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. 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.

1.2. 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++.

1.3. 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.

1.4. Słownik wybranych pojęć

- **licznik Redis** - to mechanizm wykorzystujący bazę danych Redis do przechowywania i aktualizowania liczników w czasie rzeczywistym. Redis, jako szybka baza typu klucz-wartość, pozwala na błyskawiczne operacje inkrementacji i dekrementacji wartości przypisanej do danego klucza.
- **LLVM** - (ang. *Low Level Virtual Machine*) - to zestaw narzędzi i bibliotek do budowania kompilatorów, który umożliwia generowanie, analizę i optymalizację kodu (zarówno w czasie kompilacji, jak i wykonania). LLVM nie jest maszyną wirtualną w tradycyjnym sensie, ale raczej infrastrukturą kompilatora, która operuje na pośrednim języku reprezentacji (LLVM IR), z którego może generować kod maszynowy dla różnych architektur.
- **nieustraszona współbieżność** - (ang. *fearless concurrency*) - to podejście do programowania współbieżnego, które eliminuje problemy związane z bezpieczeństwem pamięci i synchronizacją wątków. W Rust osiągnięto to dzięki systemowi własności, który zapewnia, że dane mogą być modyfikowane tylko przez jeden wątek na raz, eliminując ryzyko wyścigów danych i błędów synchronizacji.
- **pożyczanie** (ang. *borrow*) - również występujący pod inną nazwą jako przenoszenie własności [21], 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.
- **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.
- **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.
- **programowanie współbieżne** - technika programistyczna polegająca na jednoczesnym wykonywaniu wielu zadań lub ich przeplataniu w czasie, mająca na celu zwiększenie efektywności działania programu. Współbieżność może być realizowana z wykorzystaniem wielu wątków, procesów bądź mechanizmów programowania asynchronicznego, które wewnętrznie mogą operować na wątkach lub innych zasobach udostępnianych przez system operacyjny.
- **SIMD** - (ang. *Single Instruction, Multiple Data*) - pojedyncza instrukcja wykonywana na wielu danych jednocześnie. Jest to technika optymalizacji wydajności obliczeń, która wykorzystuje jednostki wektorowe dostępne w nowoczesnych procesorach.
- **wątek** - część programu wykonywana współbieżnie w obrębie jednego procesu - w jednym procesie może istnieć wiele wątków. Główna różnica między procesem a wątkiem polega na tym, że wszystkie wątki należące do tego samego procesu współdzielą przestrzeń adresową oraz inne zasoby systemowe, takie jak listy otwartych plików czy gniazda sieciowe. Natomiast każdy proces dysponuje własnym, odrębnym zestawem zasobów.
- **własność** (ang. *ownership*) - system zarządzania pamięcią, który eliminuje konieczność używania automatycznego odśmiecania, jednocześnie zapobiegając błędom takim jak użycie po zwolnieniu czy podwójne zwolnienie.
- **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.

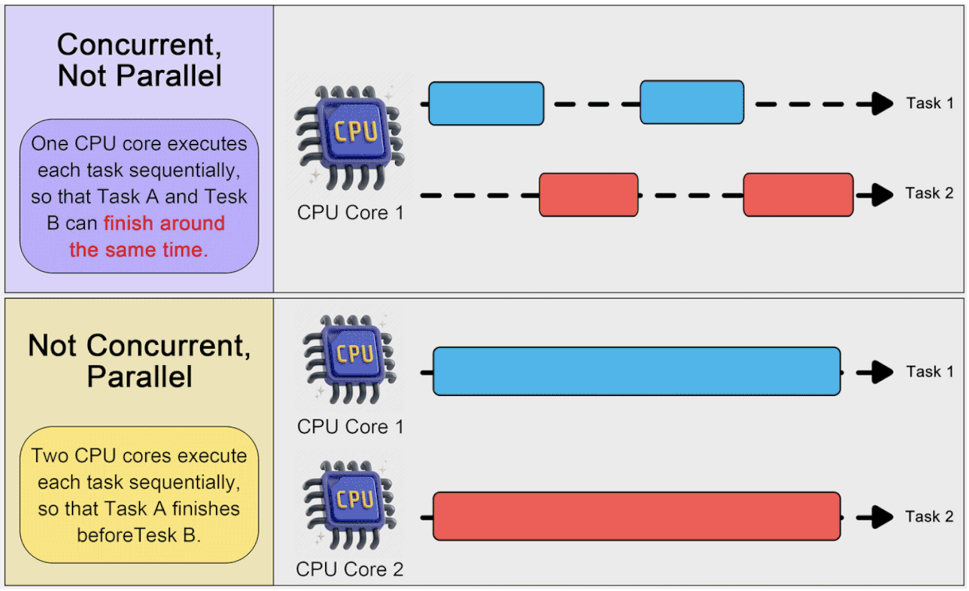
Rozdział 2

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 [22, 62, 25, 61]. W niniejszej pracy autor postanowił zbadać podejścia do programowania równoległego jak i współbieżnego 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.

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 2.1. Chociaż często używa się ich zamiennie, ich cechy i cele różnią się znacznie.



Rys. 2.1: Różnice między wykonywaniem zadań współbieżnie a równolegle [24]

2.1. Programowanie współbieżne

Programowanie współbieżne to podejście w projektowaniu aplikacji, które umożliwia równoczesne wykonywanie wielu zadań, choć faktycznie procesor wykonuje tylko jedno zadanie w danej chwili. Dzięki technice tzw. "quasi-równoległości" [50] 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 [32].

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 [22].

2.1.1. 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 [62, 23].

Kolejnym istotnym mechanizmem, który wspiera współbieżność, jest program szeregujący bądź też planista (ang. scheduler). 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. Planista zadań może być dostosowywany w zależności od wymagań aplikacji, co pozwala na osiągnięcie optymalnej wydajności i minimalizację opóźnień [62].

2.1.2. 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 [56, 62]:

- 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.

2.1.3. Zalety programowania współbieżnego

Główne zalety stosowania programowania współbieżnego w aplikacjach to [11, 62]:

- 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.
- 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.

2.1.4. Wady programowania współbieżnego

Pomimo wielu korzyści, programowanie współbieżne wiąże się również z pewnymi wyzwaniami [11, 57]:

- 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.

2.2. 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 [32, 14].

2.2.1. 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 [14].

Modele pamięci

W kontekście programowania równoległego, istnieje szereg modeli pamięci, które definiują metody przechowywania oraz dostępu do danych przez jednostki przetwarzające [14]:

- 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.

Taksonomia Flynn'a

Zrównoleglenia

2.2.2. 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 [49]:

- 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.

2.2.3. Zalety programowania równoległego

Poprzez wykorzystanie programowania równoległego można się spodziewać następujących korzyści [49]:

- 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.

2.2.4. 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 [49, 14]:

- 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.
- Narzut 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. [61, 25]
- CUDA – platforma stworzona przez firmę NVIDIA, przeznaczona do programowania na procesorach graficznych (GPU), wykorzystywana głównie w zastosowaniach związanych z uczeniem maszynowym i obróbką grafiki [14].
- MPI (Message Passing Interface) – standard komunikacji w systemach z pamięcią rozproszoną

Rozdział 3

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. Na samym wstępie zostały postawione następujące pytania do przeglądu literatury, które pomogą zrozumieć oraz sprawdzić aktualny stan wiedzy jeżeli chodzi o porównanie języków Rust oraz C++:

PPL1: *Jakie główne koncepcje/teorie dominują w literaturze dotyczącej porównania języków Rust oraz C++?*

PPL2: *Jakie metody badawcze są najczęściej stosowane do analizy różnic pomiędzy językami?*

PPL3: *Jak wygląda porównanie dostępności i dojrzałości bibliotek do programowania współbieżnego i równoległego w obu językach?*

PPL4: *Czy istnieją systematyczne metodologie porównywania języków programowania w kontekście współbieżności, które można zastosować do analizy Rust i C++?*

PPL5: *Jakie aspekty programowania współbieżnego i równoległego w Rust i C++ nie zostały dostatecznie zbadane w literaturze?*

PPL6: *Jaki jest stan wiedzy na temat wykorzystania programowania współbieżnego w ramach GPU w językach Rust i C++?*

Odpowiedzi na powyższe pytania pozwolą na zidentyfikowanie kluczowych obszarów, które wymagają dalszych badań oraz na wskazanie na potencjalne kierunki rozwoju w dziedzinie programowania współbieżnego i równoległego w językach Rust i C++.

3.0.1. Metodologia przeglądu literatury

Proces przeglądu literatury został zrealizowany zgodnie z zasadami przeglądu systematycznego, co oznaczało zastosowanie jasno określonych kryteriów selekcji i wyłączenia. Główne źródła literaturowe obejmowały artykuły naukowe, materiały konferencyjne oraz dokumentację techniczną. Wyszukiwanie przeprowadzono w renomowanych bazach danych naukowych oraz repozytoriach zawierających publikacje z zakresu inżynierii oprogramowania i języków programowania. Dodatkowo zostały również uwzględnione źródła internetowe oraz dokumentacje techniczne.

Przegląd literatury odbywał się z wykorzystaniem narzędzi baz danych oferujących wyszukiwanie, filtrowanie oraz przegląd prac: Scopus, Google Scholar.

3.0.2. Kryteria selekcji oraz wyłączenia

W procesie selekcji literatury uwzględniano przede wszystkim publikacje wydane po 2012 roku, co wynika z faktu, iż w tym właśnie roku zadebiutował język Rust [10]. Wyjątek stanowiły prace o charakterze ogólnym lub takie, które nie odnosiły się bezpośrednio do języka Rust, lecz zawierały istotne informacje dla problematyki badawczej niniejszej pracy.

Analiza obejmowała literaturę w języku polskim oraz angielskim, przy czym zdecydowana większość źródeł stanowiły publikacje anglojęzyczne. Selekcja materiałów opierała się na zgodności tematycznej z zakresem badań. W przypadku wątpliwości co do adekwatności danej pozycji, decyzja o jej włączeniu do przeglądu podejmowana była na podstawie analizy streszczenia. Jeśli po tej analizie publikacja wydawała się istotna, przechodzono do pełnej oceny jej treści.

Publikacje, które po dogłębnej analizie okazywały się nieodpowiednie dla głównego problemu badawczego, nie były uwzględniane w zasadniczej części pracy. Niemniej jednak, jeśli przyczyniły się do lepszego zrozumienia badanego zagadnienia lub pomogły w odpowiedzi na pytania do przeglądu literatury 3, były one odnotowywane jako materiały pomocnicze. Prace niespełniające powyższych kryteriów lub te, które nie są dostępne za pośrednictwem dostępnych metod (bądź też braku odpowiedzi twórców o prośbę udostępnienia pracy) były wykluczane z dalszej analizy.

3.0.3. Baza Scopus

W ramach bazy Scopus wykorzystano następujące kwerendy do wyszukiwania - tabela 3.3

Tab. 3.1: 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
3	(TITLE-ABS-KEY(("concurrent programming" OR "parallel programming") AND ("Rust" AND "C++"))) AND (TITLE-ABS-KEY("comparison" OR "evaluation" OR "benchmark"))	6
4	(TITLE-ABS-KEY(("thread" OR "async" OR "future" OR "actor model" OR "message passing" OR "shared memory") AND ("Rust" AND "C++"))) AND (TITLE-ABS-KEY("comparison" OR "performance" OR "evaluation"))	50
5	(TITLE-ABS-KEY(("Rust" AND "C++") AND ("concurrency model" OR "parallel constructs" OR "multithreading"))) AND (TITLE-ABS-KEY("comparison" OR "study"))	2

W celu identyfikacji literatury związanej z porównaniem wybranych mechanizmów programowania współbieżnego i równoległego w językach Rust i C++, opracowano pięć zapytań w bazie Scopus, z których każde miało określony cel badawczy. Pierwsze zapytanie miało na celu uzyskanie ogólnego przeglądu literatury, wyszukując wszystkie dokumenty, w których występują jednocześnie zagadnienia programowania współbieżnego lub równoległego oraz języki Rust i C++, niezależnie od kontekstu. Pozwoliło to oszacować ogólną skalę badań łączących

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

te zagadnienia. Drugie zapytanie zawężało zakres wyszukiwania poprzez dodanie słowa kluczowego „compare”, co umożliwiło wyodrębnienie publikacji, w których dokonano bezpośredniego porównania języków Rust i C++ w kontekście współbieżności lub równoległości. Dzięki temu uzyskano bardziej ukierunkowany zbiór literatury odnoszącej się do analizy porównawczej. Trzecie zapytanie charakteryzowało się większą precyzją, ograniczając wyniki do tytułów, streszczeń oraz słów kluczowych, i uwzględniało wyłącznie publikacje zawierające odniesienia do ewaluacji, porównań bądź benchmarków języków Rust i C++. Takie podejście pozwoliło wyselekcjonować najbardziej tematycznie powiązane prace. Czwarte zapytanie miało charakter bardziej techniczny, koncentrując się na konkretnych mechanizmach współbieżności, takich jak wątki, asynchroniczność, obiekty typu futury, model aktorów, przesyłanie komunikatów czy pamięć współdzielona, w połączeniu z terminami dotyczącymi wydajności i oceny. Umożliwiło to dotarcie do badań analizujących niskopoziomowe aspekty działania tych mechanizmów w obu językach. Piąte zapytanie skupiało się na poziomie koncepcyjnym, wyszukując publikacje zawierające takie terminy jak model współbieżności, konstrukty równoległe czy wielowątkowość, wraz z frazami dotyczącymi porównań lub analiz. Celem było zidentyfikowanie prac badających różnice w podejściu do współbieżności na poziomie architektury języka i jego konstrukcji wewnętrznych.

Autor zdecydował się również użyć nowego, wbudowanego 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 3.1. Zwrócone prace pokrywają się z przeglądem umieszczonym w tabeli 3.3

