Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

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

PRACA DYPLOMOWA MAGISTERSKA

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

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

Rafał Jasiński

Opiekun pracy
dr inż. Zdzisław Spławski

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

WROCŁAW 2025

Streszczenie

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

Kolejne rozdziały przedstawiają mechanizmy współbieżności, omawiając biblioteki Rust Rayon i Tokio oraz C++ OpenMP i Intel TBB. Metodologia badań obejmuje środowisko testowe 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) oraz aplikacje testowe (producent-konsument, echo-serwer) w obu językach. Analiza wyników obejmuje pomiary wydajności w MFLOPS, skalowanie względem liczby wątków oraz zużycie zasobów systemowych.

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.

Praca dostarcza pierwszego kompleksowego porównania tych języków na dwóch architekturach oraz praktycznych rekomendacji wyboru technologii.

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

Abstract

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

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

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

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.

The work provides the first comprehensive comparison of these languages across two architectures and practical technology selection recommendations.

Keywords: concurrency and parallelism, Rust and C++, ARM and x86_64

Spis treści

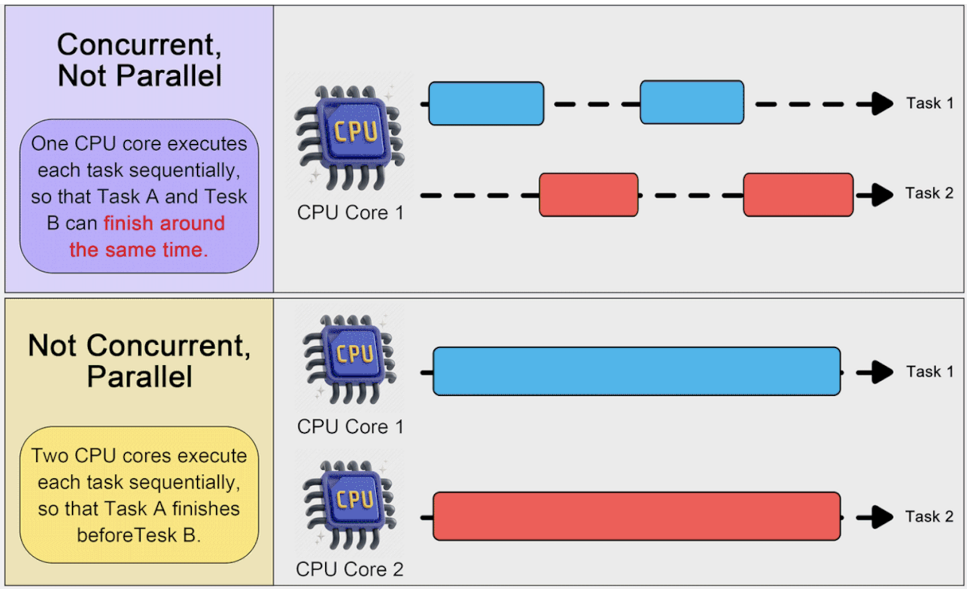
Rozdział 1

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]. W niniejszej pracy autor postanowił zbadać podejścia do programowania równoległego jak i współbieżnego zarówno w kontekście języka Rust, jak i C++. Rust jest relatywnie młodym językiem, zaprojektowanym z myślą o bezpieczeństwie pamięci i unikaniu typowych błędów wielowątkowych. Z kolei C++ to język o ugruntowanej pozycji, znany z elastyczności i wysokiej wydajności, co czyni go popularnym wyborem dla aplikacji wymagających precyzyjnego zarządzania zasobami.

Programowanie współbieżne (ang. *concurrent programming*) oraz programowanie równoległe (ang. *parallel programming*) to dwa różne, lecz uzupełniające się podejścia, które umożliwiają organizację pracy wielu zadań w aplikacji - poglądowa różnica została zamieszczona na rysunku 1.1. Chociaż często używa się ich zamiennie, ich cechy i cele różnią się znacznie.



