Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

mgr Marek Nowicki

Opracowanie nowych metod programowania równoległego w Javie w oparciu o paradygmat PGAS (Partitioned Global Address Space)

rozprawa doktorska

Promotor rozprawy
dr hab. Piotr Bała, prof. UW
Interdyscyplinarne Centrum Modelowania
Matematycznego i Komputerowego
Uniwersytet Warszawski

Promotor pomocniczy
dr Michał Chlebiej
Zakład Obliczeń Równoległych i Rozproszonych
Wydział Matematyki i Informatyki
Uniwersytet Mikołaja Kopernika

Styczeń 2015

Oświadczenie autora rozprawy:
oświadczam, że niniejsza rozprawa została napisana przeze mnie samodzielnie.

23 stycznia 2015 .....
data *mgr Marek Nowicki*

Oświadczenie promotorów rozprawy:
niniejsza rozprawa jest gotowa do oceny przez recenzentów

23 stycznia 2015 .....
data *dr hab. Piotr Bała, prof. UW*

.....
dr Michał Chlebiej

Abstract

New programming methods for parallel programming in Java based on the PGAS (Partitioned Global Address Space) paradigm

Scientific problem concerned in this dissertation is to develop the effective methods and tools for application parallelization for HPC systems. The main issues are related to the effective, scalable and efficient implementation of parallel algorithms in modern programming languages.

The subject of the research described in this dissertation was to develop new programming methods for parallel programming in Java language based on the PGAS (*Partitioned Global Address Space*) paradigm, and the implementation of a library for Java, which would not use external libraries. Moreover, the aim of the study was to investigate the performance and the scalability of the developed solution and the usefulness for applications parallelization.

The PCJ library (*Parallel Computations in Java*) has been created from scratch. It is based on a Partitioned Global Address Space (PGAS) model represented by Co-Array Fortran, Unified Parallel C, X10 or Titanium.

The PCJ library fully complies with Java standards therefore the programmer does not have to use additional libraries, which are not part of the standard Java distribution. The PCJ has ability to work on the multinode multicore systems hiding details of inter- and intranode communication. In the PCJ library, each task has its own local memory, and stores and accesses variables locally. Variables can be shared between tasks, and can be accessed, read and modified by other tasks. The library provides methods to perform basic operations like synchronization of tasks, get and put values in asynchronous onesided way. Additionally the library offers methods for creating groups of tasks, broadcasting and monitoring variables.

The library has been thoroughly tested. The results were compared to the MPI – current standard that is being used to create a distributed and parallel applications. Among the basic micro benchmarks, there are run time tests of synchronization, depending on the number of tasks, and also the maximum amount of data transferred per time unit between two nodes, depending on the size of the message being sent. Additionally, the results obtained using the PCJ library of larger applications, such as *RayTracer*, were compared to the results gained by the MPI.

Keywords: Parallel computing; Distributed computing; Java; Partitioned Global Address Space; PGAS; HPC; PCJ

ACM Computing Classification (rev. 2012):

- Computing methodologies → Distributed computing methodologies → Distributed programming languages
- Computing methodologies → Parallel computing methodologies → Parallel programming languages
- Theory of computation → Models of computation → Concurrency → Parallel computing models
- Theory of computation → Models of computation → Concurrency → Distributed computing models
- Theory of computation → Design and analysis of algorithms → Parallel algorithms
- Theory of computation → Design and analysis of algorithms → Distributed algorithms
- Computing methodologies → Modeling and simulation → Simulation types and techniques → Massively parallel and high-performance simulations

Streszczenie

Problem naukowy, którego dotyczy praca, to rozwój efektywnych metod i narzędzi do zrównoleglania aplikacji. Chodzi tutaj o kwestie związane z efektywną, skalowalną i wydajną implementacją algorytmów równoległych w nowoczesnych językach programowania.

Przedmiotem badań opisanych w niniejszej rozprawie doktorskiej było opracowanie nowych metod programowania równoległego w języku Java z wykorzystaniem paradygmatu PGAS (*Partitioned Global Address Space*), a także implementacja opracowanych rozwiązań w postaci biblioteki dla języka Java, która by nie wykorzystywała zewnętrznych bibliotek. Ponadto celem pracy było zbadanie opracowanych rozwiązań pod kątem wydajności i skalowalności oraz przydatności do zrównoleglania aplikacji.

W ramach niniejszej pracy doktorskiej została stworzona od podstaw biblioteka dla języka Java nazwana PCJ (*Parallel Computations in Java*). Bazuje ona na modelu obliczeń rozproszonych PGAS, który jest wykorzystywany przez takie języki programowania jak Co-Array Fortran, Unified Parallel C, X10 czy Titanium.

W trakcie projektowania biblioteki główny nacisk położono na zgodność ze standardami języka Java. PCJ została stworzona jako samodzielna biblioteka, która nie wymaga żadnych modyfikacji języka ani nie wykorzystuje żadnych dodatkowych bibliotek. Pozwala ona na pracę na systemach wielowęzłowych i wielordzeniowych, ukrywając szczegóły komunikacji wewnątrz węzła i pomiędzy węzłami przed programistą. W opisywanej bibliotece każdy wątek działa samodzielnie w swojej własnej lokalnej pamięci. Domyślnie zmienne są przechowywane i dostępne lokalnie. Niektóre zmienne mogą być współdzielone z innymi wątkami i mogą być odczytywane i modyfikowane przez inne wątki. Biblioteka dostarcza metody do wykonywania podstawowych operacji jak synchronizacja wątków, czy pobieranie i wysyłanie wartości zmiennych współdzielonych w sposób jednostronny i asynchroniczny. Ponadto biblioteka oferuje mechanizmy do tworzenia grup wątków, rozgłaszania i monitorowania zmiennych współdzielonych.

Stworzona i rozwijana w ramach pracy doktorskiej biblioteka została gruntownie przetestowana a uzyskane wyniki zostały porównane z aktualnym standardem używanym do tworzenia rozproszonych i równoległych aplikacji jakim jest MPI. Wśród podstawowych testów szybkości biblioteki znajdują się testy przedstawiające czas wykonywania synchronizacji w zależności od liczby wątków, a także maksymalna ilość danych przesłanych w jednostce czasu pomiędzy dwoma węzłami w zależności od wielkości przesyłanej wiadomości. Dodatkowo zostały skonfrontowane wyniki przedstawiające czas działania większych aplikacji, takich jak *RayTracer*, w zależności od liczby wykorzystanych rdzeni procesora, w przypadku wykorzystania biblioteki PCJ oraz z wykorzystaniem MPI.

Słowa kluczowe: Programowanie równoległe; Programowanie rozproszone; Java; Podzielona globalna przestrzeń adresowa; PGAS; HPC; PCJ

Spis treści

Wprowadzenie

W dzisiejszym świecie, w pogoni za coraz większą wydajnością komputerów, technologia doszła do granicy możliwości zwiększania częstotliwości taktowania procesora, co było głównym czynnikiem zwiększającym szybkość obliczeń. Granica ta związana jest głównie z problemem wydzielania ciepła przez procesor w trakcie wykonywania obliczeń. Chęć uzyskania dalszego przyśpieszenia działania komputera podyktowało zmianę podejścia. Na masową skalę zaczęto zwiększać w komputerze liczbę procesorów oraz liczbę jego rdzeni. W przypadku użytkowania komputera do gier, w biurze czy nawet w celach domowych, wiele rdzeni pozwala na wykorzystanie ich przez osobne programy uruchomione na komputerze. Programy te działają niezależnie i nie ma potrzeby by komunikowały się ze sobą. Dzięki takiemu rozwiązaniu komputery stają się bardziej responsywne i pozwalają na wydajniejszą pracę. Z drugiej strony, w przypadku obliczeń naukowych, aby wykorzystać wszystkie istniejące rdzenie procesora, potrzebne jest inne podejście. W tym podejściu to jeden program ma wykorzystać wiele rdzeni. Wiadomo, że niektóre operacje muszą wykonywać się po kolei, sekwencyjnie, ale są też takie, które mogą wykonywać się równolegle. Takie operacje z reguły przetwarzają niezależny zbiór danych. Po skończonych obliczeniach równoległych może nastąpić krok sekwencyjny, który zbierze obliczone w sposób równoległy dane wynikowe. Programowanie równoległe pozwala na takie stworzenie programu, w którym instrukcje, które mogą być wykonywane jednocześnie, wykonały się w taki sposób.

Język programowania Java od samego początku swego istnienia był nakierowany na pracę wielowątkową. Od pierwszych wydań istniała klasa Thread, służąca do tworzenia nowego wątku a każdy obiekt w języku Java zawierał mechanizmy pozwalające na prostą komunikację między wątkami. Ponadto w Java SE 5 wprowadzono do standardowej dystrybucji języka wiele udogodnień dla programisty chcącego wykorzystać wielowątkowość [1]. Wśród nich znalazły się między innymi semafory, zmienne atomowe i rygle (ang. *locks*).

Wzrastająca popularność języka Java jako języka dla wykonywania symulacji, spowodowała duży popyt na rozszerzenia, biblioteki i rozwiązania oparte o język Java służące do przetwarzania równoległego. Dobrym przykładem może być Parallel Java [2] czy Java Grande [3], ale one nie zostały szeroko zaadoptowane. Innym rozwiązaniem jest Titanium [4], który niestety jest trudny do użycia, ponieważ wprowadza nową

składnię do języka i jest zaimplementowany w oparciu o tłumaczenie kodu programu napisanego w zmodyfikowanym języku Java na język C, a następnie kompilację do kodu maszynowego.

Innym podejściem jest mechanizm RMI (*Remote Method Invocation*) [5, 6], który istnieje w języku Java prawie od samego początku jej istnienia. Mechanizm ten pozwala na zdalne uruchamianie metod obiektów, które mogą znajdować się na innych wirtualnych maszynach, na innych komputerach. Podobne podejście do RMI jest reprezentowane przez ProActive [7], który dostarcza bibliotekę dla języka Java definiującą zdalny dostęp do obiektów.

Powstają także specjalne biblioteki, które starają się implementować protokół MPI (*Message-Passing Interface*) [8] w języku Java lub które po prostu wiążą wywoływane metody obiektów z natywnymi operacjami zapisanymi w MPI [9, 10, 11].

Trwają także próby uruchamiania aplikacji równoległych w języku Java opartych na mechanizmie wątków Java (*Java Threads*) rozdzielonych na wielu wirtualnych maszynach Java (JVM). W tym podejściu aplikacja nie musi być modyfikowana i podział pracy jest wykonywany przez wirtualną maszynę, która dba o odpowiednie przesyłanie danych między instancjami wirtualnych maszyn Java. Warto zwrócić uwagę szczególnie na: dJVM [12], JESSICA2 [13] i Terracotta [14]. Aktualne implementacje rozproszonych wirtualnych maszyn wykazują niestety problemy z wydajnością.

Powyższy opis istniejących rozwiązań wspierających obliczenia równoległe w języku Java pokazuje, że aktualnie brakuje dobrych, prostych i efektywnych narzędzi, które mogłyby znacząco pomóc użytkownikom w napisaniu masywnie równoległych programów na aktualne i przyszłe architektury wielordzeniowe i wieloprocesorowe.

Istniejące próby stworzenia rozwiązań dla języka Java skończyły się niepowodzeniem, głownie ze względu na nieprzystosowanie ich do paradygmatu programowania w języku Java lub, jak w przypadku ProActive, ze względu na niską wydajność. Problem znalezienia optymalnych rozwiązań jest bardzo istotny ze względu na rosnące zapotrzebowanie na wykorzystanie języka Java w przetwarzaniu dużych danych na maszynach wielordzeniowych.

Problem badawczy

Problem naukowy, którego dotyczy praca, to rozwój efektywnych metod i narzędzi do zrównoleglania aplikacji. Chodzi tutaj o kwestie związane z efektywną i wydajną implementacją algorytmów równoległych w nowoczesnych językach programowania.

Jednym z podstawowych celów rozwoju HPC jest uzyskanie wydajności eksaskalowej w roku 2020 (lub kilka lat później). Problem ten jest przedmiotem wielu naukowych przedsięwzięć zarówno w zakresie badań jak też rozwiązań komercyjnych.

W szczególności od szeregu lat organizowane są konkursy w USA i w Europie (UE) na finansowanie różnych aspektów obliczeń eksaskalowych. Osiągnięcie wydajności eksaskalowych wymaga rozwiązania szeregu problemów:

1. zapewnienia wydajnych systemów obliczeniowych przy nakładanych ograniczeniach co do wykorzystywanej mocy elektrycznej,
2. zapewnienia odpowiednich narzędzi do zrównoleglania aplikacji na nowe architektury,
3. opracowania nowych, skalowalnych algorytmów dla istotnych problemów naukowych.

Wymienione problemy nie mogą być rozwiązywane w oderwaniu od siebie ze względu na istotne powiązania i ograniczenia. Przedstawiona rozprawa doktorska dotyczy głównie obszaru (2) i poświęcona jest opracowaniu nowych narzędzi do zrównoleglania kodu i zbadania ich stosowalności. Ponadto w jej ramach uwzględniono sprawy związane z budowa systemów obliczeniowych zwłaszcza w uzasadnieniu oceny uzyskanych wyników, a także kwestie algorytmów równoległych poprzez implementację wybranych algorytmów w celu testowania stworzonej biblioteki.

W przypadku obliczeń HPC podstawowe języki programowania wykorzystywane do implementacji algorytmów to FORTRAN i C/C++ i większość rozwiązań dla eksaskali przygotowywana jest dla tych języków programowania. Jeżeli pojawiają się nowe języki programowania takie jak X10 bardzo często są one mocno zbliżone do C czy FORTRANu zgodnie z zapotrzebowaniem użytkowników. Jednocześnie, zwłaszcza o obszarze analizy dużych danych (BigData) stosuje się nowe języki programowania takie jak Java. Stąd też rosnące zainteresowanie rozwiązaniami pozwalającymi na tworzenie skalowalnych aplikacji w języku Java co podkreślają opracowania UE.

Zgodnie z założeniami, tworzone rozwiązanie powinno być maksymalnie osadzone w realiach języka Java, nie powinno wprowadzać nowych, niestandardowych konstrukcji czy słów kluczowych. Ze względu na zalety związane z jednostronną komunikacją i łatwością zrównoleglania aplikacji wybrano paradygmat PGAS.

Cele pracy

Szczegółowymi celami pracy są:

- opracowanie nowych metod programowania równoległego w języku Java z wykorzystaniem paradygmatu PGAS,
- implementacja opracowanych rozwiązań w postaci biblioteki dla języka Java niewykorzystującej zewnętrznych bibliotek,

- zbadanie wydajności i skalowalności opracowanych rozwiązań,
- zbadanie przydatności opracowanych rozwiązań do zrównoleglania aplikacji.

W ramach niniejszej pracy doktorskiej została stworzona od podstaw biblioteka dla języka Java nazwana PCJ (*Parallel Computations in Java*), która odnosi się do wymienionych wyżej potrzeb. Biblioteka PCJ została stworzona od zera z wykorzystaniem najnowszej w momencie rozpoczęcia prac wersji języka Java SE 7. Bazuje ona na modelu obliczeń rozproszonych PGAS (*Partitioned Global Address Space*).

W trakcie projektowania biblioteki główny nacisk położono na zgodność ze standardami Java. PCJ została stworzona jako samodzielna biblioteka, która nie wymaga żadnych modyfikacji języka ani nie wykorzystuje żadnych dodatkowych bibliotek. Programista wykorzystujący PCJ nie jest zmuszony do dołączania bibliotek, które nie są częścią standardowej dystrybucji Java. Dodanie takich zależności mogłoby skutkować problemami z integracją rozwiązań w przypadku wykorzystania wielu różnych bibliotek.

Metodologia

W bibliotece PCJ każdy wątek działa samodzielnie w swojej własnej lokalnej pamięci. Domyślnie zmienne są przechowywane i dostępne lokalnie. Niektóre zmienne mogą być współdzielone z innymi wątkami. W PCJ nazywa się te zmienne *zmiennymi współdzielonymi*. Zmienne współdzielone przechowywane są w specjalnym *magazynie*. Każdy wątek zawiera dokładnie jeden *magazyn*. Ponadto istnieje klasa zwana *punktem startowym*. Jest to klasa, która zawiera odpowiednią metodę, od której zaczynają się obliczenia równoległe.

