Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

Kierunek: **Informatyka Stosowana (IST)**
Specjalność: **Inżynieria Oprogramowania (IO)**

PRACA DYPLOMOWA MAGISTERSKA

Porównanie wybranych mechanizmów programowania współbieżnego i równoległego w językach Rust i C++

Comparison of selected concurrent and parallel programming mechanisms in Rust and C++

Rafał Jasiński

Opiekun pracy
dr inż. Zdzisław Spławski

WROCŁAW, 2024

Streszczenie

TEMPLATE Praca skupia się na projekcie i implementacji aplikacji wykorzystującej algorytmy genetyczne wraz z ich wizualizacją. Pierwsza część obejmuje teoretyczne podstawy tych algorytmów, porównując je do mechanizmów biologicznej genetyki. Omówiono schemat działania, historię oraz kluczowe elementy, takie jak osobnik, populacja, selekcja, krzyżowanie, mutacja i funkcja celu. Następnie przedstawiono założenia projektowe, obejmujące kodowanie osobnika, metody selekcji, operatory krzyżowania, opcje mutacji, funkcję celu, interfejs użytkownika, przykład użycia i strukturę aplikacji. Zawierają one opis ich zasady działania.

Implementacja aplikacji została opisana w kolejnym etapie, prezentując użyte technologie, wybrany język programowania wraz z interfejsem użytkownika i inne narzędzia. Szczegółowo omówiono implementację osobnika w kodowaniu binarnym, wybór wariantów operacji, metody selekcji, krzyżowania, mutacji, funkcji przystosowania oraz wygląd interfejsu użytkownika wraz z opisem.

Analiza wyników pracy obejmuje testy na danych testowych oraz porównanie różnych metod selekcji, krzyżowania i mutacji. Wnioski z porównań są przedstawione dla każdej badanej metody, dostarczając czytelnikowi kompleksowego spojrzenia na skuteczność poszczególnych elementów algorytmów genetycznych.

Całość pracy zawiera podsumowanie, gdzie prezentowane są główne osiągnięcia oraz wnioski podczas pisania pracy. Praca dostarcza wartościowego spojrzenia na zastosowanie algorytmów genetycznych w projektowaniu aplikacji, a także oferuje praktyczne wskazówki dotyczące implementacji i optymalizacji tych algorytmów.

Abstract

TEMPLATE The thesis focuses on the design and implementation of an application utilizing genetic algorithms along with their visualization. The first part covers the theoretical foundations of these algorithms, comparing them to the mechanisms of biological genetics. The operation scheme, history, and key elements such as individual, population, selection, crossover, mutation, and fitness function are discussed. The design assumptions are then presented, including individual encoding, selection methods, crossover operators, mutation options, the fitness function, user interface, usage example, and application structure. They contain a description of their principles of operation.

The application implementation is described in the next stage, presenting the technologies used, the chosen programming language along with the user interface, and other tools. The implementation of the individual in binary encoding, the selection of operation variants, selection methods, crossover, mutation, fitness function, and the appearance of the user interface are discussed in detail.

The results analysis includes tests on test data and a comparison of different selection, crossover, and mutation methods. Conclusions from the comparisons are presented for each investigated method, providing the reader with a comprehensive view of the effectiveness of individual elements of genetic algorithms.

The entire thesis includes a conclusion where the main achievements and conclusions drawn during the writing process are presented. The paper provides a valuable perspective on the application of genetic algorithms in application design and also offers practical guidance on the implementation and optimization of these algorithms.

Spis treści

Rozdział 1

Wstęp

1.1. Problem badawczy

Wraz z rozwojem nowoczesnych technologii informatycznych i rosnącą złożonością systemów obliczeniowych, znaczenia nabierają paradygmaty programowania, które pozwalają na maksymalne wykorzystanie zasobów współczesnego sprzętu komputerowego — w szczególności architektur wielordzeniowych. Programowanie współbieżne i równoległe stanowią obecnie podstawę projektowania wydajnych i niezawodnych aplikacji w wielu obszarach, od systemów operacyjnych, przez serwery wysokiej dostępności, aż po rozwiązania z zakresu sztucznej inteligencji czy gier komputerowych.

W kontekście tych wyzwań szczególnie interesujące staje się porównanie narzędzi, jakie oferują współczesne języki programowania. Niniejsza praca magisterska koncentruje się na analizie dwóch języków: Rust oraz C++, które – mimo odmiennej filozofii projektowej – są powszechnie wykorzystywane w systemach wymagających wysokiej wydajności. Rust, jako stosunkowo młody język [**?**], zdobywa coraz większą popularność[**?**] ze względu na nowatorskie podejście do bezpieczeństwa pamięci i współbieżności, opierające się na systemie własności ((ang. *ownership*)) oraz sprawdzaniu poprawności kodu na etapie kompilacji. Dzięki temu minimalizuje ryzyko wycieków pamięci, błędów synchronizacji czy wyścigów danych. Z kolei C++ – język dojrzały, o długiej historii i ugruntowanej pozycji w przemyśle – oferuje niezwykle szeroki wachlarz możliwości, jeśli chodzi o zarządzanie zasobami i niskopoziomową optymalizację, jednak często kosztem większego ryzyka błędów programistycznych.

Wybór tych dwóch języków podyktowany jest ich rosnącym znaczeniem w obszarach wymagających efektywnego zarządzania współbieżnością i równoległością. Rust jest promowany jako bezpieczna alternatywa dla C i C++ w systemach krytycznych [**?**], natomiast C++ nadal pozostaje filarem wielu aplikacji, w tym tych o kluczowym znaczeniu dla infrastruktury informatycznej. Porównanie ich możliwości w zakresie programowania współbieżnego i równoległego dostarcza cennych informacji dla praktyków inżynierii oprogramowania, projektantów systemów oraz badaczy eksplorujących nowe podejścia do zarządzania złożonością kodu.

W związku z powyższym, głównym problemem badawczym pracy są następujące pytania:

PB1: *Jakie są różnice i podobieństwa w podejściu do programowania współbieżnego i równoległego w językach Rust oraz C++ pod względem efektywności, bezpieczeństwa oraz dostępnych narzędzi?*

PB2: *W jaki sposób wybór konkretnego języka wpływa na wydajność i stabilność aplikacji współbieżnych oraz równoległych?*

Odpowiedź na to pytanie zostanie udzielona poprzez analizę teoretyczną, przegląd literatury oraz eksperymentalne porównanie konkretnych mechanizmów oferowanych przez oba języki. W ramach pracy przeprowadzone zostaną testy wydajnościowe oraz analiza, które pozwolą na

identyfikację różnic w podejściu do programowania współbieżnego i równoległego w językach Rust oraz C++.

1.2. Struktura pracy

Struktura pracy została zaplanowana w sposób umożliwiający systematyczne przedstawienie zagadnienia oraz przeprowadzenie kompleksowej analizy porównawczej. Po niniejszym wprowadzeniu, rozdział drugi precyzuje cel oraz zakres pracy, określając, które aspekty mechanizmów współbieżności i równoległości będą poddane szczegółowej analizie. Następnie, w rozdziale trzecim, przedstawione zostały podstawowe pojęcia związane z programowaniem współbieżnym i równoległym – zarówno od strony teoretycznej, jak i praktycznej – w celu zbudowania wspólnego kontekstu dla dalszych rozważań.

Rozdział czwarty zawiera przegląd literatury oraz wcześniejszych badań dotyczących wykorzystania języków Rust i C++ w projektowaniu systemów wielowątkowych. Zidentyfikowano w nim również istniejące luki badawcze oraz przedstawiono różnice w podejściu do bezpieczeństwa, wydajności i zarządzania pamięcią.

W dalszej części pracy – odpowiednio w rozdziałach piątym i szóstym – zaprezentowano konkretne mechanizmy programowania współbieżnego i równoległego dostępne w językach Rust oraz C++. Każdy z tych rozdziałów zawiera szczegółowe omówienie modeli pamięci, używanych bibliotek (np. Tokio, Rayon, std::thread, OpenMP), metod synchronizacji (mutexy, kanały, wartości atomowe), a także wybranych konstrukcji językowych wspierających bezpieczne współdzielenie danych między wątkami.

W rozdziale siódmym dokonano bezpośredniego porównania omawianych mechanizmów, koncentrując się na takich kryteriach jak zarządzanie wątkami, efektywność synchronizacji, narzut związany z bezpieczeństwem, a także wydajność obliczeniowa i sprzętowa.

Rozdział ósmy poświęcony jest analizie wyników eksperymentów, w ramach których porównano działanie wybranych algorytmów zaimplementowanych w obu językach, ze szczególnym uwzględnieniem czasów wykonania, zużycia zasobów oraz stabilności działania.

W końcowej części pracy, rozdział dziewiąty prezentuje wnioski oraz rekomendacje dotyczące praktycznego zastosowania języków Rust i C++ w projektach wymagających wysokiej wydajności i bezpieczeństwa współbieżnego. Pracę zamyka rozdział dziesiąty, który podsumowuje najważniejsze osiągnięcia badawcze oraz wskazuje możliwe kierunki dalszych analiz i rozwijania zaproponowanych rozwiązań.

