Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Sławomir Sadziak
Nr albumu: 201091

Równoważenie obciążenia w systemie z rozproszoną stertą obiektów

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dr Janiny Mincer-Daszkiewicz
Instytut Informatyki

Wrzesień 2006

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora pracy

Streszczenie

Równoważenie obciążenia może się odbywać poprzez przenoszenie obliczeń z przeciążonych komputerów na mniej przeciążone. Algorytmy równoważące obciążenie rzadko biorą pod uwagę fakt, że obliczenia współdzielą między sobą dane i przeniesienie części obliczeń na inny komputer bez przeniesienia danych, może skutkować koniecznością przesyłania tych danych przez sieć komputerową. W ramach pracy opracowano algorytm, który równoważy obciążenie biorąc pod uwagę intensywność współdzielenia danych przez obliczenia. Algorytm zastosowano do przenoszenia wątków języka Java, które dzielą między sobą obiekty. Implementację algorytmu zrealizowano w rozproszonej wirtualnej maszynie języka Java o nazwie JESSICA2, która pozwala na przenoszenie wątków i implementuje wspólną dla wszystkich wątków rozproszoną stertę obiektów.

Słowa kluczowe

równoważenie obciążenia, rozproszona pamięć dzielona, Java, wątek, współdzielony obiekt, klaster, JESSICA2

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

C. Computer Systems Organization
C.2 Computer-communication networks
C.2.4 Distributed Systems/Distributed Applications

Tytuł pracy w języku angielskim

Load balancing in a system with a distributed object heap

Pracę dedykuję mojej żonie.

Spis treści

Rozdział 1

Wprowadzenie

W trakcie działania systemów uruchomionych na wielu komputerach często zdarzają się sytuacje, gdy jeden z komputerów staje się bardziej przeciążony, podczas gdy pozostałe są mało obciążone lub nie wykonują żadnych obliczeń. Równoważenie obciążenia [Tan95] pozwala zwiększyć wydajność w systemach składających się z wielu komputerów. Polega ono na przeniesieniu części obliczeń z jednej maszyny na drugą w trakcie działania systemu.

Strategie równoważenia obciążenia w systemach wielokomputerowych są dobrze zbadane i opisane. Wykonano wiele badań eksperymentalnych, których celem było porównanie zachowania różnych strategii [Zhu95]. Lepsze strategie pozwalają zrównoważyć obciążenie, a tym samym przyspieszyć działanie całego systemu.

Nie ma jednak strategii idealnej, która by się dobrze nadawała dla każdego przypadku, ponieważ istotną rolę odgrywa schemat wykonywanych obliczeń. Ta sama strategia może się bowiem inaczej zachowywać, gdy wykonywane są procesy o takim samym czasie życia, a inaczej gdy procesy mają różny czas życia albo robią przerwy w obliczeniach i później wznawiają swoje działanie.

Jest jeszcze bardziej złożony schemat obliczeń – gdy wątki współdzielą dane. Wówczas istnieje konieczność przesyłania danych pomiędzy komputerami, jeżeli wątki znajdują się na różnych komputerach. Strategie równoważenia obciążenia dla takiego schematu powinny brać pod uwagę fakt współdzielenia danych, gdyż przeniesienie wątku na maszynę mniej obciążoną może pogorszyć wydajność systemu. Stać się tak może w sytuacji, gdy wątek po przeniesieniu będzie musiał pobierać i zapisywać duże porcje danych przez sieć, a przed przeniesieniem nie musiał gdyż dane znajdowały się na tym samym komputerze.

Zainteresowany zagadnieniem równoważenia obciążenia wątków, postanowiłem zbadać w jakim stopniu można poprawić jego efektywność biorąc pod uwagę współdzielenie danych. Nie ma zbyt wielu prac na ten temat. Po analizie dostępnej literatury postawiłem sobie za cel opracowanie takiej właśnie strategii równoważenia obciążenia oraz eksperymentalne zbadanie jej użyteczności. Taka strategia powstała i została sprawdzona eksperymentalnie.

Do przeprowadzenia eksperymentu potrzebne było środowisko, które pozwalałoby na tworzenie wątków, które powinny mieć możliwość wymiany między sobą danych, oraz przenoszenie wątku z jednego komputera na inny w trakcie jego działania.

Do tego celu wybrane zostało środowisko rozproszonej maszyny wirtualnej języka Java. Zadaniem maszyny wirtualnej jest wykonywanie bajtkodu. Programy w języku Java nie są kompilowane do kodu maszynowego, tylko do abstrakcyjnego zestawu prostych instrukcji, który można w łatwy sposób przetłumaczyć na kod maszynowy. Ten abstrakcyjny zestaw prostych instrukcji to właśnie bajtkod. Wśród instrukcji bajtkodu jest wiele odpowiedników instrukcji procesorów x86, np. matematyczne add, inc, czy też logiczne and, or, xor.

To co odróżnia rozproszone maszyny wirtualne od zwykłych maszyn wirtualnych, to uruchamianie bajtkodu w sieci komputerowej. W sytuacji gdy program skompilowany do bajtkodu korzysta z wielu wątków języka Java, maszyna wirtualna może przyspieszyć działanie programu poprzez przenoszenie wątków na różne komputery.

Wątki w Javie mogą w sposób naturalny wymieniać między sobą dane poprzez współdzielenie obiektów (posiadanie referencji do tego samego obiektu). Z tego powodu rozproszone maszyny wirtualne są dobrym kandydatem do przeprowadzenia planowanego eksperymentu: wątki pozwalają na zrównoleglenie obliczeń, a obiekty na wymianę danych. Dodatkowo rozproszona maszyna wirtualna jest w stanie uruchomić dowolny program napisany w języku Java, zatem nie ograniczamy się do szczególnych wzorców działania.

Istnieje kilka rozproszonych maszyn wirtualnych języka Java, m.in.: *dJVM* [Zig03], *Cluster Virtual Machine for Java* [Ari99], *Jackal* [Vel], *JESSICA2* [Zhu04]. Spośród tych maszyn jedynie JESSICA2 potrafi przenosić wątki już po ich uruchomieniu, co jest wykorzystane w algorytmie powstałym w ramach tej pracy. Dodatkowo JESSICA2 jest projektem typu *open source*, więc nie istnieją przeciwwskazania, żeby pisać algorytm na podstawie tego projektu.

W pierwszym rozdziale tej pracy przedstawiam podstawowe pojęcia związane z projektem JESSICA2 takie jak: język Java, maszyna wirtualna języka Java oraz systemy rozproszone. Zaznajomienie się z tymi pojęciami będzie przydatne w zrozumieniu omawianych później zagadnień.

W drugim rozdziale opisuję projekt JESSICA2, który jest implementacją rozproszonej maszyny wirtualnej Javy. W rozdziale są przedstawione rozwiązania użyte przy budowie tej maszyny wirtualnej, mające wpływ na konstrukcję algorytmu równoważenia obciążenia.

Kolejny rozdział jest przeglądem istniejących strategii równoważenia obciążenia.

Rozdział czwarty zawiera propozycję rozwiązania problemu równoważenia obciążenia, gdy wątki współdzielą dane, zaprojektowaną dla rozproszonej maszyny wirtualnej JESSICA2.

W rozdziale piątym znajdują się szczegóły implementacji zastosowanego rozwiązania.

Rozdział szósty jest opisem przeprowadzonych testów wydajnościowych.

Pracę kończy podsumowanie.

W dodatku A zamieszczam opis zawartości płyty CD dołączonej do pracy.

Rozdział 2

Podstawowe pojęcia

Zebrany w tym rozdziale zbiór pojęć dotyczy głównie technologii Java, ponieważ strategia równoważenia obciążenia, opracowana w ramach tej pracy, odbywa się w rozproszonej maszynie wirtualnej Javy o nazwie JESSICA2. Najpierw omawiam równoważenie obciążenia, potem definiuję klaster, na którym działa JESSICA2, a w dalszej kolejności przedstawiam krótkie wprowadzenie do Javy.

2.1. Równoważenie obciążenia

Równoważenie obciążenia [Tan95] polega na przeniesieniu części obliczeń z jednego komputera na inny, najlepiej na mający więcej dostępnych zasobów. Przeniesienie ma na celu zmniejszenie obciążenia jednego z komputerów, a przez to poprawienie wydajności całego systemu.

Przeniesienie części obliczeń może się odbyć między innymi przez:

- przeniesienie (migrację) wątku z obciążonego komputera;
- ograniczenie powstawania nowych wątków na przeciążonym komputerze, co w przypadku zakończenia istniejących, zmniejszy rozmiar obliczeń;
- przeniesienie danych, do których odwołuje się wątek z innego komputera, na ten komputer. Może to zredukować liczbę żądań odczytu/zapisu danych przesyłanych przez sieć w przypadku, gdy dane po przeniesieniu znajdą się na tym samym komputerze co wątek, który się do nich odwołuje. Taka redukcja liczby żądań przez sieć może skrócić czas obliczeń.

Opisany w rozdz. 5 algorytm przenosi wątki między komputerami, żeby przyspieszyć obliczenia.

2.2. Klaster

Przez klaster [Clu] rozumiemy zbiór komputerów współpracujących ze sobą tak ściśle, że pod pewnymi względami wydają się jakby były jednym komputerem. Komputer będący częścią klastra nazywany jest węzłem. Węzły klastra połączone są siecią komputerową.

W tej pracy przez klaster rozumiem zbiór komputerów, na których uruchamiana jest rozproszona maszyna wirtualna Javy. Z punktu widzenia programów uruchamianych na tej maszynie, fakt, że działają na rozproszonej maszynie wirtualnej, jest niezauważalny.

2.3. Język Java

Java [Java05] jest to obiektowy język programowania opracowany przez firmę Sun Microsystems na początku lat dziewięćdziesiątych ubiegłego stulecia. Programy napisane w Javie są kompilowane do bajtkodu i następnie uruchamiane na maszynie wirtualnej Javy. W ostatnich latach język Java zyskał ogromną popularność wśród języków programowania i obecnie posiada duży udział wśród technologii stosowanych do budowy aplikacji.

Główne cechy języka Java to:

- obiektowość, z wyłączeniem prostych typów nieobiektowych,
- składnia zaczerpnięta z C++,
- silna kontrola typów,
- automatyczne odśmiecanie obiektów,
- brak wielodziedziczenia,
- brak przeciążenia operatorów.

2.4. Maszyna wirtualna Javy

Maszyna wirtualna języka Java pozwala na uruchamianie bajtkodu, wygenerowanego przez kompilator języka Java. Ponieważ bajtkod jest pojęciem abstrakcyjnym, jest niezależny od platformy. Raz skompilowany do bajtkodu program w języku Java może być uruchamiany na różnych platformach. Do uruchomienia jest wymagane posiadanie maszyny wirtualnej języka Java na odpowiednią platformę. Istnieje wiele maszyn wirtualnych na różne platformy systemowe.

Maszyny wirtualne Javy dzielą się na interpretery i kompilatory Just-in-time (w skrócie: JIT). Interpretery czytają bajtkod instrukcja po instrukcji i wykonują te instrukcje zgodnie z ich znaczeniem. Takie rozwiązanie charakteryzuje się niską wydajnością. Z tego względu maszyny wirtualne, które interpretują bajtkod nie są używane do zadań wymagających dużej sprawności.

Nowoczesne maszyny wirtualne stosują technikę kompilacji JIT, która polega na tym, że bajtkod jest kompilowany w trakcie działania maszyny wirtualnej. Kompilacja odbywa się do kodu maszynowego. Tak skompilowany kod jest następnie uruchamiany. Przy stosowaniu tej techniki program języka Java jest dwukrotnie kompilowany: pierwszy raz do bajtkodu, drugi raz bajtkod jest kompilowany do kodu maszynowego. Stosowanie tej techniki pozwala na uzyskanie wysokiej wydajności programów języka Java przy zachowaniu przenośności.

2.5. Wątki w Javie

Strategia równoważenia obciążenia napisana w ramach tej pracy zmniejsza obciążenie węzłów klastra poprzez przenoszenie wątków języka Java.