Wątki w PCJ są rozdzielone między wiele wirtualnych maszyn Java. Jedna wirtualna maszyna może zawierać kilka wątków PCJ. Taka architektura podyktowana jest konstrukcją nowoczesnych systemów obliczeniowych składających się z setek węzłów obliczeniowych, które z kolei zawierają po kilka lub kilkanaście rdzeni. Taka budowa hierarchii wątków wymusza wykorzystanie innych mechanizmów komunikacji wewnątrz węzła i innych przy komunikacji pomiędzy węzłami.

Dostęp do zmiennych współdzielonych jest realizowany w sposób jednostronny i asynchroniczny. Wątek, który chce przekazać wartość zmiennej do innego wątku, wykonuje odpowiednią operację *put*, w czasie której, wątek odbierający nie przerywa swoich obliczeń. Podobnie, operacja pobierania wartości z innego wątku, wykonywana przez operację *get*, również nie wymaga przerwania obliczeń na tym wątku. Przesyłanie wartości zmiennych współdzielonych realizuje się przez wywołanie odpowiedniej metody klasy PCJ z biblioteki PCJ.

Oprócz przesyłania danych, jedną z najważniejszych funkcji biblioteki do obliczeń rozproszonych jest synchronizacja wątków, zwana barierą. W PCJ synchronizacja wykonywana jest z wykorzystaniem struktury wątków tworzących pełne drzewo binarne.

Początkowa wersja biblioteki była stworzona w Interdyscyplinarnym Centrum Modelowania Matematycznego i Komputerowego (ICM UW) i na Wydziale Matematyki i Informatyki (WMiI UMK) [15, 16]. W krótkim czasie zostało zademonstrowane, że PCJ wykazuje znaczący potencjał do implementacji wysoko skalowalnych aplikacji i może skalować się do tysiąca i więcej rdzeni.

Większość aktualnie istniejących algorytmów była rozwijana w oparciu o systemy wektorowe lub równoległe, w których liczba procesorów sięgała setek a nawet tysięcy. Wielkość systemów peta- i eksaskalowych wymaga nowoczesnych algorytmów projektowanych od początku jako równoległe. Głównym zainteresowaniem jest redukcja lub nawet eliminacja synchronicznego wykonania programu. Zasadniczym problemem związanym ze skalowalnością równoległej biblioteki Java jest użycie efektywnych i skalowalnych algorytmów dla synchronizacji i komunikacji. Aktualnie używane metody, włączając w to Java Concurrency Library [17], nie są optymalne podczas używania na nowoczesnej architekturze wielordzeniowej. Tak więc, aby wykorzystać możliwości najnowszych systemów peta- i eksaskalowych, muszą zostać stworzone, zaimplementowane i przetestowane nowe algorytmy.

Stworzona i rozwijana w ramach pracy doktorskiej biblioteka została gruntownie przetestowana a uzyskane wyniki zostały porównane z aktualnym standardem używanym do tworzenia rozproszonych i równoległych aplikacji jakim jest MPI. Wśród podstawowych testów szybkości biblioteki znajdują się testy przedstawiające czas wykonywania synchronizacji w zależności od liczby wątków, a także maksymalna ilość danych przesłanych w jednostce czasu pomiędzy dwoma węzłami w zależności od wielkości przesyłanej wiadomości. Dodatkowo zostały skonfrontowane wyniki przedstawiające czas działania aplikacji specjalnie przygotowanych do testów, typu *RayTracer*, w zależności od liczby wykorzystanych rdzeni procesora, w przypadku wykorzystania biblioteki PCJ oraz z wykorzystaniem modelu MPI. Testy zostały wykonane na klastrach obliczeniowych udostępnionych przez ICM UW.

Stworzona w ramach niniejszej pracy doktorskiej biblioteka PCJ została przetestowana i opisana. Otrzymane wyniki zostały zaprezentowane na konferencjach naukowych i opublikowane. Należy podkreślić, że konferencje naukowe, na których były prezentowane wyniki charakteryzują się ostrą selekcją prac (acceptance rate na poziomie 30-35%) zgodną ze standardami IEEE. Biblioteka PCJ jest od listopada 2013 roku dostępna w Internecie, od września 2014 roku dostępny jest pełny kod źródłowy.

Wyniki pracy badawczej zaprezentowane na następujących konferencjach i szkołach:

- PRACE Spring School 2012, School for Developers of Petascale Applications, *PCJ for Parallel Computation in Java (poster)*, 16-18.05.2012, Kraków,
- PARA 2012: Workshop on State-of-the-Art in Scientific and Parallel Computing, *PCJ – New Approach for Parallel Computations in Java*, 11-13.06.2012, Helsinki, Finlandia,
- The 2012 International Conference on High Performance Computing & Simulation (HPCS 2012), *Parallel computations in Java with PCJ library*, 2-6.07.2012, Madryt, Hiszpania,
- Exascale Applications and Software Conference (EASC2013): Solving Software Challenges for Exascale, *PCJ – a Partioned Global Address Space Approach for Parallel Computations in Java*, 9-11.04.2013 r., Edynburg, Szkocja, Wielka Brytania,
- Sesja sprawozdawczo-promocyjna użytkowników KDM i POWIEW, *HPC using Java*, 12-14.05.2013, Płock,
- Java Specialists Symposium in Crete: JCrete 2013, *Parallel Computation in Java – The PCJ library*, 19-22.08.2013, Chania, Grecja,
- 7$^{\text{th}}$ International Conference on PGAS Programming Models, *The performance of the PCJ library for the massively parallel computing (poster)*, 3-4.10.2013 r., Edynburg, Szkocja, Wielka Brytania,
- Java Specialists Symposium: JCrete 2014, *Distributed Computing in Java*, 25-29.08.2014, Grecja.

Ponadto powstały także artykuły naukowe opisujące bibliotekę PCJ:

- Marek Nowicki, Piotr Bała. *Parallel computations in Java with PCJ library.* w *High Performance Computing and Simulation (HPCS 2012)*, strony 381–387, 2012.
- Marek Nowicki, Piotr Bała. *New Approach for Parallel Computations in Java.* w *PARA 2012: State-of-the-Art in Scientific and Parallel Computing*, LNCS 7782, strony 115–125, 2013.
- Marek Nowicki, Łukasz Górski, Patryk Grabrczyk, Piotr Bała. *PCJ — for high performance computing in PGAS model.* w *Proceedings of the 2014 International Conference on High Performance Computing & Simulation (HPCS 2014)*, strony 202–209, 2014.

- Rafał Kluszczyński, Marcin Stolarek, Grzegorz Marczak, Łukasz Górski, Marek Nowicki, Piotr Bała. *Development of parallel codes using PL-Grid infrastructure.* w: Cracow '14 Grid Workshop, October 27-29 2014, Krakow, Poland, strony 81–82

Biblioteka PCJ została również zgłoszona do 10. edycji prestiżowego konkursu HPC Challenge organizowanego w ramach największej konferencji poświęconej zagadnieniom HPC – Supercomputing Conference SC14. Biblioteka PCJ uzyskała nagrodę w kategorii 2 (*Best productivity*) za *Most Elegant Solution* [18]. Z tego co autorowi wiadomo, była to pierwsza polska prezentacja w ramach HPC Challenge i prawdopodobnie pierwsza prezentacja na konferencjach Supercomputing zgłoszona przez jednostki naukowe z Polski.

Struktura rozprawy

Rozprawa składa się z sześciu rozdziałów. Rozdział 1 przedstawia problematykę obliczeń równoległych i rozproszonych wraz z aktualnym stanem wiedzy z zakresu wykorzystania ugruntowanych rozwiązań takich jak MPI i OpenMP, czy rozwiązań opartych o model PGAS. Ponadto rozdział ten opisuje język Java i próby jego wykorzystania do obliczeń równoległych i rozproszonych. W rozdziale 2 przedstawione są założenia, które miały wpływ na projektowanie biblioteki PCJ a także jej architekturę i sposób działania. Następnie rozdział 3 przedstawia podstawy programowania z wykorzystaniem biblioteki PCJ. Rozdział 4 zawiera przyjętą metodykę oceny rozwiązań równoległych. Kolejne dwa rozdziały 5 i 6 przedstawiają wyniki różnych testów wydajnościowych, przy czym w rozdziale 5 pokazano wyniki podstawowych operacji biblioteki PCJ, a w rozdziale 6 opisano fragmenty aplikacji zrównoleglonych z wykorzystaniem biblioteki PCJ wraz z wynikami testów ich skalowalności. Ostatnia merytoryczna część pracy podsumowuje wyniki a także nakreśla możliwe kierunki dalszych badań i rozwoju biblioteki PCJ.

Rozdział 1

Ogólny opis problemów programowania równoległego

Istnieje szereg modeli programowania (oraz języków, które stanowią praktyczną implementację tych modeli) adresujących wyzwania programowania równoległego. Aktualnie model SPMD (*Single Program, Multiple Data*) z koncepcją przekazywania wiadomości jest modelem dominującym. Model ten wykorzystuje MPI dla komunikacji międzywęzłowej oraz sporadycznie OpenMP w celu zapewnienia równoległości wewnątrzwęzłowej. Więcej informacji na ten temat zawartych jest w podrozdziale 1.1.

Najnowsze badania pokazują, że zrównoleglanie aplikacji może być bardziej efektywne przy wykorzystaniu języków typu PGAS (*Partitioned Global Address Space*) [19]. Jednym z powodów takiego stanu rzeczy jest możliwość dostępu *implicite* do zdalnej pamięci. Wydaje się, że najbardziej ugruntowanymi językami PGAS są UPC [20] i Co-Array Fortran [21]. Ostatnio nastąpił także rozwój wielu nowych rozwiązań, wśród których najbardziej popularne to: Chapel [22], X10 [23] czy Fortress [24]. Opis tych języków przedstawiony został w podrozdziale 1.2

Powstają ponadto rozszerzenia istniejących języków jak Cilk Plus [25] czy Intel Parallel Building Blocks [26].

Język Cilk jest rozwijany od 1994 roku na MIT (*Massachusetts Institute of Technology*). Bazuje on na języku ANSI C, z dodatkiem zaledwie garstki słów kluczowych specyficznych dla Cilk. Kiedy usunie się słowa kluczowe Cilk z kodu źródłowego, w rezultacie otrzyma się poprawny program w języku C. W 2010 roku, Intel wydał jego komercyjną implementację, korzystającą z własnych kompilatorów (połączonych z kilkoma konstrukcjami dla danych równoległych), o nazwie Intel Cilk Plus. Cilk Plus jest idealny dla starszego kodu, który jeszcze nie był zrównoleglony. Aktualnie dostępne wyniki wydajnościowe sugerują, że użycie po prostu OpenMP, może być lepsze z praktycznego punktu widzenia [27].

Intel Parallel Building Blocks (PBB) jest trzyczęściowym zestawem narzędzi dla HPC (*High-performance computing*), którego główną ideą jest oprogramowanie pojedynczego wielordzeniowego węzła. PBB składa się z trzech elementów: Array Building Blocks (ArBB), Threading Building Blocks (TBB) i CilkPlus. TBB i ArBB są bibliotekami szablonów (ang. *template library*) dla języka C++, które nie rozszerzają jego składni. TBB jest przeznaczony aby wspierać zrównoleglanie zadania bez bezpośredniego zarządzania wątkami. ArBB dokonuje analizę czasu wykonania, aby wykonać optymalizacje związane ze specyficzną architekturą, takie jak wykorzystanie dostępnych sprzętowych jednostek wektorowych. Wysokopoziomowa natura TBB prawdopodobnie nie jest cechą wystarczającą do przyciągnięcia programistów HPC chcących przedłużyć żywotność kodu lub udoskonalić istniejące kody [27].

Jak stwierdzono w raporcie LLNL (*Lawrence Livermore National Laboratory*) [27] kilka z powyżej wymienionych rozwiązań lepiej wykorzystuje zasoby węzła, niż obecny model SPMD wykorzystujący MPI i OpenMP. Inne przedstawiają całkowicie nowe sposoby spojrzenia na problemy symulacji dużej skali, a jeszcze inne przyczynią się do powstawania zwięźlejszych i bardziej zrozumiałych kodów. Żaden jednak nie jest tak silny i stabilny jak obecne modele takie jak MPI czy OpenMP.

1.1 Ugruntowane modele MPI i OpenMP

Wydaje się, że najbardziej rozpowszechnionymi modelami umożliwiającymi tworzenie aplikacji równoległych są MPI (*Message-Passing Interface*) [8] i OpenMP (*Open Multi-Processing*) [28, 29]. Pierwszy jest modelem programowania z wykorzystaniem przesyłania komunikatów, drugi wykorzystuje pamięć wspólną.

Główną zaletą programowania w tych standardowych modelach, czyli MPI i OpenMP jest ich długa obecność na rynku. Programowanie z wykorzystaniem modelu MPI stało się podstawowym narzędziem nauki programowania równoległego na studiach a OpenMP pokazywany jest jako jeden z nielicznych przykładów programowania wielowątkowego.

1.1.1 MPI

MPI (*Message-Passing Interface*) [8] jest dominującym standardem opartym o przesyłanie komunikatów. Jego podstawowe implementacje istnieją dla języków takich jak C, C++ i Fortran, a dokumentacja przedstawia API (*Application Programming Interface*) dla tych języków. Istnieją także próby implementacji MPI w innych językach. W przypadku języka Java wykorzystano głównie tzw. *wrappery*, czyli metody, które odwoływały się poprzez JNI (*Java Native Interface*) do metod z lokalnie zainstalowanej biblioteki MPI [9, 30].

MPI wykorzystuje model SPMD (*Single Program, Multiple Data*). Procesy działające w ramach uruchomienia aplikacji są od siebie odseparowane a komunikacja zachodzi jedynie poprzez wysyłanie wiadomości między procesami, gdzie jeden proces nazywany jest nadawcą, a drugi nosi nazwę odbiorcy wiadomości. W ten sposób w przypadku MPI można mówić nie tylko o programowaniu równoległym ale i rozproszonym, gdyż procesy mogą, i zwykle nie są, ulokowane na jednym fizycznym węźle.

Historia MPI

Standard MPI opisuje między innymi sposób przesyłania komunikatów w przypadku połączeń punkt-punkt i komunikacji jednostronnej, operacje kolektywne, koncepcję grup i topologii. Powstał, by ułatwić programowanie z użyciem modelu przekazywania wiadomości. Jego głównym założeniem było stworzenie szeroko używanego standardu, który byłby praktyczny, przenośny, efektywny i elastyczny.

Historia MPI sięga roku 1992, w którym rozpoczęto pracę nad pierwszą wersją standardu, której szkic przedstawiono na konferencji *Supercomputing 93* w listopadzie 1993 roku, a oficjalnie finalną wersję standardu MPI-1.0 wydano w maju 1994 roku. Następnie w marcu 1995 roku rozpoczęto proces rozszerzania i korekt standardu, co już po kilku miesiącach, w czerwcu 1995 roku, zaowocowało wydaniem wersji MPI-1.1. Najnowszą wersją standardu w serii MPI-1 jest wersja MPI-1.3 wydana w 2008 roku, która nie jest już rozwijana. Od 1995 roku pracowano także nad serią MPI-2, która miała zawierać wiele nowości, ale nadal bazowała na wersji MPI-1. Pierwszą wersję serii MPI-2, czyli MPI-2.0, wydano w lipcu 1997 roku. Wśród głównych funkcjonalności nowego standardu były: procesy dynamiczne, komunikacja jednostronna i równoległe operacje wejścia/wyjścia. Aktualną wersją standardu MPI jest wersja MPI-3.0 wydana 21 września 2012 roku zaczynająca serię MPI-3. Aktualnie używany standard MPI bazuje więc na wersji sprzed około 20 lat.

Wersje MPI są kompatybilne wstecz, czyli poprawne aplikacje stworzone w oparciu o MPI-1.3 będą również poprawne w MPI-2 a także w wersji MPI-3.

Podstawy MPI

Standard MPI zawiera API dla języków C, C++ i Fortran, jednakże w dalszej części pracy prezentowane będą przykłady jedynie z użyciem składni języka C, gdyż składnia w pozostałych językach jest analogiczna.

Programy napisane w MPI muszą znajdować się pomiędzy funkcjami `MPI_Init(int *argc, char ***argv)` a `MPI_Finalize()`. Wszelkie użycie funkcji MPI poza przedziałem wyznaczonym przez te funkcje będzie skutkować błędem wykonania.

