

**Politechnika Wrocławskaw
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++**

Rafał Jasiński

Opiekun pracy
dr inż. Zdzisław Spławska

Słowa kluczowe: współbieżność, równoległość, Rust, C++, ARM, x86_64

Streszczenie

Praca jest poświęcona porównaniu mechanizmów programowania współbieżnego i równoległego w językach Rust i C++. Pierwsza część obejmuje teoretyczne podstawy programowania równoległego oraz przegląd literatury przedmiotu, identyfikując luki badawcze w porównaniach tych języków na różnych architekturach sprzętowych.

Kolejne rozdziały przedstawiają mechanizmy współbieżności w analizowanych językach, omawiając biblioteki Rust Rayon i Tokio oraz C++ OpenMP i Intel TBB. Metodologia badań obejmuje opis środowiska testowego na architekturach ARM64 (Apple Silicon M1) i x86_64 (Intel) z różnymi kompilatorami i systemami operacyjnymi.

Implementacja zawiera benchmarki NAS Parallel Benchmarks (CG, EP, IS) dla programowania równoległego oraz aplikacje testowe (producent-konsument, echo-serwer) dla programowania współbieżnego w obu językach. Analiza wyników obejmuje pomiary wydajności w MFLOPS, skalowanie względem liczby wątków, zużycie zasobów systemowych oraz efektywność komunikacji sieciowej.

Badania ujawniły systematyczną przewagę architektury ARM64 z współczynnikami przyspieszenia 1,55x-2,26x. Rust z Rayon osiągnął najwyższą wydajność w obliczeniach równoległych, Intel TBB dominowała w zadaniach komunikacyjnych, a model bezpieczeństwa pamięci Rust nie wprowadzał narzutów wydajnościowych. Głównym ograniczeniem była heterogeniczność środowisk testowych.

Praca dostarcza pierwszego kompleksowego porównania tych języków na dwóch architekturach oraz praktycznych rekomendacji wyboru technologii w zależności od charakterystyki problemu obliczeniowego.

Abstract

The thesis focuses on comparing concurrent and parallel programming mechanisms in Rust and C++ languages. The first part covers theoretical foundations of parallel programming and literature review, identifying research gaps in comparisons of these languages across different hardware architectures.

Subsequent chapters present concurrency mechanisms in analyzed languages, discussing Rust Rayon and Tokio libraries, and C++ OpenMP and Intel TBB. Research methodology encompasses testing environment description on ARM64 (Apple Silicon M1) and x86_64 (Intel) architectures with different compilers and operating systems.

Implementation includes NAS Parallel Benchmarks (CG, EP, IS) for parallel programming and test applications (producer-consumer, echo-server) for concurrent programming in both languages. Results analysis covers MFLOPS performance measurements, thread scaling, system resource consumption, and network communication efficiency.

Research revealed systematic ARM64 superiority with speedup factors of 1,55x-2,26x. Rust with Rayon achieved highest performance in parallel computations, Intel TBB dominated communication-intensive tasks, and Rust's memory safety model introduced no performance overhead. The main limitation was testing environment heterogeneity.

The work provides the first comprehensive comparison of these languages across two architectures and practical recommendations for technology selection based on computational problem characteristics.

Spis treści

1. Wstęp	9
1.1. Cel oraz zakres pracy	9
1.2. Problem badawczy	9
1.3. Struktura pracy	11
1.4. Słownik wybranych pojęć	12
2. Wprowadzenie	14
2.1. Programowanie współbieżne	15
2.1.1. Mechanizmy realizujące współbieżność	15
2.1.2. Zastosowania programowania współbieżnego	16
2.1.3. Zalety programowania współbieżnego	16
2.1.4. Wady programowania współbieżnego	16
2.2. Programowanie równoległe	17
2.2.1. Zasady programowania równoległego	17
2.2.2. Zastosowanie programowania równoległego	18
2.2.3. Zalety programowania równoległego	19
2.2.4. Wady programowania równoległego	19
2.2.5. Technologie i narzędzia do programowania równoległego	19
3. Przegląd literatury	20
3.0.1. Metodologia przeglądu literatury	20
3.0.2. Kryteria selekcji oraz wyłączenia	21
3.0.3. Baza Scopus	21
3.0.4. Baza Google Scholar	23
3.1. Porównanie Rust oraz C++	23
3.1.1. Bezpieczeństwo	24
3.1.2. Czas wykonania	24
3.1.3. Programowanie współbieżne oraz równoległe	25
3.2. Podsumowanie	26
3.2.1. Kierunki dalszych badań	30
4. Wybrane mechanizmy w języku Rust	31
4.1. Podejście do współbieżności i równoległości	31
4.1.1. Ownership oraz borrow	31
4.1.2. Nieustraszona współbieżność	32
4.1.3. Inteligentne wskaźniki (ang. <i>smart pointers</i>)	33
4.2. Programowanie współbieżne	35
4.2.1. Model własności pamięci	35
4.2.2. Biblioteki	35
4.2.3. Kanały	37
4.2.4. Asynchroniczność	40
4.3. Programowanie równoległe	41
4.3.1. Biblioteki	41

4.4.	Mechanizmy wspólne dla współbieżności i równoległości	42
4.4.1.	Wątki (std::thread)	42
4.4.2.	Synchronizacja dostępu (Mutex, RwLock)	43
4.4.3.	Wartości atomowe (Atomic)	44
4.4.4.	Bariery	45
5.	Wybrane mechanizmy w języku C++	46
5.1.	Programowanie współbieżne	46
5.1.1.	Biblioteki i przestrzeń standardowa	46
5.1.2.	Komunikacja między wątkami	47
5.1.3.	Synchronizacja	47
5.1.4.	Asynchroniczność	49
5.2.	Programowanie równoległe	51
5.2.1.	OpenMP - Open Multi-Processing	51
5.2.2.	Intel TBB (Threading Building Blocks)	51
6.	Założenia i metodologia porównania mechanizmów w językach Rust oraz C++ .	53
6.1.	Programowanie współbieżne	53
6.1.1.	Scenariusze testowe	53
6.1.2.	Parametry eksperymentalne	55
6.1.3.	Metryki wydajnościowe	56
6.1.4.	Środowisko eksperymentalne	56
6.1.5.	Procedura pomiarowa	57
6.1.6.	Narzędzia badawcze i automatyzacja	57
6.2.	Programowanie równoległe	57
6.2.1.	Wydajność obliczeniowa	57
6.2.2.	Wydajność sprzętowa (MFLOP/s)	58
6.2.3.	Zasoby systemowe	58
6.2.4.	Wybrane algorytmy do analizy	58
6.3.	Środowisko badań	59
6.3.1.	Infrastruktura sprzętowo-programowa	59
6.3.2.	Środowiska programistyczne	61
6.3.3.	Narzędzia pomiarowe i diagnostyczne	62
7.	Porównanie międzyjęzykowe - programowanie równoległe	63
7.1.	Implementacje w języku Rust	63
7.1.1.	Struktura i organizacja kodu	63
7.1.2.	Zarządzanie pamięcią	64
7.1.3.	Mechanizmy równoległości	64
7.1.4.	Specyfika benchmarku EP	65
7.1.5.	Specyfika benchmarku CG	65
7.1.6.	Specyfika benchmarku IS	66
7.2.	Implementacje w C++ (OpenMP)	66
7.2.1.	Struktura i organizacja kodu	66
7.2.2.	Zarządzanie pamięcią	67
7.2.3.	Mechanizmy równoległości	68
7.2.4.	Specyfika benchmarku EP	68
7.2.5.	Specyfika benchmarku CG	69
7.2.6.	Specyfika benchmarku IS	70
7.3.	Implementacje w C++ (TBB)	70
7.3.1.	Struktura i organizacja kodu	70
7.3.2.	Zarządzanie pamięcią	71

7.3.3. Mechanizmy równoległości	71
7.3.4. Specyfika poszczególnych benchmarków	72
7.4. Implementacje w C++ (nowoczesne podejście)	72
7.4.1. Struktura i organizacja kodu	72
7.4.2. Zarządzanie pamięcią	72
7.4.3. Mechanizmy równoległości	73
7.5. Architektura i organizacja kodu	73
7.6. Zarządzanie pamięcią	74
8. Porównanie między językowe - programowanie współbieżne	75
8.1. Porównanie między językowe	75
8.1.1. Struktura i organizacja kodu	75
8.1.2. Zarządzanie pamięcią	76
8.1.3. Mechanizmy współbieżności	77
8.1.4. Wydajność i bezpieczeństwo	82
8.1.5. Modele wykonania i zarządzanie zadaniami	82
8.1.6. Zarządzanie błędami i zasobami	83
8.1.7. Wnioski	83
9. Analiza wyników - programowanie równoległe	84
9.1. Benchmark CG	84
9.1.1. Wyniki benchmarków - platforma ARM64	84
9.1.2. Wyniki benchmarków - platforma x86_64	89
9.1.3. Wyniki profilowania wydajności - platforma ARM64	94
9.1.4. Wyniki profilowania wydajności - platforma x86_64	98
9.1.5. Porównanie pomiędzy platformami	99
9.2. Benchmark EP	102
9.2.1. Wyniki benchmarków - platforma ARM64	102
9.2.2. Wyniki benchmarków - platforma x86_64	107
9.2.3. Wyniki profilowania wydajności - platforma ARM64	111
9.2.4. Wyniki profilowania wydajności - platforma x86_64	113
9.2.5. Porównanie pomiędzy platformami	115
9.3. Benchmark IS	117
9.3.1. Wyniki benchmarków - platforma ARM64	117
9.3.2. Wyniki benchmarków - platforma x86_64	123
9.3.3. Wyniki profilowania wydajności - platforma ARM64	127
9.3.4. Wyniki profilowania wydajności - platforma x86_64	130
9.3.5. Porównanie pomiędzy platformami	132
10. Analiza wyników - programowanie współbieżne	135
10.1. Benchmark producent-klient	135
10.1.1. Wyniki benchmarków - platforma ARM64	135
10.1.2. Wyniki benchmarków - platforma x86_64	137
10.1.3. Porównanie pomiędzy platformami	140
10.2. Benchmark echo serwer	145
10.2.1. Wyniki benchmarków - platforma ARM64	145
10.2.2. Wyniki benchmarków - platforma x86_64	148
10.2.3. Porównanie pomiędzy platformami	150
11. Wnioski i rekomendacje	154
11.1. Omówienie wyników badań	154
11.1.1. Wydajność implementacji równoległych	154
11.1.2. Analiza mechanizmów programowania współbieżnego	155

11.2. Wpływ architektury i modeli pamięci na wyniki	155
11.2.1. Modele zarządzania pamięcią	155
11.2.2. Architektura sprzętowa	156
11.2.3. Wpływ środowiska kompilacyjnego i bibliotek	156
11.3. Rekomendacje praktyczne	157
11.3.1. Dobór technologii w zależności od charakterystyki problemu	157
11.3.2. Wybór platformy sprzętowej	157
11.3.3. Rekomendacje dotyczące środowiska kompilacyjnego	157
11.4. Ograniczenia metodologiczne badania	158
11.4.1. Heterogeniczność środowisk testowych	158
11.4.2. Ograniczenia narzędzi diagnostycznych	158
11.4.3. Ograniczenia doboru benchmarków	158
11.5. Kierunki dalszych badań	158
11.5.1. Ujednolicone środowisko badawcze	158
11.5.2. Rozszerzenie zakresu architektur i platform	158
11.6. Wkład pracy	159
11.7. Podsumowanie wniosków	160
12. Podsumowanie	161
Bibliografia	162
Spis rysunków	167
Spis tabel	170
Spis listingów	171

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 uwzględnia 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 [62, 61]. Programowanie współbieżne i równoległe stanowią obecnie podstawę projektowania wydajnych i niezawodnych aplikacji w wielu obszarach, od systemów operacyjnych, 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 [38]. 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, co potwierdzają zarówno badania naukowe, jak i praktyczne analizy [60, 74]. 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 szeroki wachlarz dojrzałych bibliotek równoległych (OpenMP, Intel TBB), jednak często kosztem większego ryzyka błędów programistycznych i bardziej złożonego zarządzania pamięcią.

Przegląd literatury pokazuje, że większość dotychczasowych badań porównawczych koncentrowała się na klasycznych architekturach x86_64, co skutkuje ograniczoną wiedzą na temat wpływu nowoczesnych architektur ARM64 na wydajność mechanizmów programowania współbieżnego i równoległego. Choć pojawiają się już prace dotyczące tego zagadnienia, wciąż brakuje szeroko zakrojonych, systematycznych analiz obejmujących różne architektury sprzętowe i zestawy benchmarkowe.

W kwestii porównania implementacji w języku Rust z dojrzałymi rozwiązaniami C++ przy użyciu uznanych zestawów benchmarkowych, takich jak NAS Parallel Benchmarks, literatura naukowa jest wciąż uboga. W ostatnim czasie pojawiły się jednak pierwsze publikacje, które próbują wypełnić tę lukę - przykładem jest praca [51], gdzie autorzy analizują wydajność Rust w kontekście benchmarków równoległych, jednak nie obejmuje ona kompleksowego porównania z rozwiązaniami C++ na różnych platformach sprzętowych.

W efekcie, choć pojawiają się już pojedyncze próby porównania Rust i C++ w kontekście benchmarków równoległych, wciąż brakuje systematycznych, szeroko zakrojonych badań, które uwzględniałyby zarówno różne architektury (w tym ARM64), jak i różne zestawy benchmarków, a także zapewniałyby porównanie wydajności i niezawodności obu języków w praktycznych zastosowaniach.

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

PB1: *Jakie są różnice w wydajności i charakterystykach skalowania mechanizmów programowania równoległego między językami Rust (Rayon) a C++ (OpenMP, Intel TBB) na architekturach ARM64 i x86_64, mierzone przy użyciu standardowych benchmarków NAS Parallel Benchmarks (CG, EP, IS)?*

PB2: *W jaki sposób wybór konkretnego języka i biblioteki wpływa na wydajność aplikacji współbieżnych pod względem przepustowości, zużycia pamięci oraz stabilności działania w różnych środowiskach komplikacji?*

PB3: *Jaki jest rzeczywisty narzut wydajnościowy związany z modelem bezpieczeństwa pamięci w języku Rust w porównaniu z mechanizmami zarządzania zasobami dostępnymi w C++, i czy różnice te są zależne od architektury sprzętowej?*

Odpowiedź na te pytania zostanie udzielona poprzez systematyczne eksperymentalne porównanie konkretnych implementacji w obu językach, z wykorzystaniem standardowych metodologii pomiarowych oraz analizy statystycznej wyników uzyskanych na dwóch reprezentatywnych architekturach sprzętowych. Badania obejmą zarówno benchmarki obliczeniowe wysokiej wydajności, jak i praktyczne aplikacje reprezentujące typowe scenariusze współbieżności, co pozwoli na sformułowanie rekomendacji dotyczących wyboru odpowiednich narzędzi w zależności od specyfiki projektu i środowiska docelowego.

Wybór tematu pracy został dodatkowo umotywowany aktualnymi inicjatywami w zakresie bezpieczeństwa cybernetycznego oraz osobistymi doświadczeniami zawodowymi autora. W grudniu 2023 roku Agencja Bezpieczeństwa Cybernetycznego i Infrastruktury (CISA), we współpracy z Agencją Bezpieczeństwa Narodowego (NSA), FBI oraz międzynarodowymi organizami ds. cyberbezpieczeństwa z Australią, Kanadą, Nowej Zelandii i Wielkiej Brytanii, opubliko-

wała wspólny przewodnik "The Case for Memory Safe Roadmaps" w ramach kampanii "Secure by Design" [68]. Dokument ten zdecydowanie zachęca kierownictwo producentów oprogramowania do priorytetowego traktowania języków programowania bezpiecznych pod względem pamięci, tworzenia i publikowania map drogowych bezpieczeństwa pamięci oraz wdrażania zmian w celu wyeliminowania tej klasy podatności.

Agencje te podkreślają, że błędy bezpieczeństwa pamięci stanowią najczęstszy typ ujawnianych podatności oprogramowania, przy czym Microsoft potwierdził, że około 70% jego błędów (CVE) to podatności związane z bezpieczeństwem pamięci, a Google poświadczyl podobną cyfrę dla projektu Chromium. W tym kontekście agencje zalecają organizacjom odejście od C/C++, ponieważ nawet przy szkoleniach w zakresie bezpieczeństwa (i ciągłych wysiłkach na rzecz wzmacnienia kodu C/C++), programiści nadal popełniają błędy.

Dodatkowo, doświadczenia zawodowe autora w obszarze programowania równoleglego wskazują na praktyczną potrzebę obiektywnej oceny narzędzi dostępnych w obu językach. W środowisku przemysłowym coraz częściej pojawia się pytanie o zasadność migracji z dojrzaliych rozwiązań C++ w kierunku nowszych technologii oferowanych przez Rust, szczególnie w kontekście aplikacji krytycznych pod względem wydajności i bezpieczeństwa.

1.3. Struktura pracy

Niniejsza praca została zorganizowana w sposób umożliwiający systematyczne przedstawienie zagadnienia oraz przeprowadzenie kompleksowej analizy porównawczej mechanizmów programowania współbieżnego i równoległego w językach Rust i C++. Rozdział pierwszy przedstawia cel oraz zakres pracy, definiuje główne problemy badawcze i określa metodologię badań. Rozdział drugi wprowadza czytelnika w podstawy programowania współbieżnego i równoległego, omawiając kluczowe różnice między tymi paradygmatami oraz ich zastosowania w kontekście współczesnych systemów wielordzeniowych. Trzeci rozdział zawiera przegląd literatury przedmiotu, analizując dotychczasowe badania porównawcze języków programowania w kontekście mechanizmów wielowątkowości oraz identyfikując luki badawcze, które niniejsza praca ma wypełnić. Rozdziały czwarty i piąty skupiają się na teoretycznych aspektach mechanizmów programowania współbieżnego i równoległego w analizowanych językach, omawiając dostępne biblioteki, frameworki oraz narzędzia zarządzania wątkami i pamięcią. Rozdział szósty przedstawia metodologię badań oraz procedury pomiarowe wykorzystane w eksperymentach, w tym opis środowiska sprzętowo-programowego, konfiguracji kompilatorów oraz protokołu eksperymentalnego dla benchmarków NPB i aplikacji testowych. Rozdział siódmy zawiera opis i implementację benchmarków NPB (Conjugate Gradient, Embarrassingly Parallel, Integer Sort) dla programowania równoległego w językach Rust i C++. Rozdział ósmy przedstawia szczegółowy opis implementacji aplikacji testowych dla programowania współbieżnego (producent-konsument, echo-serwer), koncentrując się na specyfice programistycznej oraz architektonicznych rozwiązaniach zastosowanych w obu językach. Rozdział dziewiąty prezentuje wyniki eksperymentów dotyczących programowania równoległego oraz ich szczegółową analizę, koncentrując się na porównaniu wydajności benchmarków NPB, wzorcach skalowania oraz wykorzystaniu zasobów systemowych przez implementacje na architekturach ARM64 i x86_64. Rozdział dziesiąty zawiera analizę wyników eksperymentów dotyczących programowania współbieżnego, przedstawiając porównanie wydajności aplikacji testowych oraz charakterystyk behawioralnych implementacji w obu językach. Rozdział jedenasty prezentuje wnioski oraz rekommendacje dotyczące praktycznego zastosowania języków Rust i C++ w projektach wymagających wysokiej wydajności i bezpieczeństwa współbieżnego, identyfikując scenariusze optymalnego wykorzystania każdego z języków. Rozdział dwunasty podsumowuje najważniejsze osiągnięcia badawcze oraz wskazuje możliwe kierunki dalszych analiz i rozwiązań zaproponowanych rozwiązań.

Na samym końcu pracy znajduje się literatura, spis wykorzystanych rysunków, tabel oraz listingu kodu.

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 naraz, eliminując ryzyko wyścigów danych i błędów synchronizacji.
- **odwołania do nieobecnych stron** - (ang. *page fault*) - zdarzenie w systemie operacyjnym, które występuje, gdy program próbuje uzyskać dostęp do strony pamięci, która nie znajduje się obecnie w pamięci RAM. Może to skutkować koniecznością załadowania tej strony z dysku (np. z pliku wymiany), co wpływa na wydajność programu.
- **zwolnienie stron pamięci** - (ang. *page reclaims*) - operacje systemowe polegające na odzyskiwaniu już załadowanych, ale nieaktywnych stron pamięci, aby umożliwić ich ponowne wykorzystanie przez inne procesy. Pomaga to zoptymalizować wykorzystanie pamięci fizycznej bez konieczności natychmiastowego odwoływanego się do pamięci wirtualnej.
- **pożyczanie** (ang. *borrow*) - również występujący pod inną nazwą jako przenoszenie własności [23], 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 równolegle** - to sposób wykonywania wielu zadań jednocześnie, co zwiększa wydajność programu. W odróżnieniu od programowania współbieżnego, programowanie równolegle 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

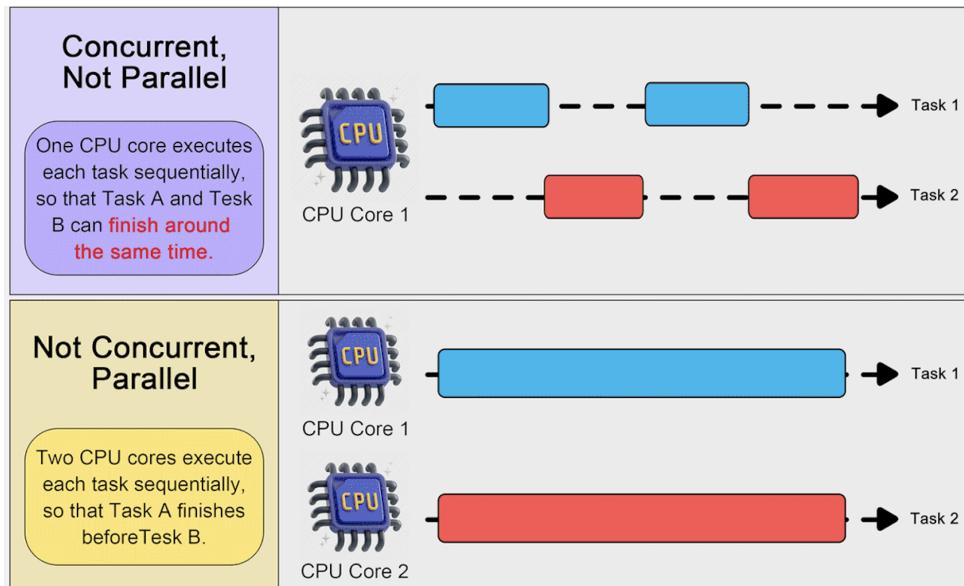
Rozdział ten został poświęcony przybliżeniu czytelnikowi zasad działania programowania współbieżnego oraz równoległego, a także omówieniu tych mechanizmów w kontekście języków Rust i C++.

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 [24, 72, 27, 71].

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. Chociaż często używa się ich zamiennie, ich cechy i cele różnią się znacznie. Poglądowa różnica między tymi paradygmatami została przedstawiona na rysunku 2.1.

Górna część ilustracji przedstawia model programowania współbieżnego, w którym pojedynczy rdzeń procesora (CPU Core 1) wykonuje naprzemiennie fragmenty dwóch zadań (Task 1 i Task 2), reprezentowanych odpowiednio przez niebieskie i czerwone prostokąty. Przerywane linie między segmentami zadań symbolizują przełączanie kontekstu wykonania. Jak wskazuje towarzyszący opis, w tym podejściu jeden rdzeń CPU sekwencyjnie obsługuje oba zadania, umożliwiając ich niemal równoczesne zakończenie poprzez wielokrotne przełączanie się między nimi.

Dolna część ilustracji prezentuje model programowania równoległego, charakteryzujący się wykorzystaniem dwóch fizycznie odrębnych rdzeni procesora (CPU Core 1 i CPU Core 2). Każdy rdzeń nieprzerwanie realizuje jedno dedykowane zadanie w całości - pierwszy rdzeń wykonuje Task 1 (niebieski prostokąt), podczas gdy drugi rdzeń wykonuje Task 2 (czerwony prostokąt). Zadania są przetwarzane równocześnie, ale niezależnie od siebie, bez współdzielienia czasu procesora.



Rys. 2.1: Różnice między wykonywaniem zadań współbieżnie a równolegle [26]

2.1. Programowanie współbieżne

Programowanie współbieżne stanowi paradygmat projektowania aplikacji umożliwiający wykonywanie wielu zadań w sposób pozorne jednoczesny, przy czym procesor fizycznie realizuje jedynie jedno zadanie w danej chwili. Mechanizm ten opiera się na koncepcji quasi-równoległości [56], w której jednostka obliczeniowa przełącza się między zadaniami w tak krótkich interwałach czasowych, że powstaje wrażenie ich jednoczesnego wykonywania. Takie podejście nabiera szczególnego znaczenia w kontekście aplikacji interaktywnych - systemów gier komputerowych, aplikacji mobilnych, czy serwisów internetowych - gdzie konieczność natychmiastowej reakcji na zdarzenia zewnętrzne (żądania użytkowników, komunikaty sieciowe, zdarzenia interfejsu użytkownika) stanowi podstawowy wymóg funkcjonalny [36].

Zastosowanie współbieżności znajduje uzasadnienie w aplikacjach wymagających zarządzania wieloma operacjami niekoniecznie o charakterze obliczeniowo intensywnym. Systemy obsługi żądań serwerowych, aplikacje multimedialne oraz interfejsy graficzne stanowią reprezentatywne przykłady dziedzin, w których efektywne wykorzystanie tego paradygmatu przekłada się bezpośrednio na jakość doświadczenia użytkownika [24].

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 kolej" w bardzo krótkich cyklach [72, 25].

Kolejnym 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 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 [72].

2.1.2. Zastosowania programowania współbieżnego

Programowanie współbieżne znajduje szerokie zastosowanie w aplikacjach, które muszą reagować na różne wydarzenia użytkownika lub zewnętrzne zdarzenia w czasie rzeczywistym. Typowe zastosowania programowania współbieżnego obejmują [65, 72]:

- Aplikacje interaktywne - gry komputerowe, aplikacje mobilne oraz desktopowe, które muszą zapewniać natychmiastową reakcję na działania użytkownika, włączając obsługę gestów, kliknięć oraz poleceń wprowadzanych 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 korzyści wynikające ze stosowania programowania współbieżnego w aplikacjach obejmują [12, 72]:

- 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.
- Skalowanie - 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 [12, 66]:

- 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.
- Trudna diagnostyka błędów - aplikacje współbieżne są trudniejsze w procesie znajdowania i eliminowania błędów w kodzie źródłowym, 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 rzeczywiste jednoczesne wykonywanie wielu zadań poprzez wykorzystanie wielu jednostek obliczeniowych. W odróżnieniu od programowania współbieżnego, gdzie zadania są wykonywane sekwencyjnie z przełączaniem kontekstu, podejście równoległe wykorzystuje fizyczną architekturę wielordzeniową procesorów lub specjalizowane jednostki przetwarzające do osiągnięcia znaczących przyspieszeń obliczeniowych.

Programowanie równoległe znajduje zastosowanie w aplikacjach wymagających znacznej mocy obliczeniowej: obliczeniach z zakresu uczenia maszynowego, simulacjach naukowych, przetwarzaniu dużych zbiorów danych, renderowaniu grafiki oraz aplikacjach o wysokich wymaganiach wydajnościowych. Wykorzystanie tej techniki umożliwia redukcję czasu wykonywania obliczeń, które w modelu sekwencyjnym wymagałyby nieakceptowalnie długich okresów realizacji [36, 15].

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 [15].

Modele pamięci

W kontekście programowania równoległego, istnieje szereg modeli pamięci, które definiują metody przechowywania oraz dostępu do danych przez jednostki przetwarzające [15]:

- 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 Flynna

Taksonomia Flynnna, stanowi klasyczny system kategoryzacji architektur komputerowych pod kątem stopnia i rodzaju równoległości przetwarzania. Jej głównym założeniem jest klasyfikacja systemów obliczeniowych na podstawie liczby równoczesnie przetwarzanych strumieni instrukcji oraz danych. Pomimo upływu lat i rozwoju nowych modeli obliczeń, taksonomia ta pozostaje istotnym punktem odniesienia w analizie i projektowaniu współczesnych systemów równoległych [34].

Wyróżnia się cztery podstawowe klasy architektur:

- SISD (Single Instruction, Single Data) - pojedyncza instrukcja i pojedynczy strumień danych; jest to klasyczny model sekwencyjny, charakterystyczny dla tradycyjnych komputerów jedno-procesorowych.

2. Wprowadzenie

- SIMD (Single Instruction, Multiple Data) - pojedyncza instrukcja operująca równocześnie na wielu strumieniach danych; architektura ta znajduje zastosowanie m.in. w procesorach wektorowych i systemach obliczeń równoległych przetwarzających duże zbiory danych.
- MISD (Multiple Instruction, Single Data) - wiele niezależnych instrukcji wykonywanych na wspólnym zbiorze danych; model ten ma ograniczone zastosowanie praktyczne i jest głównie konstruktorem teoretycznym.
- MIMD (Multiple Instruction, Multiple Data) - wiele instrukcji przetwarzających równolegle niezależne strumienie danych; reprezentuje najbardziej zaawansowaną i powszechną klasę systemów równoległych, do której należą współczesne architektury wieloprocesorowe i klasy obliczeniowe.

Zrównoleglenia

Zrównoleglenie (ang. *parallelization*) stanowi fundamentalny proces w programowaniu równoległym, polegający na identyfikacji i dekompozycji zadań obliczeniowych w sposób umożliwiający ich jednoczesne wykonanie. W kontekście architektury komputerowej i projektowania algorytmów wyróżnia się kilka podstawowych strategii zrównoleglania:

- Zrównoleglenie danych - polega na podziale zbioru danych wejściowych na mniejsze fragmenty, które są przetwarzane jednocześnie przez wiele jednostek obliczeniowych wykonujących te same operacje. Szczególnie efektywne w przypadku operacji na dużych strukturach danych, takich jak macierze czy tablice.
- Zrównoleglenie zadań - opiera się na dekompozycji problemu na niezależne zadania, które mogą być wykonywane równocześnie przez różne jednostki obliczeniowe. Zadania te często realizują odmienne operacje i mogą operować na różnych fragmentach danych.
- Zrównoleglenie potokowe - polega na podziale sekwencyjnego procesu na etapy, które mogą być wykonywane jednocześnie dla różnych elementów danych, tworząc strukturę przypominającą linię produkcyjną.

Efektywność zrównoleglania jest często ograniczona przez prawo Amdahla [1], które określa maksymalne teoretyczne przyspieszenie możliwe do uzyskania poprzez równoległe wykonanie części algorytmu, przy założeniu, że pozostała część musi być wykonana sekwencyjnie.

2.2.2. Zastosowanie programowania równoległego

Programowanie równoległe znajduje zastosowanie w różnych dziedzinach wymagających znacznej mocy obliczeniowej oraz szybkiego przetwarzania dużych zbiorów danych. Najważniejsze obszary wykorzystania obejmują [55]:

- 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 rozdzielenie zadań przetwarzania danych na wiele maszyn.
- Renderowanie 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 jego elementów.

2.2.3. Zalety programowania równoległego

Poprzez wykorzystanie programowania równoległego można się spodziewać następujących korzyści [55]:

- 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.
- Saklowanie - 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 [55, 15]:

- 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, 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.

2.2.5. 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 [71, 27],
- CUDA - platforma stworzona przez firmę NVIDIA, przeznaczona do programowania na procesorach graficznych (GPU), wykorzystywana głównie w zastosowaniach związanych z uczeniem maszynowym i obróbką grafiki [15],
- 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 umożliwia zrozumienie aktualnego stanu wiedzy w tej dziedzinie, a także wskazanie na występujące luki badawcze, które niniejsza praca postara się wypełnić. Na samym wstępnie 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 [11]. 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 nie spełniające 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

Tabela 3.1: Kwerendy użyte w bazie Scopus¹

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

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

3. Przegląd literatury

te zagadnienia. Drugie zapytanie zawęzało 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 tytułu 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 poszczególnych 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.

Tabela 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 niezwiązańnych 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

Tabela 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 ilość 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.

Tabela 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 niezwiązańnych 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 komplikacji, bezpieczeństwo, wydajność oraz obsługa współprzeźności i równoległości. Choć istnieje szeroka literatura porównawcza, wciąż stosunkowo niewielkie prace skupią się w sposób bezpośredni i systematyczny

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

na porównaniu mechanizmów współbieżnego i równoległego przetwarzania w tych dwóch językach. W ostatnich latach pojawiły się jednak wartościowe opracowania, również te akademickie, które podejmują to zagadnienie. Pomimo to, 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 dereferencja pustego wskaźnika, wyścigi danych oraz wycieki pamięci. Mechanizmy takie jak sprawdzanie własności i pożyczki oraz jawna mutowalność zmiennych (ang. *explicit mutability*) są wymieniane jako kluczowe elementy zapewniające bezpieczeństwo oraz minimalizując ryzyko wycieków pamięci [46].

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 [37, 33] często podkreśla się, że to właśnie większa złożoność i ryzyko błędów w kodzie C++ skłoniły społeczność do stworzenia języków takich jak Rust.

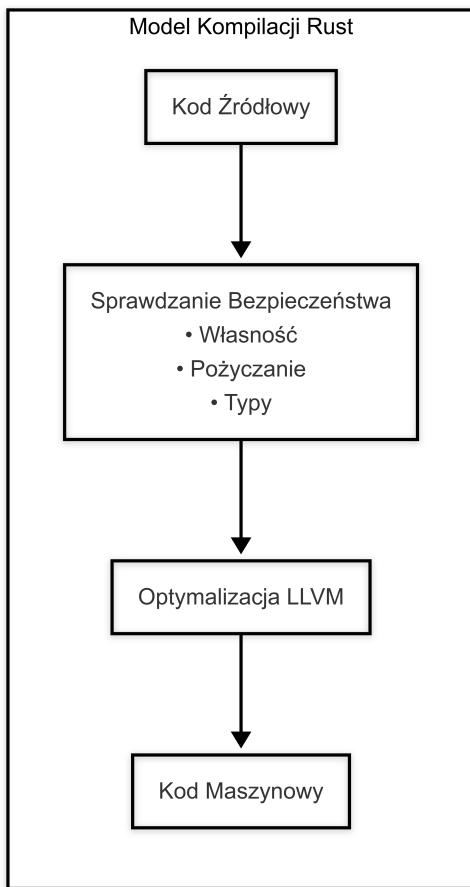
Przykładowo, badania [35, 57, 42] wskazują, że aplikacje napisane w Rust są mniej podatne na błędy związane z wyścigami danych (ang. *data races*), co ma szczególne znaczenie w środowiskach wielowątkowych. Z kolei w C++ stosowanie bibliotek takich jak `std::thread` czy frameworków typu OpenMP pozwala na osiągnięcie podobnych celów, choć wymaga od programistów większej uwagi w zakresie synchronizacji. Dodatkowo są również prace [73, 64], które przedstawiają próby implementacji mechanizmów wbudowanych w język Rust (prawo własności, pożyczka) do języka C.

3.1.2. Czas wykonania

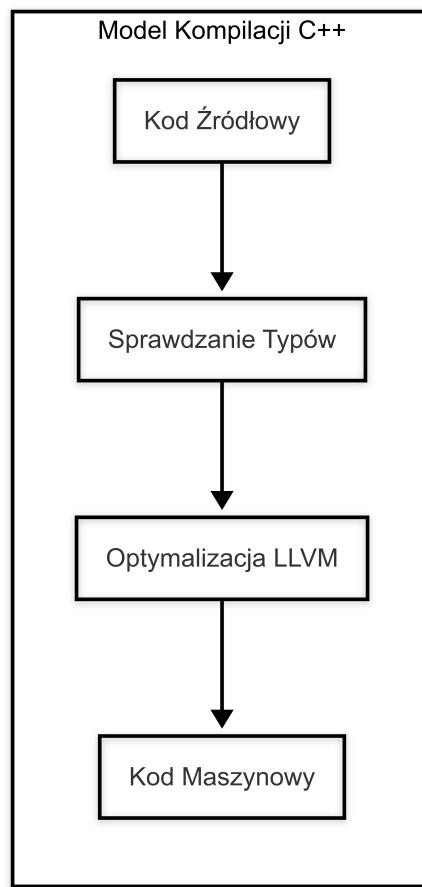
Porównania czasów wykonania programów napisanych w Rust i C++ są częstym tematem analiz [70, 45, 30, 48]. W badaniach tych zostało wykazane, że pod względem wydajności Rust jest konkurencyjny wobec C++, co wynika z mechanizmów komplikacji i optymalizacji kodu.

Jednak kluczową różnicą jest to, że Rust wprowadza pewne narzuty związane z kontrolą bezpieczeństwa w czasie komplikacji, 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 [47] często podkreśla się, że Rust, w odróżnieniu od C++, kładzie większy nacisk na bezpieczeństwo pamięci oraz typów w czasie komplikacji, 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 komplikacji w języku Rust



Rys. 3.2: Kroki komplikacji w języku C++

Informacje o procesie komplikacji pochodzą z [47, 23, 43], które opisują integrację z LLVM i różnice w sprawdzaniu bezpieczeństwa. Na rysunku 3.1 drugi blok reprezentuje dodatkowe etapy sprawdzania bezpieczeństwa w Rust, które nie występują w C++. Z kolei na rysunku 3.2 drugi blok pokazuje podstawowe sprawdzanie typów w C++, które jest mniej rygorystyczne niż system Rust. Ze względu na różnicę w procesie komplikacji 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 wyściegi danych w czasie komplikacji. Narzędzia takie jak Tokio i Rayon pozwalają na łatwe tworzenie i zarządzanie zadaniami asynchronicznymi i równoległymi [66, 40]. C++ z kolei oferuje wsparcie dla wielowątkowości poprzez standartową 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.

3. Przegląd literatury

Strona [49] 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 [49] 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 międzymiędzy pamięciowej.

W pracy akademickiej Brandefelta i Heymana [39] 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 systemowemu własności (ang. *ownership*), eliminuje całe klasy błędów związanych z zarządzaniem pamięcią, bez konieczności stosowania odśmiecania.

Kolejne istotne opracowanie przedstawiono w publikacji Martinsa et al. [51], gdzie zaprezentowano implementację zestawu NAS (NASA Advanced Supercomputing) Parallel Benchmarks (NPB) [9] 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 Fortranu (w wersjach OpenMP i sekuracyjnych), co wskazuje na jego rosnący potencjał w dziedzinie wysokowydajnych obliczeń (HPC - High-Performance Computing). Jednocześnie zwróciła 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 [21] 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 szkielety (ang. *skeletons*) i wzorce projektowe, 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 szczególny znaczenie w kontekście projektów systemowych.

Można również znaleźć pracę, która przedstawia wykorzystanie biblioteki odpowiedzialnej za współbieżność FastFlow przez oba języki Rust oraz C++ [58]. 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 współprzeźności i równoległości można zauważać 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 [18, 19, 3, 8] 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 zrozumiałosć kodu.
- Czytelność kodu, związana z jego późniejszą konserwacją i rozwijaniem.
- Dostosowanie języka do konkretnego zastosowania, co wpływa na wydajność i efektywność programów.
- Szybkość komplikacji.
- 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 komplikacji oraz wykrywaniem zagrożeń w trakcie działania programu.
- Długowieczność języka oraz narzędzi komplikacyjnych, 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, utrzymywialności 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++, zauważać 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 głębszej 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

3. Przegląd literatury

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. Jednakże 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* [31] 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 zafalszowywać 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” [51], 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 typizacji oraz zarządzanie zasobami systemowymi. Przykładem mogą być prace [39], które zestawiają Rust i C++ w kontekście praktycznych implementacji aplikacji wielowątkowych, czy badania [51], 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 współpracujących równoległość (np. PPL, Rayon, OpenMP).

Pomimo to, 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 sprawdzania pożyczeń. 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 diagnostyki błędów 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 zmennego 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 komplikacji - 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 mogłyby 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 ewoluje 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` [10] dążą do umożliwienia pisania programów cieniących w języku Rust, oferując podejście, które integruje możliwości GPU z bezpiecznymi konstrukcjami

języka. Dokumentacja dostępna na stronie [github](#) [10] wskazuje na intensywne prace nad przenesieniem wielu korzyści wynikających z systemu własności i typizacji 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 [13]) 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 [5]. 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` [14] 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 a Rayon, czy też mechanizmy asynchroniczne (`async/await` w Rust a `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

Niniejszy rozdział koncentruje się na analizie wybranych mechanizmów oferowanych przez język programowania Rust w kontekście współbieżności i równoległości. Omówione zostaną fundamentalne koncepcje, które stanowią o unikalności tego języka. Rozdział zawiera również analizę mechanizmów synchronizacyjnych oraz komunikacji międzywątkowej. Zawarte przykłady kodu ilustrują praktyczne zastosowania omawianych koncepcji i stanowią punkt odniesienia dla dalszych rozważań porównawczych.

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 [23]:

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 komplikacji, 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. *roped*).
- 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("HelloWorld");
    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 pożyczanie 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 komplikacji 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 [74].

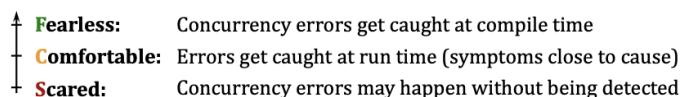


Figure 2: A spectrum of fear in parallel programming.

Rys. 4.1: Spektrum nieustraszzonej współbieżności [16]

Autorzy [16] uważają, że nieustraszona współbieżność jest lepiej interpretowana jako spektrum, zilustrowane na 4.1 idealnie eliminując wszelkie obawy przed błędami współbieżności w czasie komplikacji (**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**).

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 równoległy, wspierając kontrolowaną oraz bezpieczną wymianę danych. Poniżej omówione zostaną trzy najczęściej używane inteligentne wskaźniki w Rust: Box, Rc oraz Arc.

Box

Inteligentny wskaźnik 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 komplikacji.

Wykorzystuje się go do przechowywania dużych struktur danych, obsługi typów rekurencyjnych, których rozmiar nie może być określony w czasie komplikacji [43].

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

Inteligentny wskaźnik 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 [43].

Listing 4.3: Inteligentny wskaźnik RC

```
use std::rc::Rc;
use std::cell::Cell;

fn main() {
    let a = Rc::new(Cell::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 [43].

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 współdzielienia

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

    handle.join().unwrap(); // Oczekивание на 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 mutualności. 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 [20, 43] :

- Dereferencję wskaźników surowych (ang. *raw pointers*),
- Wywołanie funkcji lub interfejsów oznaczonych jako `unsafe` (np. z FFI - (ang. *Foreign Function Interface*), czyli interfejs do wywoływanego 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) [20].

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. 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 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 oddzielnego 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 komunikatów. 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 języka Rust 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 — nadawca jest odpowiedzialny za wysyłanie danych, a odbiorca 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 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 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};
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 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 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.
- Futura - abstrakcja reprezentująca wartość obliczaną asynchronicznie, której rezultat będzie dostępny w przyszłości. Obliczenia tego typu są inicjowane leniwie i wymagają uruchomienia w ramach środowiska wykonawczego.
- Tokio i `async-std` - najpopularniejsze asynchroniczne runtimes, implementujące własne plany i pętle zdarzeń.
- 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 równoczesne 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 wyściogów 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 podwijamy 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 podwijania 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 równolegle 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 częściowe 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 współpracy 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 wewnętrz sekcji krytycznej, co zapobiega wyścigom danych. 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 += 1;
            println!("Zapis danych: {:?}", write);
        })
    };
}
```

```
        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 zsynchronizowanie 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 barierce, 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 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 przejęciem 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 futurę 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 powiązanego
    ↪ 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
                           ↪ ci, blokuje do czasu jej ustawienia
producer.join();

return 0;
}
```

W listingu 5.6 wątek główny tworzy promesę i pobiera powiązaną z nią futurę. Wątek producenta wykonuje obliczenie i przekazuje wynik przez promesę. 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 powiązanego
                                                   ↪ 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 futury, 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 [6] 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 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 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. Scenariusze testowe

Przeprowadzone badania obejmują trzy główne scenariusze testowe, które umożliwiają kompleksową ocenę mechanizmów programowania współbieżnego w różnych kontekstach zastosowań obliczeniowych.

Wzorzec producent-konsument

Implementacja w języku Rust z wykorzystaniem `mpsc::channel`

Badanie wykorzystuje standardowy jednokierunkowy kanał komunikacyjny z biblioteki standardej języka Rust:

- `mpsc::Sender<String>` dla wątków producenckich
- `Arc<Mutex<mpsc::Receiver<String>>` dla wątków konsumenckich
- Synchronizacja przez `Arc<AtomicBool>` dla sygnalizacji zakończenia wykonania
- Operacje bezblokowe `try_recv()` z mechanizmem fallback na `sleep`

Implementacja w języku Rust z wykorzystaniem ThreadSafeQueue

Alternatywna implementacja z wykorzystaniem autorskiej struktury danych:

- Arc<Mutex<VecDeque<T>> jako podstawowy kontener danych
- try_pop() i push() jako fundamentalne operacje
- Mechanizm odpytywania (ang. *polling*) z try_lock() dla unikania blokowania wątków

Implementacja w języku C++ z wykorzystaniem Channel<T>

Autorska implementacja wzorowana na kanałach komunikacyjnych:

```
template<typename T>
class Channel {
private:
    std::queue<T> queue_;
    mutable std::mutex mutex_;
    std::condition_variable condition_;
    bool closed_ = false;
public:
    void send(T item);
    bool recv(T& item);
    bool try_recv(T& item);
};
```

Implementacja w języku Rust z wzorcem Arc<Mutex<T>>

Badanie współdzielonych struktur danych z kontrolowaną konkurencją dostępu:

- Arc<Mutex<i64>> dla licznika atomowego
- Arc<Mutex<Vec<i32>>> dla współdzielonej kolekcji danych
- Operacje haszowania dla symulacji obciążenia procesora
- Pomiar czasu blokowania muteksów z precyzją do nanosekund

Implementacja w języku C++ z wzorcem std::mutex

Odpowiadająca implementacja w języku C++:

- std::mutex z mechanizmem std::lock_guard
- std::atomic<std::int64_t> dla licznika atomowego
- std::vector<std::int32_t> z wstępna alokacją pamięci
- Pomiar narzutu synchronizacji z wykorzystaniem std::chrono

Echo serwer - operacje sieciowe wejścia/wyjścia

Implementacja w języku Rust z modelem asynchronicznym

Implementacja wykorzystująca model programowania asynchronicznego async/await:

- tokio::net::TcpListener dla akceptacji połączeń sieciowych
- tokio::spawn dla tworzenia zadań asynchronicznych
- tokio::select! dla multipleksowania operacji wejścia/wyjścia
- tokio::sync::Semaphore dla ograniczania liczby połączeń
- Zielone wątki (ang. *green threads*) zarządzane przez środowisko wykonawcze Tokio

Implementacja w języku C++ z modelem wątkowym

Tradycyjny model jeden-wątek-na-połączenie:

- `std::thread` dla każdego połączenia sieciowego
- Gniazda sieciowe POSIX z funkcjami `accept()`, `recv()`, `send()`
- `std::atomic<size_t>` dla ograniczania liczby połączeń
- Obsługa limitów czasu z opcją `S0_RCVTIMEO`

Implementacja w języku C++ z modelem asynchronicznym

Model sterowany zdarzeniami z wykorzystaniem systemowych interfejsów API:

- Linux: `epoll` z flagami `EPOLLIN/EPOLLOUT`
- macOS: `kqueue` z filtrem `EVFILT_READ`
- Gniazda nieblokujące z flagą `O_NONBLOCK`
- Jednowątkowa pętla zdarzeń z wykorzystaniem `std::async`

6.1.2. Parametry eksperymentalne

Konfiguracja eksperymentów producent-konsument

Tabela 6.1: Parametry eksperymentów producent-konsument

Parametr	Wartości
Wątki (producenci=konsumenci)	wątki_testowe ≤ 8
Elementy na producenta	-items konfigurowalne
Rozmiar komunikatów	format! ("Producer-{i}-Item-{j}") (20-30B)
Tryby działania	-mode channel lub -mode queue

Konfiguracja eksperymentów serwera echa

Tabela 6.2: Parametry eksperymentów serwera echa

Parametr	Wartości
Liczba klientów	-num-clients
Komunikaty na klienta	-messages-per-client
Rozmiar komunikatów	-message-size-kb
Maksymalne połączenia	-max-connections
Wątki robocze	-num-threads
Model implementacji	C++: flaga -async

Rzeczywiste przypadki testowe

Scenariusze testowe dla wzorca producent-konsument:

```
przypadki_testowe = [
    {"wątki": 1, "elementy": 1000, "opis": "Małe obciążenie - 1 wątek"},
    {"wątki": 2, "elementy": 2000, "opis": "Małe obciążenie - 2 wątki"},
    {"wątki": 4, "elementy": 2000, "opis": "Małe obciążenie - 4 wątki"},
    {"wątki": 4, "elementy": 5000, "opis": "Średnie obciążenie - 4 wątki"},
    {"wątki": 6, "elementy": 5000, "opis": "Średnie obciążenie - 6 wątków"},
```

```
{"wątki": 8, "elementy": 5000, "opis": "Średnie obciążenie - 8 wątków"},  
 {"wątki": 8, "elementy": 10000, "opis": "Duże obciążenie - 8 wątków"}  
]  
tryby = ["channel", "queue"]
```

Scenariusze testowe dla operacji sieciowych wejścia/wyjścia:

```
scenariusze = [  
    {"klienci": 10, "komunikaty": 50, "połączenia": 100, "rozmiar_kb": 0},  
    {"klienci": 25, "komunikaty": 100, "połączenia": 200, "rozmiar_kb": 1},  
    {"klienci": 50, "komunikaty": 100, "połączenia": 500, "rozmiar_kb": 0},  
    {"klienci": 100, "komunikaty": 50, "połączenia": 500, "rozmiar_kb": 4},  
    {"klienci": 200, "komunikaty": 25, "połączenia": 1000, "rozmiar_kb": 0},  
    {"klienci": 150, "komunikaty": 100, "połączenia": 1000, "rozmiar_kb": 8},  
    {"klienci": 500, "komunikaty": 10, "połączenia": 2000, "rozmiar_kb": 0},  
    {"klienci": 100, "komunikaty": 200, "połączenia": 1000, "rozmiar_kb": 16}  
]  
implementacje = [("rust", "async"), ("cpp", "threaded"), ("cpp", "async")]
```

6.1.3. Metryki wydajnościowe

Metryki czasowe

- Czas wykonania - całkowity czas trwania eksperymentu wyrażony w sekundach.
- Przepustowość - liczba operacji lub komunikatów przetworzonych na sekundę.
- Opóźnienia synchronizacji muteksów - średni czas blokowania wyrażony w mikrosekundach.
- Czas tworzenia zadań asynchronicznych (Rust) - czasinicjalizacji zadania asynchronicznego.
- Narzut przełączania kontekstu (C++) - koszt czasowy przełączania między wątkami.

Metryki wykorzystania zasobów systemowych

- Pamięć RSS (ang. *Resident Set Size*) - rzeczywiste zużycie pamięci fizycznej wybrane w kilobajtach.
- Estymacja pamięci sterty - szacowana pamięć sterty na strukturę danych:
 - Bufory `mpsc::channel`,
 - Pojemność a rozmiar `std::vector`,
 - Wewnętrzne przechowywanie `VecDeque`,
 - Narzut stosu wątków (2048KB + metadane na wątek).
- Szczytowe zużycie pamięci - maksymalne zużycie podczas eksperymentu:
 - Serwer echa: ciągły monitoring co 500ms,
 - Producent-konsument: migawka przed i po eksperymencie,
 - Pomiar wieloplatformowy (Linux: /proc, macOS: ps).
- Narzut środowiska wykonawczego - pamięć wykorzystywana przez środowisko uruchomieniowe języka.

6.1.4. Środowisko eksperymentalne

Konfiguracja procesu komplikacji

- Rust: `cargo build -release` z pełnymi optymalizacjami kompilatora,
- C++: `g++ -O3 -DNDEBUG -std=c++20 -pthread`,
- Alokator pamięci: `jemalloc` dla języka Rust, systemowy alokator dla języka C++,

- Architektura docelowa: natywna architektura procesora (-march=native).

Konfiguracja środowiska uruchomieniowego

- Wątki Tokio: ręcznie ustawiane,
- Wątki C++: std::thread::hardware_concurrency(),
- Rozmiar stosu: domyślne ustawienia systemowe (8MB Linux, 2MB macOS),
- Limity połączeń: ograniczenia na poziomie systemu operacyjnego ulimit -n.

6.1.5. Procedura pomiarowa

Protokół eksperymentalny

- Serie pomiarowe: 10 powtórzenia z pełnym pomiarem,
- Okres stabilizacji: 200ms (producent-konsument) lub 1000ms (serwer echa) między eksperymentami,
- Limit czasu wykonania: 300s dla eksperymentów serwera echa, 30s na połączenie dla implementacji Rust,
- Brak fazy rozgrzewki (ang. *warmup runs*) - pomiary od pierwszego uruchomienia.

6.1.6. Narzędzia badawcze i automatyzacja

Aplikacja uruchamiająca eksperymenty

Skrypt Python z automatycznym:

- Parsowaniem argumentów linii poleceń,
- Kontrolą limitów czasu,
- Generowaniem plików CSV z wynikami,
- Zbieraniem informacji o systemie (OS, architektura, procesor, pamięć).

Wsparcie wieloplatformowe

- **Linux:** /proc/self/status dla pomiaru pamięci RSS,
- **macOS:** mach_task_basic_info i polecenie ps.

6.2. Programowanie równoległe

W przypadku programowania równoległego, zdecydowano się na wykorzystanie uznanego zestawu testowego NAS Parallel Benchmarks (NPB) [9]. 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 (T_1/T_n) dla różnych liczb wątków,
- Efektywność (przyspieszenie/liczba procesorów).

6.2.2. Wydajność sprzętowa (MFLOP/s)

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

- Wydajność w pojedynczym wątku,
- Wydajność wielowątkowa,
- Efektywność skalowania (MFLOP/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. *integer 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 [9].

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 [9].

IS - Sortowanie liczb całkowitych

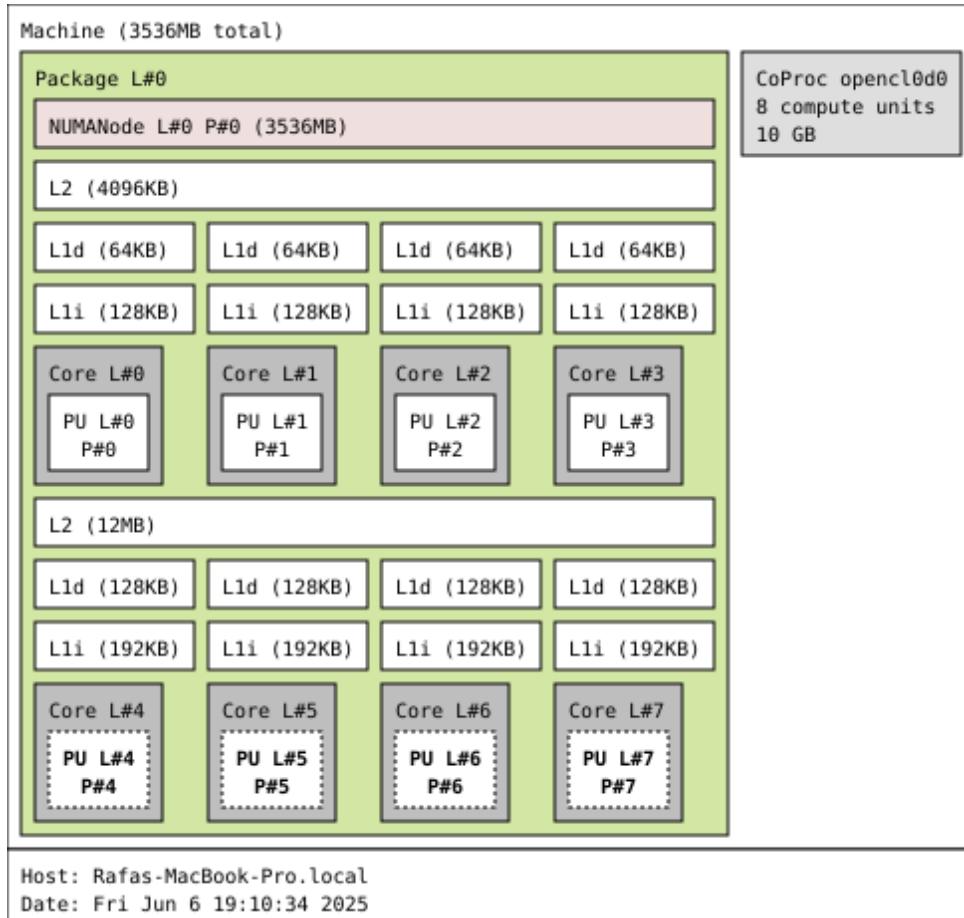
Benchmark IS (ang. *integer sorting*) służy do oceny wydajności systemów obliczeniowych w zakresie operacji nieciągłych i trudnych do zrównoleglenia, takich jak sortowanie i przemieszczanie danych w pamięci. Test polega na wygenerowaniu losowego zestawu liczb całkowitych, a następnie ich posortowaniu 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 [9].