Listing 3.1: 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")
```

Na podstawie zapytań wykonanych w bazie Scopus zidentyfikowano publikacje odpowiadające tematyce współbieżności i równoległości w językach Rust i C++ (prace w ramach poszczególny zapytaniach się powtarzały). Jednak z uwagi na ograniczenia czasowe oraz objętość wyników, szczegółowej analizie poddano jedynie pierwsze 15 stron wyników, jeżeli było ich więcej, co odpowiada około 150 pracom.

Proces selekcji, obejmujący kolejne etapy oceny tematycznej, lektury streszczeń i wybranych pełnych tekstów zamieszczono w tabeli 3.2.

Tab. 3.2: Przebieg selekcji literatury (baza Scopus)

Etap	Opis	Liczba prac
1	Prace wyjściowe (przejrzane – pierwsze 15 stron wyników)	247
2	Wstępna selekcja tematyczna – usunięcie prac niezwiązanych bezpośrednio z tematem badania	97
3	Lektura streszczeń – eliminacja pozycji bez wartości empirycznej lub porównawczej	39
4	Analiza pełnych treści – wybór prac zawierających konkretne porównania, benchmarki lub studia przypadków	12
5	Prace kluczowe dla problemu badawczego – porównania/omówienie mechanizmów w Rust i C++	9

3.0.4. Baza Google Scholar

Tab. 3.3: Kwerendy użyte w bazie Scopus [2]

Lp.	Kwerenda	Liczba wyników
1	Comparison of selected concurrent and parallel programming mechanisms in Rust and C++	321

Podobnie jak w przypadku bazy Scopus, ze względu na dużą ilosc wyników zostało wzięte pod uwagę pierwsze 10 stron bazy (100 wyników). Proces selekcji, obejmujący kolejne etapy oceny tematycznej, lektury streszczeń i wybranych pełnych tekstów, przebiegał według poniższego schematu zamieszczono w tabeli 3.4

Tab. 3.4: Przebieg selekcji literatury (baza Google Scholar)

Etap	Opis	Liczba prac
1	Prace wyjściowe (przejrzane – pierwsze 10 stron wyników)	100
2	Wstępna selekcja tematyczna – usunięcie prac niezwiązanych bezpośrednio z tematem badania	37
3	Lektura streszczeń – eliminacja pozycji bez wartości empirycznej lub porównawczej	9
4	Analiza pełnych treści – wybór prac zawierających konkretne porównania, benchmarki lub studia przypadków	8
5	Prace kluczowe dla problemu badawczego – porównania/omówienie mechanizmów w Rust i C++	4

3.1. 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. Choć istnieje szeroka literatura porównawcza, wciąż stosunkowo nieliczne prace skupiają się w sposób bezpośredni i systematyczny na porównaniu mechanizmów współbieżnego i równoległego przetwarzania w tych dwóch językach. W ostatnich latach pojawiły się jednak wartościowe opracowania, również te akademickie,

[2]Liczba wyników dla poszczególnych zapytań może się różnić w zależności od daty (wyszukiwanie przeprowadzono w okresie marzec-kwiecień 2025).

które podejmują to zagadnienie. Pomimo tego, że liczba takich prac nadal jest ograniczona, ich jakość i rosnące zainteresowanie środowiska akademickiego wskazują na istotny potencjał badawczy w obszarze porównań paradygmatów współbieżnych w nowoczesnych językach systemowych. Dostępne opracowania stanowią wartościowe punkty odniesienia i uzasadniają potrzebę kontynuacji prac empirycznych w tym zakresie, co znajduje swoje odzwierciedlenie także w niniejszej pracy.
W literaturze można znaleźć prace, które analizują różnice między Rustem a C++ w kontekście bezpieczeństwa, wydajności, zarządzania pamięcią oraz obsługi błędów.

3.1.1. 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 oraz minimalizując ryzyko wycieków pamięci [40].

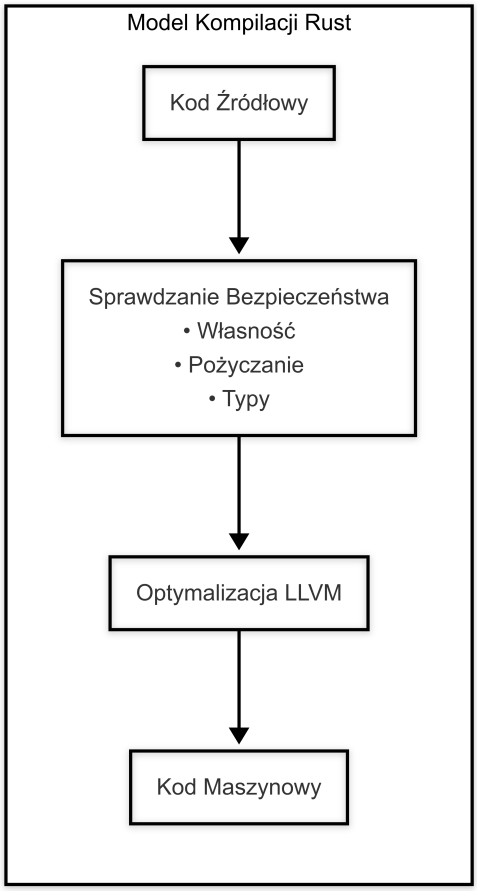
Z drugiej strony, C++ umożliwia większą kontrolę nad pamięcią, co może być zaletą w systemach wymagających maksymalnej wydajności, ale jednocześnie wiąże się z koniecznością samodzielnego zarządzania zasobami przez programistów. W literaturze [33, 30] często podkreśla się, że to właśnie większa złożoność i ryzyko błędów w kodzie C++ skłoniły społeczność do stworzenia języków takich jak Rust.

Przykładowo, badania [31, 51, 36] wskazują, że aplikacje napisane w Rust są mniej podatne na błędy związane z wyścigami danych (ang. *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. Dodatkowo są również prace [63, 55], które przedstawiają próby implementacji mechanizmów wbudowanych w język Rust (prawo własności, pożyczka) do języka C.

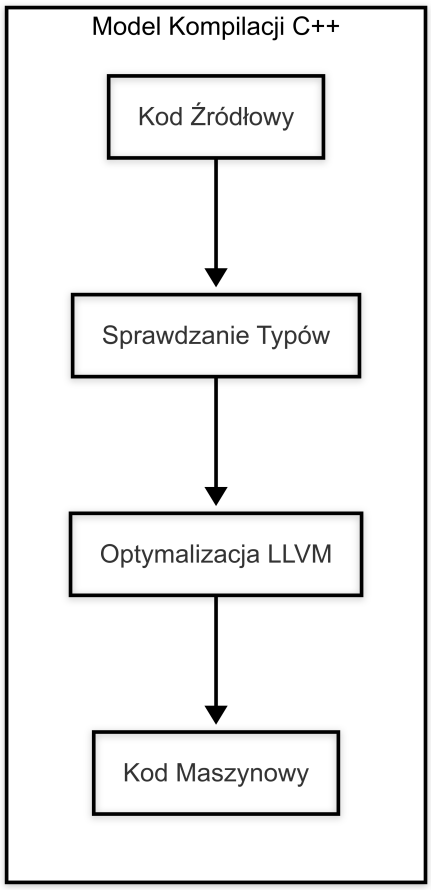
3.1.2. Czas wykonania

Porównania czasów wykonania programów napisanych w Rust i C++ są częstym tematem analiz [60, 39, 27, 42]. W badaniach tych zostało wykazane, że pod względem wydajności Rust jest konkurencyjny wobec C++, co wynika z 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.
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 [41] 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. 3.1: Kroki kompilacji w języku Rust



Rys. 3.2: Kroki kompilacji w języku C++

Informacje o procesie kompilacji pochodzą z [41, 21, 37], które opisują integrację z LLVM i różnice w sprawdzaniu bezpieczeństwa. Na diagramie 3.1 drugi blok reprezentuje dodatkowe etapy sprawdzania bezpieczeństwa w Rust, które nie występują w C++. Z kolei na diagramie 3.2 drugi blok pokazuje podstawowe sprawdzanie typów w C++, które jest mniej rygorystyczne niż system Rust. Ze względu na różnicę w procesie kompulacji kodu powstają główne różnice w bezpieczeństwie i wydajności obu języków.

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.

3.1.3. 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 własności (ang. *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 [43] 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. Jednak jak sam autor [43] wskazuje, należy zwrócić uwagę, że większość sztuczek optymalizacyjnych pokazanych w tym porównaniu to jedynie adaptacje oryginalnych rozwiązań C++ w języku Rust. Koncentruje się ona na praktycznym porównaniu języków Rust i C++ pod względem równoległego przetwarzania, szczególnie na poziomie niskopoziomowych operacji i synchronizacji wątków. Znajdują się tam benchmarki pokazujące różnice w czasie wykonywania programów, narzędzia diagnostyczne oraz zestawienie wydajności w kontekście SIMD, wątków, oraz komunikacji międzypamięciowej.

W pracy akademickiej Brandefelta i Heymana [34] dokonano porównania wydajności oraz złożoności implementacyjnej aplikacji wielowątkowej napisanej w trzech językach: Rust, C++ oraz Java. Badanie to wykazało, że Rust oferuje zbliżoną wydajność do C++, przy jednoczesnym znacznym uproszczeniu kodu (mniejsza liczba linii kodu), co przekłada się na łatwiejsze utrzymanie i potencjalnie mniejszą podatność na błędy. Autorzy wskazują, że Rust, dzięki swojemu systemowi własności (ang. *ownership*), eliminuje całe klasy błędów związanych z zarządzaniem pamięcią, bez konieczności stosowania odśmiecania.

Kolejne istotne opracowanie przedstawiono w publikacji Martinsa et al. [45], gdzie zaprezentowano implementację zestawu NAS (NASA Advanced Supercomputing) Parallel Benchmarks (NPB) [8] w języku Rust. Zestaw NPB stanowi uznany w środowisku naukowym standard do oceny wydajności systemów równoległych. W badaniach wykazano, że Rust osiąga porównywalną, a miejscami wyższą wydajność względem C++ i Fortrana (w wersjach OpenMP i sekwencyjnych), co wskazuje na jego rosnący potencjał w dziedzinie wysokowydajnych obliczeń (HPC - High-Performance Computing). Jednocześnie zwrócono uwagę na trudności implementacyjne, związane z restrykcyjnym modelem bezpieczeństwa języka Rust, które jednak mogą być złagodzone poprzez zastosowanie odpowiednich bibliotek (np. Rayon).

Równolegle, w pracy Besozziego [19] zaprezentowano bibliotekę Parallelo Parallel Library (PPL) – autorskie rozwiązanie do strukturalnego programowania równoległego w języku Rust. Biblioteka ta opiera się na wzorcach programowania równoległego takich jak skeletons i design patterns, oferując wysokopoziomowe abstrakcje ułatwiające implementację złożonych aplikacji równoległych. Przeprowadzone testy wykazały, że PPL dorównuje lub przewyższa popularne biblioteki takie jak Rayon, jednocześnie zachowując zgodność z idiomami języka Rust i zasadą fearless concurrency. Zastosowanie takich abstrakcji sprzyja zwiększeniu czytelności i poprawności kodu, co może mieć szczególne znaczenie w kontekście projektów systemowych.

Można również znaleźć pracę, która przedstawia wykorzystanie biblioteki odpowiedzialnej za współbieżność FastFlow przez oba języki Rust oraz C++ [52]. Pokazuje ona, że język Rust jest dobrą alternatywą dla języka C++ w kontekście współbieżności.

3.2. Podsumowanie

Na podstawie przeglądu literatury oraz zadanych pytań do przeglądu literatury można wskazać następujące odpowiedzi:

PPL1: Jakie główne koncepcje/teorie dominują w literaturze dotyczącej porównania języków Rust oraz C++?

W literaturze dominują badania porównawcze bezpieczeństwa, wydajności oraz zarządzania pamięcią w językach Rust i C++ - bardziej szczegółowo opisane w podrozdziałach 3.1.1 oraz 3.1.2. W kontekście współbieżności i równoległości można zauważyć znaczący wzrost prac w ubiegłych latach, co przedstawia przegląd opisany w podrozdziale 3.1.3.

PPL2: Jakie metody badawcze są najczęściej stosowane do analizy różnic pomiędzy językami ?

W literaturze oraz w dotychczasowych analizach [16, 17, 2, 7] wskazano szereg kryteriów istotnych przy porównywaniu lub ocenie języków programowania ogólnego przeznaczenia. Do najważniejszych należą:

- Prostota konstrukcji języka, mająca bezpośredni wpływ na łatwość programowania i zrozumiałość kodu
- Czytelność kodu, związana z jego późniejszą konserwacją i rozwijaniem
- Dostosowanie języka do konkretnego zastosowania, co wpływa na wydajność i efektywność programów
- Szybkość kompilacji
- Efektywność działania programu, zarówno pod względem szybkości, jak i zużycia zasobów systemowych,
- Dostępność i jakość bibliotek, frameworków oraz narzędzi wspierających rozwój oprogramowania
- Wsparcie w procesie debugowania, profilowania i testowania kodu
- Bezpieczeństwo języka, związane z eliminacją błędów w czasie kompilacji oraz wykrywaniem zagrożeń w trakcie działania programu
- Długowieczność języka oraz narzędzi kompilacyjnych, co wpływa na stabilność i przewidywalność rozwoju oprogramowania
- Przenośność między różnymi platformami i architekturami sprzętowymi.

Wszystkie te kryteria mają istotny wpływ na całkowity koszt i nakład pracy związany z tworzeniem oraz utrzymaniem oprogramowania, a także na jego jakość i przydatność w długim okresie użytkowania.

Doświadczenie wskazuje również, że te same kryteria można stosować do oceny innych komponentów wspomagających proces tworzenia oprogramowania, takich jak biblioteki klas obiektowych czy projekty typów abstrakcyjnych. Wykorzystanie odpowiednich, zewnętrznych komponentów oraz dobrze zaprojektowanych rozwiązań przyczynia się do poprawy czytelności, utrzymywalności i ogólnej jakości kodu, jednocześnie przyspieszając proces jego tworzenia.

PPL3: Jak wygląda porównanie dostępności i dojrzałości bibliotek do programowania współbieżnego i równoległego w obu językach?

Porównując dostępność i dojrzałość bibliotek do programowania współbieżnego i równoległego w językach Rust i C++, zauważyć można istotne różnice wynikające zarówno z historii rozwoju tych języków, jak i ich podejścia do bezpieczeństwa, abstrakcji oraz zarządzania zasobami.

W przypadku języka C++, biblioteki takie jak OpenMP, Intel TBB czy pthreads cechują się dużą dojrzałością oraz szerokim zastosowaniem w przemyśle, zwłaszcza w kontekście obliczeń naukowych, symulacji fizycznych i systemów o wysokiej wydajności. Są one dobrze udokumentowane, posiadają wsparcie komercyjne (np. TBB) oraz charakteryzują się dużą kompatybilnością z istniejącą infrastrukturą sprzętową i programową. Niemniej jednak, wymagają od programisty głębokiej wiedzy w zakresie zarządzania pamięcią oraz synchronizacji, co przekłada się na wyższy próg wejścia i podatność na błędy (np. wyścigi danych czy zakleszczenia).

Z kolei Rust, jako język nowszy, oferuje bardziej nowoczesny zestaw narzędzi do programowania równoległego, w tym biblioteki takie jak Tokio, Rayon czy Crossbeam. Mimo mniejszej liczby lat rozwoju, biblioteki te szybko dojrzewają i zdobywają popularność dzięki silnym gwarancjom bezpieczeństwa pamięci na poziomie kompilatora oraz ergonomicznemu API. Rust promuje bezpieczne podejście do współbieżności poprzez system własności (ang. *ownership*) i brak domyślnego współdzielenia zasobów, co w praktyce eliminuje całą klasę błędów typowych

dla C++. Dzięki temu biblioteki w Rust, choć mniej rozpowszechnione w starszych zastosowaniach, zyskują przewagę w projektach tworzonych od podstaw, zwłaszcza w środowiskach wymagających wysokiej niezawodności.

PPL4: Czy istnieją systematyczne metodologie porównywania języków programowania w kontekście problemu pracy, które można zastosować do analizy Rust i C++?

W literaturze przedmiotu można znaleźć prace proponujące sformalizowane, systematyczne metodologie służące do porównywania języków programowania w kontekście ich wsparcia dla współbieżności i równoległości. Jednakzę zdecydowana większość badań skupia się na aspektach wydajnościowych, bezpieczeństwa pamięci lub ekspresyjności języka, dodatkowo często przyjmują one podejścia ad hoc, oparte na wybranych przypadkach użycia, bez spójnych ram metodologicznych.

Niektóre opracowania, jak np. porównania publikowane w ramach blogów technicznych czy artykułów na platformach takich jak Medium, wykorzystują podejścia oparte na konteneryzacji programów testowych i ich uruchamianiu na jednorodnej infrastrukturze. Przykładowo, w metodzie *Rainbow* [28] badana jest przepustowość obliczeń przy wykorzystaniu kontenerów i zewnętrznego systemu, jak Redis, do monitorowania wyników. Choć takie podejścia są ciekawe, narażone są na zakłócenia wynikające z różnic w czasie inicjalizacji kontenerów, rozmiarze obrazu czy użyciu pamięci, co może zafałszowywać końcowe wyniki porównawcze.

Znacznie bardziej sformalizowanym podejściem jest wykorzystanie ustandaryzowanych pakietów benchmarkowych, takich jak NAS Parallel Benchmarks (NPB), które zostały opracowane przez NASA w celu obiektywnej oceny wydajności systemów wieloprocesorowych. Przykładem zastosowania tej metodologii jest praca „NPB-Rust" [45], w której autorzy przeprowadzili systematyczne porównanie implementacji benchmarków NAS w języku Rust (z użyciem biblioteki Rayon) oraz C++ (z użyciem OpenMP). Ocenie poddano nie tylko wydajność obliczeniową, ale również skalowalność, zużycie pamięci oraz nakład implementacyjny, mierząc m.in. liczbę linii kodu i szacowany koszt harmonogramowania według modelu COCOMO - (ang. *Constructive Cost Model*) - to model szacowania kosztów związanych z rozwojem oprogramowania.

Takie podejścia, oparte na zweryfikowanych zestawach testowych, umożliwiają bardziej wiarygodne i powtarzalne porównania pomiędzy językami programowania w kontekście współbieżności i równoległości. W związku z tym, zastosowanie frameworków benchmarkowych takich jak NPB powinno być traktowane jako wzorzec dla przyszłych badań w tej dziedzinie.

PPL5: Jakie aspekty programowania współbieżnego i równoległego w Rust i C++ nie zostały dostatecznie zbadane w literaturze?

Dotychczasowe badania naukowe dotyczące programowania współbieżnego w językach Rust i C++ koncentrowały się przede wszystkim na ogólnych aspektach, takich jak bezpieczeństwo pamięci, kontrola dostępu do danych, ergonomia kodu oraz wydajność kompilacji i wykonania. W szczególności analizowano podejście języków do eliminacji błędów wykonawczych (np. data races), mechanizmy typowania oraz zarządzanie zasobami systemowymi. Przykładem mogą być prace [34], które zestawiają Rust i C++ w kontekście praktycznych implementacji aplikacji wielowątkowych, czy badania [45], porównujące implementacje benchmarków równoległych.

Niemniej jednak, najnowsze publikacje wskazują na wypełnienie wielu wcześniej istniejących luk, zwłaszcza w zakresie porównania konkretnych konstrukcji językowych (np. async/await, kanały, skeletons) oraz empirycznej weryfikacji wydajności bibliotek wspierających równoległość (np. PPL, Rayon, OpenMP).

Pomimo tego, w literaturze wciąż można zidentyfikować kilka istotnych obszarów badawczych, które pozostają niewystarczająco zbadane:

- **Wpływ architektury sprzętowej na zachowanie systemów współbieżnych:**
  Większość dotychczasowych badań prowadzono na klasycznych architekturach x86_64. Brakuje natomiast analiz zachowania aplikacji wielowątkowych w Rust i C++ na alternatywnych platformach, takich jak ARM, RISC-V czy systemy heterogeniczne (np. SoC zawierające zarówno CPU, jak i akceleratory). Tego typu analizy są szczególnie istotne w kontekście rozwoju systemów wbudowanych oraz IoT, gdzie Rust zyskuje coraz większą popularność.
- **Koszt abstrakcji oraz narzut związany z modelem bezpieczeństwa Rust:**
  Choć bezpieczeństwo współbieżności w Rust jest jego kluczowym atutem, literatura rzadko podejmuje próbę precyzyjnego oszacowania narzutu wydajnościowego wynikającego z jego rygorystycznego modelu własności i borrow checkera. Istnieje potrzeba eksperymentalnych badań porównawczych, które wykazałyby, w jakim stopniu koszt ten wpływa na skalowalność aplikacji w środowiskach o dużym współczynniku równoległości.
- **Porównanie ergonomii rozwiązań współbieżnych na poziomie idiomatycznym i bibliotekowym:**
  Chociaż wiele prac omawia techniczne możliwości języków (np. std::thread, tokio, OpenMP, std::thread w C++), nadal brakuje badań jakościowych dotyczących ergonomii kodu, łatwości debugowania oraz odporności na błędy logiczne podczas implementacji systemów wielowątkowych. Takie analizy mogłyby obejmować porównania idiomatycznych podejść (np. actor model, CSP, fork-join), oferowanych przez biblioteki Rust i C++.
- **Zachowanie systemów współbieżnych w warunkach zmiennego obciążenia i konkurencyjnego dostępu:**
  Istnieje luka w badaniach dotyczących stabilności i odporności aplikacji na skoki obciążenia lub dynamiczną alokację wątków. Potrzebne są badania stresowe i profilowanie systemów w warunkach rzeczywistej konkurencji (np. serwery HTTP, silniki obliczeniowe), co pozwoliłoby ocenić adaptacyjność strategii planowania i zarządzania wątkami w Rust i C++.
- **Weryfikacja formalna i modelowanie błędów współbieżności:**
  Pomimo iż język Rust oferuje silne gwarancje bezpieczeństwa na etapie kompilacji - statyczna gwarancja bezpieczeństwa, zagadnienia związane z formalną weryfikacją własności współbieżnych, takich jak żywotność (ang. *liveness*), brak zagłodzenia (ang. *starvation-freedom*) czy deterministyczność wykonania, pozostają w literaturze stosunkowo słabo zbadane. Potencjalnie wartościowym kierunkiem dalszych analiz byłoby porównanie dostępnych narzędzi formalnych, takich jak weryfikatory modeli, w kontekście języka Rust oraz innych języków programowania współbieżnego. Tego rodzaju zestawienie mogłoby przyczynić się do lepszego zrozumienia możliwości i ograniczeń istniejących podejść do formalnej weryfikacji w środowiskach wielowątkowych.

PPL6: Jaki jest stan wiedzy na temat wykorzystania programowania współbieżnego w ramach GPU w językach Rust i C++?

W literaturze przedmiotowej oraz w praktyce programistycznej, zagadnienie wykorzystania programowania współbieżnego na GPU ewoluuje w różnym tempie w zależności od języka programowania. W przypadku języka Rust obserwujemy dynamiczny rozwój narzędzi, które umożliwiają wykorzystanie mocy obliczeniowej kart graficznych przy jednoczesnym zachowaniu wysokich gwarancji bezpieczeństwa pamięci i wątków.

Projekty takie jak rust-gpu [9] dążą do umożliwienia pisania shaderów w języku Rust, oferując podejście, które integruje możliwości GPU z bezpiecznymi konstrukcjami języka. Dokumentacja dostępna na stronie github [9] wskazuje na intensywne prace nad przeniesieniem wielu korzyści wynikających z systemu własności i typowania Rust na środowisko GPU, co może przyczynić się do zmniejszenia ryzyka błędów współbieżności, które są szczególnie krytyczne w obliczeniach równoległych.

Równolegle, biblioteka Vulkano (dostępna m.in. poprzez dokumentację [12]) stanowi wysokopoziomowy, bezpieczny interfejs do API Vulkan, które jest standardem w programowaniu GPU. Vulkano umożliwia abstrakcję złożoności niskopoziomowych interfejsów, jednocześnie oferując możliwość pełnego wykorzystania równoległości GPU. W podobnym nurcie znajduje się projekt wgpu, który implementuje standard WebGPU, umożliwiając przenośne aplikacje graficzne i obliczeniowe, a jednocześnie integrując współczesne podejścia do zarządzania zasobami i synchronizacji.

W przeciwieństwie do podejścia Rust, w środowisku C++ dominującym rozwiązaniem jest platforma CUDA, rozwijana i wspierana przez firmę NVIDIA [4]. CUDA oferuje bardzo dojrzały, zoptymalizowany i szeroko stosowany framework, który pozwala na bezpośrednią implementację algorytmów równoległych na GPU. W odróżnieniu od narzędzi rozwijanych dla Rust, CUDA posiada ugruntowaną pozycję w środowisku przemysłowym, co przekłada się na bogatą dokumentację, szerokie wsparcie techniczne oraz liczne biblioteki wspomagające rozwój aplikacji wykorzystujących GPU.

Podsumowując, stan wiedzy dotyczący programowania współbieżnego na GPU w języku Rust znajduje się na etapie intensywnego rozwoju i eksperymentacji, gdzie projekty takie jak rust-gpu, Vulkano oraz wgpu [13] pokazują potencjał tego podejścia, zwłaszcza w kontekście bezpieczeństwa i nowoczesnych abstrakcji. Z kolei w C++ platforma CUDA, dzięki swojej dojrzałości oraz szerokiemu wsparciu ze strony przemysłu, pozostaje głównym narzędziem wykorzystywanym do implementacji wysokowydajnych obliczeń równoległych na GPU.

3.2.1. Kierunki dalszych badań

Na podstawie przeglądu literatury oraz odpowiedzi na pytania do przeglądu literatury można wskazać na kilka kierunków dalszych badań w dziedzinie programowania współbieżnego i równoległego w językach Rust i C++:

- **Rozszerzone porównania systematyczne mechanizmów współbieżnych i równoległych** — z uwzględnieniem nie tylko wydajności, ale także ergonomii, bezpieczeństwa pamięci, ekspresyjności kodu oraz kosztu implementacyjnego.
- **Analiza porównawcza konkretnych mechanizmów i bibliotek** — takich jak `std::thread` w C++ i Rust, `OpenMP` vs `Rayon`, czy też mechanizmy asynchroniczne (`async/await` w Rust kontra `futures`, `std::coroutine` w C++20 i nowszych).
- **Analiza wpływu zastosowania unsafe code w Rust** — w kontekście uzyskiwanej wydajności i kompromisów względem bezpieczeństwa pamięci oraz czytelności kodu.
- **Zastosowania programowania równoległego na GPU** — porównanie możliwości Rust (np. poprzez biblioteki takie jak `rust-gpu` czy `wgpu`) z rozwiązaniami dostępnymi w ekosystemie C++ (np. CUDA, SYCL, OpenCL).
- **Badanie wpływu konstrukcji językowych na podatność na błędy współbieżności** — np. warunki wyścigu, zakleszczenia, użycie nieprawidłowych referencji lub wskaźników, z uwzględnieniem poziomu ochrony oferowanego przez kompilator.

Rozdział 4

Wybrane mechanizmy w języku Rust

o czym rozdział

4.1. 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 [21]:

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.

4.1.1. 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 4.1: 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 4.1 zaprezentowano element dotyczący użycia pożyczki w języku Rust. Błąd, oznaczony jako *Error* wynika z faktu, iż nie jest możliwe pożyczenie obiektu *s1* jako mutowalnego, ponieważ jest on jednocześnie pożyczany jako niemutowalny do obiektu *s2* (domyślnie)

4.1.2. 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 4.1.1), 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 [64].

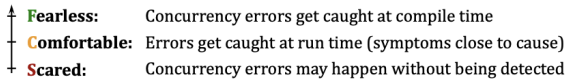


Figure 2: A spectrum of fear in parallel programming.

Rys. 4.1: Spektrum nieustraszonej współbieżności [15]

Autorzy [15] uważają, że nieustraszona współbieżność jest lepiej interpretowana jako spektrum, zilustrowane na 4.1 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)

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

Inteligentne wskaźniki 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 automatycznego systemu zarządzania pamięcią. Ponadto, wskaźniki

te są wykorzystywane zarówno w programowaniu współbieżnym, jak i równoległym, wspierając kontrolowaną oraz bezpieczną wymianę danych. Poniżej omówione zostaną trzy najczęściej używane inteligentne wskaźniki 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 4.2: Inteligentny wskaźnik Box