Każdy proces w MPI posiada swój unikatowy identyfikator będący liczbą całkowitą nie mniejszą od 0. Procesy mogą sprawdzić swój identyfikator za pomocą funkcji:

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank)
```

`MPI_COMM_WORLD` jest to stała wprowadzona przez bibliotekę MPI i oznacza komunikator, który zawiera wszystkie procesy biorące udział w obliczeniach. We wszystkich funkcjach, które wymagają komunikacji między procesami, wymagane jest podanie komunikatora. Wynik operacji, czyli identyfikator procesu w komunikatorze, zostanie zapisany w zmiennej `rank`. Liczbę procesów biorących udział w obliczeniach można sprawdzić wykonując funkcję:

```
MPI_Comm_size(MPI_COMM_WORLD, &nprocs)
```

Wynik, czyli liczba procesów w komunikatorze `MPI_COMM_WORLD`, zostanie zapisany w zmiennej `nprocs`.

Synchronizacja między procesami możliwa jest poprzez zastosowanie bariery wywołując przez każdy proces w ramach komunikatora funkcję:

```
int MPI_Barrier(MPI_Comm comm)
```

Przesyłanie danych między procesami zawsze wymaga wykonania akcji przez wszystkie wątki zaangażowane w przekazywanie danych. W przypadku przesłania danych punkt-punkt, wątek wysyłający dane i wątek odbierający dane muszą wykonać odpowiednie operacje. W najprostszej wersji, wątek wysyłający dane wywołuje funkcję:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
```

a wątek odbierający korzysta z funkcji:

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm,
             MPI_Status *status)
```

W obu funkcjach występuje wiele parametrów, które oznaczają:

- `buf` – bufor; adres w pamięci, z którego dane będą odczytywane i do którego dane będą zapisywane, w zależności, czy proces jest wysyłający czy odbierający dane,
- `count` – liczba przesyłanych elementów,
- `datatype` – typ przesyłanych danych,
- `dest/source` – identyfikator procesu docelowego i źródłowego, czyli procesu odbierającego i wysyłającego dane; może być również symbolem wieloznacznym, pozwalającym odbierać dane od dowolnego procesu,
- `tag` – znacznik, numer rozróżniający przesyłane dane; musi on się zgadzać w procesie odbierającym i wysyłającym dane; może być także symbolem

wieloznacznym, pozwalając odebrać dane niezależnie od znacznika użytego przez proces wysyłający dane,

- `comm` – wykorzystywany komunikator,
- `status` – przechowuje informacje o odebranej wiadomości, takie jak informacje o powodzeniu odbierania danych, numerze procesu wysyłającego dane i wartości znacznika.

Oprócz podstawowej wersji funkcji wysyłającej dane (`MPI_Send`) istnieje więcej funkcji realizujących przesłanie danych, różniących się sposobem działania:

- `MPI_Ssend` – funkcja synchroniczna, blokuje do momentu, aż proces odbierający dane wywoła funkcję `MPI_Recv`,
- `MPI_Bsend` – funkcja, która korzysta z osobnego bufora do przechowywania danych do wysłania i kończy działanie, gdy dane do wysłania zostaną tam skopiowane,
- `MPI_Rsend` – funkcja, która zakłada, że proces odbierający dane wywołał funkcję `MPI_Recv`, w przeciwnym wypadku zwróci błąd.

Wszystkie te funkcje są blokujące – program może kontynuować obliczenia, dopiero gdy może bezpiecznie modyfikować dane w buforze. Niezależnie od wyboru funkcji wysyłania, proces odbierający dane wykorzystuje w tym celu przedstawioną funkcję `MPI_Recv`.

MPI pozwala również wykonywać kolektywne operacje na danych, takie jak rozgłaszanie danych (`MPI_Bcast`), rozdzielenie danych między wątki (`MPI_Scatter`), zebranie danych od innych wątków (`MPI_Gather`) czy redukcję danych (`MPI_Reduce`). Wszystkie te operacje angażują wszystkie procesy w danym komunikatorze – każdy musi wywołać odpowiednią funkcję, a dodatkowo następuje oczekiwanie na wszystkie procesy w komunikatorze, by wykonać zleconą operację. Stąd, jeśli jeden proces wolniej wykonuje obliczenia, wszystkie procesy blokują swoje wykonanie – każda z wymienionych operacji jest *implicite* barierą.

Komunikacja nieblokująca

W MPI-2 wprowadzono dodatkowe funkcje pozwalające na nieblokujące operacje przesyłania danych punkt-punkt, które otrzymały przedrostek `MPI_I`: `MPI_Isend`, `MPI_Issend`, `MPI_Ibsend`, `MPI_Irsend` oraz `MPI_Irecv`. Bazują one na swoich odpowiednikach blokujących. Natomiast w MPI-3 wersji nieblokujących doczekały się funkcje wykonujące operacje kolektywne, czyli między innymi: `MPI_Ibarrier`, `MPI_Ibcast`, `MPI_Igather`, `MPI_Iscatter`, `MPI_Ireduce`.

Wszystkie wymienione wyżej metody działają w zbliżony sposób do metod blokujących, z tą różnicą, że dodano jako ostatni parametr:

```
MPI_Request *request
```

a w przypadku funkcji `MPI_Irecv` zamiast dodać to zmodyfikowano typ ostatniego parametru z `MPI_Status` na `MPI_Request`.

Parametr `MPI_Request` pozwala na sprawdzenie, czy nieblokująca operacja zakończyła się, czyli odpowiada na pytanie, czy gdyby została uruchomiona funkcja korzystająca z komunikacji blokującej, to czy nastąpiłoby wyjście z tej funkcji, czy jeszcze nie. Ponadto istnieje możliwość, by zablokować obliczenia, aż zakończy się operacja nieblokująca.

Nieblokująca operacja bariery `MPI_Ibarrier` polega na tym, że każdy proces, wykonujący tę operację, tylko informuje, że dotarł do miejsca bariery, ale natychmiast wraca z tej funkcji i może dalej wykonywać obliczenia. Gdy dojdzie do momentu, w którym nie będzie możliwa dalsza praca, bez wiedzy, że pozostałe procesy również dotarły do miejsca bariery, może wykonać operację `MPI_Wait`, która spowoduje zatrzymanie wykonania, do czasu, aż wszystkie procesy wykonają operację bariery, czyli zakończy się operacja bariery.

Wady programowania z wykorzystaniem MPI

Mało mówi się o wadach programowania z wykorzystaniem MPI, gdyż jest to podstawowe narzędzie służące do nauki programowania równoległego i rozproszonego. Niemniej jednak, model ten zawiera pewne wady, które zostaną poniżej pokrótce opisane.

Jedną z głównych wad programowania w modelu MPI jest trudność w wykrywaniu błędów. Każde wysłanie danych z jednego węzła do drugiego wymaga podania znacznika. Węzeł chcący odebrać dane musi wywołać odpowiednią funkcję i podać dokładnie taki sam znacznik jaki został podany przez wątek wysyłający. Możliwe jest również użycie znacznika wieloznacznego, ale wówczas odebrane zostaną pierwsze dostępne dane. Jest to główną przyczyną błędów w działaniu aplikacji.

Dodatkowo standardowe mechanizmy przesyłania danych między węzłami są blokujące – węzeł odbierający dane zatrzymuje swoje wykonanie, do czasu, aż węzeł wysyłający, wyśle dane. Podobnie węzeł wysyłający może zakończyć funkcję wysyłania danych dopiero gdy węzeł odbierający wykona odpowiednią funkcję odbierającą. Takie podejście do przesyłania danych powoduje, że programista musi spędzić dużo czasu na odpowiednie przygotowanie operacji wysyłania i odebrania danych. Powoduje to także dodatkowe problemy ze znajdowaniem błędów implementacji. Standard MPI-2.0 wprowadza wprawdzie komunikację nieblokującą, ale nadal wymaga ona akcji od obu wątków – odbierającego i wysyłającego, a także podania tych samych danych co w przypadku przesyłania danych w sposób blokujący.

Inną słabością modelu MPI jest to, że jest on głównie używany przy użyciu języków programowania takich jak Fortran, C czy C++, które na poziomie kompilatora nie zapewniają poprawności kodu źródłowego. Co prawda sprawdzana jest poprawność składni i wykonywane są proste testy wykrywające na przykład użycie niezainicjowanej zmiennej, ale kompilator nie wykazuje błędu ani nawet nie informuje użytkownika o tym, że program przez niego pisany, usiłuje uzyskać dostęp do pamięci, do której proces nie powinien mieć dostępu, czy której proces jeszcze nie zaalokował. To powoduje, że znalezienie błędu działania w tych językach jest szczególnie uciążliwe. Ostrzeżenia o problematycznych miejscach w kodzie programu zapewnia między innymi język Java.

Ponadto rozwiązania napisane z wykorzystaniem języków takich jak Fortran, C czy C++, których kompilatory tworzą pliki wykonywalne, w przypadku przenoszenia obliczeń na inną architekturę docelową, mogą wymagać ponownego przekompilowania kodu źródłowego z czym mogą wiązać się dodatkowe problemy. W przypadku języka Java wystarczające jest, by na architekturze docelowej zainstalowana była wirtualna maszyna Java.

1.1.2 OpenMP

OpenMP (*Open Multi-Processing*) [28, 29] jest standardem definiującym dyrektywy kompilatora, funkcje biblioteczne i zmienne środowiskowe pozwalający na łatwe zrównoleglenie programów tworzonych w językach C, C++ i Fortran.

Historia wersji OpenMP

Pierwsza wersja specyfikacji OpenMP powstała w październiku 1997 roku na potrzeby języka Fortran. Rok później powstała pierwsza wersja standardu dla języków C i C++. Kolejna wersja specyfikacji również wydana została osobno dla języka Fortran i dla C/C++. Dopiero w maju 2005 roku powstała wersja 2.5, która łączyła te trzy języki programowania. Aktualnie najnowszą wersją OpenMP jest wersja 4.0 wydana w lipcu 2013 roku [29].

Podstawy OpenMP

Nauka programowania w OpenMP jest łatwa i szybko można nauczyć się podstaw zrównoleglania aplikacji. Po prostu, w istniejącym kodzie źródłowym programu dodaje się zdefiniowane przez standard OpenMP dyrektywy. Dyrektywy te pokazują między innymi:

- kod, który ma być wykonywany równolegle,
- kod, który ma być rozdystrybuowany pomiędzy wątki,
- kod, który ma być wykonany tylko przez jeden wątek,

- gdzie ma nastąpić bariera.

Podobnie jak w przypadku MPI, w trakcie omawiania OpenMP w dalszej części pracy prezentowane będą przykłady jedynie z użyciem składni języka C. Składnia w pozostałych językach jest analogiczna.

Wszystkie dyrektywy OpenMP składają się z następujących elementów:

```
#pragma omp <dyrektywa> [klauzule]
```

Najprostszym sposobem na wykonanie kodu w sposób równoległy jest zastosowanie dyrektywy `parallel` wraz z pętlą `for`, co pokazano na listingu 1.1.

```
1 #pragma omp parallel for
2 for (i = 0; i < n; ++i)
3   y[i] = a[i] * x + b[i];
```

Listing 1.1: *Dyrektywa* `parallel` *powiązana z pętlą* `for`

Dyrektywa `parallel` pozwala na uruchamianie danego bloku instrukcji w sposób równoległy. Ma ona wiele możliwych klauzul do wykorzystania. Oprócz przedstawionej klauzuli `for`, która powoduje, że poszczególne iteracje pętli będą wykonywane w sposób równoległy, istnieją między innymi klauzule dotyczące widoczności zmiennych (m.in. `private`, `firstprivate`, `shared`), po których następuje lista nazw zmiennych danego typu, klauzula stopnia równoległości (`num_threads`), czyli liczba wątków, na których dany blok instrukcji ma być uruchomiony, a także klauzula `reduction`, która pozwala w prosty sposób wykonać operację redukcji.

Każdy wątek może sprawdzić ile wątków zostało uruchomionych w ramach danego bloku kodu za pomocą funkcji:

```
int omp_get_num_threads()
```

Ponadto wszystkie wątki mają również możliwość sprawdzenia, jaki numer został im nadany w ramach uruchomienia równoległego przez wywołanie funkcji:

```
int omp_get_thread_num()
```

Liczba zwracana przez funkcję `omp_get_thread_num()` jest liczbą całkowitą z przedziału od 0 do liczby o jeden mniejszej niż aktualna liczba wątków w grupie wykonującej kod równoległy. W przypadku, gdyby funkcja ta została uruchomiona z kodu sekwencyjnego, zawsze zwracać będzie wartość 0. W przypadku bloku kodu równoległego wynik 0 oznacza mistrza grupy wątków, czyli główny wątek.

Zmienna opisana przez klauzulę `private` oznacza, że każdy wątek posiada swoją kopię zmiennej. Klauzula `firstprivate` oznacza, że wątki posiadają swoją kopię zmiennej, ale zostanie ona zainicjalizowana za pomocą wartości, która wcześniej była przypisana do zmiennej o tej nazwie. Zmienna oznaczona przez `shared` jest zmienną współdzieloną między wszystkie wątki wykonujące operacje w danym bloku. W tym

```
1 #pragma omp parallel for private(i) \
2                           shared(a, b, c) \
3                           reduction(+:sum)
```

Listing 1.2: *Różne klauzule dołączone do dyrektywy* `parallel`

przypadku zadaniem programisty jest zadbanie o to, by dostęp do zmiennej był odpowiednio synchronizowany. Przykład przedstawiony na listingu 1.2 pokazuje w jaki sposób można wykorzystać różne klauzule w dyrektywie `parallel`.

```
1 #pragma omp single nowait
2 {
3     printf("Suma: %d\n", sum);
4 }
```

Listing 1.3: *Wykorzystanie klauzuli* `single`

Inną ważną dyrektywą jest dyrektywa `single`, która pozwala na uruchomienie bloku kodu tylko przez jeden wątek. Pozostałe wątki w tym czasie czekają w niejawnej barierze na końcu bloku. Możliwa jest zmiana tego sposobu działania przez użycie klauzuli `nowait`. Przykład zastosowania dyrektywy `single` przedstawiony na listingu 1.3, wyświetla na ekranie wartość sumy, ale tylko pojedynczego wątku.