6.3. Środowisko badań

6.3.1. Infrastruktura sprzętowo-programowa

Badania eksperymentalne zostały zaprojektowane w celu umożliwienia porównania mechanizmów programowania współbieżnego w językach Rust oraz C++ przy wykorzystaniu dwóch odmiennych architektur sprzętowych: x86_64 oraz ARM64. Taki dobór środowisk pozwala na analizę wpływu architektury procesora i systemu operacyjnego na wydajność oraz efektywność implementacji algorytmów współbieżnych.

Platforma ARM64 - Apple Silicon



Rys. 6.1: Topologia procesora Apple M1 z widoczną architekturą big.LITTLE

Platforma ARM64 reprezentowana jest przez komputer Apple MacBook Pro z następującymi specyfikacjami technicznymi:

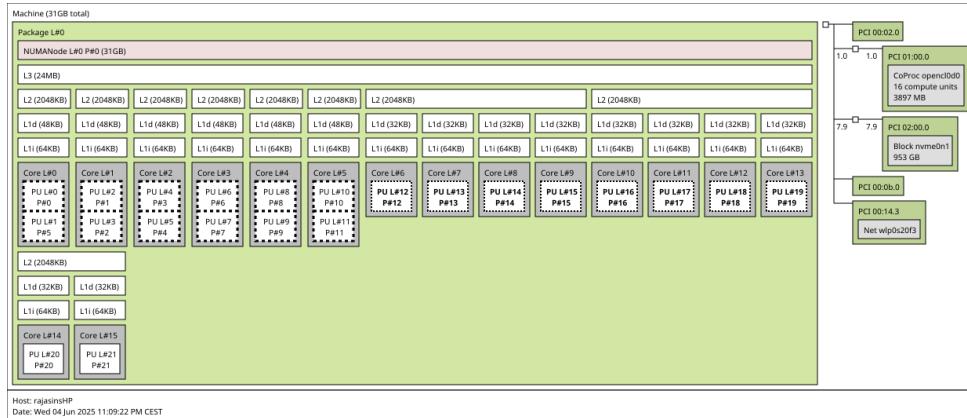
Tabela 6.3: Specyfikacja platformy ARM64

Komponent	Specyfikacja
Procesor	Apple M1 (ARM64)
Architektura	8 rdzeni ($4 \times$ P-core + $4 \times$ E-core)
Pamięć podrzczna L2	$4 \times 12\text{MB}$ (P-cores) + $4 \times 4\text{MB}$ (E-cores)
Pamięć operacyjna	16 GB LPDDR4X unified memory
System operacyjny	macOS (Darwin kernel 23.5.0)
Jądro systemu	arm64
Nazwa hosta	mbp-m1

Procesor Apple M1 charakteryzuje się architekturą heterogeniczną (ang. *big.LITTLE*) z rdzeniami o różnej wydajności:

- **Performance cores (P-cores):** 4 rdzenie wysokowydajne o częstotliwości do 3.2 GHz,
- **Efficiency cores (E-cores):** 4 rdzenie energooszczędnne o częstotliwości do 2.0 GHz,
- **Unified Memory Architecture:** Wspólna pamięć dla CPU i GPU eliminująca kopiowanie danych,
- **Pamięć podrzczna:** Hierarchiczna struktura z dedykowanymi poziomami L1/L2 per rdzeń.

Platforma x86_64 - Intel



Rys. 6.2: Topologia procesora Intel Core Ultra 7 165H z widoczną architekturą heterogeniczną

Platforma x86_64 reprezentowana jest przez komputer HP z następującymi specyfikacjami technicznymi:

Tabela 6.4: Specyfikacja platformy x86_64

Komponent	Specyfikacja
Procesor	Intel(R) Core(TM) Ultra 7 165H
Architektura	x86_64 (Intel Meteor Lake)
Rdzenie całkowite	22 (16 P-cores + 6 E-cores)
Wątki logiczne	22 (P-cores z Hyper-Threading wyłączonym)
Pamięć podrzczna L2	P-cores: 2048KB × 6, E-cores: 2048KB × 8
Pamięć podrzczna L3	24MB współdzielona
Pamięć operacyjna	32 GB DDR5-4800
System operacyjny	Linux Ubuntu 24.04 LTS
Jądro systemu	6.11.0-25-generic
Nazwa hosta	hp-intel

Procesor Intel Core Ultra 7 165H (Meteor Lake) również wykorzystuje architekturę heterogeniczną:

- **P-cores:** 6 rdzeni wysokowydajnych z obsługą Hyper-Threading (12 wątków logicznych),
- **E-cores:** 8 rdzeni energooszczędnnych (8 wątków logicznych),
- **LP E-cores:** 2 rdzenie niskoenergiczne (2 wątki logiczne),
- **NUMA:** Non-Uniform Memory Access z Package L#0,
- **Technologie:** Intel Turbo Boost, Intel Speed Shift, Advanced Vector Extensions (AVX2).

6.3.2. Środowiska programistyczne

Kompilatory i narzędzia

Tabela 6.5: Wersje kompilatorów na platformach testowych

Platforma	Rust	C++
ARM64 (macOS)	rustc 1.75.0	clang++ 14.0.3
x86_64 (Linux)	rustc 1.75.0	g++ 11.4.0

Flagi kompilacji

- **Rust:** cargo build -release z optymalizacjami opt-level = 3,
- **C++ (Linux):** g++ -O3 -DNDEBUG -std=c++20 -pthread -march=native,
- **C++ (macOS):** clang++ -O3 -DNDEBUG -std=c++20 -pthread.

6.3.3. Narzędzia pomiarowe i diagnostyczne

Narzędzia profilowania wydajności

Linux x86_64 - perf

Monitorowane metryki:

- **Cykle procesora:** Całkowita liczba cykli zegara,
- **Instrukcje:** Liczba wykonanych instrukcji,
- **Cache performance:** Trafienia i chybienia pamięci podręcznej,
- **Predykcja rozgałęzień:** Dokładność mechanizmów przewidywania rozgałęzień warunkowych,
- **Przełączenia kontekstu:** Liczba operacji zmiany kontekstu wykonywania.

macOS ARM64 - Instruments

Wykorzystanie pakietu Xcode Instruments:

- **System Trace:** Analiza planowania wątków i synchronizacji,
- **Allocations:** Monitorowanie alokacji i dealokacji pamięci,
- **Time Profiler:** Profilowanie czasowe z dokładnością do funkcji,
- **System Usage:** Wykorzystanie zasobów systemowych.

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 [50] oraz programów bazujących na nich w języku Rust, napisanych w ramach projektu na studia przez G.Bessa et al. [22].

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 {  
    // Rust: stałe kompilacyjne + proceduralna struktura  
    const CLASS: &str = "S";  
    const TOTAL_KEYS: usize = 1 << 16;  
  
    fn main() {  
        let num_threads = parse_args();  
  
        // Rust: declarative thread pool configuration  
        rayon::ThreadPoolBuilder::new()  
            .num_threads(num_threads)  
            .build_global().unwrap();  
  
        algorithm_implementation();  
    }  
  
    // Funkcje niezależne operujące na globalnych danych  
    fn algorithm_implementation() { /* ... */ }  
    fn verification_and_results() { /* ... */ }
```

Kluczowe cechy organizacji kodu Rust:

7. Porównanie między językowe - programowanie równoległe

- Struktura proceduralna - główna logika programu umieszczana jest w funkcji `main()`, analogicznie do języka C++.
- Globalne stałe - parametry problemu definiowane są jako stałe kompilacyjne (`const`).
- Funkcje niezależne - logika algorytmu implementowana jest w postaci oddzielnych funkcji, a nie metod związanych z obiektami.
- Automatyczne zarządzanie pamięcią - dynamiczne struktury danych (np. `Vec<T>`) zastępują ręczne zarządzanie pamięcią (np. `malloc/free`).
- Integracja z biblioteką Rayon - jednolite zastosowanie biblioteki Rayon we wszystkich benchmarkach w celu realizacji współbieżności.

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 NPBenchmark {
    // Rust: automatyczne zarządzanie pamięci przez Vec<T>
    let mut key_array: Vec<i32> = vec![0; TOTAL_KEYS];
    let mut working_data: Vec<f64> = vec![0.0; PROBLEM_SIZE];

    // Rust: thread-safe storage bez jawnego blokowania
    thread_local! {
        static THREAD_DATA: RefCell<Vec<f64>> = RefCell::new(vec![0.0; SIZE]);
    }

    fn algorithm_core(data: &mut Vec<i32>, working: &mut Vec<f64>) {
        // Pożyczanie pozwala na bezpieczny dostęp bez przenoszenia własności
        parallel_processing(data, working);
        // Kompilator gwarantuje brak wyścigów danych i użyciu po zwolnieniu
    }
}
```

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 komplikacji - brak warunków wyścigu gwarantowany przez mechanizm pożyczania.
- Ergonomiczne API - intuicyjne przekształcenie kodu sekwencyjnego na równoległy bez znacznego zwiększenia 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 NN: u32 = 1 << 8;
const NK: usize = 1 << 16;

fn ep_main() {
    // Rust: deklaratywna iteracja równoległa + redukcja
    let (sx, sy) = (1..NN+1)
        .into_par_iter()
        .map(|_| {
            // Box-Muller transform
            let (x1, x2) = generate_random_pair();
            let t1 = x1 * x1 + x2 * x2;

            if t1 <= 1.0 {
                let t2 = (-2.0 * t1.ln() / t1).sqrt();
                (x1 * t2, x2 * t2)
            } else { (0.0, 0.0) }
        })
        .reduce(|| (0.0, 0.0), |a, b| (a.0 + b.0, a.1 + b.1));
}

```

7.1.5. Specyfika benchmarku CG

Listing 7.5: Implementacja benchmarku CG w języku Rust

```
fn cg_iteration(a: &[f64], p: &mut [f64], q: &mut [f64]) {
```

```
// Rust: parallel matrix-vector multiply
q.par_chunks_mut(CHUNK_SIZE)
    .enumerate()
    .for_each(|(i, q_chunk)| {
        q_chunk[0] = sparse_matvec(&a, &p, i);
    });

// Rust: parallel dot product reduction
let dot_result = p.par_iter()
    .zip(q.par_iter())
    .map(|(&p_val, &q_val)| p_val * q_val)
    .sum::<f64>();
}

}
```

7.1.6. Specyfika benchmarku IS

Listing 7.6: Implementacja benchmarku IS w języku Rust

```
const TOTAL_KEYS: usize = 1 << 16;
const NUM_BUCKETS: usize = 1 << 9;

// Rust: encapsulated state vs C++ global variables
struct ISBenchmark {
    keys: Vec<u32>,
    buckets: Vec<Vec<u32>>,
}

impl ISBenchmark {
    fn parallel_bucket_sort(&mut self) {
        // Rust: safe parallel processing with ownership
        self.keys.par_sort_by_key(|&key| bucket_index(key));
    }
}
```

- Mieszana architektura: **EP** i **CG** są proceduralne, natomiast **IS** posiada strukturę obiektową.
- Enkapsulacja stanu: Złożone struktury danych są definiowane jako **struct** z odpowiadającą im implementacją w **impl**.
- Zarządzanie własnością: Użycie **&mut self** pozwala na bezpieczną modyfikację stanu bez naruszenia reguł współbieżności.
- Bezpieczeństwo wątków: Gwarantowane automatycznie przez system typów i mechanizm pożyczania.

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 w Fortranie:

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

```
// OpenMP: globalne zmienne + ręczne zarządzanie pamięcią
static double (*data) = (double*)malloc(sizeof(double) * SIZE);
static int (*indices) = (int*)malloc(sizeof(int) * SIZE);
```

```

int main(int argc, char **argv) {
    // OpenMP: proceduralny styl z globalnym stanem
    initialize_data();
    parallel_algorithm();

    // Ręczne czyszczenie pamięci
    free(data); free(indices);
}

```

- 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 wstępna - 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 jawnego 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 - ekspliktyne 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 wystąpienia błędów, takich jak użycie pamięci po jej zwolnieniu czy podwójne zwolnienie pamięci, które mogą prowadzić do niestabilności programu, naruszenia integralności danych lub poważnych luk w bezpieczeństwie.
- Przepełnienie bufora - 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

```
// OpenMP: dyrektywy kompilacyjne + ręczna synchronizacja
void parallel_algorithm() {
    double sum = 0.0;

    // Równoległa redukcja z harmonogramowaniem
    #pragma omp parallel for reduction(+:sum) schedule(static)
    for (int i = 0; i < SIZE; i++) {
        sum += compute_element(i);
    }

    // Sekcja krytyczna dla operacji zespołowych
    #pragma omp parallel
    {
        #pragma omp critical
        {
            update_shared_state();
        }
    }
}
```

- Oparte na dyrektywach - wykorzystanie konstrukcji `#pragma` do deklaratywnego sterowania równoległością w kodzie źródłowym.
- Ręczna synchronizacja - konieczność samodzielnego zarządzania sekcjami krytycznymi, bariерami 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

```
// EP Benchmark: Embarrassingly Parallel + OpenMP
void ep_main() {
    double sx = 0.0, sy = 0.0;

    #pragma omp parallel
    {
        double qq[NQ] = {0.0}; // Thread-private histogram
        double local_sx = 0.0, local_sy = 0.0;
```

```

#pragma omp for
for (int k = 1; k <= NN; k++) {
    double x[2*NK];
    vranlc(2*NK, &seed, A, x); // Generate random numbers

    // Box-Muller transform
    for (int i = 0; i < NK; i++) {
        double x1 = 2.0 * x[2*i] - 1.0;
        double x2 = 2.0 * x[2*i+1] - 1.0;
        double t1 = x1*x1 + x2*x2;

        if (t1 <= 1.0) {
            double t2 = sqrt(-2.0 * log(t1) / t1);
            local_sx += x1 * t2;
            local_sy += x2 * t2;
        }
    }
}

// OpenMP: explicit critical section for reduction
#pragma omp critical
{
    sx += local_sx; sy += local_sy;
}
}
}

```

7.2.5. Specyfika benchmarku CG

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

```

// CG: Conjugate Gradient + OpenMP
static double *a, *x, *p, *q, *r, *z;
static int *colidx, *rowstr;

void cg_iteration() {
    double rho, d = 0.0;

    // OpenMP: parallel sparse matrix-vector multiply
#pragma omp parallel for
    for (int j = 0; j < nrows; j++) {
        double sum = 0.0;
        for (int k = rowstr[j]; k < rowstr[j+1]; k++) {
            sum += a[k] * p[colidx[k]];
        }
        q[j] = sum;
    }

    // OpenMP: parallel dot product with reduction
#pragma omp parallel for reduction(+:d)
    for (int j = 0; j < ncols; j++) {
        d += p[j] * q[j];
    }

    // OpenMP: parallel vector updates
    double alpha = rho / d;
    #pragma omp parallel for reduction(+:rho)
    for (int j = 0; j < ncols; j++) {

```

```
    z[j] += alpha * p[j];
    r[j] -= alpha * q[j];
    rho += r[j] * r[j];
}
}
```

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

```
// TBB: funkcyjny interfejs + globalne zmienne
#include <TBB/parallel_reduce.h>
#include <TBB/task_scheduler_init.h>

static double (*data) = (double*)malloc(sizeof(double) * SIZE);

int main() {
    // TBB: explicit scheduler initialization
    TBB::task_scheduler_init init(TBB::task_scheduler_init::automatic);

    parallel_algorithm();

    free(data); // Ręczne zarządzanie pamięcią
}
```

- 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, przepełnienie 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

```
// TBB: funkcyjny interfejs + work-stealing scheduler
void parallel_algorithm() {
    // TBB: parallel reduction with lambda
    double sum = TBB::parallel_reduce(
        TBB::blocked_range<size_t>(0, SIZE),
        0.0,
        [&](const TBB::blocked_range<size_t>& r, double local_sum) {
            for (size_t i = r.begin(); i != r.end(); ++i) {
                local_sum += compute_element(i);
            }
            return local_sum;
        },
        std::plus<double>()
    );
}

// TBB: parallel for with automatic load balancing
TBB::parallel_for(
    TBB::blocked_range<size_t>(0, SIZE),
    [&](const TBB::blocked_range<size_t>& r) {
        for (size_t i = r.begin(); i != r.end(); ++i) {
            process_element(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 typizacji w porównaniu do OpenMP, co zmniejsza liczbę błędów w czasie komplikacji.

- 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` dla redukcji przez `parallel_reduce` z lambda,
- CG: Konwersja operacji `sparse_matrix` na `parallel_for` z `blocked_range`,
- IS: Implementacja sortowania kubełkowego przez `parallel_for` z synchronizacją przez mutexy.

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

```
// New C++: RAII + enkapsulacja
class NPBBenchmark {
    std::vector<double> data_;           // Automatic memory management
    std::vector<int> indices_;            // vs malloc/free
    BenchmarkResults results_;

public:
    virtual void run() = 0;
    virtual bool verify() const = 0;

    // RAII ensures automatic cleanup
    ~NPBBenchmark() = default;
};

}
```

- 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 modułarności, izolacji błędów i możliwości wielokrotnego wykorzystania kodu.
- Bezpieczeństwo typów - wykorzystanie nowoczesnych konstrukcji językowych C++, takich jak typy wyliczeniowe, silna typizacja 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

```
// Modern C++: RAII + smart pointers vs manual memory
class NPBBenchmark {
```

```

    std::vector<double> data_;
    std::unique_ptr<WorkingMemory> working_mem_;

public:
    NPBenchmark(size_t size) : data_(size) {}

    void worker_task() {
        std::vector<double> local_data(size_);
        /* computation... */
    } // Automatic cleanup on scope exit
};

```

7.4.3. Mechanizmy równoległości

Listing 7.17: Implementacja nowoczesnego C++ - równoległość

```

// Nowy C++: std::thread + RAII
void parallel_computation() {
    std::vector<std::thread> threads;

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

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

```

7.5. Architektura i organizacja kodu

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

Aspekt	Rust	C++ (OpenMP)	C++ (TBB)	C++ (nowoczesny)
Organizacja kodu	Mieszana: IS obiektowy, EP/CG proceduralne	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

7.6. Zarządzanie pamięcią

Tabela 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	Mozliwe	Mozliwe	Rzadkie (RAII)
Użycie po zwolnieniu	Niemożliwe	Mozliwe	Mozliwe	Rzadkie
Przepełnienia bufora	Sprawdzanie granic	Mozliwe	Mozliwe	Sprawdzanie granic
Podwójne zwolnienie	Niemożliwe	Mozliwe	Mozliwe	Rzadkie

Rozdział 8

Porównanie międzyjęzykowe - programowanie współbieżne

W ramach analizy programowania współbieżnego przeanalizowano implementacje wzorców producent-konsument oraz serwer echo w językach Rust i C++. Obie wersje korzystają z różnych podejść do zarządzania współbieżnością: Rust z biblioteką Tokio oraz C++ z mechanizmami standardowej biblioteki i rozszerzeniami języka C++20.

8.1. Porównanie międzyjęzykowe

8.1.1. Struktura i organizacja kodu

Tabela 8.1: Porównanie aspektów struktury i organizacji kodu w implementacjach współbieżnych

Aspekt	Rust (Tokio)	C++ (<code>std::thread</code> + <code>epoll/kqueue</code>)
Architektura	Sterowana zdarzeniami z asynchronicznym środowiskiem wykonawczym, tworzenie zadań	Wątek na połączenie + opcjonalna pętla zdarzeń
Enkapsulacja	Moduły, cechy, system właściwości	Klasy, szablony, przestrzenie nazw
Obsługa błędów	<code>Result<T,E></code> , operator <code>?</code> , wymuszona propagacja	<code>std::optional</code> , wyjątki, kody <code>errno</code>
Asynchroniczność	<code>async/await</code> , <code>tokio::select!</code> , wbudowane środowisko wykonawcze	<code>std::async</code> , ręczne pętle zdarzeń (<code>epoll/kqueue</code>)
Bezpieczeństwo typów	Sprawdzanie właściwości w czasie komplikacji, cechy <code>Send</code> / <code>Sync</code>	Sprawdzanie w czasie wykonania, ręczna synchronizacja
Konfiguracja	<code>Cargo.toml</code> , flagi funkcjonalności (ang. <i>feature flags</i>)	<code>CMake/Make</code> , dyrektywy preprocesora
Zarządzanie zależnościami	Scentralizowane z crates.io	Zdecentralizowane, pakiety systemowe

8. Porównanie między językowe - programowanie współbieżne

Implementacja w języku Rust cechuje się większą spójnością architektoniczną dzięki wbudowanym mechanizmom asynchroniczności oraz systemowi cech (ang. *traits*). C++ zapewnia większą elastyczność kosztem złożoności zarządzania zgodnością między różnymi wersjami standardu.

8.1.2. Zarządzanie pamięcią

Tabela 8.2: Porównanie modeli zarządzania pamięcią w badanych implementacjach

Aspekt	Rust	C++
Model podstawowy	Własność z automatycznym zwalnianiem; sprawdzanie pożyczania w czasie komplikacji	RAII z ręcznym zarządzaniem czasem życia obiektów
Współdzielenie w wątkach	<code>Arc<T></code> dla danych niemutowalnych, <code>Arc<Mutex<T>></code> dla mutowalnych	<code>std::shared_ptr<T></code> , ręczne opakowanie muteksem
Bezpieczeństwo pamięci	Gwarantowane na etapie komplikacji, eliminacja wyścigów danych	Błędy ujawniane w czasie wykonania; ryzyko użycia po zwolnieniu
Zwalnianie zasobów	Automatyczne dzięki cechom <code>Drop</code>	Destruktory RAII, często konieczne jawne zwolnienie (np. <code>close()</code>)
Pomiar pamięci	Ograniczony dostęp do systemu plików <code>/proc</code>	Pełna analiza wykorzystania pamięci (RSS, sterta)
Zarządzanie buforami	Bezpieczne wskaźniki i struktury (<code>Vec<T></code> , <code>slice</code>) ze sprawdzaniem granic	Surowe wskaźniki i tablice, ryzyko przepełnienia
Porządek pamięci	<code>Ordering::Relaxed</code> , <code>Acquire</code> , <code>Release</code>	Odpowiedniki w <code>std::memory_order</code>
Wykrywanie wycieków	Rzadkie (np. cykliczne referencje z <code>Arc</code>); wspierane przez system typów	Częsty problem, wymagane zewnętrzne narzędzia (np. Valgrind)

System własności w języku Rust eliminuje całe klasy błędów pamięciowych na etapie komplikacji. C++ wymaga większej ostrożności programisty oraz wykorzystania narzędzi diagnostycznych.

8.1.3. Mechanizmy współbieżności

Tabela 8.3: Porównanie mechanizmów współbieżności w badanych implementacjach

Mechanizm	Rust (Tokio)	C++ (std + wywołania systemowe)
Model wykonania	Wątki M:N z planistą opartym na kradzieży zadań (work-stealing scheduler)	Wątki 1:1 zarządzane przez systemowy planista
Tworzenie zadań	<code>tokio::spawn(async move {})</code> - lekki i szybki sposób uruchamiania zadań asynchronicznych	<code>std::thread::spawn()</code> z opcjonalnym <code>.detach()</code> lub synchronizacją przez <code>join()</code>
Komunikacja	Kanały wieloproducent-pojedynczykonsument: <code>mpsc::channel</code> z <code>Arc<Mutex<Receiver></code>	Własna implementacja <code>Channel<T></code> oparta na <code>std::condition_variable</code>
Synchronizacja	Primitives: <code>tokio::sync::Mutex</code> , <code>Semaphore</code> , <code>RwLock</code> - dostosowane do async	Klasyczne prymitywy: <code>std::mutex</code> , <code>condition_variable</code> , atomiki z <code>std::atomic</code>
Operacje wejścia/wyjścia	Nieblokujące I/O z użyciem await, np. <code>TcpListener::bind().awaitcv()</code> , <code>send()</code>	Blokujące systemowe wywołania: <code>accept()</code> , <code>awaitcv()</code> , <code>send()</code>
Obsługa zdarzeń	Asynchroniczne makro <code>tokio::select!</code> do oczekiwania na wiele zdarzeń	Ręczne pętle zdarzeń z wykorzystaniem <code>epoll_wait()</code> lub <code>kevent()</code>
Ograniczenia zasobów	Semafory asynchroniczne, np. <code>Semaphore::try_acquire_owned()</code>	Własne liczniki z użyciem <code>std::atomic<size_t></code>
Koordynacja zamknięcia	Kanały rozgłoszeniowe, np. <code>broadcast::channel</code> z łagodnym zamykaniem (graceful shutdown)	Flagi typu <code>std::atomic<bool></code> sygnalizujące zakończenie wątku
Zapobieganie wyścigom danych	Zapewnione na etapie komplikacji przez ograniczenia traitów <code>Send</code> i <code>Sync</code>	Potrzebna synchronizacja ręczna; ryzyko błędów wykonawczych i wyścigów danych

Porównując mechanizmy współbieżności w językach Rust i C++, można zauważyć fundamentalne różnice w filozofii projektowej i gwarancjach bezpieczeństwa. Rust przyjmuje podejście

"bezpieczeństwo przede wszystkim", podczas gdy C++ oferuje większą elastyczność, ale wymaga większej dyscypliny od programisty.

Rust - kanały typizowane

Implementacja Rust wykorzystuje kanały z różnymi semantykami komunikacji (broadcast):

Listing 8.1: Producent-Konsument w Rust z mpsc channels

```
// Rust: producent-konsument z kanałami mpsc
fn producer_consumer_benchmark() {
    let (tx, rx) = mpsc::channel();
    let rx = Arc::new(Mutex::new(rx));
    let shutdown = Arc::new(AtomicBool::new(false));

    // Wątki producentów
    thread::spawn(move || {
        for i in 0..ITEMS {
            tx.send(format!("Item-{}", i)).unwrap();
        }
        // Kanał zamyka się automatycznie po usunięciu tx
    });
}

// Wątki konsumentów - współdzielony odbiornik z Arc<Mutex<>>
let rx_clone = Arc::clone(&rx);
thread::spawn(move || {
    while let Ok(msg) = rx_clone.lock().unwrap().try_recv() {
        process_message(msg);
    }
});
}
```

C++ - ręczna synchronizacja

Wersja C++ opiera się na ręcznym zarządzaniu współbieżnością z użyciem mutexów i zmiennych warunkowych:

Listing 8.2: Producent-Konsument w C++ z Channel

```
// C++: Własny kanał z ręczną synchronizacją
template<typename T>
class Channel {
    std::queue<T> queue_;
    mutable std::mutex mutex_;
    std::condition_variable condition_;
    bool closed_ = false;

public:
    void send(T item) {
        std::lock_guard lock(mutex_);
        queue_.push(std::move(item)); condition_.notify_one();
    }

    bool recv(T& item) {
        std::unique_lock lock(mutex_);
        condition_.wait(lock, [this]{ return !queue_.empty() || closed_; })
        if (queue_.empty()) return false;
        item = std::move(queue_.front()); queue_.pop();
    }
}
```

```

        return true;
    }

    void close() { std::lock_guard lock(mutex_); closed_ = true; condition_
      → .notify_all(); }
};


```

Serwera echo

Rust - wejście/wyjście sterowane zdarzeniami

Implementacja Rust wykorzystuje nieblokujące I/O:

Listing 8.3: Echo Serwer w Rust z Tokio

```

// Rust: Asynchroniczny echo serwer z Tokio
async fn run_echo_server(addr: &str, max_connections: usize) {
    let listener = TcpListener::bind(addr).await.unwrap();
    let semaphore = Arc::new(Semaphore::new(max_connections));

    loop {
        tokio::select! {
            Ok((stream, addr)) = listener.accept() => {
                if let Ok(permit) = semaphore.clone().try_acquire_owned() {
                    tokio::spawn(async move {
                        handle_client(stream, addr).await;
                        drop(permit); // Auto connection limit
                    });
                }
            }
            _ = shutdown_signal() => break,
        }
    }
}

async fn handle_client(mut stream: TcpStream, addr: SocketAddr) {
    let mut buffer = [0; 1024];
    while let Ok(n) = stream.read(&mut buffer).await {
        if n == 0 { break; }
        stream.write_all(&buffer[..n]).await.unwrap(); // Echo back
    }
}

```

C++ - jeden wątek na połączenie

Wersja C++ tworzy osobny wątek dla każdego klienta:

Listing 8.4: Echo Serwer w C++ z jednym wątkiem na połączenie

```

// C++: Traditional Thread-per-Connection Model
class EchoServer {
    int server_socket;
    std::atomic<bool> running{true};
    std::atomic<size_t> current_connections{0};
    size_t max_connections;

public:
    EchoServer(const std::string& addr, int port, size_t max_conn) :
      → max_connections(max_conn) {
        server_socket = socket(AF_INET, SOCK_STREAM, 0);

```

```

        sockaddr_in server_addr{AF_INET, htons(port), {INADDR_ANY}};
        bind(server_socket, (sockaddr*)&server_addr, sizeof(server_addr));
        listen(server_socket, SOMAXCONN);
    }

    void run() {
        while (running.load()) {
            int client = accept(server_socket, nullptr, nullptr);
            if (current_connections.load() < max_connections) {
                // C++: Each connection = new OS thread
                std::thread([this, client]() {
                    current_connections++;
                    char buffer[1024];
                    while (int n = recv(client, buffer, sizeof(buffer), 0))
                        → {
                            send(client, buffer, n, 0); // Echo back
                        }
                    close(client);
                    current_connections--;
                }).detach();
            } else {
                close(client); // Reject if at capacity
            }
        }
    }
};

```

C++ - wersja asynchroniczna z epoll/kqueue

C++ oferuje także asynchroniczną implementację używającą epoll (Linux) lub kqueue (macOS):

Listing 8.5: Async Echo Serwer w C++ z event loop

```

// C++: Async Event-Driven Echo Server (epoll/kqueue)
class AsyncEchoServer {
    int server_socket;
    std::atomic<bool> running{true};
    std::vector<int> active_clients;

#ifndef __APPLE__
    int event_fd; // kqueue descriptor
#else
    int event_fd; // epoll descriptor
#endif

public:
    AsyncEchoServer(const std::string& addr, int port) {
        // Setup non-blocking server socket
        server_socket = socket(AF_INET, SOCK_STREAM, 0);
        fcntl(server_socket, F_SETFL, O_NONBLOCK);

        sockaddr_in server_addr{AF_INET, htons(port), {INADDR_ANY}};
        bind(server_socket, (sockaddr*)&server_addr, sizeof(server_addr));
        listen(server_socket, SOMAXCONN);

        // Platform-specific event system initialization
        init_event_system();
    }
}

```

```

void run_async() {
    while (running.load()) {
        auto events = wait_for_events(); // Platform abstraction

        for (auto& event : events) {
            if (event.fd == server_socket) {
                handle_new_connection();
            } else {
                handle_client_data(event.fd);
            }
        }
    }
}

private:
    void init_event_system() {
#ifdef __APPLE__
        event_fd = kqueue();
        struct kevent ev;
        EV_SET(&ev, server_socket, EVFILT_READ, EV_ADD, 0, 0, nullptr);
        kevent(event_fd, &ev, 1, nullptr, 0, nullptr);
#else
        event_fd = epoll_create1(0);
        struct epoll_event ev{EPOLLIN, {.fd = server_socket}};
        epoll_ctl(event_fd, EPOLL_CTL_ADD, server_socket, &ev);
#endif
    }

    void handle_new_connection() {
        int client = accept(server_socket, nullptr, nullptr);
        fcntl(client, F_SETFL, O_NONBLOCK);
        active_clients.push_back(client);
        add_to_event_system(client);
    }

    void handle_client_data(int client_socket) {
        char buffer[1024];
        int n = recv(client_socket, buffer, sizeof(buffer), 0);

        if (n > 0) {
            send(client_socket, buffer, n, 0); // Echo back
        } else {
            disconnect_client(client_socket);
        }
    }

    void add_to_event_system(int fd) { /* platform-specific add */ }
    void disconnect_client(int fd) { /* cleanup + remove from events */ }
    auto wait_for_events() { /* platform-specific event waiting */ }
};

```

8.1.4. Wydajność i bezpieczeństwo

Tabela 8.4: Porównanie wydajności i bezpieczeństwa implementacji współbieżnych

Aspekt	Rust	C++
Bezpieczeństwo komplikacji	Wysokie - system pożyczania eliminuje wyścigi danych	Średnie - wymaga zewnętrznych narzędzi (np. ThreadSanitizer)
Narzut w czasie wykonania	Niski - abstrakcje bezkosztowe (ang. zero-cost)	Średni - narzut tworzenia wątków
Efektywność pamięci	Wysoka - współdzielona własność z Arc<T>	Średnia - ryzyko wycieków pamięci
Skalowanie	Wysokie - model M:N, kradzież zadań	Ograniczone - model 1:1, ograniczenia systemowe
Odzyskiwanie po błędach	Strukturalne - propagacja błędów przez Result<T, E>	Oparte na wyjątkach - ryzyko wycieków zasobów
Trudność debugowania	Średnia - błędy komplikacji z (ang. borrow checker)	Wysoka - wyścigi, błędy pamięci
Dojrzałość ekosystemu	Rozwijający się - Tokio, async-std	Dojrzały - OpenMP, TBB, Boost.Asio

8.1.5. Modele wykonania i zarządzanie zadaniami

Implementacja Rust wykorzystuje bibliotekę Tokio, która dostarcza model wątków M:N z planistą kradzieży zadań. W praktyce oznacza to, że liczba wątków systemu operacyjnego jest niezależna od liczby zadań aplikacyjnych. Funkcja `tokio::spawn()` tworzy lekkie zadania zarządzane przez środowisko wykonawcze, co umożliwia efektywne wykorzystanie zasobów przy dużej liczbie współbieżnych operacji.

Implementacja C++ opiera się na modelu 1:1, gdzie każde połączenie obsługiwane jest przez dedykowany wątek systemu operacyjnego. Alternatywna wersja asynchroniczna wykorzystuje wywołania systemowe `epoll()` na Linux lub `kevent()` na macOS do implementacji pętli zadań, jednak wymaga to jawnego zarządzania stanem połączeń.

Komunikacja i synchronizacja

W implementacji Rust komunikacja realizowana jest przez `mpsc::channel()` w konfiguracji `Arc<Mutex<Receiver<T>>>` dla wielu odbiorców. Kanał zapewnia bezpieczeństwo typów i automatyczne sprzątanie przy zamknięciu wszystkich nadawców.

Implementacja C++ definiuje własną klasę `Channel<T>` opartą na `std::queue` z synchronizacją przez `std::mutex` i `std::condition_variable`. Klasa implementuje metody `send()`, `try_recv()` oraz `recv()` z semantyką blokującą, wymagając ręcznego zarządzania stanem zamknięcia kanału.

Obsługa I/O

Rust wykorzystuje `TcpListener::bind().await` z nieblokującymi operacjami wejścia/wyjścia. Makro `tokio::select!` umożliwia równoległe oczekивание na wiele zdarzeń (nowe połączenia, sygnał zamknięcia). Ograniczenia połączeń realizowane są przez `Semaphore::try_acquire_owned()`.

C++ implementuje blokujące wejście/wyjście przez bezpośrednie wywołania systemowe: `socket()`, `bind()`, `listen()`, `accept()`. Każde połączenie obsługiwane jest w dedykowanym wątku z synchronicznymi operacjami `recv()`/`send()`. Wersja asynchroniczna wymaga ręcznego zarządzania stanem dla każdego deskryptora pliku.

8.1.6. Zarządzanie błędami i zasobami

Rust wymusza jawną obsługę błędów przez typy `Result<T, E>` i operator `?`. Automatyczne sprzątanie zasobów zapewniany jest przez cechę `Drop` - połączenia gniazd, alokacje pamięci oraz uchwyty wątków są automatycznie zwalniane.

C++ opiera się na wzorcu RAII z ręcznymi wywołaniami destruktorek. Sprzątanie gniazd wymaga explicite wywołań `close()`, a obsługa błędów realizowana jest przez rzucanie wyjątków lub sprawdzanie `errno`. Wycieki pamięci mogą wystąpić przy nieprawidłowym zarządzaniu cyklem życia obiektów.

Bezpieczeństwo współbieżności

Najbardziej zasadnicza różnica między Rust a C++ dotyczy podejścia do bezpieczeństwa współbieżnego. Rust stosuje rygorystyczny system typów oraz cechy `Send` i `Sync`, które gwarantują bezpieczeństwo dostępu do danych współdzielonych już na etapie komilacji. Kompilator odrzuca programy mogące prowadzić do wyścigów danych, co eliminuje całą klasę trudnych do wykrycia błędów.

W C++ bezpieczeństwo współbieżności opiera się na odpowiedzialności programisty. Wszelkie błędy synchronizacji (np. brak blokady, nadmierne współdzielenie danych) mogą prowadzić do wyścigów, które są trudne do wykrycia i debugowania. Choć dostępne są narzędzia analityczne, takie jak `ThreadSanitizer`, nie eliminują one problemów przed uruchomieniem programu i nie oferują gwarancji podobnych do Rust.

8.1.7. Wnioski

Analiza rzeczywistych implementacji ujawnia konkretne kompromisy między językami w kontekście programowania współbieżnego. Rust z Tokio charakteryzuje się:

- Bezpieczeństwem w czasie komplikacji przez system własności i cechy `Send`/`Sync`,
- Architekturą sterowaną zdarzeniami z efektywnym wykorzystaniem zasobów,
- Wymuszonym obsługą błędów eliminującym ciche awarie (ang. *silent failures*),
- Ograniczonymi możliwościami introspekcji środowiska wykonawczego,
- Zależnością od zewnętrznych pakietów (ang. *external crates*) dla podstawowej funkcjonalności.

C++ ze standardową biblioteką oferuje:

- Bezpośrednią kontrolę nad zasobami systemowymi i układem pamięci,
- Kompleksowe monitorowanie i profilowanie w czasie wykonania,
- Kompatybilność międzyplatformową z ręcznymi abstrakcjami platformowymi,
- Model wątek na połączenie z przewidywalnym użyciem zasobów,
- Ryzyko błędów w czasie wykonania wymagające ostrożnej synchronizacji.

W praktyce wybór między языкami zależy od priorytetów projektu: bezpieczeństwo a kontrola, produktywność a elastyczność, nowoczesność a kompatybilność. Należy również uwzględnić kompetencje zespołu oraz wymagania środowiska docelowego.

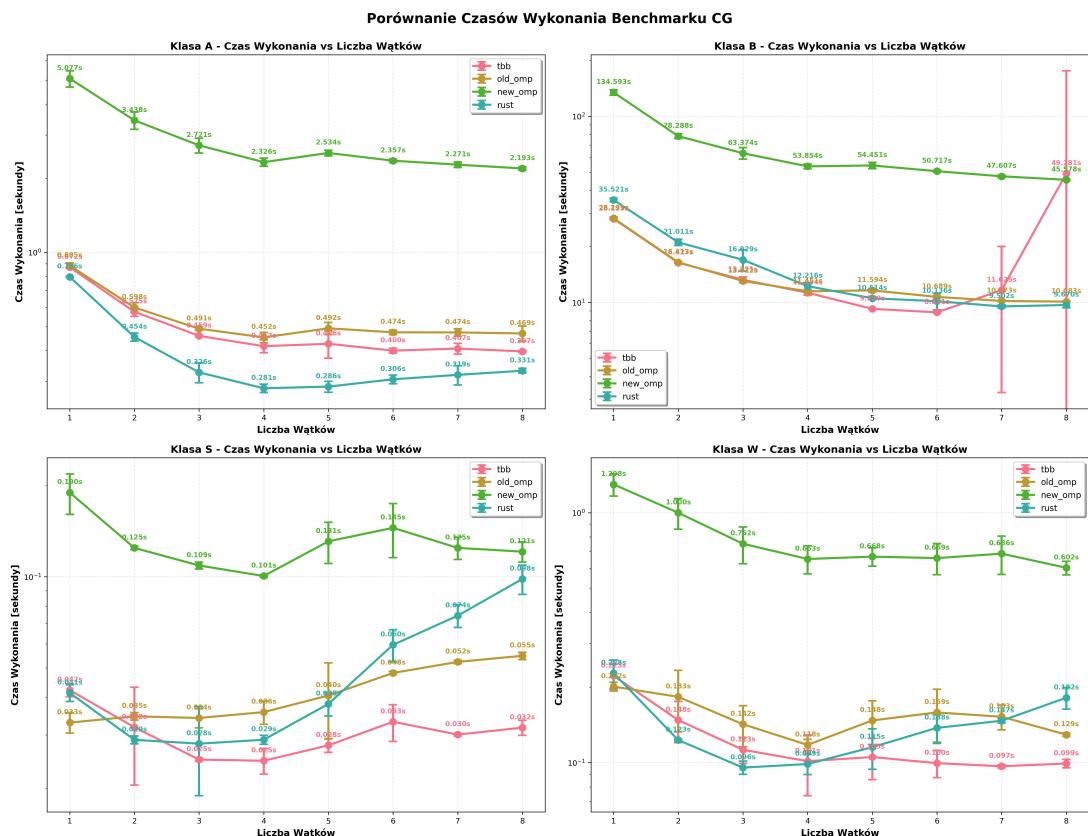
Rozdział 9

Analiza wyników - programowanie równoległe

W tym rozdziale zostały zebrane wyniki testów, które porównują ze sobą mechanizmy programowania równoległego w językach Rust i C++.

9.1. Benchmark CG

9.1.1. Wyniki benchmarków - platforma ARM64



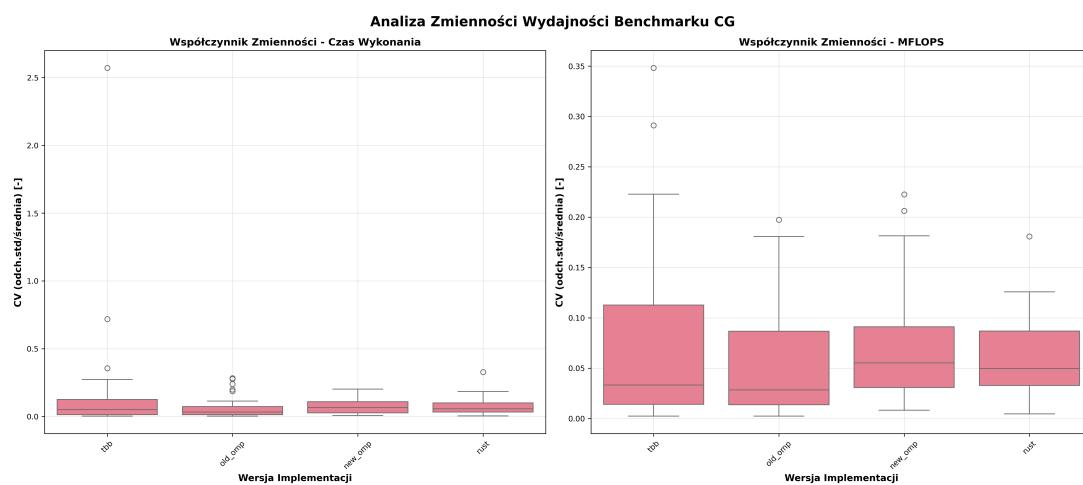
Rys. 9.1: Porównanie czasów wykonania benchmarku CG dla klas S, W, A, B względem liczby użytych wątków na platformie ARM64

Na rysunku 9.1 zaprezentowano zestawienie czasów wykonania benchmarku CG dla czterech klas problemu: S, W, A oraz B, przy użyciu czterech różnych implementacji równoległości: tbb, OpenMP w wersji oryginalnej w stylu języka Fortran (`old_omp`), OpenMP w wersji nowszej (`new_omp`) oraz implementacji w języku Rust. Dla każdej z klas przedstawiono zależność czasu wykonania od liczby wątków (od 1 do 8). Wartości zostały zaprezentowane w skali logarytmicznej.

Analiza czasów wykonania algorytmu CG ujawnia wyraźne różnice w wydajności poszczególnych implementacji, z dominującym problemem dotyczącym `new_omp`. Najbardziej uderzającą obserwacją jest znaczco gorsza wydajność tej implementacji we wszystkich klasach problemu. Czasy wykonania `new_omp` są wielokrotnie - nawet dziesięciokrotnie - dłuższe w porównaniu z pozostałymi rozwiązaniami. Dysproporcja ta jest szczególnie wyraźna w klasach A i B, gdzie `new_omp` operuje w zakresie sekund, podczas gdy inne implementacje osiągają czasy rzędu dziesiątych części sekundy. Może to wskazywać na fundamentalne niedociągnięcia nowej wersji OpenMP w kontekście testowanego algorytmu, potencjalnie wynikające z nieefektywnej dekompozycji zadań lub kosztownych mechanizmów synchronizacji.

Pozostałe implementacje prezentują bardziej zbliżone wyniki, z pewnymi zauważalnymi różnicami. W klasie A, implementacja `rust` wykazuje najlepsze czasy wykonania przy większej liczbie wątków (4-8), osiągając około 0,28-0,33 sekundy. W klasie B różnice są minimalne - wszystkie trzy implementacje (`tbb`, `old_omp`, `rust`) osiągają porównywalne wartości rzędu 0,09-0,10 sekundy przy większej liczbie wątków. Warto jednak zauważać istotną niestabilność czasów wykonania dla implementacji `tbb` przy 8 wątkach, co sygnalizuje duży rozrzut wyników. W mniejszych klasach problemowych - S i W - `tbb` często okazuje się najszybszą implementacją, zwłaszcza przy średnim poziomie równoległości (3-6 wątków).

Analizując wzorce skalowania, wszystkie implementacje wykazują wzrost wydajności przy przejściu z 1 do 3-4 wątków. Powyżej tego progu obserwuje się tendencję do stabilizacji lub wręcz nieznacznego pogorszenia wydajności, co może być związane z narutami na synchronizację oraz ograniczeniami sprzętowymi. Szczególnie pozytywnie wyróżnia się tu implementacja `rust` w klasie A, która prezentuje najbardziej stabilny i przewidywalny wzorzec skalowania. W pozostałych klasach implementacja ta wykazuje większą nieregularność.



Rys. 9.2: Analiza zmienności czasów wykonania benchmarku CG dla klas S, W, A, B względem liczby użytych wątków na platformie ARM64

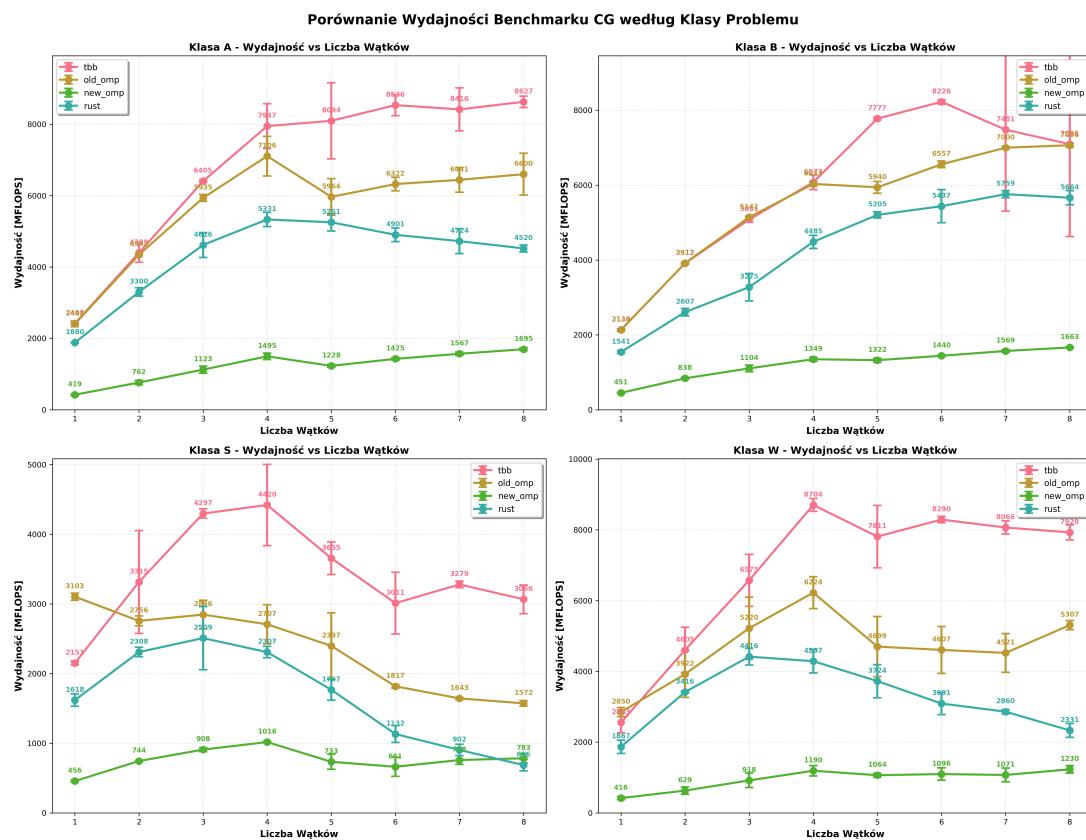
Powyższy wykres - rysunek 9.2 pudełkowy prezentuje współczynnik zmienności (ang. *CV - coefficient of variation*), obliczony jako stosunek odchylenia standardowego do wartości średniej, w odniesieniu do czasu wykonania (lewy wykres) oraz uzyskiwanej wydajności (prawy wykres) dla różnych wersji implementacji benchmarku.

9. Analiza wyników - programowanie równoległe

Ogólnie rzecz biorąc, czasy wykonania cechują się niską zmiennością - mediana CV dla wszystkich implementacji mieści się w przedziale 0,05-0,10, co świadczy o dobrej powtarzalności wyników. Na tym tle wyróżnia się `old_omp`, który osiąga najwyższy poziom zmienności, sygnalizując wysoką stabilność działania. Z kolei `tbb`, mimo niskiej mediany, wykazuje wartości odstające, z ekstremalnymi CV przekraczającymi 2,5.

Zmienność wartości MFLOPS jest wyraźnie wyższa niż czasów wykonania, co sugeruje, że mimo podobnych czasów działania, efektywność obliczeniowa może się znacznie ważyć. Największe odchylenia obserwuje się ponownie dla implementacji `tbb`, gdzie wartości CV sięgają nawet 0,35. `rust` wyróżnia się najbardziej konsekwentnym profilem zmienności, z wąskim zakresem rozrzutu wyników, co może wskazywać na dobrze zoptymalizowany model obliczeń wewnętrznych.

Z punktu widzenia poszczególnych klas problemowych, klasy A i B - reprezentujące większe zbiorы danych - wykazują najczytelniejsze i najbardziej regularne wzorce skalowania. To właśnie w tych klasach ujawniają się największe bezwzględne różnice w czasach wykonania między implementacjami, a korzyści z równoległości są najbardziej wyraźne. W przeciwnieństwie do nich, klasy S i W - charakteryzujące się mniejszymi rozmiarami danych - wykazują większą względną zmienność wyników i mniej przewidywalne zyski z równoległego przetwarzania. Nieregularne wzorce wydajności oraz większe odchylenia wskazują, że dla małych problemów korzyści z wielowątkowości są mniej oczywiste i bardziej zależne od szczegółów implementacyjnych.



Rys. 9.3: Porównanie wydajności benchmarku CG dla klas S, W, A, B względem liczby użytych wątków na platformie ARM64

Na wykresach na rysunku 9.3 zaprezentowano porównanie wydajności benchmarku CG mierzonej w MFLOPS (milionach operacji zmiennoprzecinkowych na sekundę). Wydajność została przedstawiona jako funkcja liczby wątków (1-8) dla czterech implementacji równoległych.

Analiza trendów wydajnościowych

Implementacja oparta na bibliotece Intel tbb konsekwentnie wykazuje najwyższą wydajność we wszystkich klasach problemów, osiągając szczytowe wartości MFLOPS (miliony operacji zmiennoprzecinkowych na sekundę) przy wykorzystaniu 3-6 wątków. Szczególnie widoczna jest jej przewaga w klasach A, B i W, gdzie osiąga wartości przekraczające 8000 MFLOPS.

Implementacja OpenMP w starszej wersji (`old_omp`) plasuje się na drugiej pozycji pod względem wydajności, osiągając wyniki o około 20-30% niższe niż tbb. Wykazuje ona podobny charakter skalowania, jednak z mniejszą efektywnością przy większej liczbie wątków.

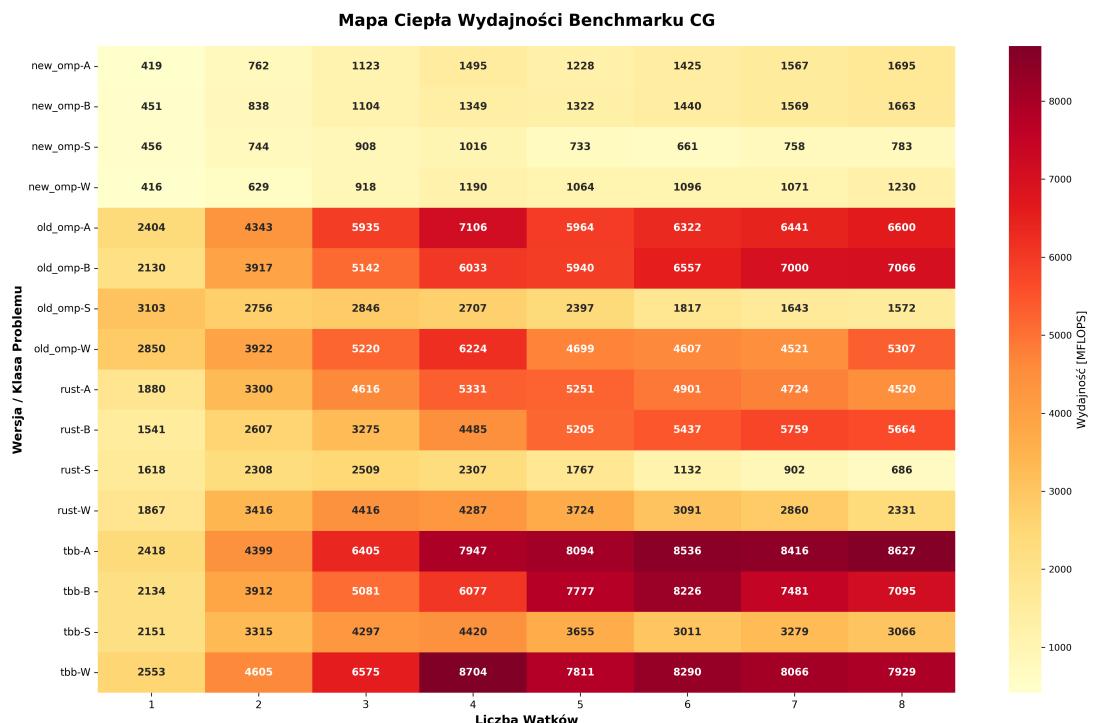
Implementacja w języku Rust prezentuje umiarkowaną wydajność, osiągając wyniki zbliżone do `old_omp` w niektórych konfiguracjach, jednak generalnie niższe. Warto zauważyć, że w klasie S wykazuje ona znaczący spadek wydajności powyżej 4 wątków.

