Różne rozwiązania problemu producenta i konsumenta

Kacper Kozubowski
Grupa laboratoryjna: wt_13:15

Contents

1 Temat ćwiczenia

Tematem ćwiczenia jest porównanie różnych implementacji rozwiązania problemu producenta i konsumenta z zastosowaniem mechanizmów synchronizacji w języku Java. Celem jest zbadanie wydajności i efektywności dwóch rozwiązań niezagładzających oraz jednego, mogącego doprowadzić do zagłodzenia wątków.

Obiektem badań są implementacje z synchronizacją za pomocą:

1. Dwóch zmiennych warunkowych (zagładzające)
2. Czterech zmiennych warunkowych (potencjalnie niezagładzające)
3. Zagnieżdżonych locków (niezagładzające)

2 Opis implementacji

Implementacja wszystkich trzech rozwiązań opiera się o wykorzystanie trzech klas:

- `Buffer` – odpowiada za synchronizację dostępu do wspólnego bufora, z którego korzystają producenci i konsumenci. Zawiera metody `produce()` oraz `consume()`, które kontrolują dodawanie i pobieranie elementów z bufora.
- `Producer` – reprezentuje wątek producenta. Producent korzysta z metody `produce()` klasy `Buffer`, aby dodawać elementy do bufora.
- `Consumer` – reprezentuje wątek konsumenta. Konsument korzysta z metody `consume()` klasy `Buffer`, aby pobierać elementy z bufora.

Między różnymi implementacjami, klasy `Producer` i `Consumer` zostają niezmienne, a modyfikacji ulega tylko sposób synchronizacji dostępu do wspólnego zasobu w klasie `Buffer`. Kluczowe jest również to, że zarówno wątki producentów, jak i konsumentów, dodają i odejmują z bufora losową liczbę elementów.

2.1 Pierwsze rozwiązanie: 2 zmienne warunkowe

Pierwsze rozwiązanie wykorzystuje dwie zmienne warunkowe oraz jeden obiekt klasy ReentrantLock w celu zapewnienia synchronizacji.

- **Główna blokada (`ReentrantLock`)** – używana do synchronizacji dostępu do wspólnego bufora.
- **Zmienna warunkowa `notFull`** – pozwala producentom czekać gdy bufor jest pełny.
- **Zmienna warunkowa `notEmpty`** – pozwala konsumentom czekać gdy bufor jest pusty.

Opis działania: Producent, przed dodaniem elementu do bufora, sprawdza, czy bufor jest pełny. Jeśli jest, czeka na zwolnienie miejsca przy użyciu zmiennej warunkowej. Konsument sprawdza, czy w buforze są dostępne elementy, a jeśli ich brakuje, czeka na sygnał od producenta.

2.2 Drugie rozwiązanie: 4 zmienne warunkowe

Drugie rozwiązanie wykorzystuje cztery zmienne warunkowe, dwie zmienne boolean oraz jeden obiekt klasy ReentrantLock w celu zapewnienia synchronizacji.

- **Główna blokada (`ReentrantLock`)** – używana do synchronizacji dostępu do wspólnego bufora.
- **Zmienna warunkowa `firstProducer`** – pozwala pierwszemu producentowi czekać, gdy bufor jest pełny.
- **Zmienna warunkowa `otherProducers`** – kontroluje czekanie pozostałych producentów, gdy pierwszy producent czeka.
- **Zmienna warunkowa `firstConsumer`** – pozwala pierwszemu konsumentowi czekać, gdy bufor jest pusty.
- **Zmienna warunkowa `otherConsumers`** – kontroluje czekanie pozostałych konsumentów, gdy pierwszy konsument czeka.
- **Zmienna boolean `firstProdWaiting`** – przechowuje informację o tym czy pierwszy producent wciąż czeka.
- **Zmienna boolean `firstConsWaiting`** – przechowuje informację o tym czy pierwszy konsument wciąż czeka.