```
1 #pragma omp master
2 {
3     printf("Suma: %d\n", sum);
4 }
5 #pragma omp barrier
```

Listing 1.4: *Dyrektywa* `master` *i operacja bariery*

Podobną dyrektywą do `single` jest dyrektywa `master`, która powoduje uruchomienie bloku kodu jedynie przez główny wątek. Dodatkowo w tym przypadku nie istnieje niejawna bariera na końcu bloku. W przypadku chęci dodania jawnej bariery, można skorzystać z dyrektywy `barrier`, która oznacza punkt synchronizacji wszystkich wątków wykonujących obliczenia w sposób równoległy. W ten sposób można połączyć obie dyrektywy i wyświetlić zawartość zmiennej `sum` przez wątek główny, przy czym, pozostałe wątki czekają, aż wątek główny wypisze tekst na ekranie. Przykład implementacji przedstawiony jest na listingu 1.4.

Ostatnią przedstawioną dyrektywą jest dyrektywa `critical`, która ogranicza uruchomienie powiązanego bloku instrukcji (zwanego w tym przypadku *regionem krytycznym*) tylko do jednego wątku w danym momencie. Pozostałe wątki, gdy dojdą do tego bloku instrukcji, czekają przed nim, by pojedynczo wykonywać go, gdy tylko stanie

```
1      int sum = 0;
2      int mysum;
3  #pragma omp parallel num_threads(16) private(mysum)
       ↪ shared(sum) default none
4      {
5          mysum = ...;
6  #pragma omp critical
7          {
8              sum = sum + mysum;
9          }
10     }
11     printf("Suma: %d\n", sum);
```

Listing 1.5: *Dyrektywa* `critical` *i* region krytyczny

się on wolny. W przeciwieństwie do `single` i `master`, blok ten wykonają wszystkie wątki, które do niego dojdą, ale specyfikacja OpenMP nie definiuje kolejności wykonywania tego bloku, to znaczy nie ma ustalonej kolejności wchodzenia wątków w region krytyczny.

Listing 1.5 przedstawia sposób zastosowania dyrektywy `critical` w szerszym kontekście. Każdy z 16 wątków oblicza wartość `mysum`, po czym wszystkie, po kolei, modyfikują współdzieloną zmienną `sum` w sposób bezpieczny, to znaczy taki, w którym nie występuje sytuacja wyścigu (ang. *race condition*), gdyż dostęp do zapisu wartości zmiennej współdzielonej jest ograniczony tylko do jednego wątku, przy czym odczytywana jest najświeższa wartość modyfikowanej zmiennej. Następnie następuje wykonanie ukrytej bariery dodawanej zawsze na koniec bloku *parallel* i wypisanie policzonej wartości przez wątek główny.

Opis wszystkich dostępnych dyrektyw i powiązanych z nimi klauzul dostępny jest w specyfikacji OpenMP [29].

Wady OpenMP

Główną wadą wykorzystania OpenMP jest możliwość skalowalności jedynie w obrębie jednego systemu komputerowego posiadającego wspólną pamięć, czyli na pojedynczym komputerze, węźle lub systemie z architekturą z pamięcią wspólną. Z tego powodu, aby zapewnić skalowalność aplikacji na systemie z wieloma węzłami zawierającymi procesy składające się z wielu rdzeni potrzebne jest wykorzystanie jednocześnie modelu MPI i OpenMP.

OpenMP ukrywa przed użytkownikiem sposób zarządzania pamięcią, co z jednej strony ułatwia programowanie, z drugiej może prowadzić do tworzenia mało wydajnego kodu. Generalnie, zrównoleglenie bardziej skomplikowanego kodu okazuję się nietrywialne czy wręcz trudne.

Kolejną wadą OpenMP jest zrównoleglanie na poziomie pojedynczych pętli czy niewielkich fragmentów kodu co prowadzi do konieczności częstego wykorzystywania barier i nie pozwala na uzyskanie wysokiej skalowalności kodu. Implementacja bardziej skomplikowanych algorytmów jest również utrudniona.

1.2 Model PGAS

Paradygmat programowania w modelu PGAS (*Partitioned Global Address Space*) odnosi się zarówno do danych jak i do sposobu wykonywania aplikacji. Wykorzystanie tych informacji daje szanse na znaczne polepszenie wydajności rozumianej w standardowy sposób, czyli wydajności związanej z czasem wykonania aplikacji, a także zwiększenie produktywności programistów, tworzących aplikacje na coraz powszechniejsze architektury wielordzeniowe. Model programowania PGAS stara się osiągnąć powyższe cele poprzez [31]:

- programowanie w ujęciu lokalnym z rozróżnieniem danych lokalnych i zdalnych;
- globalną przestrzeń adresową, która jest bezpośrednio dostępna przez każdy proces;
- komunikację związaną z referencją do zdalnych danych ukrytą przed programistą;
- komunikację jednokierunkową pomiędzy procesami dla lepszej wydajności;
- wsparcie dla rozproszonych struktur danych.

Może wydawać się, że model PGAS jest podejściem dość podobnym do OpenMP. Jednak nawet bez wgłębiania się w szczegóły widać znaczące różnice. Przykładowo w OpenMP poza pewnymi wybranymi zmiennymi oznaczonymi jako prywatne, większość zmiennych domyślnie ma status zmiennych współdzielonych. Dodatkowo OpenMP został zaprojektowany dla komputerów lub maszyn ze współdzieloną pamięcią, natomiast w modelu PGAS nie ma takich ograniczeń. Ponadto w modelu PGAS, jak to wyżej wspomniano, programuje się w ujęciu lokalnym, gdzie każdy proces ma dostęp do swoich prywatnych zmiennych, a dane współdzielone są szczególnie oznaczone. W OpenMP podstawą jest globalny zakres zmiennych i rozdział pracy – najczęściej na poziomie najbardziej zagnieżdżonej pętli. Model PGAS pozwala na implementację praktycznie dowolnego scenariusza zrównoleglania obliczeń. Kolejną znaczącą różnicą jest to, iż OpenMP bazuje na dynamicznym tworzeniu nowych wątków, co nie występuje w podstawowej wersji modelu PGAS wykorzystującej model programistyczny SPMD (ang. *Single Program, Multiple Data*), w którym ilość procesów jest ustalona w momencie rozpoczęcia działania programu równoległego [32].

Również w porównaniu do MPI model PGAS wykazuje znaczące różnice. W przypadku MPI zasadniczo nie ma wspólnej pamięci, a przekazywanie danych

odbywa się za pomocą przekazywania wiadomości, które odbiorca musi *explicite* odebrać. W modelu PGAS przekazywanie danych odbywa się przez odczyt i zapis we wspólnej globalnej przestrzeni adresowej, dostępnej dla każdego procesu. Dostęp do pamięci następuje bez jawnego udziału innego procesu w celu przekazania wiadomości.

Aktualnie powszechnie dostępne stają się następujące języki PGAS: Unified Parallel C, Co-Array Fortran oraz Titanium.

1.2.1 Unified Parallel C

Unified Parallel C (UPC) jest rozszerzeniem języka ANSI C uzupełniającym go o możliwość programowania w modelu PGAS z wykorzystaniem techniki SPMD (ang. *Single Program, Multiple Data*). Technika SPMD polega na tym, że ten sam pojedynczy program jest uruchamiany równolegle wiele razy, jego instancje mogą się ze sobą komunikować, a każda jego kopia posiada własny zbiór danych.

Specyfikacja języka UPC dodaje do języka ANSI C dodatkowe słowa kluczowe: `shared`, `strict`, `relaxed` i `upc_forall`, specjalne stałe: `THREADS` i `MYTHREAD` a także wiele funkcji, m.in. `upc_notify`, `upc_wait` i `upc_barrier`.

Stałe `THREADS` i `MYTHREAD` obrazują odpowiednio liczbę wątków UPC w aktualnym wykonaniu programu oraz indeks konkretnego wątku liczony od 0 do THREADS − 1. Stała `THREADS` jest zwłaszcza używana przy definiowaniu zmiennych współdzielonych przez wątki.

Słowo kluczowe `shared` służy do oznaczenia zmiennych jako współdzielone, natomiast słowa kluczowe `strict` i `relaxed` dodane do definicji zmiennych współdzielonych pozwalają dedykowanemu kompilatorowi na pewne optymalizacje związane z przenoszeniem kodu lub widocznością zmian w zmiennej współdzielonej przez pozostałe wątki. Deklaracja zmiennych współdzielonych informuje ile zmiennych danego typu znajduje się w całym systemie. Dystrybucją zmiennych współdzielonych do poszczególnych wątków zajmuje się UPC. Przykład zmiennej współdzielonej w UPC jest przedstawiony na Listingu 1.6.

```
1    shared int s[THREADS];
2    s[MYTHREAD] = MYTHREAD;
```

Listing 1.6: *Przykład zmiennej współdzielonej w UPC*

Możliwe jest stworzenie pojedynczej zmiennej współdzielonej przez wszystkie wątki bez precyzyjnego definiowana jej lokalizacji. Jest to szczególnie przydatne w przypadku systemów z prawdziwą pamięcią współdzieloną, dzięki czemu zmniejsza się jej wykorzystanie. Jeśli jednak programista chciałby, by jedna zmienna współdzielona była przechowywana we wszystkich wątkach, taka zmienna musi zostać zadeklarowana jako tablica wymiaru `THREADS`.

Oprócz tworzenia pojedynczych zmiennych współdzielonych oraz tablic, UPC wprowadza także możliwość tworzenia lokalnych wskaźników do współdzielonej pamięci, współdzielonych wskaźników do lokalnej pamięci wątku i współdzielonych wskaźników do współdzielonej pamięci.

Zadaniem ostatniego wypisanego słowa kluczowego, `upc_forall`, jest ułatwienie tworzenia pętli równoległych, w których każdy wątek wykonuje swoją własną iterację na przydzielonej pamięci. Wykorzystanie `upc_forall` obłożone jest wieloma obostrzeniami i ograniczeniami.

Synchronizacja między wątkami możliwa jest przez użycie pary funkcji `upc_notify` i `upc_wait` lub pojedynczej funkcji `upc_barrier`. Para funkcji `upc_notify` i `upc_wait` pozwala na bardziej wydajne wykonanie zleconego zdania, gdyż pomiędzy tymi dwoma wywołaniami możliwe jest wykonanie lokalnego kodu niemającego dostępu do pamięci współdzielonej. Funkcja `upc_barrier` działa jako bezpośrednio następujące po sobie wywołania funkcji: `upc_notify` i `upc_wait`.

Kompilator języka UPC o otwartym źródle jest rozwijany przez LBNL (*Lawrence Berkeley National Laboratory*) i UC Berkeley (*University of California, Berkeley*) [20]. Istnieje również implementacja GNU [33], która wykorzystuje, bazującą na GASNet [34], bibliotekę uruchomieniową UPC stworzoną przez LBNL.

Więcej informacji na temat UPC a także testy wydajności można zobaczyć w pracach [35, 36, 37, 38, 39].

1.2.2 Co-Array Fortran

Co-Array Fortran (CAF) jest rozszerzeniem Fortran 95/2003 stworzonym przez Roberta Numricha i Johna Reida w Cray Inc. w latach 90-tych XX wieku, służącym do przetwarzania równoległego [40]. W maju 2005 roku komitet ISO Fortran Committee zadecydował, by nowopowstający standard Fortran 2008 (ISO/IEC 1539-1:2010) zawierał CoArray (równoległe tablice – ang. *concurrent arrays*). Składnia w standardzie Fortran 2008 jest jednak nieco inna niż w oryginalnej propozycji CAF [41].

Co-Array Fortran, podobnie jak to ma miejsce w przypadku UPC, wykorzystuje model programistyczny SPMD – pojedynczy program uruchamiany jest w wielu instancjach, w których każda może pracować na innych danych. Każda instancja w przypadku CAF nazywana jest obrazem (ang. *image*). Każdy obraz posiada swój indywidualny numer – indeks obrazu. W przeciwieństwie do UPC, gdzie indeksy są kolejnymi liczbami naturalnymi od 0, w CAF indeksy zaczynają się od 1. Program może odczytać numer obrazu wywołując wewnętrzną funkcję `this_image()`, natomiast odczytanie liczby obrazów możliwe jest przez wywołanie wewnętrznej funkcji `num_images()`.

Deklaracja zmiennych współdzielonych w CAF składa się z dwóch części – części lokalnej oraz części współdzielonej. Część współdzielona modyfikuje zmienną do postaci tablicy równoległej (ang. *concurrent array – co-array*). Nie ma możliwości zadeklarowania pojedynczej zmiennej jako współdzielona jak to ma miejsce w przypadku UPC. W celu odróżnienia wymiarów lokalnych tablic od wymiarów współdzielonych (*co-dimensions*), wymiary współdzielone zapisane są w nawiasach kwadratowych. Ostatnim wymiarem tablicy współdzielonej jest zawsze wymiar `[*]` oznaczający zmienną wielkość w zależności od liczby obrazów, czyli od wartości `num_images()`. Odwołanie do danych zawartych w innym obrazie następuje poprzez podanie jego numeru jako indeksu ostatniego wymiaru w nawiasie kwadratowym.

Otwartoźródłowe kompilatory wspierające CoArray są rozwijane w Rice University oraz University of Houston. CAF był z sukcesem używany do zrównoleglania wysokoskalowych aplikacji naukowych [42, 43]. Porównanie wydajności UPC i CAF jest dostępne w pracy [38].

Rozszerzenie Co-Array Fortran

Standard rozszerzenia Co-Array Fortran pozwala kompilatorowi na optymalizację wykonania programów przez przechowywanie danych zapisanych w zmiennych współdzielonych w rejestrach, pamięci podręcznej lub innej pamięci tymczasowej. W związku z tym zmiany wprowadzane przez zdalny obraz mogą nie być widoczne w lokalnym obrazie a nawet mogą jeszcze nie być dokonane. Analogicznie zmiany wprowadzane przez lokalny obraz w zmiennych współdzielonych mogą nie być widoczne przez zdalne obrazy. W celu odświeżenia pamięci podręcznej program może wywołać wewnętrzną procedurę `sync_memory()`. Zmiany wprowadzane przez jeden wątek mogą nie być widoczne dla innych przed wywołaniem procedury `sync_memory()`, podobnie, aby inny wątek mógł zobaczyć wprowadzoną zmianę również musi wywołać procedurę `sync_memory()`.

Programista korzystający z CAF nie powinien zakładać jak długo i w jakiej kolejności wykonuje się kod poszczególnych części programu na różnych obrazach. Dzięki temu kompilator ma możliwość wprowadzania dodatkowych optymalizacji. Jedynym mechanizmem pozwalającym programiście na upewnienie się, że dana część programu została wykonana na obrazach, jest mechanizm synchronizacji. W rozszerzeniu Co-Array Fortran istnieją dwie wewnętrzne procedury służące do synchronizacji: `sync_team()` i `sync_all()`. Parametrem `sync_team()` jest albo pojedynczy indeks obrazu, z którym ma nastąpić synchronizacja obrazu wykonującego tę procedurę, albo lista indeksów obrazów do synchronizacji. Procedura `sync_all()` jest szczególnym przypadkiem procedury `sync_team()`, w której synchronizacja obejmuje wszystkie obrazy. Obie procedury służące do synchronizacji wywołują wewnętrznie `sync_memory()`, więc nie jest konieczne dodatkowe odświeżanie pamięci.

Wśród innych przydatnych funkcjonalności rozszerzenia Co-Array Fortran znajduje się mechanizm sekcji krytycznej. Jest to mechanizm służący do ograniczenia liczby obrazów, które mogą równolegle wykonywać kod sekcji krytycznej, do pojedynczego obrazu. Początek sekcji krytycznej oznaczony jest wywołaniem procedury `start_critical()`, a koniec przez wywołanie procedury `end_critical()`. Podobnie jak procedury `sync_team()` i `sync_all()`, `start_critical()` i `end_critical()` mają dodatkowo taki sam efekt jak wywołanie procedury `sync_memory()`.

Standard Fortran 2008

Występuje kilka zmian jakie zaszły w trakcie włączania rozszerzenia Co-Array Fortran do standardu Fortran 2008. Jedną z głównych jest zmiana nazwy procedury `sync_memory()` na `flush_memory()`. Inną zmianą jest syntaktyczna zmiana wejścia i wyjścia z sekcji krytycznej – w standardzie wprowadzono konstrukcję `critical` oznaczającą początek sekcji krytycznej (zamiast wywoływania procedury `start_critical()`) a jej zakończenie oznacza się przez `end critical` (zamiast wywoływania procedury `end_critical()`).

Bardzo istotną zmianą jest dodanie większej liczby procedur i funkcji synchronizacyjnych. Nowe procedury i funkcje to: `notify_team()`, `ready_team()` i `wait_team()`. Procedury te pozwalają na asynchroniczne powiadamianie o zakończeniu pewnego etapu obliczeń i, podobnie jak to miało miejsce w UPC, rozpoczęciu wykonywania obliczeń na danych lokalnych. Pierwsza wypisana procedura `notify_team()` pozwala na nieblokujące poinformowanie o dotarciu do punktu synchronizacji. Nieblokująca funkcja `ready_team()` zwraca prawdę (`true`), jeśli wszystkie obrazy, których indeksy przekazuje się w parametrze, wywołały procedurę `notify_team()` przynajmniej tyle razy, co obraz wywołujący tę funkcję. Ostatnia procedura `wait_team()` jest blokująca. Po jej wywołaniu następuje zatrzymanie obliczeń na danym obrazie, aż wszystkie obrazy dotrą do punktu synchronizacji. Ponadto wszystkie opisane w tym akapicie procedury i funkcje wewnętrznie mają taki sam efekt jak uruchomienie procedury `flush_memory()`.

1.2.3 Titanium

Titanium jest dialektem języka Java, ale nie wykorzystuje w swoim modelu obliczeń Wirtualnej Maszyny Java (JVM, ang. *Java Virtual Machine*). Kod napisany w Titanium jest początkowo tłumaczony przez kompilator Titanium do języka C a później kompilowany do postaci wykonywalnej przy pomocy dowolnego kompilatora języka C. Przy generowaniu kodu programu z Titanium do języka C, kompilator generuje dodatkowe odwołania do warstwy uruchomieniowej bazującej na GASNet. Ponadto kompilator interpretujący język Titanium, stara się zoptymalizować wynikowy kod źródłowy [4, 44].

Modelem obliczeń równoległych w przypadku Titanium, podobnie jak to ma miejsce w CAF i UPC, jest model SPMD. Titanium był rozwijany w UC Berkeley aby wspomóc wykonywanie wysokowydajnych obliczeń naukowych na masywnie równoległych superkomputerach i klastrach z rozproszoną pamięcią z jednym lub więcej procesorami na węzeł. Ponadto programy napisane w Titanium mogą również być uruchamiane na systemach ze współdzieloną pamięcią.

Titanium bazuje na składni i semantyce języka Java w wersji 1.4. Dnia 30 października 2008 roku został ogłoszony koniec życia (EOL, ang. *End of Life*) platformy *Java 2 Platform, Standard Edition 1.4.2* stowarzyszonej z językiem Java 1.4, czyli de facto koniec wsparcia dla tej wersji języka. Wydaje się, że rozwój języka Titanium zakończył się w 2007 roku.

Titanium dodaje trzynaście nowych słów kluczowych do języka Java: `broadcast`, `foreach`, `immutable`, `inline`, `local`, `nonshared`, `op`, `overlap`, `partition`, `polyshared`, `sglobal`, `single`, `template`.

Dodanie słowa kluczowego `immutable` pozwoliło na stworzenie klas nierozszerzalnych i niezmiennych (ang. *immutable classes*), które dodatkowo nie rozszerzają żadnej innej klasy włączając w to klasę `Object`. Ponadto wszystkie niestatyczne pola klasy niezmiennej muszą być zadeklarowane jako stałe. Dzięki tym zastrzeżeniom, klasy niezmienne zachowują się jak standardowe typy prymitywne języka Java.

Słowo kluczowe `single` ułatwia kompilatorowi statyczną walidację kodu a także optymalizację wynikowego kodu źródłowego. Użycie tego słowa kluczowego przy zmiennej informuje kompilator, że dana zmienna we wszystkich procesach posiada tą samą wartość, a przy definicji metody informuje, że metoda jest uruchamiana przez wszystkie procesy.

W Titanium wszystkie dane są przechowywane w globalnym obszarze nazwanym *regionem*. Wszystkie obiekty i zmienne lokalne są przechowywane w regionie tworzącego je procesu. Ponadto wszystkie referencje do obiektów w Titanium domyślnie są globalne, tzn. mogą wskazywać na obiekty w każdym regionie. Użycie słowa kluczowego `local` przy deklaracji zmiennej informuje kompilator, że zmienna będzie przechowywała jedynie referencje do danych w lokalnym regionie procesu. Pozwala to na optymalizację dostępu do zmiennych w przypadku maszyn z pamięcią rozproszoną.

JVM korzysta z tzw. odśmiecacza pamięci (GC, ang. *Garbage Collector*) zamiast mechanizmu ręcznego zwalniania pamięci. Oprócz tego mechanizmu Titanium wprowadził system zarządzania pamięcią bazujący na regionach. Możliwe jest stworzenie regionu, w ramach którego tworzone są obiekty. Po zakończeniu używania zmiennych w regionie, programista ma możliwość *explicite* uruchomienia metody `Region.delete()`, która stara się odzyskać pamięć zajmowaną przez region. Nie uda się to i zostanie rzucony odpowiedni wyjątek, jeśli występuje jakakolwiek referencja zewnętrzna (spoza usuwanego regionu) do obiektu w regionie.

Wśród innych zmian języka Java, wprowadzonych przez Titanium, jest sposób tworzenia tablic. Domyślnie tablice są wielowymiarowe. W celu stworzenia tablicy podaje się ile wymiarów ona posiada, a także dziedzinę, czyli minimalne i maksymalne indeksy tablicy. Przykład tworzenia tablicy trójwymiarowej znajduje się na listingu 1.7.