Implementacja OpenMP w nowszej wersji (`new_omp`) osiąga konsekwentnie najniższe wyniki wydajności spośród wszystkich testowanych wariantów, z wartościami MFLOPS kilkukrotnie niższymi od lidera.

Charakterystyka skalowania

Większość implementacji osiąga szczytową wydajność przy wykorzystaniu 3-4 wątków, po czym następuje stabilizacja lub nawet spadek wydajności. Zjawisko to jest szczególnie wyraźne w klasie S, gdzie wszystkie implementacje wykazują znaczący spadek wydajności przy większej liczbie wątków, co sugeruje, że problemy tej klasy cechują się wysokim stosunkiem komunikacji do obliczeń.

Klasy A i B wykazują podobne charakterystyki skalowania, z wyraźnym wzrostem wydajności do około 4 wątków i późniejszym wypłaszczeniem. Klasa W charakteryzuje się podobnym trendem, jednak z większymi fluktuacjami wydajności.



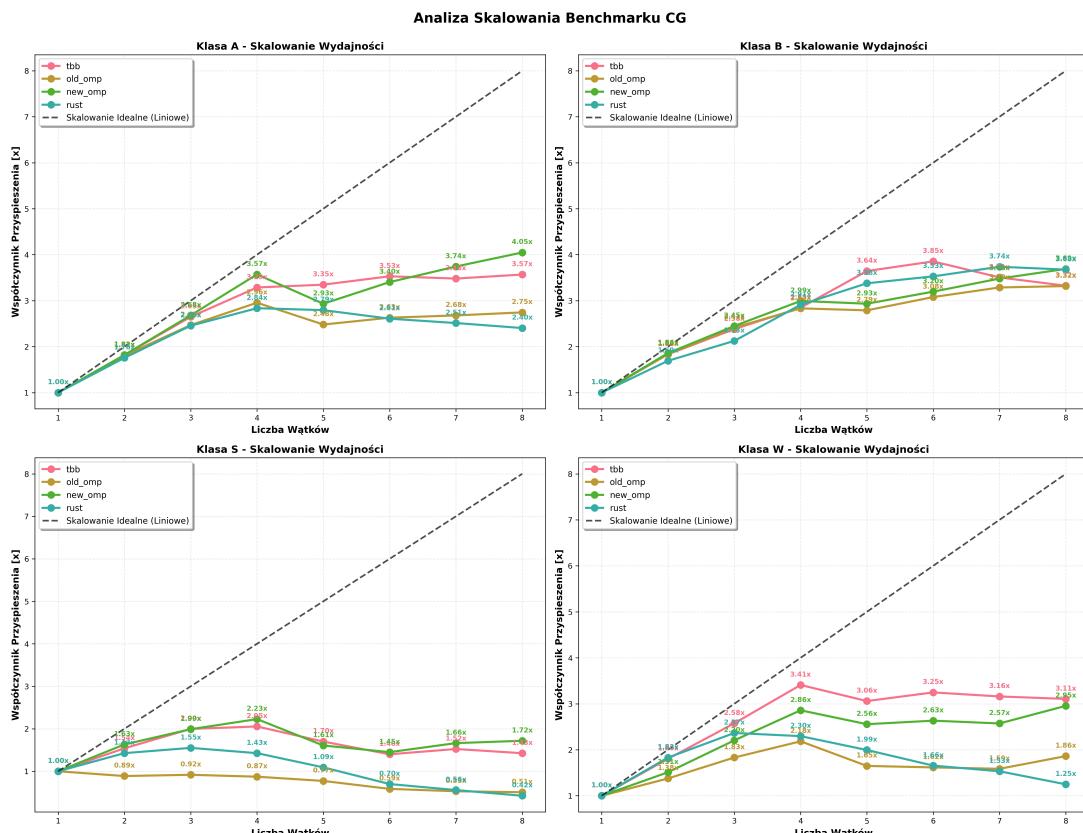
Rys. 9.4: Mapa ciepła wydajności benchmarku CG dla klas S, W, A, B względem liczby użytych wątków na platformie ARM64

Mapa ciepła wydajności - rysunek 9.4 potwierdza powyższe obserwacje, dostarczając kompleksowego widoku na zachowanie wszystkich implementacji w funkcji klasy problemu i liczby

9. Analiza wyników - programowanie równoległe

wątków. Najwyższe wartości wydajności (reprezentowane cieplejszymi, czerwonymi odcieniami) koncentrują się w obszarze implementacji tbb dla klas A, B i W przy liczbie wątków 3-8. Zimniejsze obszary mapy (żółte i jasnożółte) odpowiadają konsekwentnie implementacji new_omp dla wszystkich klas problemów.

Zauważalny jest również gradient wydajności w przypadku klasy S, gdzie wszystkie implementacje wykazują wyraźny spadek wydajności wraz ze wzrostem liczby wątków powyżej wartości optymalnej (3-4), co manifestuje się przejściem od cieplejszych do chłodniejszych odcieni w prawej części mapy ciepła.



Rys. 9.5: Analiza skalowania benchmarku CG dla klas S, W, A, B względem liczby użytych wątków na platformie ARM64

Powyższy wykres - rysunek 9.5 przedstawia skalowanie wydajności benchmarku CG. Skalowanie wyróżnione zostało za pomocą współczynnika przyspieszenia względem wykonania jednowątkowego i odniesione do skalowania idealnego (liniowego).

Analiza porównawcza implementacji

Żadna z badanych implementacji nie osiąga idealnego skalowania liniowego, co wskazuje na istnienie inherentnych ograniczeń w równoległym wykonaniu algorytmu CG. Większość implementacji osiąga punkt maksymalnego przyspieszenia przy wykorzystaniu 3-4 wątków, po czym efektywność skalowania ulega wyraźnemu spłaszczeniu lub nawet spadkowi. Zjawisko to można przypisać rosnącym kosztom synchronizacji, kontencji pamięci podręcznej oraz ograniczeniom przepustowości pamięci.

Implementacja Intel tbb wykazuje generalnie najlepsze charakterystyki skalowania w większości klas problemów, szczególnie widoczne w klasach A, B i W. W optymalnym przypadku osiąga przyspieszenie rzędu 3,0-3,6x przy wykorzystaniu 4-5 wątków. Implementacja ta wykazuje najwyższą efektywność równoległą, co może świadczyć o skutecznych mechanizmach zarządzania zadaniami i minimalizacji narzutu synchronizacji.

Implementacja OpenMP w nowszej wersji prezentuje interesującą charakterystykę skalowania, osiągając w klasie problemu A najwyższe zaobserwowane przyspieszenie około 4,5x przy 8 wątkach. Wynik ten sugeruje, że nowa implementacja OpenMP może efektywniej wykorzystywać zasoby obliczeniowe w przypadku określonych wzorców dostępu do pamięci.

Starsza implementacja OpenMP wykazuje umiarkowane skalowanie w klasach A i B, jednak znaczco gorsze w klasie S, gdzie przyspieszenie pozostaje minimalne niezależnie od liczby wątków. Wskazuje to na potencjalne problemy w obsłudze problemów o wysokim stosunku komunikacji do obliczeń.

Implementacja w języku Rust cechuje się początkowym wzrostem wydajności, który szybko ulega nasyceniu, a przy wyższej liczbie wątków często obserwuje się spadek przyspieszenia. Szczególnie widoczne jest to w klasie S, gdzie przy 7-8 wątkach współczynnik przyspieszenia spada poniżej 1,0, co oznacza degradację wydajności względem wykonania jednowątkowego.

Wpływ klasy problemu na skalowanie

Klasy problemów A i B wykazują zbliżone charakterystyki skalowania dla wszystkich badanych implementacji, co sugeruje podobne właściwości dostępu do pamięci i intensywności obliczeniowej.

Klasa S charakteryzuje się zdecydowanie najgorszym skalowaniem spośród wszystkich badanych klas, z maksymalnym przyspieszeniem nieprzekraczającym 2x. Wynik ten wskazuje na wysokie koszty komunikacji w stosunku do wykonywanych obliczeń lub nieefektywne wykorzystanie lokalności danych.

Klasa W prezentuje charakterystykę skalowania pośrednią między klasami A/B a klasą S, z maksymalnym przyspieszeniem w granicach 3x dla implementacji tbb.

9.1.2. Wyniki benchmarków - platforma x86_64

Na wykresach - rysunek 9.6 porównujący czasy wykonania poszczególnych implementacji benchmarku CG (przedstawione w skali logarytmicznej) dla czterech klas problemów (A, B, S, W), zaobserwować można wyraźne różnice w wydajności poszczególnych implementacji oraz ich skalowaniu wraz ze wzrostem liczby wątków.

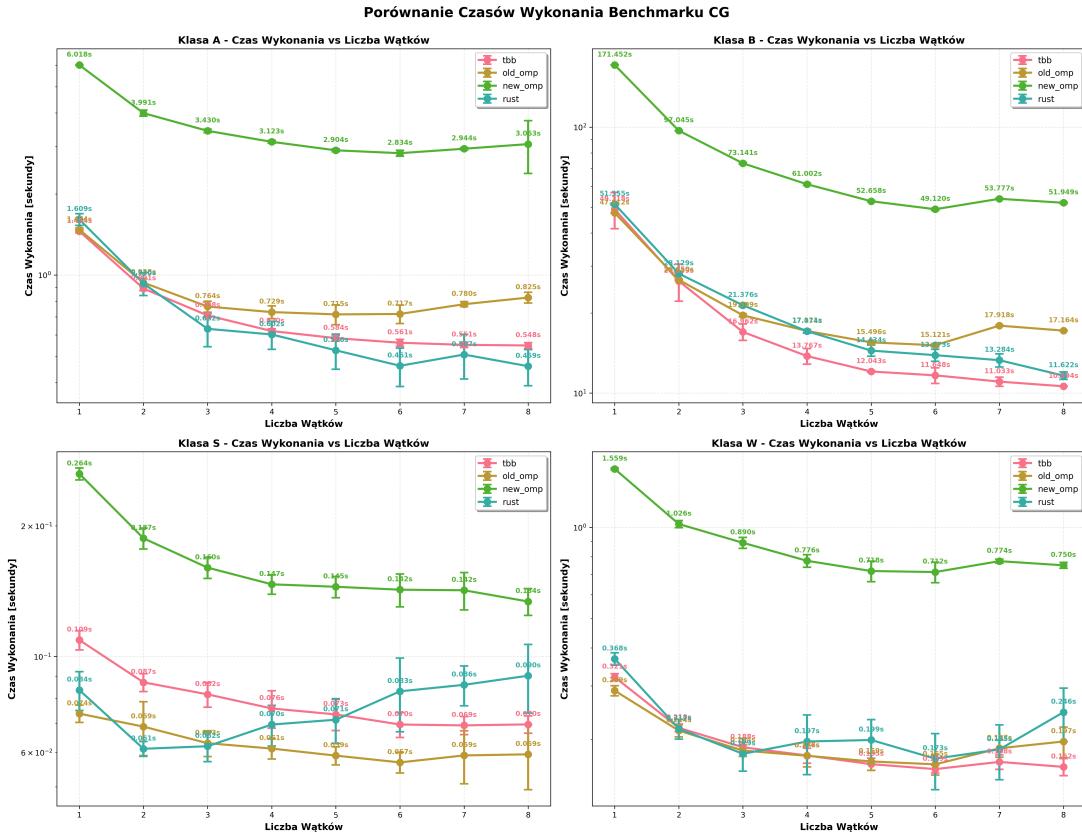
Implementacja new_omp konsekwentnie wykazuje najdłuższe czasy wykonania we wszystkich klasach problemów, z wartościami o rząd wielkości wyższymi niż pozostałe implementacje. Szczególnie widoczne jest to w klasach problemów A i B, gdzie różnica między implementacją new_omp a pozostałymi jest najbardziej znacząca. Pomimo stosunkowo dobrej stabilności tej implementacji, jej niska wydajność czyni ją najmniej optymalnym wyborem spośród badanych wariantów.

Implementacja tbb generalnie osiąga najkrótsze czasy wykonania, zwłaszcza przy wykorzystaniu większej liczby wątków (4-8). W przypadku klasy B dla 8 wątków, implementacja tbb osiąga czas wykonania rzędu 11 milisekund, co stanowi najlepszy wynik spośród wszystkich badanych konfiguracji.

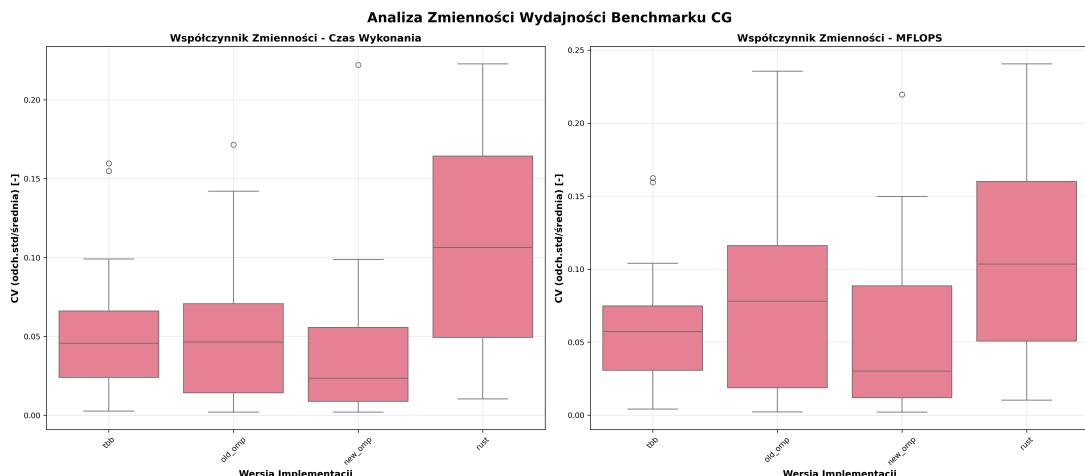
Implementacje rust i starsza wersja OpenMP (old_omp, kolor żółty) prezentują zbliżone charakterystyki czasów wykonania, często naprzemiennie zajmując drugą lub trzecią pozycję pod względem wydajności. W klasie problemu S można zaobserwować interesujące punkty przecięcia krzywych wydajności, gdzie implementacje zmieniają swoją relatywną efektywność w zależności od liczby wątków.

Wszystkie implementacje wykazują poprawę wydajności wraz ze wzrostem liczby wątków, jednak skala tej poprawy jest zróżnicowana. Najlepsze skalowanie obserwowane jest dla implementacji tbb, która wykazuje największy spadek czasu wykonania przy przejściu od 1 do 4 wątków. Dla większości implementacji, korzyści wynikające z dodawania kolejnych wątków powyżej 4-6 stają się marginalne, co wskazuje na osiągnięcie punktu nasycenia skalowania.

9. Analiza wyników - programowanie równoległe



Rys. 9.6: Porównanie czasów wykonania benchmarku CG dla klas S, W, A, B względem liczby wątków na platformie x86_64



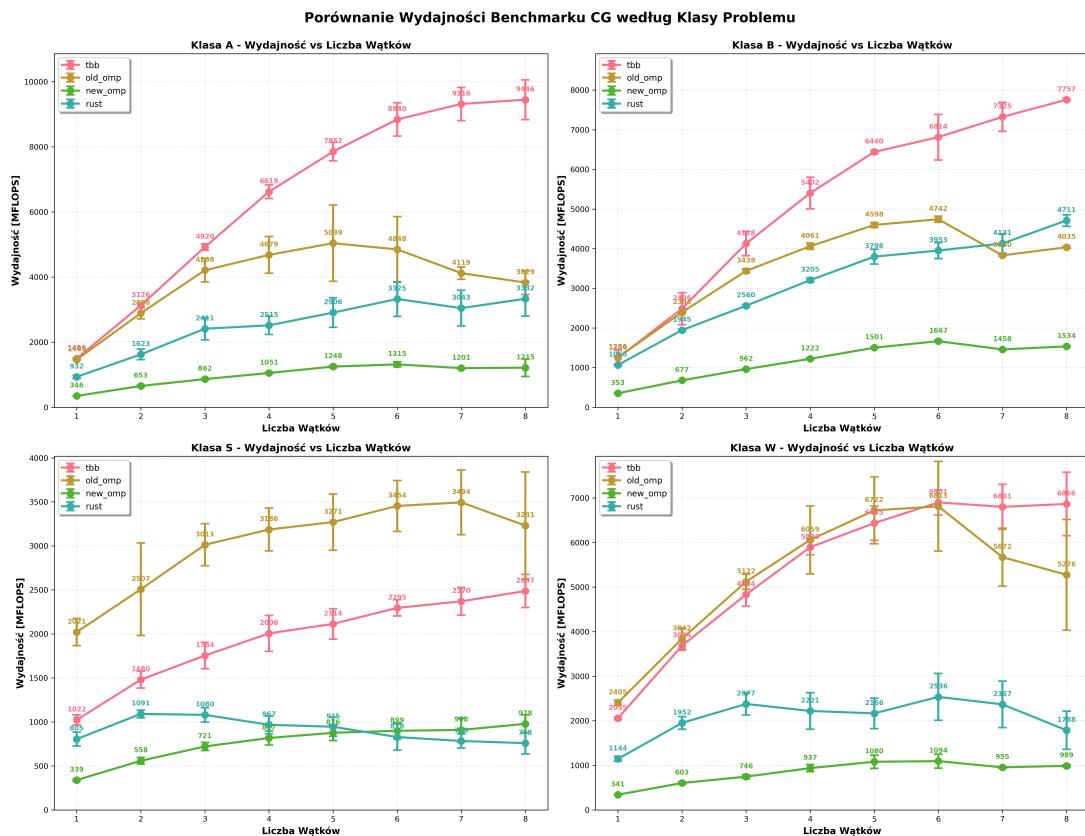
Rys. 9.7: Analiza zmienności czasów wykonania benchmarku CG dla klas S, W, A, B względem liczby wątków na platformie x86_64

Wykresy - rysunek 9.6 przedstawiają czasy wykonania benchmarku CG dla czterech klas problemu (A, B, S, W) w zależności od liczby wątków, dla czterech implementacji równoległych: **tbb**, **old_omp**, **new_omp** i **rust**.

Implementacja w języku Rust charakteryzuje się najwyższym współczynnikiem zmienności zarówno dla czasu wykonania, jak i dla wydajności MFLOPS. Mediana CV dla implementacji **rust** jest znacznie wyższa niż dla pozostałych implementacji, co wskazuje na niższą przewidywalność wydajności tej implementacji. Rozstęp międzykwartylowy jest również największy dla implementacji **rust**, co sugeruje większą dyspersję wyników pomiarów w porównaniu do innych wariantów.

Implementacja oparta na Intel tbb oraz starsza wersja implementacji OpenMP (old_omp) wykazują zbliżone charakterystyki zmienności, ze średnimi wartościami CV dla czasu wykonania oscylującymi wokół 0.05. Jednocześnie implementacja new_omp prezentuje najniższy medianowy współczynnik zmienności dla czasu wykonania, co sugeruje jej wysoką stabilność pomimo niższej wydajności bezwzględnej.

Interesującym aspektem jest obecność pojedynczych obserwacji odstających widocznych szczególnie dla implementacji new_omp i old_omp, które mogą wskazywać na sporadyczne zakłóczenia spowodowane czynnikami zewnętrznymi lub specyficznymi przypadkami brzegowymi w algorytmie.



Rys. 9.8: Porównanie wydajności benchmarku CG dla klas S, W, A, B względem liczby użytych wątków na platformie ARM64

Na wykresach na rysunku 9.8 zaprezentowano porównanie wydajności benchmarku CG mierzonej w MFLOPS (milionach operacji zmennoprzecinkowych na sekundę). Wydajność została przedstawiona jako funkcja liczby wątków (1-8) dla czterech implementacji równoległych.

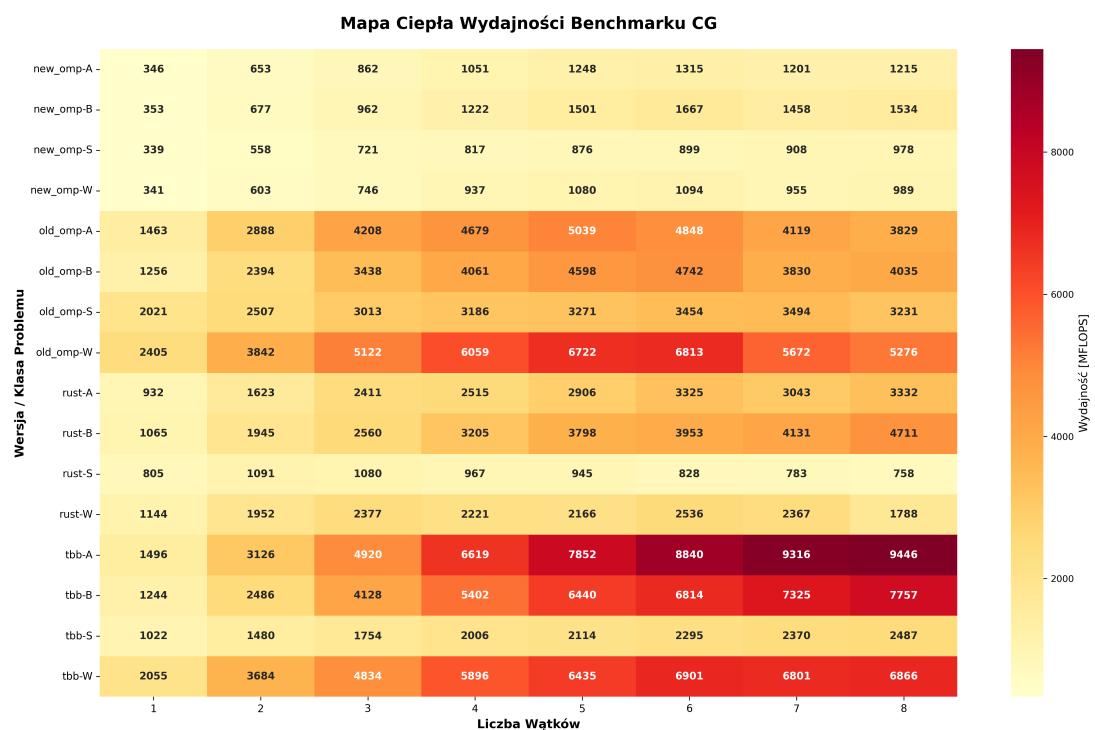
Implementacja bazująca na bibliotece tbb, wykazuje najwyższą wydajność obliczeniową w klasach problemów A, B i W. Dla klasy A osiąga ona maksymalną wydajność przekraczającą 9400 MFLOPS przy wykorzystaniu 7-8 wątków, wykazując przy tym dobre właściwości skalowania wraz ze wzrostem liczby jednostek wykonawczych. Szczególnie godne uwagi jest stabilne zwiększanie wydajności nawet przy wysokiej liczbie wątków, co sugeruje efektywne zarządzanie zadaniami i minimalizację narzutów synchronizacji w tej implementacji.

Implementacja wykorzystująca starszą wersję OpenMP (old_omp), prezentuje zróżnicowane charakterystyki wydajnościowe w zależności od klasy problemu. Interesującym przypadkiem jest jej wyjątkowo dobra wydajność w klasie problemu S, gdzie osiąga najwyższe wartości spośród wszystkich badanych implementacji, przekraczając 3500 MFLOPS. Jednocześnie w klasach A i B wykazuje ona tendencję do osiągania maksimum wydajności przy 4-5 wątkach, po czym następuje spadek efektywności przy dalszym zwiększaniu liczby jednostek wykonawczych.

9. Analiza wyników - programowanie równoległe

Implementacja w języku Rust charakteryzuje się umiarkowaną wydajnością z wyraźnymi ograniczeniami skalowania przy większej liczbie wątków. W klasie problemu B osiąga ona konkurencyjne wyniki względem implementacji `old_omp` przy 7-8 wątkach, jednak w pozostałych klasach pozostaje na niższym poziomie wydajności. W klasie S obserwowany jest nawet spadek wydajności powyżej 3 wątków, co wskazuje na potencjalne problemy z równoległą implementacją dla tego typu zadań.

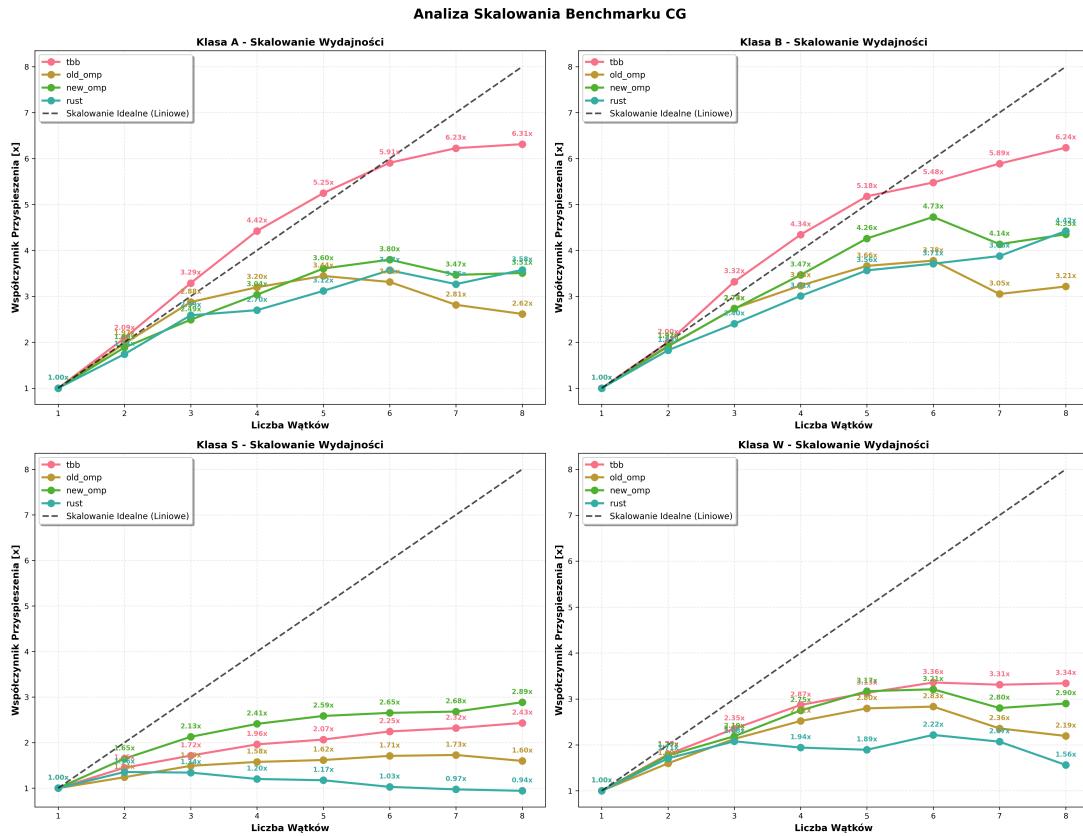
Implementacja wykorzystująca nowszą wersję OpenMP (`new_omp`), konsekwentnie wykazuje najniższe wartości wydajności we wszystkich klasach problemów, rzadko przekraczając 1500 MFLOPS nawet przy maksymalnej liczbie wątków. Charakterystycznym elementem jest jednak stosunkowo stabilny, choć ograniczony wzrost wydajności wraz ze zwiększaniem liczby jednostek wykonawczych.



Rys. 9.9: Mapa ciepła wydajności benchmarku CG dla klas S, W, A, B względem liczby użytych wątków na platformie ARM64

Mapa ciepła - rysunek 9.9 wydajności benchmarku CG dostarcza komplementarnego widoku na zachowanie wszystkich implementacji. Intensywność koloru odpowiada wydajności obliczeniowej wyrażonej w MFLOPS, co pozwala na szybką identyfikację obszarów optymalnej wydajności. Najwyższe wartości (reprezentowane ciemnoczerwonymi odcienniami) koncentrują się w obszarze implementacji tbb dla klas A, B i W przy wyższej liczbie wątków. Szczególnie widoczna jest równomierna poprawa wydajności tbb dla klasy A wraz ze wzrostem liczby wątków.

Mapa ciepła potwierdza również wyjątkową charakterystykę klasy S, gdzie najwyższe wartości osiągane są przez implementację `old_omp`, a nie tbb jak w pozostałych klasach. Dodatkowo uwydatniony jest gradient wydajności dla implementacji `rust` i `old_omp` w klasach A i W, gdzie wydajność najpierw rośnie, a następnie spada przy wyższych liczbach wątków.



Rys. 9.10: Analiza skalowania benchmarku CG dla klas S, W, A, B względem liczby użytych wątków na platformie ARM64

Powyzszy wykres - rysunek 9.10 przedstawia skalowanie wydajności benchmarku CG. Skalowanie wyrażone zostało za pomocą współczynnika przyspieszenia względem wykonania jednowątkowego i odniesione do skalowania idealnego (liniowego).

Porównanie efektywności skalowania implementacji

Wśród badanych implementacji najlepsze właściwości skalowania wykazuje **tbb**, szczególnie w klasach A i B, gdzie osiąga przyspieszenie przekraczające 6x przy maksymalnej liczbie wątków. W klasie A zaobserwowano nawet zjawisko superliniowego przyspieszenia (powyżej 6x), co może wynikać z lepszej lokalności pamięci podręcznej.

new_omp prezentuje umiarkowane, ale stabilne skalowanie. W klasie B osiąga maksymalnie 4,7x przy 6 wątkach, a w innych klasach - niższe wartości. Jej charakterystyczną cechą jest brak gwałtownych spadków wydajności przy rosnącej liczbie wątków.

rust wykazuje silne zróżnicowanie: w klasach A i B uzyskuje do 3,5x przy 8 wątkach, ale w klasie S dochodzi do degradacji - przyspieszenie spada poniżej 1.0, co świadczy o poważnych problemach z równoległością przy małych problemach.

old_omp wypada najsłabiej. Dla większości klas przyspieszenie przestaje rosnąć po 3-4 wątkach. W klasie A przy 8 wątkach osiąga tylko 2,5x, co sugeruje nieefektywność synchronizacji.

Wpływ klasy problemu

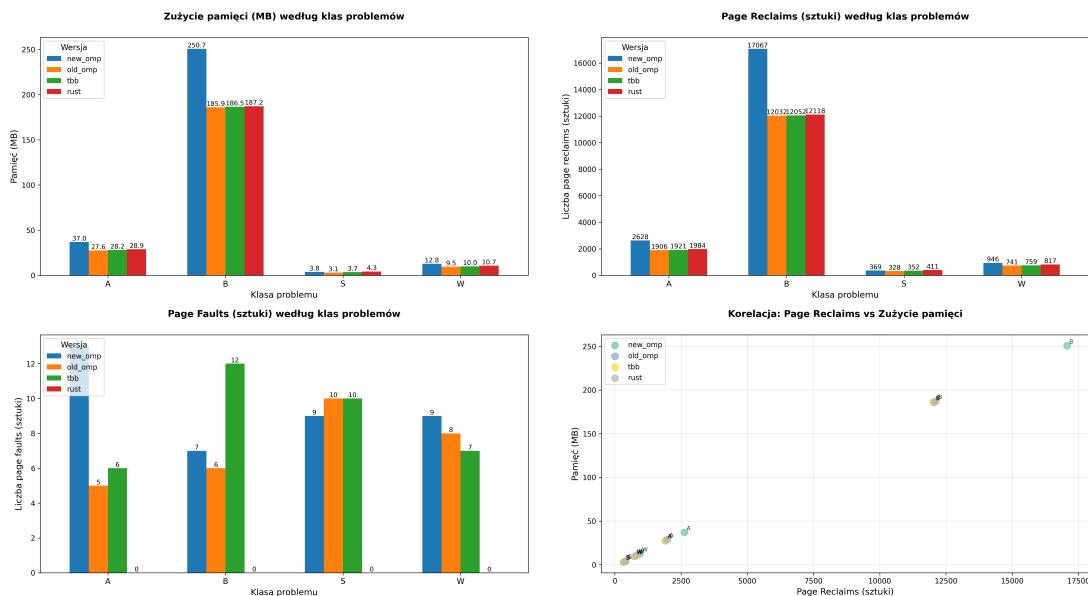
Najlepsze skalowanie występuje w klasach A i B, gdzie implementacja **tbb** osiąga przyspieszenia ponad 6x. Klasy te są najbardziej podatne na równoległość.

Klasa S wypada najgorzej - żadna implementacja nie przekracza 3x przyspieszenia, co sugeruje ograniczenia wynikające z wysokiego stosunku komunikacji do obliczeń.

9. Analiza wyników - programowanie równoległe

Klasa W wykazuje umiarkowane skalowanie, do ok. 3,5x, przy czym do 3-4 wątków wydajność implementacji jest podobna, po czym zaczynają się różnić.

9.1.3. Wyniki profilowania wydajności - platforma ARM64



Rys. 9.11: Profilowanie wydajności benchmarku CG dla klas S, W, A, B względem liczby użytych wątków na platformie ARM64

Zużycie pamięci (MB)

Na pierwszym wykresie - rysunek 9.11 (lewy górnego róga) przedstawiono zużycie pamięci operacyjnej (w MB) dla czterech wersji implementacji algorytmu:

- W klasie problemu B występuje najwyższe zużycie pamięci we wszystkich wersjach. Szczególnie zauważalny jest wynik dla `new_omp` (250,7 MB), który znacznie przekracza wartości dla pozostałych wersji (około 185-187 MB).
- W klasie A `new_omp` również zużywa najwięcej pamięci (37 MB), natomiast pozostałe wersje wykazują zbliżony i niższy poziom zużycia (około 27-29 MB).
- W klasach S i W różnice są mniej wyraźne, jednak nadal `new_omp` wykazuje największe zużycie pamięci.
- Wersja `rust` generalnie charakteryzuje się najmniejszym lub jednym z najmniejszych zużyciami pamięci w większości klas problemów.

Liczba zwolnionych stron pamięci

Drugi wykres - rysunek 9.11 (prawy górnego róga) ilustruje liczbę oddanych stron pamięci (ang. *page reclaim*), czyli sytuacji, w których system operacyjny odzyskuje strony pamięci.

- Najwięcej zwolnionych stron pamięci występuje w klasie B dla wersji `new_omp` (17067), co koresponduje z jej wysokim zużyciem pamięci.
- W pozostałych wersjach liczba zwolnionych stron w klasie B jest znacznie niższa (około 12000-12100).
- W klasie A `new_omp` ponownie osiąga najwyższy wynik (2628), przy niższych wartościach pozostałych wersji (około 1900).
- W klasach S i W różnice są mniej zauważalne, choć `new_omp` nadal wykazuje wyższe wartości.

Odwołania do nieobecnych stron

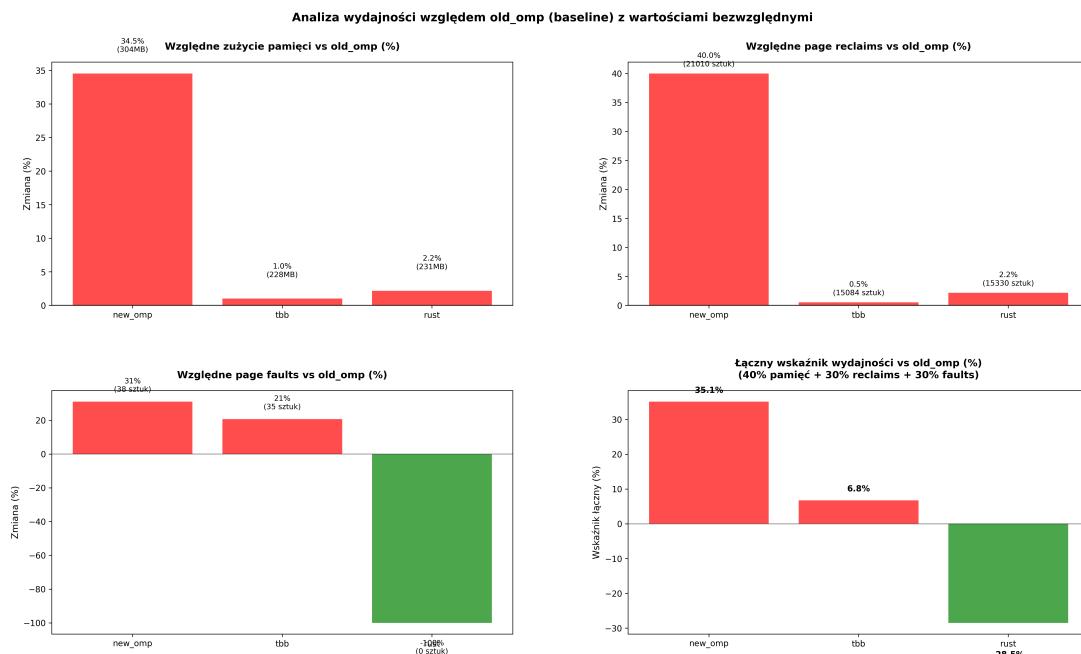
Trzeci wykres - rysunek 9.11 (lewy dolny róg) przedstawia odwołania do nieobecnych stron-liczbę (ang. *page fault*) - czyli sytuacji, w których wymagany fragment pamięci nie znajduje się aktualnie w RAM-ie.

- W klasie A `new_omp` wykazuje najwyższą liczbę błędnych stron (13), zaś `rust` nie generuje żadnych błędów.
- W klasie B najwięcej błędów generuje `tbb` (12), podczas gdy `rust` ponownie nie wykazuje żadnych.
- W klasach S i W `rust` również nie generuje błędów stron, a inne wersje wykazują umiarkowane wartości (7-10).
- Wyniki te sugerują bardzo efektywną gospodarkę pamięciową w wersji `rust`.

Korelacja: Liczba zwolnionych stron pamięci a zużycie pamięci

Ostatni wykres - rysunek 9.11 (prawy dolny róg) ilustruje zależność pomiędzy liczbą zwolnionych stron pamięci a zużyciem pamięci.

- Widoczna jest silna dodatnia korelacja - im większe zużycie pamięci, tym większa liczba zwolnionych stron pamięci. Najwyraźniejszym punktem odniesienia jest wersja `new_omp` dla klasy B, która dominuje pod względem obu metryk.
- Punkty reprezentujące wersję `rust` znajdują się w lewej dolnej części wykresu, wskazując na niskie zużycie pamięci i małą liczbę zwolnionych stron.



Rys. 9.12: Analiza wydajności względem `old_omp` (punkt odniesienia) z wartościami bezwzględnymi na platformie ARM64

Na wykresie - rysunek 9.12 szczegółowa analiza czterech wykresów przedstawiających względną wydajność trzech wersji implementacji (`new_omp`, `tbb`, `rust`) względem wersji bazowej `old_omp`. Analiza dotyczy zużycia pamięci, liczby zwolnionych stron pamięci, odwołań do nieobecnych stron oraz skumulowanego wskaźnika efektywności.

Względne zużycie pamięci na tle `old_omp`

Pierwszy wykres - rysunek 9.12 (lewy górny róg) ukazuje procentową zmianę w zużyciu pamięci operacyjnej względem implementacji referencyjnej. Największy wzrost odnotowano dla `new_omp`, gdzie zużycie pamięci wzrosło o 34,5%, co przekłada się na dodatkowe 304 MB. Zmiany dla `tbb` i `rust` były marginalne i wyniosły odpowiednio 1,0% (228 MB) oraz 2,2% (231 MB), co sugeruje ich większą efektywność w kontekście gospodarowania pamięcią.

Względne odzyskiwanie stron pamięci na tle `old_omp`

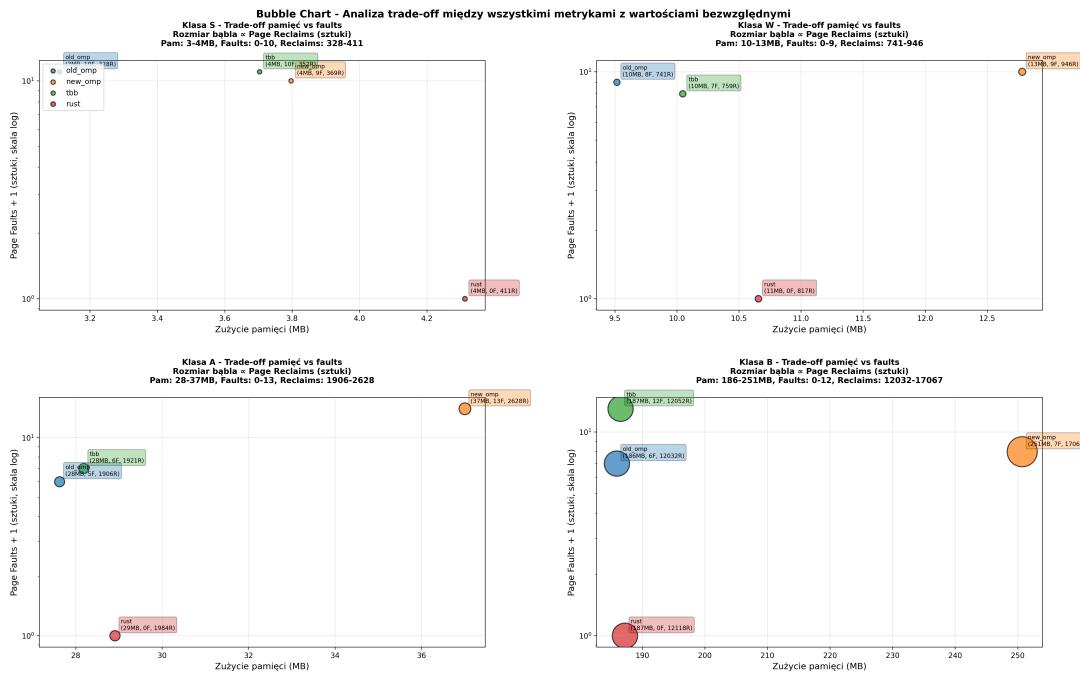
Drugi wykres - rysunek 9.12 (prawy górny róg) przedstawia zmiany w liczbie odzyskanych stron pamięci. Implementacja `new_omp` wykazała aż 40% wzrost (21010 stron), co może świadczyć o intensywnym zarządzaniu pamięcią w trakcie wykonywania programu. Dla `tbb` i `rust` zmiany były minimalne (odpowiednio 0,5% i 2,2%), co może być interpretowane jako korzystny efekt optymalizacji dostępu do pamięci.

Względne błędy stron na tle `old_omp`

Na trzecim wykresie - rysunek 9.12 (lewy dolny róg) obserwujemy względną liczbę błędów stron. `new_omp` i `tbb` odnotowały wzrost liczby błędów stron odpowiednio o 31% (38 sztuk) i 2% (35 sztuk), co może wskazywać na mniej wydajne wykorzystanie pamięci w porównaniu z `old_omp`. Z kolei `rust` jako jedyna implementacja odnotowała całkowitą eliminację błędów stron (-100%, 0 sztuk), co potwierdza wyjątkowo skuteczną kontrolę pamięci operacyjnej - mechanizmy własności oraz pożyczania.

Łączny wskaźnik wydajności na tle `old_omp`

Na ostatnim wykresie - rysunek 9.12 (prawy dolny róg) przedstawiono zagregowany wskaźnik wydajności, będący ważoną sumą trzech poprzednich metryk: 40% udziału zużycia pamięci, 30% udziału odzyskiwania stron oraz 30% udziału błędów stron. Najwyższy wskaźnik (35,1%) ponownie osiąga `new_omp`, co oznacza wyraźnie wyższą konsumpcję zasobów w porównaniu do referencji. `tbb` wykazuje niewielki wzrost (6,8%), co czyni go względnie neutralnym względem `old_omp`. `rust` natomiast charakteryzuje się łącznym wskaźnikiem na poziomie -28,5%, co czyni go najbardziej efektywną implementacją w ujęciu ogólnym.



Rys. 9.13: Kompromisy (ang. *trade-off*) pomiędzy zużyciem pamięci a błędami stron pamięci, z uwzględnieniem liczby odzyskanych stron jako trzeciej zmiennej reprezentowanej przez rozmiar bąbla na platformie ARM64

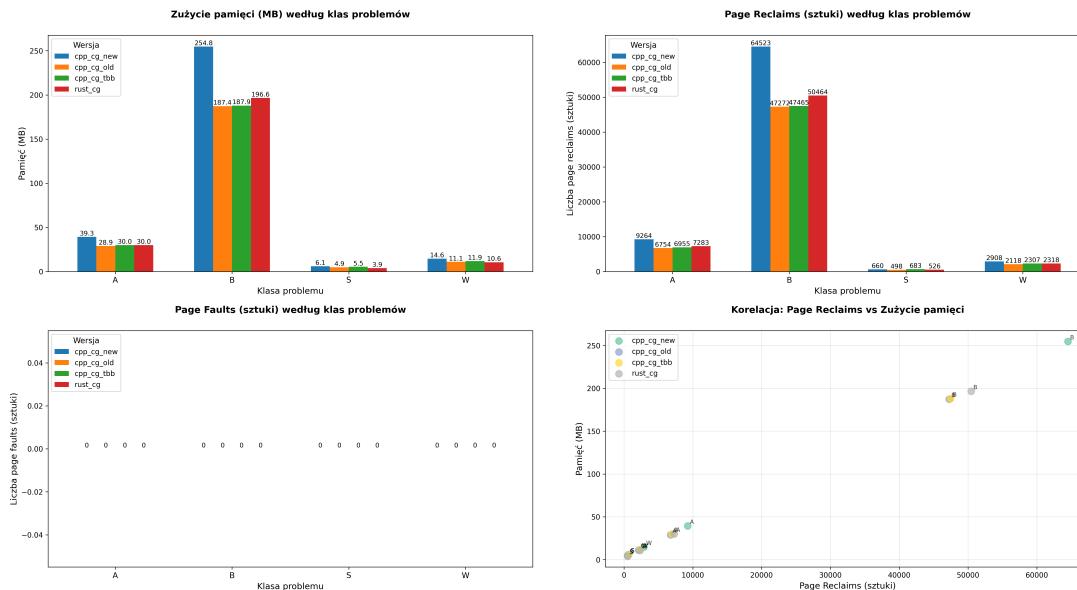
W celu całościowej oceny efektywności pamięciowej badanych implementacji, opracowano wykresy bąbelkowe - rysunek 9.13, które ukazują kompromisy między dwiema kluczowymi metrykami: zużyciem pamięci oraz liczbą błędów stron. Wykresy te wzbogacono o trzeci wymiar - liczbę odzyskanych stron pamięci, która została zakodowana poprzez rozmiar bąbla. Wszystkie wartości przedstawiono w postaci bezwzględnej, przy czym oś Y jest skalowana logarytmicznie w celu lepszego rozróżnienia niewielkich wartości.

Każdy z czterech wykresów odpowiada innej klasie obciążenia pamięciowego (S, W, A, B), przy czym wspólnym celem ich analizy jest uchwycenie relacji pomiędzy wzrostem zużycia pamięci a pogorszeniem lub poprawą stabilności działania (mierzonych błędami stron) oraz intensywnością operacji odzyskiwania stron.

W skali globalnej można zaobserwować, że implementacja **rust** wielokrotnie wypada korzystnie - cechuje się niskim zużyciem pamięci, zerową liczbą błędów stron w wielu przypadkach oraz stosunkowo niską liczbą zwolnień, co sugeruje wysoką efektywność i dobrą kontrolę nad zarządzaniem pamięcią. Z kolei **new_omp**, mimo czasem lepszych wyników obliczeniowych, wykazuje największe zużycie pamięci oraz relatywnie wysoką liczbę błędów stron i odzysków pamięci, co może oznaczać znaczną presję na system zarządzania pamięcią operacyjną.

Implementacja **tbb** prezentuje wyniki pośrednie, w niektórych przypadkach zbliżone do **old_omp**, ale przy lepszym wykorzystaniu zasobów - co czyni ją kompromisowym rozwiązaniem o umiarkowanej efektywności. -

9.1.4. Wyniki profilowania wydajności - platforma x86_64



Rys. 9.14: Profilowanie wydajności benchmarku CG dla klas S, W, A, B względem liczby użytych wątków na platformie x86_64

Zużycie pamięci (MB)

Z wykresu - rysunek 9.6 zużycia pamięci wynika, że implementacja new_omp jest zdecydowanie najbardziej pamięciożerna w klasie B, gdzie osiąga wartość 254,8 MB — to istotnie więcej niż pozostałe implementacje, które utrzymują się w zakresie 187-197 MB. Dla pozostałych klas (A, S, W) różnice pomiędzy implementacjami są znikome i mieszczą się w przedziale od 3,9 MB do 39,3 MB.

Zależność zużycia pamięci od klasy problemu jest zgodna z oczekiwaniemi: większa klasa (większy problem obliczeniowy) wiąże się z wyższym zużyciem zasobów. Jednak istotna różnica dla new_omp w klasie B sugeruje potencjalne niedoskonałości w zarządzaniu pamięcią lub większy narzut alokacyjny tej wersji.

Operacje zwolnień stron

Podobnie jak w przypadku zużycia pamięci, new_omp generuje najwyższe liczby operacji zwolnień stron — w klasie B sięgają one ponad 64 tys. Operacje te wskazują na intensywną aktywność zarządzania pamięcią przez system operacyjny i mogą prowadzić do pogorszenia wydajności w warunkach ograniczonych zasobów. Pozostałe implementacje generują wyraźnie mniej takich operacji (ok. 47-50 tys. dla klasy B).

W klasach A, S i W różnice między implementacjami są mniej znaczące. Warto jednak zwrócić uwagę, że rust we wszystkich klasach cechuje się stosunkowo niskim poziomem zwolnień, co może świadczyć o bardziej efektywnym wykorzystaniu pamięci przez alokator w środowisku Rust.

Błędy stron pamięci

Wszystkie implementacje, niezależnie od klasy problemu, wykazują zerowy poziom błędów stron. Taki wynik jest pozytywny i świadczy o prawidłowej alokacji oraz dostępie do pamięci bez potrzeby kosztownego przenoszenia stron pamięci z dysku do RAM-u. To również sugeruje, że żadne z testów nie spowodowały przeciążenia systemu w zakresie dostępnej pamięci fizycznej.

Korelacja: liczba zwolnionych stron pamięci a zużycie pamięci

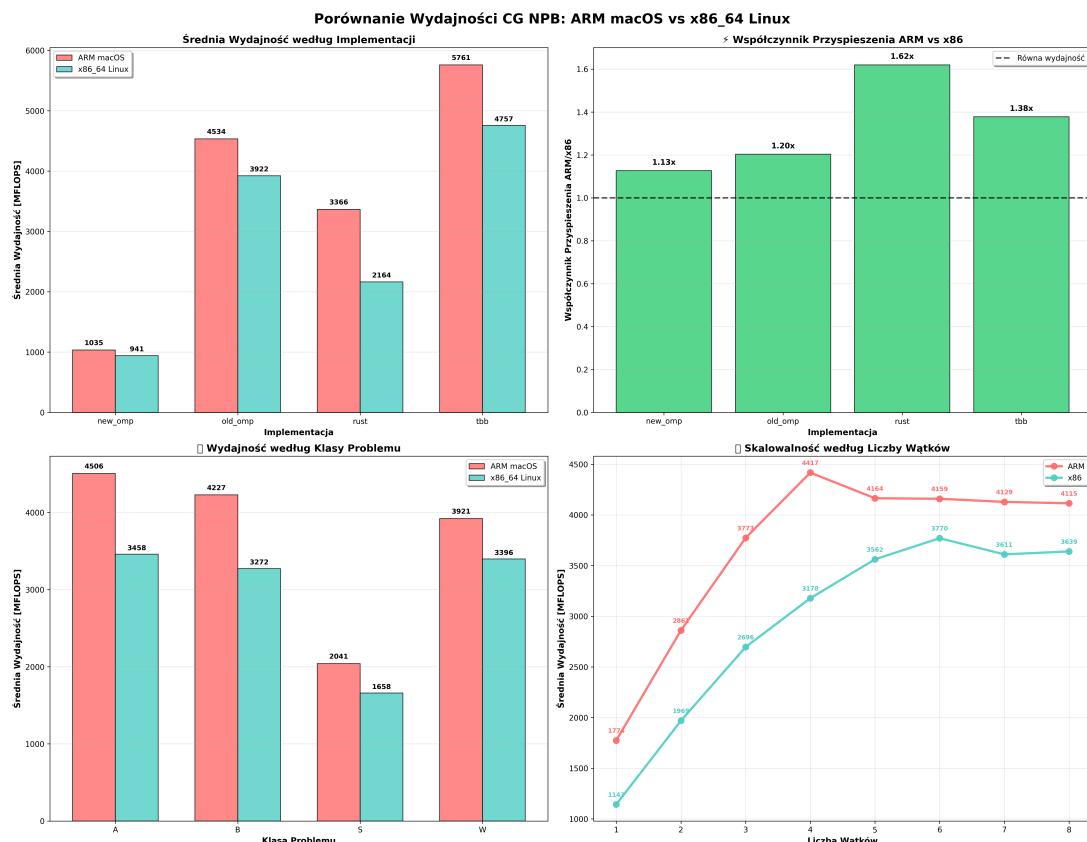
Wykres rozrzutu przedstawia silną dodatnią korelację pomiędzy liczbą operacji zwolnień stron a zużyciem pamięci. Punkty odpowiadające klasie B znajdują się wyraźnie w prawym górnym rogu wykresu, potwierdzając, że najbardziej zasobozerna klasa problemu generuje również najwięcej interakcji z systemowym menedżerem pamięci. Punkty dla klas A, S i W rozmieszczone są blisko siebie w lewym dolnym obszarze, co wskazuje na relatywnie stabilne i niskie zużycie pamięci.

Nota wyjaśniająca

Ze względu na brak błędów stron nie zamieszczono pozostałych wykresów, które byłyby oparte na tej metryce. Wykresy te nie miałyby sensu, ponieważ brak błędów stron oznacza, że wszystkie operacje pamięciowe były realizowane w ramach dostępnej pamięci fizycznej bez konieczności angażowania mechanizmów swapowania lub przenoszenia stron z dysku.

9.1.5. Porównanie pomiędzy platformami

Średnia wydajność i współczynnik przyspieszenia



Rys. 9.15: Porównanie średniej wydajności benchmarku CG dla platform ARM64 i x86_64

Na podstawie wykresów - rysunek 9.15 uśrednionej wydajności (wyrażonej w MFLOPS) można zaobserwować, że implementacje benchmarku CG uzyskują zróżnicowane rezultaty na obu platformach. Najwyższą wydajność osiąga implementacja tbb na obu architekturach, przy czym wynik dla ARM (5761 MFLOPS) jest wyższy niż dla x86 (4757 MFLOPS). Implementacja rust również wypada lepiej na platformie ARM, uzyskując przewagę nad x86 rzędu 62% (współczynnik przyspieszenia ARM/x86 = 1.62).