Opis działania: Producent, przed dodaniem elementu do bufora, sprawdza czy bufor jest pełny. Jeśli jest, czeka na zwolnienie miejsca przy użyciu warunku `firstProducer`. Jeśli przed obsłużeniem pierwszego producenta, przyjdą kolejni, to będą oni czekać na warunku `otherProducers`. Konsument sprawdza, czy w buforze są dostępne elementy, a jeśli ich brakuje, czeka na warunku `firstConsumer`. Jeśli przed obsłużeniem pierwszego konsumenta, przyjdą kolejni, to będą oni czekać na warunku `otherConsumers`.

2.3 Trzecie rozwiązanie: zagnieżdżone blokady

Trzecie rozwiązania wykorzystuje dwie zmienne warunkowe oraz trzy obiekty klasy ReentrantLock w celu zapewnienia synchronizacji.

- **Główna blokada (`ReentrantLock`)** – używana do synchronizacji dostępu do wspólnego bufora.
- **Blokada Producenta (`ReentrantLock`)** – zapewnia dodatkową warstwę synchronizacji dla wątków producentów.
- **Blokada Konsumenta (`ReentrantLock`)** – zapewnia dodatkową warstwę synchronizacji dla wątków konsumentów.
- **Zmienna warunkowa `notFull`** – pozwala producentom czekać gdy bufor jest pełny.
- **Zmienna warunkowa `notEmpty`** – pozwala konsumentom czekać gdy bufor jest pusty.

Opis działania: Producent wchodząc do bufora, najpierw przechodzi przez blokadę producentów a potem przez główną blokadę. Przed dodaniem elementu do bufora sprawdza czy bufor jest pełny. Jeśli jest, czeka na zwolnienie miejsca przy użyciu zmiennej warunkowej `notFull`. Jeśli przed obsłużeniem pierwszego producenta, przyjdą kolejni, to będą oni czekać przed blokadą producentów. Konsumenci postępują analogicznie.

3 Przeprowadzenie eksperymentów

Przeprowadzone pomiary mają na celu porównanie wydajności poszczególnych implementacji oraz przedstawienie różnic w działaniu badanych metod synchronizacji.

3.1 Opis środowiska testowego

- Procesor: Intel Core i5-1135G7 (8 rdzeni, 2.4 GHz)
- RAM: 16 GB
- System operacyjny: Windows 11
- Oprogramowanie: Java, środowisko IntelliJ IDEA

3.2 Opis przeprowadzanych pomiarów

Problem producenta i konsumenta w rozważanej wersji może posiadać znaczną liczbę parametrów, co zdecydowanie utrudnia przeprowadzenie odpowiednich eksperymentów. Parametry obecne w mojej implementacji to:

- `N` – Liczba producentów.
- `M` – Liczba konsumentów.
- `bufferCapacity` – Wielkość bufora.
- `maxProductions` – Maksymalna wielkość produkowanej porcji.
- `maxConsumptions` – Maksymalna wielkość konsumowanej porcji.

Istnieje więc bardzo duża liczba kombinacji parametrów, dających różne wyniki. W celu przeprowadzenia rzetelnych badań, wykonałem testy na 50 losowych konfiguracjach i przeanalizowałem uzyskane dane. Następnie, wyznaczyłem kilka ciekawych przypadków i przeprowadziłem na nich dodatkowe eksperymenty a ich wyniki opisałem w poniższych sekcjach.

3.3 Opis metryk

Dla każdego testu wyznaczałem limit operacji produkcji/konsumpcji. Operacje te były zliczane w buforze. Po osiągnięciu limitu, test kończył się i notowany był czas wykonania jego wykonania. Na wykresach liniowych oś `x` przedstawia limit operacji a oś `y` zmierzony czas w milisekundach.