```
1    Point<3> left = [0, 1, 2];
2    Point<3> right = [10, 20, 30];
3    RectDomain<3> domain = [left : right];
4    double [3d] dArray = new double[domain];
```

Listing 1.7: *Przykład tworzenia tablicy trójwymiarowej w Titanium*

Kolejnym nowym słowem kluczowym, przydatnym zwłaszcza przy tablicach, jest `foreach`. To słowo kluczowe tworzy nowy (jak na wersję języka Java 1.4) typ pętli. Pętle *foreach* w języku Java pojawiły się od wersji 1.5. Przy użyciu tego słowa kluczowego możliwe stało się iterowanie po wszystkich elementach tablicy, chociaż kolejność iteracji nie jest określona.

Bardzo ważną zmianą, w stosunku do języka Java, wprowadzaną przez Titanium jest możliwość przeciążania operatorów. Operatory przeciąża się tworząc metody w klasie podając za nazwę słowo kluczowe `op` i odpowiedni operator, np. `op==`, `op<` lub `op[]`.

W Titanium nie zabrakło, znanych z UPC czy CAF, metod do sprawdzania liczby procesów używanych w trakcie obliczeń, indeksu aktualnego procesu i metody synchronizującej. Funkcja `Ti.numProcs()` zwraca liczbę procesów, natomiast funkcja `Ti.thisProc()` zwraca indeks aktualnego procesu, który jest nieujemną liczbą całkowitą mniejszą niż wynik operacji `Ti.numProcs()`. Oczywiście każdy proces ma inny numer. Funkcja `Ti.barrier()` służy do synchronizacji procesów.

W odróżnieniu od UPC i CAF, w Titanium nie istnieje mechanizm pozwalający każdemu procesowi na odbiór danych z pojedynczego procesu. Pisanie do pamięci innego procesu następuje poprzez przypisanie wartości zmiennej i użycie słowa kluczowego `broadcast`, np.

```
newValue = broadcast calcValue from 0;
```

Możliwe jest również odczytanie danych z wszystkich procesów. W tym celu należy stworzyć tablicę, której indeksami są wszystkie indeksy procesów, i wywołać na tej tablicy metodę `exchange()`, której parametrem jest wartość wysyłana przez poszczególne procesy, np.

```
outpuArray.exchange(processValue);
```

Pełna specyfikacja języka Titanium dostępna jest w instrukcji [45].

1.2.4 X10

Język X10 [23] został stworzony przez firmę IBM w wyniku programu High Productivity Computing Systems (HPCS) finansowanego przez DARPA (*Defense Advanced Research Projects Agency*). Oprócz niego, w ramach projektu HPCS powstały języki Chapel (stworzony przez Cray Inc.) i Fortress (tworzony przez Sun Microsystems Inc., później Oracle). Wydaje się, że nazwa X10 wzięła się z celów, jakie zostały wyznaczone w projekcie IBM PERCS (*Productive Easy-to-use Reliable Computer Systems*) w ramach HPCS, w których założono, że do 2010 roku zostanie stworzony system, który będzie dziesięciokrotnie (10×) zwiększać produktywność rozwoju aplikacji równoległych [46].

Język X10 jest nowym językiem programowania służącym do tworzenia aplikacji równoległych i rozproszonych. Był tworzony z myślą o systemach *NUCC* (*Non-Uniform Cluster Computing*), czyli systemach łączących architekturę *NUMA* (*Non-Uniform Memory Access*, co oznacza niejednolity dostęp do pamięci) i systemy klastrowe. Stara się on łączyć cztery główne cele jakimi są:

bezpieczeństwo – język powinien przestrzegać i chronić przed typowymi błędami takimi jak błędy przepełnienia bufora, nieprawidłowa referencja wskaźników,

analizowalność – język powinien dać się analizować przez programy takie jak kompilatory (w szczególności w celu wykonania optymalizacji), statyczne analizatory kodu (między innymi wyszukiwanie błędów logicznych) czy narzędzia służące do refaktoryzacji (by utrzymywać odpowiednią jakość kodu źródłowego),

skalowalność – język powinien pozwalać na takie pisanie kodu źródłowego, by widać było szczególnie wrażliwe fragmenty mające ujemny wpływ na skalowalność,

elastyczność – język powinien umożliwiać wyrażanie algorytmów w sposób niezależny, dostosowujący się do architektury docelowej.

Specyfikacja języka X10 w podstawowej formie opiera się na języku Java. Jednakże znacznie modyfikuje on pewne konstrukcje służące do wprowadzania równoległości i tablic, a także usuwa wbudowane typy prymitywne. Zamiast nich wprowadza do języka *aktywności*, rozproszone wielowymiarowe tablice i klasy wartości (ang. *value classes*).

Aktywności, czyli lekkie, dynamiczne i asynchroniczne wątki, które mogą być uruchamiane zarówno lokalnie jak i na zdalnym systemie. Każda aktywność występuje w pewnym *miejscu* (ang. *place*). Odwołanie do miejsca działania aktywności jest możliwe poprzez użycie stałej `here`.

Język X10 modyfikuje składnię tworzenia tablic by mogła wyrażać rozproszone wielowymiarowe tablice. Każda tablica zawiera możliwy zbiór indeksów, dzięki czemu możliwe jest tworzenie tablic gęstych i rzadkich. Zbiory indeksów i elementy tablic mogą

być rozdystrybuowane w *miejscach* na różne sposoby. Każda aktywność ma możliwość sprawdzenia regionu, czyli indeksów tablicy, które są powiązane z danym miejscem. Gdy wymiar tablicy będzie zapisany w postaci `[.]`, oznaczać to będzie wymiar rozproszony, podobnie jak to ma miejsce w przypadku języka Co-Array Fortran.

W miejsce typów prymitywnych X10 wprowadza niemodyfikowalne *klasy wartości*, które samemu można tworzyć. Znane z języka Java typy prymitywne zostały zapisane w postaci klas wartości. Ponadto język X10 wprowadza klasę `complex` do reprezentowania liczb zespolonych. Dodatkowo klasa `java.lang.String` zmieniła semantykę z klasy referencyjnej (ang. *reference class*) na klasą wartości.

Język X10 wykorzystuje model PGAS. Aktywność ma możliwość tworzenia obiektów, które będą widoczne przez inne aktywności we wszystkich miejscach. Jednakże widoczność jest ograniczona. Odczyt i zapis do zmiennej jest możliwy jedynie przez aktywność w miejscu lokalnym. Operacje te są jednak możliwe do wykonania przez utworzenie nowej asynchronicznej aktywności w miejscu występowania zmiennej i wykorzystanie referencji występującej w zdalnym miejscu.

Kolejnym elementem wprowadzonym przez X10 jest konstrukcja związana z równoległym wykonywaniem pętli. Instrukcja `ateach` jest pętlą, która iteruje po wszystkich elementach zbioru danych i wykonuje obliczenia lokalnie, w miejscach, w których przechowywany jest jakikolwiek element tablicy. Aktualnie instrukcja `ateach` uznawana jest za przestarzałą i sugeruje się niekorzystanie z niej.

Zegary (ang. *clocks*), są mechanizmem włączonym do języka X10 w celu wykonywania operacji bariery w sposób odporny na zakleszczenia. Każda aktywność zarejestrowana może być w wielu zegarach, a zegary mogą zawierać wiele aktywności. Aktywności mogą zgłosić chęć przesunięcia wszystkich zegarów, do których należą, wyłącznie jednocześnie. Jeśli wszystkie aktywności przypisane do zegara, zgłoszą chęć przesunięcia zegara, wówczas zegar zostaje przesunięty a aktywności, które nie czekają na przesunięcie innych zegarów, mogą dalej kontynuować swoje działanie.

Język X10 wprowadza również mechanizmy operacji atomowych. Atomowe operacje bezwarunkowe oznaczone są przez słowo kluczowe `atomic`. Jest to podobna konstrukcja do znanej z języka Java konstrukcji `synchronized`, z tą różnicą, że nie podaje się obiektu, na którym następuje synchronizacja. Możliwe jest również warunkowe wykonanie operacji atomowych. Do tego celu służy konstrukcja `when (warunek)`, która wstrzymuje wykonywanie aktualnej aktywności do czasu, aż warunek zostanie spełniony.

Więcej informacji, a także specyfikację języka można uzyskać na stronie domowej projektu [23].

1.2.5 Fortress

Język Fortress [24] był językiem programowania zaprojektowanym i rozwijanym w Sun Microsystems Inc. (później Oracle Labs) jako wynik programu HPCS finansowanego przez DARPA. Nazwa Fortress, co po polsku znaczy *twierdza*, wzięła się z chęci, by był to *bezpieczny* następca języka Fortran jako języka do obliczeń HPC, choć z samym językiem Fortran język ten miał mało wspólnego [47]. Fortress miał zapewniać statyczną kontrolę systemu typizacji oraz niejawną równoległość wykorzystującą technikę podkradania zadań (ang. *work-stealing*). Ponadto język ten miał pozwalać na rozszerzanie zasobu znanych definicji za pomocą bibliotek. Dodatkowo składnia tego języka miała przypominać składnię matematyczną i wykorzystywany w publikacjach naukowych pseudokod, dzięki czemu tworzenie aplikacji miało być bardzo proste. Aktualnie jedynie podzbiór specyfikacji języka został zaimplementowany, a dalsze prace nad językiem wstrzymane z dniem 20 lipca 2012 roku [48].

1.2.6 Chapel

Język Chapel [22], podobnie jak X10 i Fortress, został stworzony jako wynik programu HPCS finansowanego przez DARPA. Nazwa Chapel wzięła się z akronimu *Cascade High Productivity Language*, gdzie słowo Cascade to nazwa projektu rozwijanego przez Cray Inc. w ramach HPCS a zarazem nazwa górskiego pasma – Gór Kaskadowych.

Język Chapel różni się znacznie od innych opisywanych tu języków programowania do obliczeń równoległych. Jest to zupełnie nowy język, który nie bazuje na żadnym innym jak to ma miejsce w przypadku UPC (rozszerza język C), Co-Array Fortran (powstał początkowo jako rozszerzenie języka Fortan), czy Titanium, X10 i Fortress (bazują one na języku Java).

Chapel został tworzony z myślą, że zadaniem użytkownika a nie kompilatora jest wprowadzanie równoległości, synchronizacji i odpowiednie umiejscawianie danych. Język ten implementuje model PGAS, czyli główną rolą kompilatora i bibliotek uruchomieniowych jest transparentne i efektywne przesyłanie danych między zadaniami.

Głównym założeniem filozofii języka Chapel jest programowanie w modelu *globalnego widoku* (ang. *global-view*). Polega on na tym, że algorytmy, czyli przepływ sterowania (ang. *control flow*) i wykorzystywane struktury danych wyraża się jako całość.

Globalny widok danych polega na tym, że programista tworzy i używa danych rozproszonych w zupełnie taki sam sposób jakby używał danych lokalnych. Globalny widok przepływu sterowania jest kolejną różnicą w stosunku do poprzednio opisywanych języków wykorzystujących model SPMD. Polega on na tym, że programy nie uruchamiają się od razu w wielu instancjach, ale tylko w jednej i w razie potrzeby wprowadza się dodatkową równoległość obliczeń za pomocą odpowiednich konstrukcji języka. Programista ma możliwość na szybkie i proste przejście do modelu SPMD, gdy na

początku obliczeń zastosuje rozwidlenie, wprowadzenie równoległości. Dodatkowo język Chapel wspiera różne style równoległości: styl równoległości zadań i styl równoległości danych. Ponadto Chapel umożliwia zagnieżdżanie równoległości. Przykład zagnieżdżonej równoległości przedstawiony jest na listingu 1.8. Przykład ten pokazuje sposób, w jaki można wykonać operację równoległą – wystarczy dane wyrażenie poprzedzić słowem kluczowym `begin`.