Wątek w języku Java definiuje się przez odziedziczenie klasy `Thread` oraz zdefiniowanie metody `run`, która będzie zawierać instrukcje wykonywane przez wątek. Oto przykładowa definicja wątku:

```
class HelloThread extends Thread {
    public void run() {
        System.out.println("Hello");
    }
}
```

Aby uruchomić nowy wątek klasy `HelloThread`, należy stworzyć obiekt tej klasy, a następnie wywołać na tym obiekcie metodę `start`, która jest zdefiniowana w nadklasie `Thread`. Przykładowe wywołanie wątku:

```
HelloThread p = new HelloThread();
p.start();
```

Synchronizacja wątków w Javie odbywa się poprzez zdobywanie blokad na obiektach lub klasach języka Java. Każdy obiekt i klasa mają swoją blokadę. Tylko jeden wątek może uzyskać blokadę na danym obiekcie. Pozostałe wątki czekają na zwolnienie blokady w kolejce. Programista, piszący w języku Java, nie ma możliwości bezpośredniego zdobycia blokady, a jedynie pośrednio na jeden z następujących sposobów:

- wywołanie specjalnie oznaczonej metody obiektowej (`synchronized`) na obiekcie. Wykonywanie kodu tej metody oznacza zdobycie blokady na tym obiekcie;
- blok `synchronized(obj) { ... }`. Wejście do tego bloku oznacza zdobycie blokady na obiekcie `obj`;
- dla obiektu typu `Class` poprzez wywołanie metody klasowej oznaczonej jako `synchronized`. Wejście do tej metody oznacza zdobycie blokady na obiekcie typu `Class`.

2.6. Organizacja pamięci w Javie

Opisana w rozdz. 5 strategia równoważenia obciążenia przenosi wątki, żeby zmniejszyć obciążenie węzłów. Przy podejmowaniu decyzji opiera się na zgromadzonych danych o współdzieleniu pamięci przez te wątki. W tym podrozdziale opisuję organizację pamięci w Javie.

Obiekty języka Java przechowywane są w stercie. Obiekt jest tworzony na stercie instrukcją `new`, a usuwany jest ze sterty automatycznie, gdy nie jest już potrzebny, podczas tzw. odśmiecania pamięci.

Sterta składa się z pamięci głównej i pamięci roboczych wątków. Każdy wątek ma własną pamięć roboczą. Wszystkie wątki mogą korzystać z pamięci głównej.

Specyfikacja maszyny wirtualnej Java [Java99] określa zasady jakie musi spełnić sterta realizowana przez maszynę wirtualną. Zmienne w pamięci głównej i pamięci roboczej obowiązują następujące zasady:

- pomiędzy przypisaniem wartości przez wątek T zmiennej V i wykonaniem operacji odblokuj na blokadzie L, musi zostać wykonane przeniesienie do pamięci głównej wartości przypisania zmiennej V oraz wszystkich wartości przypisań dokonanych przez wątek;
- pomiędzy wykonaniem operacji zablokuj przez wątek T na blokadzie L, a odczytaniem wartości zmiennej V przez T, musi zostać wykonane przeniesienie danych z pamięci głównej do pamięci roboczej wątku.

Opisane zasady wydają się być oczywiste w przypadku, gdy mamy do czynienia z maszyną wirtualną działającą na jednym komputerze. Wydaje się nawet, że można by przeprowadzić optymalizację pracy maszyny wirtualnej, tak aby przeniesienia nie musiały być w pewnych przypadkach wykonywane, a wątki tak naprawdę odwoływały się bezpośrednio do pamięci głównej albo tylko do własnej pamięci roboczej.

W przypadku rozproszonej maszyny wirtualnej podane zasady są kluczowe do realizacji rozproszonej sterty obiektów, gdyż definiują momenty, w których musi być dokonywane uaktualnianie danych. W rozproszonej stercie obiektów uaktualnianie danych przez wątki znajdujące się na innym węźle niż dane, musi się bowiem odbywać przez sieć komputerową i lepiej, żeby nie było wykonywane zbyt często, a tylko tak często, jak to jest konieczne.

Rozdział 3

JESSICA2 – rozproszona maszyna wirtualna Javy

JESSICA2 [Jessica] to rozproszona maszyna wirtualna języka Java. W odróżnieniu od większości dostępnych maszyn wirtualnych języka Java, które są przeznaczone na komputery jednoprocesorowe lub wieloprocesorowe, JESSICA2 może działać na klastrze. Jest przy tym zgodna ze specyfikacją maszyny wirtualnej języka Java, opracowaną przez firmę Sun Microsystems. Realizuje między innymi przezroczyste (niewidoczne z punktu widzenia programisty) przenoszenie wątków pomiędzy komputerami, rozproszoną stertę i przekierowanie operacji wejścia/wyjścia.

JESSICA2 powstała jako rozszerzenie maszyny wirtualnej Kaffe [Kaffe06], będącej projektem typu *open source*. Kaffe ma zaimplementowany wspomniany w podrozdziale 2.4 mechanizm JIT.

3.1. Migracja wątków

Migracja wątków w JESSICA2 została dostosowana do mechanizmu kompilaci JIT. Podczas kompilacji JIT kod wątków jest tłumaczony do kodu maszynowego. Elementy uruchomieniowe, takie jak licznik rozkazów (ang. Program Counter, PC), stos wywołań metod, zawartość rejestrów są zależne od niskopoziomowej architektury. Te elementy tworzą tzw. surowy kontekst wątku (ang. Raw Thread Context, RTC). Kod maszynowy, skompilowany przez JIT, przy każdym wywołaniu metody tworzy rekord aktywacji (zwany też ramką) na stosie wywołań metod. Rekord aktywacji jest to środowisko wywołania metody, na które składa się:

- adres powrotu – adres pamięci, w którym znajduje się kod maszynowy programu, do którego należy wrócić gdy wołana metoda się zakończy,
- adres poprzedniej ramki – adres pamięci na stosie wywołań, w którym znajduje się poprzednia ramka,
- zmienne lokalne – typy proste języka Java (takie jak *int*, *float*) i wskaźniki na obiekty będące zmiennymi lokalnymi,
- parametry wywołania – typy proste i wskaźniki na obiekty będące parametrami wywołania metody.

Oprócz tego w rejestrach procesora przechowywane są:

- PC – adres pamięci, pod którym znajduje się następna instrukcja wykonywanego programu,
- wskaźnik stosu – wskazuje adres końca stosu wywołań,
- adres bieżącej ramki – adres pamięci na stosie wywołań wskazujący obecnie używaną ramkę.

Przeniesienie RTC z jednego węzła klastra do innego węzła mogłoby się odbywać tylko przy założeniu, że wszystkie węzły klastra działają w tej samej architekturze. Dodatkowo przeniesienie wymagałoby aktualizacji adresów pamięci. Z tego powodu w JESSICA2 migracja wątków odbywa się poprzez tłumaczenie RTC do kontekstu wątku zorientowanego na bajtkod (ang. Bytecode-oriented Thread Context, BTC), który jest niezależny od architektury. BTC składa się z rekordów aktywacji w postaci niezależnej od architektury systemu. Adresy pamięci RTC, które wskazywały na kod programu, w BTC wskazują na instrukcje bajtkodu. Tabela 3.1 objaśnia różnice pomiędzy RTC a BTC:

	RTC	BTC
Licznik rozkazów	fizyczny adres pamięci	instrukcja języka Java
Zmienne lokalne	typy maszynowe	typy Java
Argumenty wywołania	typy maszynowe	typy Java
Adres powrotu	fizyczny adres pamięci	ramka BTC

Tabela 3.1: Różnice pomiędzy RTC i BTC

Przed migracją odbywa się przekształcenie RTC do BTC, a po migracji przekształcenie odwrotne. Następujące problemy zostały rozwiązane podczas implementacji przekształcenia RTC do BTC:

- jedna instrukcja bajtkodu, po przeprowadzeniu kompilacji JIT, może się składać z kilku instrukcji kodu maszynowego. W takim wypadku PC może wskazywać na inną niż pierwsza instrukcję kodu maszynowego. Wówczas nie powinno się tłumaczyć PC na instrukcję bajtkodu, ponieważ instrukcja bajtkodu jest w trakcie wykonywania i jest tylko częściowo wykonana;
- zmienne lokalne mogą się znaleźć w rejestrach procesora;
- nie zawsze można określić typ zmiennych podczas analizy statycznej. Na przykład instrukcja bajtkodu "f2d" zmienia typ z float do double w trakcie uruchomienia.

Do rozwiązania tych problemów stosowana jest technika rekompilacji Just-in-Time, która polega na zebraniu potrzebnych informacji poprzez ponowną kompilację bajtkodu do kodu maszynowego. Kompilacja dokonywana jest tylko dla metod znajdujących się na stosie maszynowym. Tylko te metody są interesujące. Ponowna kompilacja tych metod pozwoli na:

- wykrycie na jakie instrukcje bajtkodu wskazują PC i adresy powrotów,
- określenie typów zmiennych znajdujących się obecnie w ramce,
- wykrycie, które zmienne przechowują swoje wartości w rejestrach procesora.

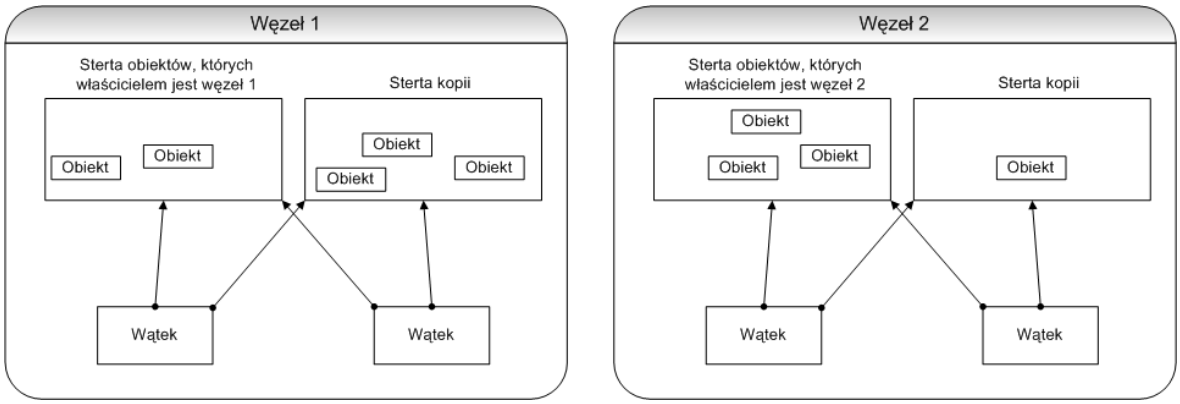
W niniejszej pracy problemy, które się pojawiły podczas prac na JESSICA2, zostały zaledwie nakreślone. Bardziej szczegółowy opis czytelnik może odnaleźć w [Wang04] oraz innych pracach poświęconych JESSICA2.

3.2. Rozproszona sterta obiektów

Tak jak to zostało opisane w podrozdziale 2.6, obiekty języka Java przechowywane są na stercie. Rozproszona sterta obiektów jest dostępna dla wątków znajdujących się na różnych węzłach klastra.

3.2.1. Architektura

Rozproszona sterta obiektów jest w JESSICA2 zaimplementowana poprzez użycie *zwielokrotniania* opisanego w [Tan95]. Obiekt ma jednego właściciela (oryginał) oraz może mieć wiele kopii. Na jedynym węźle może znajdować się tylko jeden oryginał lub kopia obiektu, co oznacza że oryginały i kopie są wykorzystywane przez wszystkie wątki węzła. Kopie tworzy się, aby zredukować liczbę odwołań do oryginału obiektu przez sieć komputerową. Właściciel posiada zawsze aktualne dane. Kopie mogą się znaleźć w trzech stanach: nieważna, tylko do odczytu, możliwość zapisu. Dostęp do nieważnych kopii skutkuje pobraniem aktualnych danych od właściciela. Architektura rozproszonej sterty obiektów przedstawiona jest na rysunku 3.1. Każdy węzeł posiada stertę obiektów oryginałów oraz stertę obiektów kopii. Suma obiektów znajdujących się w stertach zawierających oryginały obiektów jest zbiorem wszystkich obiektów rozproszonej maszyny wirtualnej. Obiekty znajdujące się na stercie mają znacznik informujący o tym, czy dany obiekt jest oryginałem, czy kopią.