Dodatkowo, każdy wątek, indywidualnie zliczał wykonywane przez siebie operacje. Wyniki tych pomiarów zostały przedstawione na wykresach słupkowych gdzie na osi `x` znajduje się ID wątku, a na osi `y` liczba wykonanych przez niego operacji. Słupki o kolorze czerwonym oznaczają wątki producentów a niebieskie, wątki konsumentów.

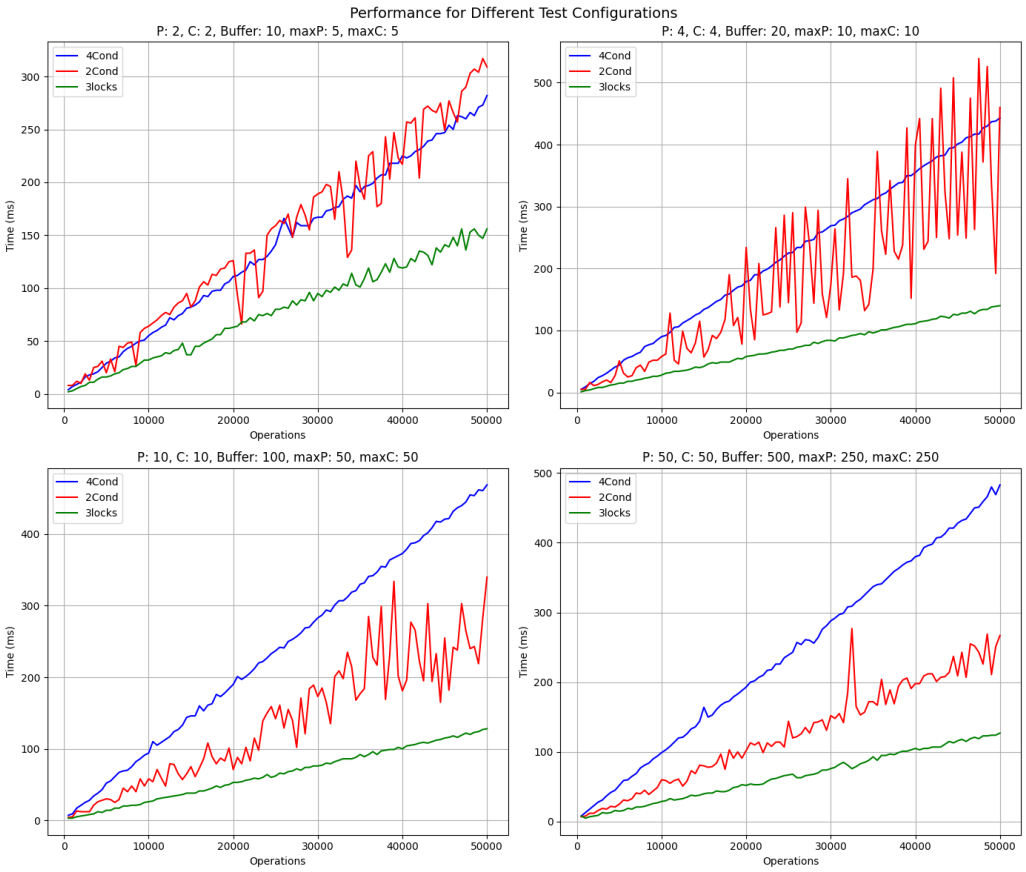
Opis oznaczeń konfiguracji na wykresach:

- `2Cond` – Rozwiązanie na dwóch zmiennych warunkowych (kolor czerwony).
- `4Cond` – Rozwiązanie na czterech zmiennych warunkowych (kolor niebieski).
- `3Locks` – Rozwiązanie na zagnieżdżonych blokadach (kolor zielony).
- `P` – Liczba producentów.
- `C` – Liczba konsumentów.
- `Buffer` – Rozmiar bufora.
- `maxP` – Maksymalna liczba produkcji.
- `maxC` – Maksymalna liczba konsumpcji.
- `Operations` – Liczba wykonanych operacji.