1.3. Słownik wybranych pojęć

- **Własność (ang. *ownership*)** - system zarządzania pamięcią, który eliminuje konieczność używania automatycznego odśmiecania, jednocześnie zapobiegając błędom takim jak użycie po zwolnieniu czy podwójne zwolnienie.
- **Pożyczanie (ang. *borrow*)** - również występujący pod inną nazwą jako przenoszenie własności [3], jest to mechanizm pozwalający na używanie wartości bez przejmowania jej na własność. Dzięki temu możemy przekazywać dane do funkcji lub między częściami programu bez ich kopiowania czy przenoszenia.
- **Programowanie współbieżne** - to sposób wykonywania wielu zadań jednocześnie lub przeplatania ich w czasie, co zwiększa wydajność programu. Może być realizowane za pomocą wątków, procesów lub programowania asynchronicznego.
- **Programowanie równoległe** - to sposób wykonywania wielu zadań jednocześnie, co zwiększa wydajność programu. W odróżnieniu od programowania współbieżnego, programowanie równoległe polega na wykonywaniu zadań w tym samym czasie, a nie przeplataniu ich w czasie.
- **Wątek** - to pojedynczy proces, który może być wykonywany równolegle z innymi procesami. Wątki mogą współdzielić zasoby, takie jak pamięć, ale wymagają synchronizacji, aby uniknąć wyścigów danych.
- **Proces** - to instancja programu, która jest wykonywana w systemie operacyjnym. Procesy są izolowane od siebie i mają własne zasoby, takie jak pamięć i przestrzeń adresowa.
- **Wyścigi danych (ang. *Race conditions*)** - to sytuacja, w której dwa lub więcej wątków lub procesów próbuje modyfikować wspólną zmienną w tym samym czasie, co prowadzi do nieprzewidywalnych wyników.

Rozdział 2

Cel oraz zakres pracy

Celem niniejszej pracy jest przeprowadzenie pogłębionej analizy oraz wszechstronnego porównania mechanizmów programowania współbieżnego i równoległego w dwóch językach programowania: Rust i C++. Celem jest przedstawienie kluczowych różnic oraz podobieństw w podejściu do zarządzania wielowątkowością, analizując jednocześnie efektywność, bezpieczeństwo oraz wygodę stosowania narzędzi dostępnych w obu językach.

W ramach pracy szczególną uwagę poświęcono omówieniu wybranych bibliotek i frameworków, które wspierają tworzenie aplikacji wielowątkowych w Rust (np. Tokio, Rayon) i C++ (np. std::thread, OpenMP, TBB). Przeanalizowane zostaną mechanizmy bezpieczeństwa oraz zarządzania pamięcią i wątkami, które odgrywają kluczową rolę w zapewnieniu stabilności i wydajności aplikacji współbieżnych i równoległych.

Dodatkowym celem jest przeprowadzenie analizy wydajności oraz efektywności implementacji aplikacji wielowątkowych, co pozwoli na ocenę szybkości działania i efektywnego zarządzania zasobami w obu językach. Badanie uwzględni również aspekty praktyczne, takie jak łatwość użycia narzędzi, dostępność wsparcia ze strony społeczności oraz dojrzałość ekosystemu każdego z języków.

Aby zilustrować wyniki teoretyczne w praktyce, przeprowadzona zostanie implementacja aplikacji współbieżnych i równoległych w obu językach, co umożliwi porównanie osiągniętych wyników wydajnościowych oraz analizę różnic w strukturze i stylu kodu. Efektem pracy będzie również identyfikacja scenariuszy, w których jeden z języków może przewyższać drugi pod względem wydajności, bezpieczeństwa, czy wygody stosowania, co pozwoli na sformułowanie rekomendacji dotyczących wyboru języka w zależności od specyficznych wymagań projektowych.

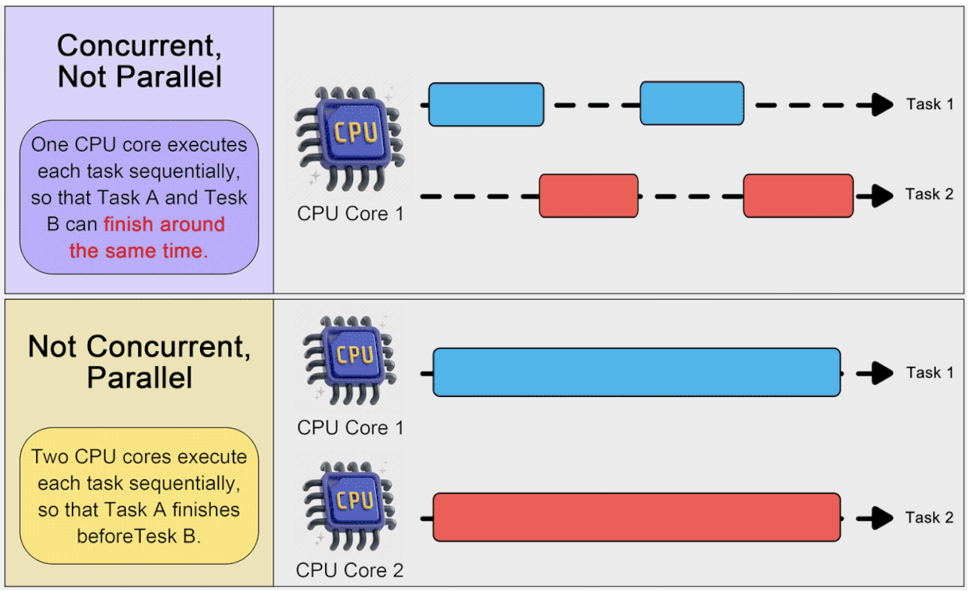
Rozdział 3

Wprowadzenie

Rozdział ten został poświęcony przybliżeniu czytelnikowi zasad działania programowania współbieżnego oraz równoległego, a także omówieniu tych mechanizmów w kontekście języków Rust i C++.

Współczesne systemy komputerowe stają się coraz bardziej złożone, a jednocześnie coraz bardziej wydajne dzięki rozwojowi technologii wielordzeniowych i wieloprocesorowych. W tej sytuacji programowanie współbieżne i równoległe zyskało kluczowe znaczenie, umożliwiając pełne wykorzystanie możliwości sprzętowych. Umiejętność projektowania oraz wdrażania aplikacji, które skutecznie zarządzają równoczesnym wykonywaniem wielu zadań, stała się niezbędna dla programistów tworzących oprogramowanie wymagające wysokiej wydajności oraz możliwości dalszego rozwoju. W niniejszej pracy autor postanowił skoncentrować się na dwóch nowoczesnych językach programowania — Rust i C++ — które oferują zaawansowane mechanizmy do zarządzania współbieżnością i równoległością, wspierając jednocześnie bezpieczeństwo i optymalne zarządzanie zasobami.

Programowanie współbieżne (ang. concurrent programming) oraz programowanie równoległe (ang. parallel programming) to dwa różne, lecz uzupełniające się podejścia, które umożliwiają organizację pracy wielu zadań w aplikacji - poglądowa różnica została zamieszczona na rysunku 3.1. Chociaż często używa się ich zamiennie, ich cechy i cele różnią się znacznie.



Rys. 3.1: Różnice między wykonywaniem zadań współbieżnie a równolegle [5]

3.1. Programowanie współbieżne

Programowanie współbieżne to podejście w projektowaniu aplikacji, które umożliwia równoczesne wykonywanie wielu zadań, choć faktycznie procesor wykonuje tylko jedno zadanie na raz. Dzięki technice tzw. "quasi-równoległości" użytkownik ma wrażenie, że zadania te są realizowane jednocześnie, ponieważ procesor przełącza się między nimi w bardzo krótkich odstępach czasu. Takie podejście jest kluczowe w aplikacjach interaktywnych, takich jak gry komputerowe, aplikacje mobilne czy serwisy internetowe, które muszą błyskawicznie reagować na różne zdarzenia (np. żądania użytkowników, kliknięcia czy komunikaty sieciowe) bez zauważalnych opóźnień.

Programowanie współbieżne pozwala na lepsze zarządzanie zadaniami w aplikacjach, które muszą obsługiwać wiele operacji jednocześnie, choć nie zawsze są to zadania wymagające intensywnych obliczeń. Przykładami takich aplikacji mogą być systemy obsługi żądań użytkowników na serwerach, aplikacje multimedialne czy interfejsy graficzne.

3.1.1. Mechanizmy realizujące współbieżność

Współbieżność różni się od programowania równoległego tym, że nie wymaga fizycznej wielordzeniowości procesora. Nawet na jednordzeniowym procesorze możliwe jest uzyskanie współbieżności, ponieważ procesor może w bardzo szybki sposób przełączać się między różnymi zadaniami. Tego rodzaju przełączanie nazywane jest "wirtualnym przełączaniem" i odbywa się na poziomie systemu operacyjnego, który odpowiedzialny jest za przeprowadzanie tego procesu w sposób niewidoczny dla użytkownika. Dzięki tej technice użytkownik nie zauważa, że procesor w danym momencie wykonuje tylko jedno zadanie, mimo że wiele z nich jest obsługiwanych "po kolei" w bardzo krótkich cyklach.