Rysunek 3.1: Architektura rozproszonej sterty obiektów

3.2.2. Protokół uaktualniania danych

Protokół rozproszonej sterty obiektów przewiduje aktualizację danych w momencie wejścia lub wyjścia z sekcji *synchronized*. Przed wyjściem z *synchronized* następuje przesłanie wszystkich zmian dokonanych na kopiach przez ten wątek do ich właścicieli. Przed wejściem do sekcji *synchronized* należy unieważnić wszystkie kopie obiektów znajdujące się w pamięci lokalnej wątku. Unieważnienie ma wymusić na wątku pobranie aktualnych danych od właścicieli. Wysłanie zmian dokonanych na kopiach powoduje, że wątek widzi aktualną wartość, gdy pobierze ją dla unieważnionej zmiennej.

3.2.3. Synchronizacja wątków

W podrozdziale 2.5 został opisany sposób w jaki wątki mogą się synchronizować. W przypadku nierozproszonej maszyny wirtualnej, każdy z obiektów posiada blokadę, o której zdobycie

ubiegają się inne wątki. Wątki na zdobycie blokady oczekują w kolejce. W przypadku JESSICA2 synchronizacją zajmują się jedynie obiekty – właściciele. Wątek chcący pozyskać blokadę do obiektu, który istnieje na innym węźle, wysyła przez sieć żądanie zdobycia blokady i oczekuje na odpowiedź. Węzeł, na którym znajduje się właściciel kolejkuje żądanie i gdy zdobycie blokady staje się możliwe, wysyła odpowiedni komunikat przez sieć.

Bardziej szczegółowy opis implementacji rozproszonej sterty obiektów można znaleźć w [Fang03].

3.3. Zmiana właściciela obiektu

Technika zmiany właściciela obiektu będąca częścią protokołu utrzymywania spójności pamięci podręcznej (ang. *Adaptive Cache Coherence Protocol*) pozwala na zmianę właściciela obiektu z jednego węzła na inny w trakcie działania maszyny wirtualnej. Zmianę właściciela obiektu będziemy w tej pracy określać także jako *migrację obiektu*.

Celem zastosowania tej techniki jest wykrywanie sytuacji, gdy tylko jeden proces wykonuje modyfikacje obiektu (ang. *single-writer pattern*). Zmiana właściciela obiektu w takiej sytuacji zmniejsza liczbę zdalnych dostępów do obiektu, gdyż po zmianie właściciela dostępy odbywać się będą do obiektu znajdującego się na tym samym węźle. Zmniejszenie liczby dostępów przez sieć komputerową powinno przynieść zysk wydajnościowy.

Według pracy [Wei04] maszyna wirtualna zlicza seryjne modyfikacje obiektu z jednego węzła, co oznacza że pomiędzy odwołaniami w serii nie mogą występować modyfikacje z innych węzłów, w tym z węzła obecnego właściciela obiektu. Obiekt zostanie przeniesiony w przypadku, gdy seria odwołań przekroczy wyznaczony *próg migracji*.

Po przeniesieniu obiektu jeżeli jakiś węzeł będzie próbował się do niego odwołać, to zostanie przekierowany, czyli zostanie wysłana informacja o zmianie węzła wraz z numerem węzła, na który nastąpiła migracja. Zdaniem autorów pracy, duża liczba przekierowań świadczy o tym, że przeniesienie obiektu wpłynęło niekorzystnie na wydajność systemu, a tym samym należy zwiększyć próg następnej migracji.

Z kolei jeżeli występuje duża liczba dostępów do obiektu z węzła, na którym ten obiekt się obecnie znajduje, to świadczy to pozytywnie o wykonanej ostatnio migracji i zdaniem autorów należy zmniejszyć następny próg migracji.

Bazując na tych obserwacjach *próg migracji* T_i, gdzie i to numer kolejnej migracji danego obiektu, wynosi:

$$T_i = \max\{(T_{i-1} + R_i - \alpha E_i), 1\}$$

gdzie:

T_i – próg migracji od ostatniej $(i-1)$ migracji ($T_0 = 1$),

R_i – liczba wysłanych przekierowań od $(i-1)$ migracji,

E_i – liczba zapisów do obiektu z węzła, na którym obiekt się znajduje od $(i-1)$ migracji,

α – współczynnik migracji.

Jednak według kodów źródłowych JESSICA2, które są dostępne w Internecie próg ten jest wyznaczany w inny sposób. Przechowywana jest lista odwołań do obiektu z 10 węzłów, które jako pierwsze się odwołały do obiektu. Lista jest posortowana. Obiekt jest migrowany w przypadku, gdy liczba odwołań z jednego z węzłów przekroczy 4/5 odwołań wszystkich pozostałych węzłów z przechowywanej listy oraz liczba tych odwołań będzie większa niż 2.

Rozdział 4

Istniejące podejścia do równoważenia obciążenia

Zagadnieniem rozważanym w tej pracy jest równoważenie obciążenia przez przenoszenie wątków, przy uwzględnianie faktu korzystania przez wątki z rozproszonej sterty obiektów. Równoważenie obciążenia polega na przenoszeniu części przetwarzania, tak aby w miarę możliwości nie istniały węzły przeciążone w klastrze.

W tym rozdziale omawiam różne podejścia do równoważenia obciążenia, zarówno te klasyczne, jak i bardziej wyspecjalizowane, z szczególnym uwzględnieniem algorytmów, które biorą pod uwagę współdzielenie danych przez węzły klastra. Opisane w tym rozdziale algorytmy pozwalają równoważyć obciążenie w systemach klastrowych, przez przenoszenie procesów lub wątków. W pierwszym podrozdziale przedstawię klasyfikację algorytmów równoważenia obciążenia. Następnie omówię kilka z nich.

4.1. Klasyfikacja

Algorytmy równoważenia obciążenia można dzielić według następujących kategorii:

1. rodzaj informacji zbieranych przez algorytm,
2. model komunikacji,
3. możliwość migracji wątków w trakcie działania.

4.1.1. Rodzaj informacji zbieranych przez algorytm

Algorytmy równoważenia obciążenia podejmują decyzje o sposobie równoważenia wykorzystując informację o obciążeniu systemu. Rodzaj zbieranych danych ma wpływ na podejmowane decyzje. Jeżeli obciążenie systemu zmienia się dynamicznie, to konieczne jest monitorowanie obciążania w określonych odstępach czasu.

Wszystkie omawiane dalej algorytmy zbierają jakieś informacje. Zwykle węzeł zbiera informacje o samym sobie i ewentualnie przekazuje je do innych węzłów. Możemy wyróżnić 3 rodzaje zbieranych danych:

1. zużycie procesora,
2. zużycie pamięci,
3. częstość komunikacji przez sieć komputerową z innymi węzłami.

Większość algorytmów równoważących obciążenie zbiera informacje tylko o zużyciu procesora i/lub pamięci. Komunikacja z innymi węzłami w trakcie uruchomienia jest rzadko badana, a może mieć duży wpływ na wydajność systemu. Algorytm opracowany w ramach tej pracy bada komunikację z innymi węzłami i uwzględnia ją w procesie podejmowania decyzji.

4.1.2. Model komunikacji

Wyróżniamy algorytmy, które stosują:

1. centralne zarządzanie – istnieje jeden węzeł, który zbiera dane od innych węzłów i podejmuje decyzje w sprawie równoważenia obciążenia,

2. rozproszone zarządzanie – nie istnieje jeden węzeł, który równoważy obciążenie.

Algorytmy z centralnym zarządzaniem mają możliwość podjęcia lepszej decyzji, ponieważ posiadają całą informację o systemie. W przypadku większych klastrów istnieje ryzyko, że węzeł centralny będzie miał problemy z wydajnością. Dlatego czasem bardziej korzystne jest rozproszone zarządzanie.

Systemy z rozproszonym zarządzaniem wymieniają między sobą dane wg ustalonego schematu i na tej podstawie podejmują decyzję o równoważeniu obciążenia.

4.1.3. Możliwość migracja wątków w trakcie uruchomienia

Jeżeli system nie potrafi migrować wątków w trakcie ich działania, to możliwe jest jedynie równoważenie poprzez początkowe przypisanie węzła jednostce obliczeniowej. Do grupy takich algorytmów należy rozwiązanie problemu wyboru węzła, który odpowiada na żądanie HTTP wysłane do serwera WWW. Równoważenie może być przeprowadzane przy użyciu serwera DNS, z wykorzystaniem algorytmów takich jak karuzelowy (ang. *round-robin*), losowy wybór i innych. Ponieważ tematem tej pracy jest dynamiczne równoważenie obciążenia, dlatego algorytmy z tej grupy nie były rozpoznawane i nie będę ich dalej rozważać.

W przypadku, gdy system pozwala na migrację wątków w trakcie działania, można stosować strategie, które będą dobrze działały w programach z nieregularnymi wzorcami działania (np. wątki zużywają czas procesora w pewnych okresach czasu bardziej, a w innych mniej).

4.2. Algorytmy

W tym podrozdziale opisuję algorytmy, które zostały przedstawione w różnych pracach naukowych. Algorytmy zostały dobrane według dwóch kluczy: klasyczne strategie równoważenia obciążenia oraz strategie, które przy równoważeniu biorą pod uwagę fakt współdzielenia danych.

4.2.1. Typowe algorytmy równoważenia obciążenia

W tej części podrozdziału omawiam algorytmy, które podczas równoważenia biorą pod uwagę obciążenie poszczególnych węzłów. Współdzielenie zasobów przez węzły klastra (np. pamięci), nie jest rozpatrywane.

Część z tych algorytmów została porównana eksperymentalnie podczas budowy systemu Amoeba [Zhu95].

Heurystyczne równoważenie przez węzeł główny

Jeden z węzłów zbiera informacje o obciążeniu pozostałych węzłów. W przypadku, gdy jeden z tych węzłów jest przeciążony, węzeł główny przenosi część obliczeń do węzła, który jest najmniej obciążony.

Losowe równoważenie

Losowe równoważenie [San96] działa tak, że w przypadku, gdy jeden z węzłów przekroczy ustalony próg obciążenia, wybiera losowo inny węzeł. Prawdopodobieństwo wyboru jednego z pozostałych węzłów wynosi:

$$\frac{1}{\text{liczba\_węzłów} - 1}$$

W najprostszej wersji algorytmu węzeł, który zostanie wybrany, zawsze akceptuje obliczenia i je uruchamia. W bardziej skomplikowanej wersji węzeł wybrany w przypadku, gdy jest przeciążony, wybiera inny węzeł i do niego przesyła obliczenia.

Odpytywanie

Równoważenie przez odpytywanie [San96] polega na tym, że gdy węzeł przekroczy ustalony próg obciążenia, wysyła zapytanie do innego węzła, czy ten przejmie część obliczeń. W przypadku odpowiedzi pozytywnej, część obliczeń jest przenoszona na ten węzeł, w przeciwnym przypadku odpytywany jest inny węzeł. Wybór węzła może być losowy albo opierać się na poprzednich odpytywaniach i nie wysyłać zapytania do węzłów, które ostatnio odpowiedziały, że są przeciążone.

Rozgłaszanie

W rozgłaszaniu [San96] wszystkie węzły wysyłają w pewnych odstępach czasu do pozostałych węzłów dane o swoim obciążeniu. W przypadku, gdy jeden z węzłów jest niedociążony, proponuje przejęcie części obliczeń od innych węzłów. Propozycja jest składana tym węzłom, które na przechowywanej liście serwerów są najbardziej obciążone.