4 Wyniki testów

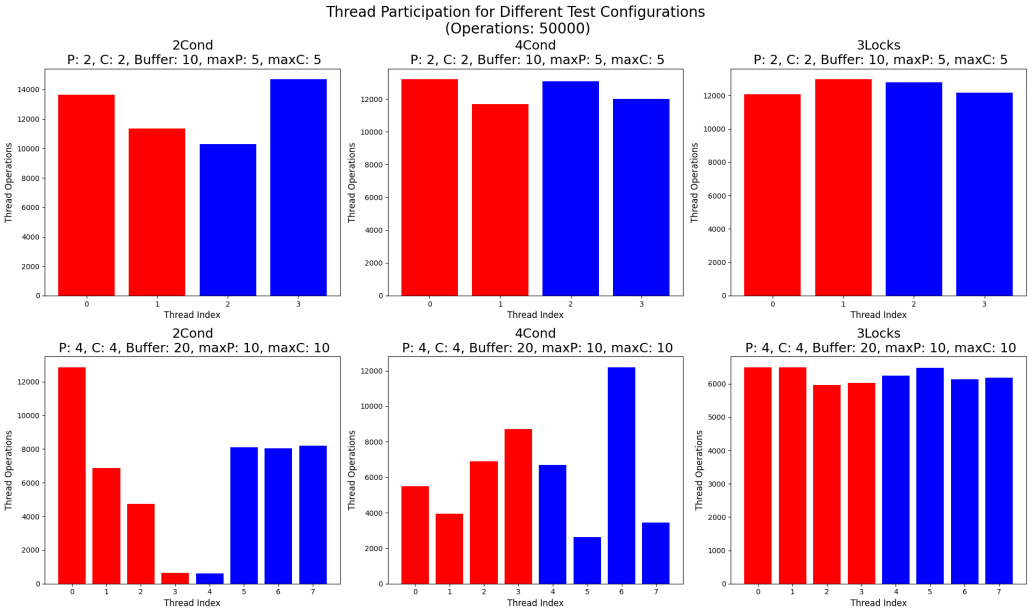
4.1 Eksperyment pierwszy

Celem tego eksperymentu było zbadanie sytuacji gdy maksymalna liczba produkcji i konsumpcji jest taka sama i równa się połowie wielkości bufora.



Rysunek 1: Porównanie wydajności

Jak widać na rysunku **1** rozwiązanie na zagnieżdżonych lockach charakteryzuje się najlepszą wydajnością we wszystkich czterech przypadkach. Pozostałe dwa rozwiązania mają zbliżone wyniki w sytuacjach gdy liczba obsługiwanych wątków nie przekracza liczby rdzeni procesora. W sytuacji gdy wątków jest więcej, rozwiązanie na czterech zmiennych jest widocznie najgorsze. Rozwiązanie na dwóch wątkach wykazuje największą niestabilność we wszystkich czterech przypadkach.