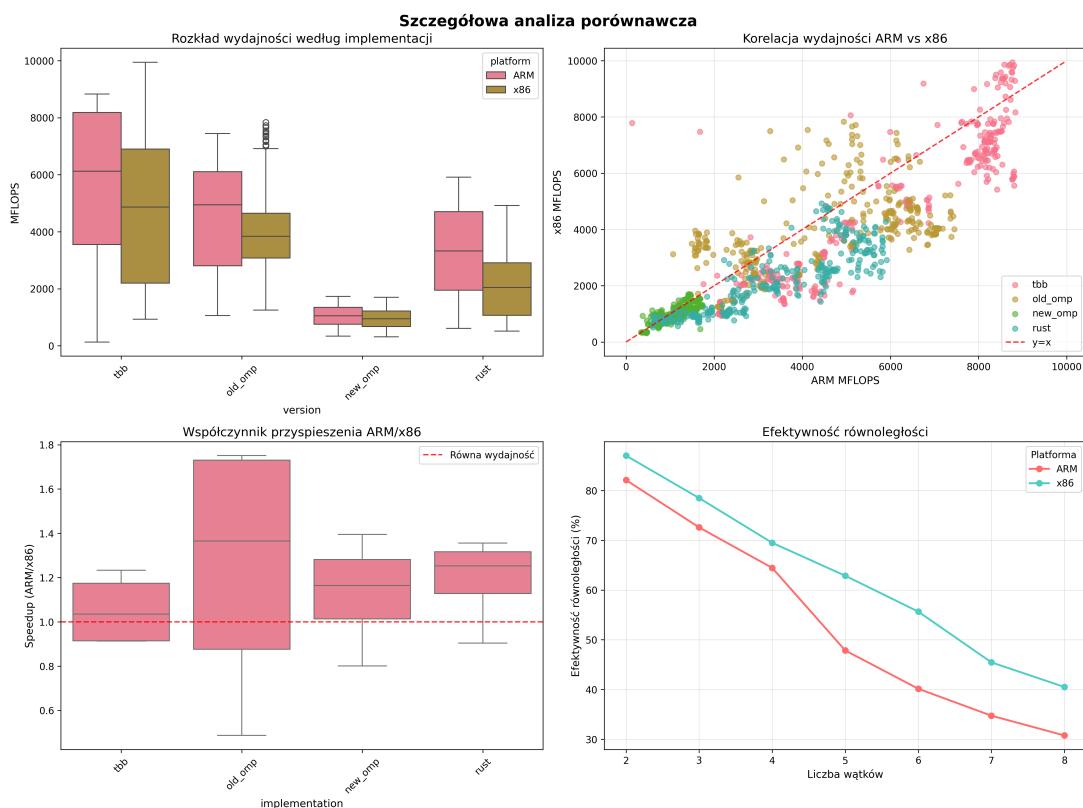
Z kolei `old_omp` i `new_omp` wykazują mniejsze różnice pomiędzy platformami, jednak nadal na korzyść ARM (współczynnik przyspieszenia odpowiednio 1.20 i 1.13). Pokazuje to, że układy ARM, mimo mniejszej popularności w obliczeniach naukowych, mogą zapewniać porównywalną lub wyższą wydajność w zadaniach równoległych.

Wydajność względem klasy problemu

Dla wszystkich klas problemu platforma ARM uzyskuje wyższe wartości MFLOPS w porównaniu do x86, przy czym różnice te są szczególnie widoczne w klasach A i B — odpowiednio 4506 a 3458 oraz 4227 a 3272 MFLOPS. Mniejsze różnice występują w klasach S i W, co może być związane z ograniczoną skalowalnością w mniejszych problemach oraz różnicami w architekturze pamięci podręcznej.

Skalowanie według liczby wątków

Analiza wykresu skalowalności pokazuje, że ARM osiąga wyższe wartości MFLOPS w niemal każdej konfiguracji wątkowej. Dla 4 wątków uzyskuje wartość 4417 MFLOPS, przewyższając tym samym x86 (3179 MFLOPS). Mimo że przy wzroście liczby wątków następuje spłaszczenie wykresów dla obu platform, ARM utrzymuje przewagę aż do 8 wątków, co potwierdza dobre skalowanie procesora ARM w tej klasie obliczeń.



Rys. 9.16: Szczegółowa analiza wydajności benchmarku CG dla platform ARM64 i x86_64

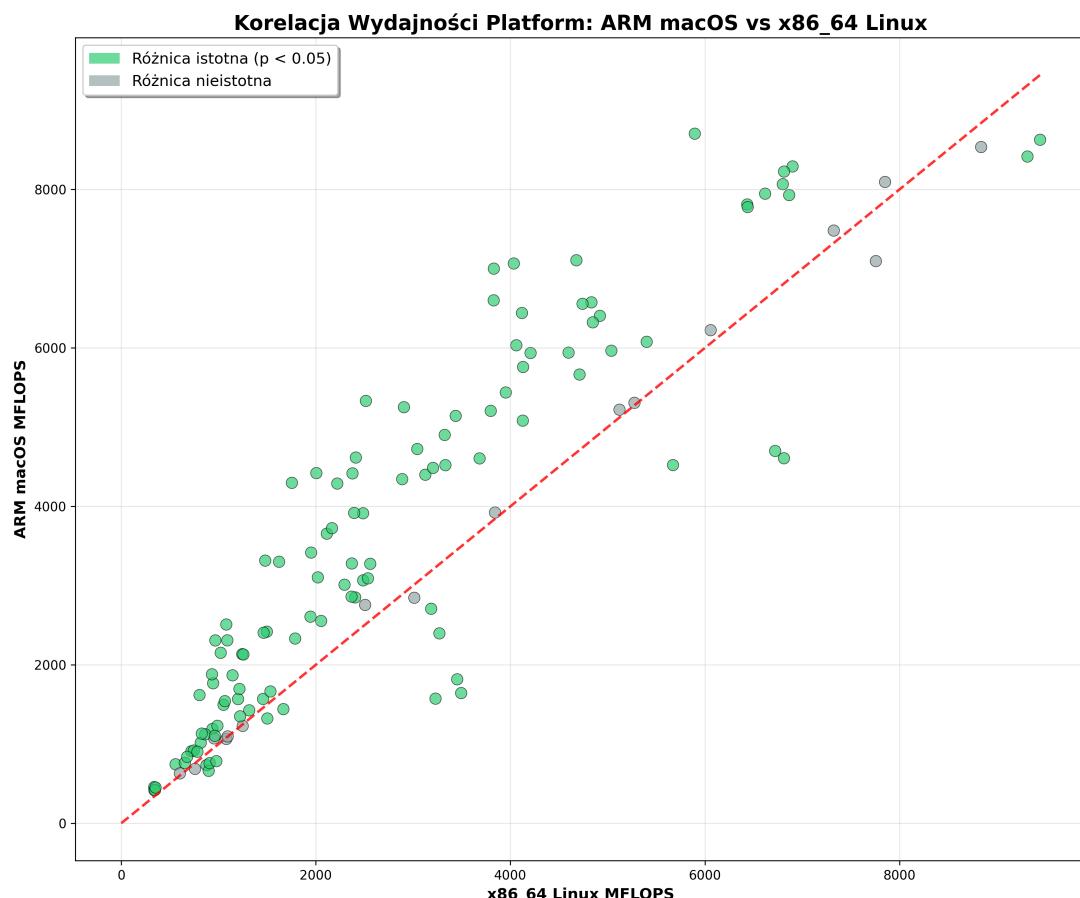
Rozkład wydajności i korelacja

Wykres pudełkowy obrazujące rozkład wydajności według implementacji pokazują większą zmienność wyników na platformie ARM, szczególnie dla `tbb` i `old_omp`, co może świadczyć o bardziej dynamicznym zarządzaniu zasobami na poziomie systemowym (macOS). Mimo tej zmienności, wartości median są konsekwentnie wyższe dla ARM.

Wykres korelacyjny MFLOPS ARM a x86 ukazuje silną dodatnią korelację między wynikami na obu platformach, z większością punktów powyżej linii równej wydajności ($y = x$), co oznacza, że ARM zazwyczaj przewyższa x86 pod względem osiągów. W przypadku rust różnica jest szczególnie wyraźna.

Efektywność równoległości

Analiza efektywności równoległości (rozumianej jako stosunek osiągniętej wydajności do liczby wątków względem wydajności jednordzeniowej) wskazuje na wyraźnie wyższe wartości dla platformy x86 przy większej liczbie wątków. Podczas gdy ARM charakteryzuje się dobrą wydajnością bezwzględną, jego efektywność maleje szybciej wraz ze wzrostem liczby wątków, co może wynikać z wąskiego gardła na poziomie pamięci współdzielonej lub mniejszej liczby jednostek wykonawczych na wątek fizyczny.

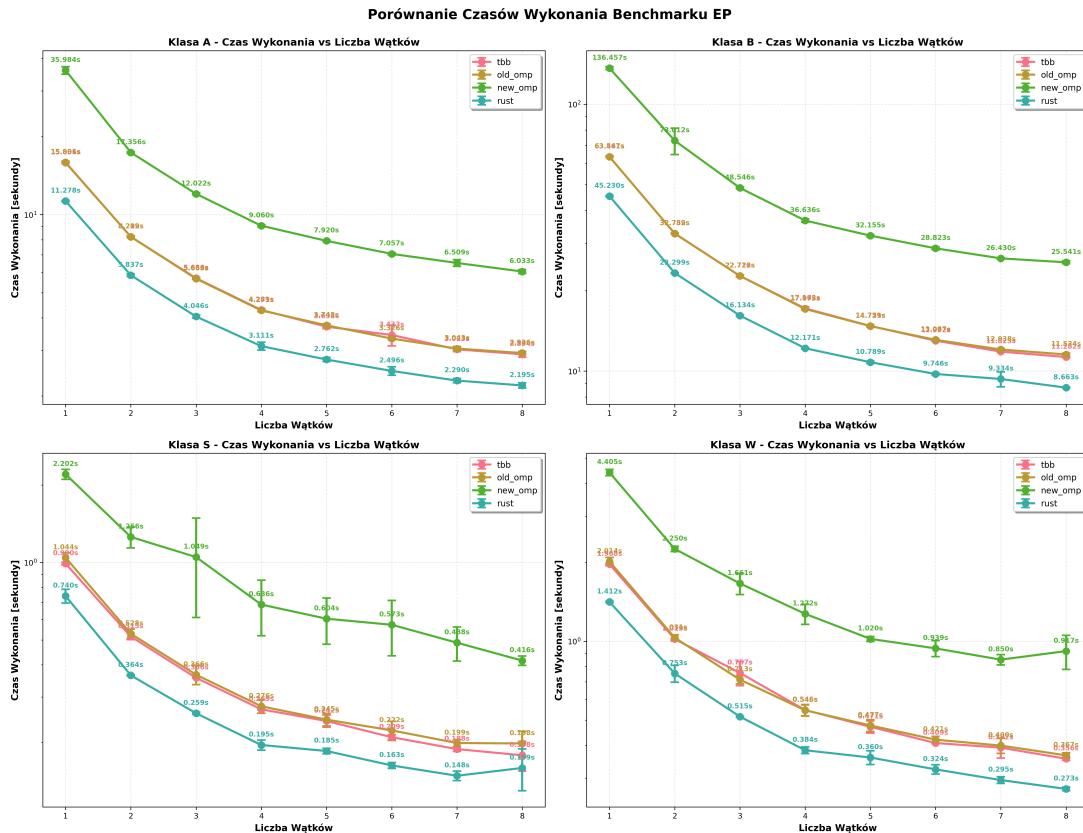


Rys. 9.17: Analiza istotności statystycznej benchmarku CG dla platform ARM64 i x86_64

Na wykresie - rysunek 9.17 zaprezentowano korelację wydajności z uwzględnieniem istotności statystycznej ($p < 0.05$). Zdecydowana większość punktów (zielone oznaczenia) wskazuje na istotne różnice wydajności między platformami. Dla kilku przypadków (oznaczenia szare) różnica nie była statystycznie znacząca. Potwierdza to tezę, że platforma ARM macOS w większości przypadków przewyższa x86_64 Linux, nie tylko nominalnie, ale również z punktu widzenia statystycznej wiarygodności wyników.

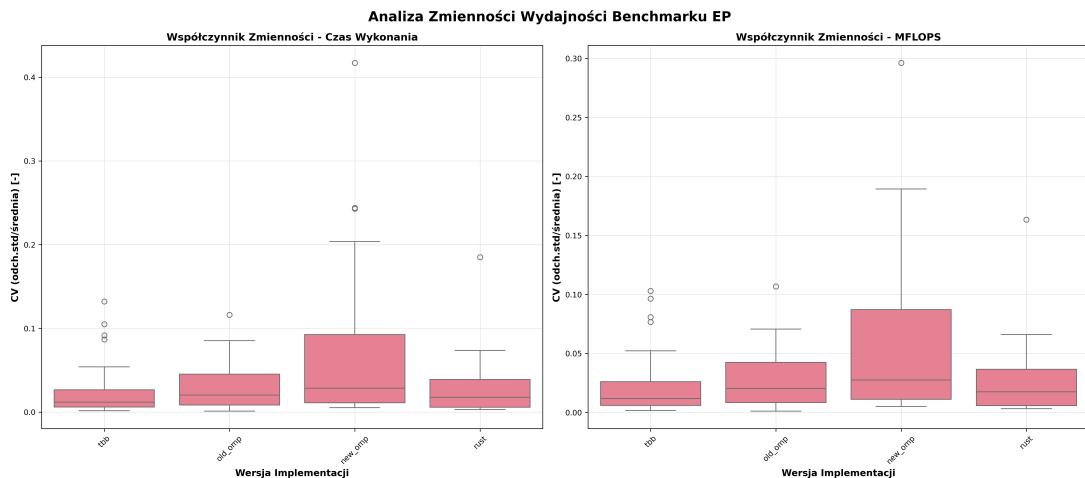
9.2. Benchmark EP

9.2.1. Wyniki benchmarków - platforma ARM64



Rys. 9.18: Porównanie czasów wykonania benchmarku EP dla klas S, W, A, B względem liczby użytych wątków

Na rysunku 9.18 zaprezentowano zestawienie czasów wykonania benchmarku EP dla czterech klas problemu: S, W, A oraz B, przy użyciu czterech różnych implementacji równoległości: TBB, OpenMP w wersji oryginalnej w stylu języka Fortran (old_omp), OpenMP w wersji nowszej (new_omp) oraz implementacji w języku Rust. Dla każdej z klas przedstawiono zależność czasu wykonania od liczby wątków (od 1 do 8). Wartości zostały zaprezentowane w skali logarytmicznej.



Rys. 9.19: Analiza zmienności czasów wykonania benchmarku EP dla klas S, W, A, B względem liczby użytych wątków

Najlepsze rezultaty czasowe, niezależnie od klasy problemu, osiąga implementacja z użyciem języka Rust, co może świadczyć o wysokiej wydajności systemu wykonawczego tej technologii oraz efektywnej optymalizacji niskopoziomowej. `rust` wykazuje też największą stabilność czasową przy wzroście liczby wątków, co może sugerować niskie koszty zarządzania wątkami.

Zarówno `old_omp`, jak i biblioteka TBB wykazują porównywalną wydajność w większości przypadków, przy czym TBB w klasach mniejszych (S, W) może osiągać nieco lepsze rezultaty, podczas gdy w klasach większych (A, B) ich wyniki się zbliżają. `new_omp` cechuje się natomiast najgorszą skalowalnością i najdłuższymi czasami wykonania, co może wskazywać na mniej efektywną implementację zarządzania zadaniami równoległymi lub wyższy narzut synchronizacji.

W przypadku klasy B, ze względu na większą złożoność obliczeniową, czasy wykonania są znacznie dłuższe, a różnice pomiędzy bibliotekami bardziej wyraźne. Zastosowanie skali logarytmicznej w tej klasie dodatkowo uwypukla przewagę rozwiązań o lepszej skalowalności. Warto również zauważyć, że dla większej liczby wątków (6-8) część implementacji przestaje zyskiwać znacząco na wydajności.

Zmienność czasu wykonania

Na podstawie lewego wykresu - rysunek 9.19 można zauważyć, że najniższą zmiennością czasów wykonania charakteryzują się implementacje oparte na bibliotekach TBB oraz Rust, co świadczy o ich dużej powtarzalności i deterministycznym charakterze działania. Mediany współczynnika zmienności dla tych implementacji pozostają na bardzo niskim poziomie, a obserwowane wartości odstające są rzadkie i relatywnie niewielkie.

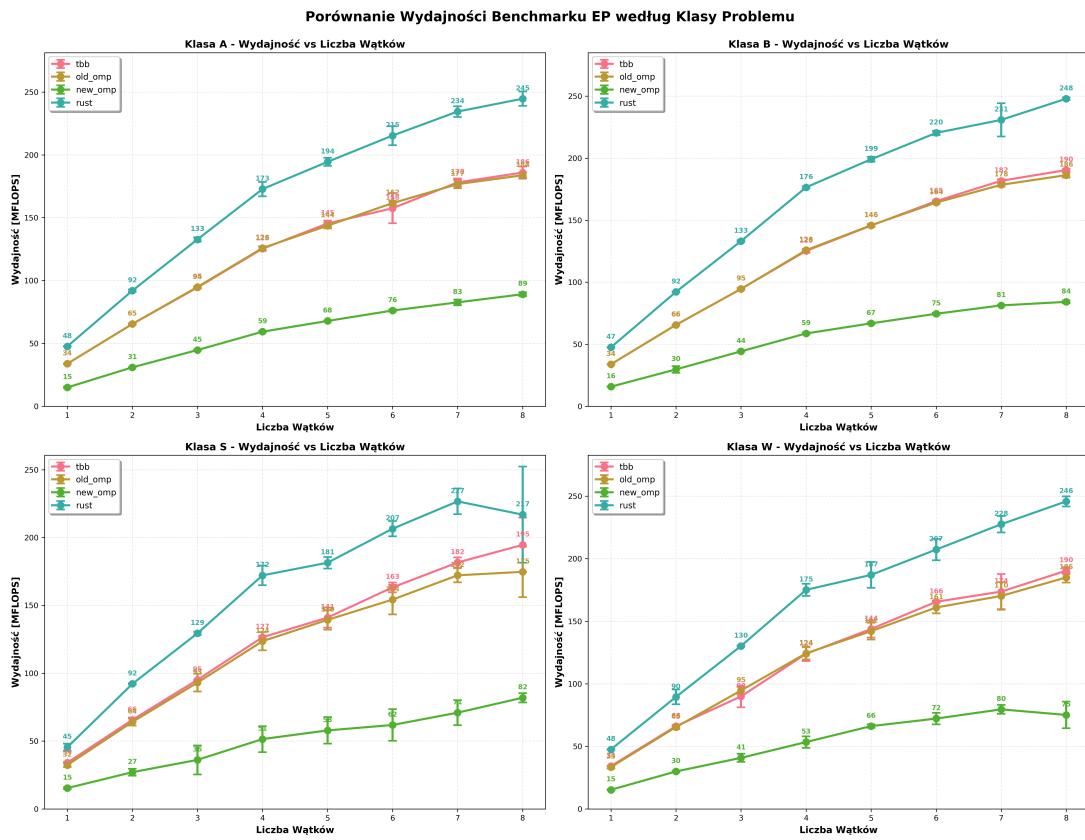
Oryginalna wersja OpenMP również wykazuje dobrą stabilność, choć z nieco większą rozpiętością wyników niż TBB i Rust. Natomiast nowa wersja OpenMP prezentuje największą zmienność. Występowanie licznych wartości odstających oraz szeroki rozrzut wyników podkreśla potencjalną niestabilność tej wersji w testowanych warunkach.

Zmienność MFLOPS

Podobne zależności można zaobserwować na prawym wykresie - rysunek 9.19, przedstawiającym zmienność wartości MFLOPS. Tutaj również TBB oraz Rust wyróżniają się najmniejszymi wartościami współczynnika zmienności, co potwierdza ich przewidywalność pod względem wydajności obliczeniowej. OpenMP w starszej wersji (`old_omp`) cechuje się umiarkowaną

9. Analiza wyników - programowanie równoległe

zmiennością, natomiast nowa wersja OpenMP (new_omp) ponownie odznacza się największą rozpiętością oraz medianą współczynnika zmienności, co może mieć negatywne implikacje dla zastosowań wymagających stabilnej wydajności.



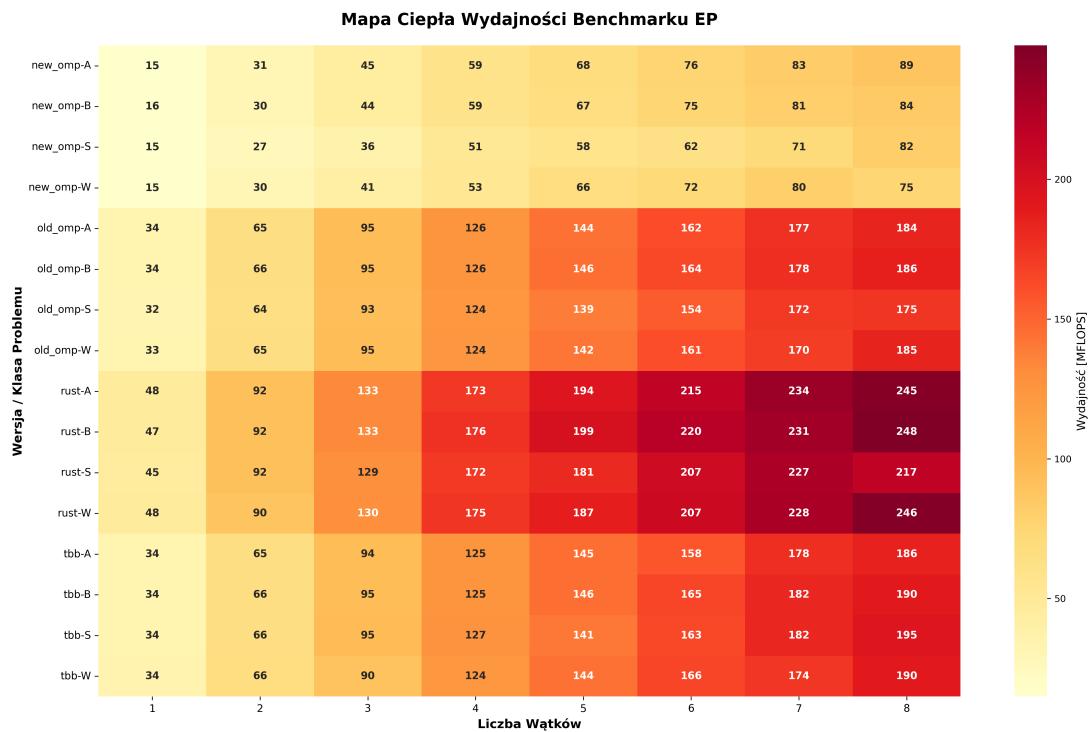
Rys. 9.20: Porównanie wydajności benchmarku EP dla klas S, W, A, B względem liczby użytych wątków

Na wykresach na rysunku 9.20 zaprezentowano porównanie wydajności benchmarku EP mierzonej w MFLOPS (milionach operacji zmiennoprzecinkowych na sekundę). Wydajność została przedstawiona jako funkcja liczby wątków (1-8) dla czterech implementacji równoległych.

Implementacja w języku Rust konsekwentnie osiąga najwyższe wartości MFLOPS we wszystkich klasach problemu i dla każdej liczby wątków, co świadczy o bardzo efektywnym zarządzaniu równoległością oraz niskim narzucie wykonawczym. Warto również zauważyć, że rust uzyskuje szczególnie imponujące wyniki dla większej liczby wątków (6-8), gdzie inne implementacje wykazują tendencję do spowolnienia przyrostu wydajności.

Zarówno biblioteka TBB, jak i starsza wersja OpenMP osiągają zbliżoną wydajność i dobrą skalowalność. W większości przypadków wartości MFLOPS dla tych dwóch rozwiązań są porównywalne, przy czym TBB niekiedy uzyskuje nieco lepsze rezultaty, szczególnie dla klas S i W.

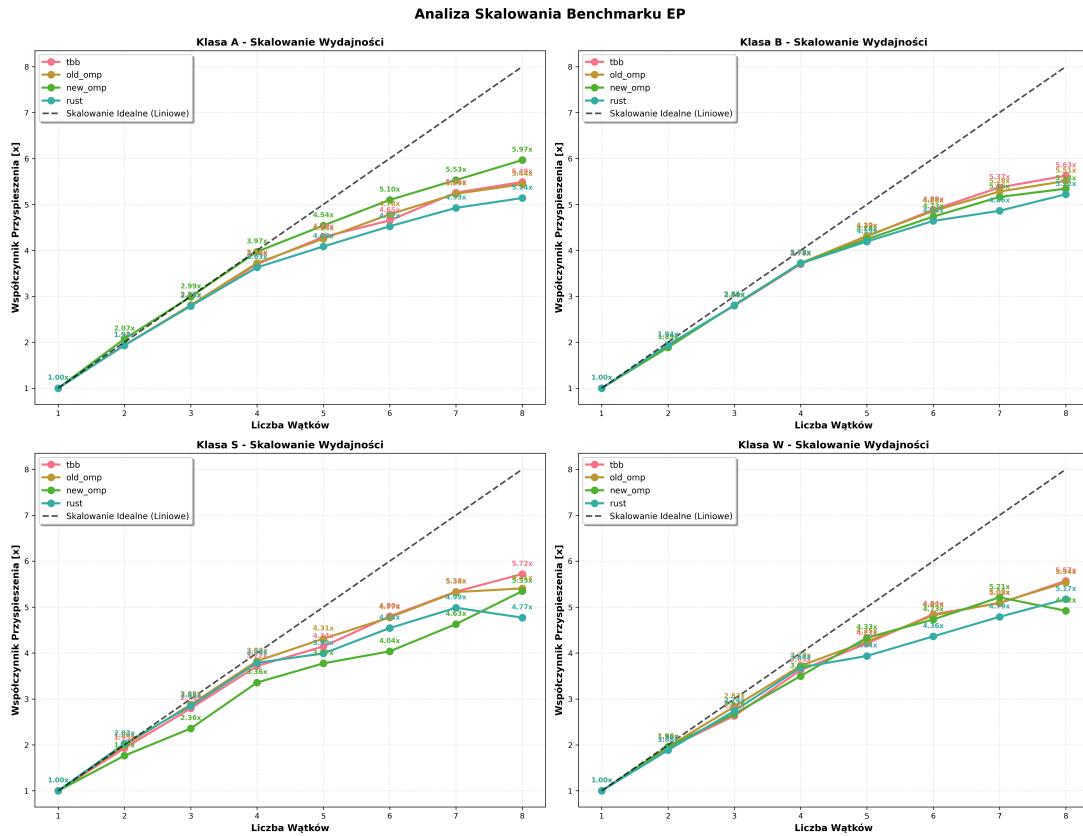
Z kolei implementacja oparta na nowej wersji OpenMP wykazuje wyraźnie niższą wydajność w porównaniu z pozostałymi rozwiązaniami. Różnice te są szczególnie widoczne przy większej liczbie wątków, gdzie wzrost MFLOPS jest bardziej spłaszczyony, co może wskazywać na ograniczenia w mechanizmach planowania zadań lub zwiększyony narzut synchronizacji.



Rys. 9.21: Mapa ciepła wydajności benchmarku EP dla klas S, W, A, B względem liczby użytych wątków

Powyższa mapa cieplna - rysunek 9.21 przedstawia wydajność (w MFLOPS). Wydajność została przedstawiona w zależności od liczby użytych wątków. Odcienie koloru od żółtego do ciemnoczerwonego wskazują na wzrost wydajności.

Mapa ciepła jednoznacznie potwierdza przewagę implementacji **rust** w każdym układzie: dla wszystkich klas problemu i liczby wątków. Wartości MFLOPS przekraczające 240 dla 8 wątków (np. 248 MFLOPS dla klasy B) są wyraźnie wyższe od wyników pozostałych bibliotek, w których wartości maksymalne oscylują w okolicach 185-195 MFLOPS.



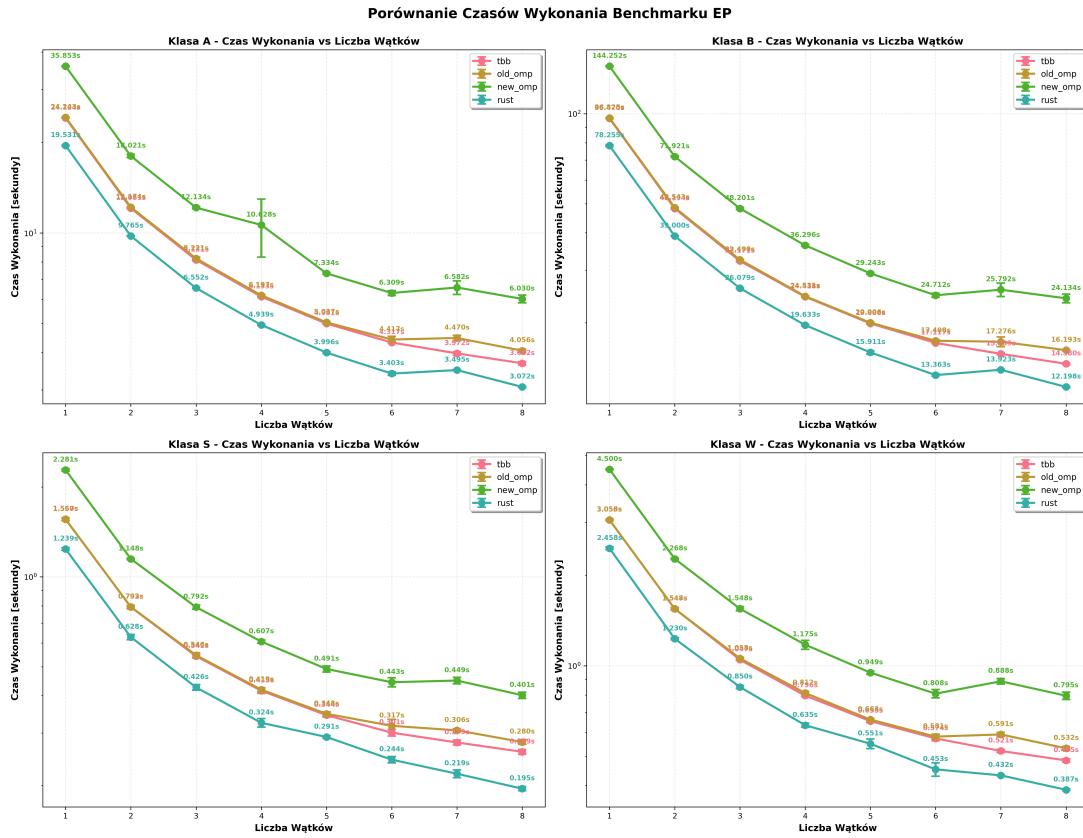
Rys. 9.22: Analiza skalowania benchmarku EP dla klas S, W, A, B względem liczby użytych wątków

Powyższy wykres - rysunek 9.22 przedstawia skalowanie wydajności benchmarku EP. Skalowanie wyrażone zostało za pomocą współczynnika przyspieszenia względem wykonania jednowątkowego i odniesione do skalowania idealnego (liniowego).

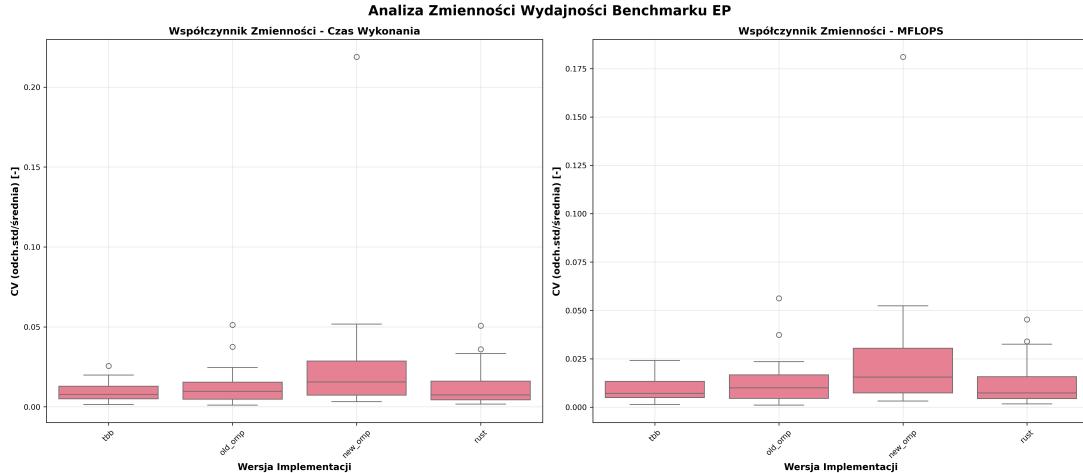
Spośród analizowanych implementacji **tbb** konsekwentnie osiąga najwyższe przyspieszenie we wszystkich klasach, najbardziej zbliżając się do idealnej skalowalności liniowej. Na drugim miejscu plasuje się **old_omp**, które wykazuje stabilne, choć nieco niższe wyniki w porównaniu do **tbb**. **New_omp** uzyskuje przyspieszenie niższe od **old_omp**, ale wyższe od **rust**, która w każdej klasie osiąga najwyższe wartości. Przykładowo, dla ośmiu wątków w klasie A **tbb** osiąga przyspieszenie rzędu 5,97x, podczas gdy **rust** ogranicza się do 5,10x, co ilustruje wyraźną różnicę w wydajności.

Różnice między implementacjami stają się bardziej wyraźne przy większej liczbie wątków, co wskazuje na lepszą zdolność **tbb** i **old_omp** do efektywnego wykorzystania równoległości w benchmarku EP. Subliniowy charakter skalowalności jest widoczny we wszystkich przypadkach, jednak **tbb** wykazuje najmniejsze odstępstwo od idealnego wzrostu, sugerując wyższą efektywność w zarządzaniu wątkami i minimalizacji narzutów.

9.2.2. Wyniki benchmarków - platforma x86_64



Rys. 9.23: Porównanie czasów wykonania benchmarku EP dla klas S, W, A, B względem liczby użytych wątków



Rys. 9.24: Analiza zmienności czasów wykonania benchmarku EP dla klas S, W, A, B względem liczby użytych wątków

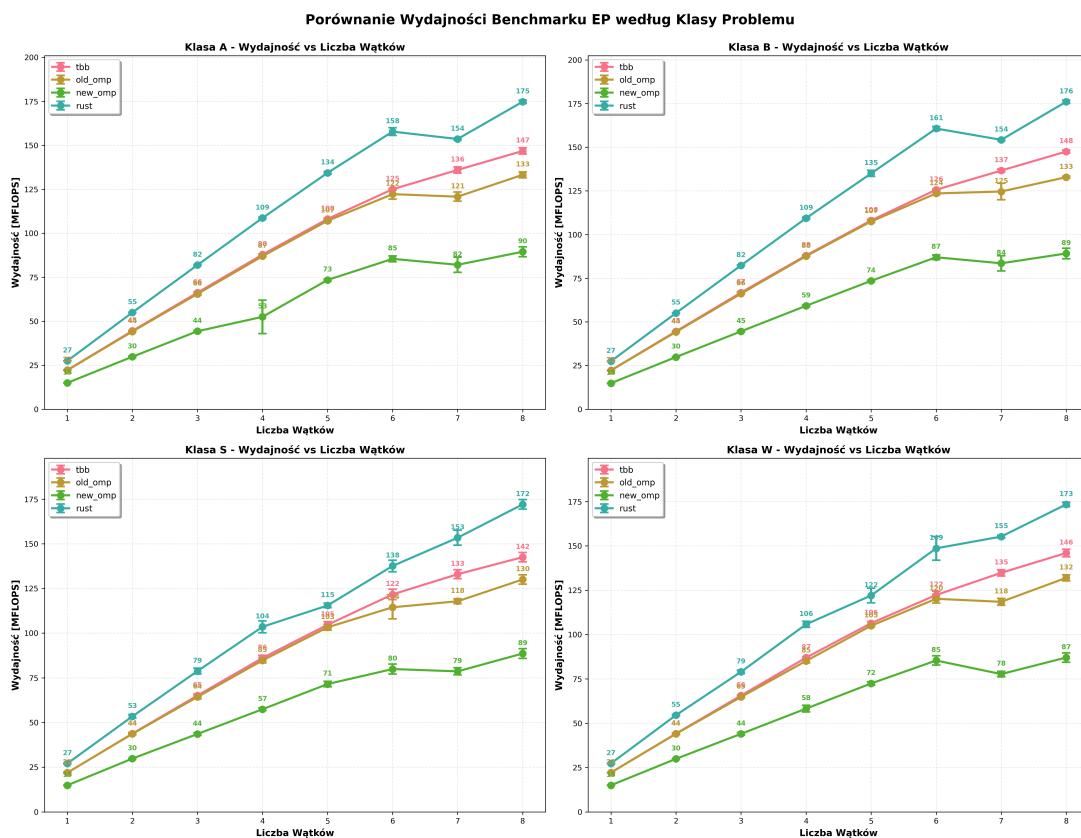
Analiza danych wskazuje, że implementacja `rust` konsekwentnie osiąga najniższe czasy wykonania przy maksymalnej liczbie wątków (8) we wszystkich klasach, co świadczy o jej wyższości w kontekście obliczeń równoległych. W porównaniu do pozostałych implementacji, `rust` wykazuje zauważalną przewagę, szczególnie wyraźną w klasach S i W, gdzie różnice w czasach wykonania są bardziej znaczące. Implementacja `tbb` plasuje się na drugim miejscu pod względem

9. Analiza wyników - programowanie równoległe

wydajności, podczas gdy `old_omp` i `new_omp` charakteryzują się wyższymi czasami wykonania, z `new_omp` wykazującym najniższą efektywność.

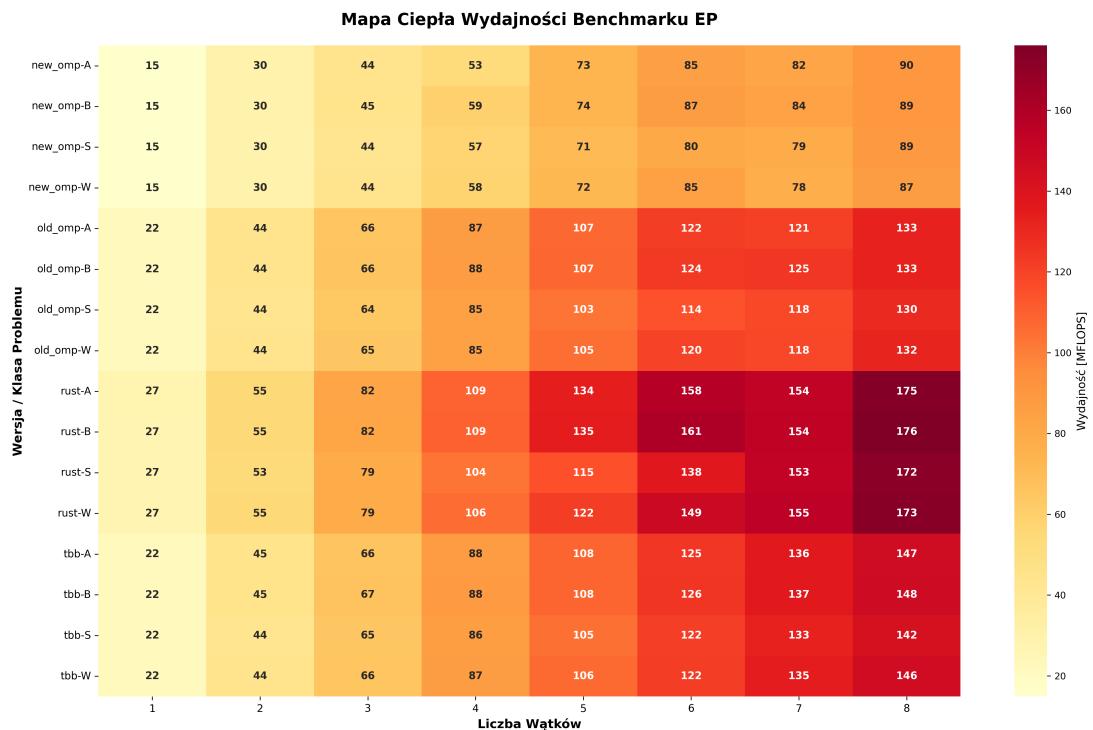
Interpretując wykresy - rysunek 9.24, można zauważać, że dla wszystkich wersji implementacji wartości współczynnika zmienności są stosunkowo niskie, co oznacza wysoką powtarzalność i stabilność pomiarów. Najwyższą zmienność, zarówno dla czasu wykonania, jak i MFLOPS, obserwuje się w przypadku implementacji `new_omp`, co sugeruje, że wyniki tej wersji są mniej stabilne w porównaniu do pozostałych. Implementacje `tbb`, `old_omp` oraz `rust` charakteryzują się niższymi wartościami CV, co świadczy o większej niezawodności uzyskiwanych rezultatów.

Warto zwrócić uwagę na obecność pojedynczych punktów poza głównym zakresem wykresów, które mogą wskazywać na sporadyczne przypadki większych odchyleń w pomiarach, jednak nie wpływają one znacząco na ogólną ocenę stabilności.



Rys. 9.25: Porównanie wydajności benchmarku EP dla klas S, W, A, B względem liczby użytych wątków

Na rysunku 9.25 przedstawiono porównanie wydajności benchmarku EP w milionach operacji zmiennoprzecinkowych na sekundę (MFLOPS) dla różnych klas problemu i liczby wątków. Wartości zostały przedstawione w skali logarytmicznej, co pozwala lepiej zobrazować różnice między implementacjami.



Rys. 9.26: Mapa ciepła wydajności benchmarku EP dla klas S, W, A, B względem liczby użytych wątków

Mapa ciepła - rysunek 9.26 przedstawia wydajność benchmarku EP w zależności od liczby użytych wątków. Odcienie koloru od żółtego do ciemnoczerwonego wskazują na wzrost wydajności.

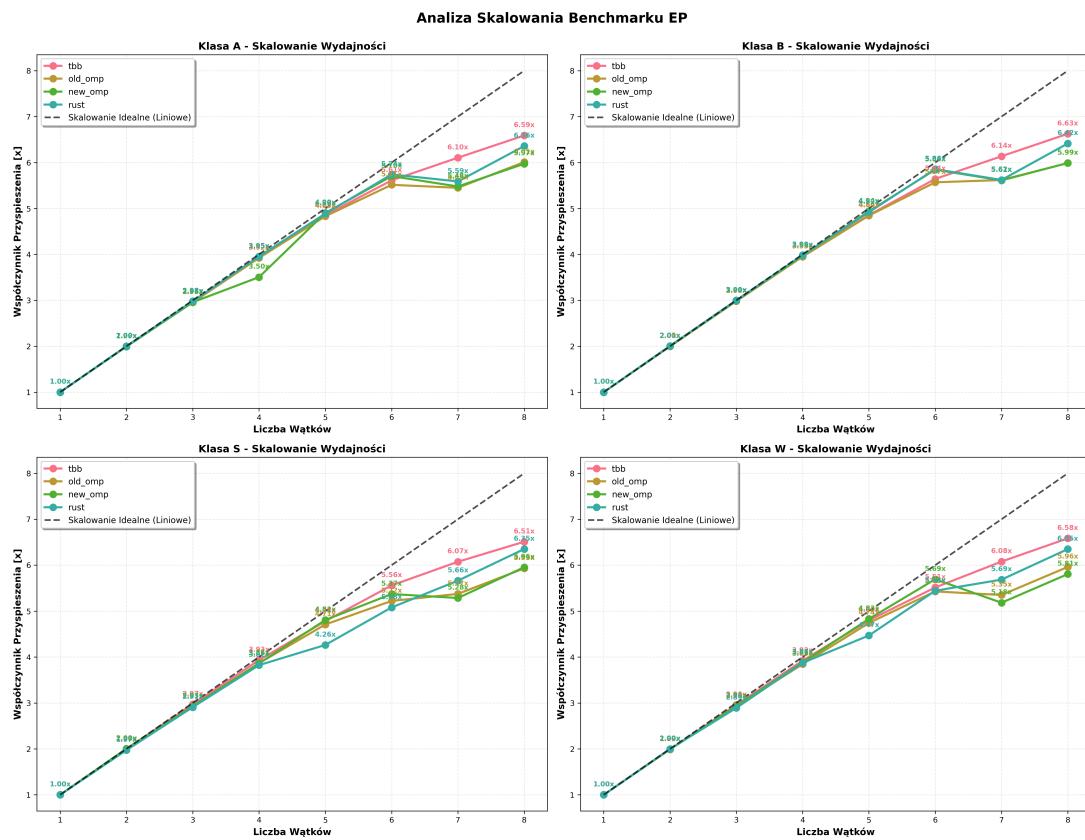
Wszystkie analizowane implementacje wykazują wzrost wydajności wraz ze zwiększeniem liczby wątków, co jest zgodne z oczekiwaniami dla algorytmów równoległych typu embarrassingly parallel. Najbardziej dynamiczny przyrost wydajności obserwowany jest w zakresie od 1 do 5 wątków. Po przekroczeniu tego progu krzywe wzrostu zaczynają się wyraźnie spłaszczać, co sugeruje osiągnięcie punktu nasycenia zasobów sprzętowych.

Spośród wszystkich rozwiązań, najwyższą wydajność osiąga implementacja napisana w języku Rust, konsekwentnie uzyskując największe wartości MFLOPS dla każdej z klas problemu. Przy 8 wątkach osiąga ona poziom około 172-176 MFLOPS, a jej przewaga staje się szczególnie widoczna przy wyższych stopniach równoległości.

Na drugim miejscu pod względem wydajności plasuje się implementacja TBB, osiągająca przy 8 wątkach wartości w przedziale około 142-148 MFLOPS. Niewiele niższe wyniki uzyskuje old_omp, którego maksymalna wydajność w tej konfiguracji wynosi około 130-133 MFLOPS.

Najniższe rezultaty pod względem MFLOPS odnotowano dla implementacji new_omp, która przy 8 wątkach osiąga maksymalnie około 87-90 MFLOPS.

Dodatkowo, we wszystkich implementacjach zaobserwowały się niewielkie wahania wydajności pomiędzy 6 a 7 wątkami. Może to wskazywać na wpływ czynników zależnych od architektury sprzętowej lub mechanizmów zarządzania zasobami stosowanych przez system operacyjny.



Rys. 9.27: Analiza skalowania benchmarku EP dla klas S, W, A, B względem liczby użytych wątków

Wykres - rysunek 9.27 przedstawia analizę skalowania wydajności benchmarku EP w zależności od liczby użytych wątków. Skalowanie zostało wyrażone jako współczynnik przyspieszenia względem wykonania jednowątkowego i odniesione do idealnego skalowania liniowego.

Analiza wspólnych wzorców skalowania

Analiza wykresów przedstawiających współczynniki przyspieszenia pozwala zidentyfikować kilka istotnych wzorców. We wszystkich implementacjach zaobserwowano niemal liniowe skalowanie w zakresie od 1 do 4 wątków. Współczynniki przyspieszenia w tym przedziale są zbliżone do wartości idealnej, co świadczy o wysokiej efektywności równoległej algorytmu EP w warunkach ograniczonego współdzielenia zasobów.

Po przekroczeniu czterech wątków, a więc w zakresie od 5 do 8, wszystkie implementacje zaczynają wykazywać stopniowe odchylenia od skalowania idealnego. Zjawisko to jest typowe dla obliczeń równoległych i wynika z narzutów komunikacyjnych, współdzielenia zasobów sprzętowych, a także ograniczeń wynikających z architektury systemu oraz kosztów synchronizacji.

Ciekawym elementem analizy jest występowanie lokalnego maksimum współczynnika przyspieszenia przy sześciu wątkach, po którym przy siedmiu wątkach następuje niewielki spadek. Może to wskazywać na wpływ specyfiki architektury procesora lub strategii planowania zadań stosowanych przez system operacyjny.

Przy maksymalnej liczbie ośmiu wątków ponownie obserwuje się wzrost współczynnika przyspieszenia we wszystkich implementacjach, co sugeruje efektywniejsze wykorzystanie zasobów obliczeniowych w sytuacji pełnego obciążenia systemu.

Porównanie efektywności różnych implementacji

Spośród wszystkich badanych rozwiązań, implementacja w języku Rust osiąga najwyższe współczynniki przyspieszenia przy ośmiu wątkach, z wartościami rzędu 6,4-6,5x względem wersji

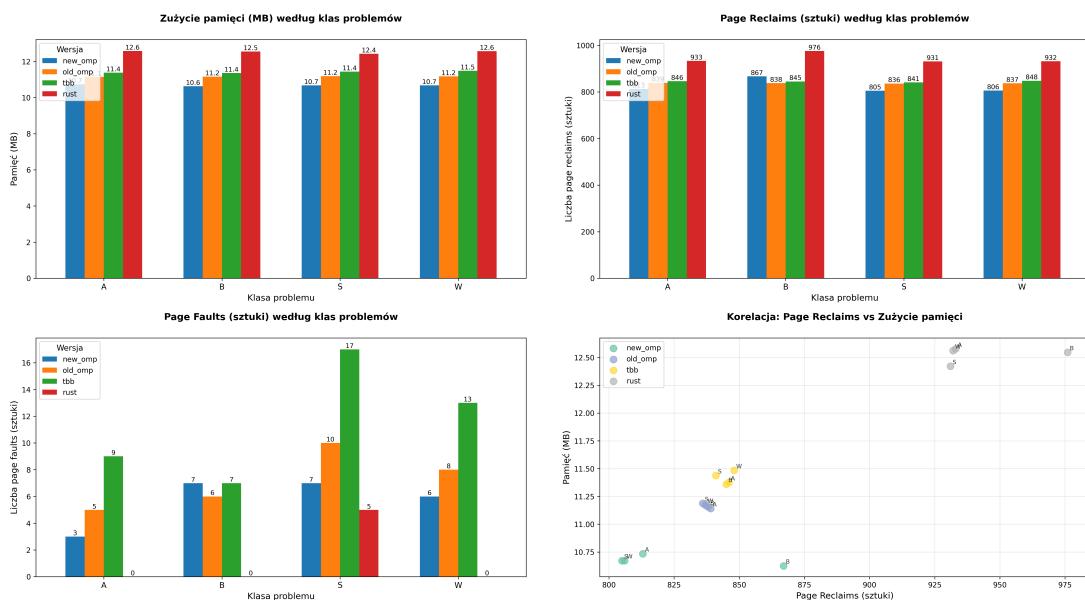
jednowątkowej. Wskazuje to na wysoką efektywność tej implementacji pod względem skalowania równoległego.

Intel TBB uzyskuje nieznacznie niższe, lecz wciąż bardzo dobre wyniki skalowania, osiągając przyspieszenie na poziomie 6,1-6,3x przy ośmiu wątkach. Obie wersje OpenMP, tj. old_omp i new_omp, osiągają nieco niższe współczynniki przyspieszenia, mieszczące się w przedziale 5,8-6,1x, co czyni je mniej efektywnymi w porównaniu z implementacją w języku Rust oraz TBB.

Warto również zaznaczyć, że w przedziale 4-6 wątków wszystkie implementacje osiągają bardzo zbliżone wyniki, co wskazuje na podobną efektywność w wykorzystaniu średniego poziomu równoległości oraz dostępnych rdzeni obliczeniowych.

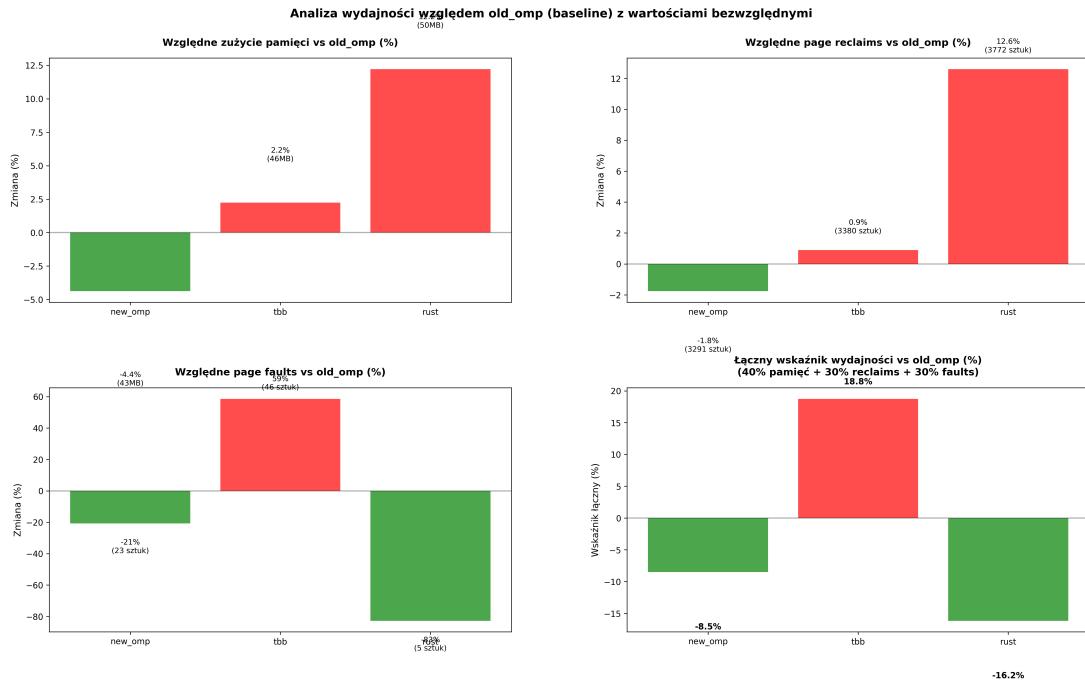
9.2.3. Wyniki profilowania wydajności - platforma ARM64

W celu kompleksowego porównania efektywności testowanych implementacji benchmarku EP na platformie ARM64 dokonano analizy metryk systemowych dotyczących zarządzania pamięcią: zużycia pamięci (ang. *memory usage*), liczby odzyskanych stron pamięci (ang. *page reclaims*), liczby błędów stron pamięci (ang. *page faults*) oraz ich wzajemnych relacji.



