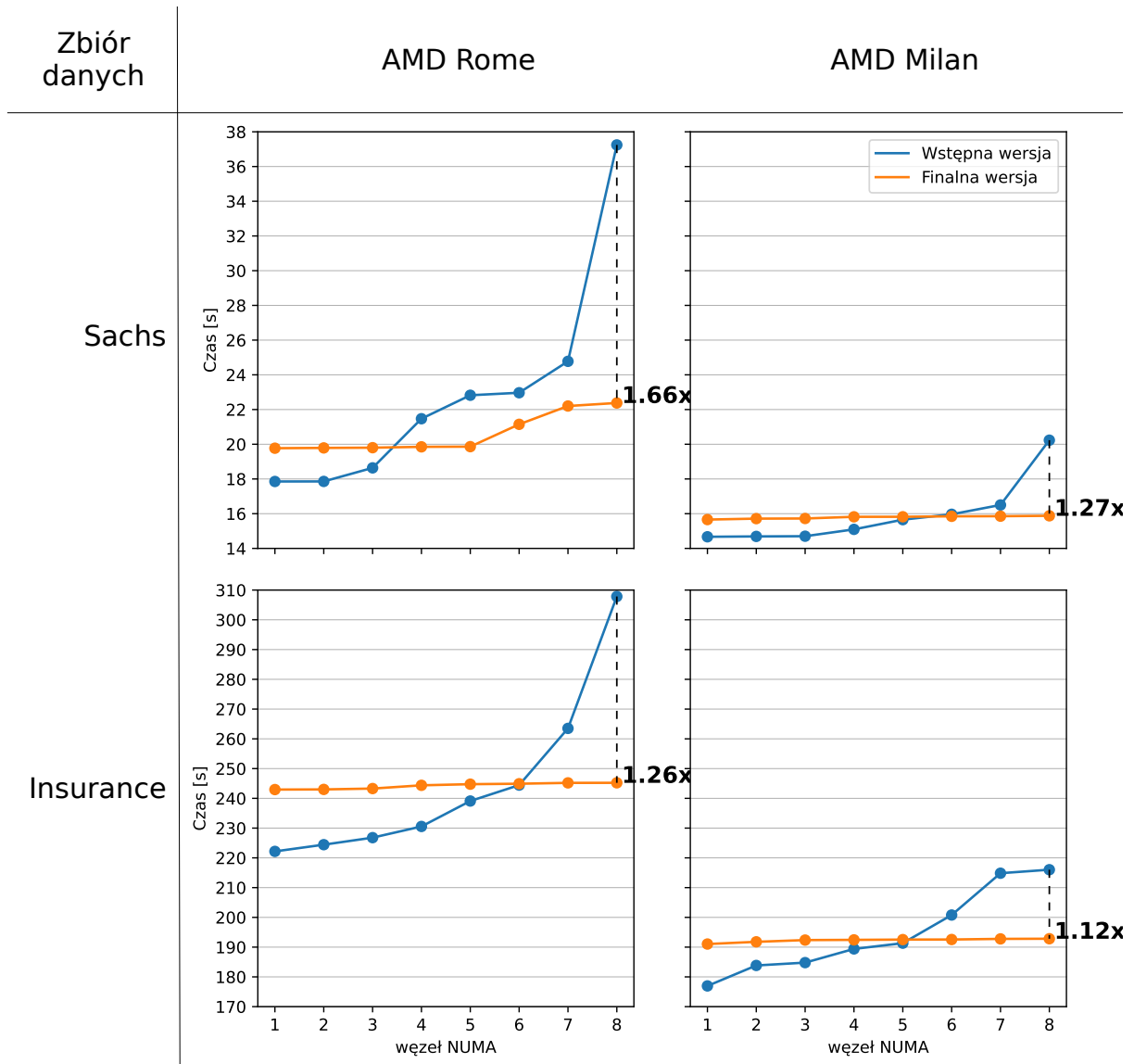
W celu rozdzielania strumienia zapytań pomiędzy wątki drugiego poziomu opracowano dynamiczny algorytm przydzielania zapytań. Algorytm ten przypisuje kolejne zapytania strumienia do dostępnych wątków na żądanie, zapewniając, że każdy bezczynny wątek natychmiast otrzymuje nowe zapytanie do przetworzenia. Takie podejście umożliwia efektywne zrównoważenie obciążenia oraz zapewnia wysokie wykorzystanie zasobów wszystkich domen NUMA przy jednoczesnym zachowaniu lokalności danych, ponieważ każdy wątek przetwarza dane alokowane w przypisanej domenie NUMA. W implementacji wykorzystano instrukcję atomową `atomic capture` (Listing 7.1, linia 8), która gwarantuje bezpieczny przydział kolejnych zapytań w środowisku wielowątkowym. Co istotne, pomimo zastosowania tego mechanizmu synchronizacji, nie zaobserwowano mierzalnego obniżenia wydajności, co wynika z naturalnej zmienności czasów wykonania poszczególnych zapytań, minimalizującej ryzyko blokowania wątków.

Listing 7.1 przedstawia finalną wersję opracowanej metody, która ewoluowała podczas przeprowadzonych eksperymentów. W szczególności, wstępna wersja metody nie uwzględniała opracowanego algorytmu dynamicznego przydzielania zapytań (Listing 7.1, linie 7–14). Zamiast tego wykorzystywano równomierny podział strumienia zapytań pomiędzy instancje silników utworzonych na poszczególnych domenach NUMA, a następnie realizowano ich przetwarzanie w oparciu o dostępne wątki z wykorzystaniem mechanizmu dynamicznego przydzielania zadań dostępnego w standardzie OpenMP. W rezultacie każda instancja silnika otrzymywała taką samą liczbę zapytań, których realizacja była zrównoleglana w ramach przypisanej domeny NUMA. Koncepcja tego podejścia została przedstawiona na Listingu 7.2.

```
1 #pragma omp parallel num_threads(NUMA_domains) {
2     eng ← create_counter(auto_scheduling, D)
3     qs_per_NUMA ← qs / NUMA_domains
4     thread_id = omp_get_thread_num()
5     start = qs_NUMA * thread_id
6     stop = qs_NUMA * (thread_id + 1)
7     omp_set_num_threads(cores_per_NUMA)
8     #pragma omp parallel for shared(Q, eng) firstprivate(F) schedule(dynamic) {
9         for(int i = 0; i < qs; ++i)
10             eng.apply(Q[i], F)
11     }
12 }
```

Listing 7.2. Wstępna wersja metody równoległej realizacji zapytań zliczających uwzględniającej architekturę systemów ccNUMA

Przeprowadzone badania wykazały jednak, że takie podejście prowadzi do istotnego nierównoważenia czasów przetwarzania pomiędzy poszczególnymi domenami NUMA. Wynika to z faktu, że pomimo równej liczby zapytań przypisanych do każdej domeny, poszczególne zapytania mogą charakteryzować się odmienną złożonością obliczeniową (np. liczbą zmiennych czy liczbą możliwych stanów). Przykłady tych dysproporcji przedstawiono na rysunku 7.1. Przedstawia on czasy przetwarzania zapytań przez poszczególne domeny NUMA w przypadku zastosowania wstępnej wersji metody (Listing 7.2), opartej na mechanizmie dynamicznego przydzielania zadań standardu OpenMP, oraz finalnej wersji (Listing 7.1), wykorzystującej opracowany algorytm dynamicznego przydzielania zapytań. Jak wynika z rysunku, zastosowanie opracowanego algorytmu pozwoliło na równomierne rozłożenie obciążenia pomiędzy domeny NUMA, co w rezultacie przyczyniło się do istotnego skrócenia całkowitego czasu przetwarzania.



Rysunek 7.1. Czasy przetwarzania zapytań przez poszczególne domeny NUMA w przypadku zastosowania wstępnej oraz finalnej wersji opracowanej metody

7.2. Kod źródłowy opracowanej metody

Przykład zastosowania opracowanego podejścia został przedstawiony w aneksie pracy na listingu 8.6. Kod ten demonstruje wykorzystanie zaproponowanej metody w przypadku równoległej realizacji strumienia losowych zapytań, której koncepcja została omówiona we wcześniejszych sekcjach, w szczególności na listingu 6.2.

Uruchomienie zaprezentowanego rozwiązania na konkretnej architekturze wymaga uprzedniego zdefiniowania liczby domen NUMA oraz liczby rdzeni w każdej domenie. Przykładowo, dla rozpatrywanych w niniejszym rozdziale platform obliczeniowych opartych o procesory AMD Rome oraz Milan, parametry te przyjmują wartości: `NUMA_domains = 8` oraz `cores_per_NUMA = 16`, co łącznie daje 128 rdzeni wykorzystywanych w oblicze-

niach.

Dodatkowo, w celu zapewnienia pożądanego przez metodę rozmieszczenia wątków oraz zapobiegnięcia ich migracji pomiędzy domenami NUMA, należy ustawić odpowiednie zmienne środowiskowe przed uruchomieniem aplikacji:

```
export OMP_PROC_BIND=spread,close  
export OMP_PLACES=threads
```

7.3. Badanie wpływu opracowanej metody na wydajność zapytań zliczających

Opracowana metoda została przetestowana na przykładzie benchmarku losowych zapytań, którego koncepcję przedstawiono w sekcji 3.4.3. Do badań wybrano najbardziej wymagające zbiory danych zawierające dzisiaj miliony obserwacji. Tak duża liczba obserwacji stanowi warunek konieczny działania opracowanej metody, ponieważ zapewnia, że dane dotyczące obserwacji nie mieszczą się we współdzielonej pamięci podręcznej procesorów, co w rezultacie powoduje, że dostęp do pamięci staje się dominującym czynnikiem ograniczającym wydajność. Warto podkreślić, że nie jest to scenariusz sztuczny, gdyż rzeczywiste aplikacje analityczne często operują na bazach danych o porównywalnej lub większej skali [65].

W celu oceny wydajności opracowanego rozwiązania dla każdego zbioru danych przygotowano strumień losowych zapytań, obejmujący dzisiaj tysiące zapytań zliczających. W każdym przypadku strumień ten przetworzono przy użyciu strategii automatycznego wyboru zapytań (AC) dwukrotnie: raz w oparciu o standardowe podejście do równoległej realizacji zapytań z wykorzystaniem OpenMP (listing 6.2), oraz raz stosując opracowaną metodę równoległej realizacji zapytań uwzględniającej architekturę systemów ccNUMA (listing 8.6)

Badania przeprowadzono na platformach obliczeniowych firmy AMD: Rome oraz Milan. Obie platformy zapewniają osiem domen NUMA, z których każda składa się z szesnastu rdzeni. Specyfikacja obu platform została przedstawiona w sekcji 3.4.1.