```
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 4.3: Inteligentny wskaźnik RC

```
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 4.4: Inteligentny wskaźnik Arc

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

Jednym z kluczowych wyróżników języka Rust jest rygorystyczny system bezpieczeństwa pamięci, realizowany poprzez model własności (ang. *ownership*), pożyczania (ang. *borrowing*) oraz statyczną kontrolę mutowalności. Niemniej jednak, w niektórych przypadkach – szczególnie przy niskopoziomowych operacjach systemowych, interoperacyjności z językiem C lub zaawansowanej optymalizacji – konieczne staje się tymczasowe zawieszenie niektórych mechanizmów ochronnych. W tym celu Rust oferuje specjalny blok językowy: unsafe.
Deklaracja unsafe nie oznacza, że kod z założenia jest błędny lub niewłaściwy. Oznacza jedynie, że kompilator przestaje gwarantować bezpieczeństwo pamięciowe i odpowiedzialność za poprawność działania spoczywa w pełni na programiście. Z poziomu unsafe można wykonać następujące operacje [18] :

- dereferencję wskaźników surowych (ang. *raw pointers*),
- wywołanie funkcji lub interfejsów oznaczonych jako unsafe (np. z FFI - (ang. *Foreign Function Interface*), czyli interfejs do wywoływania funkcji z innych języków, np. C),
- dostęp do zmiennych statycznych o niesynchronizowanym dostępie,
- implementację niektórych cech (ang. *traits*) systemowych w sposób potencjalnie niebezpieczny,
- bezpośrednią manipulację pamięcią (alokacja, kopiowanie, itd.).

Poniższy przykład demonstruje dereferencję wskaźnika surowego (*const T, *mut T), która wymaga bloku unsafe:

Listing 4.5: Przykład użycia unsafe Rust

```
fn main() {
    let x: i32 = 42;
    let ptr: *const i32 = &x;

    unsafe {
        println!("Wartość␣pod␣wskaźnikiem␣ptr:␣{}", *ptr);
    }
}
```

W powyższym kodzie listing 4.5 wskaźnik ptr jest wskaźnikiem surowym (ang. *raw pointer*), który nie posiada gwarancji poprawności, jakie zapewniają referencje bezpośrednie (&T, &mut

T). Aby móc odczytać wartość spod tego wskaźnika, konieczne jest oznaczenie operacji jako unsafe.

Warto jednak podkreślić, że Rust promuje zasadę minimalnego zaufania (ang. *principle of minimal trust*), dlatego zaleca się ograniczanie użycia unsafe do niezbędnych sekcji oraz hermetyzowanie ich w bezpiecznych abstrakcjach (np. typach własnych, modułach lub API) [18].

4.2. 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.

4.2.1. 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 4.1.1.

4.2.2. Biblioteki

Tokio

Tokio to biblioteka służąca do obsługi zadań asynchronicznych (ang. asynchronous tasks) oraz wejścia-wyjścia bez blokad (ang. *non-blocking I/O*). Wykorzystuje pętlę zdarzeń (ang. *event loop*), dzięki której możliwe jest współbieżne przetwarzanie wielu zadań bez konieczności tworzenia oddzielnych wątków systemowych dla każdego z nich. Tokio wspiera zarówno jednowątkowy jak i wielowątkowy tryb działania, co pozwala na rozwój aplikacji zgodnie z potrzebami.

Listing 4.6: Przykład użycia Tokio

```
use tokio::net::TcpListener;
use tokio::io::{AsyncReadExt, AsyncWriteExt};

#[tokio::main]
async fn main() {
    let listener = TcpListener::bind("127.0.0.1:8080").await.unwrap();
    loop {
        let (mut socket, _) = listener.accept().await.unwrap();
        tokio::spawn(async move {
            let mut buf = [0; 1024];
            let n = socket.read(&mut buf).await.unwrap();
            socket.write_all(&buf[0..n]).await.unwrap();
        });
```

```
    }
}
```

W powyższym przykładzie listing 4.6 tokio::spawn inicjuje współbieżne zadanie, które przetwarza połączenie TCP bez blokowania głównej pętli programu.

Crossbeam::channel – komunikacja między wątkami

Crossbeam to zestaw narzędzi wspierających programowanie współbieżne. Jednym z kluczowych komponentów tej biblioteki są kanały komunikacyjne (ang. *channels*), implementowane w module crossbeam::channel. Stanowią one bezpieczny i efektywny sposób przesyłania wiadomości pomiędzy wątkami, umożliwiając implementację wzorca producent–konsument.

Listing 4.7: Przykład użycia kanałów Crossbeam

```
use crossbeam::channel;
use std::thread;

fn main() {
    let (sender, receiver) = channel::unbounded();

    let producer = thread::spawn(move || {
        for i in 0..5 {
            sender.send(i).unwrap();
        }
    });

    let consumer = thread::spawn(move || {
        while let Ok(msg) = receiver.recv() {
            println!("Odebrano:␣{}", msg);
        }
    });

    producer.join().unwrap();
    consumer.join().unwrap();
}
```

W tym przypadku listing 4.7 channel::unbounded() tworzy kanał bez ograniczenia pojemności (ang. *unbounded channel*), który może służyć do swobodnej komunikacji między wątkami.

Actix

Actix to framework do tworzenia współbieżnych aplikacji opartych na modelu aktorowym. W tym podejściu jednostką obliczeniową jest aktor, który posiada własny stan i komunikuje się z innymi aktorami wyłącznie za pomocą wiadomości. Takie podejście eliminuje potrzebę stosowania blokad.

Listing 4.8: Przykład użycia Actix

```
use actix::prelude::*;

struct Ping;
```

```
impl Message for Ping {
    type Result = &'static str;
}

struct MyActor;

impl Actor for MyActor {
    type Context = Context<Self>;
}

impl Handler<Ping> for MyActor {
    type Result = &'static str;

    fn handle(&mut self, _: Ping, _: &mut Context<Self>) -> Self::
   ↪ Result {
        "pong"
    }
}

#[actix::main]
async fn main() {
    let addr = MyActor.start();
    let res = addr.send(Ping).await.unwrap();
    println!("{}", res);
}
```

W zaprezentowanym przykładzie aktor MyActor odbiera wiadomości typu Ping i odpowiada komunikatem pong. Każdy aktor działa w swoim własnym kontekście, co zapewnia izolację stanu i zwiększa bezpieczeństwo współbieżne bez użycia mutexów.

4.2.3. 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 4.9: 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 4.9:

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 4.10: Przykład z wieloma wątkami

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

```rust
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 4.10 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>>*** 4.1.3 do umożliwienia współdzielenia odbiorcy (Receiver) między wątkami. Mutex zapewnia synchronizację dostępu do kanału, dzięki czemu tylko jeden wątek w danej chwili 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 4.11: 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 4.11 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!
```

4.2.4. 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. Podstawowe elementy asynchroniczności to:

- async / await – konstrukcje językowe umożliwiające deklaratywne definiowanie funkcji asynchronicznych i ich późniejsze wywoływanie bez blokowania wątku.
- Future – futury, typ reprezentujący wartość, która będzie dostępna w przyszłości. Futures są wykonywane leniwie i wymagają uruchomienia w ramach runtime'u.
- Tokio i async-std – najpopularniejsze asynchroniczne runtimes, implementujące własne planisty i pętle zdarzeń (event loops).
- Asynchroniczne kanały (tokio::sync::mpsc, broadcast, oneshot) – umożliwiają komunikację bez blokowania wątków.
- Asynchroniczne semafory (tokio::sync::Semaphore) – pozwalają ograniczyć liczbę jednocześnie wykonywanych zadań asynchronicznych bez blokowania wątku. Są często stosowane do kontroli równoczesnych operacji na zasobach zewnętrznych, takich jak dostęp do bazy danych lub API.

W standardowej bibliotece Rust nie ma semafora, ale tokio udostępnia asynchroniczny semafor (tokio::sync::Semaphore), który można również użyć w środowisku wielowątkowym z tokio::main.

Listing 4.12: Przykład użycia semafora

```
use tokio::sync::Semaphore;
use std::sync::Arc;
use tokio::task;
use std::time::Duration;