```
1    writeln("1: przed");
2    begin {
3        writeln("2: rownolegle");
4        begin writeln("3: rownolegle");
5    }
6    writeln("1: po");
```

Listing 1.8: *Przykład zagnieżdżonej równoległości w Chapel*

Jeśli wyrażenie poprzedzone zostanie słowem kluczowym `sync`, zadanie aktualnie wykonujące to wyrażenie zostanie zatrzymane do czasu, aż wyrażenie i wszystkie jego zadania równoległe zakończą swoje działanie.

Podobnie do `sync` działa słowo kluczowe `cobegin` z tą różnicą, że wykonuje wszystkie wyrażenia w wyrażeniu złożonym w sposób równoległy i oczekuje jedynie na ich zakończenie nie czekając na ewentualne utworzone podzadania.

Innym sposobem na uruchomienie wielu wątków jest użycie pętli `coforall`, w której każda iteracja uruchamiana jest w osobnym zadaniu. Po zakończeniu wykonywania pętli `coforall` następuje czekanie na zakończenie wszystkich uruchomionych zadań.

Język Chapel został zaprojektowany wokół filozofii wielorozdzielczej (ang. *multiresolution*), co pozwala na efektywne rozpoczęcie pisania aplikacji równoległych przez napisanie bardzo abstrakcyjnego, wysokopoziomowego kodu, a następnie na przyrostowe dodawanie niskopoziomowych detali w celu zwiększania wydajności. Przykładem zastosowania wielorozdzielczości jest pętla `forall`, która podobnie jak pętla `coforall`, iteruje po wszystkich elementach w sposób równoległy, z tą różnicą, że liczba zadań równoległych przetwarzających dane z pętli zależy od iteratora używanego przez dane i możliwości sprzętowych.

Kolejną ważną cechą języka Chapel jest możliwość decydowania o umiejscowieniu danych, o ich lokalizacji (ang. *locality*). Dzięki temu mechanizmowi obliczenia równoległe mogą odbywać się blisko danych, których one dotyczą. Do tego celu służy konstrukcja

```
on variable do action(),
```

gdzie zmienna `variable` może oznaczać normalną zmienną lub element ze specjalnej tablicy `Locales`, która zawiera informacje o zasobach systemowych, na których program jest uruchomiony, czyli informacje o lokalizacjach. Wśród tych informacji są takie dane jak liczba rdzeni procesora, ilość fizycznej pamięci, liczba uruchomionych zadań, czy

identyfikator i nazwa lokalizacji. Zmienna `numLocales` przechowuje informacje o wielkości tablicy `Locales`, czyli o liczbie lokalizacji, na których program jest uruchomiony. Zadania mogą użyć zmiennej `here`, by dowiedzieć się, na której lokalizacji są uruchomione. Ponadto wszystkie zmienne posiadają metodę `locale`, która zwraca numer lokalizacji, która zawiera daną zmienną.

Język Chapel pozwala także na pisanie aplikacji w modelu programowania obiektowego, ale nie jest to wymagane. Inną cechą tego języka jest wnioskowanie typu (ang. *type inference*), który w procesie kompilacji staje się ustalonym typem. W przykładzie na listingu 1.9 funkcja `comp` może przyjmować jako parametry wszelkie typy dające się porównać i zwraca wartości w liczbach całkowitych. Możliwe jest także *explicite* podanie typów argumentów i wartości zwracanej.

```
1    proc comp(x, y) {
2        if (x == y) then return 0;
3        else if (x < y) then return -1;
4        else return 1;
5    }
```

Listing 1.9: *Wnioskowanie typu w Chapel*

Synchronizacja w języku Chapel może być bezpośrednio powiązana z danymi. Pewne zmienne mogą być nazwane zmiennymi synchronizacyjnymi (ang. *synchronization variables*), które oprócz przechowywania danych swoich typów, mają przypisaną dodatkową wartość mówiącą, czy zmienna jest pełna czy pusta (ang. *full/empty*). Przypisywanie wartości do zmiennej czyni ją pełną a odczytanie wartości powoduje, że zmienna staje się pusta. Jeśli wątek będzie próbował odczytać pustą zmienną zostanie zatrzymany do czasu jej wypełnienia. Podobnie, jeśli wątek będzie próbował zapisać dane do pełnej zmiennej, zostanie zatrzymany. Modyfikatorem typu, by zmienna zachowywała się w ten sposób jest słowo kluczowe `sync`. Ponadto istnieją alternatywne typy zmiennych synchronizacyjnych, które blokują wątek jedynie w przypadku próby odczytania zmiennej w stanie pustym i pozwalają jedynie na pojedyncze pisanie – modyfikatorem w tym przypadku jest słowo kluczowe `single`. Istnieją również zmienne atomowe, które pozwalają na wykonywanie podstawowych operacji w sposób atomowy – tym razem modyfikuje się typ zmiennej za pomocą słowa kluczowego `atomic`. Przykłady deklaracji różnych typów zmiennych wraz z modyfikatorami są dostępne na listingu 1.10.

Inną ciekawą koncepcją wprowadzoną do języka Chapel jest mechanizm tworzenia tablic (ang. *arrays*). Tablice można tworzyć używając zakresów (ang. *ranges*) lub wykorzystując mechanizm dziedzin (ang. *domains*). Tablice mogą być wielowymiarowe. Dziedzina reprezentuje możliwy zbiór indeksów, czyli wielkość i kształt tablicy. Dwa główne typy dziedzin to dziedziny gęste prostokątne i dziedziny skojarzeniowe (lub asocjacyjne). Dziedziny gęste prostokątne, to domyślne dziedziny tworzące tablice

```
1 var a : atomic int = 100,
2     b : sync bool,
3     c : single complex,
4     s : string;
```

Listing 1.10: *Deklaracja różnych typów zmiennych wraz z modyfikatorami w Chapel*

w podobny sposób jak przy wykorzystaniu zakresów. Dziedziny skojarzeniowe zawierają indeksy określonego typu jak liczby rzeczywiste, referencje do obiektów czy łańcuchy znaków – wykorzystanie takiej dziedziny zmienia tablicę na słownik. Listing 1.11 przedstawia przykład tworzenia tablic, zakresów i dziedzin.