Dyfuzja naturalna

Algorytm jest opisany w [Sch95]. Pomysł polega na tym, że w grafie G, V – to zbiór węzłów. $A_G(x, y)$, gdzie $x \in V$ i $y \in V$, to macierz krawędzi grafu G. $P_G(x, y)$ to macierz dyfuzji. Jeżeli $A_G(x, y) > 0$, to $P_G(x, y) > 0$ dla $x \neq y$. Wartość $P_G(x, y)$ oznacza, że $P_G(x, y)\%$ różnicy obciążenia pomiędzy węzłami x i y powinna zostać przeniesiona z węzła bardziej obciążonego do mniej przeciążonego. P_G może zostać określona następująco:

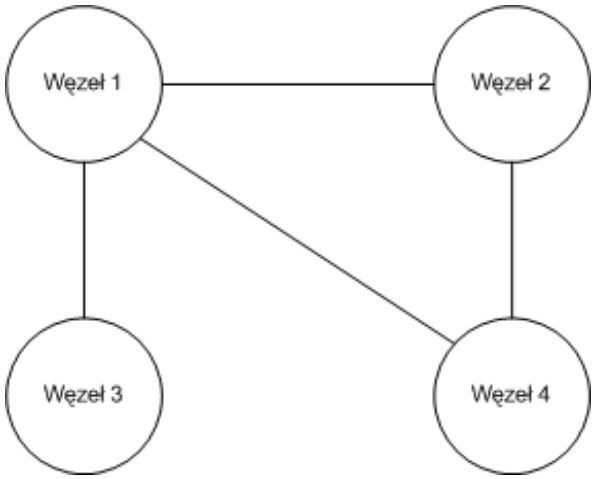
$$P_G(x,y) = \begin{cases} \frac{1}{\max(deg(x),deg(y))+1} & \text{jeżeli } A_G(x,y) > 0, x \neq y \\ 0 & \text{jeżeli } A_G(x,y) = 0, x \neq y \end{cases}$$

gdzie $deg(x)$ to stopień węzła x, czyli liczba krawędzi wychodzących z węzła x.

Po wyznaczeniu macierzy P_G pary węzłów wyliczają różnicę pomiędzy swoim obciążeniem i przenoszą $P_G(x, y)\%$ tej różnicy z bardziej przeciążonego węzła do mniej przeciążonego.

Przykład: tabela 4.1 przedstawia macierz P_G dla grafu przedstawionego na rysunku 4.1.

Przyjmijmy, że przed dyfuzją węzły miały następujące obciążenie:



Rysunek 4.1: Krawędzie grafu

	Węzeł 1	Węzeł 2	Węzeł 3	Węzeł 4
Węzeł 1		$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$
Węzeł 2	$\frac{1}{4}$		0	$\frac{1}{3}$
Węzeł 3	$\frac{1}{4}$	0		0
Węzeł 4	$\frac{1}{4}$	$\frac{1}{3}$	0	

Tabela 4.1: Macierz P_G dla rysunku 4.1

- węzeł 1 – 60%,
- węzeł 2 – 80%,
- węzeł 3 – 70%,
- węzeł 4 – 20%.

Po zastosowaniu dyfuzji przy użyciu macierzy P_G z tabeli 4.1 otrzymamy:

- węzeł 1 – 57,5%,
- węzeł 2 – 55%,
- węzeł 3 – 67,5%,
- węzeł 4 – 50%.

4.2.2. Równoważenie obciążenia w rozproszonej pamięci dzielonej

Zależność wątków od rozproszonej pamięci dzielonej jest rzadko przedmiotem badań naukowych. W tej części pracy zostaną omówione nieliczne algorytmy, które tę tematykę podejmują, a które udało mi się odnaleźć w sieci Internet.

W pracy [Lai97] autorzy przedstawiają równoważenie obciążenia dla rozproszonej pamięci dzielonej. Algorytm bierze pod uwagę zależność pomiędzy wątkami w trakcie podejmowania decyzji o migracji. Może być ona wewnętrzna (ang. *intra-thread*) lub zewnętrzna (ang. *inter-thread*). Zależność zewnętrzna oznacza, że dwa wątki z dwóch różnych węzłów dzielą strony danych. Z kolei zależność wewnętrzna odnosi się do stopnia w jakim wątki dzielą strony na tym

samym węźle. Zwykle komunikacja w rozproszonej pamięci dzielonej ma miejsce wtedy, gdy istnieje zależność zewnętrzna pomiędzy wątkami współdzielącymi stronę. Dlatego też bardziej korzystne jest wybieranie do migracji tych wątków, które mają mało zależności wewnętrznych i wiele zależności zewnętrznych. Uzasadnienie jest takie, że po migracji na inny węzeł część zależności wewnętrznej może zostać zamieniona na zależność zewnętrzną. Wątek z wysoką zależnością wewnętrzną i niską zewnętrzną nie powinien być wybierany do migracji, ponieważ przeniesienie go może zwiększyć użycie sieci komputerowej.

Inne podejście jest zaprezentowane w [Shi01]. Po zmigrowaniu wątku przenoszone są również dane skojarzone z tym wątkiem. Skojarzenie pomiędzy wątkiem a danymi jest wykrywane w trakcie jego działania. To heurystyczne podejście opiera się na obserwacji, że zadanie zawsze blokuje dane na wyłączność, gdy chce je przetwarzać. Co oznacza, że jest jedynym pisarzem w algorytmie utrzymywania spójności (ang. *cache coherence protocol* [Tan95]). Z tego powodu dane, które były ostatnio przetwarzane są przenoszone razem z wątkiem, tak aby w przyszłości dostęp do danych odbywał się lokalnie.

W pracy [Cha93] wprowadzona jest technika podawania wskazówek dla kompilatora w kodzie źródłowym. Język programowania został rozszerzony o zbiór podpowiedzi. Wskazówki pozwalają na:

- definiowanie zależności pomiędzy obiektem a zadaniem,
- definiowanie zależności pomiędzy zadaniem a procesorem,
- określenie procesora, do którego nowotworzony obiekt powinien być przypisany,
- wymuszenie migracji zadania na inny procesor.

Planista wykorzystuje te wskazówki w trakcie pracy. Aby zapewnić dobre zrównoważenie, bezczynny procesor bierze zadania z innych procesorów.

Rozdział 5

Równoważenie obciążenia w systemie z rozproszoną stertą obiektów

W tym rozdziale przedstawiam strategię równoważenia obciążenia z uwzględnieniem danych pochodzących z rozproszonej sterty obiektów. Strategia ta została opracowana i zaimplementowana w ramach tej pracy. Opisane w tym rozdziale rozwiązanie zostało spisane przy założeniu, że będzie implementowane w JESSICA2. W punkcie 5.8 opisuję warunki jakie powinien spełniać system rozproszony, aby można było w nim wykorzystać opisaną w tym rozdziale strategię.

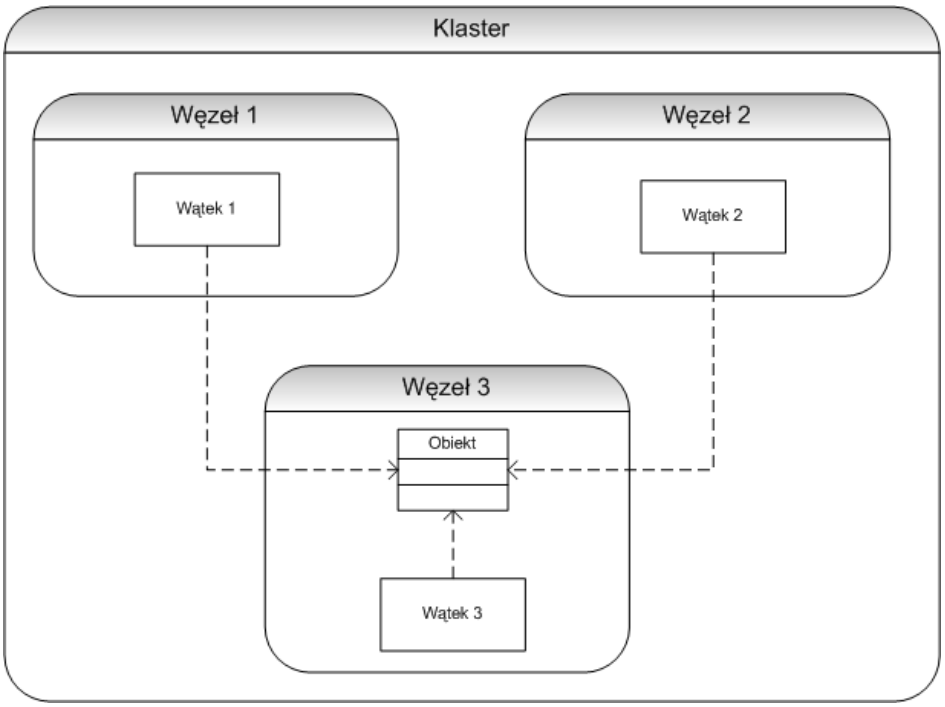
5.1. Narzut czasowy obsługi rozproszonej sterty obiektów

Rozproszona maszyny wirtualna Javy funkcjonuje bardzo dobrze w przypadku, gdy wątki współdzielą obiekty w niewielkim stopniu. W takiej sytuacji wątki mogą działać na osobnych węzłach klastra, narzut na wymianę danych pomiędzy węzłami jest nieduży, a wielowątkowy program napisany w Javie działa wydajnie.

Sytuacja się pogarsza, gdy wątki współdzielą dane i często na nich operują. W skrajnych przypadkach może się zdarzyć, że oprogramowanie będzie działać wolniej, niż gdyby zostało uruchomione na nierozproszonej maszynie wirtualnej, gdyż wątki przesyłają dużo danych między sobą przez sieć komputerową. Stanie się tak np. w sytuacji, gdy kilka wątków, znajdujących się na różnych węzłach będą intensywnie odwoływać się do jednego obiektu. Sytuacja taka jest przedstawiona na rysunku 5.1. Wątki znajdujące się na węzłach nr 1 i 2 muszą oczekiwać na przesłanie danych przez sieć komputerową, przy modyfikacji i odczycie obiektu znajdującego się na węźle nr 3. Dodatkowo obciążeniu ulega węzeł nr 3, który musi odpowiadać na żądania z węzłów nr 1 i 2.

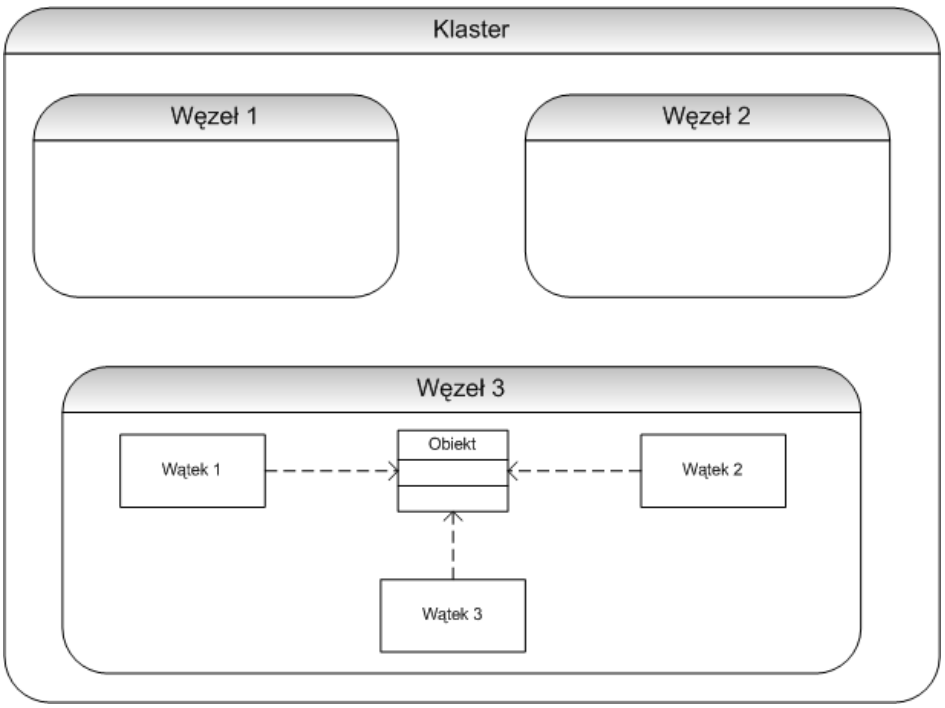
5.2. Przenoszenie wątków intensywnie wykorzystujących sieć

Przedstawiony dalej pomysł polega na przenoszeniu tych wątków, które intensywnie korzystają z obiektów znajdujących się na innym węźle. Gdy takie zachowanie zostanie wykryte, wątek zostanie przeniesiony na węzeł, na którym znajdują się używane obiekty, z wykorzystaniem opisanego wcześniej mechanizmu migracji wątków. Rysunek 5.2 przedstawia sytuację, gdy wątki z rysunku 5.1 zostały przeniesione. Co prawda w przedstawionej sytuacji nie są



Rysunek 5.1: Współdzielenie obiektu przez wątki na różnych węzłach

wykorzystywane możliwości rozproszonej maszyny wirtualnej, pozwalające na uruchamianie wątków na różnych węzłach, jednak przy założeniu, że zniknie dominujący narzut na przesyłanie danych przez sieć komputerową, da to zysk wydajnościowy.