Platforma obliczeniowa	Zbiór danych (n)	Całkowity czas realizacji [s]		Przyspieszenie
		AC	<i>ccNUMA</i> -AC	
AMD Rome	Sachs (11)	64.6	24.6	2.63
	Child (20)	837.1	189.0	4.43
	Insurance (27)	2305.9	250.9	9.19
	Mildew (35)	3017.7	399.3	7.56
	Alarm (37)	3377.7	329.4	10.25
	Barley (48)	3343.6	993.1	3.37
AMD Milan	Sachs (11)	48.3	17.6	2.74
	Child (20)	339.5	160.4	2.12
	Insurance (27)	782.8	196.9	3.98
	Mildew (35)	1564.4	331.3	4.72
	Alarm (37)	1482.2	268.8	5.51
	Barley (48)	1856.4	446.2	4.16

Tabela 7.1: Porównanie wydajności metody automatycznego wyboru strategii w benchmarku losowych zapytań w oparciu o standardowe podejście do równoległej realizacji zapytań z wykorzystaniem OpenMP oraz opracowaną metodę równoległej realizacji zapytań uwzględniającej architekturę systemów ccNUMA

Tabela 7.1 przedstawia podsumowanie eksperymentów, ukazując całkowity czas realizacji strumienia zapytań dla obu ewaluowanych podejść. We wszystkich przypadkach opracowana strategia uwzględniająca architekturę systemów ccNUMA znacząco przewyższa podejście bazowe pod względem wydajności. Najwyższe osiągnięte przyspieszenia wyniosły odpowiednio $10.25\times$ na platformie Rome oraz $5.51\times$ na platformie Milan. Uzyskane wyniki potwierdzają, że opracowana metoda może w istotny sposób poprawić wydajność wielu aplikacji uczenia maszynowego.

8. Podsumowanie oraz kierunki dalszych badań

8.1. Wnioski końcowe

Praca dotyczy poszukiwania metod poprawy efektywności obliczeniowej i energetycznej uczenia maszynowego oraz aplikacji numerycznych w środowisku wysokowydajnych systemów komputerowych. W pracy uzyskano następujące oryginalne wyniki w tym zakresie:

1. Opracowano metodę redukcji zużycia energii elektrycznej dedykowaną dla aplikacji zrównoleglanych w modelu równoległości danych. Głównym celem opracowanej metody jest złagodzenie skutków nierównomiernej dystrybucji obliczeń pomiędzy dostępne wątki lub rdzenie procesorów, powstającej na skutek zrównoleglania tego typu aplikacji w oparciu o powszechnie stosowane podejścia. Istotą metody jest zarządzanie częstotliwością odpowiednio dobranych grup rdzeni, co pozwala na lepsze dopasowanie parametrów pracy procesora do rzeczywistych obciążeń. Opracowana metoda adresuje problem nierównomiernej dystrybucji obliczeń, pozwalając na istotną redukcję zużycia energii bez jednoczesnej utraty wydajności. Proponowane podejście przewyższa efektywnością standardowo stosowane techniki, takie jak DVFS, zapewniając lepsze dopasowanie do charakterystyki danej aplikacji.
2. Zaproponowano algorytmy wyboru heterogenicznej konfiguracji częstotliwości, umożliwiające praktyczne wdrożenie opracowanej metody redukcji zużycia energii w aplikacjach zrównoleglanych w modelu równoległości danych. Opracowane algorytmy istotnie ograniczają liczbę potrzebnych konfiguracji testowych, przyspieszając proces poszukiwania optymalnej konfiguracji częstotliwości. Pierwszy z algorytmów podejmuje decyzje na podstawie pomiarów zużycia energii, natomiast drugi eliminuje potrzebę takich pomiarów, opierając się jedynie na analizie czasu wykonania aplikacji. Takie podejście sprawia, że opracowana metoda redukcji zużycia energii może być z powodzeniem wdrożona nawet w środowiskach, w których pomiar energii jest utrudniony a nawet niemożliwy.
3. Zaprojektowano oraz zaimplementowano dwie metody poprawy wydajności strategii realizacji zapytań zliczających opartej o reprezentowanie stanów poszczególnych zmiennych zapytania za pomocą bitmap oraz algorytm przeszukiwania

w głąb (DFS, Depth First Search). Pierwsza z zaproponowanych metod ukierunkowana jest na redukcję liczby wykonywanych przecięć bitmap (operacji AND), natomiast druga zmierza do minimalizacji liczby odwiedzanych węzłów drzewa. Opracowane metody zostały wdrożone przez autora niniejszej pracy w bibliotecę SABNAtk, będącej narzędziem służącym do efektywnej realizacji zapytań zliczających.

4. Opracowano metodę automatycznego wyboru strategii realizacji zapytań zliczających, której celem jest skrócenie czasu obsługi strumienia zapytań. Istota podejścia polega na połączeniu zalet trzech komplementarnych strategii — BV, RAD i CT. Metoda wykorzystuje regresję online jako mechanizm decyzyjny, który dla każdego zapytania w strumieniu wybiera strategię o najniższym estymowanym czasie wykonania. Do oryginalnych osiągnięć związanych z opracowaniem tej metody należą:
 - (a) Sformułowanie teoretycznych modeli kosztu realizacji zapytania zliczającego dla trzech różnych strategii, będące efektem dogłębnej analizy poszczególnych podejść. Opracowane modele estymują koszt wykonania zapytania na podstawie właściwości samego zapytania jak i rozmiaru zbioru danych.
 - (b) Opracowanie efektywnego algorytmu aktualizacji modeli kosztu realizacji zapytania zliczającego. Rozwiązanie to bazuje na algorytmach inkrementalnych, umożliwiając efektywną aktualizację statystyk modeli regresji na podstawie pojedynczych nowych obserwacji, bez konieczności gromadzenia dotychczasowych danych i bez narastania kosztu obliczeniowego.
 - (c) Zaprojektowanie algorytmu przetwarzania strumienia zapytań, który definiuje pełny przepływ działania metody — od inicjalizacji i uczenia modeli, przez adaptacyjny wybór strategii, po obsługę zapytań przekraczających ograniczenia pamięciowe, zapewniając spójne i praktyczne ujęcie opracowanego podejścia.
 - (d) Wdrożenie opracowanej metody w oficjalnej wersji biblioteki SABNAtk, co umożliwia jej praktyczne wykorzystanie i udostępnienie szerszemu gronu użytkowników.
5. Opracowano i zaimplementowano w pełni bezpieczną dla przetwarzania wielowątkowego (thread-safe) wersję metody automatycznego wyboru strategii realizacji zapytań zliczających. Wdrożone podejście minimalizuje ilość operacji zapisu współdzielonego oraz wykorzystuje mechanizm muteksów, skutecznie zapobiegając sytuacjom wyścigu pomiędzy wątkami podczas równoległego przetwarzania. Taka

implementacja umożliwia zastosowanie modelu równoległości zadań (task parallelism), w którym zapytania zliczające traktowane są jako niezależne zadania realizowane równolegle na wielu rdzeniach procesora, przy użyciu dowolnego narzędzia (np. OpenMP, Intel TBB).

6. Zaproponowano metodę równoległej realizacji zapytań zliczających, dedykowaną architekturom nowej generacji typu ccNUMA (cache-coherent Non-Uniform Memory Access). Systemy te składają się z wielu domen NUMA i charakteryzują się zróżnicowanymi opóźnieniami dostępu do pamięci w zależności od fizycznej lokalizacji danych względem rdzenia wykonawczego. Współczesne wysokowydajne systemy komputerowe oferują do ośmiu domen NUMA (2×4) w serwerach dwuprocessorowych opartych na procesorach AMD [7]. Opracowane rozwiązanie jest niezależne od wybranej strategii realizacji zapytań zliczających i może być stosowane z dowolnym podejściem, w szczególności z metodą automatycznego wyboru strategii realizacji zapytań zliczających.

Przeprowadzone badania eksperymentalne pozwoliły sformułować następujące wnioski będące również oryginalnymi osiągnięciami autora:

1. Wykazano poprawność zasadniczej tezy pracy mówiącej o możliwości opracowania metod i algorytmów prowadzących do poprawy efektywności obliczeniowej i energetycznej aplikacji numerycznych oraz obliczeń uczenia maszynowego osiąganą w środowisku wysokowydajnych systemów komputerowych.
2. Badania eksperymentalne wykonane na platformie Intel Xeon Gold 6240 (Cascade Lake) potwierdziły skuteczność opracowanej metody redukcji zużycia energii, dedykowanej dla aplikacji zrównoleglanych w modelu równoległości danych. Testy przeprowadzone na dwóch różnych aplikacjach numerycznych wykazały, że proponowane podejście heterogeniczne przewyższa tradycyjne, homogeniczne skalowanie częstotliwości. Przykładowo, najlepsza zaobserwowana redukcja zużycia energii w aplikacji dyfuzji wyniosła około 25%, podczas gdy w tym samym teście powszechnie stosowane podejście homogeniczne zapewniło jedynie 9%. Ponadto, we wszystkich testach opracowane podejście heterogeniczne uzyskało lepsze wyniki energetyczne, a największa przewaga nad podejściem tradycyjnym wyniosła około 20 punktów procentowych.
3. Skuteczność opracowanej metody redukcji zużycia energii, dedykowanej dla aplikacji zrównoleglanych w modelu równoległości danych została również potwierdzona z wykorzystaniem platformy AMD EPYC 7773X (Milan-X). Istotność tego osiągnięcia podkreśla fakt, że procesory te oferują znacznie bardziej ograniczony

zakres zarządzania częstotliwością w porównaniu do układów Intela — dostępne są jedynie cztery poziomy: 1.5 GHz, 1.9 GHz, 2.2 GHz oraz tryb turbo do 3.5 GHz. Mimo tych ograniczeń opracowana metoda heterogeniczna przewyższyła tradycyjne podejście homogeniczne, zapewniając dodatkowe oszczędności energetyczne rzędu 13 punktów procentowych dla aplikacji trójwymiarowej dyfuzji oraz 12 punktów procentowych dla aplikacji MPDATA.

4. Przeprowadzone badania wykazały, że opracowane algorytmy wyboru heterogenicznej konfiguracji częstotliwości charakteryzują się wysoką efektywnością, definiowaną jako procentowy stosunek redukcji zużycia energii uzyskanej przez dany algorytm do najlepszego wyniku wyznaczonego metodą „brute force”. Uśredniając wyniki dla obu analizowanych aplikacji numerycznych, skuteczność algorytmu opartego na pomiarach energii wyniosła 97,5%, natomiast algorytmu wykorzystującego wyłącznie pomiary czasu — 96%. Dodatkową zaletą opracowanych algorytmów jest istotne skrócenie czasu potrzebnego na wyznaczenie rozwiązania: podczas gdy podejście „brute force” wymagało każdorazowo analizy 45 konfiguracji częstotliwości, proponowane algorytmy osiągały ten sam cel, badając jedynie 7–11 konfiguracji.
5. Badania eksperymentalne przeprowadzone na trzech różnych platformach obliczeniowych potwierdziły skuteczność wdrożonych metod poprawy wydajności strategii realizacji zapytań BV, opartej na algorytmie przeszukiwania w głąb. Uzyskane przyspieszenie przetwarzania całego strumienia zapytań — wynikające z redukcji liczby przecięć bitmap oraz ograniczenia liczby odwiedzanych węzłów drzewa — mieści się, w zależności od analizowanego zbioru danych, w przedziale od 1.09 do 2.25 razy.
6. Skuteczność opracowanej metody automatycznego wyboru strategii realizacji zapytań zliczających (AC) została potwierdzona na przykładzie benchmarku losowych zapytań oraz rzeczywistej aplikacji uczenia maszynowego. Podejście zostało porównane z powszechnie wykorzystywanymi strategiami (ADT, HT) oraz tymi, spośród których dokonuje wyboru (tj. BV, RAD oraz CT). Przykładowo, przytaczając wyniki uzyskane dla aplikacji uczenia sieci Bayesowskich, opracowana metoda automatycznego wyboru strategii okazała się najlepszym podejściem we wszystkich przypadkach testowych, oferując istotne przyspieszenie względem pozostałych strategii realizacji zapytań. W porównaniu do ADT, AC działała od $3.3\times$ do $310\times$ szybciej, natomiast przyspieszenie względem HT było jeszcze większe i wynosiło od $68\times$ do $19418\times$. Ponadto, opracowana metoda oferowała znaczną poprawę w stosunku do strategii, na których bazuje. Przykładowo, analizując wyniki dla zbioru

Child w wariancie $m = 10K$, w którym najlepszą indywidualną strategią okazała się BV, strategia AC poprawiła ten wynik $4.61\times$. Uzyskane przyspieszenia przekładają się w bezpośredni sposób na wydajność aplikacji, ponieważ jak pokazano w badaniach udział zapytań zliczających przekracza 95% całkowitego czasu realizacji całej aplikacji.