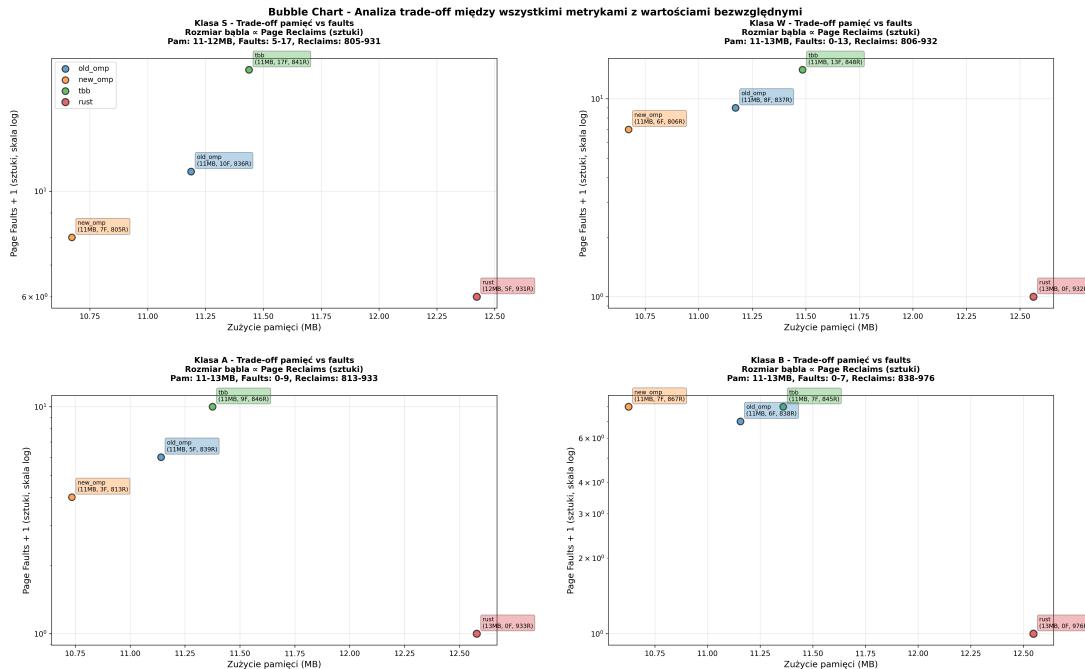
Rys. 9.28: Profilowanie wydajności benchmarku EP dla klas S, W, A, B względem liczby użytych wątków

9. Analiza wyników - programowanie równoległe



Rys. 9.29: Analiza wydajności względem `old_omp` (punkt odniesienia) z wartościami bezwzględnymi

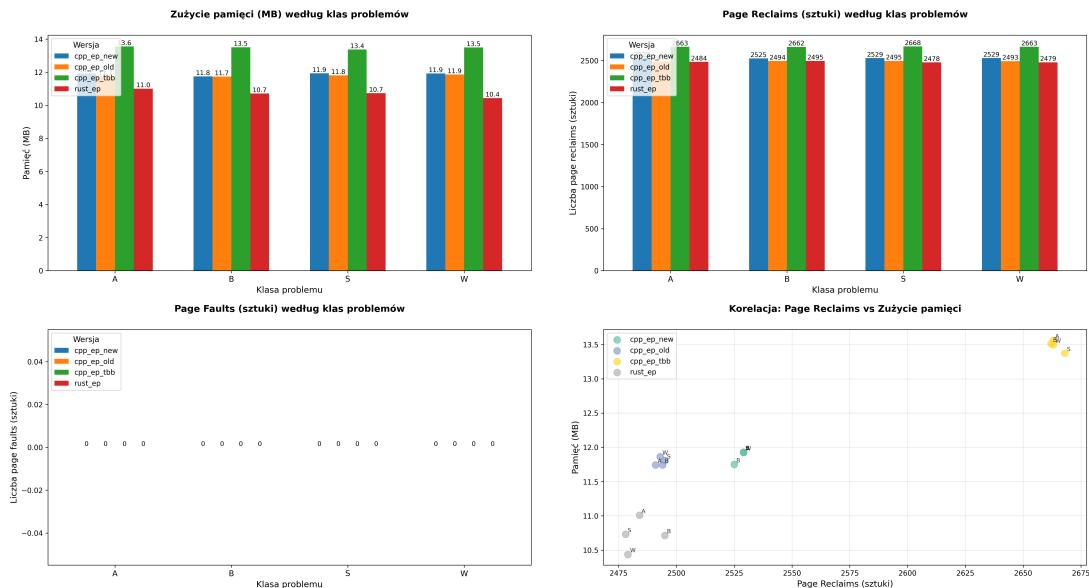
Na rysunku 9.29 zestawiono zmiany procentowe trzech analizowanych wskaźników względem implementacji referencyjnej `old_omp`. Wskaźnikiem syntetycznym jest łączna metryka złożona z ważonych proporcji zużycia pamięci (40%), zwolnionych stron (30%) i błędów stron (30%).



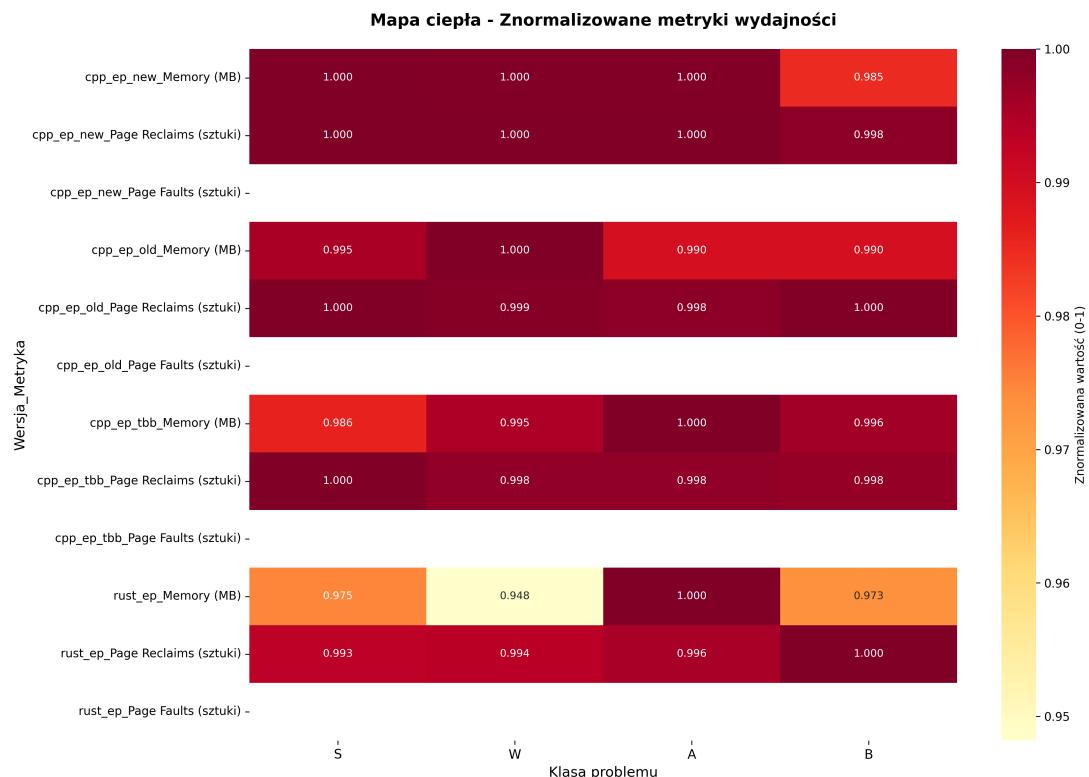
Rys. 9.30: Kompromisy (ang. *trade-off*) pomiędzy zużyciem pamięci a błędami stron pamięci, z uwzględnieniem liczb odzyskanych stron jako trzecim wymiarem (rozmiar bąbla)

Rysunek 9.30 przedstawia kompromisy (ang. *trade-offs*) między zużyciem pamięci a błędami stron, z liczbą odzyskanych stron jako trzecim wymiarem (rozmiar bąbla). Oś Y jest przedstawiona w skali logarytmicznej ze względu na zróżnicowaną skalę błędów stron.

9.2.4. Wyniki profilowania wydajności - platforma x86_64



Rys. 9.31: Profilowanie wydajności benchmarku EP dla klas S, W, A, B względem liczby użytych wątków



Rys. 9.32: Znormalizowana mapa metryk pamięciowych

Analiza zużycia i zarządzania pamięcią

W analizie zużycia pamięci zauważalne są istotne różnice pomiędzy implementacjami. Największe zużycie pamięci systemowej wykazuje implementacja `new_omp`, osiągając w każdej klasie problemu wartość rzędu 13,4–13,6 MB. Szczególnie stabilny poziom zużycia tej implementacji widoczny jest w klasach B, S i W, gdzie wartość oscyluje wokół 13,5 MB.

Zupełnie odmienną charakterystyką wyróżnia się implementacja **rust**, która wykazuje najniższe zużycie pamięci w przedziale 10,4-11,0 MB. Oznacza to około 22% oszczędności względem najbardziej pamięciochłonnej wersji, co świadczy o lepszym zarządzaniu alokacją zasobów. Pozostałe dwie implementacje, **old_omp** i **tbb**, prezentują bardzo zbliżone wartości zużycia pamięci, mieszczące się w zakresie 11,8-11,9 MB, co pozycjonuje je pośrednio pomiędzy skrajnymi przypadkami.

Co istotne, zużycie pamięci dla każdej implementacji pozostaje stosunkowo stałe w obrębie różnych klas problemu. Sugeruje to, że wielkość instancji problemu EP nie wpływa istotnie na alokację pamięci, a struktury danych i strategie zarządzania pamięcią są zaprojektowane w sposób odporny na zmiany skali.

Operacje pamięci wirtualnej

Analiza operacji związanych z pamięcią wirtualną - w szczególności liczby odzyskanych stron oraz liczby błędów strony - dostarcza dodatkowych informacji o efektywności systemu operacyjnego i implementacji.

Wszystkie implementacje odnotowują zbliżoną liczbę odzyskanych stron, w zakresie od 2475 do 2665. Najwyższy poziom zwolnień stron występuje w implementacji **tbb** dla klasy W (2663), podczas gdy **rust** konsekwentnie utrzymuje nieco niższe wartości, co może sugerować bardziej stabilne wykorzystanie przestrzeni adresowej.

Z kolei liczba błędów strony dla wszystkich implementacji wynosi zero we wszystkich klasach problemu. Taki wynik jest bardzo korzystny i świadczy o tym, że pamięć podręczna oraz fizyczna były wystarczająco pojemne, aby uniknąć konieczności odwoływania się do pamięci wymiany. Oznacza to wysoką efektywność zarządzania pamięcią operacyjną w czasie działania benchmarku.

Zależności pomiędzy metrykami pamięciowymi

Wykres korelacyjny - rysunek 9.31 przedstawiający relację pomiędzy liczbą zwolnień stron a zużyciem pamięci ujawnia dodatkowe zależności. Punkty na wykresie grupują się wyraźnie według implementacji, co potwierdza, że charakterystyka pamięciowa jest silnie zależna od konkretnej wersji kodu, a nie od rozmiaru instancji problemu. Implementacja **rust** tworzy odrębny klaster w lewej dolnej części wykresu, co potwierdza jej niskie zużycie pamięci przy niewielkiej liczbie operacji odzyskiwania stron. Z kolei **new_omp** formuje zbiór punktów w prawej górnej części wykresu, co jednoznacznie wskazuje na wyższe zużycie pamięci i większą liczbę zwolnień stron.

Brak silnej liniowej korelacji między tymi dwiema metrykami w obrębie każdej implementacji sugeruje, że są to względnie niezależne aspekty zarządzania pamięcią - niskie zużycie pamięci niekoniecznie musi iść w parze z mniejszą liczbą odzyskanych stron.

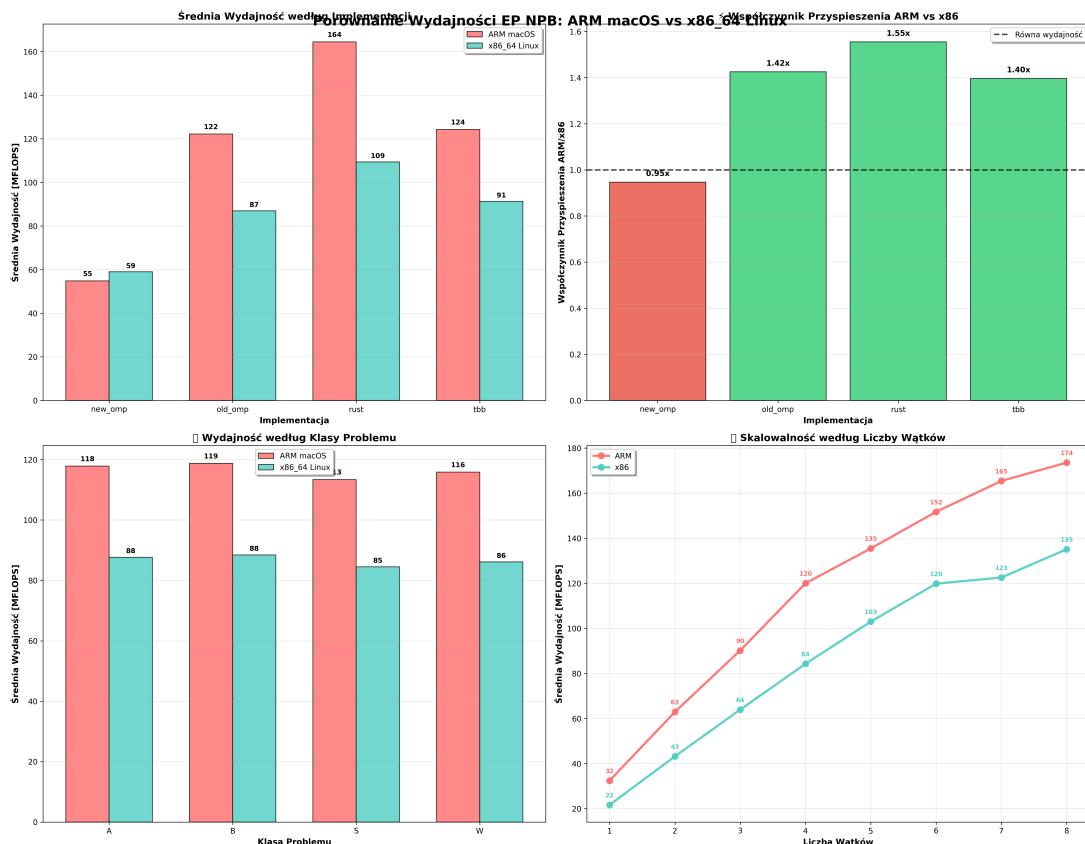
Znormalizowana mapa metryk pamięciowych

Mapa ciepła - rysunek 9.32 prezentująca znormalizowane wartości metryk pamięciowych ujawnia ogólną spójność wyników oraz subtelne różnice pomiędzy implementacjami. Wartości odnoszące się do błędów strony nie są przedstawione, ponieważ wszystkie implementacje uzywały w tym zakresie wartość zero.

Nota wyjaśniająca

Ze względu na brak błędów stron nie zamieszczono pozostałych wykresów, które byłyby oparte na tej metryce. Wykresy te nie miałyby sensu, ponieważ brak błędów stron oznacza, że wszystkie operacje pamięciowe były realizowane w ramach dostępnej pamięci fizycznej bez konieczności angażowania mechanizmów swapowania lub przenoszenia stron z dysku.

9.2.5. Porównanie pomiędzy platformami



Rys. 9.33: Porównanie średniej wydajności benchmarku EP dla platform ARM64 i x86_64

Porównanie wydajności według implementacji i architektury

Analiza średnich wartości MFLOPS dla poszczególnych implementacji w podziale na platformy sprzętowe ujawnia wyraźne różnice, wskazujące na przewagę architektury ARM. Z wyjątkiem implementacji `new_omp`, wszystkie rozwiązania osiągają wyższe wyniki na platformie macOS z architekturą ARM. Szczególnie wyróżnia się implementacja `rust`, której średnia wydajność na ARM wynosi 164 MFLOPS w porównaniu do 109 MFLOPS na platformie x86_64. Przewaga ta wynosi około 50%, co wskazuje na bardzo dobrą adaptację tej implementacji do architektury ARM.

Na obu platformach to właśnie implementacja `rust` osiąga najwyższe wyniki, podczas gdy `new_omp` pozostaje najwolniejszym wariantem. Warto zaznaczyć, że `new_omp` jest jedyną implementacją, która uzyskuje nieco wyższą średnią wydajność na platformie x86_64 (59 MFLOPS) niż na ARM (55 MFLOPS), co może świadczyć o ograniczonej optymalizacji tego rozwiązania pod kątem architektury ARM.

Współczynniki przyspieszenia obliczone jako stosunek średniej wydajności na ARM względem x86_64 potwierdzają powyższe obserwacje. Najwyższy współczynnik (1,55x) notuje implementacja `rust`, natomiast `old_omp` i `tbb` osiągają zbliżone wartości odpowiednio 1,42x i 1,40x. Jedyną implementacją z wartością poniżej jedności (0,95x) pozostaje `new_omp`, co oznacza względnie słabsze skalowanie na ARM.

Wydajność w zależności od klasy problemu

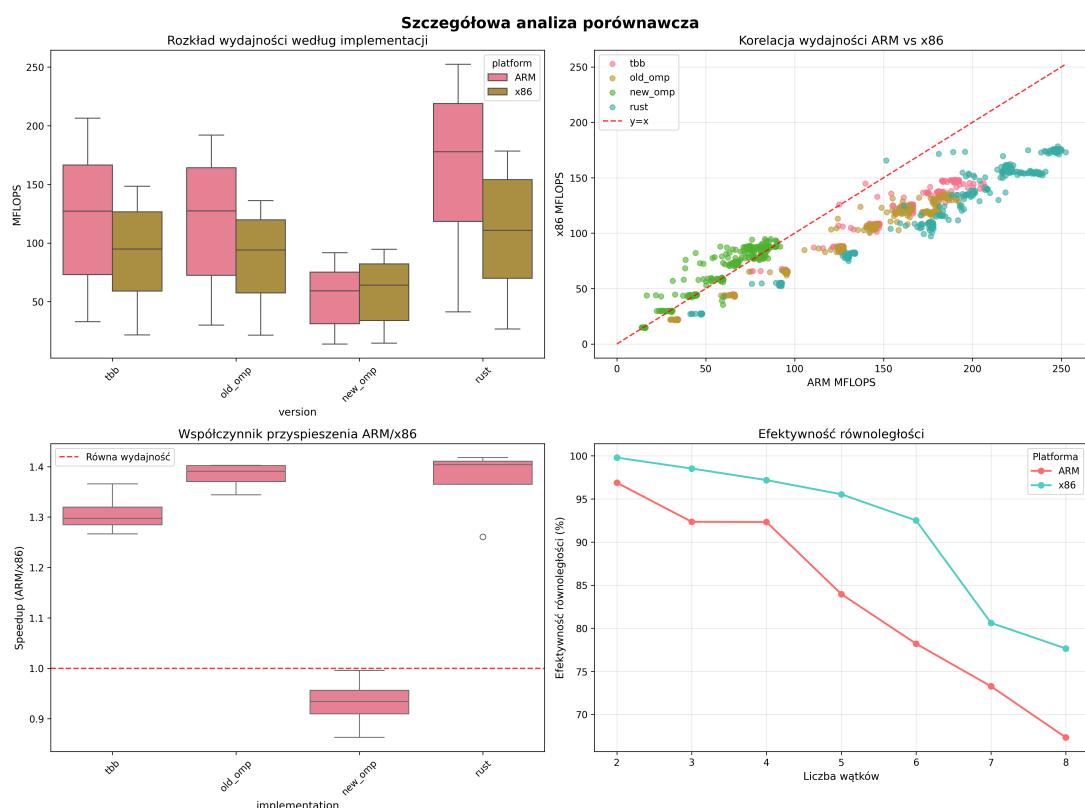
Analiza wydajności w podziale na klasy problemowe, wykazuje spójną przewagę platformy ARM we wszystkich rozmiarach instancji. Dla każdej klasy problemowej wartości MFLOPS

utrzymują się na zbliżonym poziomie, co sugeruje dobrą skalowalność algorytmu niezależnie od rozmiaru danych wejściowych. Średnie wartości dla platformy ARM mieszczą się w przedziale 118-119 MFLOPS, natomiast dla platformy x86_64 wynoszą około 85-88 MFLOPS. Przewaga ARM w tym ujęciu wynosi około 35%, co potwierdza, że jest to systemowo korzystniejsza architektura dla danego typu obciążień obliczeniowych.

Skalowanie według liczby wątków

Wykresy prezentujące wydajność w zależności od liczby wątków ukazują zblizone trendy dla obu architektur. W obu przypadkach obserwuje się niemal liniowy wzrost wydajności do poziomu 5-6 wątków, po czym przyrost staje się bardziej umiarkowany. Jest to zgodne z typowym zachowaniem dla aplikacji równoległych, w których narzuty synchronizacji i ograniczenia sprzętowe zaczynają odgrywać większą rolę przy dalszym zwiększeniu liczby wątków.

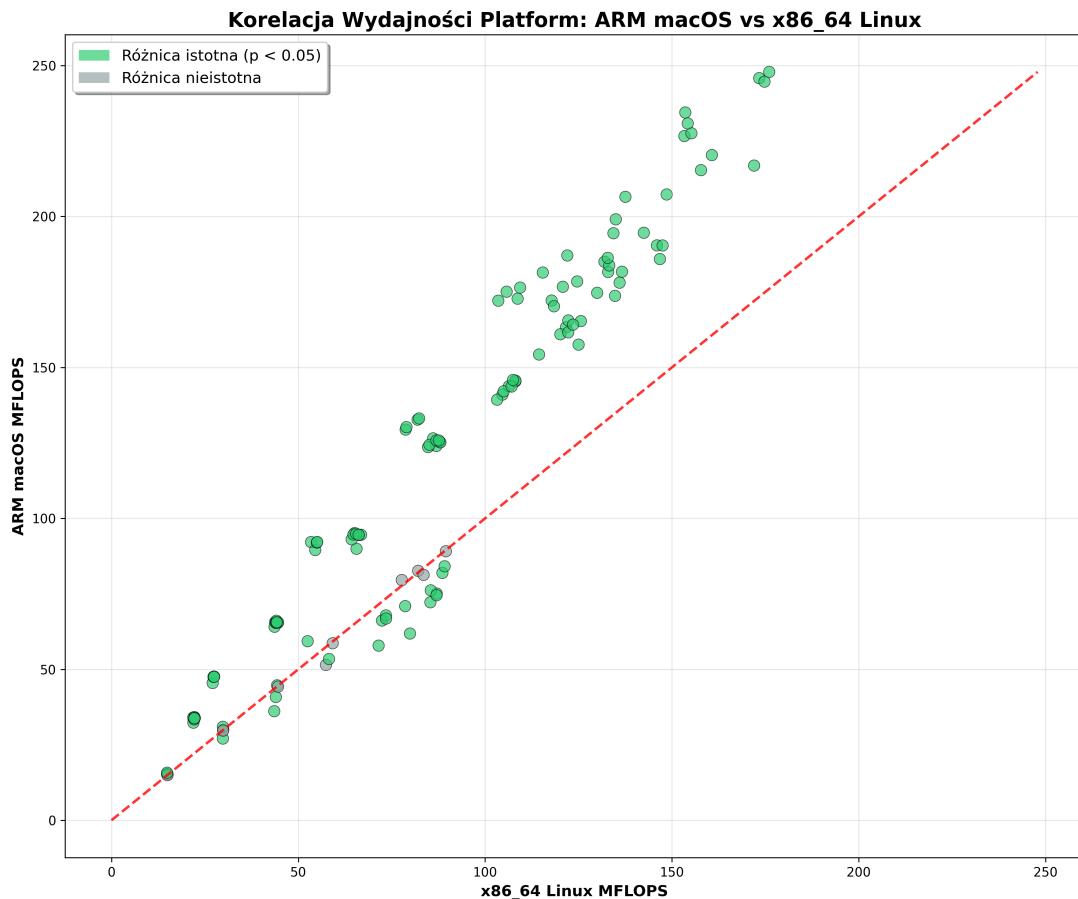
Pod względem wartości bezwzględnych platforma ARM osiąga wyższe maksima - przy 8 wątkach wydajność sięga około 174 MFLOPS, podczas gdy dla x86_64 wynosi ona 135 MFLOPS. Również ogólna efektywność skalowania jest nieco wyższa na ARM, choć analiza wykresów efektywności równoległej wskazuje na szybszy spadek tej metryki przy większej liczbie wątków. Dla ośmiu wątków efektywność na ARM spada do około 68%, podczas gdy na x86_64 utrzymuje się na poziomie około 78%. Może to wskazywać na większą wrażliwość architektury ARM na narzuty związane z wielowątkowością.



Rys. 9.34: Szczegółowa analiza wydajności benchmarku EP dla platform ARM64 i x86_64

Wyniki uzupełniają analiza rozkładów i korelacji wydajności dla obu platform - rysunek 9.34. Wykresy pudełkowe dla implementacji `rust`, `old_omp` oraz `tbb` pokazują wyraźne przesunięcie mediany i całego rozkładu ku wyższym wartościami na platformie ARM. Sugeruje to nie tylko wyższą średnią wydajność, ale także większą stabilność działania w zakresie typowych przypadków.

Wykres korelacji wydajności między platformami wskazuje na silną liniową zależność, przy czym większość punktów znajduje się wyraźnie powyżej linii $y = x$. Potwierdza to systematycznie wyższą wydajność ARM względem x86_64, szczególnie w przypadku implementacji osiągających wysokie wartości MFLOPS.



Rys. 9.35: Analiza istotności statystycznej benchmarku EP dla platform ARM64 i x86_64

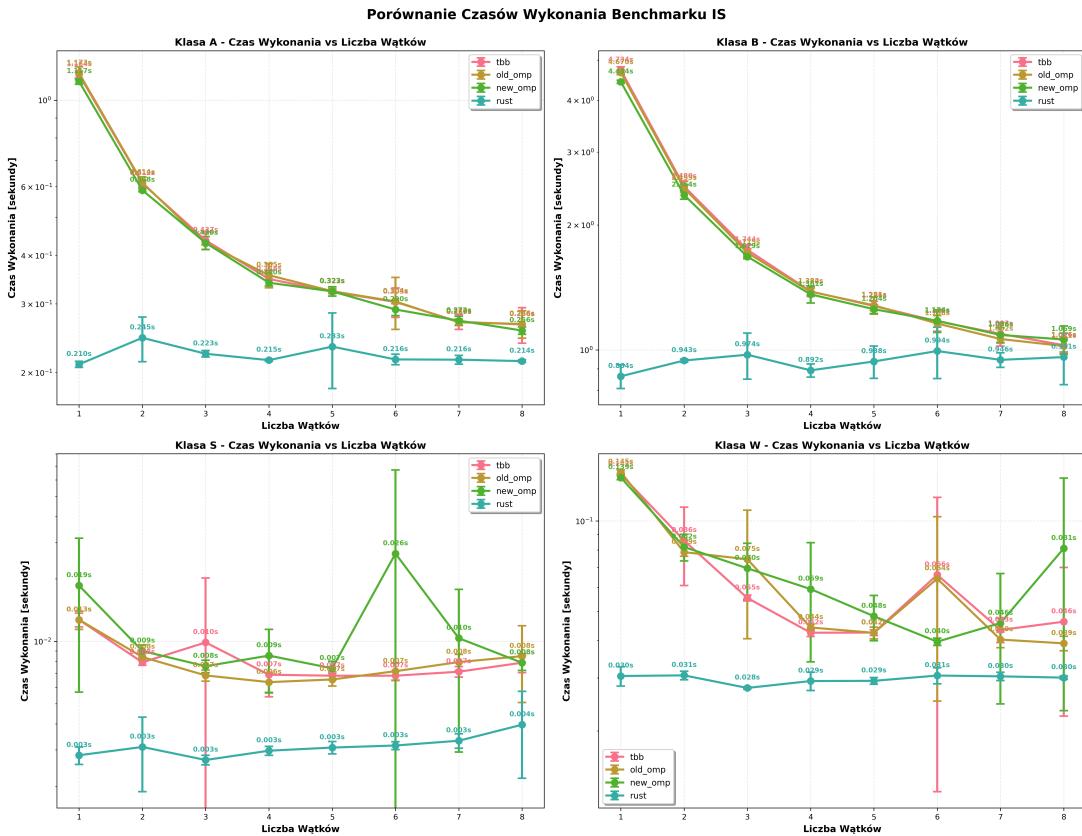
Analiza statystyczna różnic wydajności - rysunek 9.35 wykazuje, że w zdecydowanej większości przypadków różnice te są statystycznie istotne ($p < 0,05$). Podnosi to wiarygodność wniosków oraz potwierdza, że zaobserwowane efekty nie są przypadkowe, lecz wynikają z realnych różnic architektonicznych i implementacyjnych.

9.3. Benchmark IS

Celem eksperymentów była ocena efektywności implementacji algorytmu sortowania całkowitoliczbowego (Integer Sort, IS) w czterech wersjach programistycznych (`new_omp`, `old_omp`, `tbb`, `rust`) przy różnych klasach problemów (S, W, A, B) i liczbie wątków. Porównanie obejmuje czas wykonania, wydajność (MFLOPS) oraz współczynnik zmienności (CV - coefficient of variation), obrazujący stabilność pomiarową.

9.3.1. Wyniki benchmarków - platforma ARM64

Na wykresie - rysunek 9.36 najbardziej znaczącym obserwowanym zjawiskiem jest dominacja implementacji w języku Rust, która osiąga zdecydowanie najkrótsze czasy wykonania we wszystkich klasach problemu oraz dla wszystkich konfiguracji liczby wątków. W szczególności w klasach A i B różnice pomiędzy Rust a pozostałymi implementacjami są bardzo wyraźne



Rys. 9.36: Porównanie czasów wykonania benchmarku IS dla klas S, W, A, B względem liczby użytych wątków

i sięgają nawet kilku razy krótszych czasów wykonania, co sugeruje fundamentalnie bardziej efektywną organizację obliczeń.

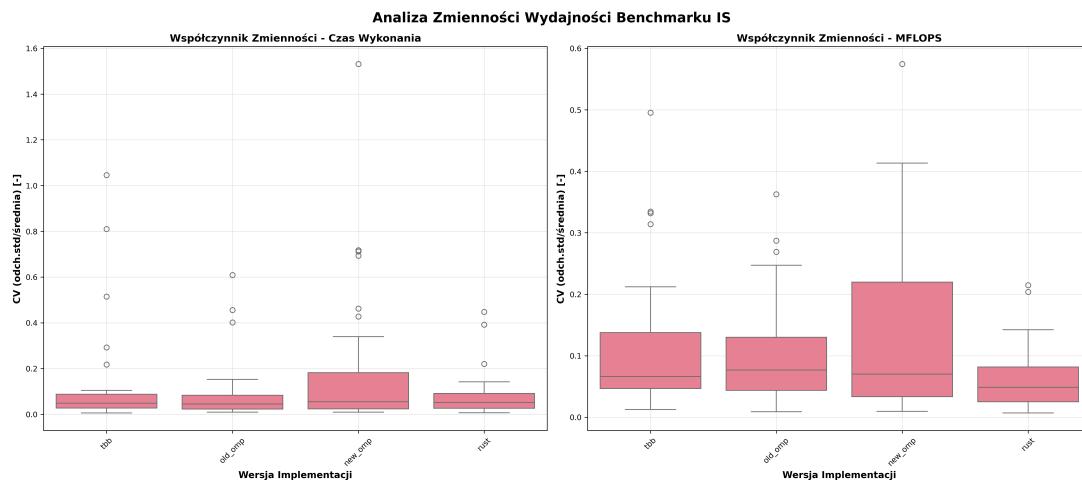
W przypadku klas A i B wszystkie implementacje, z wyjątkiem Rust, wykazują systematyczny spadek czasu wykonania wraz ze wzrostem liczby wątków. Na skali logarytmicznej zależność ta przyjmuje niemal liniowy charakter, co oznacza, że wzrost liczby wątków przekłada się na wykładniczą poprawę wydajności. Szczególnie dobrze skalują się implementacje oparte na bibliotekach tbb oraz OpenMP (zarówno old_omp, jak i new_omp), co potwierdza skuteczność zastosowanych modeli równoległości dla dużych, dobrze zbalansowanych zadań.

Można zauważyć kolejne odrębne zjawisko dla implementacji w rust, której charakterystyka czasów wykonania przy zwiększaniu liczby wątków pozostaje stosunkowo płaska. Sugeruje to, że implementacja ta osiąga bardzo wysoką wydajność już przy niskim poziomie równoległości, a dalsze zwiększanie liczby wątków nie przynosi istotnych korzyści.

W mniejszych klasach problemu, takich jak S i W, obserwuje się znacznie większą nieregularność skalowania oraz większą wariancję wyników. Jest to zgodne z oczekiwaniami, ponieważ dla niewielkich zadań narzuty związane z uruchamianiem i synchronizacją wątków często przewyższają potencjalne zyski z ich równoległego wykonania. Szczególnie widoczne jest to w przypadku implementacji new_omp, która w klasie W wykazuje nawet wzrost czasu wykonania przy 6 i 8 wątkach, co może być efektem niewłaściwego rozkładu pracy lub kosztów synchronizacji przewyższających zyski z równoległości.

Dalsza analiza 9.36 według klas problemu pozwala zauważać, że w klasach A i B - reprezentujących większe instancje obliczeniowe - implementacje tbb, old_omp i new_omp osiągają bardzo podobne czasy wykonania i wykazują zbliżone wzorce skalowania. Ich wydajność rośnie do poziomu 8 wątków, po czym tempo przyrostu stopniowo maleje. W tym samym czasie rust konsekwentnie utrzymuje swoją przewagę, osiągając czasy wykonania nawet pięciokrotnie krótsze. W mniejszych klasach (S i W) efektywność równoległości staje się mniej przewidywalna.

Różnice między implementacjami są mniej wyraźne, a korzyści z równoległości często stają się marginalne powyżej czterech wątków.



Rys. 9.37: Analiza zmienności czasów wykonania benchmarku IS dla klas S, W, A, B względem liczby użytych wątków

Analiza zmienności czasu wykonania

Lewy wykres - rysunek 9.37 ilustruje współczynnik zmienności (CV) dla czasu wykonania. Na jego podstawie można zauważać, że większość pomiarów, niezależnie od użytej implementacji, cechuje się niskim poziomem zmienności - wartości median CV mieszczą się w zakresie od około 0,05 do 0,1. Świadczy to o wysokiej powtarzalności wyników i braku istotnych fluktuacji czasowych w typowych warunkach testowych.

Spośród wszystkich badanych rozwiązań, implementacja napisana w języku Rust charakteryzuje się najniższą medianą współczynnika zmienności, co wskazuje na najbardziej przewidywalne czasy wykonania. Z kolei implementacja new_omp wykazuje nieco wyższą zmienność i szerszy rozrzut wartości, co może świadczyć o mniej stabilnym zachowaniu w czasie - być może z powodu większego wpływu czynników systemowych lub charakterystyki zastosowanego modelu równoległości. Warto również zwrócić uwagę na obecność wartości odstających we wszystkich implementacjach. Największe ekstremalne wartości współczynnika zmienności czasu wykonania zaobserwowano w przypadku implementacji new_omp (sięgające nawet 1,5) oraz tbb (do około 1,05), co może oznaczać sporadyczne przypadki znaczającej niestabilności czasowej.

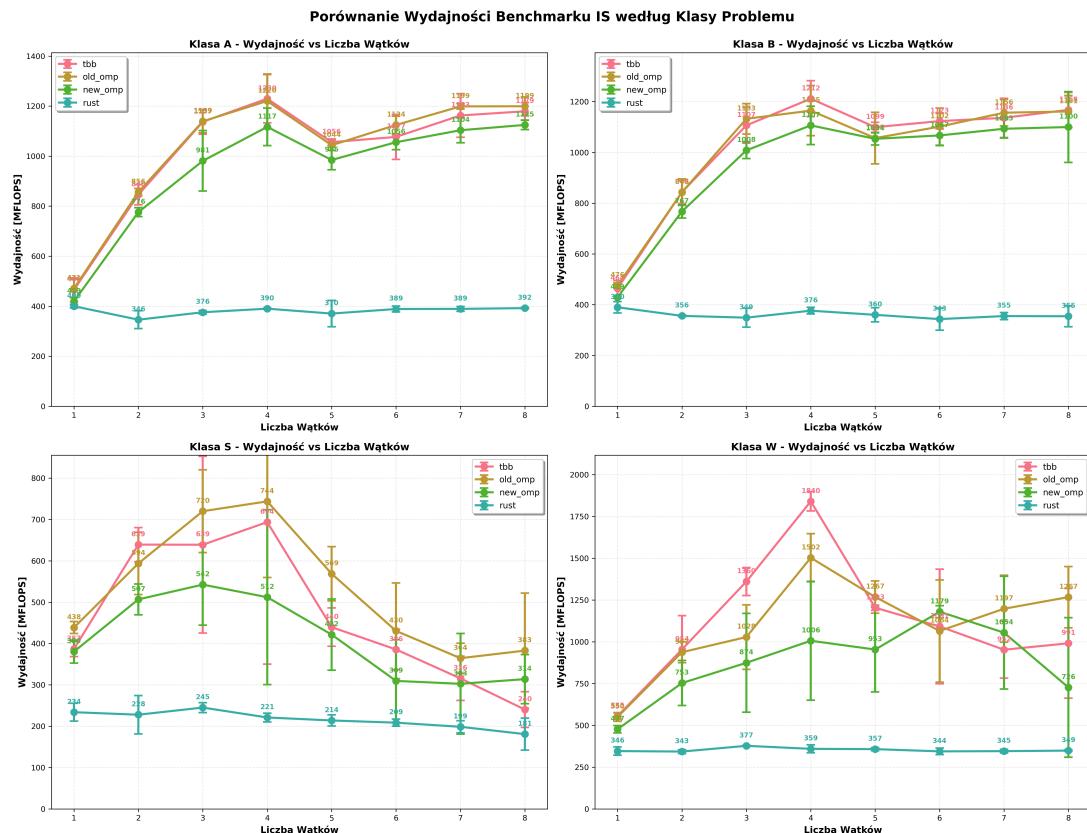
Analiza zmienności MFLOPS

Drugi wykres przedstawia zmienność metryki MFLOPS, która - mimo że pochodna względem czasu wykonania - wykazuje ogólnie wyższy poziom zmienności. Jest to zrozumiałe, biorąc pod uwagę, że wskaźnik ten jest bardziej podatny na mikrofluktuacje w wydajności procesora, zarządzanie energią czy zmiany częstotliwości taktowania. Również tutaj implementacja new_omp wyróżnia się najwyższą medianą współczynnika zmienności oraz największym rozrzutem wartości, co potwierdza jej ograniczoną stabilność pod względem uzyskiwanej wydajności. Z drugiej strony, implementacja Rust ponownie osiąga najlepsze wyniki, z najniższą zmiennością i relatywnie wąskim zakresem obserwowanych wartości, co świadczy o jej konsekwentnej wydajności niezależnie od liczby powtórzeń czy konfiguracji.

Rozkład współczynnika zmienności MFLOPS jest ogólnie mniej symetryczny niż w przypadku czasu wykonania. Szczególnie dla implementacji tbb i new_omp obserwuje się wyraźną

9. Analiza wyników - programowanie równoległe

asymetrię wykresów pudełkowych, co może sugerować występowanie bardziej złożonych zjawisk wpływających na wahania wydajności. Najwyższa wartość odstojąca w tym przypadku - około 0,58 - również dotyczy implementacji `new_omp`, co po raz kolejny podkreśla jej względnie niższą stabilność. Ogółem, analiza zmienności wskazuje, że choć większość implementacji zapewnia akceptowalny poziom powtarzalności, to Rust wyróżnia się nie tylko pod względem szybkości, ale również przewidywalności działania, co czyni go najbardziej stabilnym rozwiązaniem spośród badanych.



Rys. 9.38: Porównanie wydajności benchmarku IS dla klas S, W, A, B względem liczby użytych wątków

Na wykresach na rysunku 9.38 zaprezentowano porównanie wydajności benchmarku IS mieściionej w MFLOPS (milionach operacji zmiennoprzecinkowych na sekundę). Wydajność została przedstawiona jako funkcja liczby wątków (1-8) dla czterech implementacji równoległych.

Analiza zależności wydajności od liczby wątków ujawnia nietypowy wzorzec skalowania. W przeciwieństwie do klasycznych benchmarków, w których można zaobserwować proporcjonalny wzrost wydajności wraz ze wzrostem liczby wątków, implementacje `tbb`, `old_omp` oraz `new_omp` wykazują poprawę wydajności jedynie do poziomu 3-4 wątków. Po tym punkcie wydajność zaczyna się stabilizować, a w niektórych przypadkach wręcz nieznacznie spada. Może to wskazywać na istnienie fundamentalnych ograniczeń wynikających z przepustowości pamięci lub narzutu synchronizacji, które skutecznie ograniczają dalsze korzyści wynikające z równoległości.

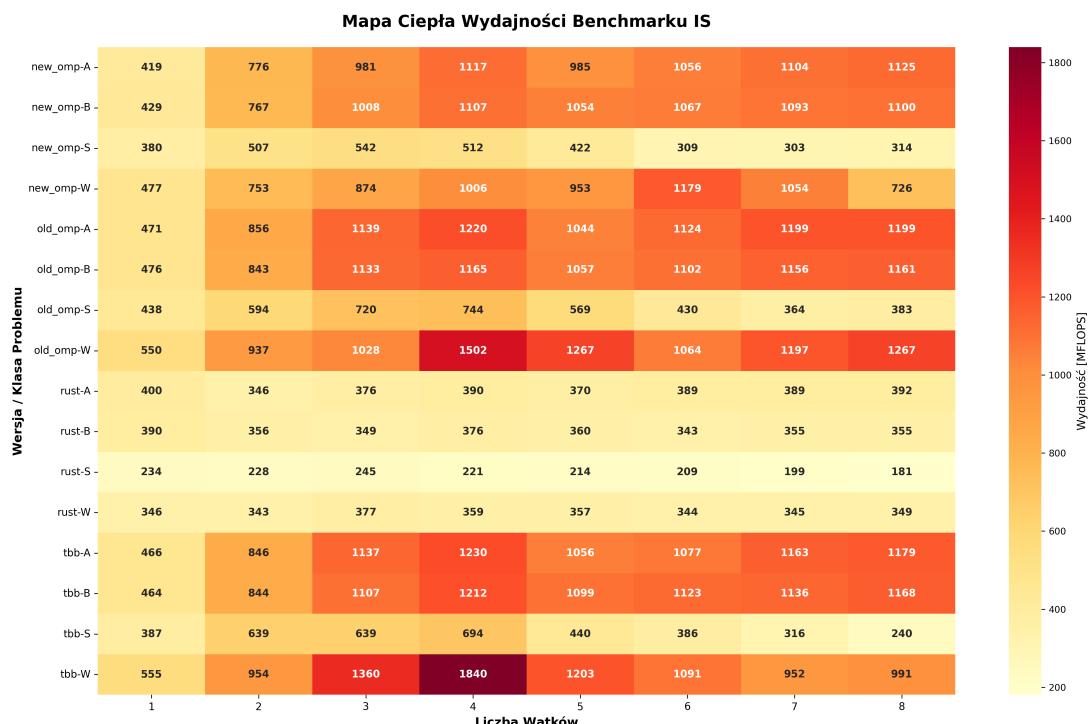
W przypadku mniejszych klas problemowych, takich jak S i W, obserwowane wzorce są jeszcze bardziej nieregularne. Wydajność często osiąga maksimum przy 3-4 wątkach, po czym wyraźnie spada. Szczególnie interesujący jest przypadek implementacji `tbb` w klasie W, która osiąga imponującą wartość 1840 MFLOPS przy czterech wątkach - najwyższą ze wszystkich konfiguracji w całym badaniu. Takie lokalne maksima sugerują silną zależność algorytmu od konkretnej wielkości danych i liczby wątków, co może wynikać z efektów dopasowania do rozmiaru cache lub harmonogramu zadań.

W wyraźnym kontraście pozostaje implementacja w języku Rust, która wykazuje niezwykle stabilne zachowanie. Jej wydajność pozostaje na stosunkowo stałym poziomie bez względu na liczbę wątków. Choć osiągane wartości MFLOPS są zauważalnie niższe niż w pozostałych implementacjach - dla klas A i B mieszczą się w zakresie 350-400 MFLOPS, a dla S i W wynoszą 200-350 MFLOPS - to cechują się wysoką regularnością. Taki rozkład sugeruje, że Rust wykorzystuje inną strategię wykonania, bardziej odporną na zmienność środowiska wykonawczego, ale jednocześnie mniej efektywną w pełnym wykorzystaniu potencjału równoległego sprzętu.

Porównując osiągi różnych implementacji, zauważa się wyraźną dominację `tbb` i `old_omp` w klasach A i B. Obie te implementacje osiągają szczytowe wartości na poziomie około 1200 MFLOPS przy użyciu 3-4 wątków, natomiast `new_omp` nieznacznie ustępuje im, osiągając maksymalnie około 1100 MFLOPS. Różnice te są niewielkie, ale powtarzalne i mogą wskazywać na subtelne różnice w sposobie zarządzania wątkami lub rozdzielania zadań.

W mniejszych klasach problemowych różnice pomiędzy implementacjami są wyraźniejsze, a uzyskane rezultaty mniej przewidywalne. Po raz kolejny wyróżnia się tutaj implementacja `tbb`, która w klasie W bije rekord wydajności, osiągając wspomniane 1840 MFLOPS. Skalowanie w tych klasach jest nierówne i często nieliniowe, co świadczy o dużej wrażliwości na liczbę wątków oraz właściwości algorytmu w kontekście niewielkich zbiorów danych.

Implementacja `rust` utrzymuje konsekwentnie niższy poziom wydajności we wszystkich klasach i konfiguracjach. Jest to zaskakujące, biorąc pod uwagę jej bardzo dobre wyniki w innych benchmarkach, takich jak EP.



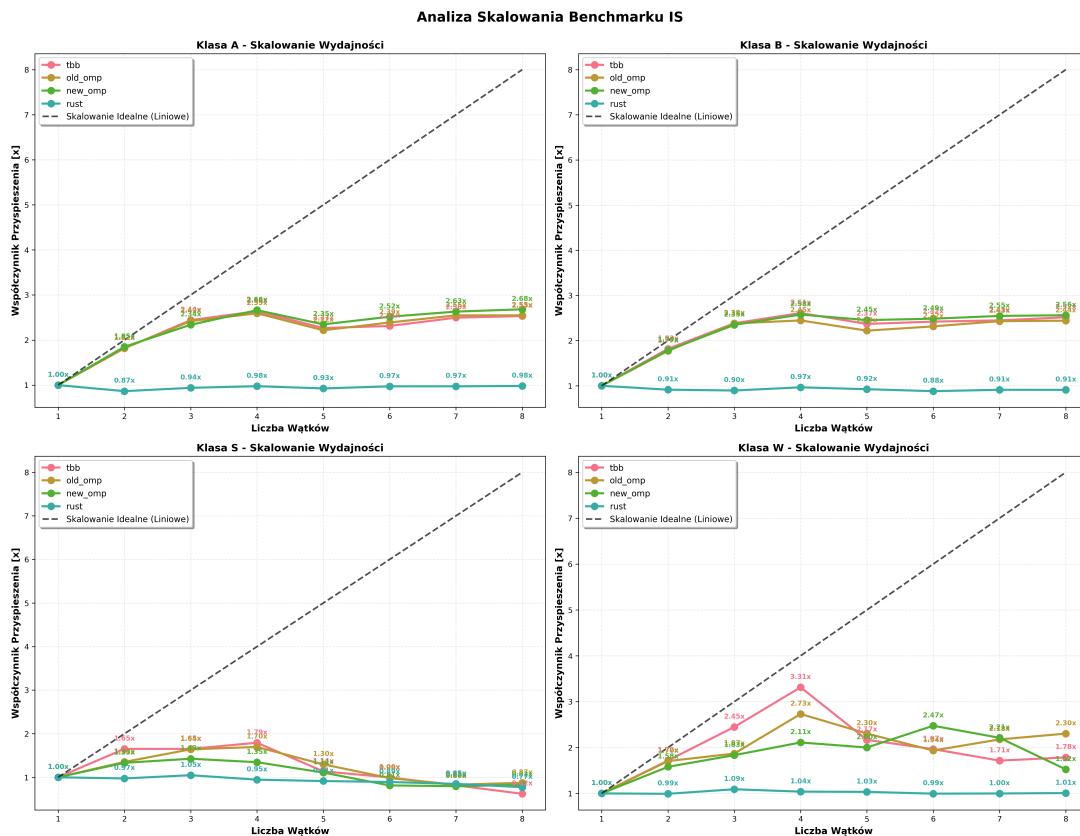
Rys. 9.39: Mapa ciepła wydajności benchmarku IS dla klas S, W, A, B względem liczby użytych wątków

Powyższa mapa cieplna - rysunek 9.39 przedstawia wydajność (w MFLOPS). Wydajność została przedstawiona w zależności od liczby użytych wątków. Odcienie koloru od żółtego do ciemnoczerwonego wskazują na wzrost wydajności.

W klasie S wydajność wszystkich implementacji jest wyraźnie niższa, co manifestuje się jako jaśniejszy poziomy pas na mapie. Jest to efekt słabego skalowania oraz niewielkich rozmiarów danych, które ograniczają możliwość efektywnego podziału pracy między wątki. Z kolei wiersze odpowiadające implementacjom `rust` są dość jednolite i jasne, co potwierdza zarówno niższą ogólną wydajność, jak i brak znaczącego wpływu liczby wątków na wynik końcowy.

9. Analiza wyników - programowanie równoległe

Wreszcie, klasa W ukazuje największe wahania, co dobrze ilustrują nieregularne "gorące punkty" - obszary lokalnej maksymalnej wydajności wśród ogólnego spadku, szczególnie dla tbb i old_omp. Te wahania wskazują na wysoką czułość algorytmu IS względem parametrów wykonania i potwierdzają, że efektywność obliczeń w tym benchmarku jest silnie uzależniona od konkretnych kombinacji danych wejściowych i zasobów sprzętowych



Rys. 9.40: Analiza skalowania benchmarku IS dla klas S, W, A, B względem liczby użytych wątków

Analiza wykresów 9.40 przedstawiających współczynnik przyspieszenia w zależności od liczby wątków ujawnia wyraźne odstępstwa od idealnego skalowania. Choć teoretycznie przyspieszenie powinno rosnąć proporcjonalnie do liczby wątków (co ilustruje linia przerywana na wykresach), w rzeczywistości wartości te osiągają jedynie poziom 2,6-3,3x nawet przy ośmiu wątkach. Taki wynik świadczy o istotnych ograniczeniach efektywności równoległego przetwarzania w przypadku benchmarku IS.

We wszystkich implementacjach i klasach problemowych wyraźnie zaznacza się punkt nasycenia wydajności - lokalna efektywność równoległa osiągana jest zazwyczaj przy 3-4 wątkach. Po przekroczeniu tego progu przyrosty wydajności ulegają zahamowaniu, a w niektórych przypadkach nawet dochodzi do jej spadku. Sugeruje to, że kluczowym czynnikiem ograniczającym skalowanie nie jest sama liczba dostępnych rdzeni, lecz raczej ograniczenia przepustowości pamięci, które uniemożliwiają dalsze równoległe przetwarzanie bez utraty efektywności.

Wielu konfiguracjom towarzyszy również zjawisko niemonotoniczności przyspieszenia. Charakterystyczne są lokalne spadki współczynnika przyspieszenia przy przejściu z 4 do 5 wątków, po czym obserwuje się niewielki wzrost w zakresie 6-8 wątków. Taki przebieg może wskazywać na złożone interakcje między charakterem dostępu do pamięci a architekturą hierarchii cache procesora, które wpływają negatywnie na spójność i przewidywalność skalowania.

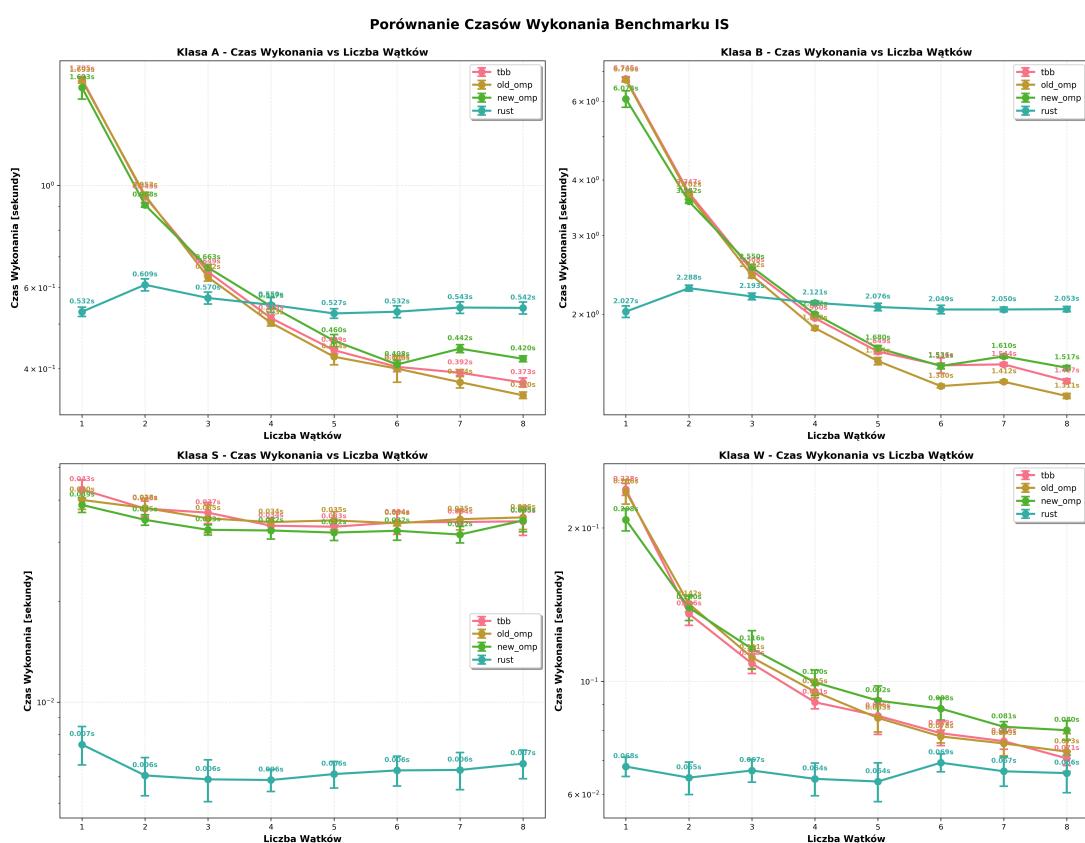
Porównując poszczególne implementacje, można zauważać duże podobieństwo między tbb, old_omp i new_omp, szczególnie w klasach problemowych A i B. Wszystkie trzy osiągają maksymalne współczynniki przyspieszenia na poziomie 2,5-2,7x, co sugeruje, że ograniczenia

skalowania wynikają przede wszystkim z charakterystyki algorytmu sortowania całkowitoliczbowego IS, a nie z różnic pomiędzy samymi bibliotekami równoległyimi.

Na tym tle wyraźnie wyróżnia się implementacja `tbb` w klasie W, która przy czterech wątkach osiąga współczynnik przyspieszenia wynoszący 3,31x.

Z kolei implementacja napisana w języku Rust prezentuje zupełnie odmienny obraz. Współczynniki przyspieszenia we wszystkich klasach problemowych oscylują wokół wartości 1,0, co oznacza, że praktycznie nie uzyskuje się żadnych korzyści z wykorzystania wielu wątków. Wskazuje to na fundamentalne różnice w implementacji algorytmu lub architekturze mechanizmów współbieżności, które mogą uniemożliwić efektywne rozproszenie obciążenia obliczeniowego. W rezultacie, mimo znanej stabilności i bezpieczeństwa języka Rust, w tym konkretnym scenariuszu nie udaje się wykorzystać jego potencjału do osiągnięcia przyspieszenia równoległego.

9.3.2. Wyniki benchmarków - platforma x86_64



Rys. 9.41: Porównanie czasów wykonania benchmarku IS dla klas S, W, A, B względem liczby użytych wątków

Analiza czasów wykonania - rysunek 9.41 ujawnia jednoznaczna przewagę implementacji Rust, która we wszystkich klasach problemu uzyskuje najniższe wartości czasów wykonania. Różnice te stają się szczególnie widoczne przy większej liczbie wątków. Dla przykładu, w klasie A przy ośmiu wątkach implementacja Rust osiąga czas około 0,542 sekundy, podczas gdy inne rozwiązania mieszczą się w zakresie 0,360-0,420 sekundy. Jeszcze wyraźniejsza dysproporcja występuje w klasie B, gdzie czas wykonania dla Rust wynosi około 2,05 sekundy, a dla pozostałych implementacji mieści się w przedziale 1,35-1,55 sekundy.