#[tokio::main]
async fn main() {
    let semaphore = Arc::new(Semaphore::new(2)); // maks. 2 ró
    ↪ wnoczesne zasoby
    let mut handles = vec![];

    for i in 0..5 {
        let sem_clone = Arc::clone(&semaphore);
        let handle = task::spawn(async move {
            let permit = sem_clone.acquire().await.unwrap();
            println!("Zadanie␣{}:␣rozpoczęło␣pracę", i);

            tokio::time::sleep(Duration::from_secs(1)).await;

            println!("Zadanie␣{}:␣kończy␣pracę", i);
            drop(permit); // zwolnienie zasobu
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.await.unwrap();
    }
}
```

W tym przykładzie listing 4.12 semafor o pojemności 2 (Semaphore::new(2)) umożliwia maksymalnie dwóm zadaniom równoczesny dostęp do sekcji krytycznej. Pozostałe zadania oczekują w kolejce, aż jedno zezwolenie zostanie zwolnione. tokio::sync::Semaphore jest semaforem asynchronicznym (nie blokuje fizycznego wątku), dlatego nadaje się zarówno do środowisk asynchronicznych, jak i do ograniczania zasobów w systemach hybrydowych (np. współbieżno-równoległych).

4.3. Programowanie równoległe

Rust oferuje nowoczesne podejście do programowania równoległego, które pozwala na bezpieczne i wydajne wykorzystanie wielu rdzeni procesora. Dzięki statycznemu systemowi typów, modelowi własności oraz bogatemu ekosystemowi bibliotek, programowanie równoległe w Rust jest zarówno ergonomiczne, jak i odporne na typowe błędy związane z współdzieleniem pamięci.

4.3.1. Biblioteki

Jednym z kluczowych komponentów wspierających programowanie równoległe w Rust jest biblioteka Rayon. Została ona zaprojektowana jako ergonomiczne narzędzie do równoległego

przetwarzania kolekcji oraz rekurencyjnych algorytmów, takich jak mapowanie, filtrowanie czy redukcja. W przeciwieństwie do tradycyjnych podejść wymagających ręcznego tworzenia i zarządzania wątkami, Rayon oferuje wysokopoziomowe abstrakcje, które ukrywają złożoność alokacji wątków oraz synchronizacji, przy zachowaniu bezpieczeństwa typów i braku wyścigów danych.
Przykładowe wykorzystanie biblioteki Rayon może wyglądać następująco:

Listing 4.13: Przykład użycia par_iter

```
use rayon::prelude::*;

fn main() {
    let data = vec![1, 2, 3, 4, 5, 6, 7, 8];

    let result: i32 = data
        .par_iter()                    // <- równoległa wersja iteratora
        .map(|x| x * 2)                // <- równolegle podwajamy każdą
    ↪ wartość
        .reduce(|| 0, |a, b| a + b); // <- redukcja do sumy, start = 0

    println!("Wynik:␣{}", result); // Oczekiwany wynik: 72
}
```

Kod przedstawiony w listingu 4.13 realizuje prostą operację podwajania wartości elementów wektora oraz ich sumowania, jednak kluczową cechą jest to, że wszystkie operacje wykonywane są równolegle. Funkcja `par_iter()` zamienia standardowy iterator sekwencyjny na jego równoległy odpowiednik, co oznacza, że kolejne elementy będą przetwarzane na wielu rdzeniach procesora w sposób automatyczny i zoptymalizowany przez bibliotekę.

Następnie funkcja `map` pozwala na równoległe zastosowanie tej samej operacji - w tym przypadku mnożenia przez 2 - do każdego elementu kolekcji. Nie wymaga to żadnej ręcznej synchronizacji ani zarządzania współbieżnością — biblioteka zajmuje się podziałem pracy i wykonaniem w sposób optymalny względem dostępnych zasobów sprzętowych.

Ostatni etap przetwarzania to `reduce`, który agreguje wyniki cząstkowe w jeden wynik końcowy. Funkcja ta działa również równolegle: najpierw sumowane są wartości lokalnie (w ramach każdego wątku roboczego), a dopiero potem następuje łączenie tych częściowych sum w jedną finalną wartość. Neutralny element funkcji redukującej to 0, co jest standardowym przypadkiem przy sumowaniu liczb całkowitych.

Cały proces ilustruje typowy model MapReduce, w którym dane są:

- dzielone na fragmenty (ang. *split*),
- transformowane (ang. *map*),
- a następnie agregowane (ang. *reduce*).

4.4. Mechanizmy wspólne dla współbieżności i równoległości

4.4.1. Wątki (std::thread)

Jednym z podstawowych narzędzi oferowanych przez standardową bibliotekę Rust jest moduł std::thread, który umożliwia tworzenie niezależnych wątków wykonawczych. Pomimo że zapewnia on niski poziom abstrakcji i bezpośrednią kontrolę nad wątkami, jego użycie wymaga większej ostrożności w kontekście synchronizacji i zarządzania danymi współdzielonymi. Przykładowa konstrukcja:

Listing 4.14: Przykład tworzenia wątku

```
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
        // kod wykonywany równolegle
    });
    handle.join().unwrap();
}
```

W przedstawionym przykładzie listing 4.14 wykorzystano funkcję thread::spawn, która tworzy nowy wątek wykonawczy, umożliwiając równoległe przetwarzanie danych lub zadań. Ciało funkcji anonimowej przekazanej do spawn zawiera kod, który zostanie wykonany w kontekście nowo utworzonego wątku. Zmienna handle przechowuje uchwyt do tego wątku, umożliwiając synchronizację z jego wykonaniem. Wywołanie handle.join().unwrap() służy do zablokowania głównego wątku programu do momentu zakończenia pracy wątku potomnego. Metoda join zwraca wynik zakończenia wątku.

Tworzenie dużej liczby wątków może być kosztowne zarówno pod względem zasobów systemowych, jak i czasu inicjalizacji. W związku z tym, Rust oferuje mechanizmy pul wątków (ang. *thread pools*), które umożliwiają wielokrotne wykorzystywanie wcześniej zainicjalizowanych wątków do realizacji wielu zadań.

Popularnym rozwiązaniem wspierającym pule wątków jest biblioteka rayon, która automatyzuje proces zarządzania wątkami w kontekście równoległego przetwarzania danych. Jednakże, również inne biblioteki, takie jak tokio (dla asynchroniczności) czy async-std, implementują własne menedżery wątków, umożliwiające bardziej zaawansowane zarządzanie zadaniami.

W celu maksymalizacji wykorzystania zasobów obliczeniowych, wiele implementacji pul wątków w Rust stosuje strategię kradzieży zadań (ang. *work stealing*). Mechanizm ten polega na dynamicznym równoważeniu obciążenia przez umożliwienie wątkom pobierania zadań z kolejek innych wątków, gdy ich własne kolejki są puste. Zwiększa to ogólną wydajność i skraca czas przetwarzania zadań.

Strategia ta znajduje zastosowanie m.in. w implementacji puli wątków biblioteki rayon, co czyni ją wysoce wydajną w przypadku zadań o nieregularnym czasie wykonania lub zróżnicowanym poziomie złożoności.

4.4.2. Synchronizacja dostępu (Mutex, 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.

Listing 4.15: Przykład użycia Mutex

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

fn main() {
    let counter = Arc::new(Mutex::new(0));

    let mut handles = vec![];
```

```
    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();
            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Wartość końcowa: {}", *counter.lock().unwrap());
}
```

Mutex<T> (ang. *mutual exclusion*), czyli wzajemne wykluczanie, to mechanizm blokady umożliwiający bezpieczny dostęp do danych przez wiele wątków. W powyższym przykładzie dane typu i32 są opakowane w Mutex, a następnie udostępniane wielu wątkom za pomocą wskaźnika liczony atomowo Arc<T>. Każdy wątek dokonuje inkrementacji zmiennej wewnątrz sekcji krytycznej, co zapobiega wyścigom danych (ang. data races). Wywołanie lock().unwrap() blokuje wątek do czasu uzyskania wyłącznego dostępu do danych. Podejście to jest typowe w programowaniu zarówno współbieżnym (dla ochrony stanu globalnego), jak i równoległym (dla synchronizacji wyników obliczeń).

Listing 4.16: Przykład użycia RwLock

```
use std::sync::{Arc, RwLock};
use std::thread;

fn main() {
    let data = Arc::new(RwLock::new(vec![1, 2, 3]));

    let reader = {
        let data = Arc::clone(&data);
        thread::spawn(move || {
            let read = data.read().unwrap();
            println!("Odczyt danych: {:?}", *read);
        })
    };

    let writer = {
        let data = Arc::clone(&data);
        thread::spawn(move || {
            let mut write = data.write().unwrap();
            write.push(4);
        })
    };

    reader.join().unwrap();
    writer.join().unwrap();
```

```
}
```

RwLock<T> (ang. *read-write lock*) czyli blokada odczytu-zapisu, umożliwia wielu wątkom jednoczesny odczyt danych przy zachowaniu wyłączności dla operacji zapisu. W przypadku często czytanych, rzadko modyfikowanych danych, takie rozwiązanie pozwala na lepszą wydajność niż klasyczny Mutex. Biblioteka standardowa Rust zapewnia gwarancje bezpieczeństwa pamięci, eliminując ryzyko naruszeń spójności danych nawet w środowiskach wielordzeniowych.

4.4.3. Wartości atomowe (Atomic*)

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

Listing 4.17: Przykład użycia Atomic

```
use std::sync::atomic::{AtomicUsize, Ordering};
use std::thread;

fn main() {
    let counter = AtomicUsize::new(0);

    let handles: Vec<_> = (0..10)
        .map(|_| {
            let counter_ref = &counter;
            thread::spawn(move || {
                for _ in 0..1000 {
                    counter_ref.fetch_add(1, Ordering::Relaxed);
                }
            })
        })
        .collect();

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Wynik:␣{}", counter.load(Ordering::Relaxed));
}
```

Typy Atomic* w Rust, takie jak AtomicUsize, AtomicBool, czy AtomicPtr, umożliwiają bezpieczne operacje na danych współdzielonych bez stosowania mechanizmów blokujących (ang. *lock-free synchronization*). W przedstawionym przykładzie listing 4.17, funkcja fetch_add wykonuje inkrementację wartości w sposób atomowy, zapewniając spójność danych nawet przy równoczesnym dostępie z wielu wątków. Choć Ordering::Relaxed zapewnia najmniejszy narzut, istnieją też silniejsze modele spójności pamięci (np. SeqCst, Acquire/Release), które mogą być konieczne przy bardziej złożonych zależnościach.

4.4.4. Bariery

Rust oferuje również podstawowe prymitywy synchronizacyjne, takie jak bariery, które umożliwiają synchronizację wątków w bardziej złożonych scenariuszach. Bariery pozwalają na zsyn-

chronizowanie grupy wątków, które muszą osiągnąć określony punkt przed kontynuowaniem pracy, natomiast semafory kontrolują dostęp do ograniczonej liczby zasobów.

Listing 4.18: Przykład użycia bariery

```
use std::sync::{Arc, Barrier};
use std::thread;

fn main() {
    let barrier = Arc::new(Barrier::new(3));
    let mut handles = vec![];

    for i in 0..3 {
        let c = Arc::clone(&barrier);
        let handle = thread::spawn(move || {
            println!("Wątek␣{}:␣przed␣barierą", i);
            c.wait(); // punkt synchronizacji
            println!("Wątek␣{}:␣po␣barierze", i);
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }
}
```

Barrier (bariera synchronizacyjna) to mechanizm służący do synchronizacji wielu wątków w ustalonym punkcie programu. Każdy wątek, który osiąga barierę, zostaje zablokowany do momentu, aż dojdą do niej wszystkie pozostałe wątki zadeklarowane przy jej utworzeniu. Przykład ten przedstawia trójwątkową synchronizację – żaden z wątków nie przejdzie do fazy „po barierze”, dopóki wszystkie nie osiągną funkcji wait(). Bariera jest przydatna w systemach opartych na etapowym przetwarzaniu, np. w modelu SIMD czy w systemach obliczeń wielofazowych.

Rozdział 5

Wybrane mechanizmy w języku C++

5.1. Programowanie współbieżne

Współbieżność w języku C++ wspierana jest od standardu C++11, który wprowadził szereg struktur i mechanizmów umożliwiających tworzenie i synchronizację wątków. W kolejnych wersjach (C++14, C++17, C++20 i C++23) język został wzbogacony o kolejne narzędzia, zwiększające bezpieczeństwo, ekspresyjność i ergonomię programowania współbieżnego.

5.1.1. Biblioteki i przestrzeń standardowa

std::thread oraz std::jthread

Standardowa biblioteka języka C++ zawiera podstawowe komponenty do obsługi wątków w module <thread>. Wraz z C++20 wprowadzono std::jthread, będący bezpieczniejszą alternatywą dla std::thread, ponieważ automatycznie dołącza wątek w destruktorze obiektu. Dzięki temu możliwe jest uniknięcie błędów takich jak niezakończony wątek (ang. *orphaned thread*) lub przedwczesne zakończenie programu.

Listing 5.1: Przykład użycia std::jthread

```
#include <iostream>
#include <thread>

void worker() {
    std::cout << "Thread␣is␣running." << std::endl;
}

int main() {
    std::jthread t(worker); // wątek zarządzany automatycznie
    // brak konieczności wywoływania join() lub detach()
}
```

W przykładzie listing 5.1 utworzono nowy wątek wątek z użyciem std::jthread, który uruchamia funkcję worker. Dzięki automatycznemu zarządzaniu zasobami przez std::jthread, nie ma potrzeby ręcznego wywoływania join(), co zmniejsza ryzyko błędów synchronizacji.

5.1.2. Komunikacja między wątkami

C++ nie posiada wbudowanych kanałów (ang. *channels*) występujących w języku Rust, lecz umożliwia komunikację poprzez konstrukcje takie jak: kolejki, zmienne warunkowe (std::condition_variable) oraz typy atomowe (std::atomic). Jednym z najczęstszych wzorców komunikacyjnych jest użycie kolejki chronionej mutexem oraz sygnalizowanej zmienną warunkową.

Listing 5.2: Przykład komunikacji między wątkami

```
#include <iostream>
#include <thread>
#include <queue>
#include <mutex>
#include <condition_variable>

std::queue<int> buffer;
std::mutex mtx;
std::condition_variable cv;

void producer() {
    std::unique_lock<std::mutex> lock(mtx);
    buffer.push(100); // produkcja danych
    cv.notify_one();  // powiadom konsumenta
}

void consumer() {
    std::unique_lock<std::mutex> lock(mtx);
    cv.wait(lock, [] { return !buffer.empty(); }); // czekaj na dane
    std::cout << "Consumed:␣" << buffer.front() << std::endl;
    buffer.pop(); // usuń dane z kolejki
}
```

Powyższy przykład listing 5.2 implementuje prosty scenariusz producent–konsument. Producent wstawia dane do kolejki i powiadamia wątek oczekujący. Konsument blokuje się, dopóki kolejka nie będzie zawierać danych. Zmienna warunkowa eliminuje konieczność aktywnego sprawdzania warunku (busy waiting), co poprawia efektywność systemu.

5.1.3. Synchronizacja

Synchronizacja w języku C++ opiera się głównie na mutexach (std::mutex) oraz ich odmianach. Od C++17 dostępny jest std::scoped_lock, pozwalający na bezpieczne blokowanie wielu mutexów jednocześnie, a od C++20 wprowadzono bardziej zaawansowane konstrukty takie jak std::latch i std::barrier, które umożliwiają synchronizację wielu wątków na określonym etapie wykonania.

Listing 5.3: Przykład użycia std::scoped_lock

```
#include <iostream>
#include <thread>
#include <mutex>

int counter = 0;
std::mutex mtx;

void increment() {
```

```
    std::lock_guard<std::mutex> lock(mtx); // automatyczna blokada
        ↪ mutexu
    ++counter;
}

int main() {
    std::thread t1(increment);
    std::thread t2(increment);
    t1.join();
    t2.join();
    std::cout << "Counter: " << counter << std::endl;
}
```

W tym przykładzie - listing 5.3 dwa wątki próbują jednocześnie zwiększyć wartość zmiennej counter. Aby zapobiec wyścigowi danych, dostęp do zasobu jest chroniony przez std::mutex. Użycie std::lock_guard zapewnia, że blokada zostanie zwolniona automatycznie po wyjściu z zakresu funkcji.

std::latch oraz std::barrier (C++20)

Mechanizmy std::latch oraz std::barrier wprowadzone w standardzie C++20 służą do synchronizacji wielu wątków w określonym punkcie programu:

- std::latch – jednorazowy licznik synchronizacyjny, który pozwala wątkom oczekującym na rozpoczęcie działania, aż inne wątki zakończą przygotowanie.
- std::barrier – wielokrotnego użytku, synchronizuje grupę wątków po osiągnięciu „cyklu bariery”.