7. Przeprowadzone badania wersji umożliwiającej wielowątkową realizację metody automatycznego wyboru strategii realizacji zapytań zliczających, przeprowadzone na przykładzie aplikacji uczenia sieci Bayesowskich, potwierdziły skuteczność proponowanej koncepcji w obliczeniach wielowątkowych. Analiza wyników uzyskanych na dwóch platformach obliczeniowych — Intel Xeon Gold 6240 (Cascade Lake) oraz AMD EPYC 7763 (Milan) — wykazała, że wydajność poszczególnych strategii indywidualnych (BV, RAD, CT) zależy nie tylko od zbioru danych, ale nawet od zastosowanej architektury sprzętowej. Przykładowo, w przypadku zbioru danych Insurance oraz platformy AMD najlepszą indywidualną strategią okazała się strategia BV, podczas gdy wyniki uzyskane dla tego samego zbioru danych na platformie Intel pokazują, że strategia RAD była dwukrotnie szybsza od BV. Opracowana metoda automatycznego wyboru strategii (AC) skutecznie rozwiązuje ten problem, dobierając optymalne strategie do poszczególnych zapytań strumienia. W rezultacie czas realizacji zapytań skrócono od $5.85\times$ do $11.26\times$ na platformie AMD oraz od $2.29\times$ do $3.99\times$ na platformie Intel, w porównaniu z najlepszą możliwą indywidualną strategią, której wybór, jak pokazano, nie jest trywialny. Wyniki te potwierdzają nie tylko skuteczność metody, ale także jej uniwersalność i stabilność w różnych środowiskach obliczeniowych.
8. Badania eksperymentalne wykazały, że opracowana metoda równoległej realizacji zapytań zliczających, zaprojektowana z myślą o architekturach nowej generacji typu ccNUMA, skutecznie eliminuje główne ograniczenia wynikające z ich specyfiki. W odróżnieniu od klasycznego podejścia opartego na OpenMP, które nie uwzględnia topologii pamięci i prowadzi do koncentracji ruchu w jednej domenie NUMA, zaproponowane rozwiązanie zapewnia równomierne wykorzystanie wszystkich dostępnych domen. Dzięki dwupoziomowej organizacji równoległości oraz kontroli przydziału wątków i danych do poszczególnych regionów NUMA, metoda znacząco poprawia lokalność pamięci i redukuje koszty zdalnych dostępow. W efekcie umożliwia pełniejsze wykorzystanie potencjału nowoczesnych platform obliczeniowych, oferując przyspieszenia sięgające $10.25\times$ (AMD Rome) i $5.51\times$ (AMD Milan) względem podejścia bazowego.
9. Przeprowadzone badania wykazały również, że uzyskanym przyspieszeniom towa-

rzyszą proporcjonalne redukcje zużycia energii, wynikające ze skrócenia całkowitego czasu pracy aplikacji. Oznacza to, że opracowane metody przyczyniają się nie tylko do zwiększenia wydajności, ale także do poprawy efektywności energetycznej obliczeń w obszarze uczenia maszynowego.

Tabela 8.1 przedstawia powiązanie opracowanych metod z odpowiadającymi im publikacjami.

Metoda	Rozdział	Publikacja	Punktacja wg wykazu MNiSW
Metoda redukcji zużycia energii elektrycznej dla aplikacji zrównoleglanych w modelu równoległości danych	2	[14] [13]	140 100
Metody poprawy wydajności strategii Bitmap	4	[12]	140
Metoda automatycznego wyboru strategii	5	[16]	140
Równoległa realizacja metody automatycznego wyboru strategii	6	[17]	140
Metoda równoległej realizacji zapytań zliczających uwzględniająca architekturę systemów ccNUMA	7	[15]	140

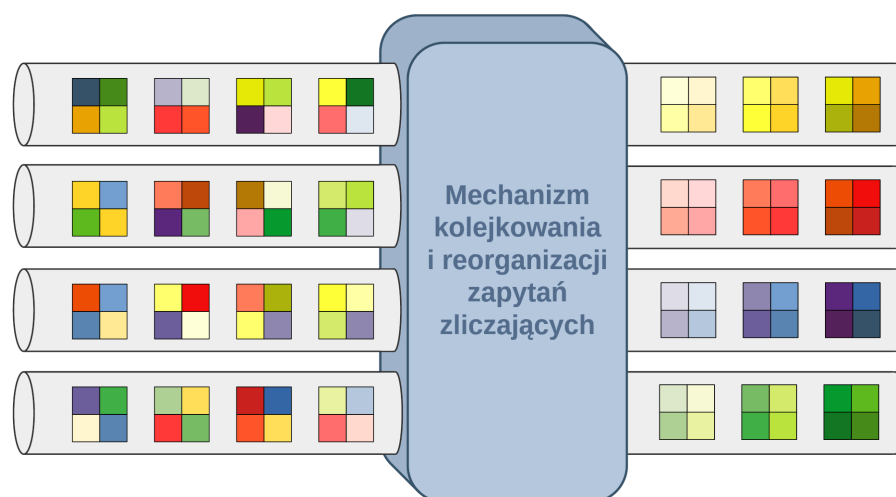
Tabela 8.1: Publikacje autora związane z metodami opracowanymi w ramach niniejszej rozprawy doktorskiej

8.2. Kierunki dalszych prac

Plany na dalsze badania autora niniejszej rozprawy koncentrują się wokół tematyki efektywnej realizacji zapytań zliczających. W szczególności, jako potencjalne kierunki rozwoju można wskazać trzy obszary:

1. *Eksploracja możliwości wykorzystania procesorów graficznych (GPU)*

Architektury GPU zaprojektowane są z myślą o masywnie równoległym wykonywaniu prostych operacji na dużych zbiorach danych. Wysoka liczba rdzeni oferowana przez tego typu procesory dobrze koresponduje z charakterem przetwarzania zapytań zliczających. Opracowywanie metod i algorytmów umożliwiających efektywną realizację zapytań na GPU stanowi naturalny kierunek dalszych badań.



Rysunek 8.1. Wizualizacja koncepcji mechanizmu kolejowania i reorganizacji zapytań zliczających

2. *Badanie potencjału architektury RISC-V*

RISC-V [53] to otwarta architektura zestawu instrukcji wywodząca się z rodziny procesorów typu RISC (Reduced Instruction Set Computer). Jej elastyczność oraz rosnąca popularność w środowiskach akademickich i przemysłowych [43] sprawiają, że stanowi ona interesującą platformę badawczą. Efektywna realizacja zapytań zliczających na procesorach opartych o architekturę RISC-V stanowi jeden z kierunków przyszłych badań.

3. *Opracowanie mechanizmu kolejowania i reorganizacji zapytań*

W praktycznych zastosowaniach zapytania zliczające często mają części wspólne, co prowadzi do wielokrotnego przetwarzania tych samych danych. Jednym z kierunków przyszłych badań jest opracowanie mechanizmu kolejowania i reorganizacji zapytań (rysunek 8.1), który pozwoliłby na identyfikację i izolowanie wspólnych fragmentów wielu zapytań, a następnie ich jednorazowe przetwarzanie. Takie podejście może ograniczyć nadmiarowe operacje dostępu do danych a tym samym poprawić efektywność realizacji zapytań w środowiskach wielowątkowych.

Aneks

```
1 // Fragment pliku BVCounter.hpp przedstawiajacy wprowadzone zmiany zwiazane z
   // opracowanymi metodami SNIJ i SNIJK
2 // Autor: Pawel Bratek <pawel.bratek@pcz.pl>
3
4 template <typename score_functor>
5 void apply(const std::vector<int>& xi_vect, const set_type& pa, std::vector<
   score_functor>& F) const {
6     int qpa = m_q_(pa);
7     for (int i = 0; i < F.size(); ++i) F[i].init(r(xi_vect[i]), qpa);
8
9     if (is_emptyset(pa)) {
10         for (int i = 0; i < F.size(); ++i) {
11             F[i](m_);
12             for (const vect& v : data_[xi_vect[i]].first) {
13                 F[i](v.weight(), m_);
14             }
15             F[i].finalize(1);
16         }
17         return;
18     }
19
20     using Iter = typename std::vector<vect>::const_iterator;
21
22     std::vector<int> pa_sorted;
23
24     for (int i : sorted_order_) {
25         if (in_set(pa, i)) {
26             pa_sorted.push_back(i);
27         }
28     }
29
30     int pa_size = pa_sorted.size();
31     int F_size = F.size();
32
33     // the table stores pairs with
34     // first = current bitvector at that level of the dfs traversal
35     // second = iterator over bitvectors of next parent to intersect with
36     std::vector<std::pair<vect, Iter>> table;
37     table.reserve(pa_size + 1);
38
39     for (int i : pa_sorted) {
```

```

40         // start with identity
41         table.push_back(std::make_pair(vect::identity(m_), std::begin(data_[i].
           first)));
42     }
43
44     std::vector<int> SNij (pa_size, 0); // storing the sum of reported Nij for
           each layer of the tree
45
46     // this is to store results needed at the bottom of the tree
47     // in other words this will be intersection on the path from root to leaf
48     // the second argument has no meaning hence std::end
49     table.push_back(std::make_pair(std::move(vect::identity(m_)), std::end(data_
           [0].first)));
50
51     int layer = 0; // we start at the top of the tree
52     int qi_obs = 0;
53
54     vect container = vect::identity(m_); // bottom layer
55
56     while (true) { // once we are finished, we will try to move one layer up from
           root
57
58         while (layer >= 0 && table[layer].second == std::end(data_[pa_sorted[layer]
           ]).first)) {
59             // reset the sum of Nij and the iterator for next traversal at this
           level
60             SNij[layer] = 0;
61             table[layer].second = std::begin(data_[pa_sorted[layer]].first);
62             --layer; // move up to continue traversal
63         }
64
65         if (layer < 0) break;
66
67         const vect& root = table[layer].first; // fetch from the table
68
69         intersect(root, *(table[layer].second), table[layer + 1].first);
70         ++table[layer].second;
71
72         int Nij = table[layer + 1].first.weight();
73         SNij[layer] += Nij;
74
75         int SNijk = 0; // storing the sum of reported Nijk
76
77         if (Nij) { // if Nij is zero we just move on
78             if (layer == (pa_size - 1)) { // we have explored all of the parents
79                 for (int i = 0; i < F_size; ++i) F[i](Nij);

```

```

80
81         ++qi_obs;
82
83         for (int i = 0; i < F_size; ++i) {
84             for (const vect& v : data_[xi_vect[i]].first) {
85                 intersect(table[layer + 1].first, v, container);
86
87                 int Nijk = container.weight();
88                 SNijk += Nijk;
89
90                 if (Nijk) {
91                     F[i](Nijk, Nij);
92                     if (SNijk == Nij) break; // we have made all the
                                           necessary calls
93                 }
94             }
95         } // for i
96     } else if (Nij == 1) {
97         // if we go down a level, we will find that this is the only call
           that would be made
98         for (int i = 0; i < F_size; ++i) F[i](Nij);
99         ++qi_obs;
100        for (int i = 0; i < F_size; ++i) F[i](1, 1);
101    } else { // there are more parents
102        ++layer; // move down a layer and keep going
103    }
104 } // if (Nij)
105
106 // we have made all the necessary calls
107 if (SNij[layer] == table[layer].first.weight()) {
108     table[layer].second = std::end(data_[pa_sorted[layer]].first);
109 }
110 } // while (true)
111
112 for (int i = 0; i < F_size; ++i) F[i].finalize(qi_obs);
113 } // apply