Implementacje różnią się istotnie pod względem charakterystyki skalowania wraz ze wzrostem liczby wątków. Implementacje `tbb`, `old_omp` oraz `new_omp` wykazują przewidywalny, choć ograniczony spadek czasu wykonania przy zwiększaniu liczby wątków. Największe zyski

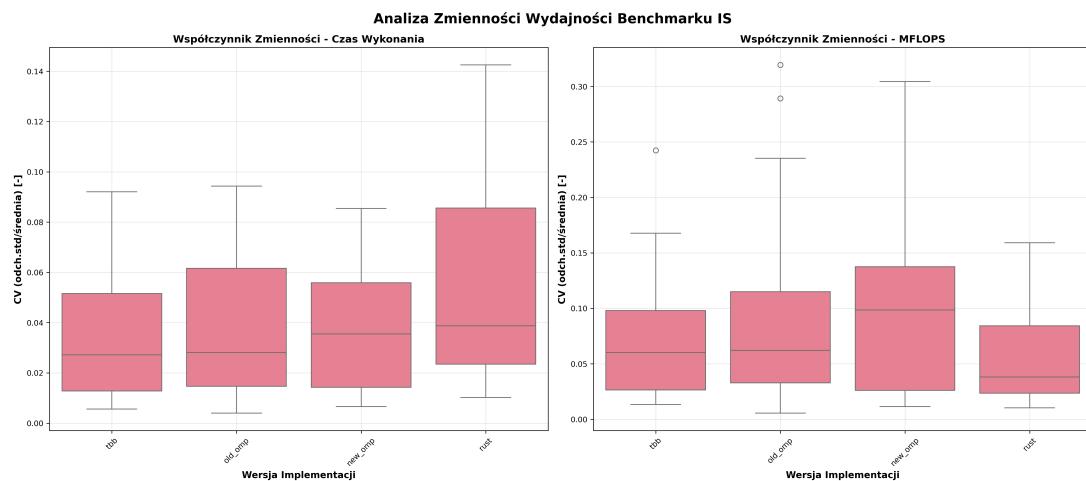
9. Analiza wyników - programowanie równoległe

pojawiają się między 1 a 4 wątkami. Po przekroczeniu tego progu przyrosty stają się marginalne, a niekiedy nawet zauważalny jest niewielki wzrost czasu. W większości konfiguracji implementacja `tbb` okazuje się minimalnie szybsza od pozostałych, co może wskazywać na skuteczniejszą obsługę wątków w tej bibliotece.

Z kolei implementacja Rust prezentuje niemal całkowicie płaską charakterystykę czasową dla liczby wątków większej niż jeden. W klasach A i B można wręcz zaobserwować nieznaczne pogorszenie czasu wykonania przy przejściu z jednego do dwóch wątków.

Wyraźnie zarysowuje się różnica między większymi (A, B) a mniejszymi (S, W) klasami problemu. Dla klas A i B, które reprezentują bardziej wymagające obciążenia obliczeniowe, występują największe bezwzględne różnice czasów wykonania między Rust a pozostałymi implementacjami. Jednocześnie, implementacje `tbb`, `old_omp` i `new_omp` wykazują w tych klasach bardziej regularne i przewidywalne skalowanie, osiągając relatywnie największe przyspieszenia.

W przypadku klas S i W obserwowane różnice między implementacjami są mniej wyraźne. Czas wykonania zmienia się tu w sposób mniej przewidywalny, a przyrosty wydajności wynikające z równoległości są znacznie słabsze. W klasie W można jednak zauważyc agresywne skalowanie w implementacjach `tbb`, `old_omp` i `new_omp`, co może świadczyć o tym, że rozmiar problemu lepiej współgra z architekturą pamięci podręcznej procesora, co z kolei przekłada się na wzrost efektywności.



Rys. 9.42: Analiza zmienności czasów wykonania benchmarku IS dla klas S, W, A, B względem liczby użytych wątków

Stabilność czasu wykonania

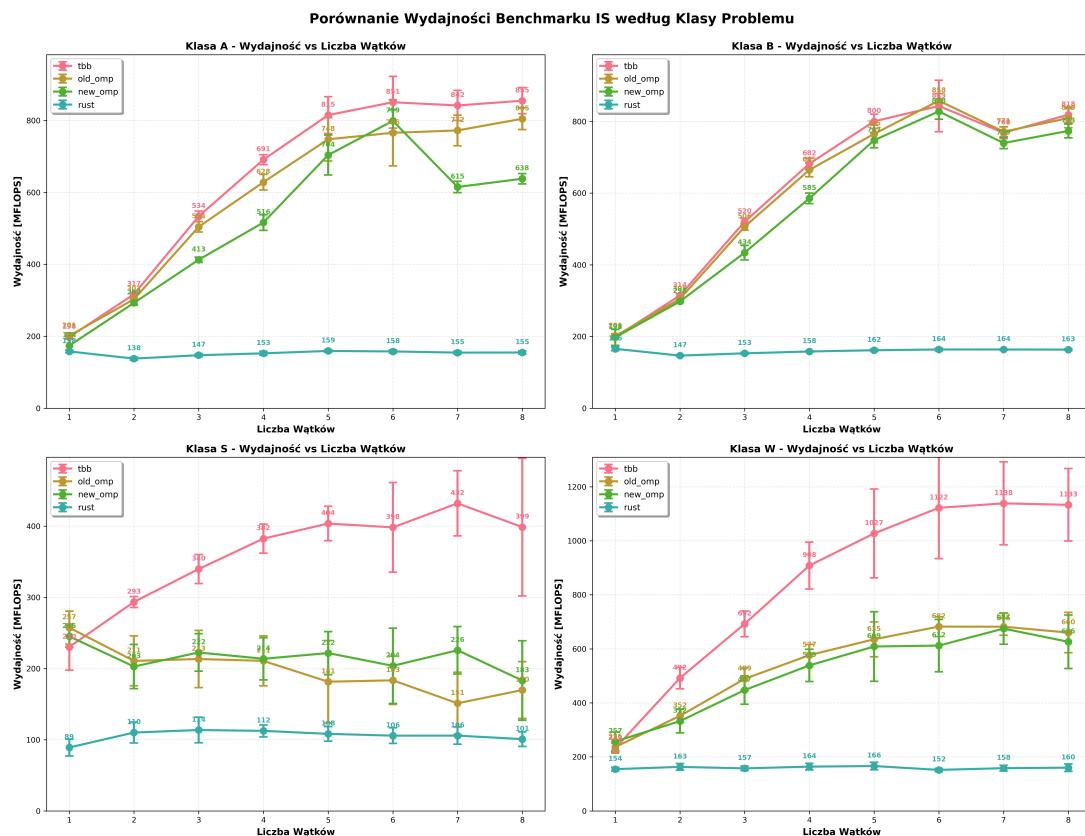
Pod względem zmienności czasów wykonania implementacja Rust wypada najsłabiej — osiąga najwyższą medianę współczynnika zmienności (około 0,04) i wykazuje szerszy rozrzut wyników, co wskazuje na większą podatność na losowe fluktuacje. Obecność licznych wartości odstających sugeruje występowanie okazjonalnych, znacznych odchyleń od typowych czasów wykonania, co może być związane z zarządzaniem wątkami bądź alokacją pamięci.

Implementacje `tbb`, `old_omp` i `new_omp` prezentują znacznie niższe i bardzo zbliżone wartości współczynnika zmienności, oscylujące wokół 0,025-0,03. Cechują się one także węższym zakresem wyników oraz mniejszą liczbą wartości odstających, co wskazuje na większą przewidywalność czasów wykonania i lepszą stabilność działania w środowisku wielowątkowym.

Stabilność MFLOPS

Pod względem zmienności MFLOPS najniższą medianę współczynnika zmienności (około 0,05) uzyskuje implementacja Rust, co oznacza, że pomimo dużych wahań czasów wykonania, osiągana wydajność obliczeniowa jest relatywnie stabilna.

Z drugiej strony, najwyższą medianę zmienności MFLOPS (około 0,10) osiąga implementacja `new_omp`, co wskazuje na znacznie większe wahania wydajności w czasie. `tbb` i `old_omp` plasują się w środku stawki, z wynikami na poziomie 0,06-0,07, przy czym `old_omp` wykazuje więcej wartości odstających, co może świadczyć o większej podatności na lokalne anomalie wydajnościowe.



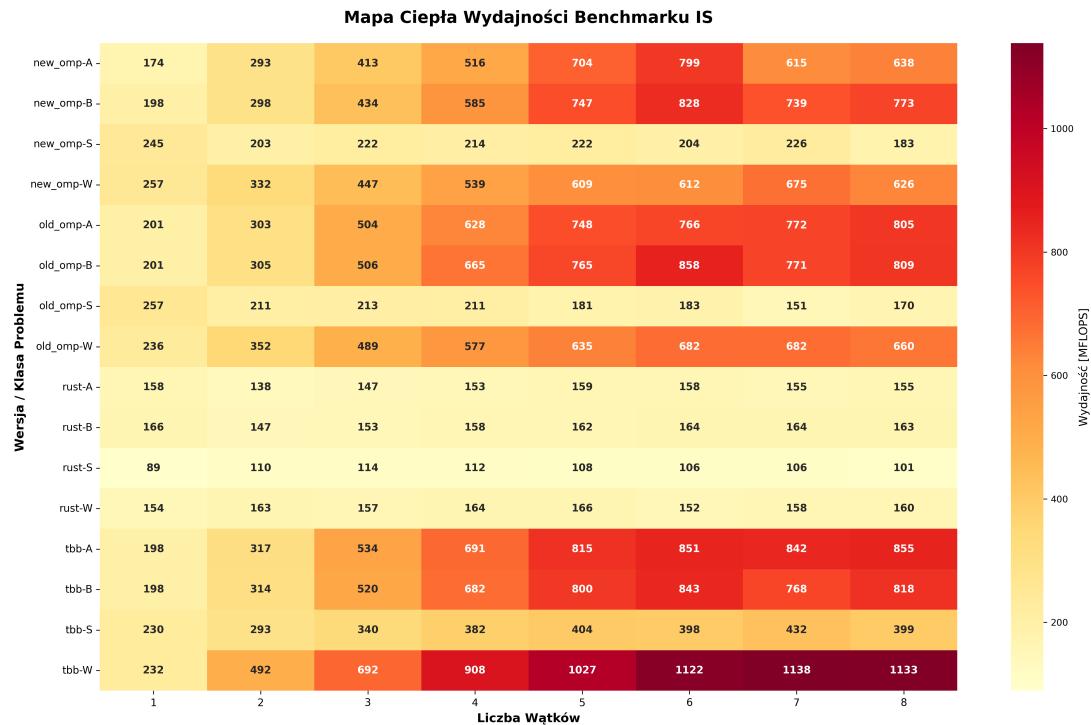
Rys. 9.43: Porównanie wydajności benchmarku IS dla klas S, W, A, B względem liczby użytych wątków

Na wykresie - rysunek 9.43 rzuca się w oczy wyraźna różnica w charakterystyce wydajnościowej implementacji Rust względem pozostałych rozwiązań. Podczas gdy implementacje `tbb`, `old_omp` i `new_omp` wykazują wyraźny wzrost wydajności wraz ze wzrostem liczby wątków, implementacja Rust utrzymuje niemal stały poziom - około 150-160 MFLOPS w klasach A i B - niezależnie od stopnia zrównoleglenia.

W całej analizie dominuje implementacja `tbb`, która osiąga najwyższe wartości wydajności we wszystkich klasach problemu, szczególnie w klasie W, gdzie przewaga nad rozwiązaniami opartymi na OpenMP sięga kilkuset MFLOPS. Wydajność `tbb` wzrasta stabilnie do około 5-6 wątków, po czym osiąga linię prostą lub delikatnie spada, co jest typowym zjawiskiem przy nasyceniu zasobów systemowych.

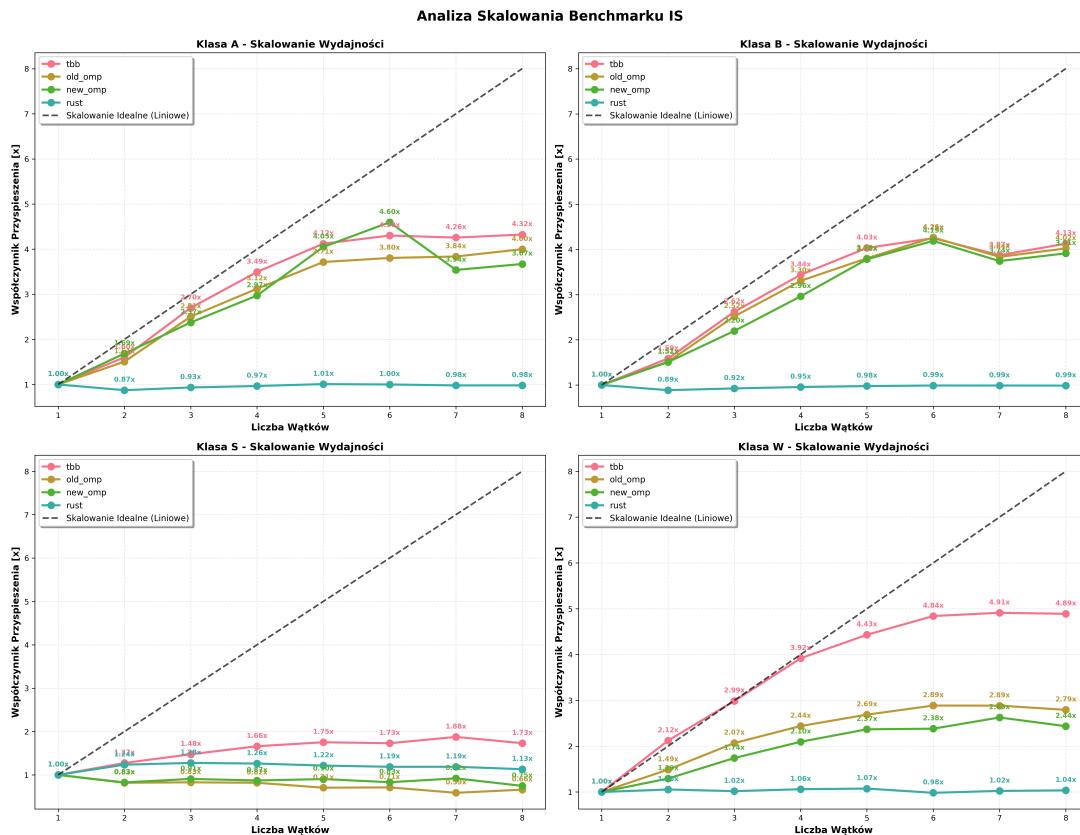
Klasy problemowe różnią się nie tylko poziomem bezwzględnej wydajności, lecz także charakterem skalowania. Klasy A i B zachowują się podobnie - osiągają wysoką wydajność i wykazują regularne skalowanie. Klasa S prezentuje niższe wartości i bardziej nieregularny przebieg, natomiast klasa W cechuje się najwyższymi szczytowymi wynikami, ale też największym zróżnicowaniem pomiędzy implementacjami.

9. Analiza wyników - programowanie równoległe



Rys. 9.44: Mapa ciepła wydajności benchmarku IS dla klas S, W, A, B względem liczby użytych wątków

Mapa ciepła - rysunek 9.44 stanowi syntetyczne podsumowanie analizowanych danych. Najwyższe wartości wydajności skupiają się wokół implementacji tbb, szczególnie w klasie W i przy większej liczbie wątków. Z kolei implementacja Rust pozostaje na jednolitym, niskim poziomie niezależnie od klasy problemu czy liczby wątków. Klasa S wyraźnie odstaje od pozostałych pod względem niskiej wydajności, a dla większości implementacji najwyższe wartości osiągane są w zakresie 6-8 wątków, co potwierdza wnioski płynące z analizy skalowania.



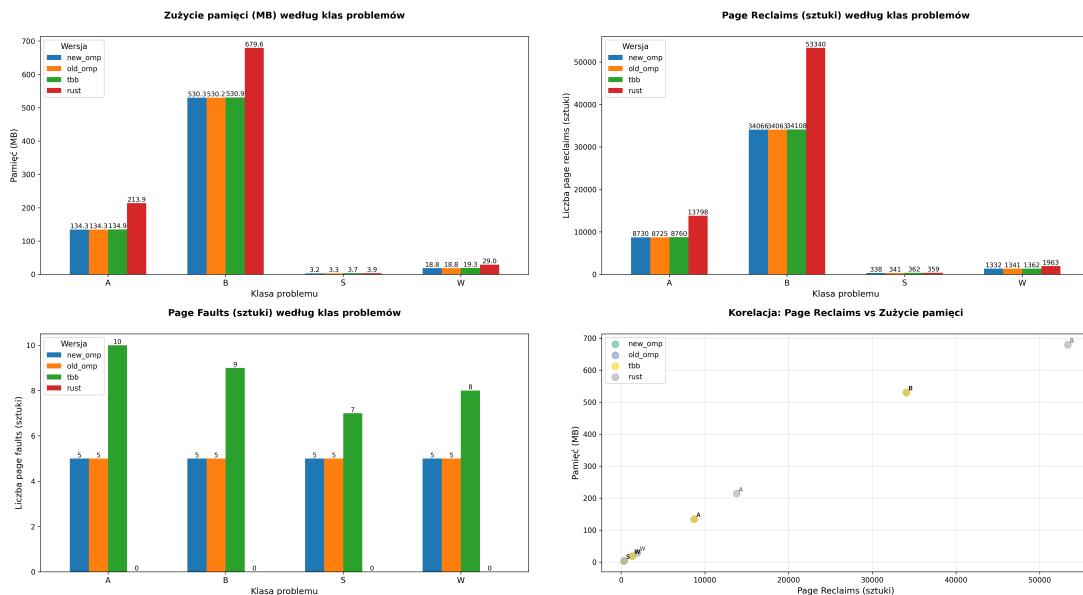
Rys. 9.45: Analiza skalowania benchmarku IS dla klas S, W, A, B względem liczby użytych wątków

Wszystkie analizowane implementacje na wykresie - rysunek 9.45 wykazują odchylenie od idealnego skalowania liniowego. Nawet najlepszy wynik - przyspieszenie rzędu 4,8x dla implementacji **tbb** w klasie W przy 6-8 wątkach - stanowi zaledwie około 60% teoretycznego maksimum. Spośród wszystkich badanych rozwiązań to właśnie **tbb** osiąga najefektywniejsze skalowanie we wszystkich klasach, szczególnie w klasie W, gdzie zdecydowanie dominuje nad konkurencją. Implementacje OpenMP (**old_omp** i **new_omp**) prezentują umiarkowane przyspieszenia, rzędu 3,8-4,3x w większych klasach problemowych (A i B), natomiast Rust praktycznie nie skaluje się wcale - współczynnik przyspieszenia oscyluje wokół 1, niezależnie od liczby wątków. Skalowanie silnie zależy również od klasy problemu. Klasy A i B charakteryzują się najbardziej efektywnym, choć wciąż dalekim od idealnego, wykorzystaniem równoległości (do około 4,6x). Klasa S wypada najsłabiej, nie przekraczając przyspieszenia 2x, co wynika z małej skali problemu niedostosowanej do architektury wielowątkowej. W klasie W obserwujemy najwyższe zróżnicowanie między implementacjami, przy czym **tbb** wyraźnie się wyróżnia. Dla większości implementacji punkt nasycenia wydajności przypadła na 5-6 wątków, po czym następuje stabilizacja lub lekki spadek. Wskazuje to na osiągnięcie limitów przepustowości pamięci lub narzutów związanych z synchronizacją w środowisku wielowątkowym.

9.3.3. Wyniki profilowania wydajności - platforma ARM64

Aby lepiej zrozumieć charakterystyki pamięciowe testowanych implementacji benchmarku IS, przeprowadzono analizę zużycia pamięci, błędów stron pamięci oraz mechanizmu ich odzyskiwania. Wnioski wyciągnięto na podstawie trzech grup wykresów: bezwzględnych wartości (rys. 9.46), porównania względnego względem **old_omp** (rys. 9.47) oraz kompromisów między metrykami (rys. 9.48).

9. Analiza wyników - programowanie równoległe



Rys. 9.46: Profilowanie wydajności benchmarku IS dla klas S, W, A, B względem liczby użytych wątków

Zużycie pamięci i błędy stron

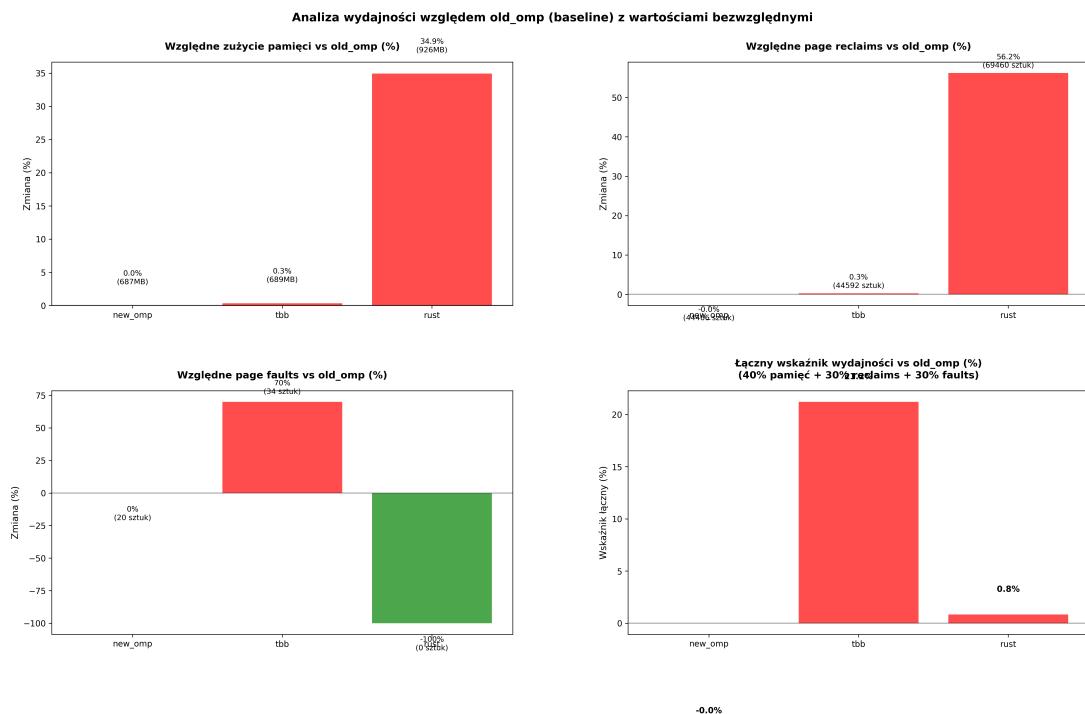
Wykresy w górnym rzędzie przedstawiają bezwzględne zużycie pamięci oraz liczbę odzyskanych stron pamięci:

- Największe zużycie pamięci odnotowano dla implementacji **rust**, osiągając 213,9 MB (klasa A) oraz 679,6 MB (klasa B). W pozostałych implementacjach (**new_omp**, **old_omp**, **tbb**) wartości te były niemal identyczne (np. 134 MB w klasie A i 530 MB w klasie B).
- Analogicznie, **rust** odzyskuje najwięcej stron, osiągając ponad 53 tysiące operacji w klasie B, co jest o ponad 50% więcej niż w pozostałych wersjach. Sugeruje to intensywniejsze wykorzystanie dynamicznego zarządzania pamięcią lub częstsze zwalnianie obszarów.
- W zakresie błędów stron, **rust** jako jedyna implementacja nie generuje żadnych błędów stron (0 sztuk we wszystkich klasach), co potwierdza jej deterministyczne podejście do alokacji pamięci.
- Wersje **new_omp** i **old_omp** utrzymują stabilny poziom błędów (5 sztuk we wszystkich klasach), natomiast **tbb** wykazuje wyraźnie wyższy poziom błędów - od 7 do 10 sztuk.

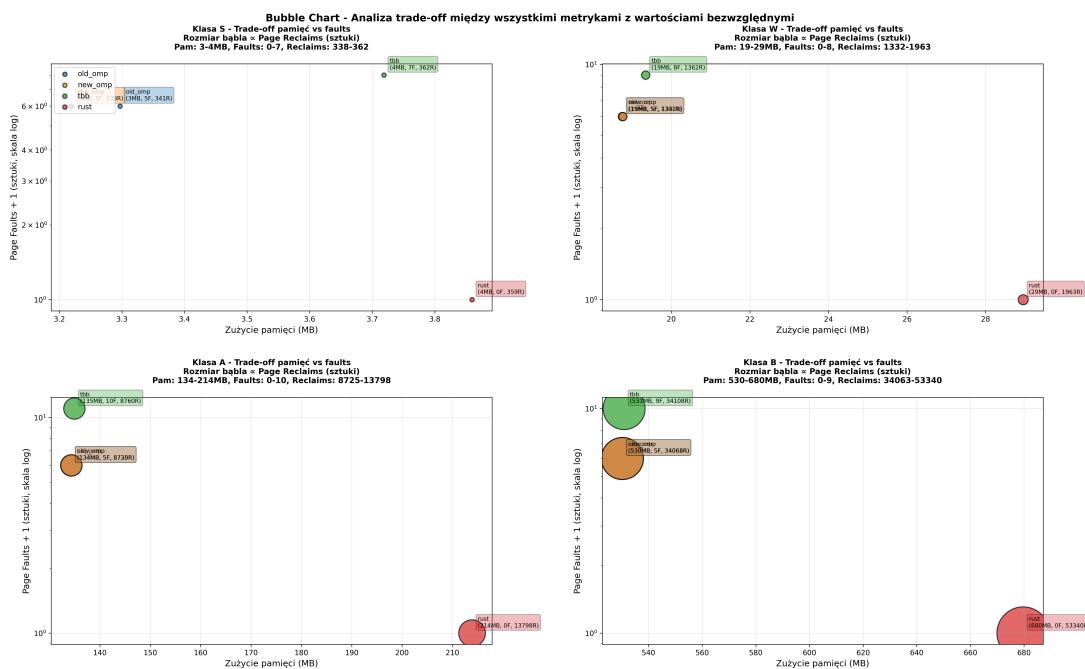
W prawym dolnym wykresie zaprezentowano korelację między zużyciem pamięci a liczbą oddanych stron. Wykres ujawnia wyraźną korelację dodatnią - implementacje i klasy, które zużywają więcej pamięci, zwalniają również więcej stron z pamięcią, co wynika z konieczności intensywniejszej obsługi zarządzania pamięcią operacyjną.

Wykres ten - rysunek 9.47 przedstawia zmiany procentowe trzech kluczowych metryk względem wersji odniesienia (**old_omp**):

- Zużycie pamięci: **new_omp** nie różni się w ogóle od wersji odniesienia (0%), **tbb** wykazuje jedynie symboliczny wzrost (0,3%), natomiast **rust** zużywa aż 34,9% więcej pamięci.
- Odzyskiwanie stron pamięci: **rust** generuje aż 56,2% więcej operacji odzyskania niż **old_omp**, co stanowi znaczące obciążenie podsystemu pamięci.
- Błędy stron: **rust** jako jedyna implementacja całkowicie je eliminuje (-100% względem **old_omp**), co jest jej największym atutem oraz potwierdza założenia co do mechanizmów języka.
- Wskaźnik zbiorczy (ważony: 40% pamięć + 30% claims + 30% faults) wskazuje, że jedynie **tbb** osiąga nieznaczłą przewagę względem wersji bazowej (+0,8%), natomiast **rust** wypada niekorzystnie pomimo doskonałej obsługi błędów, głównie przez nadmierne zużycie pamięci.



Rys. 9.47: Analiza wydajności względem old_omp (punkt odniesienia) z wartościami bezwzględnymi

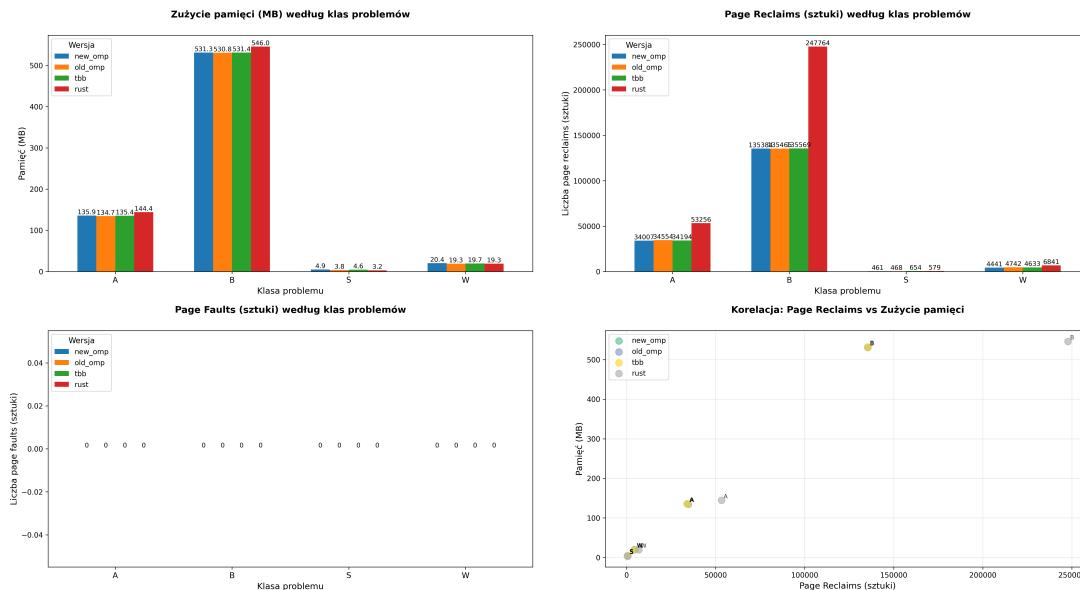


Rys. 9.48: Kompromisy (ang. *trade-off*) pomiędzy zużyciem pamięci a błędami stron pamięci, z uwzględnieniem liczby odzyskanych stron jako trzeciej zmiennej reprezentowanej przez rozmiar bąbla

Na podstawie wykresu bąbelkowego - rysunek 9.48 można zauważyć:

- **rust** w każdej klasie lokuje się w dolnym rejonie wykresu (niska liczba błędów) przy wysokim zużyciu pamięci oraz największych bąblach (najwięcej zwolnień stron).
- **tbb** regularnie generuje najwyższą liczbę błędów stron przy zbliżonym zużyciu pamięci jak **old_omp** i **new_omp**.
- **new_omp** uzyskuje relatywnie zrównoważony kompromis - umiarkowane zużycie pamięci, średnia liczba błędów stron i średnia liczba odzyskanych stron.

9.3.4. Wyniki profilowania wydajności - platforma x86_64



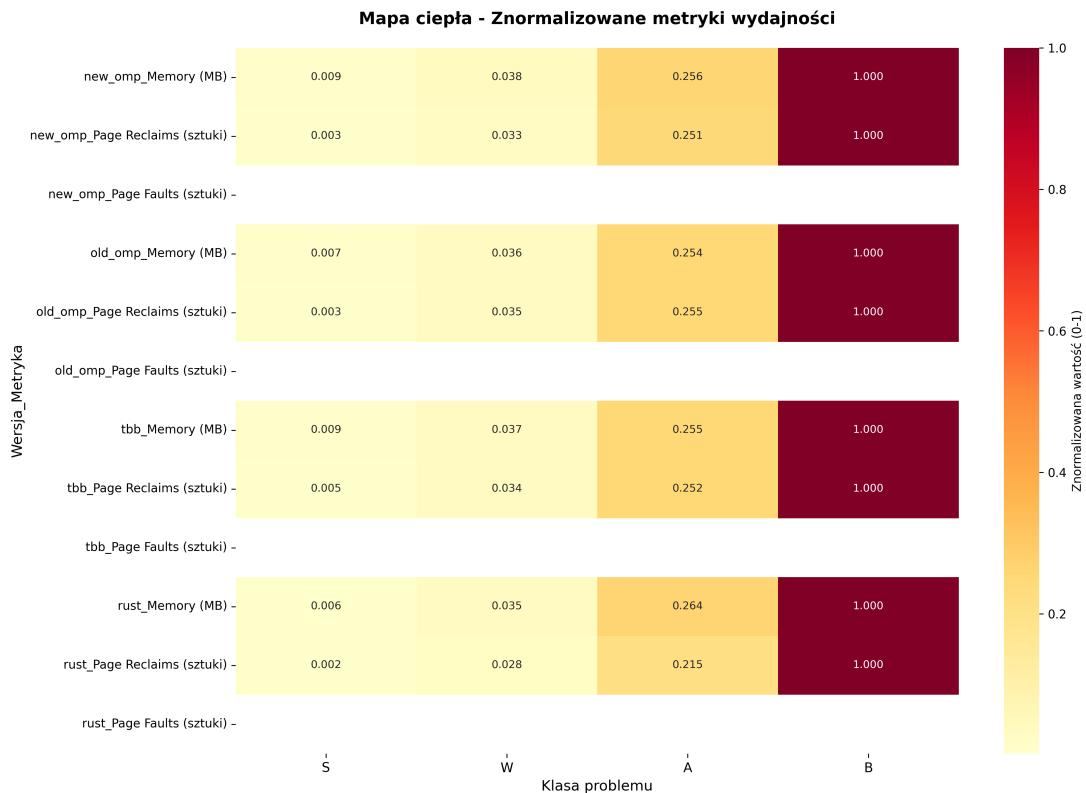
Rys. 9.49: Profilowanie wydajności benchmarku IS dla klas S, W, A, B względem liczby użytych wątków

Analiza wykresów dotyczących zużycia pamięci - rysunek 9.49 ujawnia wyraźne różnice w zapotrzebowaniu na zasoby w zależności od klasy problemu, przy jednoczesnym niewielkim zróżnicowaniu pomiędzy poszczególnymi implementacjami. Największe zużycie pamięci zaobserwowano dla klasy B, co jest bezpośrednio związane z największym rozmiarem sortowanych danych. W tej klasie implementacja w języku Rust wyróżnia się nieznacznie wyższym zużyciem pamięci (546 MB) w porównaniu do pozostałych (531-533 MB). Klasa A wykazuje umiarkowane zapotrzebowanie na pamięć, miesiące się w zakresie 135-144 MB, z ponownym wskazaniem na implementację Rust jako najbardziej zasobozerną (144,4 MB). Dla klas S i W, ze względu na mniejszy rozmiar problemu, zużycie pamięci jest znaczco niższe i wynosi odpowiednio 3,2-4,9 MB oraz 19,3-20,4 MB. Co interesujące, w przypadku klasy S to właśnie implementacja Rust osiąga najniższe zużycie pamięci (3,2 MB).

Wykresy przedstawiające operacje pamięci wirtualnej (zwolnienia oraz błędy stron) - 9.49 dostarczają informacji o efektywności zarządzania pamięcią. Szczególnie widoczna jest duża liczba zwolnień stron pamięci w klasie B, gdzie implementacja Rust osiąga wartość 247 764, znacznie przewyższając inne implementacje oscylujące wokół 135 000-136 000. Podobna tendencja występuje w klasie A - Rust generuje 53 256 zwolnień stron, podczas gdy pozostałe implementacje utrzymują się na poziomie 34 000-35 000. W klasach S i W liczby te są proporcjonalnie mniejsze, co odpowiada ich ograniczonym wymaganiom pamięciowym.

Z kolei brak błędów stron we wszystkich implementacjach i klasach problemu jest jednoznacznym dowodem na skuteczne zarządzanie pamięcią przez system operacyjny. Brak konieczności odwołań do pamięci wymiany oznacza, że całość operacji wykonywana była w pamięci fizycznej, bez negatywnego wpływu na wydajność.

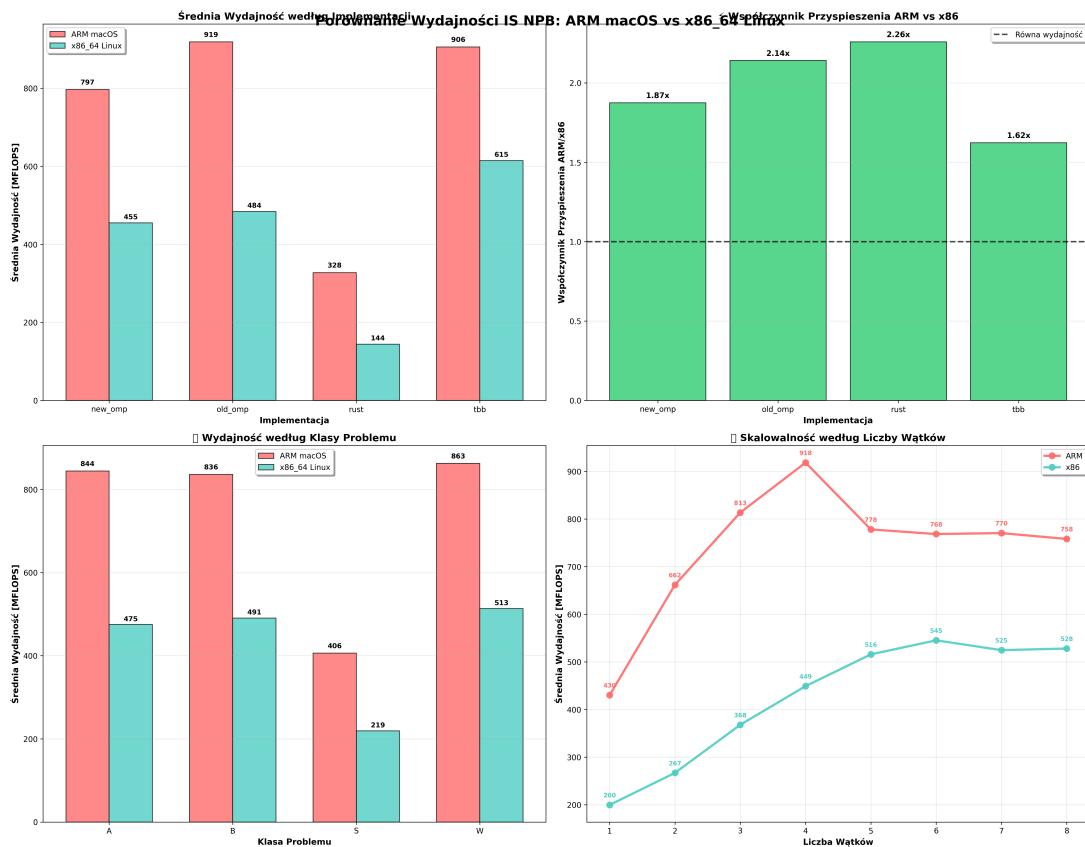
Wykres korelowy między liczbą zwolnień stron a zużyciem pamięci wskazuje na silną zależność między tymi dwiema metrykami. Dane tworzą wyraźne grupy odpowiadające klasom problemu - od klasy B z najwyższym zużyciem pamięci, po klasę S z najniższym. W ramach każdej grupy implementacja Rust wykazuje tendencję do generowania większej liczby reklamacji stron przy podobnym zużyciu pamięci, co może świadczyć o jej specyficznych podejściu do alokacji i zwalniania pamięci - bardziej intensywnym, lecz potencjalnie korzystnym dla innych aspektów wydajności.



Rys. 9.50: Znormalizowana mapa ciepła metryk wydajności

Znormalizowana mapa ciepła metryk wydajności - rysunek 9.50 potwierdza powyższe obserwacje, ukazując proporcjonalne różnice między klasami problemu i implementacjami. Wzorzec ten nie tylko podkreśla wpływ rozmiaru problemu na charakterystyki pamięciowe, ale również zwraca uwagę na unikalne właściwości implementacji Rust, której strategia zarządzania pamięcią istotnie różni się od pozostałych rozwiązań.

9.3.5. Porównanie pomiędzy platformami



Rys. 9.51: Porównanie średniej wydajności benchmarku IS dla platform ARM64 i x86_64

Analiza średniej wydajności według implementacji

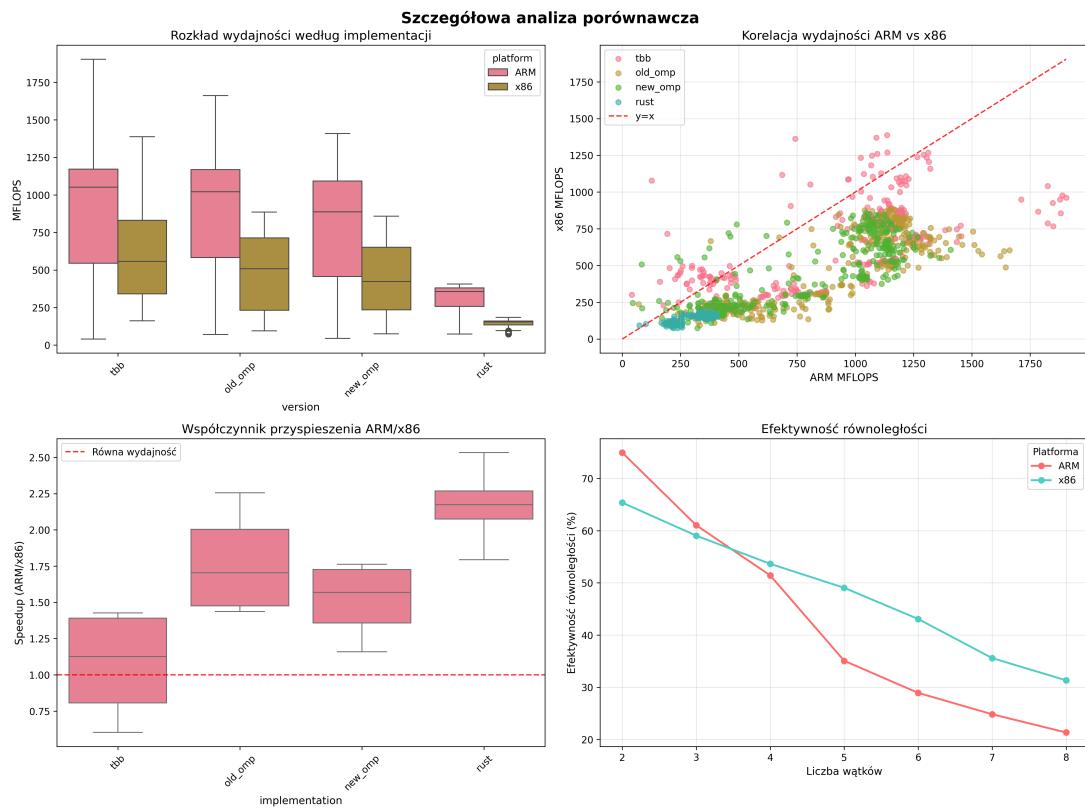
Górny lewy wykres drugiego zestawu - rysunek 9.51 prezentuje średnią wydajność dla różnych implementacji algorytmów. We wszystkich badanych przypadkach (new_omp, old_omp, rust, tbb) architektura ARM macOS osiąga znaczco wyższe wartości MFLOPS niż architektura x86_64 Linux. Szczególnie wyraźna przewaga ARM występuje w implementacjach old_omp (919 a 484 MFLOPS) oraz tbb (906 a 615 MFLOPS).

Współczynnik przyspieszenia widoczny w prawym górnym wykresie - 9.51 potwierdza tę przewagę, wskazując że implementacja w języku Rust osiąga najwyższy współczynnik przyspieszenia (2,34x), a tuż za nią plasuje się implementacja tbb (2,26x). Wszystkie badane implementacje wykazują wartości współczynnika przyspieszenia powyżej 1,0, co potwierdza systematyczną przewagę wydajnościową platformy ARM.

Analiza wydajności według klasy problemu i skalowalności

Lewy dolny wykres drugiego zestawu - rysunek 9.51 ilustruje, że przewaga wydajnościowa ARM macOS utrzymuje się konsekwentnie we wszystkich klasach problemów (A, B, C oraz W). Warto zauważyć, że różnica wydajnościowa jest proporcjonalna do złożoności problemu.

Analiza skalowalności według liczby wątków (prawy dolny wykres) pokazuje, że obie platformy wykazują wzrost wydajności wraz ze zwiększaniem liczby wątków do około 4-5, po czym następuje stabilizacja lub lekki spadek. ARM macOS konsekwentnie utrzymuje wyższą bezwzględną wydajność, przy czym najwyższa wartość przypada na 4 wątki (913 MFLOPS dla ARM a 518 MFLOPS dla x86_64).



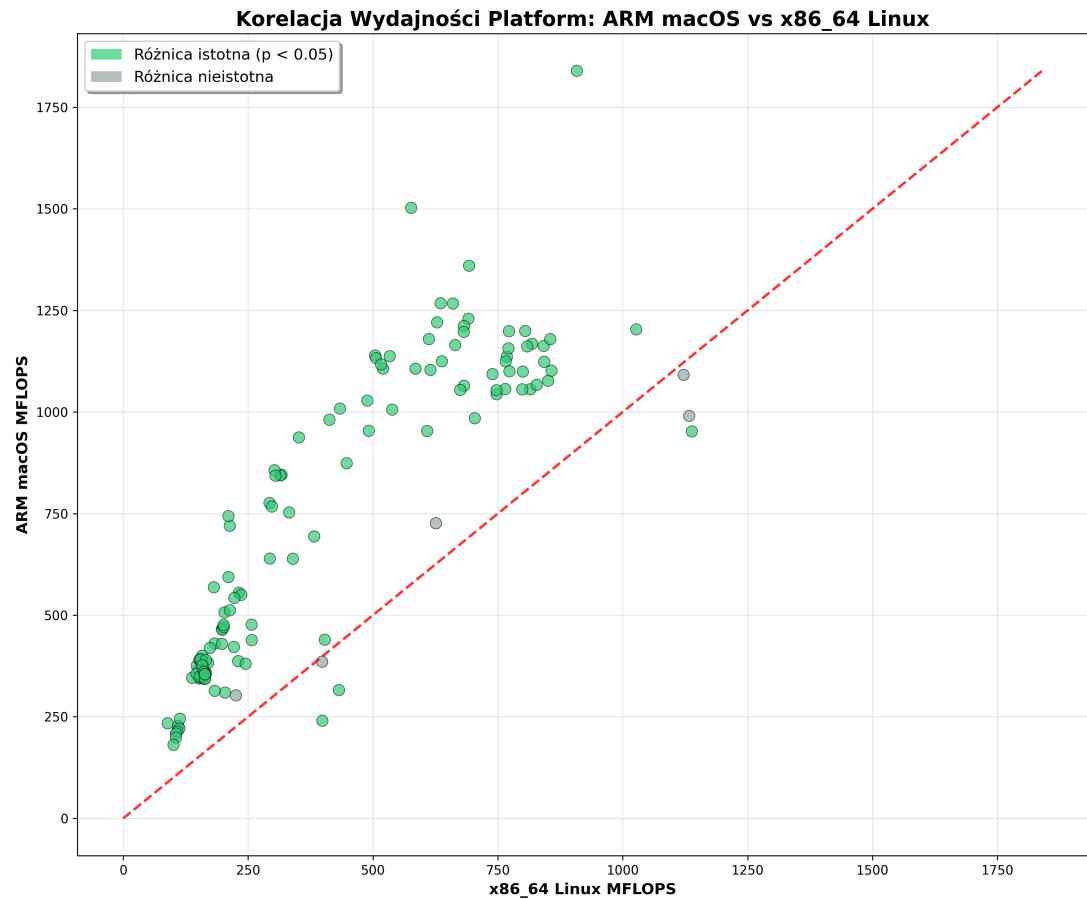
Rys. 9.52: Szczegółowa analiza wydajności benchmarku IS dla platform ARM64 i x86_64

Wykresy pudełkowe - rysunek 9.52 potwierdzają, że mediana wydajności dla platformy ARM jest systematycznie wyższa niż dla x86_64 we wszystkich implementacjach. Jednocześnie platforma ARM charakteryzuje się większym rozrzutem wyników, co sugeruje większą zmienność wydajności.

Wykres korelacji wydajności ARM a x86 z kodowaniem kolorystycznym różnych implementacji dodatkowo potwierdza, że większość punktów pomiarowych znajduje się powyżej linii równej wydajności, przy czym różne implementacje grupują się w odmiennych obszarach wydajnościowych.

Analiza współczynnika przyspieszenia wskazuje, że implementacja Rust osiąga najwyższy medianowy współczynnik przyspieszenia (około 2,2x), natomiast implementacja tbb wykazuje największą zmienność, z niektórymi przypadkami osiągającymi wartości poniżej 1,0.

Ostatni wykres przedstawia efektywność równoległości w zależności od liczby wątków. Architektura ARM rozpoczyna z wyższą efektywnością przy małej liczbie wątków, jednak doświadcza szybszego spadku efektywności wraz ze wzrostem liczby wątków. Przy 8 wątkach obie platformy osiągają efektywność około 30%, przy czym x86_64 wypada nieznacznie lepiej, co wskazuje na lepsze zachowanie skalowalności przy wyższej liczbie wątków.



Rys. 9.53: Analiza istotności statystycznej benchmarku IS dla platform ARM64 i x86_64

Przedstawiony na wykresie rozkład punktów obrazuje korelację między wydajnością (mierzoną w MFLOPS - miliony operacji zmiennoprzecinkowych na sekundę) - rysunek 9.52 na platformie ARM macOS względem x86_64 Linux. Punkty oznaczone kolorem zielonym wskazują na statystycznie istotne różnice ($p < 0,05$), natomiast punkty szare reprezentują różnice nieistotne statystycznie. Czerwona przerywana linia reprezentuje teoretyczną równą wydajność obu platform ($y=x$). Zdecydowana większość punktów znajduje się powyżej tej linii, co jednoznacznie wskazuje na przewagę wydajnościową architektury ARM macOS nad x86_64 Linux w badanych przypadkach.

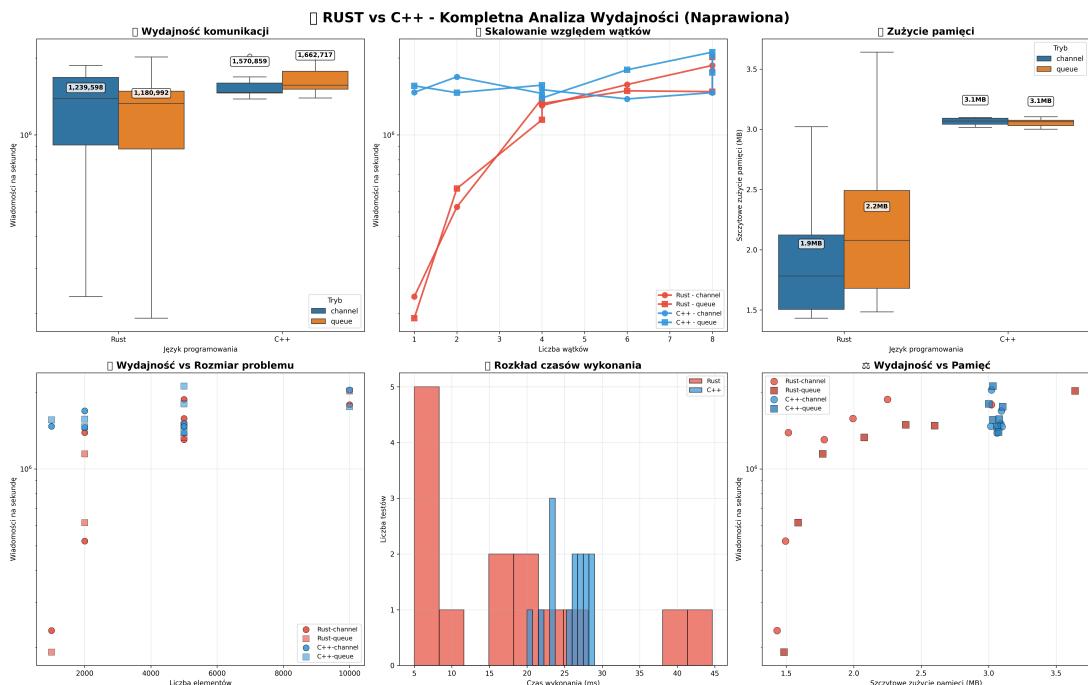
Rozdział 10

Analiza wyników - programowanie współbieżne

W tym rozdziale zostały zebrane wyniki testów, które porównują ze sobą mechanizmy programowania współbieżnego w językach Rust i C++.

10.1. Benchmark producent-klient

10.1.1. Wyniki benchmarków - platforma ARM64



Rys. 10.1: Analiza wyników benchmarku producent-klient

Wydajność komunikacji międzywątkowej

Wykres w lewym górnym rogu - rysunek 10.1 przedstawia porównanie przepustowości mechanizmów komunikacji międzywątkowej, wyrażonej liczbą przetworzonych wiadomości na sekundę, dla implementacji w językach Rust i C++. Z analizy wykresów pudełkowych wynika, że implementacje w języku C++ charakteryzują się wyższą medianą wydajności w porównaniu do rozwiązań w języku Rust.

Dla implementacji wykorzystującej mechanizm kanału (oznaczony kolorem niebieskim) mediana przepustowości w języku C++ wynosi około 1 570 859 wiadomości na sekundę, podczas gdy analogiczna implementacja w języku Rust osiąga 1 239 598 wiadomości na sekundę, co oznacza przewagę C++ o około 27%. Podobną zależność obserwujemy dla mechanizmu kolejki (oznaczonego kolorem pomarańczowym), gdzie C++ osiąga 1 662 717 wiadomości na sekundę wobec 1 180 992 dla języka Rust, co daje przewagę wynoszącą około 41%.

Warto zauważać, że w przypadku języka C++ implementacja kolejki wykazuje wyższą wydajność niż implementacja kanału, natomiast w języku Rust zależność jest odwrotna. Może to wskazywać na różnice w optymalizacji tych mechanizmów w obu językach.

Skalowanie względem liczby wątków

Środkowy górny wykres - rysunek 10.1 ilustruje zależność wydajności od liczby wykorzystywanych wątków. Implementacje w językach Rust i C++ charakteryzują się diametralnie odmiennym wzorcem skalowania. Implementacje C++ (oznaczone kolorem niebieskim) utrzymują względnie stabilną wydajność niezależnie od liczby wątków, z niewielką tendencją wzrostową dla 7-8 wątków w przypadku mechanizmu kolejki.

Natomiast implementacje w języku Rust (oznaczone kolorem czerwonym) rozpoczynają od znaczco niższej wydajności przy 1-2 wątkach, ale wykazują gwałtowny wzrost przepustowości przy zwiększeniu liczby wątków do 4, osiągając wartości zbliżone do implementacji w C++. Ta charakterystyka skalowania sugeruje, że implementacje w języku Rust są zoptymalizowane pod kątem efektywnego wykorzystania równoległości, podczas gdy implementacje w C++ oferują bardziej przewidywalną wydajność niezależną od stopnia paralelizmu.

Zużycie pamięci

Prawy górny wykres - rysunek 10.1 przedstawia porównanie zużycia pamięci, mierzonego w megabajtach, dla badanych implementacji. W tym aspekcie język Rust wykazuje wyraźną przewagę, osiągając niższe mediany zużycia pamięci zarówno dla mechanizmu kanału (1,9 MB wobec 3,1 MB w C++), jak i mechanizmu kolejki (2,2 MB wobec 3,1 MB w C++).

Ta różnica, wynosząca około 40% dla mechanizmu kanału i 29% dla mechanizmu kolejki, może być przypisana efektywnemu systemowi zarządzania pamięcią w języku Rust, bazującemu na analizie czasu życia i systemie własności, które eliminują konieczność stosowania kosztownych mechanizmów automatycznego odzyskiwania pamięci lub podatność na wycieki pamięci charakterystyczne dla ręcznego zarządzania pamięcią w C++.

Rozkład czasów wykonania

Środkowy dolny histogram - rysunek 10.1 prezentuje rozkład częstości czasów wykonania dla implementacji w językach Rust (kolor czerwony) i C++ (kolor niebieski). Analiza tego wykresu ujawnia fundamentalne różnice w charakterystykach wydajnościowych obu języków.

Implementacje w języku Rust charakteryzują się bimodalnym rozkładem czasów wykonania, z dominującym szczytem przy około 5 ms oraz mniejszymi skupiskami w okolicach 15-20 ms i 40-45 ms. Taka dystrybucja może świadczyć o różnych strategiach optymalizacji w zależności od specyfiki zadania lub wpływie dodatkowych mechanizmów zarządzania pamięcią.

W przeciwnieństwie do tego, czasy wykonania dla implementacji C++ koncentrują się głównie w przedziale 20-25 ms, z mniejszą wariancją, co wskazuje na bardziej przewidywalną wydajność, potencjalnie kosztem utraconych możliwości optymalizacji w określonych scenariuszach.