Rysunek 2: Porównanie udziału wątków

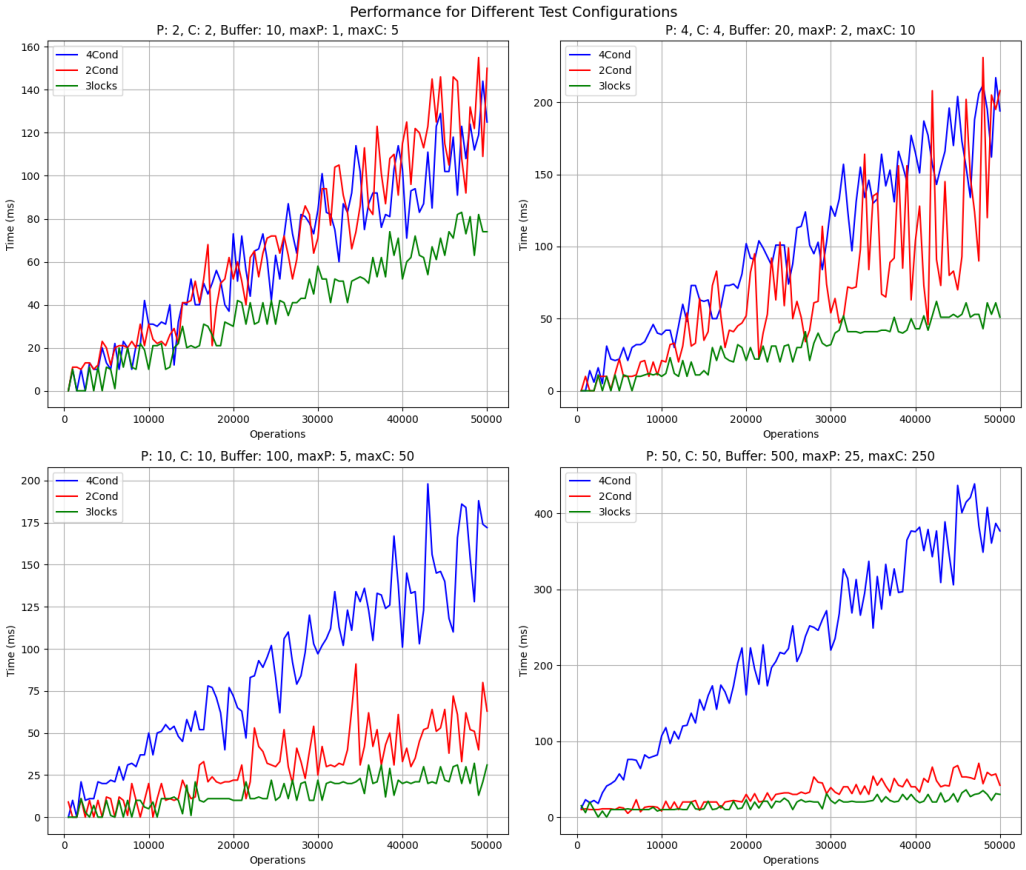
Pomiary przedstawione na rysunku **2** zostały przeprowadzone tylko dla konfiguracji w których liczbą wątków nie przekraczała liczby rdzeni procesora, ponieważ takie wyniki pozwalają nam, z większą pewnością, zidentyfikować zagładzanie wątków.

Wykresy dla konfiguracji z czterema wątkami nie wykazują znacznego zagładzania. Jedynie pierwszy wykres dla tej konfiguracji może sugerować pewną przewagę wątków 1 i 4 oraz mniejszą aktywność pozostałych dwóch.

Wykresy dla konfiguracji z ośmioma wątkami pokazują już wyraźnie, że rozwiązanie na dwóch zmiennych warunkowych mocno zagładza niektóre wątki. Rozwiązanie na czterech zmiennych również przejawia skłonność do częściowego zagładzania. Natomiast w rozwiązaniu na zagnieżdżonych lockach wszystkie wątki pracują równomiernie.