```
1 const dziedzina : domain(1) = {-3..3};  // dziedzina 1D
2 var tabZakres : [0..#n, 0..10] int,     // tablica 2D
3     tabDziedzina : [dziedzina] complex; // tablica 1D
```

Listing 1.11: *Tablice, zakresy i dziedziny w Chapel*

Awansowanie (ang. *promotion*) jest mechanizmem mocno związanym z tablicami. Polega on na tym, że po przekazaniu tablicy jako argumentu funkcji w miejsce gdzie spodziewana jest wartość numeryczna, następuje *implicite* wywołanie pętli `forall` na wszystkich elementach tablicy i wykonanie funkcji na tych elementach.

Język Chapel jest bardzo aktywnie rozwijany i wciąż pojawiają się nowe jego wersje. Więcej informacji na temat języka Chapel dostępne jest w pracy [37] oraz na stronie domowej projektu [22].

1.3 Język i platforma Java

Język Java został zaprojektowany w taki sposób, by wspomagać rozwój bezpiecznych i wydajnych aplikacji mających działać w heterogenicznych środowiskach rozproszonych. Ze względu na chęć zastosowania aplikacji Java w różnorodnych systemach i by zapewnić jak największą przenośność gotowych aplikacji postanowiono, by był to język interpretowany, bazujący na kodzie bajtowym (ang. *byte code*) i niezależny od architektury. Z tych cech wyłoniła się koncepcja wirtualnej maszyny Java. Ponadto, by nauka nowego języka nie była zbyt trudna, zdecydowano się, by był on prosty i podobny do aktualnie wykorzystywanych i nowoczesnych języków programowania. Stąd też kolejna cecha – język ma pozwalać na programowanie obiektowe (ang. *object-oriented programming*). Z tych powodów język Java bardzo przypomina składnię języka C++. Dodatkowo język Java miał pozwalać na pracę równoległą, więc kolejną właściwością, którą miał spełniać, była wielowątkowość [49].

1.3.1 Historia rozwoju języka i platformy Java

Pierwsza ogólnie dostępna wersja języka Java została wydana 23 stycznia 1996 roku przez Sun Microsystems i nosiła numer wersji 1.0.2 [50]. Było to wydanie zawierające pełen zbiór narzędzi potrzebny do tworzenia, rozwijania i uruchamiania aplikacji Java w środowisku Windows jak i Solaris. Od pierwszego wydania platforma Java zawierała klasę `java.lang.Thread` służącą do reprezentowania wątku obliczeń. Ponadto każdy obiekt w języku Java zawierał mechanizmy pozwalające na prostą komunikację między wątkami.

Kolejna wersja, oznaczona numerem 1.1, została wydana 19 lutego 1997 roku. Wśród niektórych nowości jakie wprowadzała ta wersja były mechanizmy serializacji obiektów i RMI (*Remote Method Invocation*) [51].

Po prawie blisko dwóch latach, bo 8 grudnia 1998 roku, wydano wersję J2SE 1.2. W ramach tej wersji dodano do platformy Java między innymi słowo kluczowe `strictfp` a także kompilator JIT (*just-in-time*). Słowo kluczowe `strictfp` służy do wskazywania klas i metod, które mają używać standardu IEEE 754 do przechowywania i obliczeń na zmiennych zmiennoprzecinkowych (ang. *floating-point*) [52, 53]. Wykorzystanie tego słowa kluczowego powoduje, że wyniki obliczeń na liczbach zmiennoprzecinkowych będą jednakowe na wszystkich platformach. Bez tego modyfikatora wyniki obliczeń na liczbach zmiennoprzecinkowych mogą zależeć od platformy i w niektórych sytuacjach generować dokładniejsze wyniki zamiast błędów przepełnienia [54].

Kolejną znaczącą wersją pod względem udogodnień wprowadzonych dla programisty chcącego wykorzystać wielowątkowość, była wersja 1.5, która została przemianowana na J2SE 5.0. Wersja ta została wydana 30 września 2004 roku. Wprowadzała ona do standardowej dystrybucji języka zestaw narzędzi w pakiecie `java.util.concurrent`. Wśród nich znalazły się między innymi semafory, bariery, rygle (ang. *locks*), zmienne atomowe i wysoko wydajne kolekcje równoległe [1, 55]. Ponadto wersja ta wprowadziła do języka między innymi pętlę *for-each*, zmienne generyczne, adnotacje i mechanizm zwany *autoboxing/unboxing* służący do automatycznej konwersji między typem prymitywnym a obiektowym opakowującym typ prymitywny. Mechanizm *autoboxing* wykorzystywany jest w przypadku konwersji typu, rzutowania na typ obiektowy. Następuje wtedy automatyczne opakowanie typu prymitywnego (np. `int`) do typu obiektowego (np. `java.lang.Integer`). Natomiast *unboxing* rozpakowuje typ obiektowy (np. `java.lang.Double`) do typu prymitywnego (np. `double`) w przypadku używania typu obiektowego do obliczeń numerycznych.

Następna wersja platformy Java SE 6, wydana 11 grudnia 2006 roku, wprowadziła szereg zmian, wśród których najważniejszymi, z punktu widzenia obliczeń równoległych, było radykalne poprawienie wydajności platformy, udoskonalenia odnoszące się do wirtualnej maszyny Java, zwłaszcza optymalizacja synchronizacji i kompilacji JIT,

a także wykorzystanie nowych algorytmów dla odśmiecacza pamięci (GC, ang. *Garbage Collector*) [56].

Java SE 6 była wersją, która na rynku najdłużej utrzymywała status najnowszej, bo aż 4,5 roku. W międzyczasie, bo 27 stycznia 2010 roku, firma Sun Microsystems została przejęta przez Oracle Corporation, przez co zmienił się właściciel języka i platformy Java [57].

Kolejną stabilną wersją języka i platformy Java była wersja Java SE 7, która została wydana 28 lipca 2011 roku. Wśród zmian związanych z równoległością wartą wspomnienia jest szkielet (ang. *framework*) *fork/join* [58]. Bazuje on na klasie `java.util.concurrent.ForkJoinPool`, w ramach której przechowywana jest pula wątków, która dzięki technice podkradania zadań (ang. *work-stealing*) w pełni wykorzystuje dostępne procesory, co pozwala efektywnie uruchamiać dużą liczbę zadań. Wykorzystując szkielet *fork/join* można efektywnie i w sposób równoległy wykonywać algorytmy bazujące na metodzie *dziel i zwyciężaj*.

Dnia 18 marca 2014 roku wydano stabilną wersji języka i platformy Java o nazwie Java SE 8. Wprowadza ona wiele zmian [59], wśród których najbardziej widoczne jest wprowadzenie:

- wyrażeń lambda,
- referencji do metod,
- interfejsów funkcyjnych,
- domyślnych metod interfejsów,
- masowych operacji na danych (znanych jako *strumienie*).

1.3.2 Programowanie równoległe i rozproszone

Wzrastająca popularność języka Java jako języka dla wykonywania symulacji, spowodowała duży popyt na rozszerzenia i biblioteki do tego języka służące do przetwarzania równoległego. Dobrym przykładem może być Parallel Java [2] czy Java Grande [3], ale one nie zostały szeroko zaadoptowane.

Innym podejściem jest mechanizm RMI (*Remote Method Invocation*) [5, 6]. Jest to mechanizm, który istnieje w języku Java od pierwszych jego wersji. Mechanizm ten pozwala na zdalne uruchamianie metod obiektów, które mogą znajdować się w innych wirtualnych maszynach, a co za tym idzie, mogą znajdować się na innych komputerach.

Podobne podejście do RMI jest reprezentowane przez ProActive [7], który dostarcza bibliotekę Javy definiującą zdalny dostęp do obiektów. Niestety, implementacja ProActive, z powodu problemów z efektywną serializacją obiektów istotnie ujemnie wpływa na wydajność, zarówno w symulacjach jak przy transferze danych.

O chęci wykorzystania języka Java do obliczeń równoległych może świadczyć między innymi to, że powstały specjalne biblioteki, które albo wiążą wywoływane metody obiektów Java z natywnymi operacjami zapisanymi w MPI za pomocą interfejsu JNI (*Java Native Interface*), albo starają się implementować interfejs MPI w języku Java (MPJ, *Message-Passing in Java*) [9, 10, 11]. Tworzone są też rozszerzenia nadbudowujące MPJ, jak na przykład rozwiązanie o nazwie *MPJ-Cache* przedstawione w [51], które dodaje wysokopoziomowe API (*Application Programming Interface*) korzystające z MPJ, czy *F-MPJ* opracowane w [60], wykorzystujące autorski mechanizm *Java Fast Sockets*.

Warto również zwrócić uwagę na istniejące próby uruchamiania aplikacji równoległych w języku Java opartych na wątkach Java (*Java Threads*) rozdzielonych wśród różnych wirtualnych maszyn Java. W tym podejściu aplikacja nie musi być modyfikowana i podział pracy jest wykonywany przez wirtualną maszynę, która przedstawia pamięć w postaci logicznie spójnej przestrzeni adresowej dla całego systemu, mimo że fizycznie pamięć jest podzielona. Wirtualne maszyny same dbają o przekazywanie danych między sobą. Dobrym przykładem implementacji rozwiązań są: dJVM [12], JESSICA2 [13] i Terracotta [14]. Takie podejście jest bardzo obiecujące, jednak wydajność jest problemem. Aktualne implementacje rozproszonych wirtualnych maszyn mogą być użyte do uruchamiania trywialnie równoległych zadań takich jak wiele instancji serwisów web (*web services*), ale mają poważne wady w efektywności, podczas wykonywania komunikacji między instancjami.

Mechanizmy równoległe w języku Java

Wykorzystanie wątków (ang. *Threads*) to podstawowy i niskopoziomowy sposób na wykorzystanie równoległości w języku Java. Tworzenie nowego wątku polega na stworzeniu nowej instancji obiektu `java.lang.Thread` i wywołaniu na tym obiekcie metody `start()`. Wykorzystując dodatkowo mechanizmy czekania (`wait()`) i notyfikacji (`notify()`, `notifyAll()`) dostępne w każdym obiekcie a także tworząc odpowiednie klasy rozszerzające `java.lang.Thread` lub wykorzystując interfejs `java.lang.Runnable` możliwe jest stworzenie złożonych aplikacji równoległych.

Jednak taki sposób programowania jest trudny i niestety bardzo podatny na proste błędy. W celu uproszczenia wykorzystania równoległości i wyeliminowania, choć po części, możliwych błędów, do języka Java wprowadzono mechanizmy i obiekty, które operują na równoległości, ale przedstawiają programiście abstrakcję wyższego poziomu. Wśród takich obiektów są między innymi klasy przedstawiające kolejki (m.in. `ConcurrentLinkedQueue` – skalowalna, wątkowo-bezpieczna (ang. *thread-safe*), nieblokująca kolejka FIFO; `ArrayBlockingQueue` – blokująca wykonanie wątku, przy próbie odczytu wartości z pustej kolejki i dodania wartości do pełnej kolejki; `PriorityBlockingQueue` – blokująca wykonywanie wątku przy próbie odczytu wartości przy pustej kolejce), klasy pozwalające na synchronizację wątków (m.in. `Semaphore`,

`CountDownLatch`, `CyclicBarrier`, `Exchanger`) czy kolekcje (m.in. `ConcurrentHashMap`, `CopyOnWriteArrayList`, `CopyOnWriteArraySet`).

Dodatkowo stworzono klasy rygli (ang. *locks*) i warunków (ang. *conditions*), które pozwalają na większą elastyczność w stosowaniu synchronizacji niż w przypadku wbudowanych mechanizmów. Możliwe jest dopisanie do pojedynczego rygla wielu warunków. Java 8 wprowadza także nowy, w pewnych warunkach bardziej wydajny, mechanizm blokady nazwany `StampedLock`, który pozwala na *optymistyczne* czytanie, czyli wykonanie odczytu wartości zmiennej bez nakładania blokady dostępu do zmiennej a następnie sprawdzenia, czy odczytana wartość zmiennej mogła być w międzyczasie zmieniona.

Dodano także klasy obsługujące bezryglowe, wątkowo-bezpieczne zmienne, zwane zmiennymi atomowymi. Rozszerzają one pojęcie zmiennej oznaczonej słowem kluczowym `volatile`, gdyż pozwalają na wykonanie atomowych, czyli niepodzielnych, operacji zmiany wartości zmiennych.

Ponadto stworzono pule tak zwanych wykonawców (ang. *executors*), które dbają o tworzenie nowych wątków i wykorzystanie stworzonych, by jak najlepiej wykorzystać dostępne zasoby procesora. Wśród pul wątków, znajduje się pula `ForkJoinPool`, która wykorzystywana jest do przetwarzania olbrzymich zbiorów danych za pomocą strumieni równoległych wprowadzonych do języka z wersją Java 8.

RMI

Osobno na uwagę zasługuje mechanizm RMI (*Remote Method Invocation*) [5].

RMI pozwala na tworzenie aplikacji rozproszonych, w których możliwe jest wywołanie metody pewnego obiektu na jednej wirtualnej maszynie Java, przez aplikację występującą na innej wirtualnej maszynie Java. Obie wirtualne maszyny mogą znajdować się na jednym komputerze lub na różnych komputerach, czyli w różnych przestrzeniach adresowych. Wywołanie takiej metody jest przezroczyste i z punktu widzenia programisty nie różni się niczym od wywołania lokalnego.

RMI wykorzystuje mechanizm serializacji, by przesyłać parametry wywołania metod i wartości zwrócone. Obiekty przesyłane są w całości, razem z ich definicjami i w przypadku potrzeby odwołania się do obiektów w różnych przestrzeniach adresowych, wykonywana jest komunikacja w celu przesłania odpowiednich danych. Dzięki temu wspierana jest pełna obiektowość danych.

Mechanizm RMI podobny jest do mechanizmu RPC (*Remote Procedure Call*), który pozwala na wywoływanie zdalnych funkcji i procedur, w sposób równie prosty, ale nie wspiera on reprezentacji obiektowej danych jak to czyni RMI.

Obiekty, których metody mogą być wywoływane z różnych wirtualnych maszyn Java, nazywają się *obiektami zdalnymi* (ang. *remote objects*). Obiekty

zdalne implementują pewien *zdalny interfejs* (ang. *remote interface*), który rozszerza interfejs `java.rmi.Remote`, a każda metoda interfejsu deklaruje zrzucanie wyjątku `java.rmi.RemoteException`. Wywołanie metod na takich obiektach nie różni się niczym od wywoływania metod lokalnie, gdyż w systemach korzystających z RMI na lokalnej wirtualnej maszynie tworzone są namiastki (ang. *stub*) implementujące dokładnie ten sam interfejs co obiekty zdalne, z tą różnicą, że metody z tego interfejsu nadzorują wywołaniem odpowiedniej metody na zdalnym obiekcie.

W aplikacji używającej RMI wykorzystane są dwa typy programów. Pierwszy nazywany jest serwerem, a drugi klientem. Serwer tworzy zdalne obiekty, przesyła referencje do nich do specjalnego rejestru RMI i czeka na wywołanie metod na tych obiektach. Dzięki rejestrowi RMI możliwe jest odwołanie się do obiektów, gdyż z niego korzysta klient chcący skorzystać ze zdalnego obiektu. Klient, po odebraniu referencji do obiektu z rejestru, może wywołać na nim odpowiednią metodę zdalną, czyli występującą w definicji zdalnego interfejsu. Wątek wywołujący taką metodę jest blokowany do czasu otrzymania odpowiedzi.

Szczególną cechą RMI jest przesyłanie definicji klasy przekazywanego obiektu, w sytuacji, gdy obiekt nie jest znany wirtualnej maszynie Java, która odbiera obiekt. Dzięki temu zachowane jest zachowanie obiektów co pozwala na dynamiczne rozszerzanie możliwości i funkcji oferowanych przez aplikację.

Schemat działania RMI można przedstawić w postaci następujących kroków:

1. serwer rejestruje *obiekt zdalny* w rejestrze RMI; obiekty zdalne rozróżniane są za pomocą nazwy,
2. klient wyszukuje za pomocą nazwy *obiekt zdalny* w rejestrze RMI i po pomyślnym znalezieniu, zwracana jest namiastka obiektu,
3. następuje wywołanie metody zdalnej przez klienta; jeśli argumenty nie są typów pierwotnych, w sytuacji, gdy obiekt nie jest zdefiniowany po stronie serwera, następuje przesłanie definicji klasy, czym zajmuje się RMI,
4. wątek klienta wstrzymuje swoje działanie czekając na zakończenie obliczeń i wynik,
5. klient odbiera wynik.

RMI jest powszechnie wykorzystywanym rozwiązaniem w aplikacjach sieciowych, jednak jego zastosowanie do obliczeń równoległych nie powidło się. Głównym powodem takiego stanu rzeczy może być to, iż tworzenie aplikacji wykorzystujących RMI jest dość skomplikowane, wymaga dodatkowych elementów takich jak rejestr RMI a przede wszystkim nie udaje się uzyskać odpowiednio wydajnej komunikacji [61]. Podstawowym przyczyną jest serializacja i deserializacja obiektów wykonywana podczas każdego wywołania zdalnej procedury. Standardowe mechanizmy zaimplementowane w języku Java są na tyle wolne, że znacząco obniżają szybkość przekazywania danych. Jednocześnie

stosowanie własnych procedur serializacji i deserializacji jest w ogólnym przypadku bardzo skomplikowane.

1.3.3 ProActive

Biblioteka ProActive Programming (dalej zwana biblioteką ProActive) jest biblioteką dla języka Java pozwalającą na wykonywanie obliczeń równoległych i rozproszonych [7]. Aktualnie rozwijana wersja biblioteki dostępna w postaci kodu źródłowego nosi numer 6.1.0 [62]. Jej poprzednia, czyli można przypuszczać, że stabilna wersja oznaczona jest numerem 6.0.1. Dzięki tej bibliotece ułatwione jest programowanie i uruchamianie aplikacji na procesorach wielordzeniowych, w sieci lokalnej, klastrach i systemach w infrastrukturze GRID. W celu usprawnienia obliczeń i uproszczenia pisania kodu źródłowego aplikacji, ProActive wprowadził podział wykorzystanych komputerów-węzłów na: węzły fizyczne i węzły wirtualne.

Na pojedynczej wirtualnej maszynie Java może znajdować się jeden lub więcej węzeł fizyczny (ang. *Node*). Uruchomienie wirtualnej maszyny Java zawierającej węzły fizyczne na zdalnej maszynie wymaga użycia odpowiedniego protokołu, np. rsh, ssh, login, GLOBUS, UNICORE. Na początku swojego działania rejestruje się on w lokalnym serwerze nazw, np. RMI lub RMI-SSH, JINI, HTTP Registry. Węzeł fizyczny tworzy i zapewnia dostęp do aktywnych obiektów.

Węzeł wirtualny (ang. *Virtual Node*) jest pojęciem abstrakcyjnym. Dzięki węzłom wirtualnym możliwe jest odseparowanie zaprojektowanej infrastruktury od faktycznej infrastruktury fizycznej. Węzły wirtualne pozwalają na elastyczne rozmieszczenie aplikacji, szybką zamianę komputerów-węzłów wykonujących obliczenia i eliminację szczegółów konfiguracji (nazwy hostów, wykorzystane protokoły, porty i interfejsy sieciowe) z kodu źródłowego.

Główne zasady działania ProActive można przedstawić w punktach:

- obliczenia są rozproszone,
- obiekty są dostępne zdalnie przez tak zwane *obiekty aktywne* (ang. *Active Object*),
- wyniki przechowywane są w specjalnych *przyszłych obiektach* (ang. *Future Object*),
- komunikacja i uruchamianie obliczeń następuje, jeśli to możliwe, poprzez asynchroniczne wywoływanie metod,
- oczekiwanie na wynik następuje dopiero wtedy, gdy jest to wymagane (mechanizm *wait-by-necessity*).

Obiekt aktywny może być stworzony z klasy prawie dowolnego typu – klasa musi być publiczna i rozszerzalna (definicja klasy nie może zawierać modyfikatora `final`)

oraz posiadać bezargumentowy konstruktor, choć inne konstruktory też mogą w niej występować. Obiekty aktywne opakowują wywołania publicznych, ale nie *finalnych* metod klasy, stąd użycie aktywnego obiektu jest przezroczyste – stworzony aktywny obiekt jest rzutowany na typ klasy, z którego powstał, a komunikacja z nim odbywa się przez odwołania do publicznych metod obiektu. Każde wywołanie tworzy komunikat zawierający parametry będące zserializowanymi obiektami Java. Wykonywanie ciała metody obiektu aktywnego nazywa się *aktywnością*.

Obiekty aktywne posiadają swój własny wątek odpowiedzialny za przetwarzanie wywołań publicznych metod w odpowiedniej kolejności. Domyślnie każde nadchodzące wywołanie metody jest obsługiwane w kolejności FIFO, czyli pierwszy otrzymany komunikat zostaje pierwszy obsłużony. Poza pewnym wyjątkiem, gdy metodę obiektu aktywnego oznacza się adnotacją, że ma być wykonywana natychmiastowo (`@ImmediateService`), kolejne wykonanie żądania następuje dopiero po zakończeniu wykonania poprzedniego. Stworzeniem wątków przetwarzających zapytania zajmuje się sam ProActive i programista nie potrzebuje ich implementować, chyba że chciałby zmienić kolejność obsługi zapytań.

Obiekty aktywne mogą być tworzone na dowolnym komputerze zaangażowanym w wykonywane obliczenie. Umiejscowienie aktywnego obiektu jest ustalane przez bibliotekę ProActive, choć istnieje możliwość, by programista sam zdecydował o jego lokalizacji.

W większości przypadków, po wywołaniu asynchronicznej metody na obiekcie aktywnym natychmiast zwracany jest *przyszły obiekt* zwany dalej obiektem *Future*. Zawiera on tymczasowy rezultat obliczeń, wypełniany gdy metoda asynchroniczna zakończy działanie. W przypadku użycia obiektu *Future* przed jego wypełnieniem, czyli przed zakończeniem działania metody asynchronicznej, następuje zatrzymanie wykonywania wątku do czasu aż rezultat będzie dostępny – jest to mechanizm zwany *wait-by-necessity*. Jeśli metoda deklaruje, że nie zwróci żadnego wyniku, czyli gdy jest typu `void`, oraz nie deklaruje zrzucenia wyjątku, to wywołanie jest asynchroniczne i nie jest zwracany obiekt *Future*. Natomiast, w przypadku, gdy zwracany typ jest typem prostym (np. `boolean`, `byte`, `int`, `double`), tablicą lub klasą z modyfikatorem `final` (np. `java.lang.String`) lub metoda deklaruje możliwość zrzucenia wyjątku, to nie jest możliwe asynchroniczne wywołanie metody.

Obiekty aktywne i obiekty Future tworzą dodatkowe komponenty pośredniczące w wysyłaniu zapytań i odbiorze odpowiedzi: namiastki (ang. *Stub*), *Proxy* i *Body*. Komponent będący namiastką jest pośrednikiem między klasą stworzoną przez programistę a komponentem *Proxy*. Komponent *Proxy* pozwala na asynchroniczne wywołania metod – wysyła zapytanie do zdalnego węzła zawierającego obiekt aktywny. Dodatkowo obiekt aktywny przechowuje kolejkę żądań w komponencie *Body* i oczywiście posiada logikę klasy, którą opakowuje.

Domyślnie w przypadku wywołań metod w obiekcie aktywnym, następuje wykonanie połączenia bezpośredniego, pomiędzy węzłem wysyłającym zapytanie a węzłem zawierającym obiekt aktywny. Jednakże ProActive pozwala także na komunikację grupową. Na grupę składa się zbiór obiektów aktywnych mogących występować na wielu węzłach biorących udział w obliczeniach. Wynikiem grupowej komunikacji jest również grupa, przy czym każda odpowiedź posiada właściwości obiektu Future. Oprócz możliwości wysyłania żądań i odbierania odpowiedzi, dostępny jest również mechanizm synchronizacji – można oczekiwać na odpowiedzi od wszystkich obiektów aktywnych, oczekiwać na N-tą odpowiedź lub na odpowiedź z zadanego obiektu aktywnego. Wszystkie obiekty aktywne w grupie muszą być tego samego typu.

Przykładowy schemat wykonania obliczenia przy pomocy ProActive przedstawia się następująco:

1. ProActive tworzy rejestr RMI,
2. ProActive rozmieszcza obiekty aktywne na węzłach,
3. w momencie wywołania metody na obiekcie aktywnym, tworzona jest odpowiednia instancja klasy zapytania przez namiastkę, która przez *Proxy* komunikuje się z odpowiednim węzłem; w tym samym czasie zwracany jest obiekt Future, a program kontynuuje swoje działanie,
4. na węźle sprawdzana jest kolejka żądań i w odpowiednim momencie żądanie jest przetwarzane,
5. po przetworzeniu żądania, wynik zwracany jest do wywołującego programu i zapisywany w zwróconym obiekcie Future.

Plik deskryptora rozmieszczenia

Węzły wirtualne definiowane są w pliku deskryptora rozmieszczenia (ang. *deployment descriptor*), opisującym rozmieszczenie, parametry i sposoby uruchamiania wirtualnych maszyn Java, a także sposoby łączenia się z istniejącymi wirtualnymi maszynami Java i ich przejmowania. Plik deskryptora rozmieszczenia, to plik w formacie XML, w którym zdefiniowane są wirtualne węzły i należące do nich wirtualne maszyny Java, zawierające węzły fizyczne. Każdy węzeł wirtualny identyfikowany jest przez unikalną nazwę.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <ProActiveDescriptor
3   xmlns="urn:proactive:deployment:3.3"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5   xsi:schemaLocation="urn:proactive:deployment:3.3
6     http://www-sop.inria.fr/oasis/ProActive/schemas/deployment/3.3/deployment.xsd">
7   <variables>
8     <descriptorVariable name="TEST_HOME" value="/home/user/benchmark"/>
9   </variables>
```

Ciąg dalszy na następnej stronie

Ciąg dalszy z poprzedniej strony

```
10    <componentDefinition><virtualNodesDefinition>
11        <virtualNode name="Workers" property="multiple"/>
12    </virtualNodesDefinition></componentDefinition>
13    <deployment>
14      <mapping><map virtualNode="Workers">
15          <jvmSet><vmName value="Jvm0"/><vmName value="Jvm1"/></jvmSet>
16      </map></mapping>
17      <jvms>
18        <jvm name="Jvm0" askedNodes="64">
19          <creation><processReference refid="localJVM"/></creation></jvm>
20        <jvm name="Jvm1" askedNodes="64">
21          <creation><processReference refid="sshProcess"/></creation></jvm>
22      </jvms>
23    </deployment>
24    <infrastructure>
25      <processes>
26        <processDefinition id="templateJVM">
27          <jvmProcess class="org.objectweb.proactive.core.process.JVMNodeProcess">
28            <javaPath>
29              <absolutePath value="/scratch/software/jdk1.8.0/bin/java"/></javaPath>
30            <classpath>
31              <absolutePath value="${TEST_HOME}/lib/'*'"/></classpath>
32            <policyFile>
33              <absolutePath value="${TEST_HOME}/proactive.java.policy"/></policyFile>
34            <jvmParameters>
35              <parameter value="-Xmx128g"/>
36              <parameter value="-Dproactive.communication.protocol=rmi"/>
37              <parameter value="-Dproactive.rmi.port=8089"/>
38              <parameter value="-Dproactive.net.nolocal=true"/>
39              <parameter value="-Dproactive.net.interface=eth0"/>
40              <parameter value="-Dproactive.useIPaddress=true"/>
41            </jvmParameters>
42          </jvmProcess>
43        </processDefinition>
44        <processDefinition id="localJVM">
45          <jvmProcess class="org.objectweb.proactive.core.process.JVMNodeProcess">
46            <extendedJvm refid="templateJVM"/>
47            <jvmParameters>
48              <parameter value="-Dproactive.hostname=wn4001"/></jvmParameters>
49          </jvmProcess>
50        </processDefinition>
51        <processDefinition id="sshProcess">
52          <sshProcess class="org.objectweb.proactive.core.process.ssh.SSHProcess"
                ↪ hostname="wn4002">
53            <processReference refid="remoteJVM"/>
54          </sshProcess>
55        </processDefinition>
56        <processDefinition id="remoteJVM">
57          <jvmProcess class="org.objectweb.proactive.core.process.JVMNodeProcess">
58            <extendedJvm refid="templateJVM"/>
59            <jvmParameters>
60              <parameter value="-Dproactive.hostname=wn4002"/></jvmParameters>
61          </jvmProcess>
62        </processDefinition>
63      </processes>
64    </infrastructure>
65  </ProActiveDescriptor>
```

Listing 1.12: *Plik deskryptora rozmieszczenia dla ProActive*

Przykładowy plik deskryptora rozmieszczenia znajduje się na listingu 1.12. Zawiera on definicję jednego wirtualnego węzła o nazwie *Workers*, w ramach którego zdefiniowane są dwie maszyny wirtualne *Jvm0* i *Jvm1*. Maszyna wirtualna *Jvm0* będzie uruchomiona lokalnie, czyli na węźle *wn4001*, a maszyna wirtualna *Jvm1* będzie uruchomiona na węźle *wn4002* przy pomocy polecenia *ssh*. Obie wirtualne maszyny mają uruchomić po 64 węzły fizyczne. Wspólne ustawienia uruchamiania wirtualnej maszyny Java znajdują się w szablonie *templateJVM*, który jest rozszerzany zarówno przez opis uruchomienia lokalnej maszyny Java (*localJVM*) jak i przez opis uruchomienia maszyny zdalnej (*remoteJVM*). ProActive wykorzystuje te ustawienia, by stworzyć komendę do uruchomienia wirtualnej maszyny Java.