```

Listing 8.1. Kod źródłowy przedstawiający wprowadzone zmiany związane z opracowanymi metodami SNIJ i SNIJK

```

1 // Plik: LRmodel.hpp
2 // Autor: Pawel Bratek <pawel.bratek@pcz.pl>
3
4 #ifndef LR_MODEL_HPP
5 #define LR_MODEL_HPP
6

```

```

 7  #include <cmath>
 8  #include "counters.hpp"
 9
10
11  namespace sabnatk {
12
13      namespace detail {
14
15          class LRmodel {
16          public:
17              explicit LRmodel(counter_type ct = Rad, bool il = true)
18                  : counter_(ct), is_linear_(il) { }
19
20              int n() const { return n_; }
21
22              counter_type counter() const { return counter_; }
23
24              double a() const { return cov_xy_ / var_x_; }
25
26              double b() const { return mean_y_ - a() * mean_x_; }
27
28              double evaluate(int x, double K) const {
29                  if (is_linear_) return a() * x + b();
30                  return std::pow(K, b()) * std::pow(K, (a() * x));
31              } // evaluate
32
33              void update(int x, long long int y, double K) {
34                  n_++;
35
36                  double dx = x - mean_x_;
37
38                  var_x_ += ((dx * dx * (n_ - 1) / n_) - var_x_) / n_;
39                  mean_x_ += dx / n_;
40
41                  double dy = is_linear_ ? y : std::log(y) / std::log(K);
42
43                  dy -= mean_y_;
44                  cov_xy_ += ((dx * dy * (n_ - 1) / n_) - cov_xy_) / n_;
45                  mean_y_ += dy / n_;
46              } // update
47
48          private:
49              counter_type counter_;
50
51
52              bool is_linear_;

```

```

53
54         int n_ = 0;
55
56         double mean_x_ = 0.0;
57         double var_x_ = 0.0;
58         double mean_y_ = 0.0;
59         double cov_xy_ = 0.0;
60
61     }; // class LRmodel
62
63 } // namespace detail
64
65 } // namespace sabnatk
66
67 #endif // LR_MODEL_HPP

```

Listing 8.2. Kod źródłowy przedstawiający model kosztu realizacji zapytania

```

1 // Plik: AutoCounter.hpp
2 // Autor: Paweł Bratek <pawel.bratek@pcz.pl>
3
4 #ifndef AUTO_COUNTER_HPP
5 #define AUTO_COUNTER_HPP
6
7 #include "counters.hpp"
8
9 #include "BVCounter.hpp"
10 #include "CTCounter.hpp"
11 #include "RadCounter.hpp"
12
13 #include "LRmodel.hpp"
14
15 #include "estimator.hpp"
16 #include <random>
17 #include <chrono>
18
19
20 namespace sabnatk {
21
22     template <int N, typename Data = uint8_t> class AutoCounter {
23     public:
24         using set_type = uint_type<N>;
25         using data_type = Data;
26
27         counter_type type() const { return Auto; }
28

```

```

71         }
72     } else {
73         if (Rmodels_to_train_size_ > 0) {
74             model = m_round_robin__(Rmodels_to_train_,
75                                     Rmodels_to_train_size_, Rmodels_trained_,
76                                     Rmodels_trained_size_, Npa);
77         } else {
78             model = m_select_best__(Rmodels_trained_,
79                                     Rmodels_trained_size_, Npa);
80         }
81     }
82
83     all_models_trained_ = Lmodels_to_train_size_ == 0 &&
84                           Rmodels_to_train_size_ == 0;
85
86     } else {
87         if (Npa <= NO_) {
88             model = m_select_best__(Lmodels_trained_, Lmodels_trained_size_,
89                                     Npa);
90         } else {
91             model = m_select_best__(Rmodels_trained_, Rmodels_trained_size_,
92                                     Npa);
93         }
94     }
95
96     ty = m_run_query__(model, xi, pa, F);
97
98     model->update(Npa, ty, K_);
99
100 } // apply
101
102 private:
103     RadCounter<N> rad_;
104     BVCounter<N> bv_;
105     CTCCounter<N> ct_;
106
107     detail::LRmodel model_rad_;
108     detail::LRmodel model_bv1_, model_bv2_;
109     detail::LRmodel model_ct1_, model_ct2_;
110
111     // two groups (L, R) of models based on NO
112     detail::LRmodel* Lmodels_to_train_[3];
113     detail::LRmodel* Rmodels_to_train_[3];
114     int Lmodels_to_train_size_;
115     int Rmodels_to_train_size_;
116
117     detail::LRmodel* Lmodels_trained_[3];

```

```

111     detail::LRmodel* Rmodels_trained_[3];
112     int Lmodels_trained_size_;
113     int Rmodels_trained_size_;
114
115     bool all_models_trained_;
116
117     // query statistics
118     double K_;
119     int NO_;
120
121     int num_first_phase_choices_;
122
123     int required_instances_; // the minimum number of data required to use the
                             // model during auto selection
124     long long unsigned int CT_memory_limit_;
125
126     std::vector<int> recent_pa_variables_;
127     std::vector<int> recent_xi_variables_;
128
129
130     detail::LRmodel* m_round_robin__(detail::LRmodel** models_to_train, int&
                                     to_train_size,
131                                     detail::LRmodel** models_trained, int&
                                     trained_size, int Npa) {
132         int idx = num_first_phase_choices_ % to_train_size;
133
134         if (((models_to_train[idx] == &model_ct2_) || (models_to_train[idx] == &
            model_ct1_)) && !m_CT_possible__()) {
135             if (to_train_size > 1) idx = (num_first_phase_choices_ + 1) %
                to_train_size;
136             else return m_select_best__(models_trained, trained_size, Npa);
137         }
138
139         detail::LRmodel* selected_model = models_to_train[idx];
140
141         if (selected_model->n() >= required_instances_) {
142             if (idx != (to_train_size - 1)) {
143                 models_to_train[idx] = models_to_train[to_train_size - 1];
144                 models_to_train[to_train_size - 1] = selected_model;
145             }
146
147             to_train_size--;
148
149             models_trained[trained_size] = selected_model;
150             trained_size++;
151         }

```

```

152
153         return selected_model;
154     } // m_round_robin__
155
156     detail::LRmodel* m_select_best__(detail::LRmodel** models, int models_size,
157         int Npa) {
158         std::vector<double> costs;
159         costs.reserve(models_size);
160         for (int i = 0; i < models_size; ++i) costs.push_back(models[i]->
161             evaluate(Npa, K_));
162
163         int idx = std::min_element(costs.begin(), costs.end()) - costs.begin();
164         detail::LRmodel* best_model = models[idx];
165
166         if (((best_model == &model_ct2_) || (best_model == &model_ct1_)) && !
167             m_CT_possible__()) {
168             costs[idx] = std::numeric_limits<double>::max();
169             best_model = models[std::min_element(costs.begin(), costs.end()) -
170                 costs.begin()];
171         }
172
173         return best_model;
174     } // m_select_best__
175
176     bool m_CT_possible__() const {
177         long long unsigned int vec_size = sizeof(int);
178
179         // overflow possible so checking vector size after each xi and pa
180         // variable
181         for (int i = 0; i < recent_xi_variables_.size(); ++i) {
182             vec_size *= ct_.r(recent_xi_variables_[i]);
183             if (vec_size > CT_memory_limit_) return false;
184         }
185
186         for (int i = 0; i < recent_pa_variables_.size(); ++i) {
187             vec_size *= ct_.r(recent_pa_variables_[i]);
188             if (vec_size > CT_memory_limit_) return false;
189         }
190
191         return true;
192     } // m_CT_possible__
193
194     template <typename score_functor>
195     long long unsigned int m_run_query__(detail::LRmodel* model, const set_type&
196         xi, const set_type& pa, std::vector<score_functor>& F) {
197         auto start = std::chrono::steady_clock::now();

```

```

192
193         if (model->counter() == Rad) rad_.apply(xi, pa, F);
194         else if (model->counter() == BV) bv_.apply(xi, pa, F);
195         else ct_.apply(xi, pa, F);
196
197         auto end = std::chrono::steady_clock::now();
198         auto time = (static_cast<long long unsigned int>(std::chrono::
                duration_cast<std::chrono::nanoseconds>(end - start).count()));
199
200         return time;
201     } // m_run_query__
202
203     template <int M, typename Iter>
204     friend AutoCounter<M> create_AutoCounter(int, int, Iter, int, int, int);
205 }; // class AutoCounter
206
207
208 template <int N, typename Iter> AutoCounter<N> create_AutoCounter(int n, int m,
    Iter it, int burn_in = -1, int CT_memory_limit = 32, int required_instances
    = 3) {
209     AutoCounter<N> p;
210
211     p.rad_ = create_RadCounter<N>(n, m, it);
212     p.bv_ = create_BVCounter<N>(n, m, it);
213     p.ct_ = create_CTCounter<N>(n, m, it);
214
215     p.CT_memory_limit_ = static_cast<long long unsigned int>(CT_memory_limit) *
        1048576; // MiB to B
216
217     p.required_instances_ = required_instances;
218
219     p.model_bv1_ = detail::LRmodel(BV, false);
220     p.model_bv2_ = detail::LRmodel(BV, true);
221     p.model_ct1_ = detail::LRmodel(CT, true);
222     p.model_ct2_ = detail::LRmodel(CT, false);
223     p.model_rad_ = detail::LRmodel(Rad, true);
224
225     p.Lmodels_to_train_[0] = &p.model_rad_;
226     p.Lmodels_to_train_[1] = &p.model_bv1_;
227     p.Lmodels_to_train_[2] = &p.model_ct1_;
228     p.Rmodels_to_train_[0] = &p.model_rad_;
229     p.Rmodels_to_train_[1] = &p.model_bv2_;
230     p.Rmodels_to_train_[2] = &p.model_ct2_;
231
232     p.Lmodels_to_train_size_ = 3;
233     p.Rmodels_to_train_size_ = 3;

```

```

234
235     for (int i = 0; i < 3; ++i) p.Lmodels_trained_[i] = nullptr;
236     for (int i = 0; i < 3; ++i) p.Rmodels_trained_[i] = nullptr;
237
238     p.Lmodels_trained_size_ = 0;
239     p.Rmodels_trained_size_ = 0;
240
241     p.all_models_trained_ = false;
242
243     p.K_ = 0.0;
244
245     if (n > 0) {
246         for (int i = 0; i < n; ++i) p.K_ += p.ct_.r(i);
247         p.K_ /= n;
248     }
249
250     p.NO_ = static_cast<int>(std::log(p.m()) / std::log(p.K_));
251
252     p.num_first_phase_choices_ = 0;
253
254     if (burn_in > 0) {
255         std::random_device rd;
256         std::default_random_engine eng(rd());
257         std::uniform_int_distribution<int> size(0, n-1);
258         std::vector<estimator> F(1);
259
260         std::vector<int> v;
261         for (int i = 0; i < n; ++i) v.push_back(i);
262
263         for (int i = 0; i < burn_in; ++i) {
264             std::shuffle(v.begin(), v.end(), eng);
265
266             // first elements of v become the parents, and the last element of v
267             // becomes xi
268             std::vector<int> pa (v.begin(), v.begin() + size(eng));
269             std::vector<int> xi {v[n-1]};
270
271             p.apply(as_set<uint_type<N>>(xi.begin(), xi.end()), as_set<uint_type<N>>
272                   >>(pa.begin(), pa.end()), F);
273         }
274     }
275
276     return p;
277 } // create_AutoCounter
278
279 } // namespace sabnatk