Rysunek 5.2: Współdzielenie obiektu przez wątki na jednym węźle

Nie bez znaczenia jest koszt czasowy przeniesienia wątku. Przeniesienie wątku wymaga zatrzymania wątku, przesłania jego stosu przez sieć komputerową na inny węzeł oraz wznowienia jego działania. Najczęściej trzeba wykonywać jeszcze dodatkowe czynności, które powiększają koszt, takie jak np. przekształcenie BTC do RTC i odwrotne w JESSICA2.

Z tego powodu wykonywanie migracji wątku powinno się odbywać z pewną ostrożnością, tak by nie pogarszać wydajności systemu.

5.3. Zbieranie informacji

Aby móc przenosić wątki, tak jak to zostało opisane w p. 5.2, każdy wątek rozproszonej maszyny wirtualnej zlicza odwołania do obiektów w rozproszonej pamięci dzielonej. Zliczane są te odwołania, które wymagają dostępu poprzez sieć komputerową. Wątki przechowują dane o liczbie wysłanych żądań do każdego innego węzła klastra. Dane o odwołaniach zbierane są cyklicznie przez węzeł główny, który otrzymuje te dane od wszystkich wątków maszyny wirtualnej. Określenie, który węzeł jest główny odbywa się podczas uruchamiania maszyny wirtualnej i się nie zmienia w czasie jej działania. Węzeł główny uczestniczy w przyłączaniu nowych węzłów. W tym węźle odbywa się również podejmowanie decyzji o równoważeniu obciążenia poprzez migrację wątków.

5.4. Pomiar szybkości sieci

W celu precyzyjnego określenia jak dużo czasu pochłania wątkowi dostęp do danych znajdujących się na innych węzłach klastra, szacowana jest szybkość sieci komputerowej łączącej węzły. Dzieje się to przy starcie maszyny wirtualnej.

Pomiar sieci jest wykonany przy założeniu, że wszystkie łącza pomiędzy węzłami klastra są tej samej jakości i ich jakość nie zmienia się w trakcie działania. W przyszłości można by rozszerzyć algorytm o sprawdzanie jakości innych połączeń oraz na cykliczny pomiar połączenia w pewnych odstępach czasu, co miałoby znaczenie w przypadku, gdy przepustowość łączy zmienia się w czasie.

Test połączenia polega na wysłaniu X żądań, co w założeniu ma odpowiadać X dostępom do obiektów znajdujących się w innym węźle rozproszonej pamięci dzielonej. Pomiar wykonany jest pomiędzy węzłem głównym a jednym z węzłów maszyny rozproszonej. W tym miejscu wykorzystywane jest założenie, że wszystkie łącza pomiędzy węzłami są tej samej jakości. Program mierzy czas w jakim udało się przesłać X żądań. Po pomiarze można stwierdzić jaki jest średni czas dostępu do obiektu rozproszonej pamięci dzielonej, znajdującego się na innym węźle. Czas ten wynosi:

$$\frac{zmierzony\_czas\_trwania\_testu}{X}.$$

5.5. Wyliczanie progu migracji

W zaprojektowanej przeze mnie strategii węzeł główny rozproszonej maszyny wirtualnej dostaje informacje od innych węzłów o odwołaniach z wątków uruchomionych na tych węzłach, do zdalnych obiektów maszyny wirtualnej. Informacje są zbierane w określonych odstępach czasu. Żądanie dostarczenia liczby odwołań przesyła węzeł główny. Pozostałe węzły odpowiadają na to żądanie. Po ich wysłaniu węzły zerują swoje liczniki odwołań. W ten sposób węzeł główny posiada informacje o liczbie odwołań jaka miała miejsce w ostatnim odstępie czasu.

W strategii równoważenia obciążenia wątek jest migrowany na inny węzeł, gdy przekroczony zostanie próg migracji. Próg migracji P_n jest określany indywidualnie dla każdego węzła n. Wątek znajdujący się na węźle n przekroczy próg migracji, gdy liczba odwołań wymagających dostępu przez sieć, na jeden z pozostałych węzłów x przekroczy P_n. Wówczas wątek zostanie przemigrowany na węzeł x.

Próg migracji P_n wyliczany jest wg wzoru:

$$P_n = \frac{delay}{requestTime} * \frac{1}{T(n)} * c \tag{5.1}$$

gdzie:

$delay$ – odstęp czasu pomiędzy zbieraniem danych o wątkach,
$requestTime$ – średni czas zapytania wyliczony podczas pomiaru szybkości sieci,
$T(n)$ – liczba wątków działających na węźle n,
c – stała, liczba rzeczywista większa niż 0 i mniejsza od 1.

Próg P_n ma następujące znaczenie: $\frac{delay}{requestTime}$ oznacza ile żądań mogło być wysłane w ostatnim odstępie czasowym. Liczba ta jest mnożona przez $\frac{1}{T(n)}$, ponieważ na węźle n działa $T(n)$ wątków i każdy z nich średnio zużywa $\frac{1}{T(n)}$ czasu węzła. Liczby $delay$ i współczynnik c są ustalone i nie zmieniają się w trakcie działania maszyny wirtualnej.

Przykładowo jeżeli $delay$ wynosi 2 sekundy, a zmierzona szybkość sieci wynosi 10 milisekund, to:

$$\frac{delay}{requestTime} = \frac{20000ms}{10ms} = 2000$$

co oznacza, że w czasie 2 s wątki na węźle n mogły wykonać średnio około 2000 odwołań do obiektów znajdujących się na innych węzłach.

Ponieważ na węźle n jest uruchomionych $T(n)$ wątków, więc średnio $1/T(n)$ czasu $delay$ zużył jeden z nich. Opisywana strategia opiera się w tym miejscu na założeniach średniego zużycia procesora przez wątki, co może w szczególnych przypadkach powodować różnice pomiędzy przyjętym założeniem a rzeczywistym zużyciem. Opisany algorytm należy do klasy algorytmów heurystycznych.

Jeżeli w naszym przykładzie na węźle n uruchomione są 4 wątki, to:

$$\frac{delay}{requestTime} * \frac{1}{T(n)} = 2000 * \frac{1}{4} = 500$$

co oznacza, że jeden wątek na węźle n mógł wykonać średnio nie więcej niż 500 dostępów do obiektów na innych węzłach.

Współczynnik c oznacza jaki procent z wyliczonej poprzednio wartości musi zostać przez wątek przekroczony, aby został zmigrowany. Jeżeli współczynnik c zostanie ustalony na $0,25$, to:

$$P_n = \frac{delay}{requestTime} * \frac{1}{T(n)} * c = 500 * 0,25 = 125$$

co oznacza, że jeżeli wątek odwoła się do innego węzła x więcej niż 125 razy, to zostanie przeniesiony na węzeł x.

W przyszłości można by mierzyć rzeczywisty czas jaki wątek spędza na dostępie do obiektów znajdujących się na węźle x i tym samym posiadać dokładną informację o czasie jaki zajmuje wątkowi korzystanie z rozproszonej sterty obiektów.

5.6. Migracja

Węzeł główny zgłasza żądanie migracji do węzła, na którym znajduje się wątek, w przypadku, gdy wątek przekroczył wyliczony próg migracji. Migracja realizowana jest przez istniejący w JESSICA2 mechanizm migracji wątków. Mechanizm nie gwarantuje, że wątek zostanie zmigrowany. Do migracji może nie dojść w przypadku, gdy wątek ma aktywne blokady zakładane przez maszynę wirtualną. Nie należy mylić tej blokady, zakładanej na wątku, z blokadą opisaną w p. 2.5, zakładaną na obiekcie i służącą do synchronizacji wątków. Ta blokada na wątku jest rozwiązaniem technicznym użytym przy konstrukcji maszyny wirtualnej. Blokada jest zakładana, gdy w wątku odbywa się modyfikowanie rozmiaru napisu lub gdy wątek wszedł do monitora (*synchronized*). Założenie blokady uniemożliwia migrację wątku.

W trakcie implementacji algorytmu stworzonego w ramach tej pracy, została wprowadzona zmiana w JESSICA2, szerzej opisana w p. 6.5. Zmiana ta polega na ponawianiu co pewien czas żądania migracji, tak by wątek został zmigrowany, gdy tylko migracja będzie możliwa.

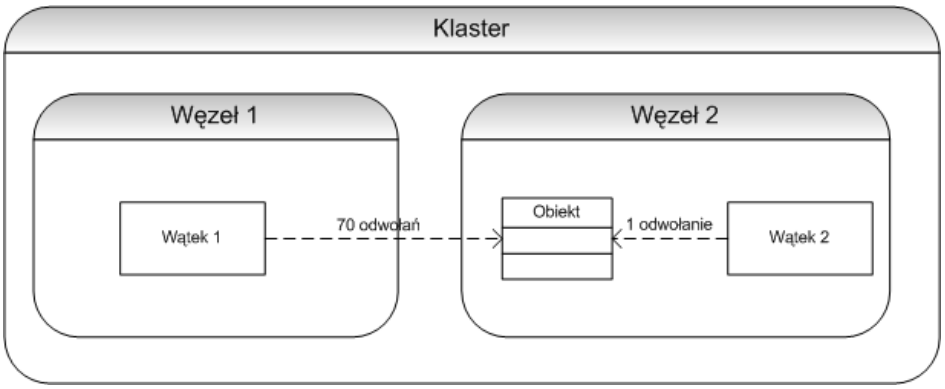
W przyszłości warto by wprowadzić taką zmianę w JESSICA2, aby migracja wątku nie musiała być kolejkowana i wielokrotnie ponawiana, tylko, żeby wątek wykonywał migrację tak szybko, jak to tylko będzie możliwe, czyli gdy skończy modyfikację długości napisu lub gdy wyjdzie z sekcji *synchronized*.

5.7. Współpraca z protokołem zmiany właściciela obiektu

JESSICA2 ma zaimplementowany mechanizm zmiany położenia obiektów [Wei04]. Obiekty istniejące w rozproszonej pamięci dzielonej mają swój jeden oryginał i pewną liczbę kopii. Wątki odwołują się do oryginału, kiedy mają nieważną kopię lub gdy przeprowadzają synchronizację na obiekcie. Zaproponowany w JESSICA2 protokół zmiany właściciela obiektu powoduje zmianę położenia oryginału w przypadku, gdy liczba odwołań do tego obiektu z jednego z węzłów jest większa od sumy odwołań z pozostałych węzłów. Bardziej szczegółowe informacje o algorytmie zmiany właściciela obiektu znajdują się w p. 3.3.

Strategia zaproponowana w tej pracy wstrzymuje migrację wątku na X odstępów czasu, dając tym samym szansę na migrację obiektów przy użyciu algorytmu zmiany właściciela obiektu. Wątek jest migrowany dopiero wtedy, gdy przy X odstępie czasu wątek nadal przekracza wyliczony próg. Rozważmy sytuację przedstawioną na rysunku 5.3. Dwa wątki odwołują się do jednego obiektu, z czego jeden intensywnie, a drugi mało intensywnie. Czekając na akcję algorytmu zmiany właściciela obiektu, daje się możliwość przeniesienia obiektu na węzeł 1. W sytuacji, która jest przedstawiona na rysunku, obiekt powinien zostać przeniesiony na węzeł 1, co spowoduje, że migracja wątku nie będzie już konieczna. Przeniesienie obiektu jest bardziej korzystne niż przeniesienie wątku, ponieważ operacja przeniesienia obiektu jest mniej kosztowna.