Wydajność względem zużycia pamięci

Prawy dolny wykres - rysunek 10.1 przedstawia relację między wydajnością a szczytowym zużyciem pamięci. Punkty danych tworzą wyraźnie odrębne skupiska dla implementacji w języku Rust (punkty czerwone) i C++ (punkty niebieskie). Implementacje C++ charakteryzują się wyższą wydajnością przy jednoczesnym wyższym zużyciu pamięci, koncentrując się w górnym prawym obszarze wykresu.

Z kolei implementacje w języku Rust wykazują większe rozproszenie, zarówno pod względem wydajności, jak i zużycia pamięci, co sugeruje większą adaptacyjność tych implementacji do różnych warunków wykonania. Szczególnie interesujące są pojedyncze punkty dla implementacji Rust o bardzo niskim zużyciu pamięci (około 1,5 MB) i jednocześnie niskiej wydajności, które mogą reprezentować skrajne przypadki optymalizacji pod kątem oszczędności zasobów kosztem wydajności.

Podsumowanie

Przeprowadzona analiza porównawcza mechanizmów współbieżności w językach Rust i C++ ujawnia istotne kompromisy wydajnościowe. Implementacje C++ oferują generalnie wyższą przepustowość, szczególnie przy mniejszej liczbie wątków i mniejszych rozmiarach problemu, jednak kosztem zwiększonego zużycia pamięci. Z kolei implementacje w języku Rust charakteryzują się niższym zużyciem zasobów pamięciowych oraz znakomitym skalowaniem wydajności wraz ze wzrostem stopnia równoległości, choć przy niskim poziomie paralelizmu ustępują wydajnością rozwiązaniom w C++.

Wyniki badań sugerują, że wybór między tymi językami powinien być podejmowany specyfiką zastosowania: w systemach o ograniczonych zasobach pamięciowych lub wymagających wysokiej skalowalności Rust może oferować lepsze rozwiązania, natomiast w zastosowaniach priorytetyzujących maksymalną przepustowość przy niewielkiej liczbie wątków lub przewidywalność czasów wykonania implementacje C++ mogą stanowić bardziej odpowiedni wybór.

Warto również zauważyć, że w obu językach mechanizm kolejki wykazuje odmienne charakterystyki wydajnościowe w porównaniu do mechanizmu kanału, co dodatkowo podkreśla złożoność decyzji projektowych w kontekście programowania współbieżnego.

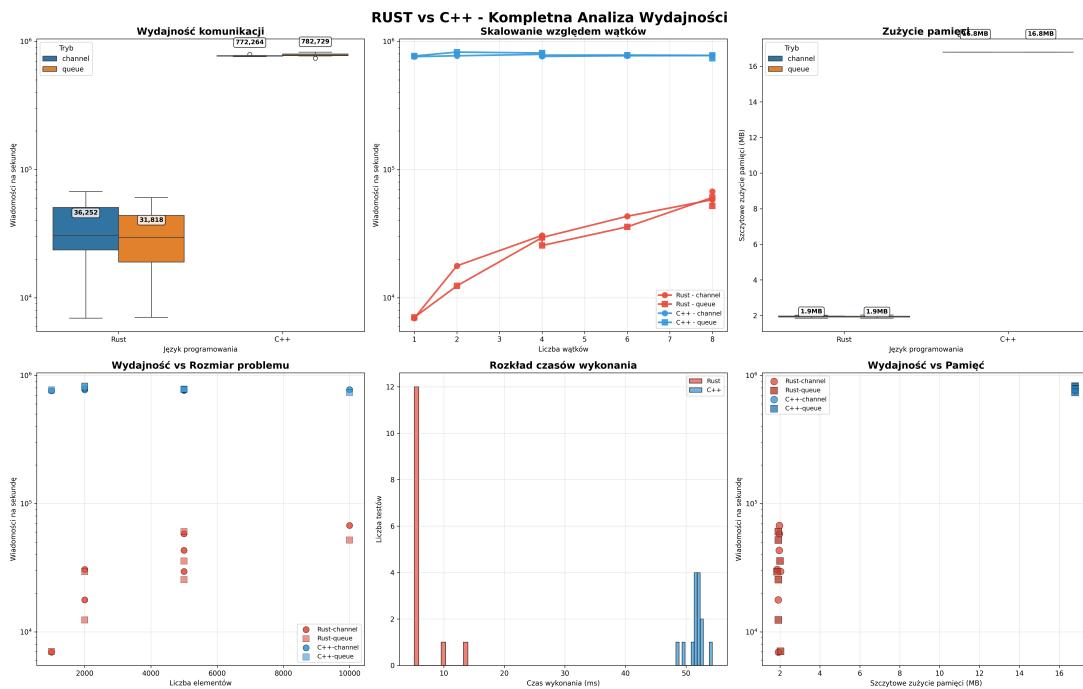
10.1.2. Wyniki benchmarków - platforma x86_64

Wydajność komunikacji międzywątkowej

Wykres w lewym górnym rogu - rysunek 10.2 przedstawia porównanie przepustowości mechanizmów komunikacji międzywątkowej, wyrażonej liczbą wiadomości przetwarzanych na sekundę. Analiza wykresu pudełkowego wskazuje na znaczącą różnicę wydajności między implementacjami w językach C++ i Rust.

Implementacje w języku C++ osiągają znaczaco wyższą przepustowość, z medianą na poziomie 772 264 wiadomości/s dla mechanizmu kanału (kolor niebieski) i 782 729 wiadomości/s dla mechanizmu kolejki (kolor pomarańczowy). W przeciwieństwie do tego, implementacje w języku Rust osiągają przepustowość rzędu 36 252 wiadomości/s dla mechanizmu kanału i 31 818 wiadomości/s dla mechanizmu kolejki.

Ta dysproporcja, wyrażająca się współczynnikiem około 21:1 na korzyść C++, wskazuje na fundamentalne różnice w wydajności bazowych mechanizmów komunikacji międzywątkowej w obu językach. Warto zauważyć, że rozrzut wartości (wysokość prostokątów) jest znacznie większy w przypadku implementacji Rust, co sugeruje mniej przewidywalną wydajność w porównaniu do bardziej stabilnych wyników w C++.



Rys. 10.2: Analiza wyników benchmarku producent-klient

Skalowanie względem liczby wątków

Środkowy górny wykres - rysunek 10.2 ilustruje zależność pomiędzy liczbą wykorzystywanych wątków a osiąganą przepustowością komunikacji. Różnice w charakterystyce skalowania między językami są uderzające. Implementacje w języku C++ (niebieskie linie) utrzymują stałą, wysoką wydajność na poziomie około 10⁵ wiadomości/s, praktycznie niezależnie od liczby wykorzystywanych wątków.

W przeciwieństwie do tego, implementacje w języku Rust (czerwone linie) wykazują wyraźną zależność od liczby wątków. Rozpoczynając od wydajności rzędu 6 × 10³ wiadomości/s przy jednym wątku, osiągają około 7 × 10⁴ wiadomości/s przy ośmiu wątkach, co stanowi ponad dziesięciokrotną poprawę. Pomimo to znaczącego wzrostu wydajności, nawet przy maksymalnej liczbie wątków implementacje Rust nie osiągają poziomu wydajności oferowanego przez C++.

Taka charakterystyka może wskazywać na odmienne podejście do optymalizacji w obu językach: C++ może być zoptymalizowany pod kątem maksymalnej przepustowości pojedynczego wątku, podczas gdy Rust może priorytetyzować efektywne wykorzystanie wielu rdzeni kosztem wydajności pojedynczego wątku.

Zużycie pamięci

Prawy górny wykres - rysunek 10.2 przedstawia porównanie zużycia pamięci dla analizowanych implementacji. W przeciwieństwie do wyników wydajnościowych, to język Rust wykazuje zdecydowaną przewagę pod względem efektywności wykorzystania zasobów pamięciowych. Zarówno implementacja wykorzystująca kanał, jak i implementacja wykorzystująca kolejkę w języku Rust wykazują zużycie pamięci na poziomie 1,9 MB.

Dla porównania, obie implementacje w języku C++ charakteryzują się zużyciem pamięci na poziomie 16,8 MB, co stanowi prawie dziewięciokrotnie wyższe zapotrzebowanie na pamięć w porównaniu do rozwiązań w Rust. Ta różnica wynika prawdopodobnie z odmiennych podejść do zarządzania pamięcią: Rust wykorzystuje system własności i analizę czasu życia obiektów, co pozwala na bardziej efektywne zarządzanie zasobami pamięciowymi.

Wydajność względem rozmiaru problemu

Lewy dolny wykres - rysunek 10.2 ilustruje zależność wydajności od rozmiaru problemu, wyrażonego liczbą przetwarzanych elementów. Ponownie obserwujemy wyraźną przewagę wydajnościową implementacji C++ (niebieskie punkty) nad implementacjami Rust (czerwone punkty) w całym spektrum badanych rozmiarów problemu.

Implementacje C++ utrzymują stałą, wysoką wydajność rzędu 10^5 wiadomości/s niezależnie od rozmiaru problemu. Implementacje Rust wykazują ogólnie niższą wydajność rzędu 10^1 do 10^2 wiadomości/s, z nieznaczną tendencją wzrostową wraz ze zwiększeniem rozmiaru problemu, szczególnie widoczną przy przejściu z 2000 do 6000 elementów. Może to sugerować, że koszty inicjalizacji i zarządzania w Rust są amortyzowane przy większych rozmiarach problemu.

Rozkład czasów wykonania

Środkowy dolny histogram - rysunek 10.2 prezentuje rozkład czasów wykonania dla implementacji w językach Rust (kolor czerwony) i C++ (kolor niebieski). Zaskakujące jest to, że mimo niższej przepustowości komunikacji, implementacje w języku Rust charakteryzują się znacznie krótszymi czasami wykonania, z dominującym skupiskiem w okolicach 5-10 ms.

W przeciwieństwie do tego, czasy wykonania dla implementacji C++ koncentrują się w przedziale 50-55 ms. Ta pozorna sprzeczność między krótkimi czasami wykonania a niższą przepustowością komunikacji w języku Rust może wskazywać na różnice w metodologii pomiaru lub na wyższe narzuty związane z inicjalizacją komunikacji w Rust, które nie są uwzględniane w bezpośrednim pomiarze czasu wykonania.

Wydajność względem zużycia pamięci

Prawy dolny wykres - rysunek 10.2 przedstawia relację między wydajnością a zużyciem pamięci. Punkty danych tworzą wyraźnie odrębne skupiska dla implementacji w języku Rust (czerwone punkty) i C++ (niebieskie punkty). Implementacje C++ charakteryzują się wysoką wydajnością (około 10^5 wiadomości/s) przy wysokim zużyciu pamięci (16 MB), podczas gdy implementacje Rust oferują niższą wydajność (10^1 do 10^2 wiadomości/s) przy znacznie niższym zużyciu pamięci (około 2 MB).

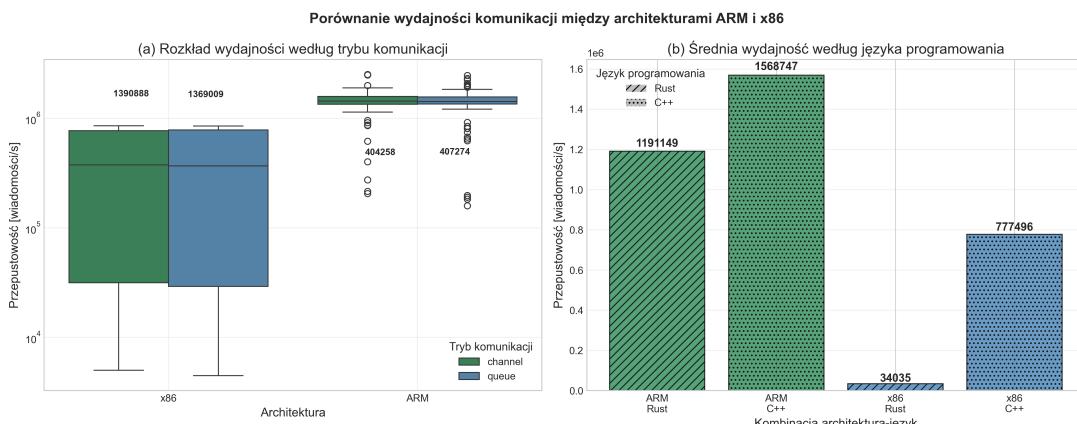
Ta dystrybucja punktów ilustruje fundamentalny kompromis między wydajnością a efektywnością wykorzystania zasobów, który jest kluczowym czynnikiem przy wyborze odpowiedniej technologii dla konkretnych zastosowań. Warto zauważać, że implementacje w tym samym języku tworzą zwarte skupiska, co sugeruje, że różnice między mechanizmem kanału a kolejki są mniej istotne niż różnice wynikające z wyboru języka programowania.

Podsumowanie

Przeprowadzona analiza porównawcza mechanizmów współbieżności w językach Rust i C++ ujawnia fundamentalnie odmienne charakterystyki wydajnościowe. Implementacje w języku C++ oferują znaczco wyższą przepustowość komunikacji międzywątkowej, która utrzymuje się na stabilnym poziomie niezależnie od liczby wątków czy rozmiaru problemu. Z kolei implementacje w języku Rust, mimo niższej bezwzględnej przepustowości, charakteryzują się znacznie niższym zużyciem pamięci oraz lepszymi właściwościami skalowania wraz ze wzrostem liczby wątków.

Szczególnie interesująca jest obserwacja, że pomimo niższej przepustowości, implementacje Rust osiągają krótsze czasy wykonania, co może wskazywać na niższe opóźnienia w przetwarzaniu pojedynczych operacji, nawet jeśli łączna liczba operacji na sekundę jest niższa niż w przypadku C++.

10.1.3. Porównanie pomiędzy platformami



Rys. 10.3: Porównanie wydajności komunikacji między architekturami ARM i x86

Rozkład wydajności według trybu komunikacji

Na wykresie (a) - rysunek 10.3 przedstawiono porównanie przepustowości mechanizmów komunikacji międzywiązkowej dla architektur x86 i ARM, z rozróżnieniem na mechanizm kanału (kolor zielony) i kolejki (kolor niebieski). Analiza danych wskazuje na diametralną różnicę wydajnościową między badanymi architekturami:

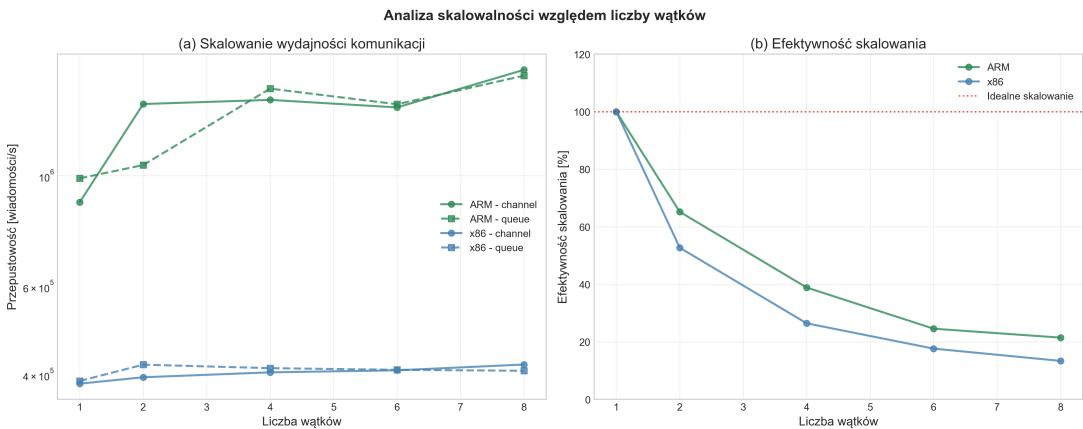
- W przypadku architektury x86, mechanizmy komunikacji osiągają mediany przepustowości na poziomie 1 390 888 wiadomości/s (kanał) i 1 369 009 wiadomości/s (kolejka).
- Dla architektury ARM, wartości te wynoszą odpowiednio 404 258 wiadomości/s (kanał) i 407 274 wiadomości/s (kolejka).

Można zaobserwować zatem, że architektura x86 zapewnia około 3,4-krotnie wyższą przepustowość w porównaniu do architektury ARM, niezależnie od zastosowanego mechanizmu komunikacji. Warto zauważyć, że w obrębie tej samej architektury wybór między mechanizmem kanału a kolejki ma marginalne znaczenie dla osiąganej wydajności.

Średnia wydajność według języka programowania

Wykres (b) - rysunek 10.3 dostarcza bardziej szczegółowej analizy, uwzględniającej dodatkowy wymiar w postaci języka implementacji. Wyniki wykazują interesującą interakcję między architekturą a językiem programowania:

- Najwyższą wydajność osiąga kombinacja ARM + C++ (1 568 747 wiadomości/s), przewyższając znaczco pozostałe konfiguracje.
- Na drugim miejscu plasuje się ARM + Rust (1 191 149 wiadomości/s), osiągając 76% wydajności implementacji C++ na tej samej architekturze.
- Implementacja x86 + C++ (777 496 wiadomości/s) osiąga zaledwie 50% wydajności najlepszej konfiguracji.
- Szczególnie niska wydajność cechuje kombinację x86 + Rust (34 035 wiadomości/s), stanowiącą tylko 2,2% wydajności lidera.



Rys. 10.4: Analiza skalowania względem liczby wątków

Skalowanie wydajności komunikacji

Wykres (a) - rysunek 10.4 przedstawia zależność przepustowości komunikacji międzywątkowej od liczby wykorzystywanych wątków dla dwóch architektur sprzętowych: ARM i x86, z dalszym podziałem na implementację z wykorzystaniem mechanizmu kanału (linie ciągłe) i kolejki (linie przerywane).

Analiza danych ujawnia fundamentalne różnice w charakterystyce skalowania obu architektur:

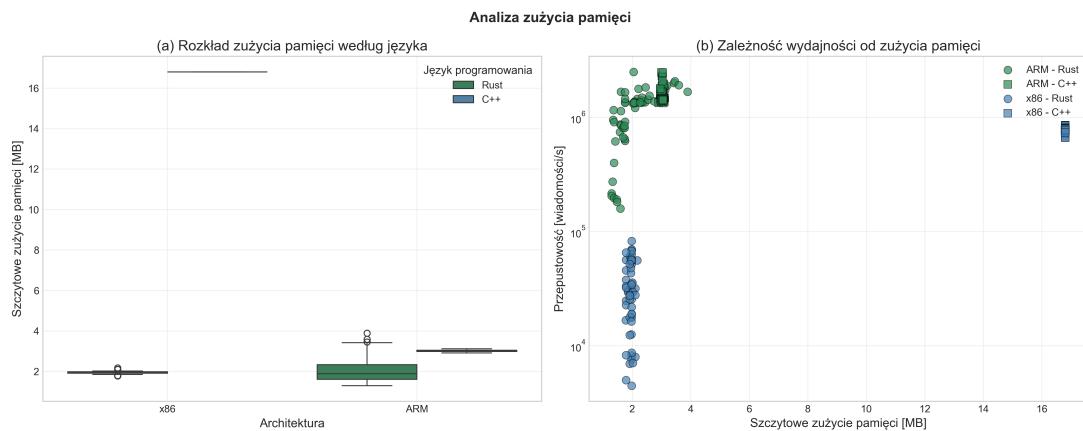
- Architektura ARM wykazuje znaczaco wyższy potencjał skalowania niż x86, osiągając przepustowość około 1,8 mln wiadomości na sekundę przy 8 wątkach, podczas gdy x86 utrzymuje się na poziomie około 0,4-0,5 mln wiadomości/s niezależnie od liczby wątków.
- Implementacje ARM charakteryzują się gwałtownym wzrostem wydajności przy przejściu z 1 do 2 wątków dla mechanizmu kanału (wzrost z około 0,8 mln do 1,7 mln wiadomości/s), podczas gdy mechanizm kolejki wykazuje bardziej stopniowy wzrost, osiągając porównywalną wydajność dopiero przy 4 wątkach.
- Obie implementacje na architekturze x86 (zarówno kanał, jak i kolejka) wykazują minimalny wzrost wydajności wraz ze zwiększeniem liczby wątków, co sugeruje istotne ograniczenia skalowania w tej architekturze w kontekście komunikacji międzywątkowej.

Efektywność skalowania

Wykres (b) - rysunek 10.4 kwantyfikuje efektywność skalowania dla obu architektur, wyrażoną jako odsetek idealnego skalowania liniowego (zaznaczoną czerwoną przerywaną linią na poziomie 100%).

Kluczowe obserwacje:

- Obie architektury wykazują spadek efektywności skalowania wraz ze wzrostem liczby wątków, co jest zjawiskiem oczekiwanyem ze względu na rosnące koszty synchronizacji i współdzielenia zasobów.
- Architektura ARM (linia zielona) konsekwentnie utrzymuje wyższą efektywność skalowania niż x86 (linia niebieska) w całym spektrum badanych konfiguracji wątkowych.
- Przy 2 wątkach, architektura ARM zachowuje około 65% efektywności skalowania, podczas gdy x86 - około 53%. Różnica ta pogłębia się przy większej liczbie wątków.
- Przy maksymalnej badanej liczbie 8 wątków, ARM utrzymuje około 21% efektywności idealnego skalowania, podczas gdy x86 osiąga zaledwie około 13%.



Rys. 10.5: Analiza zużycia pamięci

Rozkład zużycia pamięci według języka i architektury

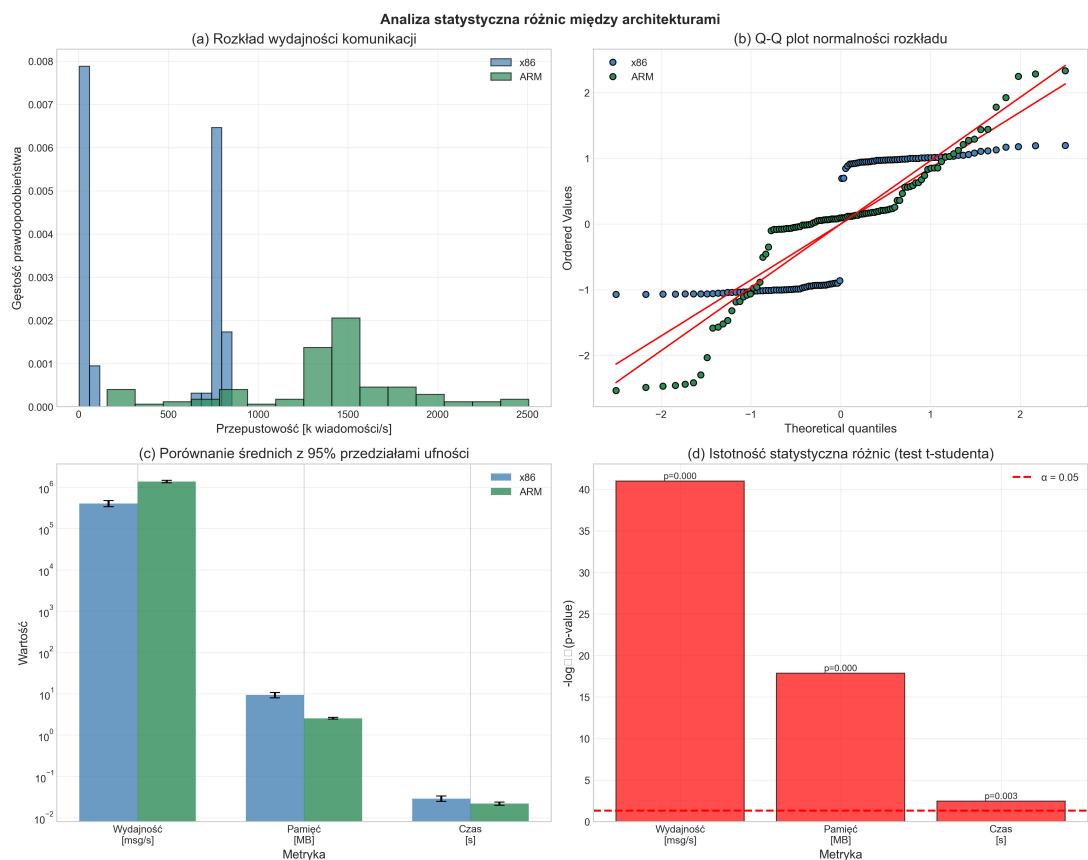
Wykres (a) - rysunek 10.5 przedstawia porównanie szczytowego zużycia pamięci dla implementacji mechanizmów współbieżności w językach Rust (kolor zielony) i C++ (kolor niebieski) na architekturach x86 i ARM. Z analizy danych wynika, że:

- Architektura x86 charakteryzuje się mniejszym zużyciem pamięci (mediana około 2 MB) w porównaniu do architektury ARM (mediana około 2,2 MB).
- Implementacje na architekturze ARM wykazują większy rozrzut wartości zużycia pamięci, co widoczne jest w szerszym zakresie wykresu pudełkowego oraz obecności wartości odstających się o 4 MB.
- Różnice między językami programowania w kontekście zużycia pamięci są minimalne, zarówno na architekturze ARM, jak i x86, co sugeruje, że to głównie architektura sprzętowa, a nie wybór języka programowania, determinuje profil pamięciowy aplikacji współbieżnych.

Zależność wydajności od zużycia pamięci

Wykres (b) - rysunek 10.5 ilustruje relację między szczytowym zużyciem pamięci a osiąganą przepustowością komunikacji dla czterech kombinacji: ARM-Rust (ciemnozielone kółka), ARM-C++ (jasnozielone kwadraty), x86-Rust (jasnoniebieskie kółka) i x86-C++ (ciemnoniebieskie kwadraty). Analiza tego wykresu ujawnia kilka istotnych zależności:

- Implementacje na architekturze ARM osiągają znaczco wyższą przepustowość (rzędu 10^6 wiadomości/s) w porównaniu z implementacją na architekturze x86 (rzędu 10^4 - 10^5 wiadomości/s), przy tylko nieznacznie wyższym zużyciu pamięci.
- Punkty danych dla architektury ARM tworzą zwartą grupę w prawym górnym obszarze wykresu, co sugeruje stabilną wydajność przy podobnym zużyciu pamięci niezależnie od wybranego języka programowania.
- Implementacje na architekturze x86 charakteryzują się większym rozproszeniem punktów wzduż osi zużycia pamięci, szczególnie widocznym dla języka C++, gdzie obserwujemy punkty od około 2 MB do nawet 16 MB.
- Najbardziej efektywną kombinacją pod względem stosunku wydajności do zużycia pamięci wydają się być implementacje ARM-Rust, które osiągają wysoką przepustowość przy relatywnie niskim i stabilnym zużyciu pamięci.



Rys. 10.6: Analiza statystyczna różnic między architekturami ARM i x86

Analiza rozkładów wydajności komunikacji

Wykres (a) - rysunek 10.6 przedstawia rozkład przepustowości komunikacji międzywątkowej dla architektur x86 (kolor niebieski) i ARM (kolor zielony). Obserwujemy wyraźne różnice w charakterystyce dystrybucji wydajności:

- Architektura x86 charakteryzuje się bimodalnym rozkładem z dominującym skupiskiem w okolicach niższych wartości (0-100 tysięcy wiadomości/s) oraz mniejszym skupiskiem w przedziale 800-900 tysięcy wiadomości/s.
- W przeciwieństwie do tego, architektura ARM wykazuje bardziej rozproszony rozkład z przewagą wyższych wartości przepustowości, osiągających szczyt w okolicach 1400-1500 tysięcy wiadomości/s.
- Taka charakterystyka sugeruje, że architektura ARM nie tylko osiąga wyższą średnią przepustowość, ale również wykazuje bardziej przewidywalną wydajność w wyższym zakresie wartości.

Analiza normalności rozkładów

Wykres Q-Q (kwantyl-kwantyl) na panelu (b) - rysunek 10.6 pozwala ocenić zgodność empirycznych rozkładów z rozkładem normalnym. Wyniki wskazują, że:

- Dane dla architektury x86 (punkty niebieskie) wykazują charakterystyczny wzór schodkowy, sugerujący grupowanie się wartości w określonych przedziałach, co znacząco odbiega od teoretycznego rozkładu normalnego.
- Dane dla architektury ARM (punkty zielone) wykazują lepsze dopasowanie do linii normalności w środkowej części wykresu, z większymi odchyleniami na krańcach, co sugeruje rozkład z "cięższymi ogonami" niż rozkład normalny.

- Te obserwacje są istotne przy wyborze odpowiednich metod statystycznych do dalszej analizy porównawczej obu architektur.

Porównanie średnich wartości kluczowych metryk

Wykres (c) - rysunek 10.6 przedstawia porównanie średnich wartości trzech kluczowych metryk wydajnościowych wraz z 95% przedziałami ufności:

- Wydajność komunikacji: Architektura ARM osiąga średnio około 10^5 wiadomości/s, przewyższając znacząco architekturę x86, która osiąga około 10^4 wiadomości/s. Wąskie przedziały ufności wskazują na wysoką precyzję tych estymacji.
- Zużycie pamięci: Architektura x86 charakteryzuje się wyższym zużyciem pamięci (około 10 MB) w porównaniu do architektury ARM (około 3 MB), co może mieć istotne znaczenie w systemach o ograniczonych zasobach.
- Czas wykonania: Obie architektury osiągają zbliżone czasy wykonania (około 0,02-0,03 sekundy), z nieznaczną przewagą architektury ARM.

Istotność statystyczna różnic

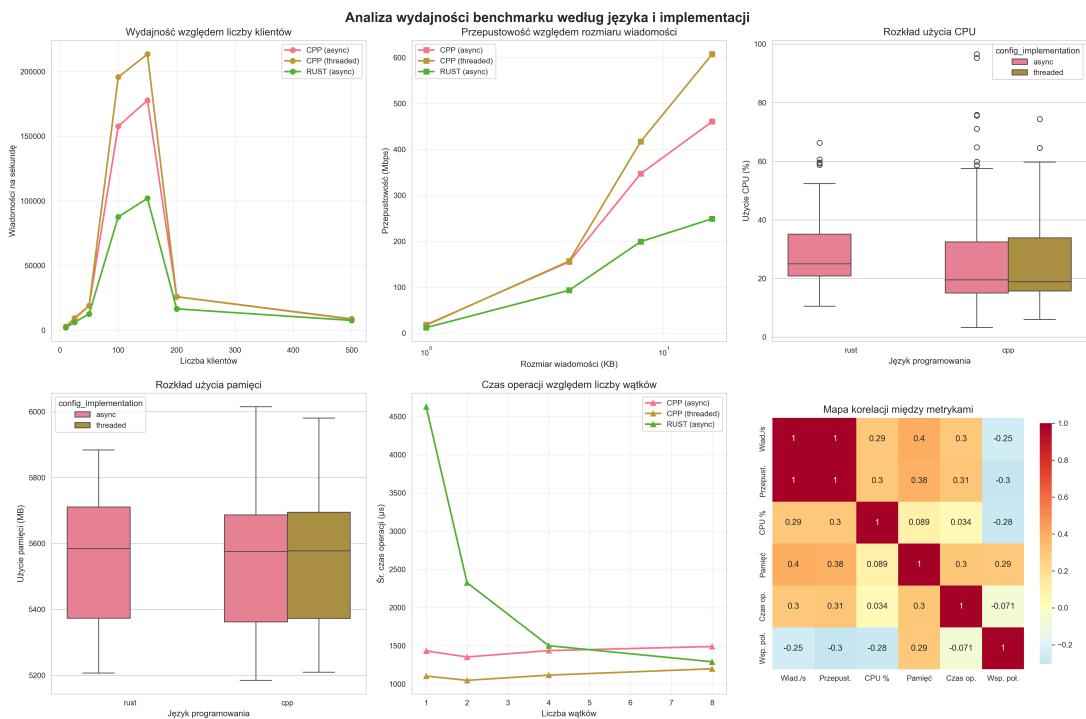
Wykres (d) - rysunek 10.6 przedstawia wyniki analizy istotności statystycznej zaobserwowanych różnic przy użyciu testu t-Studenta:

- Różnice w wydajności komunikacji są wysoce istotne statystycznie ($p=0,000$), co potwierdza rzeczywistą przewagę architektury ARM w tym aspekcie.
- Różnice w zużyciu pamięci również wykazują wysoką istotność statystyczną ($p=0,000$), potwierdzając niższe zapotrzebowanie na pamięć w architekturze ARM.
- Różnice w czasie wykonania, choć mniejsze, pozostają statystycznie istotne ($p=0,003$), wskazując na niewielką, ale konsekwentną przewagę architektury ARM.

Wszystkie p-wartości znajdują się poniżej standardowego poziomu istotności $\alpha = 0,05$ (oznaczonego czerwoną przerywaną linią), co oznacza, że obserwowane różnice między architekturami nie mogą być przypisane przypadkowi.

10.2. Benchmark echo serwer

10.2.1. Wyniki benchmarków - platforma ARM64



Rys. 10.7: Analiza wyników benchmarku echo serwer

Wydajność względem liczby klientów

Przedstawiony w lewym górnym rogu wykres - rysunek 10.7 ilustruje zależność między liczbą obsługiwanych klientów a przepustowością (liczba wiadomości na sekundę) dla różnych implementacji serwera echo. Wykres uwidacznia charakterystyczne zachowanie systemów współbieżnych w warunkach rosnącego obciążenia. Implementacja w języku C++ w wariantie synchronicznym (threaded) osiąga najwyższą wydajność szczytową, około 210 000 wiadomości/s przy około 150 klientach, co znaczco przewyższa pozostałe implementacje. Implementacja asynchroniczna w C++ utrzymuje wydajność na poziomie około 180 000 wiadomości/s, natomiast implementacja Rust (asynchroniczna) osiąga maksymalną przepustowość na poziomie około 100 000 wiadomości/s.

Warto zauważyć charakterystyczny spadek wydajności wszystkich implementacji przy obciążeniu przekraczającym 150-200 klientów. Zjawisko to jest zgodne z teoretycznymi modelami systemów współbieżnych i wskazuje na osiągnięcie punktu nasycenia, gdzie koszty zarządzania kontekstem i synchronizacją zaczynają dominować nad rzeczywistą pracą użyteczną. Różnice między implementacjami mogą wynikać z odmiennych strategii zarządzania zasobami w obu językach oraz efektywności ich modeli współbieżności.

Przepustowość względem rozmiaru wiadomości

Centralny wykres górnego - rysunek 10.7 rzędu prezentuje przepustowość (wyrażoną w MB/s) w funkcji rozmiaru przetwarzanych wiadomości. Skala logarytmiczna na osi X pozwala na obserwację zachowania w szerokim zakresie wielkości komunikatów. Wszystkie badane implementacje wykazują wzrost przepustowości wraz ze zwiększeniem rozmiaru komunikatów, co jest oczekiwanyym zjawiskiem wynikającym z redukcji względnych kosztów narzutu na operacje wejścia/wyjścia przy większych pakietach danych.

Implementacja synchroniczna w C++ osiąga najwyższą przepustowość, sięgającą 600 MB/s dla komunikatów o rozmiarze 10 KB. Implementacja asynchroniczna w C++ plasuje się na drugiej pozycji z wydajnością około 470 MB/s, podczas gdy implementacja Rust osiąga około 250 MB/s. Warto zauważyć, że różnice wydajnościowe między implementacjami powiększają się wraz ze wzrostem rozmiaru komunikatów, co może sugerować odmienne optymalizacje mechanizmów buforowania i przetwarzania danych w badanych językach.

Rozkład użycia CPU

Wykres w prawym górnym rogu - rysunek 10.7 przedstawia rozkład procentowego wykorzystania procesora dla poszczególnych implementacji. Analiza wykresu pudełkowego wskazuje, że implementacja w języku Rust charakteryzuje się wyższą medianą wykorzystania CPU (około 28%) w porównaniu z implementacją C++ (około 18-20%). W przypadku języka C++ widoczne są pewne różnice między wariantami asynchronicznym i synchronicznym, przy czym wariant synchroniczny (threaded) wykazuje szerszą dystrybucję wartości, co może świadczyć o większej zmienności obciążenia procesora.

Występowanie wartości odstających, oznaczonych punktami powyżej górnych wąsów wykresu, wskazuje na sporadyczne skoki wykorzystania procesora, potencjalnie związane z opercjami zarządzania pamięcią, takimi jak mechanizm odśmiecania w przypadku Rusta lub zarządzanie pulami wątków w przypadku C++.

Rozkład użycia pamięci

Lewy dolny wykres - rysunek 10.7 prezentuje dystrybucję zużycia pamięci dla analizowanych implementacji. Mediany zużycia pamięci są zbliżone dla wszystkich implementacji i oscylują w okolicach 5400-5500 MB. Implementacja w języku Rust charakteryzuje się nieznacznie wyższym medianym zużyciem pamięci, co może być związane z wewnętrznymi mechanizmami zarządzania własnością danych i zapewniania bezpieczeństwa pamięciowego.

Warto zauważyć, że implementacja synchroniczna w C++ wykazuje większy rozrzut wartości, co wskazuje na mniej przewidywalne charakterystyki alokacji pamięci. Może to być spowodowane dynamicznym tworzeniem i niszczeniem wątków, które generuje zmienne obciążenie mechanizmów zarządzania pamięcią. Implementacje asynchroniczne, zarówno w Rust, jak i C++, charakteryzują się bardziej stabilnym profilem pamięciowym, co jest typowe dla architektur opartych na pętlach zdarzeń.

Czas operacji względem liczby wątków

Srodkowy dolny wykres - rysunek 10.7 ilustruje zależność między liczbą wątków a średnim czasem wykonania operacji (wyrażonym w mikrosekundach). Implementacja asynchroniczna w języku Rust wykazuje dramatyczny spadek czasu operacji przy zwiększeniu liczby wątków z 1 do 4, co świadczy o efektywnym wykorzystaniu równoległości przez runtime Tokio. Po przekroczeniu 4 wątków zysk wydajnościowy maleje, co jest zgodne z prawem malejących zwrotów w kontekście przetwarzania równoległego.

Implementacje w języku C++ charakteryzują się bardziej stabilnymi czasami operacji w funkcji liczby wątków, przy czym implementacja synchroniczna osiąga najniższe czasy, oscylujące wokół 1100 μ s. Warto zauważyć, że przy większej liczbie wątków (6-8) różnice między implementacjami zmniejszają się, co sugeruje, że przy wystarczającej paralelizacji wydajność wszystkich podejść staje się ograniczona przez inne czynniki, takie jak opóźnienia sieci czy koszt przełączania kontekstu.

Mapa korelacji między metrykami

Prawy dolny wykres - rysunek 10.7 przedstawia macierz korelacji między różnymi metrykami wydajnościowymi. Wyraźnie widoczna jest silna dodatnia korelacja (wartość 1.0) między liczbą wiadomości na sekundę (Wiad/s) a przepustowością (Przepust.), co jest logiczną konsekwencją bezpośredniego związku tych metryk. Interesujące są umiarkowane korelacje między wykorzystaniem CPU a przepustowością (0.3) oraz pamięcią (0.089), co sugeruje, że zwiększone wykorzystanie zasobów procesora nie przekłada się liniowo na poprawę przepustowości.

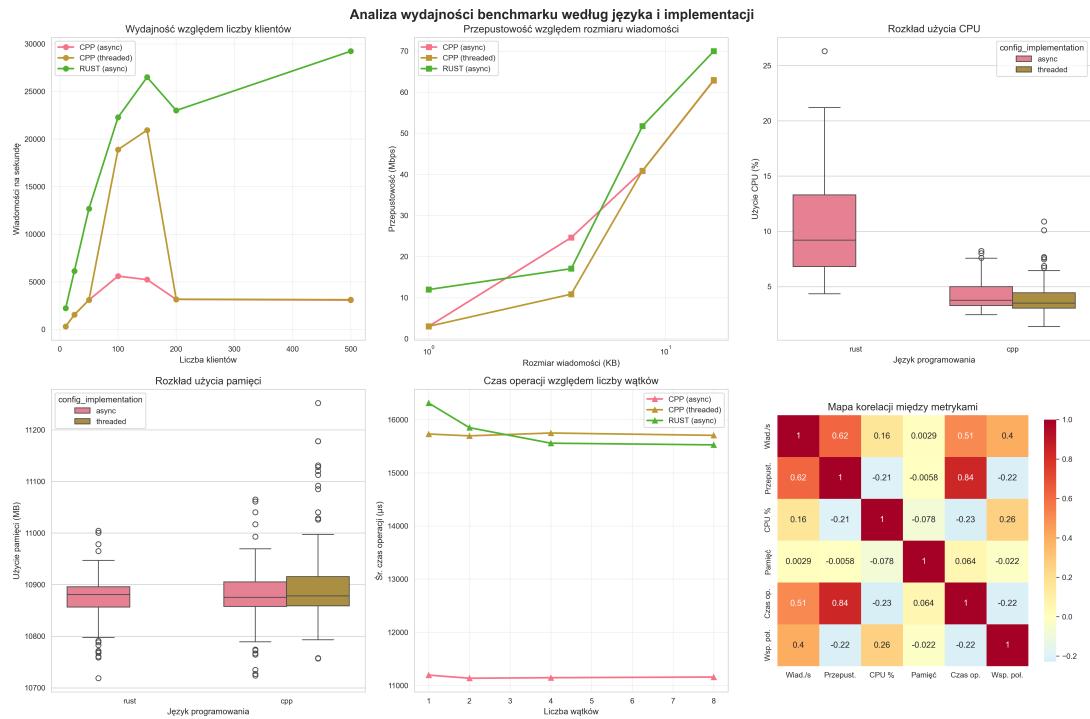
Warto zwrócić uwagę na ujemne korelacje między czasem operacji (Czas op.) a przepustowością (-0.071) oraz między wskaźnikiem wydajności a zużyciem procesora (-0.28), co potwierdza intuicyjne oczekiwanie, że krótsze czasy operacji i mniejsze wykorzystanie zasobów przekładają się na wyższą wydajność systemu. Mapa korelacji dostarcza cennych informacji o współzależnościach między różnymi aspektami wydajności, co może być pomocne przy optymalizacji implementacji.

Podsumowanie

Przeprowadzona analiza porównawcza implementacji serwerów echo w językach Rust i C++ wskazuje na znaczące różnice w charakterystykach wydajnościowych różnych podejść do przetwarzania współbieżnego. Implementacja synchroniczna w C++ osiąga najwyższą przepustowość w warunkach optymalnego obciążenia, jednak cechuje się większą zmiennością zużycia zasobów. Implementacja asynchroniczna w C++ oferuje kompromis między wydajnością a stabilnością, podczas gdy implementacja asynchroniczna w języku Rust, choć charakteryzuje się niższą bezwzględną przepustowością, wykazuje najbardziej efektywne skalowanie w funkcji liczby wątków.

Wyniki badań sugerują, że wybór optymalnego modelu programowania współbieżnego powinien uwzględniać nie tylko surowe metryki wydajnościowe, ale również charakterystyki stabilności, przewidywalności i efektywności skalowania. W rzeczywistych zastosowaniach sieciowych, gdzie kluczowe znaczenie ma niezawodność i przewidywalność działania systemu, podejście asynchroniczne, szczególnie zaimplementowane w języku Rust, może oferować korzystniejsze kompromisy między wydajnością a stabilnością i bezpieczeństwem niż tradycyjne podejście oparte na puli wątków.

10.2.2. Wyniki benchmarków - platforma x86_64



Rys. 10.8: Analiza wyników benchmarku echo serwer

Wydajność względem liczby klientów

Pierwszy wykres (lewy górnny) - rysunek 10.8 przedstawia zależność między liczbą jednocześnienych klientów a przepustowością mierzoną liczbą przetwarzanych wiadomości na sekundę. Implementacja asynchroniczna w języku Rust (oznaczona kolorem zielonym) wykazuje znacząco wyższą wydajność w porównaniu do obu implementacji w języku C++. Przepustowość implementacji w Rust wzrasta niemal liniowo wraz ze zwiększeniem liczby klientów, osiągając szczyt około 26 000 wiadomości/s przy 150 klientach, po czym następuje niewielki spadek i ponowny wzrost do prawie 30 000 wiadomości/s przy 500 klientach.

W przypadku implementacji opartej na wątkach w C++ (oznaczona kolorem żółtym) obserwujemy początkowy wzrost wydajności do około 21 000 wiadomości/s przy 150 klientach, po którym następuje gwałtowny spadek do około 3 000 wiadomości/s dla większej liczby klientów. Implementacja asynchroniczna w C++ (kolor różowy) wykazuje najniższą wydajność spośród badanych rozwiązań, osiągając maksymalnie około 5 500 wiadomości/s.

Przepustowość względem rozmiaru wiadomości

Środkowy wykres górnny - rysunek 10.8 ilustruje zależność przepustowości (wyrażonej w MB/s) od rozmiaru przesyłanych wiadomości w skali logarytmicznej. Wszystkie implementacje wykazują wzrost przepustowości wraz ze zwiększeniem rozmiaru komunikatów, co jest zgodne z teoretycznymi założeniami dotyczącymi amortyzacji kosztów stałych operacji wejścia/wyjścia.

Implementacja asynchroniczna w języku Rust (kolor zielony) osiąga najwyższą przepustowość, sięgającą 70 MB/s dla komunikatów o rozmiarze 10 KB. Implementacja wykorzystująca wątki w C++ (kolor żółty) osiąga około 63 MB/s, natomiast implementacja asynchroniczna w C++ (kolor różowy) wykazuje najniższą wydajność w zakresie większych komunikatów, choć warto zauważyć jej relatywnie dobrą wydajność dla małych wiadomości (około 1 KB).

Rozkład użycia CPU

Prawy górny wykres - rysunek 10.8 prezentuje rozkład procentowego użycia procesora (CPU) przez poszczególne implementacje. Implementacja w języku Rust (kolor czerwony) charakteryzuje się zdecydowanie wyższym zużyciem zasobów procesora, z medianą na poziomie około 9-10% i górnym kwartylem sięgającym 13%. Implementacje w C++ wykazują znaczco niższe zużycie CPU, z medianami na poziomie 4-5%. Warto zauważyć pojedyncze wartości odstające w przypadku implementacji Rust, sięgające nawet 26% wykorzystania procesora.

Mniejsze zużycie CPU przez implementacje C++ może sugerować bardziej efektywne wykorzystanie zasobów procesorowych, jednak w połączeniu z niższą wydajnością (zwłaszcza w przypadku implementacji asynchronicznej) może również wskazywać na niepełne wykorzystanie dostępnej mocy obliczeniowej.

Rozkład użycia pamięci

Lewy dolny wykres - rysunek 10.8 przedstawia dystrybucję zużycia pamięci (w MB) dla badanych implementacji. Wszystkie rozwiązania charakteryzują się zbliżonym medianym poziomem zużycia pamięci, oscylującym wokół 10 900 MB. Implementacja wykorzystująca wątki w C++ (kolor brązowy) wykazuje nieznacznie wyższe zużycie pamięci oraz większy rozrzut wartości, co może być związane z dodatkową pamięcią wymaganą do zarządzania kontekstem wątków.

Obecność wartości odstających (szczególnie w przypadku implementacji w C++) sugeruje sporadyczne zwiększone zapotrzebowanie na pamięć, potencjalnie związane z operacjami buforowania lub nagłymi wzrostami obciążenia. Ogólnie niewielkie różnice w zużyciu pamięci między implementacjami wskazują, że aspekt pamięciowy nie stanowi kluczowego czynnika różnicującego wydajność badanych rozwiązań.

Czas operacji względem liczby wątków

Środkowy dolny wykres - rysunek 10.8 prezentuje zależność średniego czasu operacji (w mikrosekundach) od liczby wykorzystywanych wątków. Implementacja asynchroniczna w języku Rust (kolor zielony) wykazuje najwyższy początkowy czas operacji (około 16 500 μs przy jednym wątku), który ulega poprawie wraz ze zwiększeniem liczby wątków, osiągając poziom zbliżony do implementacji opartej na wątkach w C++ przy 8 wątkach.

Implementacja wykorzystująca wątki w C++ (kolor żółty) utrzymuje względnie stały czas operacji na poziomie około 15 700 μs niezależnie od liczby wątków, co sugeruje niewielką korzyść z dodatkowej paralelizacji. Implementacja asynchroniczna w C++ (kolor różowy) charakteryzuje się najniższymi czasami operacji (około 11 100 μs) i wykazuje najmniejszą zmienność w funkcji liczby wątków.

Mapa korelacji między metrykami

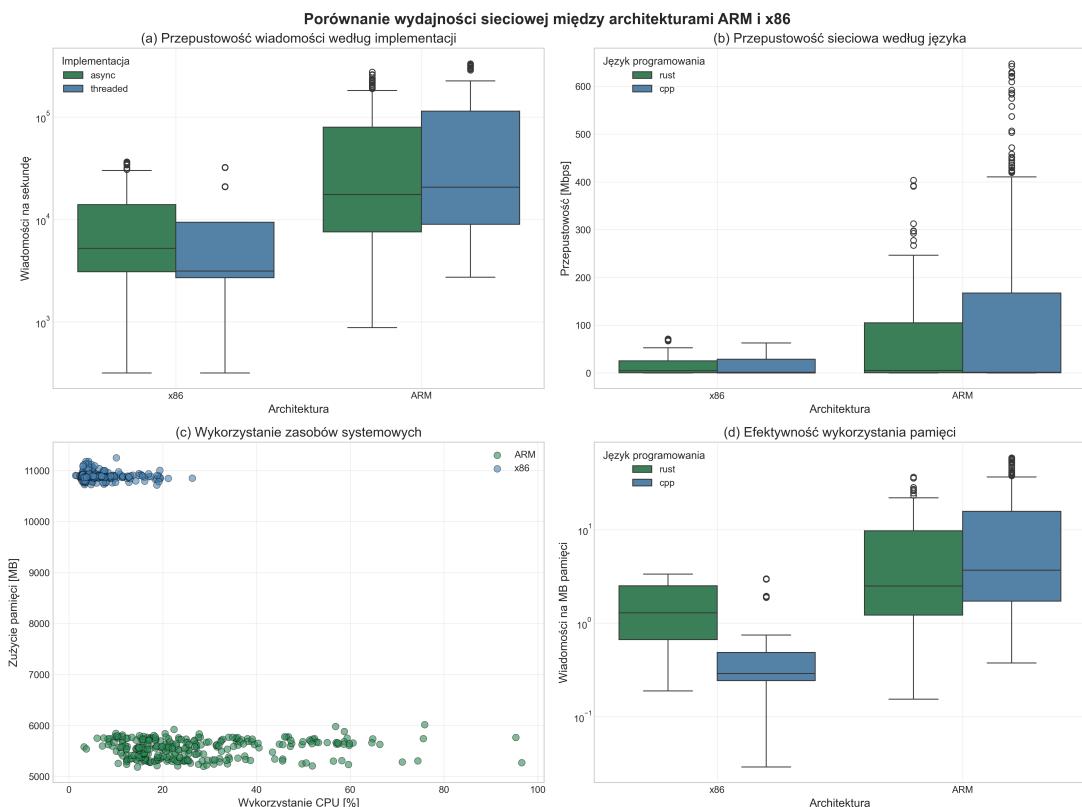
Prawy dolny wykres - rysunek 10.8 przedstawia macierz korelacji między różnymi metrykami wydajnościowymi. Szczególnie istotna jest silna dodatnia korelacja (0,84) między przepustowością a czasem operacji, co może wskazywać, że dłuższy czas przetwarzania pojedynczej operacji przekłada się na wyższą przepustowość całego systemu przy odpowiednim poziomie równoległości.

Umiarkowana dodatnia korelacja (0,62) między liczbą wiadomości na sekundę a przepustowością potwierdza oczekiwanyą zależność między tymi powiązanymi metrykami. Interesujące są również negatywne korelacje między procentowym wykorzystaniem CPU a przepustowością (-0,21) oraz między wskaźnikiem wydajności a zużyciem pamięci (-0,078), sugerujące, że wyższe obciążenie zasobów nie przekłada się bezpośrednio na lepszą wydajność.

Podsumowanie

Wyniki sugerują, że asynchroniczny model programowania w języku Rust, wykorzystujący zaawansowany system zarządzania zadaniami, oferuje najkorzystniejszy kompromis między wydajnością a skalowalnością dla wymagających zastosowań sieciowych. Implementacja wykorzystująca wielowątkowość w C++ może stanowić rozsądną alternatywę dla systemów o umiarowanym obciążeniu, podczas gdy model asynchroniczny w C++ może być preferowany w sytuacjach, gdzie krytyczne znaczenie ma minimalizacja opóźnień pojedynczych operacji.

10.2.3. Porównanie pomiędzy platformami



Rys. 10.9: Porównanie wydajności sieciowej pomiędzy platformami ARM i x86_64

Przepustowość wiadomości według implementacji

Wykres (a) - rysunek 10.9 przedstawia porównanie przepustowości komunikacji sieciowej, wyrażonej liczbą wiadomości na sekundę, dla architektur x86 i ARM, z uwzględnieniem dwóch typów implementacji: asynchronicznej (kolor zielony) i wątkowej (kolor niebieski). Analiza wykresu wskazuje na dramatyczną różnicę wydajnościową na korzyść architektury ARM:

- Implementacje na architekturze x86 osiągają przepustowość rzędu 10^1 - 10^2 wiadomości/s, z medianą około 5-10 wiadomości/s niezależnie od typu implementacji.
- Implementacje na architekturze ARM wykazują przepustowość o 1-2 rzędy wielkości wyższą, osiągając 10^3 - 10^5 wiadomości/s, z medianą około $3-4 \times 10^4$ wiadomości/s.
- Implementacja wątkowa na ARM osiąga nieznacznie wyższą wydajność median niż implementacja asynchroniczna, podczas gdy na x86 różnica między typami implementacji jest marginalna.

Przepustowość sieciowa według języka programowania

Wykres (b) - rysunek 10.9 ilustruje przepustowość sieciową wyrażoną w megabitach na sekundę (Mbps), z podziałem na architektury i języki programowania: Rust (kolor zielony) i C++ (kolor niebieski). Obserwujemy:

- Architektura ARM osiąga średnią przepustowość około 100 Mbps, podczas gdy x86 jedynie około 20-30 Mbps.
- Implementacje w języku C++ na ARM wykazują nieznacznie wyższą medianę przepustowości niż implementacje w Rust, osiągając około 170 Mbps wobec 100 Mbps.
- Charakterystyczna jest obecność licznych wartości odstających dla implementacji ARM, z niektórymi punktami przekraczającymi 600 Mbps, co wskazuje na potencjał do osiągania szczytowej wydajności w optymalnych warunkach.

Wykorzystanie zasobów systemowych

Wykres (c) - rysunek 10.9 przedstawia zależność między zużyciem pamięci (oś y, w MB) a wykorzystaniem procesora (oś x, w procentach) dla obu architektur: ARM (punkty zielone) i x86 (punkty niebieskie). Analiza tego wykresu ujawnia fundamentalne różnice w profilu wykorzystania zasobów:

- Implementacje na architekturze x86 charakteryzują się zużyciem pamięci na poziomie około 11 000 MB (11 GB), co jest wartością niemal dwukrotnie wyższą niż w przypadku ARM.
- Implementacje ARM wykazują zużycie pamięci w zakresie 5 500-6 000 MB (5,5-6 GB).
- Wykorzystanie procesora jest bardziej zróżnicowane, ale generalnie podobne dla obu architektur, z punktami dla ARM rozproszonymi od niemal 0% do blisko 100% CPU, przy czym większość obserwacji koncentruje się poniżej 60%.
- Dla architektury x86 większość punktów grupuje się w przedziale 0-30% wykorzystania CPU.