```

278

279 `#endif // AUTO_COUNTER_HPP`

Listing 8.3. Kod źródłowy przedstawiający opracowaną metodę automatycznego wyboru strategii

```

1 // Plik: LRmodel.hpp
2 // wersja wspierająca równoległą realizację zapytań
3 // Autor: Paweł Bratek <pawel.bratek@pcz.pl>
4
5 #ifndef LR_MODEL_HPP
6 #define LR_MODEL_HPP
7
8 #include <cmath>
9 #include "counters.hpp"
10 #include <mutex>
11
12
13 namespace sabnatk {
14
15     namespace detail {
16
17         class LRmodel {
18         public:
19             explicit LRmodel(counter_type ct = Rad, bool il = true)
20                 : counter_(ct), is_linear_(il) {
21                 mutex_ = new std::mutex();
22             }
23
24             int n() const {
25                 std::scoped_lock lock(*mutex_);
26                 return n_;
27             }
28
29             counter_type counter() const { return counter_; }
30
31             double a() const { return cov_xy_ / var_x_; }
32
33             double b() const { return mean_y_ - a() * mean_x_; }
34
35             double evaluate(int x, double K) const {
36                 std::scoped_lock lock(*mutex_);
37                 if (is_linear_) return a() * x + b();
38                 return std::pow(K, b()) * std::pow(K, (a() * x));
39             } // evaluate
40
41             void update(int x, long long int y, double K) {

```

```

42         std::scoped_lock lock(*mutex_);
43         n_++;
44
45         double dx = x - mean_x_;
46
47         var_x_ += ((dx * dx * (n_ - 1) / n_) - var_x_) / n_;
48         mean_x_ += dx / n_;
49
50         double dy = is_linear_ ? y : std::log(y) / std::log(K);
51
52         dy -= mean_y_;
53         cov_xy_ += ((dx * dy * (n_ - 1) / n_) - cov_xy_) / n_;
54         mean_y_ += dy / n_;
55     } // update
56
57
58     private:
59         std::mutex* mutex_;
60         counter_type counter_;
61         bool is_linear_;
62
63         int n_ = 0;
64
65         double mean_x_ = 0.0;
66         double var_x_ = 0.0;
67         double mean_y_ = 0.0;
68         double cov_xy_ = 0.0;
69
70     }; // class LRmodel
71
72 } // namespace detail
73
74 } // namespace sabnatk
75
76 #endif // LR_MODEL_HPP

```

Listing 8.4. Kod źródłowy przedstawiający model kosztu realizacji zapytania w wersji wspierającej przetwarzanie wielowątkowe

```

1 // Plik: AutoCounter.hpp
2 // wersja wspierająca równoległą realizację zapytań
3 // Autor: Paweł Bratek <pawel.bratek@pcz.pl>
4
5 #ifndef AUTO_COUNTER_HPP
6 #define AUTO_COUNTER_HPP
7

```

```

 8  #include "counters.hpp"
 9
10  #include "BVCounter.hpp"
11  #include "CTCounter.hpp"
12  #include "RadCounter.hpp"
13
14  #include "LRmodel.hpp"
15
16  #include "estimator.hpp"
17  #include <random>
18  #include <chrono>
19
20  #include <mutex>
21  #include <fstream>
22
23  namespace sabnatk {
24
25      template <int N, typename Data = uint8_t> class AutoCounter {
26      public:
27          using set_type = uint_type<N>;
28          using data_type = Data;
29
30          counter_type type() const { return Auto; }
31
32
33          int n() const { return rad_.n(); }
34
35          int m() const { return rad_.m(); }
36
37          int r(int i) const { return rad_.r(i); }
38
39
40          bool is_reorderable() { return true; }
41
42          bool reorder(const std::vector<int>& norder) {
43              rad_.reorder(norder);
44              bv_.reorder(norder);
45              ct_.reorder(norder);
46              return true;
47          } // reorder
48
49
50          template <typename score_functor>
51          void apply(const set_type& xi, const set_type& pa, std::vector<score_functor
52              >& F) {

```

```

53         std::vector<int> pa_vector;
54         std::vector<int> xi_vector;
55
56         as_vector(pa, pa_vector);
57         as_vector(xi, xi_vector);
58
59         // regression models ty = f(Npa, K)
60         detail::LRmodel* model = nullptr;
61         long long unsigned int ty = 0;
62
63         int Npa = pa_vector.size();
64
65         if (Npa <= NO_) {
66             if (Lmodels_to_train_size_ > 0) {
67                 std::scoped_lock lock(*Lmutex_);
68                 model = m_round_robin__(Lmodels_to_train_,
69                                         Lmodels_to_train_size_, Lmodels_trained_,
70                                         Lmodels_trained_size_, L_rr_counter_, Npa, xi_vector,
71                                         pa_vector);
69             } else {
70                 model = m_select_best__(Lmodels_trained_, Lmodels_trained_size_,
71                                         Npa, xi_vector, pa_vector);
71             }
72         } else {
73             if (Rmodels_to_train_size_ > 0) {
74                 std::scoped_lock lock(*Rmutex_);
75                 model = m_round_robin__(Rmodels_to_train_,
76                                         Rmodels_to_train_size_, Rmodels_trained_,
77                                         Rmodels_trained_size_, R_rr_counter_, Npa, xi_vector,
78                                         pa_vector);
76             } else {
77                 model = m_select_best__(Rmodels_trained_, Rmodels_trained_size_,
78                                         Npa, xi_vector, pa_vector);
78             }
79         }
80
81         ty = m_run_query__(model, xi, pa, F);
82
83         model->update(Npa, ty, K_);
84
85     } // apply
86
87 private:
88     RadCounter<N> rad_;
89     BVCounter<N> bv_;
90     CTCCounter<N> ct_;

```

```

91
92     detail::LRmodel model_rad_;
93     detail::LRmodel model_bv1_, model_bv2_;
94     detail::LRmodel model_ct1_, model_ct2_;
95
96     // two groups (L, R) of models based on NO
97     detail::LRmodel* Lmodels_to_train_[3];
98     detail::LRmodel* Rmodels_to_train_[3];
99     int Lmodels_to_train_size_;
100    int Rmodels_to_train_size_;
101
102    detail::LRmodel* Lmodels_trained_[3];
103    detail::LRmodel* Rmodels_trained_[3];
104    int Lmodels_trained_size_;
105    int Rmodels_trained_size_;
106
107    std::mutex* Lmutex_;
108    std::mutex* Rmutex_;
109
110    // query statistics
111    double K_;
112    int NO_;
113
114    // storing number of round-robin selections for L and R models
115    int L_rr_counter_;
116    int R_rr_counter_;
117
118    int required_instances_; // the minimum number of data required to use the
                             // model during auto selection
119    long long unsigned int CT_memory_limit_;
120
121
122    detail::LRmodel* m_round_robin__(detail::LRmodel** models_to_train, int&
                                     to_train_size,
123                                     detail::LRmodel** models_trained, int&
                                     trained_size, int& rr_counter, int Npa,
                                     const std::vector<int>& xi, const std
                                     ::vector<int>& pa) {
124
125        // required to check as parallel execution may cause that all model are
        // already trained (threads are waiting for the mutex before entering
        // m_round_robin__)
126        if (to_train_size > 0) {
127            rr_counter++;
128            int idx = rr_counter % to_train_size;
129

```

```

130         // change strategy if CT is not able to handle the query
131         if (((models_to_train[idx] == &model_ct2_) || (models_to_train[idx] ==
132             &model_ct1_)) && !m_CT_possible__(xi, pa)) {
133             if (to_train_size > 1) idx = (rr_counter + 1) % to_train_size;
134             else return m_select_best__(models_trained, trained_size, Npa, xi,
135                 pa);
136         }
137
138         detail::LRmodel* selected_model = models_to_train[idx];
139
140         if (selected_model->n() >= required_instances_) {
141             if (idx != (to_train_size - 1)) {
142                 models_to_train[idx] = models_to_train[to_train_size - 1];
143                 models_to_train[to_train_size - 1] = selected_model;
144             }
145
146             to_train_size--;
147
148             models_trained[trained_size] = selected_model;
149             trained_size++;
150         }
151
152         return selected_model;
153     }
154     else return m_select_best__(models_trained, trained_size, Npa, xi, pa);
155 } // m_round_robin__
156
157 detail::LRmodel* m_select_best__(detail::LRmodel** models, int models_size,
158     int Npa, const std::vector<int>& xi, const std::vector<int>& pa) {
159     std::vector<double> costs;
160     costs.reserve(models_size);
161     for (int i = 0; i < models_size; ++i) costs.push_back(models[i]->
162         evaluate(Npa, K_));
163
164     int idx = std::min_element(costs.begin(), costs.end()) - costs.begin();
165     detail::LRmodel* best_model = models[idx];
166
167     if (((best_model == &model_ct2_) || (best_model == &model_ct1_)) && !
168         m_CT_possible__(xi, pa)) {
169         costs[idx] = std::numeric_limits<double>::max();
170         best_model = models[std::min_element(costs.begin(), costs.end()) -
171             costs.begin()];
172     }
173
174     return best_model;
175 } // m_select_best__

```

```

170
171     bool m_CT_possible__(const std::vector<int>& xi, const std::vector<int>& pa)
172         const {
173         long long unsigned int vec_size = sizeof(int);
174
175         // overflow possible so checking vector size after each xi and pa
176         variable
177         for (int i = 0; i < xi.size(); ++i) {
178             vec_size *= ct_.r(xi[i]);
179             if (vec_size > CT_memory_limit_) return false;
180         }
181
182         for (int i = 0; i < pa.size(); ++i) {
183             vec_size *= ct_.r(pa[i]);
184             if (vec_size > CT_memory_limit_) return false;
185         }
186
187         return true;
188     } // m_CT_possible__
189
190     template <typename score_functor>
191     long long unsigned int m_run_query__(detail::LRmodel* model, const set_type&
192         xi, const set_type& pa, std::vector<score_functor>& F) {
193         auto start = std::chrono::steady_clock::now();
194
195         if (model->counter() == Rad) rad_.apply(xi, pa, F);
196         else if (model->counter() == BV) bv_.apply(xi, pa, F);
197         else ct_.apply(xi, pa, F);
198
199         auto end = std::chrono::steady_clock::now();
200         auto time = (static_cast<long long unsigned int>(std::chrono::
201             duration_cast<std::chrono::nanoseconds>(end - start).count()));
202
203         return time;
204     } // m_run_query__
205
206     template <int M, typename Iter>
207     friend AutoCounter<M> create_AutoCounter(int, int, Iter, int, int, int);
208 }; // class AutoCounter
209
210     template <int N, typename Iter> AutoCounter<N> create_AutoCounter(int n, int m,
211         Iter it, int burn_in = -1, int CT_memory_limit = 32, int required_instances
212         = 5) {
213         AutoCounter<N> p;
214     }