Listing 5.4: Przykład użycia std::latch oraz std::barrier

```
#include <iostream>
#include <thread>
#include <latch>
#include <barrier>

constexpr int num_threads = 3;
std::latch start_latch(num_threads);
std::barrier sync_barrier(num_threads);

void worker(int id) {
    std::cout << "Thread " << id << " is initializing.\n";
    start_latch.arrive_and_wait(); // czekaj aż wszystkie wątki się
        ↪ przygotują
    for (int i = 0; i < 2; ++i) {
        std::cout << "Thread " << id << " is processing iteration "
            ↪ << i << ".\n";

        // Synchronizacja między iteracjami
        sync_barrier.arrive_and_wait();

        std::cout << "Thread " << id << " passed the barrier in
            ↪ iteration " << i << ".\n";
    }
}
int main() {
```

```
    std::thread threads[num_threads];
    for (int i = 0; i < num_threads; ++i)
    threads[i] = std::thread(worker, i + 1);
    for (auto& t : threads)
        t.join();
}
```

W listingu 5.4 każdy wątek najpierw dochodzi do punktu synchronizacji std::latch, oczekując aż wszystkie inne wątki również zakończą fazę inicjalizacji. Następnie, w dwóch kolejnych iteracjach przetwarzania danych, zastosowany zostaje std::barrier, który gwarantuje, że wszystkie wątki ukończą daną iterację przed przejściem do kolejnej. Takie podejście zwiększa spójność przetwarzania i eliminuje potencjalne niespójności wynikające z wyścigów czasowych między wątkami.

5.1.4. Asynchroniczność

Programowanie asynchroniczne w C++ możliwe jest dzięki konstrukcjom takim jak std::async, std::future i std::promise. std::async uruchamia funkcję w tle i umożliwia jej obserwację za pomocą obiektu future. Podejście to ułatwia uruchamianie zadań bez konieczności jawnego zarządzania wątkiem.

Listing 5.5: Przykład użycia std::async

```
#include <iostream>
#include <future>

int compute() {
    return 2 * 21;
}

int main() {
    std::future<int> result = std::async(std::launch::async,
       ↪ compute);
    std::cout << "Result: " << result.get() << std::endl;
}
```

W powyższym listingu 5.5 funkcja compute() zostaje uruchomiona asynchronicznie. std::future pozwala na uzyskanie wyniku, gdy ten będzie dostępny. W ten sposób możemy kontynuować inne działania, a wynik odebrać w późniejszym czasie — co jest przydatne w aplikacjach wymagających wysokiej responsywności.

std::promise

Obiekt std::promise (obietnica) w języku C++ umożliwia przekazywanie wartości z jednego wątku do drugiego. Stanowi uzupełnienie mechanizmu std::future, ponieważ pozwala manualnie ustawić wartość, którą future później odbierze. Dzięki temu rozdzielona zostaje produkcja i konsumpcja danych między wątkami, umożliwiając bardziej elastyczne projektowanie asynchronicznych przepływów sterowania.

Listing 5.6: Przykład użycia std::promise

```
#include <iostream>
#include <thread>
#include <future>
```

```
// Funkcja symulująca kosztowne obliczenie
int compute(int x) {
    return x * 2;
}

int main() {
    std::promise<int> promise; // utworzenie obiektu obietnicy
    std::future<int> result = promise.get_future(); // pobranie powią
       ↪ zanego future

    std::thread producer([&promise]() {
        int value = 21;
        int result = compute(value);
        promise.set_value(result); // ustawienie wartości, która
           ↪ zostanie odebrana przez future
    });

    std::cout << "Result: " << result.get() << std::endl; // odbiór
       ↪ wartości, blokuje do czasu jej ustawienia
    producer.join();

    return 0;
}
```

W listingu 5.6 wątek główny tworzy promise i pobiera powiązany z nim future. Wątek producer wykonuje obliczenie i przekazuje wynik przez promise. Funkcja result.get() wstrzymuje główny wątek do czasu dostępności wyniku.

std::packaged_task

std::packaged_task to obiekt otaczający dowolną wywoływalną funkcję (np. funkcję, lambda, std::bind), który integruje się z future. Pozwala to uruchomić zadanie w wątku i obserwować jego wynik.

Listing 5.7: Przykład użycia std::packaged_task

```
#include <iostream>
#include <thread>
#include <future>

// Funkcja do opakowania w packaged_task
int compute(int x) {
    return x * 2;
}

int main() {
    std::packaged_task<int(int)> task(compute); // utworzenie
       ↪ zapakowanego zadania
    std::future<int> result = task.get_future(); // pobranie powią
       ↪ zanego future

    std::thread worker(std::move(task), 21); // uruchomienie zadania
       ↪ w wątku z parametrem 21
```

```
    std::cout << "Result: " << result.get() << std::endl; // odbiór
        ↪ wyniku
    worker.join();

    return 0;
}
```

W powyższym przykładzie listing 5.7 funkcja compute została opakowana w packaged_task, a następnie uruchomiona w osobnym wątku z argumentem 21. Wynik trafia do future, który umożliwia jego odbiór w wątku głównym. przetwarzanie zadań.

C++23 – std::task i std::execution

Standard C++23 (najnowszy w chwili tworzenia niniejszej pracy) wprowadza nowe pojęcia:

- std::task – reprezentuje zadanie, które można uruchomić z użyciem określonego planisty wykonania.
- std::execution – zestaw polityk (strategii) określających sposób wykonywania zadań, takich jak sekwencyjnie, współbieżnie, równolegle.

Mechanizmy te są częścią trwającej transformacji C++ w kierunku deklaratywnego modelu programowania współbieżnego i równoległego. Mają one zostać wprowadzone wstępnie w wersji C++26 [5] Ponieważ wsparcie dla tych mechanizmów jest na etapie wdrażania, nie będą one szczegółowo omawiane w tej pracy. Warto jednak zauważyć, że ich celem jest uproszczenie i ujednolicenie podejścia do programowania współbieżnego, podobnie jak ma to miejsce w języku Rust.

5.2. Programowanie równoległe

5.2.1. OpenMP - Open Multi-Processing

OpenMP to biblioteka umożliwiająca programowanie równoległe w modelu pamięci współdzielonej. Jest dostępna dla języków C, C++ oraz Fortran i opiera się na użyciu dyrektyw preprocesora (ang. compiler directives) pozwalających na uproszczone rozproszenie zadań pomiędzy wątki w sposób deklaratywny.

Podstawowym celem OpenMP jest ułatwienie implementacji aplikacji równoległych poprzez maksymalne ograniczenie ręcznego zarządzania wątkami i synchronizacją. W kodzie C++ użycie tej technologii wymaga aktywacji flagi -fopenmp podczas kompilacji przy użyciu kompilatora g++. Przedstawia podstawowe pojęcia:

- #pragma omp parallel — dyrektywa inicjująca blok kodu, który ma zostać wykonany równolegle przez wiele wątków,
- shared — oznacza zmienne współdzielone pomiędzy wszystkimi wątkami,
- private — każdemu wątkowi przypisywana jest prywatna kopia zmiennej,
- cache locality — pamięć podręczna (ang. cache) może znacznie poprawić wydajność przetwarzania, choć kosztem większego zużycia pamięci.

Listing 5.8: Przykład użycia OpenMP w C++

```
#include <omp.h>
#include <iostream>

int main() {
    int sum = 0;
```

```
    #pragma omp parallel for reduction(+:sum)
    for (int i = 1; i <= 100; ++i) {
        sum += i;
    }
    std::cout << "Suma: " << sum << std::endl;
    return 0;
}
```

W powyższym przykładzie w listingu 5.8 zmienna sum jest sumą liczb od 1 do 100 obliczaną równolegle. Dzięki użyciu dyrektywy #pragma omp parallel for każda iteracja pętli może być wykonana w osobnym wątku. Atrybut reduction(+:sum) zapewnia bezpieczne sumowanie wyników lokalnych wątków do jednej wartości globalnej. OpenMP automatycznie zarządza synchronizacją i agregacją wyników, dzięki czemu użytkownik nie musi implementować ręcznego zarządzania zasobami współdzielonymi .

5.2.2. Intel TBB (Threading Building Blocks)

Intel Threading Building Blocks (TBB) to nowoczesna biblioteka programistyczna dla języka C++, przeznaczona do tworzenia aplikacji równoległych w sposób wysoce elastyczny i skalowalny. W przeciwieństwie do OpenMP, TBB opiera się na programowaniu funkcyjnym i komponentowym, umożliwiając dekompozycję zadań (ang. *task-based parallelism*), a nie operacji niskiego poziomu.

Cechy charakterystyczne:

- Deklaratywny styl programowania, który umożliwia oddelegowanie decyzji o wykonaniu do systemu planowania zadań.
- Dynamiczna alokacja wątków w oparciu o dostępność zasobów.
- Wbudowana obsługa synchronizacji oraz struktur danych przystosowanych do środowisk wielowątkowych (np. concurrent_vector, concurrent_queue).

Listing 5.9: Przykład użycia Intel TBB w C++

```
#include <tbb/tbb.h>
#include <iostream>
#include <vector>

int main() {
    std::vector<int> data(1000, 1);
    int sum = 0;

    tbb::parallel_reduce(
        tbb::blocked_range<size_t>(0, data.size()),
        0,
        [&](const tbb::blocked_range<size_t>& r, int local_sum) {
            for (size_t i = r.begin(); i < r.end(); ++i)
                local_sum += data[i];
            return local_sum;
        },
        std::plus<int>()
    );

    std::cout << "Suma: " << sum << std::endl;
    return 0;
}
```

W powyższym przykładzie w listingu 5.9 funkcja tbb::parallel_reduce automatycznie dzieli zakres danych na bloki (blocked_range), które przetwarzane są równolegle przez dostępne wątki. Funkcja lambda odpowiada za lokalne przetwarzanie danych (w tym przypadku sumowanie wartości), a następnie lokalne wyniki są agregowane przy użyciu funkcji std::plus<int>. TBB samodzielnie zarządza planowaniem zadań oraz synchronizacją, co czyni go potężnym narzędziem w budowie skalowalnych aplikacji równoległych.

Rozdział 6

Założenia i metodologia porównania mechanizmów w językach Rust oraz C++

W tym rozdziale przedstawiono metryki oraz algorytmy do analizy wydajności mechanizmów programowania współbieżnego i równoległego w językach Rust oraz C++.

6.1. Programowanie współbieżne

Dla mechanizmów programowania współbieżnego autor nie znalazł obecnie zunifikowanego, powszechnie uznanego zestawu benchmarków odpowiadający randze NPB. W związku z tym, w ramach niniejszej pracy, opracowano własny zestaw mini-aplikacji testowych, zaprojektowanych w taki sposób, aby odzwierciedlały typowe scenariusze współbieżności: komunikację między wątkami, synchronizację, obsługę asynchronicznych operacji wejścia/wyjścia, sytuacje ryzyka zakleszczenia, a także przypadki intensywnego przetwarzania danych z wykorzystaniem wielu wątków.

6.1.1. Zarządzanie wątkami

Porównanie tworzenia i zarządzania wątkami obejmuje następujące parametry:

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

6.1.2. 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).

6.1.3. 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).

6.1.4. Wybrane algorytmy do analizy

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.

6.2. Programowanie równoległe

W przypadku programowania równoległego, zdecydowano się na wykorzystanie uznanego zestawu testowego NAS Parallel Benchmarks (NPB) [8]. Zestaw ten jest szeroko stosowany w środowisku naukowym do oceny wydajności systemów wysokowydajnych (HPC) i stanowi wiarygodny punkt odniesienia przy analizie efektywności obliczeniowej.

6.2.1. 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).

6.2.2. 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.

6.2.3. 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.

6.2.4. Wybrane algorytmy do analizy

Dla porównania mechanizmów równoległości wybrano następujące algorytmy z zestawu testowego NPB:

- CG - (ang. *conjugate gradient*) - gradient sprzężony
- EP - (ang. *embarrassingly parallel*) - problem trywialnie równoległy
- IS - (ang. *intiger sorting*) - sortowanie liczb całkowitych

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.

CG - Gradient sprzężony

Benchmark CG (ang. *conjugate gradient*) służy do oceny wydajności systemów wysokowydajnych (HPC) w kontekście rozwiązywania rzadkich układów równań liniowych metodą iteracyjną. Algorytm ten znajduje zastosowanie w wielu dziedzinach nauk obliczeniowych, takich jak mechanika płynów czy analiza strukturalna, gdzie układy równań wynikają z dyskretyzacji równań różniczkowych cząstkowych. W benchmarku CG generowana jest syntetyczna macierz rzadkich współczynników o dużych rozmiarach, a następnie przeprowadzana jest iteracyjna procedura wyznaczania przybliżonego rozwiązania układu równań. Test ten charakteryzuje się intensywnym wykorzystaniem operacji wektorowych i punktowych operacji na danych rozproszonych, co czyni go szczególnie użytecznym przy ocenie efektywności komunikacji między wątkami oraz przepustowości pamięci w systemach równoległych [8].

EP - Problem trywialnie równoległy

Benchmark EP (ang. *embarrassingly parallel*) został zaprojektowany w celu oceny wydajności systemów obliczeniowych w scenariuszach, w których niemal całkowicie eliminuje się konieczność komunikacji między procesami lub wątkami. Test ten polega na generowaniu dużej liczby losowych punktów i przeprowadzaniu na nich niezależnych obliczeń statystycznych, takich jak estymacja wartości π lub momentów rozkładu. Dzięki swojej naturze, EP umożliwia niemal idealną skalowalność równoległą i jest wykorzystywany przede wszystkim do pomiaru czystej mocy obliczeniowej procesorów, efektywności rozdziału zadań oraz narzutu wynikającego z zarządzania wątkami. Ze względu na minimalne wymagania względem synchronizacji i komunikacji, benchmark ten stanowi punkt odniesienia przy analizie teoretycznego maksimum wydajności danego systemu dla obciążeń równoległych [8].

IS - Sortowanie liczb całkowitych

Benchmark IS (ang. *integer sorting*) służy do oceny wydajności systemów obliczeniowych w zakresie operacji nieciągłych i trudnych do zrównoleglenia, takich jak sortowanie i przemieszczanie danych w pamięci. Test polega na wygenerowaniu losowego zestawu liczb całkowitych, a następnie ich posortowaniu przy użyciu metody sortowania kubełkowego (ang. *bucket sort*) z zastosowaniem rozproszonej synchronizacji i komunikacji między wątkami lub procesami. Benchmark IS jest szczególnie przydatny do analizy przepustowości podsystemów pamięciowych, efektywności komunikacji w architekturach wieloprocesorowych oraz odporności systemu na nierównomierne rozłożenie danych. Ze względu na swoją nieregularną strukturę dostępu do danych i znaczną liczbę operacji porządkowania, IS stanowi istotne uzupełnienie pozostałych testów NPB, koncentrując się na problemach wymagających intensywnej pracy z pamięcią i synchronizacją [8].

6.3. Metodologia badań

Badania eksperymentalne zostały zaprojektowane w taki sposób, aby umożliwić porównanie mechanizmów współbieżnych i równoległych w językach Rust oraz C++ przy wykorzystaniu dwóch odmiennych architektur sprzętowych: x86_64 (architektura tradycyjna, Windows/Linux) oraz ARM64 (architektura Apple Silicon – M1, macOS). Dzięki temu możliwa będzie analiza wpływu typu procesora i systemu operacyjnego na wydajność oraz efektywność implementacji.

6.3.1. Środowisko testowe

W ramach środowiska testowego zostały wykorzystane następujące urządzenia wraz z oprogoramowaniem:

Architektura ARM

W ramach architektury ARM został wykorzystany laptop firmy Apple - MacBook Pro z następującymi specyfikacjami:

- procesor Apple M1
- pamięć RAM 16 GB
- system - macOS Sonoma wersja 14.5 (23F79)

Architektura x86_64

W ramach architektury x86_64 został wykorzystany laptop firmy HP z następującymi specyfikacjami:

- procesor
- pamięć RAM 32 GB
- system - Linux Ubuntu 24.04 LST

6.3.2. Procedura testowa

Procedura testowa będzie obejmowała uruchomienie zestawu benchmarków w różnych konfiguracjach, takich jak liczba wątków oraz różne klasy algorytmów w przypadku NPB oraz na dwóch architekturach procesora. Każdy test będzie uruchamiany wielokrotnie - 10 razy, aby uzyskać uśrednione wyniki. Algorytmy zawierają również swoje logi zdzarzeń, które zostaną użyte do weryfikacji poprawności działania samego algorytmu jak i do późniejszej analizy wyników.

6.3.3. Narzędzia pomiarowe

W ramach przeprowadzania testów zostaną wykorzystane następujące narzędzia pomiarowe:

- perf - narzędzie do profilowania kodu, które pozwala na analizę wydajności aplikacji w czasie rzeczywistym (jednakże nie jest dostępne na systemach Apple)
- instruments (Xcode) - oficjalne narzędzie do profilowania od firmy Apple - zamiennik narzędzia *perf* w tym przypadku,
- hwloc - narzędzie pozwalające zbadać zachowanie programu jeżeli chodzi o dostęp do podzespołów komputera
- threadsanitizer - flaga do kompilatora (dla języka C++ w tym przypadku), która pozwala sprawdzić czy w fazie kompilwoania nie zachodzi sytuacja wyścigów

Rozdział 7

Porównanie międzyjęzykowe - programowanie równoległe

W ramach programów równoległych wykorzystano jako wzorzec, gotowe implementacje problemów z zestawu NPB w ramach istniejącej pracy The NAS Parallel Benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures [44] oraz programów bazujących na nich w języku Rust, napisanych w ramach proejktu na studia przez G.Bessa et al. [20].