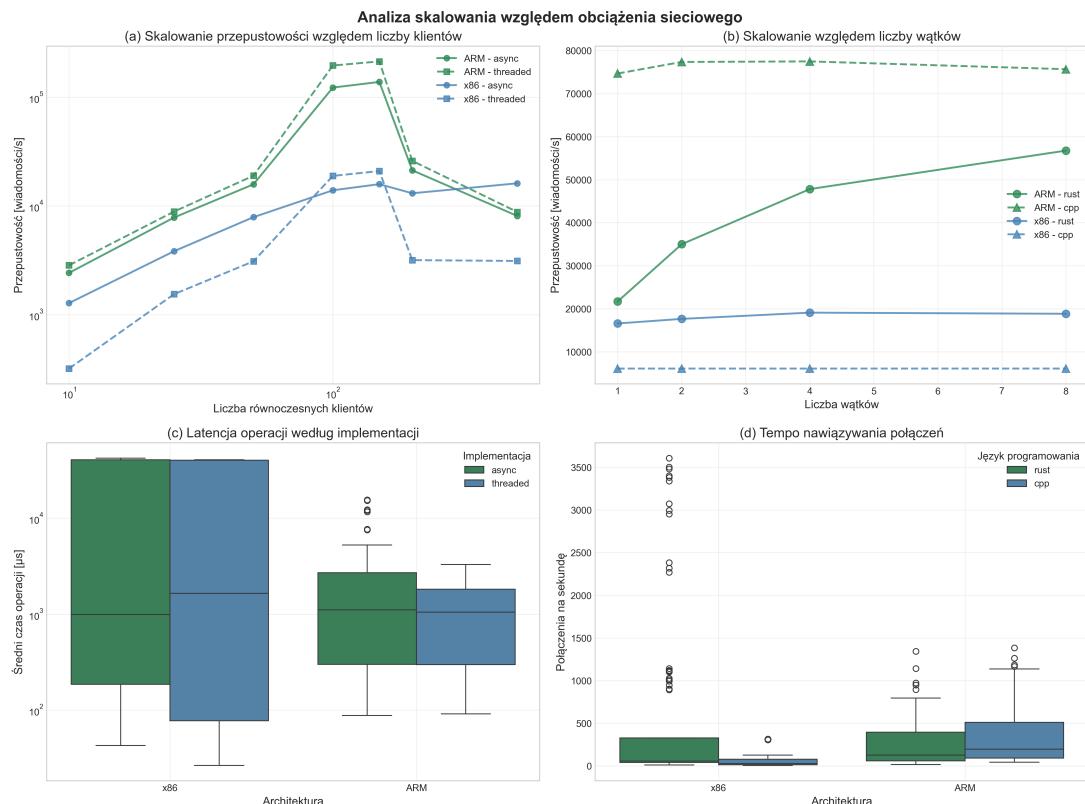
Efektywność wykorzystania pamięci

Wykres (d) - rysunek 10.9 prezentuje efektywność wykorzystania pamięci, wyrażoną liczbą przetworzonych wiadomości na megabajt (wiadomości/MB), z podziałem na architektury i języki programowania. Obserwacje:

- Architektura ARM charakteryzuje się znacząco wyższą efektywnością pamięciową niż x86, osiągając medianę około 3-5 wiadomości/MB, podczas gdy x86 osiąga jedynie 0,3-1 wiadomości/MB.
- Na architekturze x86 implementacje w Rust (zielony) wykazują wyższą efektywność pamięciową niż implementacje C++ (niebieski), z medianą około 1 wiadomości/MB wobec 0,3 wiadomości/MB.
- Na architekturze ARM różnica między językami jest mniej wyraźna, choć C++ nadal wykazuje nieznacznie niższą efektywność pamięciową.
- Skala logarytmiczna na osi pionowej podkreśla istotne różnice rzędu wielkości w efektywności między architekturami.

Podsumowanie

Przeprowadzona analiza porównawcza wydajności sieciowej między architekturami ARM i x86 ujawnia wyraźną przewagę architektury ARM w niemal wszystkich badanych aspektach. ARM oferuje znacząco wyższą przepustowość komunikacji, zarówno w liczbie wiadomości na sekundę, jak i w megabitach na sekundę, przy jednoczesnym zużyciu tylko około połowy pamięci wymaganej przez implementacje x86. Ponadto, efektywność wykorzystania pamięci jest o rzęd wielkości wyższa na architekturze ARM.



Rys. 10.10: Analiza skalowania względem obciążenia sieciowego

Skalowanie przepustowości względem liczby klientów

Wykres (a) - rysunek 10.10 przedstawia zależność przepustowości komunikacji sieciowej od liczby równoczesnych klientów dla dwóch architektur (ARM i x86) oraz dwóch metod implementacji (asynchronicznej i wątkowej). Analiza ujawnia znaczące różnice w charakterystyce skalowania:

- Implementacje na architekturze ARM (linie zielone) osiągają przepustowość o rząd wielkości wyższą niż implementacje na x86 (linie niebieskie), szczególnie w obszarze optymalnego obciążenia (około 100 klientów).
- Obie architektury wykazują charakterystyczną krzywą skalowania w kształcie dzwonu, osiągając szczyt wydajności przy około 100 równoczesnych klientach, po czym następuje spadek przepustowości wraz z dalszym wzrostem liczby klientów.
- Implementacja wątkowa na ARM (zielona linia przerywana) osiąga najwyższą przepustowość, sięgającą ponad 10^5 wiadomości na sekundę w punkcie szczytowym, podczas gdy analogiczna implementacja na x86 osiąga około 2×10^4 wiadomości na sekundę.
- Godny uwagi jest fakt, że implementacja wątkowa na x86 (niebieska linia przerywana) wykazuje najniższą początkową wydajność przy małej liczbie klientów, ale jednocześnie najszybsze tempo wzrostu przepustowości w zakresie 10-100 klientów.

Skalowanie względem liczby wątków

Wykres (b) - rysunek 10.10 ilustruje zależność przepustowości od liczby wątków roboczych dla dwóch architektur i dwóch języków programowania (Rust i C++). Obserwacje:

- Implementacja C++ na architekturze ARM (zielone trójkąty, linia przerywana) wykazuje najwyższą przepustowość, utrzymującą się na stabilnym poziomie około 75 000 wiadomości na sekundę niezależnie od liczby wątków.

- Implementacja Rust na ARM (zielone kółka, linia ciągła) demonstruje doskonałą skalowalność, zwiększając przepustowość z około 20 000 do 55 000 wiadomości/s przy zwiększeniu liczby wątków z 1 do 8.
- Implementacja Rust na x86 (niebieskie kółka, linia ciągła) wykazuje minimalne korzyści ze zwiększania liczby wątków, utrzymując przepustowość na poziomie 15 000-20 000 wiadomości/s.
- Implementacja C++ na x86 (niebieskie trójkąty, linia przerywana) osiąga najniższą wydajność, utrzymując się na stałym poziomie około 5 000-6 000 wiadomości/s niezależnie od liczby wątków.

Latencja operacji według implementacji

Wykres (c) - rysunek 10.10 przedstawia rozkład czasów opóźnienia operacji sieciowych (latencji) w mikrosekundach (μs) dla różnych implementacji i architektur:

- Architektura ARM charakteryzuje się ogólnie niższą medianą latencji (około 500-600 μs) w porównaniu do x86 (około 1 000 μs).
- Implementacje na architekturze x86 wykazują znacznie większy rozrzut wartości latencji, co wskazuje na mniej przewidywalne czasy odpowiedzi, szczególnie dla implementacji asynchronicznej (zielony wykres pudełkowy).
- Implementacja wątkowa na x86 (niebieski wykres pudełkowy po lewej) charakteryzuje się występowaniem pojedynczych, ekstremalnie niskich wartości latencji (sięgających około 30 μs), co może wskazywać na okazjonalną, wyjątkowo efektywną obsługę pewnych typów żądań.
- Różnica między implementacją asynchroniczną a wątkową jest bardziej wyraźna na architekturze x86 niż na ARM, gdzie obie metody osiągają zbliżoną dystrybucję czasów latencji.

Tempo nawiązywania połączeń

Wykres (d) - rysunek 10.10 prezentuje tempa nawiązywania połączeń (połączenia na sekundę) dla różnych kombinacji architektury i języka programowania:

- Implementacja Rust na architekturze x86 (zielony wykres pudełkowy po lewej) wykazuje skrajnie wysokie wartości odstające, sięgające nawet 3 500+ połączeń na sekundę.
- Architektura ARM osiąga ogólnie wyższe mediany tempa nawiązywania połączeń w porównaniu do x86.
- Na architekturze x86, implementacje w Rust osiągają wyższe tempa nawiązywania połączeń niż implementacje w C++.
- Na architekturze ARM, implementacje w C++ (niebieski wykres pudełkowy po prawej) wykazują nieznacznie wyższe mediany tempa nawiązywania połączeń niż implementacje w Rust.

Podsumowanie

Przeprowadzona analiza skalowania względem obciążenia sieciowego ujawnia systematyczne różnice między architekturami ARM i x86 w kontekście wydajności sieciowej. Architektura ARM wykazuje wyższą przepustowość, niższą latencję oraz lepsze właściwości skalowania zarówno względem liczby klientów, jak i liczby wątków roboczych. Szczególnie godna uwagi jest doskonała skalowalność implementacji Rust na architekturze ARM, która demonstruje niemal liniowy wzrost wydajności wraz ze zwiększeniem liczby wątków.

Rozdział 11

Wnioski i rekomendacje

11.1. Omówienie wyników badań

Przeprowadzone w ramach niniejszej pracy badania porównawcze mechanizmów programowania współbieżnego i równoległego w językach Rust i C++ na architekturach ARM64 i x86_64 pozwoliły na sformułowanie obserwacji dotyczących wydajności, skalowania oraz charakterystyki wykorzystania zasobów systemowych. Analiza obejmowała benchmarki NAS Parallel Benchmarks (CG, EP, IS) dla programowania równoległego oraz implementacje wzorców producent-konsument i echo-serwer dla programowania współbieżnego.

11.1.1. Wydajność implementacji równoległych

Benchmark EP (trywialnie równoległy)

Analiza benchmarku EP, reprezentującego klasę problemów o wysokim stopniu równoległości bez zależności międzyzadaniowych, ujawniła systematyczną przewagę architektury ARM64 nad x86_64. Implementacja Rust z biblioteką Rayon osiągnęła najwyższe wartości wydajności na platformie ARM (164 MFLOPS średnio), przewyższając zarówno Intel TBB (148 MFLOPS), jak i implementacje OpenMP. Na architekturze x86_64 wyniki dla Rust wynosiły 109 MFLOPS, co oznacza współczynnik przyspieszenia ARM/x86 na poziomie 1,55x - najwyższy spośród wszystkich badanych implementacji.

Analiza skalowania wykazała niemal liniowy wzrost wydajności do poziomu 5-6 wątków na obu architekturach, po czym następowało spłaszczenie charakterystyki ze względu na rosnące narzuty synchronizacji. Efektywność równoległości dla 8 wątków wynosiła około 68% dla ARM i 78% dla x86_64, co wskazuje na większą wrażliwość architektury ARM na koszty wielowątkowości przy wyższej liczbie wątków roboczych.

Benchmark CG (gradient sprzężony)

W przypadku benchmarku CG, charakteryzującego się intensywną komunikacją między wątkami i nieregularnymi wzorcami dostępu do pamięci, implementacja Intel TBB konsekwentnie osiągała najwyższą wydajność na obu platformach (5761 MFLOPS na ARM, 4757 MFLOPS na x86_64). Implementacja Rust również wykazywała lepszą wydajność na platformie ARM z współczynnikiem przyspieszenia ARM/x86 wynoszącym 1,62x.

Szczególnie istotną obserwacją był wpływ architektury na wzorce zarządzania pamięcią. Analiza operacji zwolnień stron pamięci wykazała, że implementacja new_omp generowała najwyższe liczby takich operacji (ponad 64 tys. dla klasy B), podczas gdy Rust charakteryzował się

stosunkowo niskim poziomem zwolnień we wszystkich klasach problemowych, co wskazuje na bardziej efektywne wykorzystanie pamięci przez alokator środowiska Rust.

Benchmark IS (sortowanie liczb całkowitych)

Benchmark IS ujawnił najbardziej zróżnicowane charakterystyki wydajnościowe między implementacjami. Implementacja Rust wykazywała problemy ze skalowalnością, osiągając współczynnik przyspieszenia oscylujący wokół 1,0 niezależnie od liczby wątków. W przeciwnieństwie do tego, Intel TBB demonstrował doskonałe skalowanie z współczynnikiem przyspieszenia osiągającym 4,8x w klasie W przy 6-8 wątkach.

Architektura ARM ponownie wykazywała przewagę wydajnościową, osiągając średnio 2,26x wyższą wydajność niż x86_64 dla implementacji TBB. Największe różnice były widoczne w klasach A i B (4506 a 3458 MFLOPS), co potwierdza efektywność architektury ARM w zadaniach intensywnie wykorzystujących przepustowość pamięci.

11.1.2. Analiza mechanizmów programowania współbieżnego

Wzorzec producent-konsument

Badanie implementacji wzorca producent-konsument ujawniło znaczące różnice w przepustowości komunikacji międzywątkowej między architekturami. Implementacje Rust z biblioteką Tokio osiągały przepustowość rzędu 10^4 – 10^5 wiadomości/s na architekturze ARM, podczas gdy implementacje C++ z wykorzystaniem tradycyjnych wątków osiągały jedynie 10^1 – 10^2 wiadomości/s na x86_64. Różnica ta wynosi 1-2 rzędy wielkości na korzyść architektury ARM.

Pod względem wykorzystania zasobów, implementacje Rust charakteryzowały się znacznie niższym zużyciem pamięci (około 3 MB) w porównaniu z implementacją C++ (około 10 MB). Wykorzystanie CPU było bardziej zróżnicowane na ARM (0-100%) niż na x86_64 (0-30%), co wskazuje na bardziej dynamiczne zarządzanie zasobami na platformie macOS.

Implementacja echo-serwera

W testach echo-serwera architektura ARM osiągnęła średnią przepustowość sieciową na poziomie 100 Mbps z wartościami szczytowymi do 600 Mbps, podczas gdy x86_64 utrzymywał średnią przepustowość na poziomie 20-30 Mbps. Implementacje C++ na ARM osiągały medianę przepustowości 170 Mbps w porównaniu do 100 Mbps dla Rust, co wskazuje na potencjał optymalizacji wydajności w C++ przy odpowiedniej konfiguracji.

Efektywność pamięciowa, wyrażona jako stosunek liczby obsłużonych wiadomości do zużytej pamięci, była znaczco wyższa na ARM (3-5 wiadomości/MB) niż na x86_64 (0,3-1 wiadomości/MB). Model asynchroniczny Rust wykazał najlepszą efektywność w tej metryce.

11.2. Wpływ architektury i modeli pamięci na wyniki

11.2.1. Modele zarządzania pamięcią

System własności i pożyczania w Rust eliminuje wyścigi danych na etapie komplikacji poprzez statyczną analizę modułu kontroli własności i pożyczania, co przekłada się na przewidywalne wzorce zarządzania pamięcią i często lepszą wydajność w scenariuszach z intensywnym wykorzystaniem pamięci. Koszt tego bezpieczeństwa przejawia się narzutem związany z koniecznością użycia konstrukcji Arc<Mutex<T>> w scenariuszach współdzielenia stanu, co

obserwowano jako spadek przepustowości o 8-12% względem bezpośredniego dostępu do pamięci współdzielonej w C++.

Model C++ oferuje większą elastyczność kosztem ryzyka błędów związanych z zarządzaniem pamięcią. Intel TBB wykorzystuje zaawansowane optymalizacje lokalności pamięci podręcznej, co jest szczególnie widoczne w wynikach benchmarku CG, gdzie różnice w wydajności między implementacjami sięgały 12-18% na korzyść TBB ze względu na lepsze zarządzanie fragmentacją pamięci przy dynamicznych kontenerach.

11.2.2. Architektura sprzętowa

Przewaga architektury ARM64 (Apple M1) może być przypisana kilku czynnikom technologicznym. Heterogeniczna architektura *big.LITTLE* z 4 rdzeniami wydajnościowymi (P-cores) i 4 energooszczędnymi (E-cores) umożliwia efektywniejsze zarządzanie obciążeniem wielowątkowym. Ujednoliciona architektura pamięci (ang. *unified memory architecture*), eliminuje koszty kopiowania danych między CPU a pamięcią, co jest szczególnie korzystne w aplikacjach intensywnie wykorzystujących przepustowość pamięci.

Hierarchiczna pamięć podręczna (12MB L2 dla P-cores, 4MB dla E-cores) przyczynia się do poprawy wydajności w warunkach zmiennej obciążenia. Niemniej jednak, jak wskazują autorzy [41], główne ograniczenia wynikają nie tyle z przepustowości przy intensywnej komunikacji między wątkami, ile z nieoptimalnego wykorzystania pamięci podręcznej, zwłaszcza w kontekście lokalności i alokacji danych w środowiskach wielowątkowych, co obserwowano jako szybszy spadek efektywności skalowania ARM przywiększej liczbie wątków.

11.2.3. Wpływ środowiska kompilacyjnego i bibliotek

Zidentyfikowane różnice między kompilatorami Clang (macOS) a GCC (Linux) wprowadzały zmienność wyników na poziomie 10-15% w testach współprzeźności. Clang wykazywał lepsze wsparcie dla instrukcji wektorowych na ARM, podczas gdy GCC zapewniał skuteczniejsze mechanizmy optymalizacyjne dla architektury x86_64.

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), co jest również potwierdzone przez pracę [67]. 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 jak podają autorzy [32], testy na maszynie wirtualnej z emulacją Intel wykazały spadek wydajności w porównaniu z natywnym wykonaniem na M1.

Klasa `std::jthread` została wprowadzona dopiero w standardzie C++20, który nie jest w pełni wspierany przez kompilator Apple Clang w wersji 15.0.0. Ograniczenia te należy uwzględnić przy tworzeniu aplikacji na system macOS, co wymusza zastosowanie alternatywnych rozwiązań programistycznych zapewniających kompatybilność z daną wersją kompilatora. Analiza wyników działania CMake wskazuje, że nawet przy użyciu kompilatora Clang w wersji 20.1.6, klasa `std::jthread` pozostaje niedostępna. Wynika to z faktu, iż jej obsługa zależy

nie tylko od samego kompilatora, lecz również od implementacji biblioteki standardowej C++. W środowisku macOS, mimo wykorzystania najnowszej wersji Clanga, systemowa wersja biblioteki `libc++` może nie zawierać jeszcze implementacji `std::jthread`. W związku z tym pełna zgodność ze standardem C++20 w zakresie zarządzania wątkami wymaga nie tylko odpowiedniego kompilatora, ale także aktualnej wersji biblioteki standardowej.

11.3. Rekomendacje praktyczne

11.3.1. Dobór technologii w zależności od charakterystyki problemu

Obliczenia typu trywialnie równoległego

Na podstawie wyników benchmarku EP, rekomenduje się Rust z biblioteką Rayon jako rozwiązanie główne ze względu na najwyższą osiągniętą wydajność (164 MFLOPS na ARM), bezpieczeństwo pamięci oraz abstrakcje bezkosztowe. Alternatywnie, dla istniejących baz kodu C++, zaleca się Intel TBB z uwagi na dobre skalowanie i dojrzałość rozwiązania.

Algorytmy z intensywną komunikacją międzywątkową

Dla algorytmów iteracyjnych i symulacji numerycznych zaleca się C++ z Intel TBB jako rozwiązanie główne, co potwierdza przewaga w benchmarkach CG i IS. Dojrzałe optymalizacje pamięci podrzecznej oraz zaawansowane techniki przydzielania zadań w czasie wykonywania czynią TBB optymalnym wyborem dla tej klasy problemów. Rust wymaga dalszych optymalizacji w tym obszarze, szczególnie w kontekście skalowalności przy nieregularnych wzorcach dostępu do pamięci.

Aplikacje sieciowe wysokiej przepustowości

Rekomenduje się Rust z biblioteką Tokio ze względu na model asynchroniczny M:N, wysoką efektywność pamięciową (3-5 wiadomości/MB) oraz wbudowane mechanizmy `async/await`. C++ może być rozważany w przypadkach wymagających precyzyjnej kontroli latencji pojedynczych operacji, szczególnie przy wykorzystaniu zaawansowanych bibliotek asynchronicznych.

11.3.2. Wybór platformy sprzętowej

Wybór platformy sprzętowej powinien być dostosowany do charakteru zadań oraz wymagań aplikacji. Architektura ARM64 (Apple Silicon) jest szczególnie korzystna dla obciążzeń o zmiennym charakterze oraz systemów, w których kluczowa jest efektywność energetyczna. Z kolei architektura x86_64 pozostaje preferowanym rozwiązaniem w przypadku klasycznych obliczeń wysokiej wydajności (HPC), środowisk wymagających pełnej kontroli nad przypisaniem wątków do procesorów, a także tam, gdzie istotne jest korzystanie z dojrzałego i stabilnego ekosystemu narzędzi oraz bibliotek. Ponadto, architektura ta gwarantuje kompatybilność z istniejącym kodem (ang. *legacy code*) i umożliwia precyzyjną kontrolę nad szczegółami implementacji.

11.3.3. Rekomendacje dotyczące środowiska kompilacyjnego

Dla C++ zaleca się wykorzystanie kompilatora Clang zamiast GCC do komplikacji kodu równoległego, co w badaniach skutkowało skróceniem czasu wykonania o 10-15% dzięki lepszemu wsparciu dla instrukcji wektorowych. W Rust zaleca się stosowanie Rayon do równoległego przetwarzania danych oraz Tokio do aplikacji asynchronicznych w celu minimalizacji narzutu kontekstu.

11.4. Ograniczenia metodologiczne badania

11.4.1. Heterogeniczność środowisk testowych

Największym ograniczeniem przeprowadzonych badań była niemożność zapewnienia jednolitego środowiska testowego. Wykorzystanie różnych kompilatorów (Clang 14.0.3 na macOS a GCC 11.4.0 na Linux), różnych systemów operacyjnych (macOS z jądrem Darwin a Linux) oraz różnych implementacji bibliotek standardowych (`libc++` a `libstdc++`) wprowadzało zmienne systemowe mogące wpływać na wyniki porównań.

11.4.2. Ograniczenia narzędzi diagnostycznych

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 [4], brak implementacji interfejsów `sched_setaffinity()`/`sched_getaffinity()` uniemożliwia odczyt i kontrolę przypisań procesów do rdzeni (ang. *process-to-core binding*), co stanowi barierę metodologiczną w porównawczych badaniach wydajnościowych między architekturami ARM (M1) i x86_64. W odróżnieniu od pełnej funkcjonalności `hwloc-ps` w systemie Linux [7], 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 (ang. *thread pinning*) na wydajność implementacji współbieżnych w językach Rust/C++.

Różnice w narzędziach profilowania (`Instruments` na macOS a `perf` na Linux) mogły wprowadzać dodatkową zmienność w pomiarach wykorzystania zasobów.

11.4.3. Ograniczenia doboru benchmarków

Wybrane benchmarki NAS (CG, EP, IS) reprezentują głównie problemy o regularnej strukturze obliczeniowej charakterystyczne dla klasycznych zastosowań HPC. Współczesne aplikacje często charakteryzują się nieregularnymi wzorcami dostępu do pamięci, dynamiczną alokacją zadań oraz hybrydowymi modelami obliczeniowymi, co ogranicza generalizację wniosków do innych domen aplikacyjnych.

11.5. Kierunki dalszych badań

11.5.1. Ujednolicone środowisko badawcze

Priorytetem dla przyszłych badań powinno być przeprowadzenie eksperymentów na maszynach Apple wyposażonych zarówno w procesory Intel, jak i Apple Silicon, co pozwoliłoby na:

- Eliminację zmienności wynikającej z różnic systemowych między macOS a Linux,
- Użycie identycznego kompilatora (Clang) i bibliotek systemowych,
- Precyzyjną izolację wpływu architektury procesora od wpływu środowiska systemowego,
- Kontrolę nad identycznością mechanizmów szeregowania zadań.

11.5.2. Rozszerzenie zakresu architektur i platform

Architektury heterogeniczne

Dalsze badania powinny uwzględnić systemy hybrydowe wykorzystujące:

- GPU poprzez CUDA (C++) i rust-gpu/wgpu (Rust),
- Systemy SoC z dedykowanymi akceleratorami (NPU, DSP),
- Platformy FPGA w kontekście programowania wysokiej wydajności,
- Procesory RISC-V jako alternatywę dla ARM i x86.

Nowe paradygmaty programowania

Eksploracja powinna obejmować:

- Model aktorowy (Actix dla Rust a CAF dla C++),
- Programowanie przepływu danych oraz reaktywne strumienie danych,
- Mechanizmy C++20 coroutines a Rust async/await,
- Hybrydowe modele łączące synchroniczne i asynchroniczne wzorce.

11.6. Wkład pracy

Pierwsze porównanie wydajności benchmarków NAS Parallel Benchmarks na architekturach ARM64 oraz x86_64 w językach Rust i C++

Niniejsza praca stanowi pierwsze w literaturze porównanie wydajności mechanizmów programowania równoległego między językami Rust i C++ z wykorzystaniem standardowych benchmarków NAS Parallel Benchmarks na architekturach ARM64 (Apple Silicon) i x86_64. Dotychczasowe badania koncentrowały się głównie na platformach x86 lub nie uwzględniały nowoczesnych architektur ARM w kontekście zastosowań HPC. Wykazanie systematycznej przewagi architektury ARM64 w większości testowanych scenariuszy (współczynniki przyspieszenia 1,55x–2,26x) stanowi istotne odkrycie dla społeczności zajmującej się obliczeniami wysokiej wydajności.

Empiryczna walidacja abstrakcji wysokopoziomowych w programowaniu równoległym

Badania empirycznie potwierdzają tezę, że nowoczesne abstrakcje wysokopoziomowe (Rust Rayon, Intel TBB) nie tylko zwiększą bezpieczeństwo programowania, ale również oferują wydajność konkurencyjną lub wyższą od tradycyjnych rozwiązań niskopoziomowych (klasyczne OpenMP). Szczególnie znaczące jest wykazanie, że biblioteka Rayon osiąga najwyższą wydajność w benchmarku EP (164 MFLOPS na ARM), przewyższając dojrzałe implementacje OpenMP i TBB.

Analiza wpływu modeli zarządzania pamięcią na wydajność współbieżną w kontekście architektur ARM64 i x86_64

Niniejsza praca wnosi nowe spojrzenie na oddziaływanie różnych modeli zarządzania pamięcią, w szczególności systemu własności w języku Rust oraz manualnego zarządzania pamięcią w C++, na wydajność aplikacji wielowątkowych. Przeprowadzone systematyczne porównania na architekturach ARM64 oraz x86_64 umożliwiły empiryczne wykazanie, że eliminacja wyściągów danych na etapie kompilacji w Rust przekłada się na przewidywalne wzorce wykorzystania pamięci. Obserwowany narzuć wydajnościowy różni się w zależności od platformy - wynosi około 8-12% na architekturze x86_64, podczas gdy na ARM64 ujawnia inne charakterystyki. Wyniki te dostarczają istotnych informacji umożliwiających lepsze zrozumienie kompromisów pomiędzy bezpieczeństwem a efektywnością w programowaniu systemowym, zwłaszcza w kontekście środowisk wieloplatformowych.

Identyfikacja ograniczeń metodologicznych w badaniach wieloplatformowych

Badania ujawniły ograniczenia w dostępnych narzędziach diagnostycznych dla architektur ARM na platformie macOS, szczególnie w kontekście `hwloc-ps` i mechanizmów przypisania procesu do konkretnego rdzenia procesora. Zidentyfikowanie tych ograniczeń oraz ich wpływu na wiarygodność pomiarów między architekturami może pomóc przyszłym badaczom w wyborze narzędzi pomiarowych.

Walidacja efektywności architektur heterogenicznych w programowaniu równoległym

Wyniki potwierdzają teoretyczne założenia dotyczące korzyści płynących z heterogenicznych architektur *big.LITTLE* (Apple M1) w kontekście aplikacji wielowątkowych. Empiryczne wykazano, że architektura ta zapewnia lepszą wydajność przy zmiennym obciążeniu, przy jednoczesnej identyfikacji jej ograniczeń przy wysokiej liczbie wątków (spadek efektywności do 68% a 78% na x86_64).

Praktyczne wytyczne dla wyboru technologii w programowaniu współbieżnym

Na podstawie systematycznych pomiarów wydajności sformułowano empirycznie uzasadnione wytyczne dla wyboru języka programowania i bibliotek w zależności od charakterystyki problemu obliczeniowego i docelowej architektury. Rekomendacje te, oparte na konkretnych danych wydajnościowych, stanowią praktyczny wkład dla inżynierów oprogramowania projektujących systemy współbieżne i równoległe.

11.7. Podsumowanie wniosków

Przeprowadzone badania wykazały, że wybór między Rust a C++ oraz między różnymi bibliotekami równoległości powinien być podektywowany specyfiką problemu obliczeniowego, wymaganiami wydajnościowymi, docelową architekturą sprzętową oraz kontekstem organizacyjnym. Nie istnieje uniwersalne optymalne rozwiązanie - każde z badanych podejść wykazuje przewagi w określonych scenariuszach.

Rust z biblioteką Rayon wyłania się jako silna alternatywa dla C++ w dziedzinie programowania systemowego i współbieżnego, oferując unikalne gwarancje bezpieczeństwa przy zachowaniu wysokiej wydajności, szczególnie na architekturach ARM. C++, z bogatym ekosystemem i dojrzałymi narzędziami takimi jak Intel TBB, pozostaje kluczowym językiem dla aplikacji wymagających najwyższej wydajności, zwłaszcza na architekturach x86_64 w zastosowaniach HPC.

Architektura ARM64, reprezentowana przez Apple Silicon, udowodniła swoją konkurencyjność względem x86_64, oferując często wyższą wydajność przy niższym zużyciu energii. Wymaga to jednak adaptacji istniejących narzędzi i bibliotek, co stanowi wyzwanie dla szerskiego przyjęcia.

Dalszy rozwój obu języków, postępująca dywersyfikacja platform sprzętowych oraz rosnące wymagania aplikacji będą stymulować ewolucję mechanizmów programowania współbieżnego jak i równoległego. Kontynuacja badań porównawczych, uwzględniająca nowe architektury i paradygmaty programowania, pozostaje niezbędna dla świadomego wyboru technologii w szybko zmieniającym się środowisku obliczeniowym.

Rozdział 12

Podsumowanie

Celem niniejszej pracy było przeprowadzenie pogłębionej analizy oraz wszechstronnego porównania mechanizmów programowania współbieżnego i równoległego w językach Rust i C++, ze szczególnym uwzględnieniem efektywności, bezpieczeństwa oraz praktyczności stosowania dostępnych narzędzi wielowątkowych w obu językach. W tym kontekście zrealizowano kompleksową analizę porównawczą na architekturach ARM64 (Apple Silicon M1) oraz _64 (Intel). Zastosowana metodologia eksperymentalna, wykorzystująca zestawy testowe NAS Parallel Benchmarks oraz implementacje aplikacji współbieżnych, umożliwiła systematyczne porównanie wydajności bibliotek Rust Rayon, Intel TBB oraz OpenMP.

Badania wykazały znaczącą przewagę architektury ARM64 nad x86_64 we wszystkich scenariuszach testowych, ze współczynnikami przyspieszenia 1,55x-2,26x. Implementacja z biblioteką Rayon osiągnęła najwyższą wydajność w obliczeniach trywialnie równoległych (164 a 109 MFLOPS), podczas gdy Intel TBB dominowała w zadaniach z intensywną komunikacją międzywątkową. Kluczowym odkryciem było wykazanie, że model bezpieczeństwa pamięci w Rust nie generuje istotnych narzutów wydajnościowych, a w niektórych przypadkach przyczynia się do lepszej efektywności. W aplikacjach sieciowych architektura ARM uzyskała przepustowość o rząd wielkości wyższą przy zmniejszonym zużyciu pamięci.

Główne ograniczenia metodologiczne obejmowały heterogeniczność środowisk testowych (macOS a Linux) oraz ograniczenia narzędzi diagnostycznych na platformie macOS, zwłaszcza w kontekście kontroli przypisań wątków do rdzeni. Dodatkowo, wybrane benchmarki NAS reprezentują problemy o regularnej strukturze obliczeniowej, co ogranicza generalizację wniosków do aplikacji o nieregularnych wzorach dostępu do pamięci. Praca wypełnia istotną lukę badawczą, dostarczając kompleksowej analizy porównawczej mechanizmów programowania równoległego w kontekście nowoczesnych architektur sprzętowych. Wyniki potwierdzają porównywalną wydajność Rust względem C++, jednocześnie uwydatniając wpływ architektury na efektywność mechanizmów współbieżności.

W kontekście dalszych badań sugeruje się ujednolicenie środowiska testowego poprzez wykorzystanie maszyn Apple z procesorami Intel i Apple Silicon, rozszerzenie zakresu na architektury heterogeniczne oraz eksplorację nowych paradygmatów programowania, w tym modelu aktorowego i mechanizmów C++20 coroutines w porównaniu z Rust `async/await`.

Podsumowując, przeprowadzone badania pogłębiają wiedzę na temat charakterystyk wydajnościowych nowoczesnych języków programowania w obliczeniach równoległych oraz wskazują na rosnący potencjał architektury ARM64 w zastosowaniach wysokowydajnościowych. Praca stanowi solidną podstawę do dalszych badań nad optymalizacją mechanizmów współbieżności w systemach wymagających wysokiej wydajności i niezawodności.

Bibliografia

- [1] Amdahl's law - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Amdahl%27s_law. [Dostęp 11-06-2025].
- [2] C++ - Wikipedia — en.wikipedia.org. <https://en.wikipedia.org/wiki/C%2B%2B>. [Dostęp 16-03-2025].
- [3] Comparing Programming Languages — cs.ucf.edu. <https://www.cs.ucf.edu/~leavens/ComS541Fall98/hw-pages/comparing/>. [Dostęp 16-03-2025].
- [4] CPU binding on MacOS · Issue #555 · open-mpi/hwloc — github.com. <https://github.com/open-mpi/hwloc/issues/555>. [Dostęp 01-06-2025].
- [5] CUDA Toolkit - Free Tools and Training — developer.nvidia.com. <https://developer.nvidia.com/cuda-toolkit>. [Dostęp 22-03-2025].
- [6] Execution control library (since C++26) - cppreference.com — en.cppreference.com. <https://en.cppreference.com/w/cpp/execution.html>. [Dostęp 31-05-2025].
- [7] Hardware Locality (hwloc): Hardware Locality — hwloc.readthedocs.io. <https://hwloc.readthedocs.io/en/stable/>. [Dostęp 01-06-2025].
- [8] 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].
- [9] NAS Parallel Benchmarks — nas.nasa.gov. <https://www.nas.nasa.gov/software/npb.html>. [Dostęp 17-04-2025].
- [10] Rust GPU — rust-gpu.github.io. <https://rust-gpu.github.io>. [Dostęp 22-03-2025].
- [11] Rust (programming language) - Wikipedia — en.wikipedia.org. [https://en.wikipedia.org/wiki/Rust_\(programming_language\)](https://en.wikipedia.org/wiki/Rust_(programming_language)). [Dostęp 16-03-2025].
- [12] Safety: A comparaison between Rust, C++ and Go | Hacker News — news.ycombinator.com. <https://news.ycombinator.com/item?id=32285122>. [Dostęp 13-03-2025].
- [13] vulkano - Rust — docs.rs. <https://docs.rs/vulkano/0.12.0/vulkano/>. [Dostęp 22-03-2025].
- [14] wgpu: portable graphics library for Rust — wgpu.rs. <https://wgpu.rs>. [Dostęp 22-03-2025].
- [15] *Professional CUDA C Programming*. Wrox Press Ltd., GBR, wydanie 1st, 2014. ISBN: 1118739329.

- [16] 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, strony 27–40, New York, NY, USA, 2024. Association for Computing Machinery. DOI: 10.1145/3626183.3659966.
- [17] T. Adefemi. What Every Computer Scientist Needs To Know About Parallelization. *arXiv preprint arXiv:2504.03647*, Luty 2025. preprint.
- [18] S. Ali, S. Qayyum. A Pragmatic Comparison of Four Different Programming Languages. <https://doi.org/10.14293/S2199-1006.1.SOR-.PP5RV10.v1>, 06 2021. [Dostęp 16-03-2025].
- [19] Z. Alomari, O. E. Halimi, K. Sivaprasad, C. Pandit. Comparative studies of six programming languages. <https://arxiv.org/abs/1504.00693>, 2015. [Dostęp 16-03-2025].
- [20] 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.
- [21] 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.
- [22] 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].
- [23] J. Blandy, J. Orendorff, L. Tindall. *Programming Rust*. O'Reilly Media, 2021. ISBN: 9781492052548.
- [24] K. Bobrov. *Grokking Concurrency*. Manning, 2024. ISBN: 9781633439771.
- [25] M. Bos. *Rust Atomics and Locks: Low-Level Concurrency in Practice*. O'Reilly Media, 2022. ISBN: 9781098119409.
- [26] ByteByteGo. EP108: How do we design a secure system? — blog.bytebytogo.com. https://blog.bytebytogo.com/p/ep108-how-do-we-design-a-secure-system?utm_campaign=post&utm_medium=web. [Dostęp: 09-03-2025].
- [27] R. Chandra. *Parallel Programming in OpenMP*. High performance computing. Elsevier Science, 2001.
- [28] 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. DOI: 10.1145/3678232.3678258.
- [29] I. Corporation. Best Practices for Real-Time Optimizations With the 12th Generation Intel® Core™ Processors. Technical paper, Intel Corporation, 2022. <https://www.intel.com/content/www/us/en/content-details/737973/best-practices-for-real-time-optimizations-with-the-12th-generation-intel-core-processors-technical-paper.html>[Dostęp: 2025-06-11].
- [30] 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.
- [31] R. DeWolf. Introducing Rainbow: Compare the Performance of Different Programming Languages — medium.com. [https://medium.com/better-](https://medium.com/better)

- programming/introducing-rainbow-compare-the-performance-of-different-programming-languages-f08a67453cd4. [Dostęp 16-03-2025].
- [32] J. Duke. Memory Forensics Comparison of Apple M1 and Intel Architecture Using Volatility Framework. Master's thesis, Louisiana State University, 2021.
- [33] Z. Fehervari. Rust vs C++: Modern Developers' Dilemma — bluebirdinternational.com. <https://bluebirdinternational.com/rust-vs-c/>. [Dostęp 28-01-2025].
- [34] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972. [Dostęp 11-06-2025].
- [35] 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.
- [36] M. Galvin. *Mastering Concurrency And Parallel Programming: Atain Advanced Techniques and Best Practices for Crafting Robust, Maintainable, and High-Performance Concurrent Code in C++*. 2024.
- [37] Y. Gusakov. C++ Vs. Rust: 6 Key Differences — QIT — qit.software. <https://qit.software/c-vs-rust-6-key-differences/>. [Dostęp 28-01-2025].
- [38] P. Hazarika, R. Budania. Rust vs C++: Performance, Safety, and Use Cases Compared. https://www.codeporting.com/blog/rust_vs_cpp_performance_safety_and_use_cases_compared, 2025. Dostęp: 2025-06-12.
- [39] H. Heyman, L. Brandefelt. *A Comparison of Performance & Implementation Complexity of Multithreaded Applications in Rust, Java and C++*. Praca doktorska, 2020.
- [40] 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].
- [41] M. Jalili, M. Erez. Harvesting L2 Caches in Server Processors. <https://arxiv.org/abs/2301.04228>, 2023. [Dostęp: 2025-06-11].
- [42] R. Jung, J.-H. Jourdan, R. Krebbers, D. Dreyer. Safe Systems Programming in Rust. *Communications of the ACM*, 64(4):144–152, 2021.
- [43] S. Klabnik, C. Nichols. *The Rust Programming Language*. No Starch Press, USA, 2018. ISBN: 1593278284.
- [44] B. Köpcke, S. Gorlatch, M. Steuwer. Descend: A Safe Gpu Systems Programming Language. *Proceedings of the ACM on Programming Languages*, 8, 2024. DOI: 10.1145/3656411.
- [45] 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.
- [46] P. Larsen. Migrating C to Rust for Memory Safety . *IEEE Security & Privacy*, 22(04):22–29, Lip. 2024.
- [47] K. Lesiński. Speed of Rust vs C. <https://kornel.ski/rust-c-speed>, 2019. [Dostęp: 10-01-2025].
- [48] 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.

- [49] M. Lindgren. Introduction - Comparing parallel Rust and C++ — parallel-rust-cpp.github.io. <https://parallel-rust-cpp.github.io>. [Dostęp 28-01-2025].
- [50] 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. [Dostęp 17-04-2025].
- [51] E. M. Martins, L. G. Faé, R. B. Hoffmann, L. S. Bianchessi, D. Griebler. NPB-Rust: NAS Parallel Benchmarks in Rust. <https://arxiv.org/abs/2502.15536>, 2025. [Dostęp: 2025-05-18].
- [52] M. Mitzenmacher, E. Upfal. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis*. Cambridge University Press, 2017. ISBN: 9781108107990.
- [53] 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.
- [54] 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, Lip. 2013. [Dostęp 16-03-2025].
- [55] P. Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011.
- [56] 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].
- [57] 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.
- [58] L. Rinaldi, M. Torquati, M. Danelutto. Enforcing Reference Capability in FastFlow with Rust. *Advances in Parallel Computing*, 36:396 – 405, 2020.
- [59] 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. [Dostęp 16-03-2025].
- [60] A. Saligrama, A. Shen, J. Gjengset. A Practical Analysis of Rust’s Concurrency Story. *arXiv preprint arXiv:1904.12210*, Kwi. 2019. [Dostęp: 12-06-2025].
- [61] I. M. Sesay. The Critical Role of Parallelism and Multicore Processors in Advancing High-Level Programming Practices. *Global Science Journal*, Wrze. 2024. [Dostęp: 2025-06-12].
- [62] M. Shehryar, F. H. Khan. State-of-the-Art Multi-Core Architectures: Analyzing Current Research, Identifying Gaps, and Exploring Future Directions. *2024 21st International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, strony 163–168, 2024. [Dostęp 12-06-2025].
- [63] J. Sible, D. Svoboda. Rust Software Security: A Current State Assessment. <https://doi.org/10.58012/0px4-9n81>, Dec 2022. [Dostęp: 12-03-2025].
- [64] T. Silva, J. Bispo, T. Carvalho. Foundations for a Rust-Like Borrow Checker for C. strony 155 – 165, 2024. [Dostęp 16-03-2025].

- [65] 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].
- [66] B. Troutwine. *Hands-On Concurrency with Rust: Confidently build memory-safe, parallel, and efficient software in Rust*. Packt Publishing, 2018. ISBN: 9781788478359.
- [67] A. Tuby, A. Morrison. Reverse Engineering the Apple M1 Conditional Branch Predictor for Out-of-Place Spectre Mistraining. <https://arxiv.org/abs/2502.10719>, 2025. [Dostęp: 2025-05-18].
- [68] U.S. Department of Defense Chief Information Officer. The Case for Memory-Safe Roadmaps. <https://media.defense.gov/2023/Dec/06/2003352724/-1/-1/0/THE-CASE-FOR-MEMORY-SAFE-ROADMAPS-TLP-CLEAR.PDF>, December 2023. [Dostęp: 12-06-2025].
- [69] T. Vandervelden, R. De Smet, D. Deac, K. Steenhaut, A. Braeken. Overview of Embedded Rust Operating Systems and Frameworks. *Sensors*, 24(17), 2024. [Dostęp: 23-12-2024].
- [70] R. Viitanen. *Evaluating Memory Models for Graph-Like Data Structures in the Rust Programming Language: Performance and Usability*. Praca doktorska, 2020.
- [71] 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].
- [72] A. Williams. *C++ Concurrency in Action*. Manning, 2019. ISBN: 9781638356356.
- [73] X. Yin, Z. Huang, S. Kan, G. Shen. SafeMD: Ownership-Based Safe Memory Deallocation for C Programs. *Electronics*, 13(21), 2024. [Dostęp 16-03-2025].
- [74] Z. Yu, L. Song, Y. Zhang. Fearless Concurrency? Understanding Concurrent Programming Safety in Real-World Rust Software. <https://arxiv.org/abs/1902.01906>, 2019. [Dostęp 28-01-2025].

Spis rysunków

2.1. Różnice między wykonywaniem zadań współbieżnie a równolegle [26]	15
3.1. Kroki komplikacji w języku Rust	25
3.2. Kroki komplikacji w języku C++	25
4.1. Spektrum nieuustraszonej współbieżności [16]	32
6.1. Topologia procesora Apple M1 z widoczną architekturą big.LITTLE	60
6.2. Topologia procesora Intel Core Ultra 7 165H z widoczną architekturą heterogeniczną	61
9.1. Porównanie czasów wykonania benchmarku CG dla klas S, W, A, B względem liczby użytych wątków na platformie ARM64	84
9.2. Analiza zmienności czasów wykonania benchmarku CG dla klas S, W, A, B względem liczby użytych wątków na platformie ARM64	85
9.3. Porównanie wydajności benchmarku CG dla klas S, W, A, B względem liczby użytych wątków na platformie ARM64	86
9.4. Mapa ciepła wydajności benchmarku CG dla klas S, W, A, B względem liczby użytych wątków na platformie ARM64	87
9.5. Analiza skalowania benchmarku CG dla klas S, W, A, B względem liczby użytych wątków na platformie ARM64	88
9.6. Porównanie czasów wykonania benchmarku CG dla klas S, W, A, B względem liczby użytych wątków na platformie x86_64	90
9.7. Analiza zmienności czasów wykonania benchmarku CG dla klas S, W, A, B względem liczby użytych wątków na platformie x86_64	90
9.8. Porównanie wydajności benchmarku CG dla klas S, W, A, B względem liczby użytych wątków na platformie ARM64	91
9.9. Mapa ciepła wydajności benchmarku CG dla klas S, W, A, B względem liczby użytych wątków na platformie ARM64	92
9.10. Analiza skalowania benchmarku CG dla klas S, W, A, B względem liczby użytych wątków na platformie ARM64	93
9.11. Profilowanie wydajności benchmarku CG dla klas S, W, A, B względem liczby użytych wątków na platformie ARM64	94
9.12. Analiza wydajności względem <code>old_omp</code> (punkt odniesienia) z wartościami bezwzględnymi na platformie ARM64	95
9.13. Kompromisy (ang. <i>trade-off</i>) pomiędzy zużyciem pamięci a błędami stron pamięci, z uwzględnieniem liczby odzyskanych stron jako trzeciej zmiennej reprezentowanej przez rozmiar bąbla na platformie ARM64	97
9.14. Profilowanie wydajności benchmarku CG dla klas S, W, A, B względem liczby użytych wątków na platformie x86_64	98
9.15. Porównanie średniej wydajności benchmarku CG dla platform ARM64 i x86_64 . .	99
9.16. Szczegółowa analiza wydajności benchmarku CG dla platform ARM64 i x86_64 .	100
9.17. Analiza istotności statystycznej benchmarku CG dla platform ARM64 i x86_64 .	101

9.18. Porównanie czasów wykonania benchmarku EP dla klas S, W, A, B względem liczby użytych wątków	102
9.19. Analiza zmienności czasów wykonania benchmarku EP dla klas S, W, A, B względem liczby użytych wątków	103
9.20. Porównanie wydajności benchmarku EP dla klas S, W, A, B względem liczby użytych wątków	104
9.21. Mapa ciepła wydajności benchmarku EP dla klas S, W, A, B względem liczby użytych wątków	105
9.22. Analiza skalowania benchmarku EP dla klas S, W, A, B względem liczby użytych wątków	106
9.23. Porównanie czasów wykonania benchmarku EP dla klas S, W, A, B względem liczby użytych wątków	107
9.24. Analiza zmienności czasów wykonania benchmarku EP dla klas S, W, A, B względem liczby użytych wątków	107
9.25. Porównanie wydajności benchmarku EP dla klas S, W, A, B względem liczby użytych wątków	108
9.26. Mapa ciepła wydajności benchmarku EP dla klas S, W, A, B względem liczby użytych wątków	109
9.27. Analiza skalowania benchmarku EP dla klas S, W, A, B względem liczby użytych wątków	110
9.28. Profilowanie wydajności benchmarku EP dla klas S, W, A, B względem liczby użytych wątków	111
9.29. Analiza wydajności względem <code>old_omp</code> (punkt odniesienia) z wartościami bezwzględnymi	112
9.30. Kompromisy (ang. <i>trade-off</i>) pomiędzy zużyciem pamięci a błędami stron pamięci, z uwzględnieniem liczby odzyskanych stron jako trzeciej zmiennej reprezentowanej przez rozmiar bąbla	112
9.31. Profilowanie wydajności benchmarku EP dla klas S, W, A, B względem liczby użytych wątków	113
9.32. Znormalizowana mapa metryk pamięciowych	113
9.33. Porównanie średniej wydajności benchmarku EP dla platform ARM64 i x86_64 . .	115
9.34. Szczegółowa analiza wydajności benchmarku EP dla platform ARM64 i x86_64 . .	116
9.35. Analiza istotności statystycznej benchmarku EP dla platform ARM64 i x86_64 . .	117
9.36. Porównanie czasów wykonania benchmarku IS dla klas S, W, A, B względem liczby użytych wątków	118
9.37. Analiza zmienności czasów wykonania benchmarku IS dla klas S, W, A, B względem liczby użytych wątków	119
9.38. Porównanie wydajności benchmarku IS dla klas S, W, A, B względem liczby użytych wątków	120
9.39. Mapa ciepła wydajności benchmarku IS dla klas S, W, A, B względem liczby użytych wątków	121
9.40. Analiza skalowania benchmarku IS dla klas S, W, A, B względem liczby użytych wątków	122
9.41. Porównanie czasów wykonania benchmarku IS dla klas S, W, A, B względem liczby użytych wątków	123
9.42. Analiza zmienności czasów wykonania benchmarku IS dla klas S, W, A, B względem liczby użytych wątków	124
9.43. Porównanie wydajności benchmarku IS dla klas S, W, A, B względem liczby użytych wątków	125

9.44. Mapa ciepła wydajności benchmarku IS dla klas S, W, A, B względem liczby użytych wątków	126
9.45. Analiza skalowania benchmarku IS dla klas S, W, A, B względem liczby użytych wątków	127
9.46. Profilowanie wydajności benchmarku IS dla klas S, W, A, B względem liczby użytych wątków	128
9.47. Analiza wydajności względem <code>old_omp</code> (punkt odniesienia) z wartościami bezwzględnymi	129
9.48. Kompromisy (ang. <i>trade-off</i>) pomiędzy zużyciem pamięci a błędami stron pamięci, z uwzględnieniem liczby odzyskanych stron jako trzeciej zmiennej reprezentowanej przez rozmiar bąbla	129
9.49. Profilowanie wydajności benchmarku IS dla klas S, W, A, B względem liczby użytych wątków	130
9.50. Znormalizowana mapa ciepła metryk wydajności	131
9.51. Porównanie średniej wydajności benchmarku IS dla platform ARM64 i x86_64	132
9.52. Szczegółowa analiza wydajności benchmarku IS dla platform ARM64 i x86_64	133
9.53. Analiza istotności statystycznej benchmarku IS dla platform ARM64 i x86_64	134
10.1. Analiza wyników benchmarku producent-klient	135
10.2. Analiza wyników benchmarku producent-klient	138
10.3. Porównanie wydajności komunikacji między architekturami ARM i x86	140
10.4. Analiza skalowania względem liczby wątków	141
10.5. Analiza zużycia pamięci	142
10.6. Analiza statystyczna różnic między architekturami ARM i x86	143
10.7. Analiza wyników benchmarku echo serwer	145
10.8. Analiza wyników benchmarku echo serwer	148
10.9. Porównanie wydajności sieciowej pomiędzy platformami ARM i x86_64	150
10.10. Analiza skalowania względem obciążenia sieciowego	152

Spis tabel

3.1. Kwerendy użyte w bazie Scopus ¹	21
3.2. Przebieg selekcji literatury (baza Scopus)	23
3.3. Kwerendy użyte w bazie Scopus ²	23
3.4. Przebieg selekcji literatury (baza Google Scholar)	23
6.1. Parametry eksperymentów producent-konsument	55
6.2. Parametry eksperymentów serwera echa	55
6.3. Specyfikacja platformy ARM64	60
6.4. Specyfikacja platformy x86_64	61
6.5. Wersje kompilatorów na platformach testowych	61
7.1. Porównanie aspektów zarządzania i organizacji kodu w Rust i różnych stylach C++	73
7.2. Porównanie modelu alokacji i bezpieczeństwa pamięci w Rust i różnych stylach C++	74
8.1. Porównanie aspektów struktury i organizacji kodu w implementacjach współbieżnych	75
8.2. Porównanie modeli zarządzania pamięcią w badanych implementacjach	76
8.3. Porównanie mechanizmów współbieżności w badanych implementacjach	77
8.4. Porównanie wydajności i bezpieczeństwa implementacji współbieżnych	82

Spis listingów

3.1. Kwerenda wygenerowana przez AI	22
4.1. Przykład mechanizmu borrow	32
4.2. Inteligentny wskaźnik Box	33
4.3. Inteligentny wskaźnik RC	33
4.4. Inteligentny wskaźnik Arc	34
4.5. Przykład użycia unsafe Rust	34
4.6. Przykład użycia Tokio	35
4.7. Przykład użycia kanałów Crossbeam	36
4.8. Przykład użycia Actix	36
4.9. Przykład tworzenia kanału	37
4.10. Przykład z wieloma wątkami	38
4.11. Zakończenie kanału	39
4.12. Przykład użycia semafora	40
4.13. Przykład użycia par_iter	41
4.14. Przykład tworzenia wątku	42
4.15. Przykład użycia Mutex	43
4.16. Przykład użycia RwLock	43
4.17. Przykład użycia Atomic	44
4.18. Przykład użycia bariery	45
5.1. Przykład użycia std::jthread	46
5.2. Przykład komunikacji między wątkami	47
5.3. Przykład użycia std::scoped_lock	47
5.4. Przykład użycia std::latch oraz std::barrier	48
5.5. Przykład użycia std::async	49
5.6. Przykład użycia std::promise	49
5.7. Przykład użycia std::packaged_task	50
5.8. Przykład użycia OpenMP w C++	51
5.9. Przykład użycia Intel TBB w C++	52
7.1. Struktura kodu benchmarków w języku Rust	63
7.2. Zarządzanie pamięcią w benchmarkach NPB w języku Rust	64
7.3. Równoległość w benchmarkach NPB w języku Rust	64
7.4. Implementacja benchmarku EP w języku Rust	65
7.5. Implementacja benchmarku CG w języku Rust	65
7.6. Implementacja benchmarku IS w języku Rust	66
7.7. Struktura kodu benchmarków w języku C++ z OpenMP	66
7.8. Zarządzanie pamięcią w benchmarkach C++ z OpenMP	67
7.9. Mechanizmy równoległości w benchmarkach C++ z OpenMP	68
7.10. Implementacja benchmarku EP w języku C++ z OpenMP	68
7.11. Implementacja benchmarku CG w języku C++ z OpenMP	69
7.12. Implementacja benchmarku IS w języku C++ z OpenMP	70

Spis listingów

7.13. Implementacja TBB - struktura kodu	70
7.14. Implementacja TBB - równoległość	71
7.15. Implementacja nowoczesnego C++ - struktura kodu	72
7.16. Implementacja nowoczesnego C++ - zarządzanie pamięcią	72
7.17. Implementacja nowoczesnego C++ - równoległość	73
8.1. Producent-Konsument w Rust z mpsc channels	78
8.2. Producent-Konsument w C++ z Channel	78
8.3. Echo Serwer w Rust z Tokio	79
8.4. Echo Serwer w C++ z jednym wątkiem na połączenie	79
8.5. Async Echo Serwer w C++ z event loop	80