```

```

210     p.rad_ = create_RadCounter<N>(n, m, it);
211     p.bv_ = create_BVCounter<N>(n, m, it);
212     p.ct_ = create_CTCounter<N>(n, m, it);
213
214     p.CT_memory_limit_ = static_cast<long long unsigned int>(CT_memory_limit) *
        1048576; // MiB to B
215
216     p.required_instances_ = required_instances;
217
218     p.model_bv1_ = detail::LRmodel(BV, false);
219     p.model_bv2_ = detail::LRmodel(BV, true);
220     p.model_ct1_ = detail::LRmodel(CT, true);
221     p.model_ct2_ = detail::LRmodel(CT, false);
222     p.model_rad_ = detail::LRmodel(Rad, true);
223
224     p.Lmodels_to_train_[0] = &p.model_rad_;
225     p.Lmodels_to_train_[1] = &p.model_bv1_;
226     p.Lmodels_to_train_[2] = &p.model_ct1_;
227     p.Rmodels_to_train_[0] = &p.model_rad_;
228     p.Rmodels_to_train_[1] = &p.model_bv2_;
229     p.Rmodels_to_train_[2] = &p.model_ct2_;
230
231     p.Lmodels_to_train_size_ = 3;
232     p.Rmodels_to_train_size_ = 3;
233
234     for (int i = 0; i < 3; ++i) p.Lmodels_trained_[i] = nullptr;
235     for (int i = 0; i < 3; ++i) p.Rmodels_trained_[i] = nullptr;
236
237     p.Lmodels_trained_size_ = 0;
238     p.Rmodels_trained_size_ = 0;
239
240     p.Lmutex_ = new std::mutex();
241     p.Rmutex_ = new std::mutex();
242
243     p.K_ = 0.0;
244
245     if (n > 0) {
246         for (int i = 0; i < n; ++i) p.K_ += p.ct_.r(i);
247         p.K_ /= n;
248     }
249
250     p.NO_ = static_cast<int>(std::log(p.m()) / std::log(p.K_));
251
252     p.L_rr_counter_ = 0;
253     p.R_rr_counter_ = 0;
254

```

```

255     if (burn_in > 0) {
256         std::random_device rd;
257         std::default_random_engine eng(rd());
258         std::uniform_int_distribution<int> size(0, n-1);
259         std::vector<estimator> F(1);
260
261         std::vector<int> v;
262         for (int i = 0; i < n; ++i) v.push_back(i);
263
264         for (int i = 0; i < burn_in; ++i) {
265             std::shuffle(v.begin(), v.end(), eng);
266
267             // first elements of v become the parents, and the last element of v
                becomes xi
268             std::vector<int> pa (v.begin(), v.begin() + size(eng));
269             std::vector<int> xi {v[n-1]};
270
271             p.apply(as_set<uint_type<N>>(xi.begin(), xi.end()), as_set<uint_type<N>
                >>(pa.begin(), pa.end()), F);
272         }
273     }
274
275     return p;
276 } // create_AutoCounter
277
278 } // namespace sabnatk
279
280 #endif // AUTO_COUNTER_HPP

```

Listing 8.5. Kod źródłowy przedstawiający opracowaną metodę automatycznego wyboru strategii w wersji wspierającej przetwarzanie wielowątkowe

```

1 // Plik: ccNUMA.cpp
2 // Autor: Pawel Bratek <pawel.bratek@pcz.pl>
3
4 // Należy określić:
5 // liczbę domen NUMA (linia 18)
6 // liczbę rdzeni w każdej domenie (linia 19)
7
8 // Konfiguracja: export OMP_PROC_BIND=spread,close; export OMP_PLACES=threads;
9
10 #include <iostream>
11 #include <vector>
12 #include <sabnatk.hpp>
13 #include "estimator.hpp"
14 #include "csv.hpp"

```

```

15 #include "read_queries.hpp"
16 #include <omp.h>
17
18 #define NUMA_domains 8
19 #define cores_per_NUMA 16
20
21 int main(int argc, char* argv[]) {
22     if (argc != 3) {
23         std::cout << "usage: random_query_omp csv_file dat_file" << std::endl;
24         return -1;
25     }
26
27     const int N = 1;
28     using set_type = uint_type<N>;
29     using data_type = uint8_t;
30
31     std::vector<data_type> D;
32     auto [res, n, m] = read_csv(argv[1], D);
33     if (res == false) {
34         std::cout << "error: unable to read input data" << std::endl;
35         return -1;
36     }
37
38     std::vector<std::pair<set_type, set_type>> Q;
39     if (!read_queries(argv[2], Q)) {
40         std::cout << "error: unable to read queries" << std::endl;
41         return -1;
42     }
43     int qs = Q.size();
44
45     std::vector<estimator> F(1);
46
47     int processed_queries = 0;
48
49     omp_set_nested(1);
50     #pragma omp parallel num_threads(NUMA_domains)
51     {
52         auto eng = sabnatk::create_counter<N, sabnatk::Auto>(n, m, std::begin(D),
53             -1, 1024, 5);
54
55         omp_set_num_threads(cores_per_NUMA);
56         #pragma omp parallel shared(Q, eng, processed_queries), firstprivate(F)
57         {
58             int id;
59             while (processed_queries < qs) {
60                 #pragma critical

```

```
60         {
61             id = processed_queries;
62             processed_queries++;
63         }
64
65         if (id < qs) eng.apply(Q[id].first, Q[id].second, F);
66     }
67 }
68 }
69
70 return 0;
71 }
```

Listing 8.6. Kod źródłowy przedstawiający opracowaną metodę równoległej realizacji zapytań zliczających uwzględniającej architekturę systemów ccNUMA

Spis rysunków

2.1	Czas realizacji oraz zużycie energii elektrycznej uzyskane dla aplikacji trójwymiarowej dyfuzji podczas stosowania homogenicznej metody DFVS	23
2.2	Czas realizacji oraz zużycie energii elektrycznej uzyskane dla aplikacji MPDATA podczas stosowania homogenicznej metody DFVS	26
2.3	Porównanie redukcji zużycia energii elektrycznej uzyskanego za pomocą homogenicznego oraz heterogenicznego wariantu metody DVFS zastosowanych dla domen aplikacji trójwymiarowej dyfuzji o różnym rozmiarze	30
2.4	Porównanie redukcji zużycia energii elektrycznej uzyskanego za pomocą homogenicznego oraz heterogenicznego wariantu metody DVFS zastosowanych dla domen aplikacji MPDATA o różnym rozmiarze	31
2.5	Schemat blokowy algorytmu wyboru heterogenicznej konfiguracji częstotliwości w oparciu o pomiary energii elektrycznej	34
2.6	Schemat blokowy algorytmu wyboru heterogenicznej konfiguracji częstotliwości w oparciu o pomiary czasu trwania aplikacji	36
2.7	Szacowany koszt poszukiwania heterogenicznej konfiguracji częstotliwości w oparciu o opracowane algorytmy	40
2.8	Porównanie redukcji zużycia energii elektrycznej uzyskanego za pomocą homogenicznego oraz heterogenicznego wariantu metody DVFS na platformie obliczeniowej firmy AMD	42
3.1	Przykład prostej bazy danych	44
3.2	Tabela kontyngencji dla zapytania zliczającego $\text{COUNT}(X_1 \mid \{X_2, X_3\})$. .	45
3.3	Interpretacja liczb N_{ij} oraz N_{ijk} na przykładzie tabeli kontyngencji	46
3.4	Bitmapy zmiennych X_1 , X_2 oraz X_3 z bazy danych D przedstawionej na rysunku 3.1	52
3.5	Przykład wykonania zapytania zliczającego $\text{COUNT}(X_1 \mid \{X_2, X_3\})$ z użyciem strategii Bitmap	55
3.6	Przykład wykonania zapytania zliczającego $\text{COUNT}(X_1 \mid \{X_2, X_3\})$ strategią Radix	58
3.7	Wyniki realizacji zapytania zliczającego $\text{COUNT}(X_1 \mid \{X_2, X_3\})$ przy użyciu SABNatk (listing 3.2)	61
3.8	Fragment pliku przedstawiający 25 obserwacji dziesięciu pierwszych zmiennych losowych zbioru danych Child	65
3.9	Fragment pliku przedstawiający dziesięć pierwszych losowych zapytań dla zbioru danych Child	66

4.1	Liczba wykonywanych przecięć bitmap w ostatniej warstwie drzewa DFS w zależności od liczby zmiennych wchodzących w skład zapytania przy założeniu, że każda zmienna przyjmuje trzy stany	70
4.2	Redukcja liczby przecięć bitmap w przykładowym fragmencie drzewa DFS uzyskana dzięki zastosowaniu opracowanej metody <i>SNIIK</i>	72
4.3	Fragment drzewa DFS pokazujący obserwację wykorzystywaną w metodzie <i>SNIIJ</i>	74
4.4	Redukcja liczby węzłów w przykładowym drzewie DFS uzyskana dzięki zastosowaniu opracowanej metody <i>SNIIJ</i>	76
4.5	Wykresy skrzypcowe dla różnych zbiorów danych prezentujące przyspieszenie zapytań zliczających uzyskane dzięki opracowanym metodom dla strategii Bitmap	81
4.6	Analiza przyspieszenia oraz redukcji liczby wywołań jądra obliczeniowego na przykładzie zbioru Mildew	82
5.1	Drzewa DFS powstałe podczas realizacji zapytań: (a) $\text{COUNT}(X_i \mid \{X_a, X_b\})$ oraz (b) $\text{COUNT}(X_i \mid \{X_a, X_b, X_c, X_d\})$, przy założeniu, że wszystkie zmienne wchodzące w skład zapytań są binarne	85
5.2	Drzewa DFS powstałe podczas realizacji zapytania $\text{COUNT}(X_i \mid \{X_a, X_b\})$ przy założeniu, że wszystkie zmienne wchodzące w skład zapytania mogą przyjmować: (a) dwa stany, (b) cztery stany	86
5.3	Łączna liczba węzłów w drzewie DFS w zależności od rozmiaru zapytania oraz liczby stanów przyjmowanych przez jego zmienne	87
5.4	Operacja AND wykonana dla dwóch przykładowych bitmap bazy danych zawierającej m obserwacji	88
5.5	Porównanie wydajności strategii Tabeli Kontyngencji (CT), Bitmap (BV) oraz Radix (RAD) w zależności od rozmiaru zapytań zliczających oraz zbioru danych	89
5.6	Potencjalny zysk z opracowania metody automatycznego wyboru strategii (AUTO) do realizacji zapytań zliczających	91
5.7	Koncepcja opracowanej metody automatycznego wyboru strategii do realizacji zapytań zliczających	92
5.8	Drzewo DFS powstałe dla przykładowego zapytania zliczającego	97
5.9	Prawdopodobieństwo nieusunięcia węzła warstwy L dla różnych wartości parametru k oraz (a) $m = 1000$ i (b) $m = 100000$	101
5.10	Przykład wykorzystania opracowanych modeli regresji do oszacowania kosztu realizacji zapytania zliczającego na podstawie jego rozmiaru	106

5.11	Modele regresji dla dwóch zbiorów danych: (a) Hailfinder ($n = 56$, $m = 1K$) i b) Water ($n = 32$, $m = 100K$) po przetworzeniu pierwszych dziesięciu, stu i tysiąca zapytań ze strumienia (odpowiednio: pierwszy, drugi i trzeci wiersz każdej kolumny)	113
5.12	Wykresy skrzypcowe prezentujące rozkład czasów realizacji zapytań o loso- wym rozmiarze	119
7.1	Czasy przetwarzania zapytań przez poszczególne domeny NUMA w przypadku zastosowania wstępnej oraz finalnej wersji opracowanej metody	140
8.1	Wizualizacja koncepcji mechanizmu kolejkowania i reorganizacji zapytań zli- czających	149