7.1. Implementacje w języku Rust

7.1.1. Struktura i organizacja kodu

Implementacje w języku Rust charakteryzują się proceduralną strukturą podobną do implementacji C++, wykorzystującą globalne stałe i funkcje standalone. Wszystkie benchmarki (EP, CG, IS) następują jednolity wzorzec architektoniczny oparty na funkcji main():

Listing 7.1: Struktura kodu benchmarków w języku Rust

```
pub struct NPBBenchmark {
// Globalne parametry problemu - stałe w czasie kompilacji
const CLASS: &str = "S";
const M: u32 = 24;
const MM: u32 = M - MK;
const NN: u32 = 1 << MM;
const NK: usize = 1 << MK;
const NPBVERSION: &str = "4.1";
const COMPILETIME: &str = "2024-01-15";

// Główna funkcja proceduralna
fn main() {
    let args: Vec<String> = env::args().collect();
    let num_threads = args.get(1)
        .map(|s| s.parse::<usize>().unwrap_or(1))
        .unwrap_or(1);

    // Konfiguracja Rayon thread pool
    rayon::ThreadPoolBuilder::new()
```

```
        .num_threads(num_threads)
        .build_global()
        .unwrap();

    // Proceduralna logika benchmarku
    algorithm_implementation();
    verification_and_results();
}

// Standalone functions operujące na globalnych danych
fn algorithm_implementation() { /* ... */ }
fn verification_and_results() { /* ... */ }
```

Kluczowe cechy organizacji kodu Rust:

- Proceduralna struktura: główna logika w funkcji main() podobnie jak w C++
- Globalne stałe: parametry problemu definiowane jako const w czasie kompilacji
- Standalone functions: logika algorytmu w oddzielnych funkcjach, nie metodach
- Automatyczne zarządzanie pamięcią: Vec<T> zamiast malloc/free
- Rayon integration: jednolite użycie biblioteki Rayon we wszystkich benchmarkach

7.1.2. Zarządzanie pamięcią

Rust wykorzystuje system własności z automatycznym zarządzaniem pamięcią przez Vec<T>, eliminując ręczną alokację:

Listing 7.2: Zarządzanie pamięcią w benchmarkach NPB w języku Rust

```
impl NPBBenchmark {
fn main() {
    // Automatyczna alokacja przez Vec<T> - odpowiednik malloc w C++
    let mut key_array: Vec<i32> = vec![0; TOTAL_KEYS];
    let mut key_buff1: Vec<i32> = vec![0; MAX_KEY];
    let mut working_data: Vec<f64> = vec![0.0; PROBLEM_SIZE];

    // Thread-local storage dla bezpieczeństwa współbieżności
    thread_local! {
        static THREAD_X: RefCell<Vec<f64>> = RefCell::new(vec![0.0;
    ↪ NK_PLUS]);
    }

    // Główna logika - Vec<T> automatycznie zarządza pamięcią
    algorithm_core(&mut key_array, &mut working_data);

    // Automatyczna dealokacja po wyjściu z zakresu - brak potrzeby
    ↪ free()
}

fn algorithm_core(data: &mut Vec<i32>, working: &mut Vec<f64>) {
    // Borrowing pozwala na bezpieczny dostęp bez przenoszenia
    ↪ ownership
    parallel_processing(data, working);
```

```
    // Kompilator gwarantuje brak data races i use-after-free
}
```

Zalety modelu własności:

- Automatyczna dealokacja: Vec<T> zwalnia pamięć automatycznie po wyjściu z zakresu
- Brak wycieków pamięci: gwarancja na poziomie kompilatora
- Abstrakcje bez narzutu kosztów: brak narzutu wydajnościowego
- Bezpieczeństwo wątków: mechanizm sprawdzania pożyczania eliminuje wyścigi danych

7.1.3. Mechanizmy równoległości

Wszystkie implementacje Rust wykorzystują bibliotekę Rayon dla spójnego podejścia do równoległości:

Listing 7.3: Równoległość w benchmarkach NPB w języku Rust

```
use rayon::prelude::*;

fn parallel_computation(data: &mut Vec<f64>, num_threads: usize) {
    // Konfiguracja thread pool - odpowiednik task_scheduler_init w
    ↪ TBB
    rayon::ThreadPoolBuilder::new()
        .num_threads(num_threads)
        .build_global()
        .unwrap();

    // Równoległe przetwarzanie chunks - odpowiednik #pragma omp
    ↪ parallel for
    let results: Vec<_> = data
        .par_chunks_mut(optimal_chunk_size())
        .map(|chunk| process_chunk(chunk))
        .collect();

    // Równoległa redukcja - odpowiednik reduction(+:sum) w OpenMP
    let final_result = data.par_iter()
        .fold(|| 0.0, |acc, &x| acc + compute_element(x))
        .reduce(|| 0.0, |acc1, acc2| acc1 + acc2);
}

fn process_chunk(chunk: &mut [f64]) -> f64 {
    // Bezpieczne przetwarzanie bez mutexów dzięki par_chunks_mut
    // Rayon gwarantuje thread safety przez podział danych
    chunk.iter_mut().map(|x| *x * 2.0).sum()
}
```

- Harmonogram z kradzieżą pracy (ang. *work-stealing scheduler*) – automatyczne równoważenie obciążenia pomiędzy wątkami poprzez dynamiczne przydzielanie zadań.
- Równoległość danych – naturalne ukierunkowanie na przetwarzanie kolekcji danych w sposób równoległy.
- Bezpieczeństwo w czasie kompilacji – brak warunków wyścigu gwarantowany przez mechanizm pożyczania.

- Ergonomiczne API – intuicyjne przekształcenie kodu sekwencyjnego na równoległy bez znacznego zwiększania złożoności.
- Jednolita abstrakcja: ten sam wzorzec we wszystkich benchmarkach NPB

7.1.4. Specyfika benchmarku EP

Listing 7.4: Implementacja benchmarku EP w języku Rust

```
const CLASS: &str = "S";
const M: u32 = 24;
const MK: u32 = 16;
const MM: u32 = M - MK;
const NN: u32 = 1 << MM;
const NK: usize = 1 << MK;
const NQ: u32 = 10;
const A: f64 = 1220703125.0;
const S: f64 = 271828183.0;

fn main() {
    let args: Vec<String> = env::args().collect();
    let num_threads: usize = args.get(1)
        .map(|s| s.parse::<usize>().unwrap_or(1))
        .unwrap_or(1);

    rayon::ThreadPoolBuilder::new()
        .num_threads(num_threads)
        .build_global()
        .unwrap();

    // Globalne dane - odpowiednik static arrays w C++
    let mut x: Vec<f64> = Vec::with_capacity(NK_PLUS);
    let q: [f64; NQ as usize] = [0.0; NQ as usize];

    // Thread-local storage dla dużych tablic roboczych
    thread_local! {
        static THREAD_X: RefCell<Vec<f64>> = RefCell::new(vec![0.0;
↪ NK_PLUS]);
    }

    // Główna pętla równoległa - odpowiednik #pragma omp parallel for
↪ reduction
    let result = (1..NN+1)
        .collect::<Vec<_>>()
        .par_chunks(chunk_size)
        .fold(|| (0.0, 0.0), |mut acc, chunk| {
            for &k in chunk {
                THREAD_X.with(|x_cell| {
                    let mut x = x_cell.borrow_mut();
                    randdp::vranlc((2 * NK) as i32, &mut t1, A, &mut x
↪ );
```

```
                    // Box-Muller transform w chunks dla lepszej
    ↪ wektoryzacji
                    for chunk_start in (0..NK).step_by(CHUNK_SIZE) {
                        let chunk_end = (chunk_start + CHUNK_SIZE).min
    ↪ (NK);

                        for i in chunk_start..chunk_end {
                            let x1 = 2.0 * x[2 * i] - 1.0;
                            let x2 = 2.0 * x[2 * i + 1] - 1.0;
                            let t1 = x1 * x1 + x2 * x2;

                            if t1 <= 1.0 {
                                let t2 = (-2.0 * t1.ln() / t1).sqrt();
                                let t3 = x1 * t2;
                                let t4 = x2 * t2;
                                let l = t3.abs().max(t4.abs()) as
    ↪ usize;

                                if l < NQ as usize {
                                    local_counts[l] += 1;
                                    acc.0 += t3;
                                    acc.1 += t4;
                                }
                            }
                        }
                    }
                });
            }
            acc
        })
        .reduce(|| (0.0, 0.0), |mut acc1, acc2| {
            acc1.0 += acc2.0;
            acc1.1 += acc2.1;
            acc1
        });

    // Agregacja wyników
    sx = result.0;
    sy = result.1;
}
```

7.1.5. Specyfika benchmarku CG

Listing 7.5: Implementacja benchmarku CG w języku Rust

```
const CLASS: &str = "S";
const NA: i32 = 1400;
const NONZER: i32 = 7;
```

```
const NITER: i32 = 15;
const SHIFT: f64 = 10.0;
const NZ: i32 = NA * (NONZER + 1) * (NONZER + 1);

fn main() {
    let num_threads = if args.len() > 1 {
        args[1].parse::<usize>().unwrap_or(1)
    } else { 1 };

    rayon::ThreadPoolBuilder::new()
        .num_threads(num_threads)
        .build_global()
        .unwrap();

    // Globalne struktury danych - odpowiednik static arrays w C++
    let mut colidx: Vec<i32> = vec![0; NZ as usize];
    let mut rowstr: Vec<i32> = vec![0; (NA + 1) as usize];
    let mut a: Vec<f64> = vec![0.0; NZ as usize];
    let mut x: Vec<f64> = vec![1.0; (NA + 2) as usize];
    let mut z: Vec<f64> = vec![0.0; (NA + 2) as usize];
    let mut p: Vec<f64> = vec![0.0; (NA + 2) as usize];
    let mut q: Vec<f64> = vec![0.0; (NA + 2) as usize];
    let mut r: Vec<f64> = vec![0.0; (NA + 2) as usize];

    // Generowanie macierzy rzadkiej - standalone function
    makea(&mut naa, &mut nzz, &mut a, &mut colidx, &mut rowstr,
          &firstrow, &lastrow, &firstcol, &lastcol,
          &mut arow, &mut acol, &mut aelt, &mut iv, &mut tran, &
↪ amult);

    // Główna pętla iteracyjna
    for it in 1..=NITER {
        conj_grad(&mut colidx, &mut rowstr, &mut x, &mut z, &mut a,
                  &mut p, &mut q, &mut r, &mut rnorm, &naa, &lastcol
↪ ,
                  &firstcol, &lastrow, &firstrow);

        // Równoległe dot products - odpowiednik reduction w OpenMP
        let len = (lastcol - firstcol + 1) as usize;

        norm_temp1 = x[..len]
            .par_iter()
            .zip(&z[..len])
            .map(|(&xi, &zi)| xi * zi)
            .sum();

        norm_temp2 = z[..len]
            .par_iter()
            .map(|&zi| zi * zi)
            .sum();
```

```
        // Równoległa aktualizacja wektora x - odpowiednik #pragma omp
    ↪ for
        x[..=(lastcol - firstcol) as usize].par_iter_mut()
            .zip(&z[..=(lastcol - firstcol) as usize])
            .for_each(|(xi, &zi)| {
                *xi = norm_temp2 * zi;
            });
    }
}

// Standalone function - odpowiednik funkcji C
fn conj_grad(colidx: &mut Vec<i32>, rowstr: &mut Vec<i32>,
             x: &mut Vec<f64>, z: &mut Vec<f64>, a: &mut Vec<f64>,
             p: &mut Vec<f64>, q: &mut Vec<f64>, r: &mut Vec<f64>,
             rnorm: &mut f64, /* inne parametry */) {
    let cgitmax: i32 = 25;

    // Sekwencyjna inicjalizacja - jak w C++
    for j in 0..=*naa {
        let j = j as usize;
        q[j] = 0.0;
        z[j] = 0.0;
        r[j] = x[j];
        p[j] = r[j];
    }

    // Równoległe obliczenie rho - odpowiednik #pragma omp for
    ↪ reduction(+:rho)
    rho = (0..=(*lastcol - *firstcol))
        .into_par_iter()
        .map(|j| {
            let idx = j as usize;
            r[idx] * r[idx]
        })
        .sum();

    for _cgit in 1..=cgitmax {
        // Równoległe mnożenie macierz-wektor - odpowiednik #pragma
    ↪ omp for
        q.par_chunks_mut(1)
            .enumerate()
            .for_each(|(j, q_slice)| {
                if j <= (*lastrow - *firstrow) as usize {
                    let mut sum = 0.0;
                    for k in rowstr[j]..rowstr[j + 1] {
                        let k = k as usize;
                        let cidx = colidx[k];
                        if cidx >= 0 && (cidx as usize) < p.len() {
                            sum += a[k] * p[cidx as usize];
```

```
                    }
                }
                q_slice[0] = sum;
            }
        });

        // Równoległe obliczenie dot product
        d = (0..=(*lastcol - *firstcol))
            .into_par_iter()
            .map(|j| {
                let j = j as usize;
                p[j] * q[j]
            })
            .sum();

        // Równoległe aktualizacje wektorów
        let range = 0..=(*lastcol - *firstcol) as usize;
        z[range.clone()].par_iter_mut()
            .zip(&p[range.clone()])
            .for_each(|(z_val, &p_val)| {
                *z_val = *z_val + alpha * p_val;
            });

        r[range.clone()].par_iter_mut()
            .zip(&q[range.clone()])
            .for_each(|(r_val, &q_val)| {
                *r_val = *r_val - alpha * q_val;
            });

        // Nowe rho z równoległą redukcją
        rho = r[range.clone()].par_iter()
            .map(|&r_val| r_val * r_val)
            .sum();
    }
}
```

7.1.6. Specyfika benchmarku IS

Listing 7.6: Implementacja benchmarku IS w języku Rust

```
const CLASS: &str = "S";
const TOTAL_KEYS_LOG_2: u32 = 16;
const MAX_KEY_LOG_2: u32 = 11;
const NUM_BUCKETS_LOG_2: u32 = 9;
const TOTAL_KEYS: usize = 1 << TOTAL_KEYS_LOG_2;
const MAX_KEY: usize = 1 << MAX_KEY_LOG_2;
const NUM_BUCKETS: usize = 1 << NUM_BUCKETS_LOG_2;
```

```rust
// Struktura enkapsulująca złożony stan bucket sort - wyjątek od
    ↪ proceduralnej reguły
struct ISBenchmark {
    key_array: Vec<KeyType>,
    key_buff1: Vec<KeyType>,
    key_buff2: Vec<KeyType>,
    bucket_size: Vec<Vec<KeyType>>,
    bucket_ptrs: Vec<KeyType>,
    num_threads: usize,
}

impl ISBenchmark {
    fn new(num_threads: usize) -> Self {
        ISBenchmark {
            key_array: vec![0; TOTAL_KEYS],
            key_buff1: vec![0; MAX_KEY],
            key_buff2: vec![0; TOTAL_KEYS],
            bucket_size: vec![vec![0; NUM_BUCKETS]; num_threads],
            bucket_ptrs: vec![0; NUM_BUCKETS + 1],
            num_threads,
        }
    }

    fn rank(&mut self, iteration: i32) {
        let shift = MAX_KEY_LOG_2 - NUM_BUCKETS_LOG_2;

        // Równoległe bucket counting - odpowiednik #pragma omp
    ↪ parallel for
        let chunk_size = (TOTAL_KEYS + self.num_threads - 1) / self.
    ↪ num_threads;

        self.key_array
            .par_chunks(chunk_size)
            .zip(&mut self.bucket_size)
            .for_each(|(chunk, thread_bucket)| {
                for &key in chunk {
                    let bucket_idx = (key >> shift) as usize;
                    if bucket_idx < NUM_BUCKETS {
                        thread_bucket[bucket_idx] += 1;
                    }
                }
            });

        // Równoległe bucket sort z atomics
        (0..NUM_BUCKETS).into_par_iter().for_each(|i| {
            let key_buff1_shared = Arc::new(Mutex::new(&mut self.
    ↪ key_buff1));
            // Complex parallel bucket redistribution
        });
    }
```

```
}

fn main() {
    let num_threads = if args.len() > 1 {
        args[1].parse::<usize>().unwrap_or(1)
    } else { 1 };

    rayon::ThreadPoolBuilder::new()
        .num_threads(num_threads)
        .build_global()
        .unwrap();

    let mut benchmark = ISBenchmark::new(num_threads);

    // Proceduralne wywołania na strukturze
    benchmark.create_seq();

    for iteration in 1..=MAX_ITERATIONS {
        benchmark.rank(iteration);
    }

    benchmark.full_verify();
}
```

- Losowy dostęp do pamięci – nieprzewidywalny wzorzec odwołań do pamięci, który utrudnia efektywne wykorzystanie pamięci podręcznej procesora.
- Nierównomierne obciążenie – niejednorodne rozłożenie kluczy pomiędzy kubełki (buckets), prowadzące do nieefektywnego wykorzystania zasobów obliczeniowych.
- Narzut synchronizacji – konieczność stosowania operacji atomowych podczas równoczesnej aktualizacji kubełków, co może obniżać wydajność.