Kolejnym istotnym mechanizmem, który wspiera współbieżność, jest harmonogram zadań (ang. scheduler). Harmonogram jest odpowiedzialny za zarządzanie dostępem do procesora i przydzielanie zasobów obliczeniowych poszczególnym zadaniom. Dzięki zaawansowanym algorytmom harmonogramowania, system operacyjny decyduje, które zadanie ma być wykonane w danym czasie, jak długo ma trwać jego wykonanie, oraz kiedy procesor ma przełączyć się na inne zadanie. Harmonogram zadań może być dostosowywany w zależności od wymagań aplikacji, co pozwala na osiągnięcie optymalnej wydajności i minimalizację opóźnień.

3.1.2. Zastosowania programowania współbieżnego

Programowanie współbieżne jest niezwykle ważne w aplikacjach, które muszą reagować na różne wydarzenia użytkownika lub zewnętrzne zdarzenia w czasie rzeczywistym. Typowe zastosowania programowania współbieżnego obejmują:

- Aplikacje interaktywne – gry komputerowe, aplikacje mobilne, aplikacje desktopowe, które muszą natychmiast reagować na akcje użytkownika, jak kliknięcia, gesty czy komendy wprowadzane z klawiatury.
- Systemy serwerowe – serwisy internetowe, bazy danych, aplikacje chmurowe, które muszą jednocześnie obsługiwać wielu użytkowników, wykonując różne operacje, takie jak przetwarzanie zapytań, zapisywanie danych, czy obsługę sesji użytkowników.
- Przetwarzanie zdarzeń w czasie rzeczywistym – systemy monitoringu, systemy alarmowe, aplikacje do analizy danych strumieniowych, które muszą przetwarzać i reagować na dane napływające w czasie rzeczywistym.
- Multimedia – odtwarzanie wideo, transmisje strumieniowe, edycja audio i wideo, gdzie aplikacje muszą równocześnie obsługiwać wiele wątków.

3.1.3. Zalety z programowania współbieżnego

Główne zalety stosowania programowania współbieżnego w aplikacjach to:

- Zwiększenie responsywności – dzięki szybkiemu przełączaniu między zadaniami aplikacje stają się bardziej responsywne i wydajne, co jest szczególnie ważne w przypadku interfejsów użytkownika oraz aplikacji reagujących na dynamicznie zmieniające się dane.
- Lepsze wykorzystanie zasobów procesora – współbieżność pozwala na efektywne wykorzystanie mocy obliczeniowej procesora, nawet w przypadku procesorów jednordzeniowych. Przełączanie między zadaniami pozwala na ich efektywne wykonywanie w krótkich cyklach czasowych.
- Lepsze wykorzystanie zasobów procesora – współbieżność pozwala na efektywne wykorzystanie mocy obliczeniowej procesora, nawet w przypadku procesorów jednordzeniowych. Przełączanie między zadaniami pozwala na ich efektywne wykonywanie w krótkich cyklach czasowych.
- Skalowalność – aplikacje wykorzystujące współbieżność mogą być łatwiej skalowane na wiele rdzeni procesora lub urządzeń, dzięki czemu mogą obsługiwać większą liczbę użytkowników lub większe ilości danych.

3.1.4. Wady programowania współbieżnego

Pomimo wielu korzyści, programowanie współbieżne wiąże się również z pewnymi wyzwaniami:

- Złożoność synchronizacji – w przypadku współdzielenia zasobów, takich jak pamięć, konieczne jest odpowiednie zarządzanie dostępem do nich. Błędy synchronizacji mogą prowadzić do problemów takich jak wyścigi danych (ang. race conditions) lub zakleszczenia (ang. deadlocks), które mogą uniemożliwić poprawne działanie aplikacji.
- Problemy związane z wydajnością – chociaż współbieżność pozwala na szybsze przetwarzanie wielu zadań, jej realizacja może prowadzić do narzutów związanych z przełączaniem kontekstu i synchronizacją. W aplikacjach o dużym stopniu współzależności zadań, narzut ten może negatywnie wpływać na wydajność.
- Trudności w debugowaniu – aplikacje współbieżne są trudniejsze do debugowania, ponieważ błędy mogą występować sporadycznie i w zależności od kolejności przełączania wątków, co utrudnia ich wykrywanie i naprawę.

3.2. Programowanie równoległe

Programowanie równoległe to technika, która umożliwia równoczesne wykonywanie wielu zadań, wykorzystując wiele jednostek obliczeniowych. W tym podejściu zadania są fizycznie realizowane jednocześnie na różnych rdzeniach procesora lub innych jednostkach przetwarzających. Programowanie równoległe jest szczególnie użyteczne w aplikacjach wymagających znacznej mocy obliczeniowej, takich jak obliczenia w dziedzinie uczenia maszynowego, symulacje naukowe, przetwarzanie dużych zbiorów danych, rendering grafiki oraz aplikacje o wysokiej wydajności. Dzięki tej technice możliwe jest zredukowanie czasu wykonywania obliczeń, które w tradycyjnym, sekwencyjnym modelu zajmowałyby znacznie więcej czasu.

3.2.1. Zasady programowania równoległego

Programowanie równoległe opiera się na podziale złożonych zadań na mniejsze części, które mogą być realizowane jednocześnie. Aby osiągnąć równoległość, aplikacje muszą być zaprojektowane w sposób umożliwiający rozdzielenie obliczeń pomiędzy liczne rdzenie procesora

lub urządzenia obliczeniowe, takie jak karty graficzne (GPU). Każda część zadania, tak zwany wątek, może wykonywać obliczenia na niezależnych danych, a na końcu wyniki są zbierane i łączone, aby uzyskać końcowy rezultat.

W kontekście programowania równoległego, istnieje szereg modeli pamięci, które definiują metody przechowywania oraz dostępu do danych przez jednostki przetwarzające:

- Pamięć współdzielona (ang. Shared Memory Model) – wszystkie jednostki obliczeniowe dzielą wspólną pamięć, co umożliwia łatwą wymianę danych, ale wymaga odpowiedniej synchronizacji.
- Pamięć rozproszona (ang. Distributed Memory Model) – każda jednostka obliczeniowa ma swoją własną pamięć, a komunikacja między jednostkami odbywa się za pomocą przesyłania wiadomości (np. przy użyciu protokołu MPI – Message Passing Interface).
- Model hybrydowy – łączy elementy obu powyższych modeli, gdzie pamięć współdzieloną wykorzystują jednostki w ramach jednego węzła, a komunikacja między węzłami odbywa się przez przesyłanie wiadomości.

3.2.2. Zastosowanie programowania równoległego

Programowanie równoległe znajduje szerokie zastosowanie w różnych dziedzinach, w których wymagana jest ogromna moc obliczeniowa oraz szybkie przetwarzanie dużych zbiorów danych. Jednymi z kilku najczęściej wykorzystywanych zastosowań są:

- Uczenie maszynowe i sztuczna inteligencja (ang. AI) – w szczególności w kontekście głębokiego uczenia (ang. deep learning), gdzie trening modeli na dużych zbiorach danych wymaga wykonywania tysięcy operacji matematycznych jednocześnie. Dzięki równoległości można przyspieszyć proces uczenia, wykorzystując jednostki GPU, które są zoptymalizowane do obliczeń równoległych.
- Symulacje naukowe – w dziedzinach takich jak fizyka, chemia, biologia, gdzie tworzenie symulacji wymagających obliczeń na dużą skalę (np. symulacje molekularne, modelowanie zjawisk atmosferycznych, dynamika płynów) są realizowane na dużych klastrach komputerowych.
- Przetwarzanie dużych zbiorów danych (ang. Big Data) – analiza danych w czasie rzeczywistym lub w partiach, które pozwalają na rozdzielanie zadań przetwarzania danych na wiele maszyn.
- Rendering grafiki 3D – w grach komputerowych, filmach animowanych i inżynierii wizualnej, gdzie renderowanie obrazów i animacji wymaga intensywnych obliczeń graficznych. Programowanie równoległe umożliwia szybkie generowanie wysokiej jakości obrazów przez równoczesne przetwarzanie wielu elementów obrazu.

3.2.3. Zalety programowania równoległego

Poprzez wykorzystanie programowania równoległego można się spodziewać następujących korzyści:

- Zwiększenie wydajności – dzięki równoczesnemu przetwarzaniu wielu zadań, czas realizacji obliczeń jest znacznie skrócony.
- Lepsze wykorzystanie zasobów obliczeniowych – współczesne procesory, w tym wielordzeniowe CPU i GPU, oferują dużą moc obliczeniową, którą można efektywnie wykorzystać przy pomocy technik równoległych.
- Skalowalność – aplikacje równoległe mogą być skalowane w zależności od dostępnych zasobów obliczeniowych, umożliwiając zwiększenie wydajności przy rozwoju systemu.