Spis tabel

2.1	Wykorzystane konfiguracje częstotliwości (w GHz) podczas testowania opracowanej metody	29
2.2	Porównanie podejścia „brute force” z opracowanymi algorytmami (AE – bazujący na pomiarach energii, AT – bazujący na pomiarach czasu) dla trójwymiarowej aplikacji dyfuzji i sześciennych domen obliczeniowych ($X = Y = Z$)	37
2.3	Porównanie podejścia „brute force” z opracowanymi algorytmami (AE – bazujący na pomiarach energii, AT – bazujący na pomiarach czasu) dla aplikacji MPDATA i sześciennych domen obliczeniowych ($X = Y = Z$)	38
3.1	Specyfikacja platform obliczeniowych wykorzystanych podczas ewaluacji opracowanych metod i algorytmów w ramach tematyki zapytań zliczających . .	64
3.2	Zbiory danych wykorzystywane podczas testowania opracowanych rozwiązań	65
4.1	Dobór liczby zapytań do liczby obserwacji zbioru danych podczas testowania opracowanych metod dla strategii Bitmap	78
4.2	Przyspieszenie uzyskane w benchmarku losowych zapytań dzięki opracowanym metodom <i>SNIJ</i> i <i>SNIJK</i> dla strategii Bitmap. Platforma obliczeniowa: Intel Cascade Lake	79
4.3	Przyspieszenie uzyskane w benchmarku losowych zapytań dzięki opracowanym metodom <i>SNIJ</i> i <i>SNIJK</i> dla strategii Bitmap. Platforma obliczeniowa: AMD Rome	83
4.4	Przyspieszenie uzyskane w benchmarku losowych zapytań dzięki opracowanym metodom <i>SNIJ</i> i <i>SNIJK</i> dla strategii Bitmap. Platforma obliczeniowa: AMD Milan	83
5.1	Przyspieszenie uzyskane dla prototypu metody automatycznego wyboru strategii do realizacji zapytań zliczających w porównaniu do losowego wyboru strategii	104
5.2	Przyspieszenie opracowanej metody automatycznego wyboru strategii w porównaniu do różnych podejść dla benchmarku strumienia stu tysięcy losowych zapytań	121
5.3	Procent zrealizowanych zapytań przez strategię wybierane przez opracowaną metodę automatycznego wyboru	122
5.4	Całkowita liczba zapytań zrealizowanych przez solver wyznaczania zbioru rodziców (w milionach)	124
5.5	Całkowity czas realizacji solvera wyznaczania zbioru rodziców (w sekundach)	125

5.6	Udział zapytań zliczających w całkowitym czasie realizacji solvera wyznaczania zbioru rodziców	127
6.1	Całkowity czas realizacji solvera wyznaczania zbioru rodziców (w sekundach)	134
7.1	Porównanie wydajności metody automatycznego wyboru strategii w benchmarku losowych zapytań w oparciu o standardowe podejście do równoległej realizacji zapytań z wykorzystaniem OpenMP oraz opracowaną metodę równoległej realizacji zapytań uwzględniającej architekturę systemów ccNUMA	142
8.1	Publikacje autora związane z metodami opracowanymi w ramach niniejszej rozprawy doktorskiej	148

Spis listingów

2.1	Zrównoleglenie jądra obliczeniowego aplikacji trójwymiarowej dyfuzji przy użyciu biblioteki OpenMP	19
2.2	Zrównoleglenie jądra obliczeniowego aplikacji MPDATA przy użyciu biblioteki OpenMP	20
3.1	Przykładowy obiekt funkcyjny C++ implementujący operator BDeu dany równaniem (3.9). Źródło: [30]	50
3.2	Wykorzystanie biblioteki SABNAtk do realizacji zapytania zliczającego COUNT ($X_1 \mid \{X_2, X_3\}$)	59
3.3	Zrównoleglenie realizacji strumienia zapytań zliczających z wykorzystaniem biblioteki OpenMP	63
3.4	Realizacja strumienia zapytań Q na zbiorze danych D z wykorzystaniem biblioteki SABNAtk	67
5.1	Realizacja strumienia zapytań Q na zbiorze danych D z wykorzystaniem opracowanej metody automatycznego wyboru strategii	114
6.1	Przykład wykorzystania muteksu i klasy <code>std::scoped_lock</code> do ochrony sekcji krytycznej przed sytuacją wyścigu	129
6.2	Równoległa realizacja strumienia zapytań Q na zbiorze danych D z wykorzystaniem opracowanej metody automatycznego wyboru strategii oraz standardu OpenMP	131
7.1	Koncepcja metody równoległej realizacji zapytań zliczających uwzględniająca architekturę systemów ccNUMA	137
7.2	Wstępna wersja metody równoległej realizacji zapytań zliczających uwzględniającej architekturę systemów ccNUMA	138
8.1	Kod źródłowy przedstawiający wprowadzone zmiany związane z opracowanymi metodami SNIJ i SNIJK	150
8.2	Kod źródłowy przedstawiający model kosztu realizacji zapytania	152
8.3	Kod źródłowy przedstawiający opracowaną metodę automatycznego wyboru strategii	154
8.4	Kod źródłowy przedstawiający model kosztu realizacji zapytania w wersji wspierającej przetwarzanie wielowątkowe	161
8.5	Kod źródłowy przedstawiający opracowaną metodę automatycznego wyboru strategii w wersji wspierającej przetwarzanie wielowątkowe	162
8.6	Kod źródłowy przedstawiający opracowaną metodę równoległej realizacji zapytań zliczających uwzględniającej architekturę systemów ccNUMA	169

Bibliografia

- [1] *Implementacja strategii alternujących drzew decyzyjnych (adt)*. <https://gitlab.com/SCoRe-Group/SABNatk-Benchmarks/-/blob/master/src/ADTree.hpp>.
- [2] *Implementacja strategii tablic haszujących (ht)*. <https://gitlab.com/SCoRe-Group/SABNatk-Benchmarks/-/blob/master/src/Hasher.hpp>.
- [3] B. ACUN, K. CHANDRASEKAR, L. KALE, *Fine-Grained Energy Efficiency Using Per-Core DVFS with an Adaptive Runtime Systems*, 2019 Tenth Int. Green and Sustainable Computing Conf. (IGSC), Vol. 1, 2019, 1–8.
- [4] B. ACUN, P. MILLER, L. KALE, *Variation Among Processors Under Turbo Boost in HPC Systems*, ICS'16: Proc. 2016 Int. Conf. on Supercomputing, 2016, 1–12.
- [5] R. AGRAWAL, T. IMIELINSKI, A. SWAMI, *Mining association rules between sets of items in large databases*, ACM SIGMOD International Conference on Management of Data, 22, 1993, 207–216.
- [6] *AMD EPYC 7773X Specification*. <https://www.amd.com/en/products/processors/server/epyc/7003-series/amd-epyc-7773x.html>.
- [7] *5th Generation AMD EPYC™ Processors*. <https://www.amd.com/en/products/processors/server/epyc/9005-series.html>.
- [8] B. ANDERSON, A. MOORE, *ADtrees for fast counting and for fast learning of association rules*, Conference on Knowledge Discovery and Data Mining, 1998.
- [9] *Bayesian network repository*. <https://www.bnlearn.com/bnrepository>.
- [10] P. BRATEK, *Wybrane metody numeryczne i analityczne dla dwuwymiarowego równania dyfuzji*, Praca magisterska, Politechnika Częstochowska, 2019.
- [11] P. BRATEK, *Projekt i implementacja warstwy komunikacyjnej dla klastrów obliczeniowych w środowisku MPI*, Praca inżynierska, Politechnika Częstochowska, 2020.
- [12] P. BRATEK, L. SZUSTAK, *Optimizing the dfs-based strategy for efficient execution of counting queries in machine learning applications*, 33rd International Conference on Information System Development, 2025.

- [13] P. BRATEK, L. SZUSTAK, R. WYRZYKOWSKI, T. OLAS, *Reducing energy consumption using heterogeneous voltage frequency scaling of data-parallel applications for multicore systems*, Journal of Parallel and Distributed Computing, 175 (2023), 121–133.
- [14] P. BRATEK, L. SZUSTAK, R. WYRZYKOWSKI, T. OLAS, T. CHMIEL, *Heterogeneous voltage frequency scaling of data-parallel applications for energy saving on homogeneous multicore platforms*, Euro-Par 2021: Parallel Processing Workshops, Cham, 2022, Springer International Publishing, 141–153.
- [15] P. BRATEK, L. SZUSTAK, J. ZOLA, *Boosting performance of counting queries in machine learning applications with a ccnuma-aware implementation*, Euro-Par 2025: Parallel Processing Workshops.
- [16] P. BRATEK, L. SZUSTAK, J. ZOLA, *Parallelization and auto-scheduling of data access queries in ml workloads*, Euro-Par 2021: Parallel Processing Workshops, 2022, 525–529.
- [17] P. BRATEK, L. SZUSTAK, J. ZOLA, *Parallel auto-scheduling of counting queries in machine learning applications on hpc systems*, Euro-Par 2023: Parallel Processing Workshops, 2024, 327–333.
- [18] E. CALORE, A. GABBANA, S. SCHIFANO, R. TRIPICCIONE, *Software and DVFS tuning for performance and energy-efficiency on Intel KNL processors*, Journal of Low Power Electronics and Applications, 8 (2018).
- [19] M. CIZNICKI, K. KUROWSKI, *Resource management strategies with energy profiles for stencil computing*, HiStencil 2015: 2nd Int. Workshop on High-Performance Stencil Computing, 2015, 943–950.
- [20] G. COOPER, E. HERSKOVITS, *A Bayesian method for the induction of probabilistic networks from data*, Machine Learning, 9 (1992), 309–347.
- [21] J. CRANK, *The Mathematics of Diffusion*, Clarendon Press, Oxford, 1975.
- [22] L. DE CAMPOS, *A scoring function for learning Bayesian networks based on mutual information and conditional independence tests*, Journal of Machine Learning Research, 7 (2006), 2149–2187.
- [23] M. FAHAD, A. SHAHID, R. MANUMACHU, A. LASTOVETSKY, *Energy Predictive Models of Computing: Theory, Practical Implications and Experimental Analysis on Multicore Processors*, IEEE Access, 9 (2021), 63149–63172.

- [24] W. GREBLICKI, *Statystyka matematyczna*, Wydawnictwo Naukowe PWN, Warszawa, 2009.
- [25] *Green500 list*. <https://top500.org/lists/green500/>.
- [26] J. HAJ-YAHYA, A. MENDELSON, Y. BEN-ASHER, *Energy Efficient High Performance Processors: Recent Approaches for Designing Green High Performance Computing*, Springer, 2018.
- [27] K. HALBINIAK, L. SZUSTAK, A. KULAWIK, P. GEPNER, *Performance optimizations for parallel modeling of solidification with dynamic intensity of computation*, Parallel Processing and Applied Mathematics, R. Wyrzykowski, E. Deelman, J. Dongarra, K. Karczewski (red.), Cham, 2020, Springer International Publishing, 370–381.
- [28] IBM, *Breaking the petaflop barrier*. <https://www.ibm.com/history/petaflop-barrier>, 2008.
- [29] *Intel one Api Threading Building Blocks*. <https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html>.
- [30] S. KARAN, M. EICHHORN, B. HURLBURT, G. IRACI, J. ZOLA, *Fast counting in machine learning applications*, Uncertainty in Artificial Intelligence, 2018.
- [31] K. N. KHAN, M. HIRKI, T. NIEMI, J. K. NURMINEN, Z. OU, *Rapl in action: Experiences in using rapl for power measurements*, ACM Trans. Model. Perform. Eval. Comput. Syst., 3 (2018).
- [32] R. KOHAVI, *Scaling up the accuracy of Naive-Bayes classifiers: A decision-tree hybrid*, International Conference on Knowledge Discovery and Data Mining, 1996, 202–207.
- [33] M. KOIVISTO, *Parent assignment is hard for the MDL, AIC, and NML costs*, International Conference on Computational Learning Theory, 2006, 289–303.
- [34] D. KOLLER, N. FRIEDMAN, *Probabilistic Graphical Models: Principles and Techniques*, MIT Press, 2009.
- [35] T. KOLPE, A. ZHAI, S. SAPATNEKAR, *Enabling Improved Power Management in Multicore Processors through Clustered DVFS*, 2011 Design, Automation & Test in Europe, 2011, 1–6.