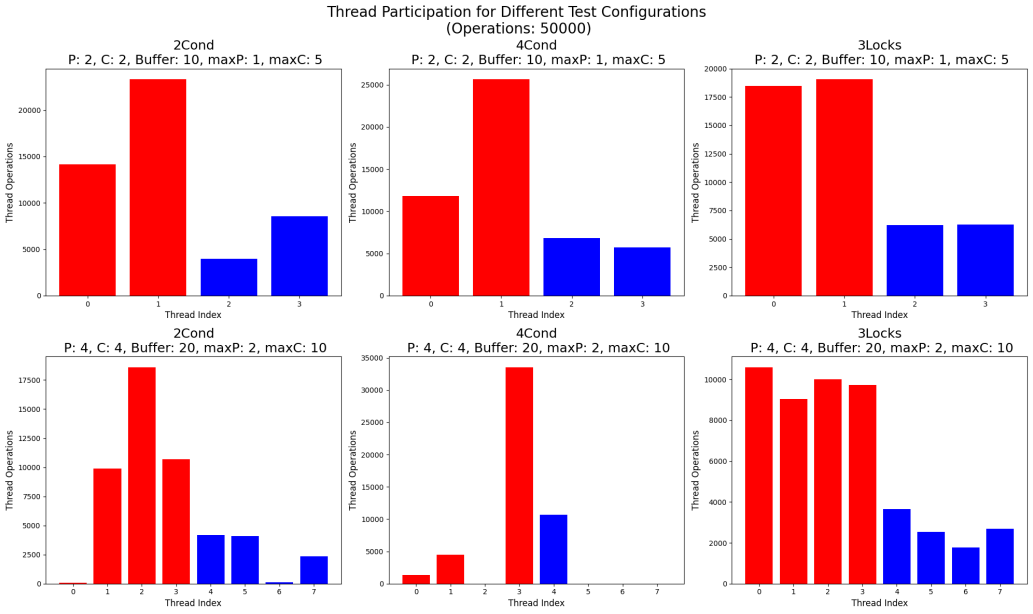
4.2 Eksperyment drugi

Celem drugiego eksperymentu było zbadanie sytuacji w której jedna ze stron (producenci lub konsumenci) może produkować/konsumować znacznie większe porcje od drugiej.



Rysunek 3: Porównanie wydajności

Jak widać na rysunku **3** zachowanie badanych rozwiązań znacząco różni się od tego, które obserwowaliśmy w pierwszym eksperymencie. Otrzymane wyniki wskazują, że duża dysproporcja produkcji i konsumpcji ma negatywny wpływ na stabilność wszystkich trzech implementacji. Możemy również zaobserwować, że w porównaniu wydajności zachowany jest ten sam porządek, tj. `3locks` jest najszybszy, kolejny jest `2Cond` a ostatni `4Cond`. Jednak interesujące jest to, jak wyraźnie wolniejszy jest `4Cond` w ostatnim teście.