W przyszłości można by pobierać dane od algorytmu zmiany właściciela obiektu, żeby posiadać informację o tym, że pewne obiekty mogą być w najbliższym czasie zmigrowane i jeżeli tak, to wstrzymywać migrację, a w przeciwnym przypadku migrować. Algorytm zmiany właściciela obiektu mógłby określać z jakim prawdopodobieństwem obiekty mogą zostać przeniesione, bazując na znajomości tego, jak bliski osiągnięcia progu migracji obiektu jest któryś z węzłów. Na decyzję, czy czekać z migracją, czy nie czekać, powinna mieć także wpływ liczba obiektów, które mają szansę być w najbliższym czasie przeniesione.



Rysunek 5.3: Współdzielenie obiektu przez dwa węzły

5.8. Ogólność rozwiązania

Pomimo, że zaprezentowane przeze mnie rozwiązanie jest zaimplementowane dla konkretnego systemu rozproszonego JESSICA2, może ono zostać przeniesiony na dowolny system rozproszony, który spełnia następujące warunki:

- system jest uruchamiany na klastrze – ma to znaczenie przy pomiarze sieci, gdyż istnieje założenie, że czas przesłania komunikatu przez 2 węzły nie jest znacząco różny;
- procesy wymieniają między sobą dane;
- procesy można przenosić w trakcie ich działania;
- system pozwala na wybranie jednego węzła, który mógłby odpytywać pozostałe i wydawać żądania migracji.

Rozdział 6

Opis implementacji

Punktem wyjścia dla implementacji maszyny wirtualnej JESSICA2 był projekt typu *open source* o nazwie Kaffe. JESSICA2 jako modyfikacja Kaffe jest również oprogramowaniem typu *open source*. Kaffe jest maszyną wirtualną Javy napisaną bez wglądu do architektury maszyny wirtualnej napisanej przez Sun Microsystems, co oznacza, że została stworzona bez sprawdzania jak działa maszyna wirtualna napisana przez twórców języka Java. Ponadto Kaffe nie ma akceptacji Sun Microsystems jako maszyna wirtualna Javy, ponieważ nie dostarcza części funkcjonalności wymaganej przez specyfikację Javy – mechanizmu bezpieczeństwa uruchamianego kodu.

Istnieje wiele wersji Kaffe na różne platformy systemowe. Podczas implementacji JESSICA2 nie została zachowana przenośność Kafee i obecnie istnieje wersja tylko dla systemu operacyjnego Linux. Całość kodu źródłowego Kaffe i JESSICA2 jest napisana w języku C.

6.1. Odwołania do obiektów

Każdy wątek maszyny wirtualnej przechowuje wskaźnik do tablicy liczb. W tej tablicy przechowywana jest liczba odwołań do obiektów z rozproszonej sterty wymagających przesłania danych przez sieć komputerową do innych węzłów klastra. Rysunek 6.1 przedstawia tablice, które przechowują liczbę odwołań do innych węzłów dla każdego z wątków. Wątki zliczają odwołania oddzielnie dla każdego węzła bez podziału na poszczególne obiekty. Duża liczba odwołań wątku do obiektów z innego węzła może spowodować jego migrację właśnie na ten węzeł.

6.2. Identyfikacja wątków

Każdy wątek w momencie jego tworzenia otrzymuje swój identyfikator unikatowy, w obrębie jednego węzła. Identyfikator będący adresem pamięci struktury, w której przechowywane są dane o wątku, nie jest dobry. Może się bowiem zdarzyć, że wątek zakończy swoje działanie i następnie zostanie utworzony nowy wątek, którego struktura zajmie zwolniony adres pamięci. Z tego powodu w swojej implementacji stosuję identyfikatory z sekwencji, będące kolejnymi liczbami naturalnymi.

Sekwencja jest zaimplementowana przez użycie typu 64 bitowego, co daje możliwość uzyskania 18446744073709551616 kolejnych numerów. Przy założeniu, że w ciągu sekundy tworzonych jest milion wątków, sekwencja przepełni się za około 585 tysięcy lat, co jest znacząco większe od czasu życia obecnych komputerów.

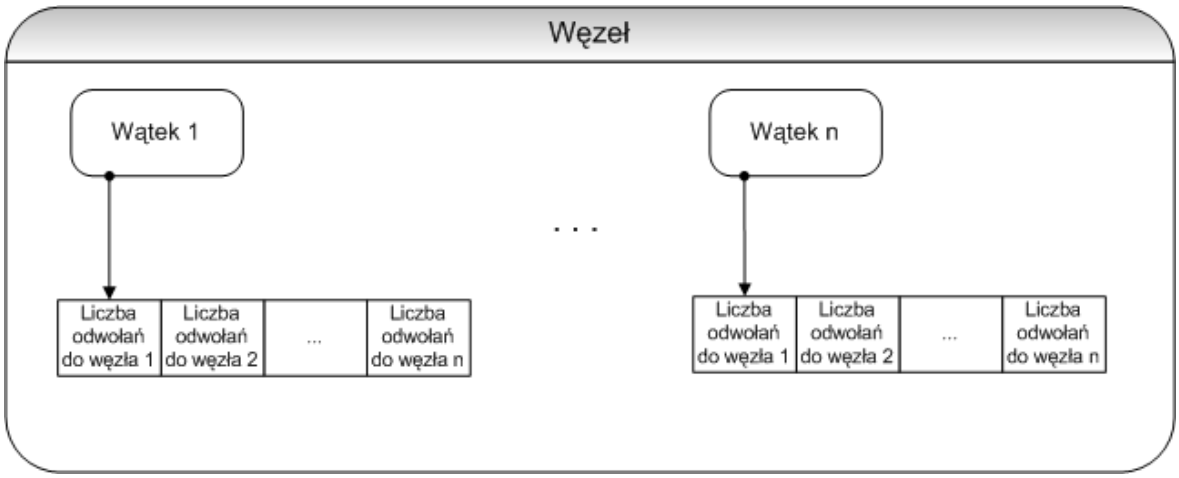
6.3. Tablica mieszająca wszystkich wątków

Do obsługi żądań migracji konieczne jest wyszukanie wątku po identyfikatorze. W implementacji zastosowana została tablica mieszająca dostarczona razem z Kaffe, w której kluczem jest identyfikator, wartością jest struktura wątku, a funkcją mieszającą jest identyczność, czyli wartością funkcji mieszającej jest identyfikator wątku. Ta wartość jest następnie wykorzystywana przez moduł tablicy mieszającej Kaffe i dopasowywana do bieżącej wielkości tablicy. Takie rozwiązanie gwarantuje obsługę żądania wyszukania wątku, pojawiające się przy migracji wątku, w czasie $O(1)$.

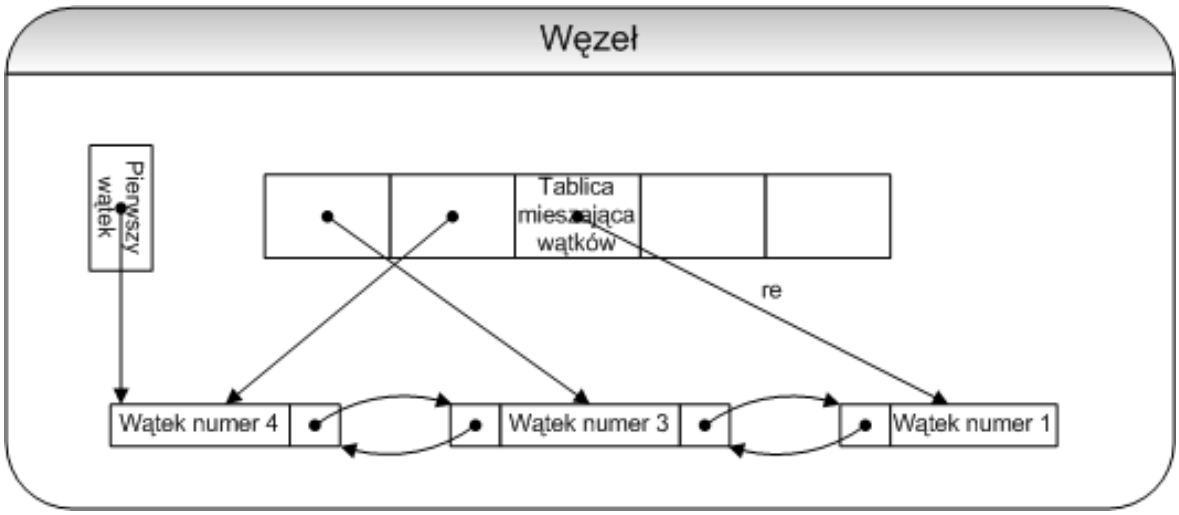
Każdy węzeł przechowuje jedną taką tablicę. Rysunek 6.2 ilustruje tablicę mieszającą wszystkich wątków na jednym z węzłów.

Do tablicy mieszającej wątki dodawane są w momencie ich tworzenia, a usuwane w jednym z przypadków:

- zakończenie działania wątku,
- zmigrowania wątku na inny węzeł.



Rysunek 6.1: Tablice odwołań do innych węzłów



Rysunek 6.2: Tablica mieszająca wątków i lista wątków gotowych do migracji

6.4. Lista wątków, które można zmigrować

Każdy węzeł, aby móc przesłać do węzła głównego dane o odwołaniach do obiektów rozproszonej sterty obiektów, musi mieć dostęp do struktur wątków. Dostęp do tych struktur zapewnia lista wątków gotowych do migracji. Do tej listy wątki dołączane są w momencie ich powstania. Usunięcie z listy odbywa się w jednym z przypadków:

- zakończenie działania wątku,
- otrzymanie żądania migracji wątku na inny węzeł.

6.5. Zbiór wątków, które otrzymały polecenie migracji

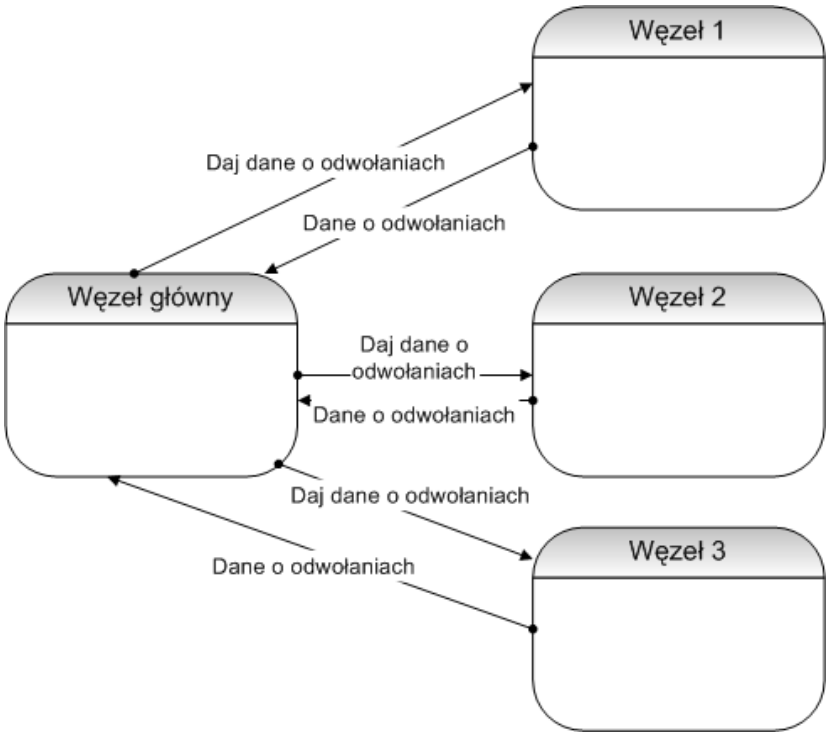
W punkcie 5.6 opisane są możliwe przyczyny, z powodu których migracja może być w danym momencie niemożliwa.