3.2.4. Wady programowania równoległego

Każde rozwiązania niesie ze sobą zalety jak i wady czy też wyzwania implementacyjne, które się z nim wiążą. Programowanie równoległe wiąże się z kilkoma wyzwaniami, które wymagają szczególnej uwagi projektanta systemów:

- Złożoność projektowania – projektowanie systemów równoległych jest bardziej skomplikowane niż projektowanie aplikacji sekwencyjnych. Należy odpowiednio podzielić zadania na mniejsze jednostki, które można wykonać jednocześnie, oraz zadbać o ich synchronizację.
- Synchronizacja danych – w przypadku używania pamięci współdzielonej, należy odpowiednio synchronizować dostęp do danych, aby uniknąć błędów takich jak wyścigi danych (race conditions), które mogą prowadzić do nieprzewidywalnych wyników.
- Problemy komunikacyjne – w systemach rozproszonych, komunikacja między jednostkami przetwarzającymi może stać się wąskim gardłem, obniżającym wydajność systemu. W takich przypadkach konieczne jest optymalizowanie przepływu danych i unikanie zbędnych operacji komunikacyjnych.
- Overhead związany z równoległością – chociaż programowanie równoległe przyspiesza obliczenia, wprowadza również dodatkowy narzut związany z przełączaniem kontekstu między zadaniami, synchronizacją wątków i komunikacją. W przypadku niewielkich zadań, zysk z równoległości może nie przewyższać kosztów narzutu.

Technologie i narzędzia do programowania równoległego Do realizacji obliczeń równoległych dostępnych jest wiele narzędzi i bibliotek wspierających programistów w implementacji równoległych aplikacji. Do najpopularniejszych należą:

OpenMP (Open Multi-Processing) – biblioteka dla języków C, C++ i Fortran, która umożliwia programowanie równoległe w modelu pamięci współdzielonej. CUDA – platforma stworzona przez firmę NVIDIA, przeznaczona do programowania na procesorach graficznych (GPU), wykorzystywana głównie w zastosowaniach związanych z uczeniem maszynowym i obróbką grafiki. MPI (Message Passing Interface) – standard komunikacji w systemach z pamięcią rozproszoną

W tej pracy zostaną zbadane te dwa podejścia zarówno w kontekście języka Rust, jak i C++. Rust jest relatywnie młodym językiem, zaprojektowanym z myślą o bezpieczeństwie pamięci i unikaniu typowych błędów wielowątkowych. Z kolei C++ to język o ugruntowanej pozycji, znany z elastyczności i wysokiej wydajności, co czyni go popularnym wyborem dla aplikacji wymagających precyzyjnego zarządzania zasobami.

Rozdział 4

Przegląd literatury

Celem niniejszego rozdziału jest przedstawienie dotychczasowych badań i publikacji dotyczących mechanizmów programowania współbieżnego i równoległego w językach Rust i C++. Analiza literatury umożliwi zrozumienie aktualnego stanu wiedzy w tej dziedzinie, a także wskazanie na występujące luki badawcze, które niniejsza praca postara się wypełnić.

Przegląd literatury odbywał się z wykorzystaniem narzędzi baz danych oferujących wyszukiwanie, filtrowanie oraz przegląd prac: Scopus, Google Scholar.
W ramach bazy scopus wykorzystano następujące kwerendy do wyszukiwania - tabela 4.1

Tab. 4.1: Kwerendy użyte w bazie Scopus [1]

Lp.	Kwerenda	Liczba wyników
1	ALL ("concurrent programming" OR "parallel programming") AND (ALL ("Rust") AND ALL ("C++"))	444
2	ALL ("concurrent programming" OR "parallel programming") AND (ALL ("Rust") AND ALL ("C++")) AND (ALL ("compare"))	28

Autor zdecydował się również użyć nowego, wbudowane narzędzia w systemie Scopus - Scopus AI. Narzędzie to oparte na sztucznej inteligencji, wspomaga eksplorację akademicką w oparciu o dane z platformy Scopus. Dzięki integracji z narzędziem Copilot optymalizuje wyszukiwania, łącząc metody semantyczne i dopasowanie słów kluczowych. Choć Scopus AI ułatwia badania, jego wyniki warto weryfikować, ponieważ mogą zawierać nieścisłości lub stronniczość. Po wprowadzeniu tytułu pracy w języku angielskim jako kwerendę, Scopus AI zwrócił 9 wyników, biorąc pod uwagę kwerendę stworzoną na podstawie tytyułu pracy, zamieszczoną w listingu 4.1. Zwrócone prace pokrywają się z przeglądem umieszczonym w tabeli 4.1

Listing 4.1: Kwerenda wygenerowana przez AI

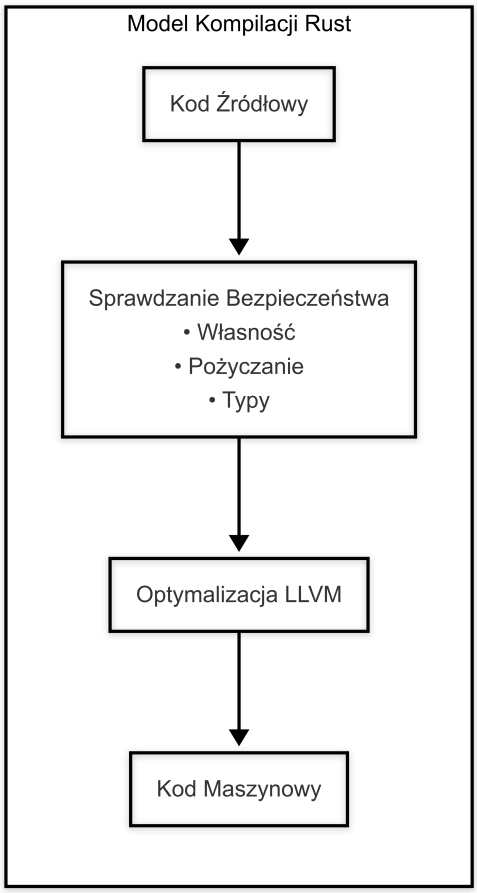
```
("concurrent programming" OR "parallel programming" OR "
   ↪ multithreading" OR "asynchronous")
AND ("Rust" OR "C++" OR "programming languages" OR "software
   ↪ development")
AND ("performance" OR "efficiency" OR "scalability" OR "resource
   ↪ management")
AND ("synchronization" OR "thread safety" OR "deadlock" OR "race
   ↪ condition")
AND ("libraries" OR "frameworks" OR "tools" OR "APIs")
```

[1] Ilość wyników dla poszczególnych zapytań może się różnić w zależności od daty (wyszukiwanie przeprowadzono w okresie listopad-luty 2024/25).

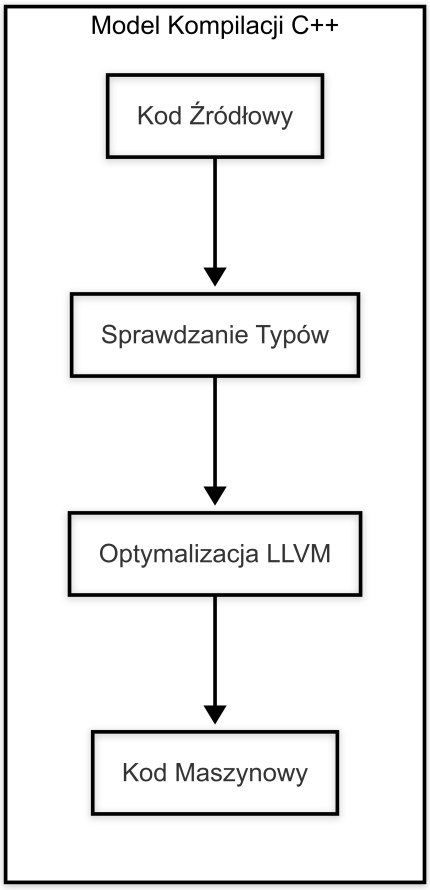
4.1. Porównanie Rust oraz C++

Porównanie języków programowania Rust i C++ jest przedmiotem licznych publikacji, które analizują ich różnorodne aspekty, takie jak struktura kodu, sposób kompilacji, bezpieczeństwo, wydajność oraz obsługa współbieżności i równoległości.

Jednym z podstawowych kryteriów porównań jest model kompilacji. Rust i C++ są językami kompilowanymi, co oznacza, że dedykowany kompilator tłumaczy kod źródłowy na kod maszynowy przed jego wykonaniem. Dzięki temu możliwe jest uzyskanie wysokiej wydajności programów. W literaturze [19] często podkreśla się, że Rust, w odróżnieniu od C++, kładzie większy nacisk na bezpieczeństwo pamięci oraz typów w czasie kompilacji, co ma kluczowe znaczenie w nowoczesnym oprogramowaniu. W kontekście C++ wskazuje się na jego większą elastyczność oraz bogaty ekosystem, który pozwala na szeroką gamę zastosowań, ale jednocześnie wymaga większej uwagi programistów w zakresie zarządzania pamięcią i synchronizacji wątków.