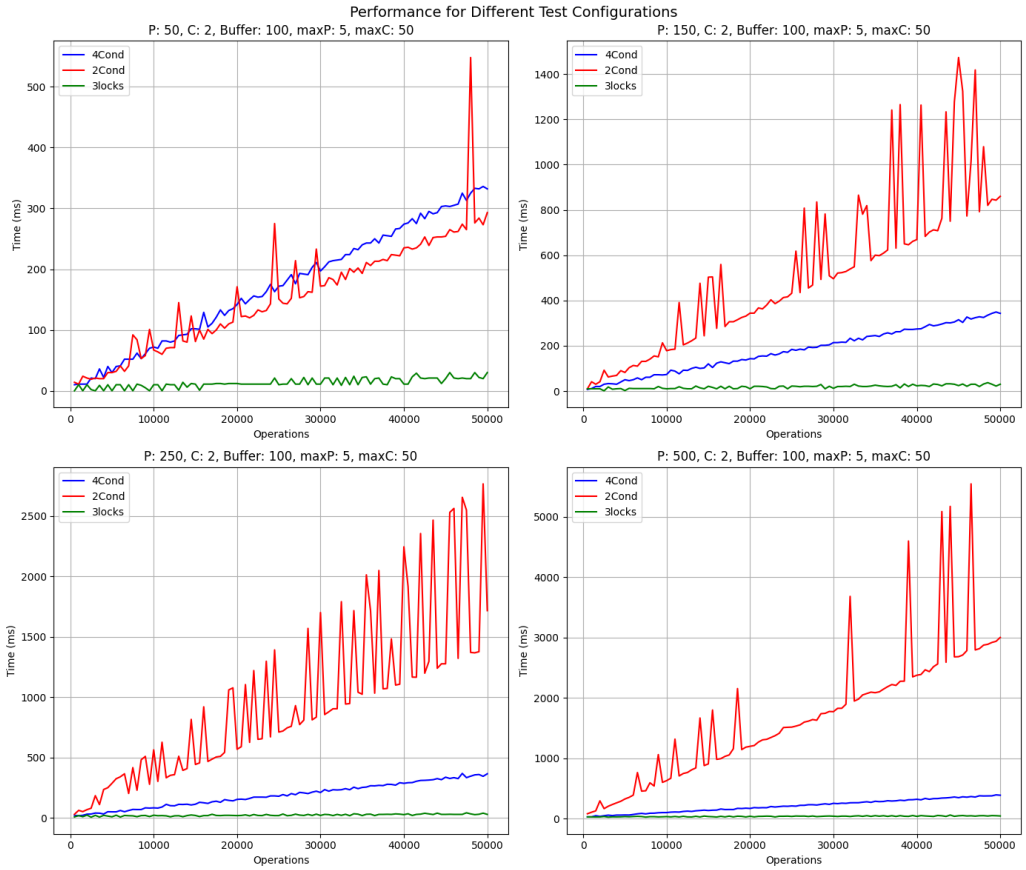
Rysunek 4: Porównanie udziału wątków

Tutaj również otrzymujemy wyniki zdecydowanie różniące się od tych z pierwszego eksperymentu. Należy jednak pamiętać, że wątki producentów produkują tutaj znacznie mniej niż konsumenci konsumują, więc takie dysproporcje w wykonanych operacjach są jak najbardziej spodziewane.

Mimo wszystko, w tym eksperymencie również widoczne jest zagładzanie wątków. Jest ono szczególnie wyraźne dla konfiguracji z ośmioma wątkami, a konkretnie dla rozwiązania `4Cond`.

4.3 Eksperyment trzeci

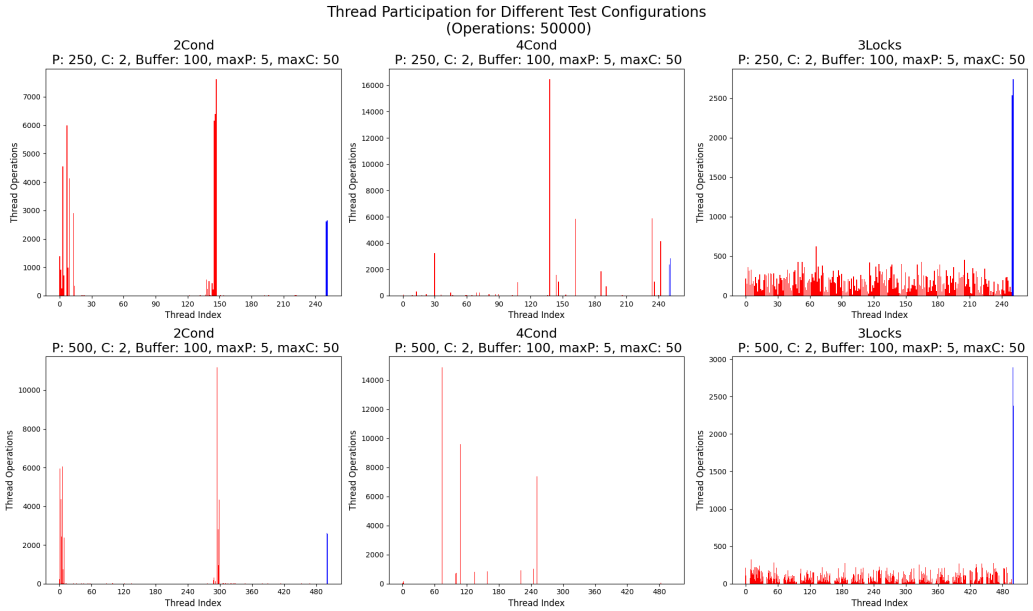
Celem trzeciego eksperymentu było zbadanie konfiguracji, w których występuje znacznie więcej wątków jednego rodzaju oraz dysproporcja produkcji i konsumpcji odwrotna do proporcji wątków (te, których jest mało produkują lub konsumują dużo).