Ponieważ nie jest pewne, że migracja jest w danym momencie możliwa, więc wątki oczekują na migrację w kolejce. W pewnych odstępach czasowych przeprowadzana jest ponowna próba migracji wątków. W momencie, gdy migracja się powiedzie wątek usuwany jest ze zbioru wątków czekających na migrację. Zbiór wątków czekających na migrację jest zrealizowany przy użyciu tablicy.

6.6. Przesyłanie danych do węzła głównego

Węzeł główny w ustalonych odstępach czasowych, przed rozpoczęciem wykonania algorytmu równoważącego obciążenie, wysyła żądanie do pozostałych węzłów o podanie liczby odwołań do obiektów. Każdy z węzłów posiada wątek maszyny wirtualnej odpowiedzialny za odpowiadanie na żądania konieczne do realizacji rozproszonej maszyny wirtualnej. Wątki te są tworzone przez istniejący w Kaffe mechanizm tworzenia wątków. W Kaffe został użyty do realizacji wątku wykonującego odśmiecanie.

Gdy węzeł dostanie żądanie przesłania liczby odwołań, wówczas przesyła przechowywane tablice liczby odwołań wątków gotowych do migracji, wraz z numerami tych wątków. Po zebraniu tych danych zeruje liczniki. Węzeł główny dostaje zatem informację o odwołaniach, które miały miejsce od ostatniego wysłania zapytania (przyrost). Przepływ danych ilustruje rysunek 6.3.



Rysunek 6.3: Przepływ danych o odwołaniach do obiektów

Rozdział 7

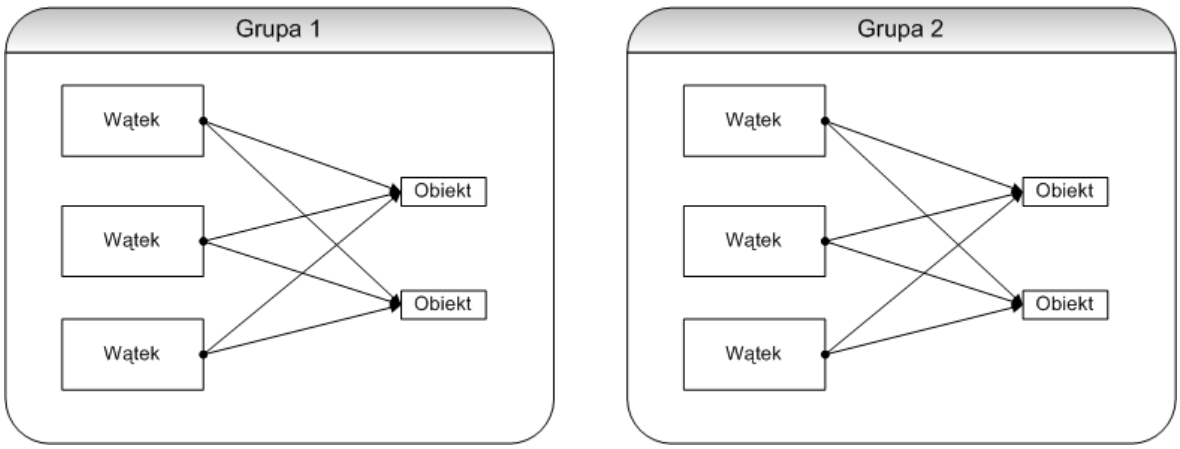
Testy wydajnościowe

7.1. Środowisko przeprowadzania testów

Testy przeprowadzono w laboratorium komputerowym Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego. Laboratorium składa się z komputerów PC Pentium 4 2.60 GHz, 1GB RAM, połączonych siecią Fast Ethernet 100 Mb/s podłączone do 2 różnych przełączników (ang. *switch*).

7.2. Metodologia

Testy polegają na uruchomieniu programu w języku Java, który tworzy x grup po y wątków. Każda z grup ma z obiektów, do których mogą się odwoływać wyłącznie obiekty z danej grupy. Rysunek 7.1 przedstawia sytuację, gdy istnieją 2 grupy po 3 wątki i każda z grup ma po 2 obiekty. Strzałki ilustrują możliwe odwołania do obiektów. Każdy z wątków dostaje w



Rysunek 7.1: Podział na grupy

konstruktorze zbiór obiektów, do których może się odwoływać. Poniżej znajduje się implementacja metody run() wykonywanej przez każdy z wątków. Wątek najpierw pobiera dane do obliczeń od jednego z obiektów dzielonych z grupy. Obiekty te są przechowywane w tablicy obiektów dzielonych *so*. Następnie wykonywane są obliczenia arytmetyczne, po czym ich wynik jest oddawany do obiektu dzielonego. Oto kod metody run:

```
public void run(){
```

```
        for (int iter = 0; iter < 2000; iter++){
            long d;
            // pobranie danych od obiektu dzielonego
            if (so.length > 0)
                d = so[iter % so.length].nextData();
            long gotData = d;

            // obliczenia z użyciem pobranych danych
            for (int i = 0; i < 100; i++){
                long val = d;
                for (int j = 0; j < i + 3; j++)
                    d = d * val;
            }

            // oddanie danych
            if (so.length > 0)
                so[iter % so.length].putData(gotData, d);

            // sleep 10 ms
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

        }
    }
```

Do uruchamiania testów zostało wykorzystane narzędzie *Grinder* [Grinder]. Zaprojektowany test porównywał czas trwania programu na maszynie wirtualnej z zaimplementowanym w ramach tej pracy algorytmem oraz na maszynie wirtualnej JESSICA2 w wersji pobranej z Internetu.

Wersja pobrana z Internetu nie stosuje żadnej strategii równoważenia obciążenia i nie stosuje przenoszenia wątków w trakcie ich uruchomienia. Według wyjaśnień jednego z twórców tej maszyny wirtualnej, równoważenie obciążenia zostało wyłączone z powodów wydajnościowych. Stosowana jest natomiast migracja wątków przed ich uruchomieniem.

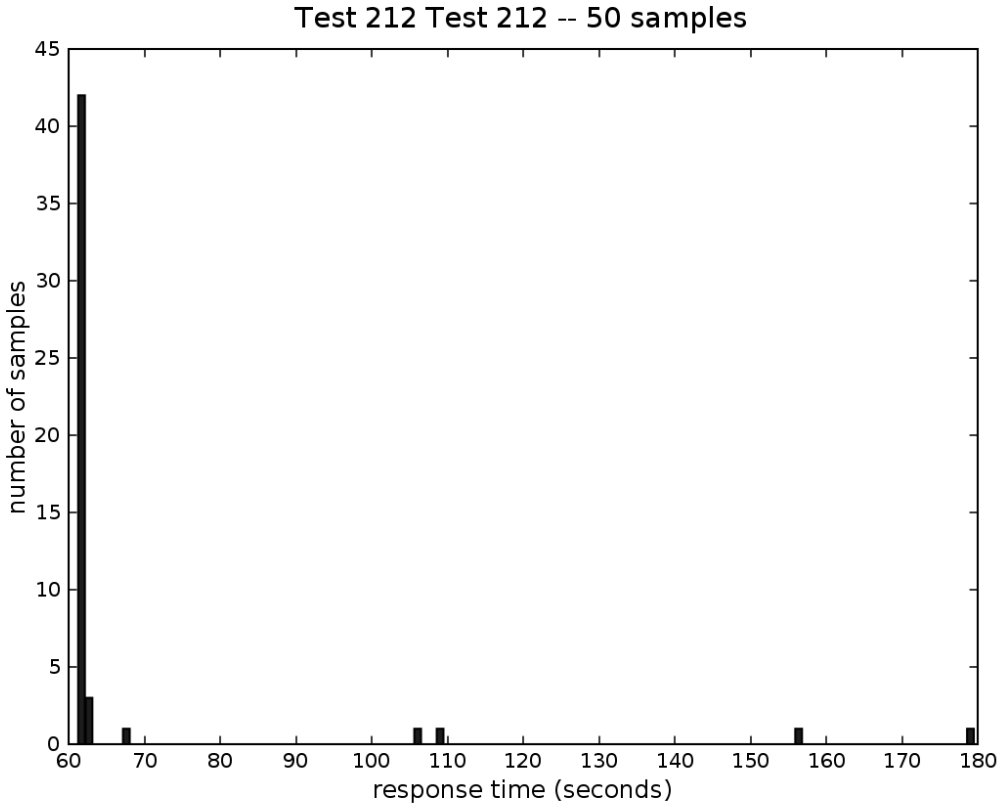
7.3. Wyniki testów

Tabela 7.1 przedstawia wyniki testów. Jeden wiersz tabeli odpowiada jednemu testowi. Testy były powtórzone 50 razy. Wersja nr 1 to test opisany w p. 7.2. Wersja nr 2 zostanie opisana dalej.

Maszyna wirtualna z algorytmem opisanym w tej pracy działała szybciej w testach przeprowadzonych na 3 węzłach i przy współdzielonych obiektach. Zysk w tych przypadkach wyniósł od 25% do 56%. W pozostałych przypadkach zwykle następowało niewielkie pogorszenie wydajności. O ile w przypadku, gdy obiekty nie są współdzielone (0 obiektów), to sytuacja jest naturalna, gdyż nie dochodzi do żadnych migracji, gdyż wątki nie odwołują się zdalnie do obiektów, to w przypadku gdy test został uruchomiony na 10 węzłach, sprawa jest

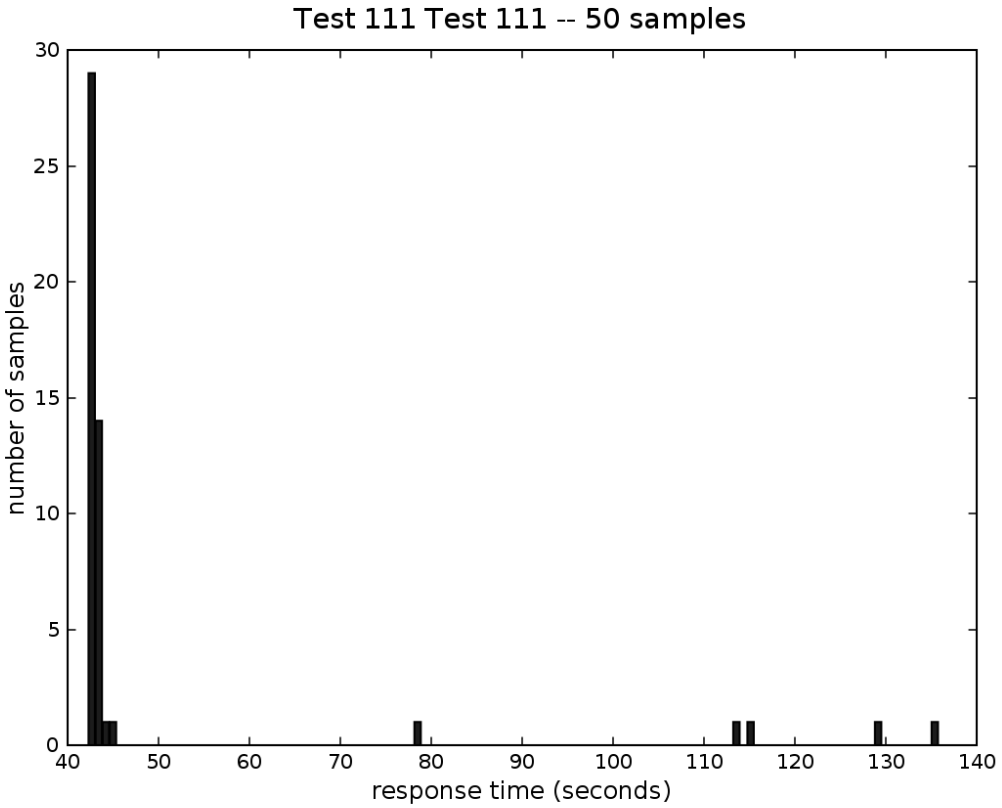
zastanawiająca. Być może algorytm nie zdążył przenieść wszystkich wątków do odpowiednich obiektów i gdyby test trwał dłużej, to wynik byłby lepszy.