7.2. Implementacje w C++ (OpenMP)

7.2.1. Struktura i organizacja kodu

Implementacje C++ z OpenMP następują klasyczny wzorzec proceduralny wywodzący się z oryginalnych implementacji Fortran:

Listing 7.7: Struktura kodu benchmarków w języku C++ z OpenMP

```
// Globalne zmienne statyczne dla wszystkich benchmarków
#if defined(
   ↪ DO_NOT_ALLOCATE_ARRAYS_WITH_DYNAMIC_MEMORY_AND_AS_SINGLE_DIMENSION
   ↪ )
static DataType primary_array[MAX_SIZE];
static DataType secondary_array[MAX_SIZE];
#else
static DataType (*primary_array)=(DataType*)malloc(sizeof(DataType)*(
   ↪ MAX_SIZE));
static DataType (*secondary_array)=(DataType*)malloc(sizeof(DataType)
   ↪ *(MAX_SIZE));
```

```
#endif

// Funkcje proceduralne operujące na globalnych danych
static void algorithm_core(/* parametry */);
static void initialize_data(/* parametry */);
static void verify_results(/* parametry */);

int main(int argc, char **argv){
    // Główna logika bez enkapsulacji
    initialize_data(/* argumenty */);
    algorithm_core(/* argumenty */);
    verify_results(/* argumenty */);
}
```

- Globalny stan – wszystkie dane przechowywane są jako zmienne globalne, co upraszcza współdzielenie zasobów pomiędzy wątkami.
- Styl proceduralny – funkcje operują bezpośrednio na globalnych strukturach danych, zgodnie z tradycyjnym podejściem programowania proceduralnego.
- Minimalna enkapsulacja – brak hermetyzacji danych i logiki prowadzi do luźnej struktury kodu oraz ograniczonej kontroli nad jego modyfikacjami.
- Kompatybilność wsteczna – zachowanie zgodności z kodem Fortran ułatwia migrację i integrację z istniejącymi systemami HPC.
- Warunkowa alokacja – możliwość wyboru pomiędzy alokacją statyczną a dynamiczną umożliwia dostosowanie strategii zarządzania pamięcią do charakterystyki platformy.

7.2.2. Zarządzanie pamięcią

Listing 7.8: Zarządzanie pamięcią w benchmarkach C++ z OpenMP

```
// Warunkowa strategia alokacji pamięci
#if defined(
   ↪ DO_NOT_ALLOCATE_ARRAYS_WITH_DYNAMIC_MEMORY_AND_AS_SINGLE_DIMENSION
   ↪ )
static int colidx[NZ];
static int rowstr[NA+1];
static double a[NZ];
static double x[NA+2];
#else
static int (*colidx)=(int*)malloc(sizeof(int)*(NZ));
static int (*rowstr)=(int*)malloc(sizeof(int)*(NA+1));
static double (*a)=(double*)malloc(sizeof(double)*(NZ));
static double (*x)=(double*)malloc(sizeof(double)*(NA+2));
#endif

int main(int argc, char **argv) {
    // Brak sprawdzania błędów alokacji
    // Brak eksplicytnego zwalniania pamięci
    // Potencjalne wycieki pamięci przy wcześniejszym wyjściu

    // Warunkowe zwalnianie (jeśli w ogóle)
    #if !defined(
       ↪ DO_NOT_ALLOCATE_ARRAYS_WITH_DYNAMIC_MEMORY_AND_AS_SINGLE_DIMENSION
       ↪ )
```

```
    // Często brakuje free() calls
    #endif
}
```

- Ręczne zarządzanie pamięcią – eksplicytne wywołania `malloc` i `free` bez wsparcia automatyzacji, co zwiększa ryzyko błędów programistycznych.
- Brak sprawdzania błędów alokacji – operacje przydziału pamięci mogą zakończyć się niepowodzeniem, a ich wyniki często nie są weryfikowane.
- Wycieki pamięci – brak gwarancji, że zaalokowane zasoby zostaną zwolnione, co prowadzi do stopniowego wzrostu zużycia pamięci.
- Niezdefiniowane zachowanie – możliwość błędów takich jak *use-after-free* czy *double-free*, które mogą skutkować niestabilnością programu lub lukami bezpieczeństwa.
- Przekroczenia bufora (buffer overflows) – brak automatycznego sprawdzania granic tablic sprzyja nadpisywaniu pamięci poza przydzielonym obszarem.

7.2.3. Mechanizmy równoległości

Listing 7.9: Mechanizmy równoległości w benchmarkach C++ z OpenMP

```
static void parallel_computation(/* parametry */) {
    double local_sum = 0.0;

    // Klasyczne dyrektywy OpenMP
    #pragma omp parallel for reduction(+:local_sum) schedule(
        ↪ static)
    for (int64_t i = 0; i < problem_size; i++) {
        // Równoległe przetwarzanie z ręczną kontrolą
            ↪ synchronizacji
        local_sum += compute_element(i);
    }

    // Sekcje krytyczne dla complex operations
    #pragma omp parallel
    {
        double thread_local_data[THREAD_ARRAY_SIZE];

        #pragma omp for schedule(dynamic, chunk_size)
        for (int64_t j = 0; j < iterations; j++) {
            // Thread-local processing
        }

        #pragma omp critical
        {
            // Synchronization of shared state
            update_global_results(thread_local_data);
        }
    }
}
```

- Oparte na dyrektywach – wykorzystanie konstrukcji `#pragma` do deklaratywnego sterowania równoległością w kodzie źródłowym.

- Ręczna synchronizacja – konieczność samodzielnego zarządzania sekcjami krytycznymi, barierami oraz mechanizmami ochrony danych współdzielonych.
- Jawne planowanie – programista ma bezpośrednią kontrolę nad strategiami podziału pracy, takimi jak `static`, `dynamic` czy `guided`.
- Model oparty na wątkach – równoległość realizowana przez bezpośrednie zarządzanie wątkami systemowymi, co wpływa na wydajność i kontrolę niskopoziomową.
- Dojrzały ekosystem – bogaty zbiór dyrektyw, opcji konfiguracyjnych i narzędzi wspierających optymalizację oraz profilowanie programów równoległych.

7.2.4. Specyfika benchmarku EP

Listing 7.10: Implementacja benchmarku EP w języku C++ z OpenMP

```
#pragma omp parallel
{
    double t1, t2, t3, t4, x1, x2;
    int kk, i, ik, l;
    double qq[NQ];        // Prywatna kopia q[0:NQ-1]
    double x[NK_PLUS];    // Lokalna tablica na stosie

    for (i = 0; i < NQ; i++) qq[i] = 0.0;

    #pragma omp for reduction(+:sx,sy)
    for(k=1; k<=np; k++){
        kk = k_offset + k;
        t1 = S;
        t2 = an;
        int thread_id = omp_get_thread_num();

        // Znajdowanie ziarna dla tego kk
        for(i=1; i<=100; i++){
            ik = kk / 2;
            if((2*ik)!=kk){t3=randlc(&t1,t2);}
            if(ik==0){break;}
            t3=randlc(&t2,t2);
            kk=ik;
        }

        // Generowanie liczb pseudolosowych
        if(timers_enabled && thread_id==0){timer_start(2);}
        vranlc(2*NK, &t1, A, x);
        if(timers_enabled && thread_id==0){timer_stop(2);}

        // Box-Muller transform
        if(timers_enabled && thread_id==0){timer_start(1);}
        for(i=0; i<NK; i++){
            x1 = 2.0 * x[2*i] - 1.0;
            x2 = 2.0 * x[2*i+1] - 1.0;
            t1 = pow2(x1) + pow2(x2);
            if(t1 <= 1.0){
                t2 = sqrt(-2.0 * log(t1) / t1);
                t3 = (x1 * t2);
                t4 = (x2 * t2);
```

```
                    l = max(fabs(t3), fabs(t4));
                    qq[l] += 1.0;
                    sx = sx + t3;
                    sy = sy + t4;
                }
            }
            if(timers_enabled && thread_id==0){timer_stop(1);}
        }

        #pragma omp critical
        {
            for (i = 0; i <= NQ - 1; i++) q[i] += qq[i];
        }
    }
```

7.2.5. Specyfika benchmarku CG

Listing 7.11: Implementacja benchmarku CG w języku C++ z OpenMP

```
// Globalne zmienne statyczne - identyczne w OpenMP i TBB
#if defined(
   ↪ DO_NOT_ALLOCATE_ARRAYS_WITH_DYNAMIC_MEMORY_AND_AS_SINGLE_DIMENSION
   ↪ )
static int colidx[NZ];
static int rowstr[NA+1];
static int iv[NA];
static int arow[NA];
static int acol[NAZ];
static double aelt[NAZ];
static double a[NZ];
static double x[NA+2];
static double z[NA+2];
static double p[NA+2];
static double q[NA+2];
static double r[NA+2];
#else
static int (*colidx)=(int*)malloc(sizeof(int)*(NZ));
static int (*rowstr)=(int*)malloc(sizeof(int)*(NA+1));
static int (*iv)=(int*)malloc(sizeof(int)*(NA));
static int (*arow)=(int*)malloc(sizeof(int)*(NA));
static int (*acol)=(int*)malloc(sizeof(int)*(NAZ));
static double (*aelt)=(double*)malloc(sizeof(double)*(NAZ));
static double (*a)=(double*)malloc(sizeof(double)*(NZ));
static double (*x)=(double*)malloc(sizeof(double)*(NA+2));
static double (*z)=(double*)malloc(sizeof(double)*(NA+2));
static double (*p)=(double*)malloc(sizeof(double)*(NA+2));
static double (*q)=(double*)malloc(sizeof(double)*(NA+2));
static double (*r)=(double*)malloc(sizeof(double)*(NA+2));
#endif

int main(int argc, char **argv){
    // Inicjalizacja bez sprawdzania błędów
```

```
    makea(naa, nzz, a, colidx, rowstr, firstrow, lastrow, firstcol,
       ↪ lastcol,
            arow, (int(*)[NONZER+1])(void*)acol,
            (double(*)[NONZER+1])(void*)aelt, iv);

    // Główna pętla iteracyjna
    for(it = 1; it <= NITER; it++){
        if(timeron){timer_start(T_CONJ_GRAD);}
        conj_grad(colidx, rowstr, x, z, a, p, q, r, &rnorm);
        if(timeron){timer_stop(T_CONJ_GRAD);}

        // Sekwencyjne obliczenia norm
        norm_temp1 = 0.0;
        norm_temp2 = 0.0;
        for(j = 0; j < lastcol - firstcol + 1; j++){
            norm_temp1 = norm_temp1 + x[j]*z[j];
            norm_temp2 = norm_temp2 + z[j]*z[j];
        }
        norm_temp2 = 1.0 / sqrt(norm_temp2);
        zeta = SHIFT + 1.0 / norm_temp1;

        // Sekwencyjna aktualizacja x
        for(j = 0; j < lastcol - firstcol + 1; j++){
            x[j] = norm_temp2 * z[j];
        }
    }
}

static void conj_grad(int colidx[], int rowstr[], double x[], double
   ↪ z[],
                      double a[], double p[], double q[], double r
                         ↪ [],
                      double* rnorm){
    int j, k;
    int cgit, cgitmax;
    double d, sum, rho, rho0, alpha, beta;

    cgitmax = 25;
    rho = 0.0;

    // Sekwencyjna inicjalizacja
    for(j = 0; j < naa+1; j++){
        q[j] = 0.0;
        z[j] = 0.0;
        r[j] = x[j];
        p[j] = r[j];
    }

    // OpenMP: Równoległe obliczenie rho
    #pragma omp parallel for reduction(+:rho)
    for(j = 0; j < lastcol-firstcol+1; j++){
        rho += r[j]*r[j];
    }
```

```
for(cgit = 1; cgit <= cgitmax; cgit++){
    // OpenMP: Równoległe mnożenie macierz-wektor
    #pragma omp parallel for
    for(j = 0; j < lastrow-firstrow+1; j++){
        double sum = 0.0;
        for(k = rowstr[j]; k < rowstr[j+1]; k++){
            sum = sum + a[k]*p[colidx[k]];
        }
        q[j] = sum;
    }

    // OpenMP: Równoległe obliczenie dot product
    d = 0.0;
    #pragma omp parallel for reduction(+:d)
    for (j = 0; j < lastcol-firstcol+1; j++) {
        d += p[j]*q[j];
    }

    alpha = rho / d;
    rho0 = rho;
    rho = 0.0;

    // OpenMP: Równoległe aktualizacje wektorów z redukcją
    #pragma omp parallel for reduction(+:rho)
    for(j = 0; j < lastcol-firstcol+1; j++){
        z[j] += alpha*p[j];
        r[j] -= alpha*q[j];
        rho += r[j]*r[j];
    }

    beta = rho / rho0;

    // OpenMP: Równoległa aktualizacja p
    #pragma omp parallel for
    for(j = 0; j < lastrow-firstrow+1; j++){
        p[j] = r[j] + beta*p[j];
    }
}

// Obliczenie residuum
sum = 0.0;
#pragma omp parallel for
for(j = 0; j < lastrow-firstrow+1; j++){
    double d = 0.0;
    for(k = rowstr[j]; k < rowstr[j+1]; k++){
        d = d + a[k]*z[colidx[k]];
    }
    r[j] = d;
}

#pragma omp parallel for reduction(+:sum)
for(j = 0; j < lastcol-firstcol+1; j++){
```

```
        double d = x[j] - r[j];
        sum += d*d;
    }

    *rnorm = sqrt(sum);
}
```

7.2.6. Specyfika benchmarku IS

Listing 7.12: Implementacja benchmarku IS w języku C++ z OpenMP

```
static void rank(/* parametry */) {
    // Parallel histogram construction
    #pragma omp parallel
    {
        int key_buff_ptr[MAX_KEY];

        #pragma omp for
        for (int i = 0; i < num_keys; i++) {
            key_buff_ptr[key_buff[i]]++;
        }

        // Complex synchronization for bucket sort
        #pragma omp barrier

        #pragma omp for
        for (int i = 0; i < num_buckets; i++) {
            // Redistribute keys
        }
    }
}
```

7.3. Implementacje w C++ (TBB)

7.3.1. Struktura i organizacja kodu

Implementacje TBB zachowują proceduralną strukturę z minimalnymi modyfikacjami względem wersji OpenMP:

Listing 7.13: Implementacja TBB - struktura kodu

```
#include "tbb/parallel_for.h"
#include "tbb/parallel_reduce.h"
#include "tbb/blocked_range.h"
#include "tbb/task_scheduler_init.h"
#include "tbb/mutex.h"

// Identyczne globalne zmienne jak w wersji OpenMP
#if defined(
    ↪ DO_NOT_ALLOCATE_ARRAYS_WITH_DYNAMIC_MEMORY_AND_AS_SINGLE_DIMENSION
    ↪ )
static DataType primary_array[MAX_SIZE];
static DataType secondary_array[MAX_SIZE];
```

```
#else
static DataType (*primary_array)=(DataType*)malloc(sizeof(DataType)*(
    ↪ MAX_SIZE));
static DataType (*secondary_array)=(DataType*)malloc(sizeof(DataType)
    ↪ *(MAX_SIZE));
#endif

int main(int argc, char **argv){
    // Inicjalizacja TBB scheduler
    int num_workers;
    if(const char * nw = std::getenv("TBB_NUM_THREADS")) {
        num_workers = atoi(nw);
    } else {
        num_workers = 1;
    }

    tbb::task_scheduler_init init(num_workers);

    // Reszta kodu identyczna jak w wersji klasycznej
    algorithm_implementation();
}
```

- Minimalne zmiany strukturalne – zachowanie proceduralnego stylu programowania bez konieczności znaczącej przebudowy istniejącego kodu.
- Jawna inicjalizacja planisty zadań – konieczność ręcznego utworzenia i konfiguracji obiektu planisty (`task_scheduler_init`) w celu zarządzania wątkami.
- Równoległość oparta na funkcjach – zastąpienie dyrektyw kompilatora wywołaniami funkcji bibliotecznych, takich jak `parallel_for` czy `parallel_reduce`.
- Te same problemy z zarządzaniem pamięcią – biblioteka nie wprowadza dodatkowych zabezpieczeń; nadal istnieje ryzyko błędów takich jak wycieki pamięci, przekroczenia bufora czy dostęp do już zwolnionej pamięci.

7.3.2. Zarządzanie pamięcią

TBB nie wprowadza ulepszeń w zarządzaniu pamięcią - wykorzystuje identyczny model jak OpenMP z tymi samymi problemami.