Rysunek 5: Porównanie wydajności

Patrząc na pierwszy wykres na rysunku **5** nie zauważamy nic nadzwyczajnego, jednak już na drugim widzimy, że nagle rozwiązanie `2Cond` stało się najwolniejsze ze wszystkich i zaczeło zachowywać się nieco chaotycznie. Na trzecim wykresie, rozwiązanie `2Cond` przyjmuje bardzo niestabilną, lecz w pewnym sensie regularną formę. Co ciekawe zdaje się, że jest to wynik stosunkowo powtarzalny, ponieważ dla tych konkretnych parametrów osiągnąłem podobny wynik już kilkukrotnie. Na czwartym wykresie wspomniana ciekawa struktura znika przez co przypomina bardziej wykres drugi.

Ważnym spostrzeżeniem jest to, że rozwiązanie `4Cond` może być szybsze od `2Cond` dla niektórych konfiguracji.



Rysunek 6: Porównanie udziału wątków

Wyniki tych pomiarów nie są aż tak miarodajne, ponieważ przy takiej ilości wątków dochodzą inne niekontrolowane czynniki, mogące mieć wpływ na to, który wątek otrzyma zasób. Mimo wszystko, przy tak jednoznacznych wynikach, możemy w miarę bezpiecznie założyć, że tutaj również dochodzi do zagłodzenia wątków. Możemy również, bardzo wyraźnie zaobserwować tu różnicę w działaniu rozwiązania na zagnieżdżonych lockach względem pozostałych dwóch.

5 Wnioski

Na podstawie przeprowadzonych eksperymentów można wyciągnąć następujące wnioski:

1. **Wydajność różnych rozwiązań**: Rozwiązanie oparte na zagnieżdżonych blokadach (`3Locks`) wykazało się najlepszą wydajnością we wszystkich przypadkach, zarówno pod względem czasu wykonania, jak i równomiernego rozkładu operacji między wątkami. Jest to spowodowane zastosowaną metodą synchronizacji, zapewniającą odpowiednią kolejność dostępu wątków do dzielonego zasobu. Oznacza to, że wątki nie mogą "wpychać" się przed inne, które teoretycznie powinny wykonywać swoje operacje jako pierwsze.

   W większości przypadków rozwiązanie `4Cond` było zdecydowanie najwolniejsze. Główne różnice między tą implementacją, a rozwiązaniem `2Cond` to dodatkowe dwie zmienne warunkowe i dwie pętle while, co sugeruje, że obsługa tych właśnie elementów jest bardzo obciążająca.

2. **Zagładzanie wątków**: We wszystkich przeprowadzonych eksperymentach udało się zaobserwować, w mniejszym lub większym stopniu, zagładzanie wątków w rozwiązaniach na dwóch i na czterech zmiennych warunkowych. Z tego powodu, implementacje te powodują marnowanie zasobów komputera.

3. **Stabilność**: Rozwiązania `4Cond` i `3Locks`, mimo swojej bardziej zaawansowanej synchronizacji w porównaniu do `2Cond`, wykazały niestabilność w specyficznych konfiguracjach. Szczególnie widoczne było to przy dużej różnicy w liczbie maksymalnych produkcji i konsumpcji.

4. **Skalowalność**: Implementacja `3Locks` wykazała się największą skalowalnością, co sugeruje, że może być najlepszym wyborem w sytuacjach wymagających wysokiej wydajności i dużej liczby wątków.

5. **Zastosowanie praktyczne**: W praktycznych zastosowaniach, gdzie wymagane jest zapewnienie wysokiej wydajności przy jednoczesnym minimalizowaniu zagładzania, jedynym rozsądnym wyborem zdaje się być rozwiązanie z zagnieżdżonymi blokadami (`3Locks`). Pozostałe dwie metody nie cechują się znacznie prostszą implementacją, ani żadnymi innymi cechami usprawiedliwiającymi ich zastosowanie nawet w prostych problemach.