Rys. 4.1: Kroki kompilacji w języku Rust



Rys. 4.2: Kroki kompilacji w języku C++

Rys. 4.3: Porównanie kroków kompilacji w językach Rust i C++

Informacje o procesie kompilacji pochodzą z [19, 3, 15], które opisują integrację z LLVM i różnice w sprawdzaniu bezpieczeństwa. Na diagramie 4.1 drugi blok reprezentuje dodatkowe etapy sprawdzania bezpieczeństwa w Rust, które nie występują w C++. Z kolei na diagramie 4.2 drugi blok pokazuje podstawowe sprawdzanie typów w C++, które jest mniej rygorystyczne niż system Rust.

Kolejnym ważnym aspektem poruszanym w badaniach są różnice w modelach obsługi błędów. Rust implementuje mechanizm ownership oraz lifetimes, które pozwalają na jednoznaczne

zarządzanie cyklem życia zasobów w czasie kompilacji, minimalizując ryzyko wycieków pamięci [18]. C++ oferuje alternatywne rozwiązania, takie jak wskaźniki inteligentne (smart pointers), ale pozostawia programiście większą swobodę, co może prowadzić do potencjalnych błędów. [12, 9]

4.1.1. Bezpieczeństwo

Bezpieczeństwo języków Rust i C++ jest jednym z najczęściej analizowanych tematów w literaturze. W przypadku Rusta duży nacisk kładziony jest na eliminację całych klas błędów, takich jak null pointer dereferencing, data races oraz wycieki pamięci. Mechanizmy takie jak ownership, borrow checker oraz obowiązkowa mutowalność zmiennych (explicit mutability) są wymieniane jako kluczowe elementy zapewniające bezpieczeństwo.

Z drugiej strony, C++ umożliwia większą kontrolę nad pamięcią, co może być zaletą w systemach wymagających maksymalnej wydajności, ale jednocześnie wiąże się z koniecznością samodzielnego zarządzania zasobami przez programistów. W literaturze często podkreśla się, że to właśnie większa złożoność i ryzyko błędów w kodzie C++ skłoniły społeczność do stworzenia języków takich jak Rust.

Przykładowo, badania wskazują, że aplikacje napisane w Rusta są mniej podatne na błędy związane z wyścigami danych (data races), co ma szczególne znaczenie w środowiskach wielowątkowych. Z kolei w C++ stosowanie bibliotek takich jak std::thread czy frameworków typu OpenMP pozwala na osiągnięcie podobnych celów, choć wymaga od programistów większej uwagi w zakresie synchronizacji. [10, 23, 14]

4.1.2. Czas wykonania

Porównania czasów wykonania programów napisanych w Rust i C++ są częstym tematem analiz. W większości badań wskazuje się, że pod względem wydajności Rust jest konkurencyjny wobec C++, co wynika z podobnych mechanizmów kompilacji i optymalizacji kodu.

Jednak kluczową różnicą jest to, że Rust wprowadza pewne narzuty związane z kontrolą bezpieczeństwa w czasie kompilacji, które mogą wydłużyć czas budowania programu, ale nie wpływają znacząco na czas wykonania. W badaniach eksperymentalnych często porównuje się wydajność aplikacji w obliczeniach numerycznych, przetwarzaniu danych oraz w zadaniach wymagających intensywnego dostępu do pamięci.

C++ nadal pozostaje językiem preferowanym w projektach o krytycznym znaczeniu wydajnościowym, takich jak gry komputerowe, symulacje fizyczne czy systemy wbudowane, choć Rust zaczyna zdobywać popularność w tych obszarach ze względu na większe bezpieczeństwo przy porównywalnej wydajności. [28, 17, 8, 20]

4.1.3. Programowanie współbieżne oraz równoległe

Współbieżność i równoległość to kluczowe elementy programowania w językach Rust i C++. Oba języki oferują zaawansowane narzędzia i biblioteki do zarządzania wielowątkowością.

Rust wyróżnia się systemem ownership i wbudowanym mechanizmem wykrywania błędów współbieżności, co eliminuje wyścigi danych w czasie kompilacji. Narzędzia takie jak Tokio i Rayon pozwalają na łatwe tworzenie i zarządzanie zadaniami asynchronicznymi i równoległymi.

C++ z kolei oferuje wsparcie dla wielowątkowości poprzez standardową bibliotekę (std::thread) oraz zaawansowane szkielety aplikacyjne (frameworks), takie jak OpenMP czy TBB (Threading Building Blocks). Chociaż te narzędzia są niezwykle potężne, nie zapewniają automatycznej ochrony przed błędami współbieżności, co wymaga większej ostrożności ze strony programistów.

Strona [21] szczegółowo analizuje różnice w podejściu do współbieżności w obu językach, podkreślając, że Rust dzięki swojemu modelowi zarządzania pamięcią oferuje większe bezpieczeństwo, podczas gdy C++ pozostaje bardziej elastyczny, co może być korzystne w bardziej specyficznych scenariuszach.

W odniesieniu do artykułów, czasopism oraz materiałów konferencyjnych opublikowanych w latach 2022-2024, zidentyfikowanie prac, które jednoznacznie koncentrują się na problematyce pracy, jest wyzwaniem. Można natomiast znaleźć publikacje, które wykorzystują wspomniane języki (Rust oraz C++) i ich porównanie w kontekście złożonym do problemu pracy - chociażby wykorzystanie języka Rust w programowaniu układów wbudowanych [27], jako zamiennik dla dotychczasowych języków z rodziny C.

Można również znaleźć pracę, która przedstawia wykorzystanie biblioteki odpowiedzialnej za współbieżność FastFlow przez oba języki Rust oraz C++ [24]. Pokazuje ona, że język Rust jest dobrą alternatywą dla języka C++ w kontekście współbieżności.

Rozdział 5

Wybrane mechanizmy w języku Rust

o czym rozdział

5.1. Podejście do współbieżności i równoległości

Rozwój języka Rust oferuje szereg funkcji, które czynią go dobrym wyborem do programowania współbieżnego oraz równoległego. Kluczowymi elementami, które wyróżniają go na tle innych języków programowania są [3]:

1. Własność (ang. *ownership*) i pożyczanie (ang. *borrow*): Model własności języka Rust zapewnia, że dane są bezpiecznie współdzielone między wątkami bez ryzyka wyścigów danych (ang. *races*).
2. Nieustraszona współbieżność (ang. *Fearless Concurrency*): System typów Rusta wymusza reguły w czasie kompilacji, pozwalając programistom na pisanie współbieżnego kodu bez obawy o typowe pułapki.
3. Inteligentne wskaźniki (ang. *smart pointers*): Abstrakcje wspierające korzystanie oraz przesyłanie danych w programowaniu współbieżnym oraz równoległym.

5.1.1. Ownership oraz borrow

Własność (ang. *ownership*) jest fundamentalnym konceptem w języku Rust, który zapewnia bezpieczeństwo pamięci bez konieczności stosowania mechanizmu odśmiecania (ang. *garbage collection*). Każda wartość posiada jednego właściciela, który jest odpowiedzialny za zwolnienie pamięci zajmowanej przez tę wartość, gdy wychodzi ona z zakresu (ang. *scope*). Model własności zapobiega wyścigom danych i zapewnia efektywne zarządzanie pamięcią.
Kluczowe zasady własności:

- Każda wartość ma jednego właściciela.
- Gdy właściciel wychodzi z zakresu, wartość zostaje usunięta (ang. *ropped*).
- Własność może zostać przeniesiona (ang. *moved*) na inną zmienną.

Pożyczanie (ang. *borrowing*) umożliwia tworzenie referencji do wartości bez przejmowania jej własności. Jest to kluczowe dla umożliwienia różnym częściom programu dostępu do danych bez ich duplikowania. Rust pozwala na pożyczanie wartości według dwóch zasad:

- Pożyczanie niemutowalne (domyślne) (ang. *immutable*): Można utworzyć wiele referencji, ale żadna z nich nie może modyfikować wartości.

- Pożyczanie mutowalne (ang. *mutable*): W danym momencie może istnieć tylko jedna mutowalna referencja, co zapobiega wyścigom danych.

Listing 5.1: Przykład mechanizmu borrow

```
fn main() {
    let s1 = String::from("Hello World");
    let s2 = &s1; // Immutable borrow - pożyczka niemutowalna
    println!("{}", s2); // Ok
    // let s3 = &mut s1; // Error
}
```

W powyższym listingu 5.1 zaprezentowano element dotyczący użycia pożyczki w języku Rust. Błąd, oznaczony jako *Error* wynika z faktu, iż nie jest możliwe pożyczenie obiektu *s3* jako mutowalnego, ponieważ jest on jednocześnie pożyczany jako niemutowalny do obiektu *s2* (domyślnie)

5.1.2. Nieustraszona współbieżność

Mechanizm ten wynika bezpośrednio z rygorystycznych reguł systemu typów i modelu własności, które są integralną częścią tego języka. Dzięki temu Rust pozwala na tworzenie kodu współbieżnego, minimalizując ryzyko wystąpienia typowych błędów, takich jak wyścigi danych czy nieokreślone zachowanie wynikające z użycia wskaźników do już zwolnionej pamięci (ang. *dangling pointers*).