7.3.3. Mechanizmy równoległości

TBB zastępuje dyrektywy OpenMP funkcyjnym API z work-stealing scheduler:

Listing 7.14: Implementacja TBB - równoległość

```
// Zastąpienie OpenMP reduction przez TBB parallel_reduce
static void parallel_computation(/* parametry */) {
// Parallel reduction z lambda expressions
double result = tbb::parallel_reduce(
    tbb::blocked_range<size_t>(0, problem_size),
    0.0,
    [&](const tbb::blocked_range<size_t>& r, double worker_sum){
        for (size_t i = r.begin(); i != r.end(); i++) {
            worker_sum += compute_element(i);
        }
```

```
        return worker_sum;
    },
    std::plus<double>()
);

// Parallel for z blocked ranges
tbb::parallel_for(
    tbb::blocked_range<size_t>(0, iterations),
    [&](const tbb::blocked_range<size_t>& r){
        for (size_t i = r.begin(); i != r.end(); i++) {
            process_iteration(i);
        }
    }
);
}
```

- Funkcyjny interfejs programowania – zastosowanie wyrażeń lambda i funkcji wyższego rzędu zamiast dyrektyw kompilatora, co sprzyja większej elastyczności kompozycyjnej.
- Harmonogram z kradzieżą pracy (ang. *work-stealing scheduler*) – automatyczne równoważenie obciążenia pomiędzy wątkami poprzez dynamiczne przydzielanie zadań zależnie od dostępnych zasobów.
- Podziały blokowe – automatyczny podział przestrzeni danych na zakresy, co ułatwia równoległe przetwarzanie dużych kolekcji bez konieczności ręcznego zarządzania iteracjami.
- Bezpieczeństwo typów – silniejsze mechanizmy typowania w porównaniu do OpenMP, co zmniejsza liczbę błędów w czasie kompilacji.
- Większy narzut składniowy – bardziej rozwlekła i złożona składnia niż w przypadku OpenMP, co może wpływać na czytelność i prostotę kodu.

7.3.4. Specyfika poszczególnych benchmarków

TBB implementacje różnią się głównie sposobem wyrażenia równoległości, zachowując identyczną logikę algorytmów:

- EP: Zastąpienie #pragma omp for reduction przez parallel_reduce z lambda
- CG: Konwersja sparse matrix operations na parallel_for z blocked_range
- IS: Implementacja bucket sort przez parallel_for z mutex synchronization

7.4. Implementacje w C++ (nowoczesne podejście)

7.4.1. Struktura i organizacja kodu

Nowoczesne implementacje C++ wykorzystują obiektowy design z RAII i enkapsulacją:

Listing 7.15: Implementacja nowoczesnego C++ - struktura kodu

```
namespace npb {

class NPBBenchmark {
public:
    explicit NPBBenchmark(char class_type, int num_threads = 1);
    virtual ~NPBBenchmark() = default;

    virtual void run() = 0;
```

```
    virtual bool verify() const = 0;
    virtual double get_mops() const = 0;

protected:
    // Enkapsulowane dane z automatycznym zarządzaniem
    struct ProblemParameters {
        int64_t size;
        char class_type;
        int iterations;
    } params_;

    // RAII dla wszystkich zasobów
    std::vector<double> primary_data_;
    std::vector<double> secondary_data_;

    // Wyniki i stan
    BenchmarkResults results_;
    bool verified_ = false;
    bool timers_enabled_ = false;
    int num_threads_;

private:
    virtual void init() = 0;
    virtual void compute() = 0;
    virtual bool verify_results() = 0;
};

class EPBenchmark : public NPBBenchmark {
    // Konkretna implementacja
};

}
```

- RAII (Resource Acquisition Is Initialization) – automatyczne zarządzanie zasobami przy użyciu konstruktorów i destruktorów, co eliminuje potrzebę jawnego zwalniania pamięci oraz innych zasobów systemowych.
- Enkapsulacja – hermetyzacja danych oraz logiki w obrębie klas i struktur, co sprzyja modularności, izolacji błędów i możliwości wielokrotnego wykorzystania kodu.
- Bezpieczeństwo typów – wykorzystanie nowoczesnych konstrukcji językowych C++, takich jak typy wyliczeniowe, silne typowanie i szablony, pozwalające na wczesne wykrywanie błędów.
- Zarządzanie zasobami – użycie inteligentnych wskaźników (`std::unique_ptr`, `std::shared_ptr`) oraz kontraktów programistycznych (np. `noexcept`, `[[nodiscard]]`), co zwiększa niezawodność i bezpieczeństwo kodu.

7.4.2. Zarządzanie pamięcią

Listing 7.16: Implementacja nowoczesnego C++ - zarządzanie pamięcią

```
class NPBBenchmark {
private:
    // std::vector automatycznie zarządza pamięcią
    std::vector<double> data_;
```

```
    std::vector<int64_t> indices_;
    std::unique_ptr<WorkingMemory> working_memory_;

public:
    explicit NPBBenchmark(const ProblemParameters& params)
        : params_(params),
          data_(params.size),
          indices_(params.index_size),
          working_memory_(std::make_unique<WorkingMemory>(params)) {
        // Automatyczna inicjalizacja z exception safety
        init_data_structures();
    }

    void worker_task(int tid, int num_workers) {
        // Automatyczne zarządzanie lokalnych zasobów
        std::vector<double> local_working_array(params_.local_size);
        std::vector<double> local_results(params_.result_size, 0.0);

        // RAII gwarantuje cleanup nawet przy wyjątkach
        process_local_data(local_working_array, local_results);

        // Thread-safe aggregation
        aggregate_results(local_results);
    }

    // Destruktor automatycznie zwalnia wszystkie zasoby
    ~NPBBenchmark() = default;
};
```

7.4.3. Mechanizmy równoległości

Listing 7.17: Implementacja nowoczesnego C++ - równoległość

```
class NPBBenchmark {
    protected:
        void parallel_computation() {
            // Modern C++ threading z RAII
            std::vector<std::thread> threads;
            threads.reserve(num_threads_);

            // Exception-safe thread management
            try {
                for (int i = 0; i < num_threads_; i++) {
                    threads.emplace_back([this, i]() {
                        this->worker_task(i, num_threads_);
                    });
                }

                // RAII ensures proper cleanup
                for (auto& thread : threads) {
                    thread.join();
                }
            } catch (...) {
```

```
                // Exception safety - cleanup threads
                for (auto& thread : threads) {
                    if (thread.joinable()) {
                        thread.join();
                    }
                }
                throw;
            }
        }

        void thread_safe_aggregation(const LocalResults& local) {
            // Modern synchronization primitives
            std::lock_guard<std::mutex> lock(results_mutex_);
            results_.aggregate(local);
        }
    };
```

7.5. Porównanie międzyjęzykowe

7.5.1. Architektura i organizacja kodu

Tab. 7.1: Porównanie aspektów zarządzania i organizacji kodu w Rust i różnych stylach C++

Aspekt	**Rust**	**C++ (OpenMP)**	**C++ (TBB)**	**C++ (nowoczesny)**
Organizacja kodu	Modułowa, strukturalna	Proceduralna, globalna	Proceduralna, globalna	Obiektowa, enkapsulowana
Zarządzanie stanem	Ownership + borrowing	Zmienne globalne	Zmienne globalne	RAII + smart pointers
Enkapsulacja	Wysoka (moduły + traits)	Brak	Brak	Średnia (klasy)
Bezpieczeństwo typów	Compile-time guarantees	Runtime checks	Runtime checks	Mixed approach
Obsługa błędów	`Result<T,E>` + `Option<T>`	Minimalna / brak	Minimalna / brak	Wyjątki + `std::optional`
Bezpieczeństwo pamięci	Gwarantowane	Ręczne zarządzanie	Ręczne zarządzanie	RAII-based

7.5.2. Zarządzanie pamięcią

Tab. 7.2: Porównanie modelu alokacji i bezpieczeństwa pamięci w Rust i różnych stylach C++

Aspekt	**Rust**	**C++ (OpenMP)**	**C++ (TBB)**	**C++ (nowoczesny)**
Model alokacji	Automatyczny (ownership)	Ręczny (`malloc`/`free`)	Ręczny (`malloc`/`free`)	Automatyczny (`std::vector`)
Wycieki pamięci	Niemożliwe	Możliwe	Możliwe	Rzadkie (RAII)
Use-after-free	Niemożliwe	Możliwe	Możliwe	Rzadkie
Przepełnienia bufora	Sprawdzanie granic	Możliwe	Możliwe	Sprawdzanie granic
Podwójne zwolnienie	Niemożliwe	Możliwe	Możliwe	Rzadkie

Rozdział 8

Porównanie międzyjęzykowe - programowanie współbieżne

Rozdział 9

Analiza wyników

9.1. Programowanie współbieżne

9.2. Programowanie równoległe

Rozdział 10

Wnioski oraz rekomendacje

Przeprowadzone badanie ujawniło fundamentalne ograniczenie narzędzia hwloc-ps na platformie macOS, szczególnie w architekturze Apple Silicon, wynikające z celowych restrykcji systemowych w jądrze XNU. Jak wykazano w [3], brak implementacji interfejsów sched_setaffinity()/sched_getaffinity() uniemożliwia odczyt i kontrolę przypisań procesów do rdzeni (process-to-core binding), co stanowi istotną barierę metodologiczną w porównawczych badaniach wydajnościowych między architekturami ARM (M1) i x86_64. W odróżnieniu od pełnej funkcjonalności hwloc-ps w systemie Linux [6], gdzie narzędzie precyzyjnie raportuje przypisania wątków, na macOS możliwe jest jedynie wykrywanie topologii sprzętowej przez lstopo – przy użyciu mechanizmów sysctl. Ograniczenie to, uniemożliwiło bezpośrednią weryfikację wpływu przypisań wątków (thread pinning) na wydajność implementacji współbieżnych w językach Rust/C++.

W trakcie prac nad implementacją aplikacji w języku C++ z wykorzystaniem biblioteki Threading Building Blocks (TBB) zaobserwowano istotne różnice w procesie budowania i konfiguracji projektu pomiędzy platformami opartymi na architekturze x86-64 (Linux) a systemem macOS z procesorem Apple M1 (ARM64). Jest to również potwierdzone przez pracę [58]. Aplikacja, która kompilowała się bezproblemowo i działała optymalnie w środowisku x86, wymagała licznych modyfikacji przy próbie przeniesienia jej na platformę Apple Silicon. W szczególności konieczne było ręczne dostosowanie flag kompilatora, aktualizacja konfiguracji CMake z uwzględnieniem architektury ARM oraz zastosowanie społecznościowych łatek w celu rozwiązania problemów z rozpoznawaniem architektury („Unknown architecture flag: -arch armv4t”). Te trudności potwierdzają, że proces przenoszenia aplikacji opartych na TBB na macOS z procesorem M1 nie jest trywialny i wymaga świadomego podejścia projektowego oraz głębszego zrozumienia różnic międzyplatformowych — zarówno na poziomie sprzętowym, jak i systemowym. Dodatkowo są też prace, które pokazują, że testy na maszynie wirtualnej z emulacją Intel wykazały spadek wydajności w porównaniu z natywnym wykonaniem na M1 [29].

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 lan- guages. 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 11

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] C++ - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/C%2B%2B. [Dostęp 16-03-2025].

[2] Comparing Programming Languages — cs.ucf.edu. https://www.cs.ucf.edu/~leavens/ComS541Fall98/hw-pages/comparing/. [Dostęp 16-03-2025].

[3] CPU binding on MacOS · Issue #555 · open-mpi/hwloc — github.com. https://github.com/open-mpi/hwloc/issues/555. [Dostęp 01-06-2025].

[4] CUDA Toolkit - Free Tools and Training — developer.nvidia.com. https://developer.nvidia.com/cuda-toolkit. [Dostęp 22-03-2025].

[5] Execution control library (since C++26) - cppreference.com — en.cppreference.com. https://en.cppreference.com/w/cpp/execution.html. [Dostęp 31-05-2025].

[6] Hardware Locality (hwloc): Hardware Locality — hwloc.readthedocs.io. https://hwloc.readthedocs.io/en/stable/. [Dostęp 01-06-2025].

[7] How do you compare programming languages in a coding interview? — linkedin.com. https://www.linkedin.com/advice/1/how-do-you-compare-programming-languages-coding. [Dostęp 16-03-2025].

[8] NAS Parallel Benchmarks — nas.nasa.gov. https://www.nas.nasa.gov/software/npb.html. [Accessed 17-04-2025].

[9] Rust GPU — rust-gpu.github.io. https://rust-gpu.github.io. [Dostęp 22-03-2025].

[10] Rust (programming language) - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Rust_(programming_language). [Dostęp 16-03-2025].

[11] Safety: A comparaison between Rust, C++ and Go | Hacker News — news.ycombinator.com. https://news.ycombinator.com/item?id=32285122. [Dostęp 13-03-2025].

[12] vulkano - Rust — docs.rs. https://docs.rs/vulkano/0.12.0/vulkano/. [Dostęp 22-03-2025].

[13] wgpu: portable graphics library for Rust — wgpu.rs. https://wgpu.rs. [Dostęp 22-03-2025].

[14] *Professional CUDA C Programming*. Wrox Press Ltd., GBR, wydanie 1st, 2014.

[15] 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.

[16] S. Ali, S. Qayyum. A pragmatic comparison of four different programming languages, 06 2021.

[17] Z. Alomari, O. Halimi, K. Sivaprasad, C. Pandit. Comparative studies of six programming languages. 04 2015.

[18] 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.

[19] V. Besozzi. Ppl: Structured parallel programming meets rust. *2024 32nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, strony 78–87, 2024.

[20] G. L. Bessa, L. M. D. S. Borela, J. A. Soares. Npb-rust: Nas parallel benchmarks in rust. `https://github.com/glbessa/NPB-Rust`, 2025. Dostęp: 2025-05-18.

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

[22] K. Bobrov. *Grokking Concurrency*. Manning, 2024.

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

[24] 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 09-03-2025].

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

[26] 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.

[27] 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.

[28] R. DeWolf. Introducing Rainbow: Compare the Performance of Different Programming Languages — medium.com. `https://medium.com/better-programming/introducing-rainbow-compare-the-performance-of-different-programming-languages-f08a67453cd4`. [Dostęp 16-03-2025].

[29] J. Duke. Memory forensics comparison of apple m1 and intel architecture using volatility framework. Master's thesis, Louisiana State University, 2021.

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

[31] 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.

[32] 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.

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

[34] H. Heyman, L. Brandefelt. A comparison of performance & implementation complexity of multithreaded applications in rust, java and c++, 2020.

[35] 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].

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

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

[38] 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.

[39] 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.

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

[41] K. Lesiński.

[42] 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.

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

[44] J. Löff, D. Griebler, G. Mencagli, G. Araujo, M. Torquati, M. Danelutto, L. G. Fernandes. The nas parallel benchmarks for evaluating c++ parallel programming frameworks on shared-memory architectures. *Future Generation Computer Systems*, 125:743–757, 2021.

[45] E. M. Martins, L. G. Faé, R. B. Hoffmann, L. S. Bianchessi, D. Griebler. Npb-rust: Nas parallel benchmarks in rust, 2025.

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

[47] S. Nanz, F. Torshizi, M. Pedroni, B. Meyer. A comparative study of the usability of two object-oriented concurrent programming languages. *Information and Software Technology*, 55, 11 2010.

[48] S. Nanz, F. Torshizi, M. Pedroni, B. Meyer. Design of an empirical study for comparing the usability of concurrent programming languages. *Information and Software Technology*, 55(7):1304–1315, 2013.

[49] P. Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011.

[50] W. Paluszynski. Systemy Czasu Rzeczywistego i Sieci Komputerowe - Witold Paluszynski — kcir.pwr.edu.pl. `https://kcir.pwr.edu.pl/~witold/scrsk/#literatura`. [Dostęp 10-03-2025].

[51] 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.

[52] 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.

[53] M. P. Rooney, S. J. Matthews. Evaluating fft performance of the c and rust languages on raspberry pi platforms. *2023 57th Annual Conference on Information Sciences and Systems (CISS)*, strony 1–6, 2023.

[54] 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].

[55] T. Silva, J. Bispo, T. Carvalho. Foundations for a rust-like borrow checker for c. strona 155 – 165, 2024. Cited by: 0; All Open Access, Hybrid Gold Open Access.

[56] H. Team. Rust vs C++: A Quick Guide for Developers | Hostwinds — hostwinds.com. `https://www.hostwinds.com/blog/rust-vs-c-a-quick-guide-for-developers`. [Dostęp 13-03-2025].

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

[58] A. Tuby, A. Morrison. Reverse engineering the apple m1 conditional branch predictor for out-of-place spectre mistraining, 2025.

[59] 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.

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

[61] 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].

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

[63] X. Yin, Z. Huang, S. Kan, G. Shen. Safemd: Ownership-based safe memory deallocation for c programs. *Electronics*, 13(21), 2024.

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