W części testów widoczne były duże odchylenia standardowe. Taka sytuacja miała miejsce, gdy test wykonywany był na 10 komputerach. W przypadku oryginalnej maszyny wirtualnej rys. 7.2 przedstawia rozkład trwania testów, gdy test był przeprowadzony na 10 węzłach z jedną grupą 25 wątków, gdzie grupa miała 100 obiektów współdzielonych. W tym przypadku istnieje kilka uruchomień o długim czasie trwania, które mogły być przyczyną dużego odchylenia standardowego. Z kolei rys. 7.3 przedstawia rozkład dla zmienionej maszyny wirtualnej, gdy test był przeprowadzony na 10 węzłach z jedną grupą 25 wątków, gdzie grupa miała 10 obiektów współdzielonych. W tym przypadku również istnieje kilka uruchomień o długim czasie trwania.



Rysunek 7.2: Test na niezmienionej JESSICA2 na 10 węzłach z jedną grupą 25 wątków, gdzie grupa miała 100 obiektów współdzielonych

Wersja nr 2 testu różni się nieznacznie od wersji nr 1. Zamiast typu prymitywnego long, do reprezentacji liczb został użyty typ obiektowy Long. Co za tym idzie migracja wątków nie powodowała migracji obiektów, gdyż w trakcie migracji przenoszony jest tylko stos, a obiekty na które wskazują referencje znajdujące się na stosie zostają na maszynie, na której wątek był poprzednio uruchomiony. W przypadku tego testu przeniesiony wątek był jedynym wątkiem, który posiadał referencje na te obiekty, zatem należałoby przenieść te obiekty w trakcie migracji razem z wątkiem. Podobny pomysł jest prezentowany w pracy [Shi01] opisanej w p. 4.2.2.



Rysunek 7.3: Test na zmienionej JESSICA2 na 10 węzłach z jedną grupą 25 wątków, gdzie grupa miała 10 obiektów współdzielonych

Wersja	Węzły	Grupy	Wątków w grupie	Obiektów w grupie	Liczba testów	Oryginalna JESSICA2		Zmieniona JESSICA2		Zmiana [%]
						Średni czas [s]	Odchylenie standardowe [s]	Średni czas [s]	Odchylenie standardowe [s]	
1	3	5	5	0	50	42,03	0,36	42,44	0,33	100,99
1	3	5	5	10	50	58,74	2,26	43,93	0,53	74,79
1	3	5	5	100	50	91,72	3,02	46,76	1,73	50,98
1	10	5	5	0	50	44,29	8,39	45,92	16,15	103,67
1	10	5	5	10	50	45,54	12,25	46,00	11,99	101,02
1	10	5	5	100	50	45,19	9,90	47,88	18,34	105,97
1	3	1	25	0	50	41,93	0,09	42,33	0,47	100,96
1	3	1	25	10	50	61,69	0,46	44,21	0,21	71,66
1	3	1	25	100	50	115,78	7,34	53,15	3,26	45,9
1	10	1	25	0	50	44,83	8,44	46,09	13,03	102,81
1	10	1	25	10	50	44,42	7,75	50,09	22,48	112,78
1	10	1	25	100	50	67,97	22,52	53,36	10,29	78,51
2	3	5	5	100	50	159,85	0,68	201,31	8,00	135,64

Tabela 7.1: Wyniki testów

7.4. Błąd w JESSICA2

W trakcie prac nad implementacją JESSICA2 został wykryty błąd, który objawiał się w następujący sposób: raz na kilka lub kilkanaście uruchomień programu testowego, test się nie kończył. Śledzenie przyczyny błędu było bardzo trudne i żmudne z uwagi na fakt, że maszyna wirtualna jest rozproszona, z uwagi na dużą złożoność maszyny wirtualnej oraz na rzadką powtarzalność błędu. Do śledzenia stosów wywołań maszyn wirtualnych uruchomionych na różnych węzłach oraz do przeglądania struktur danych zostało użyte narzędzie *gdb* [GDB]. Kilkanaście dni śledzenia struktur danych oraz zachowania programu dało rozwiązanie problemu. Błąd występował w funkcji *_unlockMutex*. Oto jej wersja przed poprawką:

```
void
_unlockMutex(iLock** lkp, void* where)
{
        uintp val;

        val = (uintp)*lkp;
        if(currentJThread)
                currentJThread->nativeFactors --;
        if ((val & 1) != 0) {
                slowUnlockMutex(lkp, where);
        }
        else if ((val == (uintp)where) /* XXX squirrely bit */
                && !COMPARE_AND_EXCHANGE(lkp, (iLock*)where, LOCKFREE)) {
                slowUnlockMutex(lkp, where);
        }
}
```

Funkcja ta jest wywoływana w momencie zwalniania wewnętrznej blokady maszyny wirtualnej. Założenie blokady uniemożliwia przeprowadzenie migracji wątku, co zostało opisane w punkcie 5.6. Pole *nativeFactors* struktury opisującej wątek maszyny wirtualnej zawiera liczbę zdobytych przez ten wątek blokad. Jeżeli $nativeFactors > 0$, to wątek nie może zostać zmigrowany. W przedstawionym kodzie liczba blokad jest zmniejszana, a następnie blokada jest zwalniana. W związku z tym zachodziło następujące zjawisko: gdy liczba blokad została zmniejszona, najprawdopodobniej przychodziło przerwanie, które przerywało program w tym miejscu, nim blokada została zdjęta. Następnie do działania dochodził wątek, który przeprowadzał migrację zawieszonego wątku (mógł zmigrować, bo $nativeFactors == 0$). Wątek zostawał migrowany, zatem kod odblokowujący blokadę był uruchamiany już na innym węźle. Kilka kolejnych wątków czekało na zdobycie tej blokady, ale blokada nigdy nie została zwolniona.

Błąd ten objawiał się tylko w sytuacji, gdy podejmowana była próba migracji wątku w trakcie jego uruchomienia. W kodzie JESSICA2 pobranym z Internetu migracja wątków w trakcie uruchomienia była wyłączona, co powodowało, że opisane zjawisko nie występowało.

Rozdział 8

Podsumowanie

Równoważenie obciążenia z badaniem użycia danych przez procesy nie cieszyło się do tej pory zainteresowaniem badaczy i było rzadko stosowane. Praca ta może się stać impulsem do dalszych badań nad przenoszeniem przetwarzania bliżej danych, bowiem takie przenoszenie może dawać dobre rezultaty i zwiększać wydajność systemów rozproszonych.

Jak wykazały przeprowadzone przeze mnie testy wydajnościowe, zysk czasowy stosowania opisanej strategii równoważenia jest w wielu przypadkach znaczący i był obserwowalny pomimo dużej prostoty zastosowanego algorytmu. Jednak w pewnych przypadkach, gdy uruchomiony program korzystał z utworzonych przez siebie obiektów, algorytm nie sprawdzał się dobrze. Żeby usunąć tę wadę warto zaimplementować przenoszenie danych razem z wątkiem.

Zaletą opisanego w tej pracy algorytmu jest fakt, że można go zastosować w systemach rozproszonych, w których procesy używają danych, których właściwa wartość znajduje się na jednym z węzłów oraz pozwalają przenosić procesy w trakcie uruchomienia. Z tego względu zaproponowana strategia jest ogólna i przenośna.

Przeprowadzone testy wykazały również, że konieczne jest przenoszenie danych razem z migracją przetwarzania. Brak takiego przenoszenia danych przyczynił się do znaczącego obniżenia wydajności w jednym z przeprowadzonych testów. W przyszłości rozszerzenie JESSICA2 o takie przenoszenie może poprawić wydajność migracji wątków.

Dodatek A

Zawartość płyty CD

ssadziak.pdf	–	praca magisterska w wersji PDF
jessica2-i386-linux-jitr.tar.gz	–	JESSICA2 w wersji pobranej z Internetu, spakowana Linuksowym narzędziem *tar*
jessica2-i386-linux-jitr-ssadziak.tar.gz	–	JESSICA2 ze zmianami wprowadzonymi w ramach pracy, spakowane Linuksowym narzędziem *tar*
tests/	–	katalog zawierający testy użyte do badania wydajności

Bibliografia

[Ari99] Y. Aridor, M. Factor, A. Teperman, *cJVM: a Cluster Aware JVM*, Proceedings of the First Annual Workshop on Java for High-Performance Computing in conjunction with the 1999 ACM International Conference on Supercomputing (ICS), Rhodes, Greece, June 20, 1999

[Cha93] R. Chandra, A. Gupta, and J. Hennessy, *Data locality and load balancing in COOL*, In Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, USA, 1993

[Clu] *Clusterbuilder*, http://www.clusterbuilder.org/

[Fang03] Weijian Fang, Cho-Li Wang, Francis C.M. Lau, *On the Design of Global Object Space for Efficient Multi-threading Java Computing on Clusters*, Special Issue on Parallel and Distributed Scientific and Engineering Computing in the Parallel Computing, Vol.29, pp. 1563-1587, 2003

[GDB] *GDB: The GNU Project Debugger*, http://www.gnu.org/software/gdb/

[Grinder] *The Grinder, a Java Load Testing Framework*, http://grinder.sourceforge.net/,

[Java99] T. Lindholm, F. Yellin, *The Java Virtual Machine Specification, Second Edition*, Addison Wesley, 1999

[Java05] J. Gosling, B. Joy, G. Steele, G. Bracha, *The Java Language Specification, Third Edition*, http://java.sun.com/docs/books/jls/,

[Jessica] *JESSICA2: A Parallel Java Computing Engine with Thread Migration* http://www.csis.hku.hk/∼clwang/projects/JESSICA2.html

[Kaffe06] Zespół twórców Kaffe JVM, *Kaffe.org home page*, http://www.kaffe.org/

[Lai97] An-Chow Lai, Ce-Kuen Shieh, Jyh-Chang Ueng, Yih-Tzye Kok, Ling-Yang Kung, *Load Balancing in Software Distributed Shared Memory Systems*, International Performance, Computing, and Communications Conference, Arizona, 1997

[San96] L. P. P. dos Santos, *Load Distribution: a Survey*, Universidade do Minho, Portugal,

[Sch95] Christian A. Scheurer, Hans K. Scheurer, Peter Kropf, *Load Balancing Driven Process Migration*, Institute of Informatics and Applied Mathematics, University of Berne, 1995

[Shi01] Weisong Shi, Zhimin Tang, *Load Balancing in Home-based Software DSMs*, Special Issue on Parallel and Distributed Systems of International Journal of Foundation of Computer Science. World Scientific Publishing Co., USA, 2001

[Tan95] Andrew S. Tanenbaum, *Rozproszone systemy operacyjne*, Wydawnictwo Naukowe PWN, Warszawa 1997

[Vel] R. Veldema, R.A.F. Bhoedjang, H.E. Bal, *Jackal, A Compiler Based Implementation of Java for Clusters of Workstations*, Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands

[Wang04] Wenzhang Zhu, Cho-Li Wang, Weijian Fang, Francis C.M. Lau, *A New Transparent Java Thread Migration System Using Just-in-Time Recompilation*, The 16th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2004), pp. 766-771, MIT Cambridge, MA, USA, November 9-11, 2004

[Wei04] Weijian Fang, *Distributed Object Sharing for Cluster-based Java Virtual Machine*, Ph. D. thesis, University of Hong Kong, 2004

[Zhu04] Wenzhang Zhu, *Distributed Java Virtual Machine with Thread Migration* Ph. D. thesis, University of Hong Kong, 2004

[Zhu95] Weiping Zhu, C.F. Steketee, *An Experimental Study of Load Balancing on Amoeba*, School of Computer and Information Science, University of South Australia, Adelaida, 1995

[Zig03] John N Zigman, Ramesh Sankaranarayana, *Designing a Distributed JVM on a cluster*, Proceedings of the 17th European Simulation Multiconference, Nottingham, United Kingdom, June 2003