Rust wymusza bezpieczeństwo współbieżności w czasie kompilacji poprzez analizę własności i okresu życia danych (ang. *lifetimes*). Mechanizm ten zapobiega jednoczesnemu mutowalnemu dostępowi do tych samych danych w różnych wątkach (opisane w punkcie 5.1.1), co eliminuje potrzebę ręcznego zarządzania pamięcią czy synchronizacji przez programistę. Takie podejście czyni współbieżność w Rust nie tylko bezpieczną, ale również przewidywalną, co znacząco ułatwia jej implementację. [31]

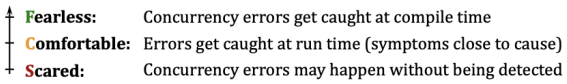


Figure 2: A spectrum of fear in parallel programming.

Rys. 5.1: Spektrum nieustraszonej współbieżności [1]

Autorzy [1] uważają, że nieustraszona współbieżność jest lepiej interpretowana jako spektrum, zilustrowane na 5.1 idealnie eliminując wszelkie obawy przed błędami współbieżności w czasie kompilacji (Fearless), ale jeśli nie jest to możliwe, utrzymując objawy błędów w czasie wykonywania blisko ich przyczyn (Comfortable) lub w inny sposób nie dając gwarancji odtworzenia przyczyny ani objawu (Scared).

Przetłumaczono z DeepL.com (wersja darmowa)

5.1.3. Inteligentne wskaźniki (ang. *Smart Pointers*)

Smart pointery w języku Rust stanowią zaawansowane abstrakcje, które poszerzają funkcjonalność tradycyjnych wskaźników poprzez wbudowane automatyczne zarządzanie pamięcią oraz semantykę własności. Stanowią one kluczowy element gwarantujący bezpieczeństwo pamięci w Rust, umożliwiając pisanie bezpiecznego oraz odpornego na błędy kodu bez konieczności

korzystania z garbage collectora. Ponadto, smart pointery są wykorzystywane zarówno w programowaniu współbieżnym, jak i równoległym, wspierając kontrolowaną oraz bezpieczną wymianę danych. Poniżej omówione zostaną trzy najczęściej używane smart pointery w Rust: Box, Rc oraz Arc.

Box

Własność Danych na Stercie Smart pointer Box służy do alokacji pamięci na stercie, oferując prosty i efektywny sposób zarządzania własnością dużych struktur danych lub typów rekurencyjnych. Przenosząc dane na stertę, Box redukuje wykorzystanie stosu, co czyni go idealnym rozwiązaniem w sytuacjach, gdy rozmiar danych może się zmieniać lub nie jest znany w czasie kompilacji.
Wykorzystuje się go do przechowywania dużych struktur danych, obsługi typów rekurencyjnych, których rozmiar nie może być określony w czasie kompilacji.

Listing 5.2: Box smart pointer

```
fn main() {
    let b = Box::new(5); // Alokacja liczby całkowitej na stercie
    println!("{}", b);   // Dostęp do wartości za pomocą wskaźnika Box
}
```

Rc: Wskaźnik z Liczeniem Referencji

Smart pointer Rc (Reference Counted) umożliwia współdzielenie własności danych przez wiele części programu. Automatycznie śledzi liczbę referencji do danych i usuwa je dopiero wtedy, gdy ostatnia referencja zostanie usunięta. Rc nie jest jednak bezpieczny dla wątków i powinien być używany tylko w przypadkach programów jednowątkowych.
Najczęściej wykorzystywany podczas współdzielenia danych pomiędzy różnymi częściami programu w środowiskach jednowątkowych, implementacji struktur danych, takich jak grafy czy drzewa z węzłami współdzielonymi.

Listing 5.3: Rc smart pointer

```
use std::rc::Rc;

fn main() {
    let a = Rc::new(5);       // Utworzenie licznika referencji
    let b = Rc::clone(&a);    // Klonowanie zwiększa licznik referencji
    println!("{}", b);        // Dostęp do współdzielonych danych
}
```

Arc: Atomowy Wskaźnik z Liczeniem Referencji

Dla programowania współbieżnego Rust oferuje Arc (Atomic Reference Counted), czyli wersję **Rc**, która jest bezpieczna dla wątków. Wykorzystuje operacje atomowe do bezpiecznego współdzielenia danych pomiędzy wątkami, gwarantując poprawne aktualizacje liczników referencji bez ryzyka wyścigów danych.

Listing 5.4: Arc smart pointer

```
use std::sync::Arc;
```

```rust
use std::thread;

fn main() {
    let a = Arc::new(5);      // Utworzenie atomowego licznika
    ↪ referencji
    let a_clone = Arc::clone(&a); // Klonowanie dla bezpiecznego współ
    ↪ dzielenia

    let handle = thread::spawn(move || {
        println!("{}", a_clone); // Dostęp do współdzielonych danych w
    ↪  nowym wątku
    });

    handle.join().unwrap();  // Oczekiwanie na zakończenie wątku
}
```

Niebezpieczny Rust (Unsafe Rust)

TBD [2]

5.2. Programowanie współbieżne

Współbieżność to zdolność systemu do obsługi wielu zadań, które potencjalnie mogą się nakładać w czasie. Współbieżność odnosi się do zdolności systemu do jednoczesnego obsługiwania wielu zadań, które mogą mieć miejsce w tym samym czasie. Język Rust został zaprojektowany z uwzględnieniem bezpieczeństwa i wydajności w kontekście współbieżności, co czyni go niezwykle atrakcyjnym narzędziem dla programistów zajmujących się systemami wielowątkowymi.

5.2.1. Model własności pamięci

Centralnym elementem podejścia Rusta do współbieżności jest model własności pamięci, który umożliwia programistom tworzenie kodu współbieżnego bez ryzyka wystąpienia błędów związanych z niebezpiecznym dostępem do współdzielonej pamięci. Mechanizm ten, oparty na koncepcjach własności, pożyczania oraz systemu typów, pozwala kompilatorowi Rust na statyczne wykrywanie potencjalnych problemów, takich jak wyścigi danych (data races). Własność pamięci zapewnia, że tylko jeden wątek może posiadać mutowalny dostęp do danej zmiennej w danym czasie, eliminując tym samym możliwość niespodziewanej ingerencji ze strony innych wątków 5.1.1.

5.2.2. Biblioteki

std::thread

Tokio

Rayon

crossbeam::channel

actix

5.2.3. Wątki

Pule wątków (ang. *Thread Pools*)

Kradzież zadania (ang. *Work Stealing*)

5.2.4. Kanały

Kanały (ang. *channels*) to mechanizm w języku Rust, który umożliwia bezpieczną i efektywną komunikację między wątkami. Kanały są jednym z kluczowych elementów modelu współbieżności w tym języku, zapewniając sposób przesyłania danych z jednego wątku do drugiego, jednocześnie minimalizując ryzyko problemów związanych z współdzieleniem pamięci. Kanały są oparte na wzorcu producenta-konsumenta, gdzie jeden wątek (producent) wysyła dane, a inny wątek (konsument) je odbiera.

W Rust kanały są częścią standardowej biblioteki i implementują model komunikacji oparty na kolejkach **FIFO** (*First In, First Out*). W kontekście Rust'a kanały są realizowane przez struktury ***Sender*** (nadawca) i ***Receiver*** (odbiorca), które stanowią mechanizm do przesyłania danych między wątkami. Kanały zapewniają synchronizację pomiędzy wątkami, eliminując potrzebę bezpośredniego współdzielenia pamięci w sposób, który mógłby prowadzić do niepożądanych efektów ubocznych, takich jak błędy związane z równoległym dostępem do tej samej przestrzeni pamięci.

Kanały w Rust działają na zasadzie przekazywania wartości z wątku do wątku. Są one bezpieczne, ponieważ Rust zapewnia, że nie wystąpią wyścigi danych — Sender jest odpowiedzialny za wysyłanie danych, a Receiver za ich odbiór. Rust automatycznie zapewnia synchronizację, więc nie ma potrzeby stosowania dodatkowych mechanizmów, jak blokady mutexów, do zarządzania dostępem do pamięci.

Tworzenie kanałów

Kanał można stworzyć za pomocą funkcji *mpsc::channel()* z modułu *std::sync*. Ta funkcja zwraca dwie wartości: nadawcę (Sender) i odbiorcę (Receiver).

Listing 5.5: Przykład tworzenia kanału

```
use std::sync::mpsc;
use std::thread;

fn main() {
    // Tworzenie kanału
    let (tx, rx) = mpsc::channel();

    // Tworzenie wątku producenta
```

```
    thread::spawn(move || {
        let value = String::from("Hello from the producer!");
        tx.send(value).expect("Failed to send message");
    });

    // Odbiór wiadomości w wątku konsumenta
    let received = rx.recv().expect("Failed to receive message");
    println!("Received: {}", received);
}
```

Opis wykonanych kroków w celu utworzenia kanałów w listingu 5.5:

1. Tworzymy kanał za pomocą *mpsc::channel()*, który zwraca parę (tx, rx) — tx jest nadawcą, a rx odbiorcą.
2. Tworzymy nowy wątek (producenta), który wysyła wiadomość ("*Hello from the producer!*") do kanału.
3. W głównym wątku (konsument) odbieramy wiadomość za pomocą *recv()* i drukujemy ją na ekranie.