Rys. 1.1: Różnice między wykonywaniem zadań współbieżnie a równolegle [26]

1.1. Programowanie współbieżne

Programowanie współbieżne to podejście w projektowaniu aplikacji, które umożliwia równoczesne wykonywanie wielu zadań, choć faktycznie procesor wykonuje tylko jedno zadanie w danej chwili. Dzięki technice tzw. "quasi-równoległości" [56] użytkownik ma wrażenie, że zadania te są realizowane jednocześnie, ponieważ procesor przełącza się między nimi w bardzo krótkich odstępach czasu. Takie podejście jest kluczowe w aplikacjach interaktywnych, takich jak gry komputerowe, aplikacje mobilne czy serwisy internetowe, które muszą błyskawicznie reagować na różne zdarzenia (np. żądania użytkowników, kliknięcia czy komunikaty sieciowe) bez zauważalnych opóźnień [36].

Programowanie współbieżne pozwala na lepsze zarządzanie zadaniami w aplikacjach, które muszą obsługiwać wiele operacji jednocześnie, choć nie zawsze są to zadania wymagające intensywnych obliczeń. Przykładami takich aplikacji mogą być systemy obsługi żądań użytkowników na serwerach, aplikacje multimedialne czy interfejsy graficzne [24].

1.1.1. Mechanizmy realizujące współbieżność

Współbieżność różni się od programowania równoległego tym, że nie wymaga fizycznej wielordzeniowości procesora. Nawet na jednordzeniowym procesorze możliwe jest uzyskanie współbieżności, ponieważ procesor może w bardzo szybki sposób przełączać się między różnymi zadaniami. Tego rodzaju przełączanie nazywane jest "wirtualnym przełączaniem" i odbywa się na poziomie systemu operacyjnego, który odpowiedzialny jest za przeprowadzanie tego procesu w sposób niewidoczny dla użytkownika. Dzięki tej technice użytkownik nie zauważa, że procesor w danym momencie wykonuje tylko jedno zadanie, mimo że wiele z nich jest obsługiwanych "po kolei" w bardzo krótkich cyklach [72, 25].

Kolejnym istotnym mechanizmem, który wspiera współbieżność, jest program szeregujący bądź też planista (ang. *scheduler*). Jest odpowiedzialny za zarządzanie dostępem do procesora i przydzielanie zasobów obliczeniowych poszczególnym zadaniom. Dzięki zaawansowanym algorytmom harmonogramowania, system operacyjny decyduje, które zadanie ma być wykonane w danym czasie, jak długo ma trwać jego wykonanie, oraz kiedy procesor ma przełączyć się na inne zadanie. Planista zadań może być dostosowywany w zależności od wymagań aplikacji, co pozwala na osiągnięcie optymalnej wydajności i minimalizację opóźnień [72].

1.1.2. Zastosowania programowania współbieżnego

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

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

1.1.3. Zalety programowania współbieżnego

Główne zalety stosowania programowania współbieżnego w aplikacjach to [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.
- Skalowalność - aplikacje wykorzystujące współbieżność mogą być łatwiej skalowane na wiele rdzeni procesora lub urządzeń, dzięki czemu mogą obsługiwać większą liczbę użytkowników lub większe ilości danych.

1.1.4. 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.
- Trudności w debugowaniu - aplikacje współbieżne są trudniejsze do debugowania, ponieważ błędy mogą występować sporadycznie i w zależności od kolejności przełączania wątków, co utrudnia ich wykrywanie i naprawę.

1.2. Programowanie równoległe

Programowanie równoległe to technika, która umożliwia równoczesne wykonywanie wielu zadań, wykorzystując wiele jednostek obliczeniowych. W tym podejściu zadania są fizycznie realizowane jednocześnie na różnych rdzeniach procesora lub innych jednostkach przetwarzających. Programowanie równoległe jest szczególnie użyteczne w aplikacjach wymagających znacznej mocy obliczeniowej, takich jak obliczenia w dziedzinie uczenia maszynowego, symulacje naukowe, przetwarzanie dużych zbiorów danych, rendering grafiki oraz aplikacje o wysokiej wydajności. Dzięki tej technice możliwe jest zredukowanie czasu wykonywania obliczeń, które w tradycyjnym, sekwencyjnym modelu zajmowałyby znacznie więcej czasu [36, 15].