- [36] W. KRYSICKI, L. WŁODARSKI, *Rachunek prawdopodobieństwa i statystyka matematyczna w zadaniach. Cz. 1*, Wydawnictwo Naukowe PWN, Warszawa, wydanie xiv red., 2005.
- [37] M. LAWENDA, K. KHLOPONIN, K. SAMBORSKI, SZUSTAK, *Profiling and optimization of multi-card gpu machine learning jobs*, Concurrency and Computation - Practice and Experience, (2025).
- [38] H. LEE, R. GROSSE, R. RANGANATH, A. NG, *Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations*, International Conference on Machine Learning, ACM, 2009, 609–616.
- [39] LINUX FOUNDATION, *numastat - NUMA policy statistics tool*, 2024. <https://man7.org/linux/man-pages/man8/numastat.8.html>.
- [40] J. MAIR, Z. HUANG, D. EYERS, Y. CHEN, *Quantifying the Energy Efficiency Challenges of Achieving Exascale Computing*, 15th IEEE/ACM Int. Symp. Cluster, Cloud and Grid Computing, 2015, 943–950.
- [41] A. MOORE, M. LEE SOON, *Cached sufficient statistics for efficient machine learning with large datasets*, Journal of Artificial Intelligence Research, 8 (1998), 67–91.
- [42] D. NARAYANAN, M. SHOEYBI, J. CASPER, P. LEGRESLEY, M. PATWARY, V. A. KORTHIKANTI, D. VAINBRAND, P. KASHINKUNTI, J. BERNAUER, B. CATTANZARO, A. PHANISHAYEE, M. ZAHARIA, *Efficient large-scale language model training on gpu clusters using megatron-lm*, arXiv, (2021). <https://arxiv.org/abs/2104.04473>.
- [43] T. OLAS, *Advances in adapting memory-bound cfd computations to risc-v multi-core architecture*, ICCS 2025: International Conference on Computational Science, Springer, 2025, 181–195.
- [44] *OpenMP*. <http://www.openmp.org/>.
- [45] A. PAPADIMITRIOU, A. CHATZIDIMITRIOU, D. GIZOPOULOS, *Adaptive Voltage-Frequency Scaling and Core Allocation for Balanced Energy and Performance on Multicore CPUs*, 2019 IEEE Int. Symp. High Performance Computer Architecture (HPCA), 2019, 133–146.
- [46] A. PRAKASH, S. WANG, A. IRIMIEA, T. MITRA, *Energy-efficient execution of data-parallel applications on heterogeneous mobile platforms*, 2015 33rd IEEE Int. Conf. on Computer Design (ICCD), IEEE, 2015.

- [47] W. H. PRESS, S. A. TEUKOLSKY, W. T. VETTERLING, B. P. FLANNERY, *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, New York, NY, USA, 3 red., 2007.
- [48] PRESS GAZETTE, *How google ai overviews is fuelling zero-click searches for top publishers*, 2025.
https://pressgazette.co.uk/media-audience-and-business-data/media_metrics/how-google-ai-overviews-is-fuelling-zero-click-searches-for-top-publishers/.
- [49] J. QUINLAN, *Induction of decision trees*, Machine Learning, 1 (1986), 81–106.
- [50] J. QUINLAN, *Bagging, boosting, and C4.5*, AAAI Innovative Applications of Artificial Intelligence Conferences, 1996, 725–730.
- [51] J. RAMOS, *Using tf-idf to determine word relevance in document queries*, Instructional Conference on Machine Learning, 2003, 133–142.
- [52] T. RAUBER, G. RÜNGER, M. STACHOWSKI, *Performance and energy metrics for multi-threaded applications on DVFS processors*, Sustainable Computing: Informatics and Systems, 17 (2018), 55–68.
- [53] RISC-V. <https://riscv.org/>.
- [54] K. ROJEK, A. ILIC, R. WYRZYKOWSKI, L. SOUSA, *Energy-aware mechanism for stencil-based MPDATA algorithm with constraints*, Concurrency Computat. Pract. Exper., 29 (2017).
- [55] K. ROJEK, E. QUINTANA-ORTI, R. WYRZYKOWSKI, *Modeling power consumption of 3D MPDATA and the CG method on ARM and Intel multicore architectures*, Concurrency Computat. Pract. Exper., 73 (2017), 4373–4389.
- [56] B. ROSA, L. SZUSTAK, A. WYSZOGRODZKI, K. ROJEK, D. WOJCIK, R. WYRZYKOWSKI, *Adaptation of multidimensional positive definite advection transport algorithm to modern high-performance computing platforms*, Int. Journal of Modeling and Optimization, 5 (2015), 171–176.
- [57] SABNA – Scalable Accelerated Bayesian Network Analytics. <https://gitlab.com/SCoRe-Group/SABNA>.
- [58] SABNAtk. <https://gitlab.com/SCoRe-Group/SABNAtk>.
- [59] R. SALAKHUTDINOV, G. HINTON, *Deep Boltzmann machines*, Int. Conference on Artificial Intelligence and Statistics, 2009, 448–455.

- [60] K. SANKARALINGAM, S. KECKLER, W. MARK, D. BURGER, *Universal Mechanisms for Data-Parallel Applications*, Proc. 36th Int. Symp. Microarchitecture (MICRO-36), 2003.
- [61] G. SCHWARZ, *Estimating the dimension of a model*, The Annals of Statistics, 6 (1978), 461–464.
- [62] SIMILARWEB, *Ai referral traffic winners by industry*. <https://www.similarweb.com/blog/insights/ai-news/ai-referral-traffic-winners/>, 2025.
- [63] P. SMOLARKIEWICZ, *Multidimensional Positive Definite Advection Transport Algorithm: An Overview*, Int. J. Numer. Meth. Fluids, 50 (2006), 1123–1144.
- [64] *Sparse AD-Tree package in Python*. <https://github.com/uraplutonium/adtree-py>.
- [65] M. STONEBRAKER, D. J. ABADI, A. BATKIN, X. CHEN, M. CHERNIACK, M. FERREIRA, E. LAU, A. LIN, S. MADDEN, E. O’NEIL, P. O’NEIL, A. RASIN, N. TRAN, S. ZDONIK, *C-store: A column-oriented dbms*, Proceedings of the 31st International Conference on Very Large Data Bases (VLDB), VLDB Endowment, 2005, 553–564.
- [66] L. SZUSTAK, *Metody zrównoleglenia i równoważenia obciążenia architektur hybrydowych w obliczeniach numerycznych*, Praca doktorska, Politechnika Częstochowska, 2012.
- [67] L. SZUSTAK, *Strategy for Data-Flow Synchronizations in Stencil Parallel Computations on Multi-/Manycore Systems*, The Journal of Supercomputing, 74 (2018), 1534–1546.
- [68] L. SZUSTAK, P. BRATEK, *Performance portable parallel programming of heterogeneous stencils across shared-memory platforms with modern Intel processors*, Int. Journal of High Performance Computing Applications, 33 (2019), 507–526.
- [69] L. SZUSTAK ET AL., *Architectural Adaptation and Performance-Energy Optimization for CFD Application on AMD EPYC Rome*, IEEE Trans. Parallel Distrib. Syst., 32 (2021), 2852 – 2866.
- [70] L. SZUSTAK, K. HALBINIAK, R. WYRZYKOWSKI, O. JAKL, *Unleashing the performance of ccNUMA multiprocessor architectures in heterogeneous stencil computations*, The Journal of Supercomputing, 75 (2019), 7765–7777.

- [71] L. SZUSTAK, M. LAWENDA, S. ARMING, G. BANKHAMER, C. SCHWEIMER, R. ELSÄSSER, *Profiling and optimization of python-based social sciences applications on hpc systems by means of task and data parallelism*, Future Generation Computer Systems, 148 (2023), 623–635.
- [72] L. SZUSTAK, R. WYRZYKOWSKI, K. K. HALBINIAK, P. BRATEK, *Toward Heterogeneous MPI+MPI Programming: Comparison of OpenMP and MPI Shared Memory Models*, Proc. Euro-Par 2019: Parallel Processing Workshops, 11997, Lect. Notes Comp. Sci., 2020, 270–281.
- [73] L. SZUSTAK, R. WYRZYKOWSKI, O. T., V. MELE, *Correlation of Performance Optimizations and Energy Consumption for Stencil-Based Application on Intel Xeon Scalable Processors*, IEEE Trans. Parallel Distrib. Syst., 31 (2020), 2582 – 2593.
- [74] TECHCRUNCH, *Google denies ai search features are killing website traffic*. <https://techcrunch.com/2025/08/06/google-denies-ai-search-features-are-killing-website-traffic/>, 2025.
- [75] *Technology Guide: Intel Speed Select Technology - Core Power*, 2022. <https://networkbuilders.intel.com/solutionslibrary/intel-speed-select-technology-core-power-intel-sst-cp-overview-technology-guide>.
- [76] D. TERPSTRA, H. JAGODE, H. YOU, J. DONGARRA, *Collecting performance data with PAPI-C*, Tools for High Performance Computing, 2010, 157–173.
- [77] *Top500 list*. <https://top500.org/lists/top500/>.
- [78] J. TREIBIG, G. HAGER, G. WELLEIN, *Likwid: A lightweight performance-oriented tool suite for x86 multicore environments*, Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures, San Diego CA, 2010.
- [79] R. VAN DAM, I. LANGKILDE-GEARY, D. VENTURA, *Adapting ADtrees for high arity features*, AAAI Conference on Artificial Intelligence, 2008.
- [80] R. VAN DAM, I. LANGKILDE-GEARY, D. VENTURA, *Adapting ADtrees for improved performance on large datasets with high-arity features*, Knowledge and Information Systems, 35 (2013).

-
- [81] A. WILLIAMS, *Język C++ i przetwarzanie współbieżne w akcji*, HELION, 2 red., 2019.
- [82] J. WINTER, C. ALBONESI D.H. SHOEMAKER, *Scalable Thread Scheduling and Global Power Management for Heterogeneous Many-Core Architecture*, Proc. PACT'10, 2010, 29–40.
- [83] R. WYRZYKOWSKI, K. ROJEK, L. SZUSTAK, *The Impact of Voltage-Frequency Scaling for the Matrix-Vector Product on the IBM POWER8*, Euro-Par 2016: 22nd European Conference on Parallel Processing, 9833, Springer, Lect. Notes Comp. Sci., 2016, 103–116.
- [84] C. YUAN, B. MALONE, X. WU, *Learning optimal Bayesian networks using A* search*, International Joint Conference on Artificial Intelligence, 2011, 2186–2191.
- [85] D. G. ZILL, W. S. WRIGHT, *Differential Equations with Boundary-Value Problems*, Brooks/Cole, Cengage Learning, Boston, MA, 8 red., 2012.