Wysyłanie i odbieranie danych

Nadawca (Sender) jest używany do wysyłania danych do kanału. Można wysłać dowolny typ, który jest przesyłany przez kanał. Funkcja send() jest używana do wysyłania wartości, a recv() w odbiorcy blokuje wątek do momentu otrzymania danych.
Odbiorca (Receiver) odbiera dane z kanału. Jest to blokująca operacja, co oznacza, że wątek odbiorcy będzie czekał, aż dane będą dostępne do odebrania.

Przykład wielokrotnego odbioru

Kanały w Rust są domyślnie jednokierunkowe, co oznacza, że tylko jeden odbiorca może odbierać wiadomości z kanału. Możemy jednak tworzyć wiele kanałów i różne wątki odbiorcze, aby efektywnie rozdzielać zadania.

Listing 5.6: Przykład z wieloma wątkami

```
use std::sync::{mpsc, Arc, Mutex};
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();
    let rx = Arc::new(Mutex::new(rx));

    // Tworzenie 3 wątków konsumentów
    for i in 0..3 {
        let rx = Arc::clone(&rx);  // Klonowanie Arc, nie Receiver
        thread::spawn(move || {
            let message = rx.lock().unwrap().recv().expect("Failed to
    ↪ receive message");
            println!("Consumer {} received: {}", i, message);
        });
    }

    // Wysyłanie wiadomości do konsumentów
```

```
    for i in 0..3 {
        let message = format!("Message␣{}", i);
        tx.send(message).expect("Failed␣to␣send␣message");
    }

    // Zatrzymanie wątku głównego, aby konsument mógł zakończyć pracę
    thread::sleep(std::time::Duration::from_secs(1));
}
```

W tym przykładzie 5.6 tworzymy trzy wątki konsumentów, z których każdy odbiera wiadomości z tego samego kanału. Wątek główny wysyła trzy wiadomości, które są odbierane przez konsumentów.

- Dodatkowo została użyta struktura ***Arc<Mutex<Receiver»*** 5.1.3 do umożliwienia współdzielenia odbiorcy (Receiver) między wątkami. Mutex zapewnia synchronizację dostępu do kanału, dzięki czemu tylko jeden wątek na raz może odbierać wiadomości.
- ***Arc::clone*** klonuje Arc, a nie Receiver. *Arc* tworzy nowy uchwyt do tego samego obiektu w pamięci, dzięki czemu wszystkie wątki mogą uzyskać dostęp do tego samego kanału.
- ***lock()*** Każdy wątek przed odebraniem wiadomości blokuje mutex, aby uzyskać dostęp do kanału w bezpieczny sposób.

Przykładowa odpowiedź programu:

```
Consumer 0 received: Message 0
Consumer 2 received: Message 2
Consumer 1 received: Message 1
```

Zakończenie pracy z kanałami

Kanał w Rust jest "ograniczony"— po wysłaniu wszystkich danych nadawca (sender) automatycznie zakończy swoje działanie, co powoduje, że funkcja recv() w odbiorcy zwróci błąd, jeśli nie będzie więcej wiadomości. Można także zakończyć kanał, jeśli w danym wątku nie ma już nadawców.

Listing 5.7: Zakończenie kanału

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        tx.send(String::from("End␣of␣messages")).expect("Send␣failed")
    ↪ ;
    });

    match rx.recv() {
        Ok(msg) => println!("Received:␣{}", msg),
        Err(_) => println!("No␣more␣messages!"),
    }
    match rx.recv() {
        Ok(msg) => println!("Received:␣{}", msg),
```

```
        Err(_) => println!("No more messages!"),
    }
}
```

W tym przykładzie 5.7 po wysłaniu wiadomości przez nadawcę wątku głównego, kanał zostaje zamknięty, co sprawia, że dalsze wywołanie recv() w odbiorcy zwróci błąd.
Odpowiedź programu:

```
Received: End of messages
No more messages!
```

Kanały poprzeczne (ang. *Crossbeam Channels*)

Aktorzy

5.2.5. Synchronizacja

5.2.6. Mutex i RwLock

Dla sytuacji wymagających współdzielenia pamięci Rust oferuje synchronizowane struktury, takie jak Mutex (ang. *mutual exclusion*) oraz RwLock. Umożliwiają one zarządzanie dostępem do danych w sposób bezpieczny, jednocześnie wymagając od programisty jawnego określenia momentów blokady i odblokowania zasobów.

5.2.7. Wartości atomiczne

Rust obsługuje także operacje na typach atomowych, które pozwalają na wykonywanie niepodzielnych operacji na współdzielonych zmiennych bez potrzeby stosowania bardziej zaawansowanych mechanizmów synchronizacji.

5.2.8. Bariery i semafory

5.2.9. Asynchroniczność

Chociaż współbieżność i asynchroniczność nie są tożsame, w Rust oba te podejścia są ze sobą ściśle powiązane. Rust implementuje asynchroniczność za pomocą konstrukcji takich jak async/await oraz bibliotek takich jak Tokio czy async-std. Podejście to pozwala na wykonywanie wielu operacji współbieżnie w ramach pojedynczego wątku, eliminując narzut związany z tworzeniem wielu wątków.

5.3. Programowanie równoległe

5.3.1. Biblioteki

Rozdział 6

Wybrene mechanizmy w języku C++

6.1. Programowanie współbieżne

6.2. Programowanie równoległe

6.2.1. OpenMP

Biblioteka ta pozwala uruchomić wybraną część kodu na wielu wątkach. Działa to podobnie do uruchamiania kody z wykorzystaniem GPU - kod, który zostanie uruchomiony na wielu wątkach umieszczany jest w specjalnym zakresie (ang. *scope*). Kompilacja kodu z OpenMP odbywa się z wykorzystaniem flagi `-fopenmp` w przypadku kompilatora `g++`.
Użycie pamięci cache może pomóc w zwiększeniu wydajności czasowej programu kosztem pamięci. W przypadku OpenMP, zmienna `shared` oznacza, że zmienna jest współdzielona pomiędzy wątkami, a `private` oznacza, że każdy wątek ma swoją kopię zmiennej?

Rozdział 7

Porównanie mechanizmów w językach Rust oraz C++

7.1. Programowanie współbieżne

7.1.1. Zarządzanie wątkami

Porównanie tworzenia i zarządzania wątkami obejmuje następujące parametry:

- czas tworzenia wątku (w mikrosekundach),
- narzut pamięciowy na wątek (w KB)

7.1.2. Wydajność synchronizacji

Kluczowe aspekty synchronizacji to:

- liczba operacji blokowania i odblokowywania mutexa na sekundę,
- opóźnienie przesyłania wiadomości przez kanały (w mikrosekundach),
- czas przełączania kontekstu pomiędzy wątkami (w mikrosekundach).

7.1.3. Narzut bezpieczeństwa

W kontekście narzutu związanego z mechanizmami bezpieczeństwa w językach Rust i C++ analizowane będą następujące metryki:

- czas kompilacji kodu współbieżnego (w sekundach),
- rozmiar pliku binarnego (w KB),
- zużycie pamięci podczas wykonywania programu (w MB).

7.1.4. Wybrane algorytmy do analizy wydajności

Dla porównania mechanizmów współbieżności wybrano następujące algorytmy:

- Model producent-konsument (z wykorzystaniem aktorów),
- Problem filozofów (synchronizacja dostępu do zasobów).

Algorytmy te pozwalają na analizę zdolności języków Rust i C++ do efektywnego zarządzania współbieżnością w obliczeniach numerycznych.

7.2. Programowanie równoległe

7.2.1. Wydajność obliczeniowa

Główne metryki oceny wydajności algorytmów równoległych to:

- czas wykonania algorytmów równoległych (w milisekundach),
- współczynnik przyspieszenia (T1/Tn) dla różnych liczby wątków,
- efektywność (przyspieszenie/liczba procesorów).

7.2.2. Wydajność sprzętowa (GFLOP/s)

Wydajność obliczeniowa mierzona w jednostkach GFLOP/s (gigaflops per second) pozwala na ocenę efektywności wykorzystania sprzętu:

- wydajność w pojedynczym wątku,
- wydajność wielowątkowa,
- efektywność skalowania (GFLOP/s na rdzeń).

Dodatkowo przeprowadzona zostanie analiza zgodnie z prawem Amdahla w celu określenia teoretycznych ograniczeń przyspieszenia obliczeń.

7.2.3. Zasoby systemowe

Analiza zużycia zasobów przez algorytmy równoległe obejmuje:

- procentowe wykorzystanie CPU,
- zużycie pamięci w warunkach obciążenia,
- współczynnik nietrafień w cache,
- liczbę operacji wejścia-wyjścia na sekundę.

7.2.4. Wybrane algorytmy do analizy wydajności

Dla porównania mechanizmów równoległości wybrano następujące algorytmy:

- Mnożenie macierzy (Matrix Multiplication),
- Transformata Fouriera (Fast Fourier Transform, FFT).

Algorytmy te pozwolą na analizę efektywności zarządzania zasobami oraz synchronizacji w językach Rust i C++.

Wybór powyższych benchmarków pozwoli na szczegółową analizę wydajności oraz stabilności obu języków w kontekście programowania współbieżnego i równoległego, umożliwiając sformułowanie rekomendacji dotyczących wyboru narzędzi w zależności od specyfiki projektu.

Rozdział 8

Analiza wyników

8.1. Programowanie współbieżne

8.2. Programowanie równoległe

Rozdział 9

Wnioski oraz rekomendacje

The option of developing new computer languages may be the clean- est and most efficient way to provide support for parallel processing. However, practical issues make the wide acceptance of a new computer language close to impossible. Nobody likes to rewrite old code to new languages. It is difficult to justify such effort in most cases. Also, educating and convincing a large enough group of developers to make a new lan- guage gain critical mass is an extremely difficult task.

Rozdział 10

Podsumowanie

W trakcie realizacji pracy udało się osiągnąć zrealizować wszystkie postawione cele, którym było stworzenie aplikacji umożliwiającej wizualizację zmian w populacji dla różnych wariantów algorytmu genetycznego. W ramach projektu udało się skutecznie zaimplementować interaktywne narzędzie, umożliwiające śledzenie dynamiki ewolucji populacji w czasie rzeczywistym. Wykonanie tego zadania nie obyło się jednak bez wyzwań implementacyjnych. Jednym z głównych problemów był wstępny brak używania konstruktora kopiującego, co wprowadziło nieścisłości podczas analizy wyników. Dodatkowo literatury, który reprezentowały selekcję rankingową znacząco się od siebie różniły, co wprowadzało pewne zamieszanie podczas interpretacji. Implementacja algorytmów genetycznych, różnych wariantów selekcji, krzyżowania i mutacji, wymagała skrupulatnego podejścia do detali oraz zoptymalizowania procesów obliczeniowych. Podczas pracy nad projektem jak i ich praktyczna implementacja pozwoliła na głębsze zrozumienie działania algorytmów genetycznych oraz ich zastosowań w dziedzinie optymalizacji.
Praca ta stanowiła również doskonałą okazję do rozwinięcia umiejętności programistycznych, szczególnie w obszarze tworzenia interfejsu graficznego i manipulacji danymi w czasie rzeczywistym.
W kontekście dalszego rozwoju projektu sugeruje się rozważenie dodania nowych wariantów parametrów algorytmów genetycznych (selekcja, krzyżowanie, mutacja), tak aby użytkownik miał możliwość porównania ich efektywności. Jeżeli chodzi zaś o architekturę projektu, to można by wydzielić komponenty, aby umożliwić w przyszłości wymianę biblioteki odpowiedzialnej za interfejs użytkownika. Pozwoli to na implementację zaawansowanych funkcji wizualizacyjnych, które mogą ułatwić zrozumienie procesu ewolucji populacji
Podsumowując, praca na projektem aplikacji pozwoliła na pogłębienie wiedzy z zakresu algorytmów ewolucyjnych oraz rozwinięcie umiejętności programistycznych w języku Java. Praca ta stanowi dobrą podstawę do dalszych badań nad algorytmami genetycznymi i ich praktycznym zastosowaniem.

Spis rysunków

Spis tabel

Spis listingów

Bibliografia

[1] J. Abdi, G. Posluns, G. Zhang, B. Wang, M. C. Jeffrey. When is parallelism fearless and zero-cost with rust? *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '24, strona 27–40, New York, NY, USA, 2024. Association for Computing Machinery.

[2] V. Astrauskas, C. Matheja, F. Poli, P. Müller, A. J. Summers. How do programmers use unsafe rust? *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–27, 2020.

[3] J. Blandy, J. Orendorff, L. Tindall. *Programming Rust*. O'Reilly Media, 2021.

[4] M. Bos. *Rust Atomics and Locks: Low-Level Concurrency in Practice*. O'Reilly Media, 2022.

[5] ByteByteGo. EP108: How do we design a secure system? — blog.bytebytego.com. `https://blog.bytebytego.com/p/ep108-how-do-we-design-a-secure-system?utm_campaign=post&utm_medium=web`. Dostęp 12-03-2025.

[6] R. Chandra. *Parallel Programming in OpenMP*. High performance computing. Elsevier Science, 2001.

[7] T.-C. Chen, M. Dezani-Ciancaglini, N. Yoshida. On the preciseness of subtyping in session types: 10 years later. *Proceedings of the 26th International Symposium on Principles and Practice of Declarative Programming*, PPDP '24, New York, NY, USA, 2024. Association for Computing Machinery.

[8] M. Costanzo, E. Rucci, M. Naiouf, A. De Giusti. Performance vs programming effort between rust and c on multicore architectures: Case study in n-body. *2021 XLVII Latin American Computing Conference (CLEI)*, strony 1–10. IEEE, 2021.

[9] Z. Fehervari. Rust vs C++: Modern Developers' Dilemma — bluebirdinternational.com. `https://bluebirdinternational.com/rust-vs-c/`. [Dostęp 28-01-2025].

[10] K. R. Fulton, A. Chan, D. Votipka, M. Hicks, M. L. Mazurek. Benefits and drawbacks of adopting a secure programming language: Rust as a case study. *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*, strony 597–616, 2021.

[11] M. Galvin. *Mastering Concurrency And Parallel Programming: Atain Advanced Techniques and Best Practices for Crafting Robust, Manitainable, and High-Performance Concurrent Code in C++*. 2024.

[12] Y. Gusakov. C++ Vs. Rust: 6 Key Differences — QIT — qit.software. `https://qit.software/c-vs-rust-6-key-differences/`. [Dostęp 28-01-2025].

[13] R. Innovation. Mastering Rust Concurrency & Parallelism: Ultimate Guide 2024 — rapidinnovation.io. `https://www.rapidinnovation.io/post/concurrent-and-`

parallel-programming-with-rust#2-basics-of-rust-for-concurrent-programming. [Dostęp 23-12-2024].

[14] R. Jung, J.-H. Jourdan, R. Krebbers, D. Dreyer. Safe systems programming in rust. *Communications of the ACM*, 64(4):144–152, 2021.

[15] S. Klabnik, C. Nichols. *The Rust Programming Language*. No Starch Press, USA, 2018.

[16] B. Köpcke, S. Gorlatch, M. Steuwer. Descend: A safe gpu systems programming language. *Proceedings of the ACM on Programming Languages*, 8, 2024. Cited by: 0; All Open Access, Gold Open Access, Green Open Access.

[17] S. Lankes, J. Breitbart, S. Pickartz. Exploring rust for unikernel development. *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*, strony 8–15, 2019.

[18] P. Larsen. Migrating C to Rust for Memory Safety . *IEEE Security & Privacy*, 22(04):22–29, Lip. 2024.

[19] K. Lesiński.

[20] Y. Lin, S. M. Blackburn, A. L. Hosking, M. Norrish. Rust as a language for high performance gc implementation. *ACM SIGPLAN Notices*, 51(11):89–98, 2016.

[21] M. Lindgren. Introduction - Comparing parallel Rust and C++ — parallel-rust-cpp.github.io. https://parallel-rust-cpp.github.io. [Dostęp 28-01-2025].

[22] M. Mitzenmacher, E. Upfal. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis*. Cambridge University Press, 2017.

[23] A. Pinho, L. Couto, J. Oliveira. Towards rust for critical systems. *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, strony 19–24. IEEE, 2019.

[24] L. Rinaldi, M. Torquati, M. Danelutto. Enforcing reference capability in fastflow with rust. *Advances in Parallel Computing*, 36:396 – 405, 2020. Cited by: 0; All Open Access, Gold Open Access.

[25] J. Sible, D. Svoboda. Rust software security: A current state assessment. Carnegie Mellon University, Software Engineering Institute's Insights (blog), Dec 2022. Dostęp: 2025-Mar-12.

[26] B. Troutwine. *Hands-On Concurrency with Rust: Confidently build memory-safe, parallel, and efficient software in Rust*. Packt Publishing, 2018.

[27] T. Vandervelden, R. De Smet, D. Deac, K. Steenhaut, A. Braeken. Overview of embedded rust operating systems and frameworks. *Sensors*, 24(17), 2024. Cited by: 0; All Open Access, Gold Open Access.

[28] R. Viitanen. Evaluating memory models for graph-like data structures in the rust programming language: Performance and usabiliy, 2020.

[29] V. Q. T. (vuquangtrong@gmail.com). Introduction to Parallel Programming in C++ with OpenMP - Stephan O'Brien — physics.mcgill.ca. https://www.physics.mcgill.ca/~obriens/Tutorials/parallel-cpp/. [Dostęp 23-12-2024].

[30] A. Williams. *C++ Concurrency in Action*. Manning, 2019.

[31] Z. Yu, L. Song, Y. Zhang. Fearless concurrency? understanding concurrent programming safety in real-world rust software. *arXiv preprint arXiv:1902.01906*, 2019.