1.2.1. 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 Flynna, stanowi klasyczny system kategoryzacji architektur komputerowych pod kątem stopnia i rodzaju równoległości przetwarzania. Jej głównym założeniem jest klasyfikacja systemów obliczeniowych na podstawie liczby równocześnie 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 jednoprocesorowych.
- 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 konstruktem 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 klastry 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 zrównoleglenia 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.

1.2.2. Zastosowanie programowania równoległego

Programowanie równoległe znajduje szerokie zastosowanie w różnych dziedzinach, w których wymagana jest ogromna moc obliczeniowa oraz szybkie przetwarzanie dużych zbiorów danych. Jednymi z kilku najczęściej wykorzystywanych zastosowań są [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 rozdzielanie zadań przetwarzania danych na wiele maszyn.
- Rendering grafiki 3D - w grach komputerowych, filmach animowanych i inżynierii wizualnej, gdzie renderowanie obrazów i animacji wymaga intensywnych obliczeń graficznych. Programowanie równoległe umożliwia szybkie generowanie wysokiej jakości obrazów przez równoczesne przetwarzanie wielu elementów obrazu.

1.2.3. 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.
- Skalowalność - aplikacje równoległe mogą być skalowane w zależności od dostępnych zasobów obliczeniowych, umożliwiając zwiększenie wydajności przy rozwoju systemu.

1.2.4. 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 (race conditions), które mogą prowadzić do nieprzewidywalnych wyników.
- Problemy komunikacyjne - w systemach rozproszonych, komunikacja między jednostkami przetwarzającymi może stać się wąskim gardłem, obniżającym wydajność systemu. W takich przypadkach konieczne jest optymalizowanie przepływu danych i unikanie zbędnych operacji komunikacyjnych.
- Narzut związany z równoległością - chociaż programowanie równoległe przyspiesza obliczenia, wprowadza również dodatkowy narzut związany z przełączaniem kontekstu między zadaniami, synchronizacją wątków i komunikacją. W przypadku niewielkich zadań, zysk z równoległości może nie przewyższać kosztów narzutu.

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

- OpenMP (Open Multi-Processing) - biblioteka dla języków C, C++ i Fortran, która umożliwia programowanie równoległe w modelu pamięci współdzielonej. [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

Spis rysunków

Spis tabel

Spis listingów

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). [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.

[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.

[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. 06 2021.

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

[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.

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

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

[26] ByteByteGo. EP108: How do we design a secure system? — blog.bytebytego.com. `https://blog.bytebytego.com/p/ep108-how-do-we-design-a-secure-system?utm_campaign=post&utm_medium=web`. [Dostęp 09-03-2025].

[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.

[29] I. Corporation. Best practices for real-time optimizations with the 12th generation intel® core™ processors. Technical paper, Intel Corporation, 2022. [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-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.

[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, Manitainable, 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. Accessed: 2025-06-12.

[39] H. Heyman, L. Brandefelt. A comparison of performance & implementation complexity of multithreaded applications in rust, java and c++, 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, 2023.

[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.

[44] B. Köpcke, S. Gorlatch, M. Steuwer. Descend: A safe gpu systems programming language. *Proceedings of the ACM on Programming Languages*, 8, 2024.

[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.

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

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

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

[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, 2013.

[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.

[60] A. Saligrama, A. Shen, J. Gjengset. A practical analysis of rust's concurrency story. *arXiv preprint arXiv:1904.12210*, Kwi. 2019. [Dostęp: 2025-06-12].

[61] I. M. Sesay. The critical role of parallelism and multicore processors in advancing high-level programming practices. *Global Science Journal*, Wrze. 2024. Accessed via ResearchGate.

[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.

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

[64] T. Silva, J. Bispo, T. Carvalho. Foundations for a rust-like borrow checker for c. strony 155 – 165, 2024.

[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.

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

[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: 2025-06-12.

[69] T. Vandervelden, R. De Smet, D. Deac, K. Steenhaut, A. Braeken. Overview of embedded rust operating systems and frameworks. *Sensors*, 24(17), 2024.

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

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

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