Po stworzeniu pliku deskryptora możliwe staje się uruchomienie aplikacji w ProActive. Przykładowa aplikacja została pokazana na listingu 1.13. Wykonuje ona operację na grupie aktywnych obiektów. W celu ułatwienia zrozumienia kodu źródłowego, pominięto wszelkie sprawdzenia poprawności parametrów, a także zadeklarowano możliwość rzucanie wyjątku obiektu typu `Throwable` z funkcji `main` zamiast łapania i obsługiwania odpowiednich wyjątków.

Aplikacja ta na początek wczytuje konfigurację węzłów wirtualnych i fizycznych i aktywuje je wszystkie. Po aktywowaniu tworzy grupę aktywnych obiektów typu `ActiveObject` na wszystkich fizycznych węzłach jednego wirtualnego węzła, po czym wykonuje metodę `rand()`. Metoda ta wykonywana jest na wszystkich aktywnych obiektach w grupie, a zwracany wynik (`rand`) również jest grupą. Następnie wykonywane jest czekanie na zwrócenie wartości przez wszystkie aktywne obiekty, po czym następuje jest ich wypisanie. Na koniec aplikacja jest zamykana.

Programowanie w modelu SPMD w ProActive

Biblioteka ProActive daje możliwość programowania w modelu SPMD poprzez stworzenie grupy SPMD za pomocą odpowiedniej metody specjalnej klasy `org.objectweb.proactive.api.PASPMD`. Aktywne obiekty wchodzące w skład tej grupy mają możliwość poznania wielkości grupy i indywidualnego identyfikatora obiektu w grupie. Niestety komunikacja między nimi jest utrudniona, m.in. ze względu na wspomnianą wcześniej kolejność obsługi żądań.

Ponadto dostępne mechanizmy bariery działają w sposób inny niż w innych rozwiązaniach opartych o SPMD. Bariery w ProActive działają w sposób nieblokujący, czyli aktywność docierająca do bariery nie zatrzymuje się, ale kontynuuje swoje wykonanie, a każde wywołanie metod innych aktywnych obiektów oznaczone jest jako wykonane po barierze i może być przetworzone przez inny aktywny obiekt dopiero po wywołaniu odpowiedniej operacji bariery.

Istnieją trzy wersje operacji bariery:

1. *Total Barrier* – bariera wykonywana przez wszystkich członków grupy SPMD,

```
1  import java.util.Random;
2  import org.objectweb.proactive.api.*;
3  import org.objectweb.proactive.core.descriptor.data.*;
4  import org.objectweb.proactive.core.node.Node;
5  import org.objectweb.proactive.core.util.wrapper.IntWrapper;
6
7  public class PaSmallTestGroup {
8      public static class ActiveObject {
9          public static final Random r = new Random();
10         public IntWrapper rand() {
11             return new IntWrapper(r.nextInt());
12         }
13     }
14
15     public static void main(String[] args) throws Throwable {
16         String descriptor = "test.proactive.cfg.xml";
17         ProActiveDescriptor pad = PADeployment
18                 .getProactiveDescriptor(descriptor);
19         pad.activateMappings();
20
21         VirtualNode[] virtualNodes = pad.getVirtualNodes();
22         Node[] nodes = virtualNodes[0].getNodes();
23         ActiveObject group = (ActiveObject) PAGroup.newGroup(
24                 ActiveObject.class.getName(),
25                 new Object[]{}, nodes);
26
27         IntWrapper rand = group.rand();
28         PAGroup.waitAll(rand);
29         for (int i = 0; i < nodes.length; ++i) {
30             IntWrapper r = (IntWrapper) PAGroup.get(rand, i);
31             System.out.println(i + " - " + r.getIntValue());
32         }
33         pad.killall(false);
34         System.exit(0);
35     }
36 }
```

Listing 1.13: *Przykład komunikacji grupowej w ProActive*

2. *Neighbour Barrier* – bariera wykonywana przez grupę aktywnych obiektów stworzoną przez programistę,

3. *Method-Based Barrier* – bariera wstrzymująca przetwarzanie kolejnych wywołań do czasu, aż zostanie wysłane żądanie wykonania metody (lub metod) podanej w argumencie bariery.

Wady programowania z użyciem biblioteki ProActive

ProActive, będący bardzo obiecującym rozwiązaniem pozwalającym na tworzenie aplikacji równoległych i rozproszonych w języku Java, nie spełnił oczekiwań, głownie ze względów wydajnościowych. Zrównoleglenie prostych algorytmów nie stanowiło problemu, jednak dla aplikacji mających większe wymagania pamięciowe czy korzystających z obiektów o złożonej strukturze czas wykonania był znacząco dłuższy, nawet przy wykorzystaniu niewielkiej liczby wątków. Dodatkowo pojawiały się problemy wynikające z dużego zapotrzebowania na pamięć – znacznie większego niż w przypadku wersji niewykorzystującej ProActive.

Dodatkowo domyślnie model programowania równoległego w ProActive opiera się na aktywnych obiektach, w których każde żądanie przetwarzane jest w kolejności nadejścia, przez co kontaktowanie się dwóch aktywnych obiektów ze sobą, w trakcie przetwarzania innego żądania, jest utrudnione lub nawet niemożliwe.

Wykorzystanie dostępnych w ProActive mechanizmów programowania w modelu SPMD również ma swoje wady wynikające głównie z innego podejścia do wykonywania operacji bariery, a także z problemów z komunikacją między aktywnymi obiektami w grupie, związanej z kolejnością obsługi żądań.

Inną wadą jest brak odseparowania zmiennych statycznych klas między aktywnymi obiektami. Każdy aktywny obiekt znajdujący się w danej wirtualnej maszynie Java ma dostęp do statycznych pól innych obiektów, przez co może modyfikować ich zachowanie, np. przez korzystanie z generatora liczb pseudolosowych. Przy uruchomieniu tej samej aplikacji na innym zestawie węzłów takie zachowanie może generować niespodziewane rezultaty.

Znaczącym utrudnieniem w wykorzystaniu biblioteki ProActive do obliczeń na klastrach obliczeniowych jest plik deskryptora rozmieszczenia. W celu wygenerowania go na klastrze obliczeniowym wykorzystującym standardowe systemy kolejkowe, trzeba stworzyć dedykowany skrypt lub program przetwarzający listę przydzielonych węzłów do odpowiedniego formatu XML odczytywanego przez bibliotekę ProActive.

Opis wcześniejszych prób wykorzystania biblioteki ProActive do obliczeń równoległych i uzyskane rezultaty znajdują się w pracach magisterskich [63, 64, 65].

Rozdział 2

PCJ – założenia i architektura

Biblioteka PCJ (*Parallel Computations in Java*) jest to biblioteka stworzona dla języka Java, która pozwala na tworzenie aplikacji z wykorzystaniem modelu PGAS. Została ona stworzona w ramach niniejszej pracy doktorskiej od zera z wykorzystaniem najnowszej stabilnej wersji języka Java – Java SE 7. Użycie najnowszej wersji języka Java pozwala m.in. na zwiększenie wydajności komunikacji, korzystającej z sieci InfiniBand, przez wykorzystanie protokołu SDP (*Sockets Direct Protocol*), którego implementacja jest częścią platformy Java od wersji Java SE 7.

Dodatkowo w bibliotece PCJ wykorzystano klasy *New I/O* (*Java NIO*, `java.nio.*`). Dzięki zastosowaniu *NIO* komunikacja może być nieblokująca, stąd wykorzystano tylko jeden wątek wirtualnej maszyny Java do obsługi przychodzących danych. Samo odczytywanie danych odbywa się poprzez bufor, którego wielkość jest domyślnie ustalona na 256 kB. Wartość tę można zmienić stosując odpowiedni przełącznik wirtualnej maszyny Java. Tym przełącznikiem jest `-Dpcj.buffersize=X`, w którym parametr X oznacza wielkość bufora liczona w bajtach.

2.1 Założenia

W trakcie rozważań na temat rozwiązań, które mogłyby ułatwić programowanie równoległe i rozproszone w języku Java, zostały wybrane następujące założenia:

- rozwiązanie ma być zgodne z językiem Java – nie ma to być nowy język czy rozszerzenie modyfikujące składnię;
- rozwiązanie ma dać się uruchomić na jak największej liczbie systemów zawierających implementację wirtualnej maszyny Java (JVM, ang. *Java Virtual Machine*);
- w ramach rozwiązania powinno dać się uruchomić wiele instancji obliczeń działających równolegle;

- instancje mogą działać równolegle nie tylko w ramach jednej wirtualnej maszyny, ale w ramach wielu wirtualnych maszyn, które mogą znajdować się w innych fizycznych lokalizacjach;
- rozwiązanie ma bazować na paradygmacie PGAS;
- każda instancja ma swoją indywidualną pamięć lokalną, na której pracuje;
- każda instancja może korzystać z pamięci współdzielonej, do której dostęp mają wszystkie instancje obliczeń, w celu przekazywania zmiennych między instancjami.

Zgodność ze standardami

Podczas projektowania biblioteki główny nacisk położono na zgodność ze standardami języka Java. PCJ została stworzona jako samodzielna biblioteka, która nie wymaga żadnych modyfikacji języka ani nie wykorzystuje żadnych dodatkowych bibliotek. Ponadto, aby być w zgodzie ze sztandarowym hasłem języka Java – *Napisz raz, uruchamiaj wszędzie* (ang. *Write once, run anywhere*) [66] – biblioteka nie wykorzystuje dostępnych natywnych mechanizmów mogących w szczególnych przypadkach poprawiać wydajność aplikacji poprzez zastosowanie interfejsu JNI (*Java Native Interface*).

JNI pozwala na dostęp do niskopoziomowych funkcji systemu operacyjnego. Ponadto może zwiększać wydajność przez użycie zoptymalizowanego przez programistę i kompilator natywnego kodu. Jest to jednak stara hipoteza nieuwzględniająca nowoczesnych maszyn wirtualnych Java, które posiadają coraz inteligentniejsze kompilatory JIT (*just-in-time*) potrafiące automatycznie optymalizować wynikowy kod maszynowy biorąc pod uwagę m.in. statystykę najczęstszych ścieżek wykonania. Zastosowanie interfejsu JNI w PCJ wiązałoby się z koniecznością przygotowania specjalnych bibliotek dla wielu maszyn, dla których istnieje implementacja wirtualnej maszyny Java, gdyż natywny kod używany w JNI nie daje się łatwo przenosić między różnymi architekturami i systemami operacyjnymi. Ponadto dodanie dodatkowych bibliotek w zależności od systemu operacyjnego jest sprzeczne z założeniami biblioteki PCJ.

Programista wykorzystujący PCJ nie jest zmuszony do dołączania bibliotek, które nie są częścią standardowej dystrybucji Java.

Pamięć współdzielona i punkt startowy

W bibliotece PCJ każdy wątek działa samodzielnie w swojej własnej lokalnej pamięci. Domyślnie zmienne są przechowywane i dostępne lokalnie. Niektóre zmienne mogą być współdzielone z innymi wątkami. W PCJ nazywa się te zmienne *zmiennymi współdzielonymi* (ang. *shared variables*). Zmienne współdzielone przechowywane są w specjalnym *magazynie* (ang. *Storage*). Każdy wątek zawiera dokładnie jeden *magazyn*.

Ponadto istnieje klasa zwana *punktem startowym* (ang. *Start Point*). Jest to klasa, która zawiera odpowiednią metodę, od której zaczynają się obliczenia równoległe.

Instancje obliczeń – wątki PCJ

Obliczenia z wykorzystaniem biblioteki PCJ mogą być wykonywane w ramach jednej lub w ramach wielu wirtualnych maszyn Java. Pojedyncze wystąpienie JVM nazywane jest węzłem fizycznym, węzłem PCJ lub po prostu węzłem. Możliwe jest uruchomienie na jednym komputerze, na jednym węźle klastra, wielu węzłów PCJ, choć jest to zachowanie niepożądane. Zamiast tego, w ramach jednego komputera, czy jednego węzła klastra, uruchamia się jedną instancję JVM, czyli jeden węzeł PCJ. Dopiero w jego ramach uruchamia się wątki PCJ. Architektura ta podyktowana jest konstrukcją nowoczesnych systemów obliczeniowych składających się z setek węzłów obliczeniowych, które z kolei zawierają po kilka procesorów posiadających od kilku do kilkunastu rdzeni. Taka budowa hierarchii wątków wymusza wykorzystanie innych mechanizmów komunikacji wewnątrz węzła i innych przy komunikacji pomiędzy węzłami.

2.2 Architektura

Wymienione powyżej założenia budowy biblioteki mają swoje odzwierciedlenie w jej wynikowej architekturze.

2.2.1 Separacja kodu

Biblioteka PCJ zawiera wiele klas i interfejsów, ale użytkownik biblioteki powinien używać jedynie klas, interfejsów i adnotacji (ang. *annotation*) z pakietu (ang. *package*) `pl.umk.mat.pcj`. Klasy z pakietu `pl.umk.mat.pcj.internal` i jego podpakietów (ang. *subpackages*) zawierają implementacje wewnętrznych mechanizmów biblioteki i nie powinny być używane bezpośrednio.

W celu wyeksponowania tylko najważniejszych klas interfejsu programistycznego nastąpił podział wszystkich klas na wewnętrzne i ogólnodostępne. Poniżej przedstawiona jest lista klas, interfejsów i adnotacji ogólnodostępnych, z których użytkownik może korzystać:

- `pl.umk.mat.pcj.PCJ` – klasa zawierająca najważniejsze i najczęściej używane metody statyczne;
- `pl.umk.mat.pcj.StartPoint` – interfejs, który należy zaimplementować w klasie mającej być punktem startowym obliczeń;

- `pl.umk.mat.pcj.Storage` – klasa, którą należy rozszerzyć, w celu przechowywania zmiennych współdzielonych;
- `pl.umk.mat.pcj.Shared` – adnotacja informująca, że dana zmienna jest współdzielona;
- `pl.umk.mat.pcj.Group` – klasa pozwalająca na wykonywanie operacji na grupie wątków PCJ;
- `pl.umk.mat.pcj.FutureObject` – klasa ułatwiająca asynchroniczne operacje w PCJ.

2.2.2 Wielowątkowość w węźle

Pierwsze eksperymentalne wersje biblioteki PCJ działały w taki sposób, że na jednej wirtualnej maszynie Java działać mógł tylko jeden wątek PCJ. W przypadku maszyn zawierających po kilka procesorów, a co za tym idzie po kilka rdzeni, uruchamianie tylko jednej wirtualnej maszyny Java wiązało się z niewykorzystaniem dostępnych zasobów. Natomiast uruchamianie wielu instancji wirtualnej maszyny Java na jednej maszynie powodowało niepotrzebne zużywanie zasobów przez wielokrotne uruchamianie identycznych wątków nadzorujących pracę biblioteki, a także wewnętrznych wątków samej wirtualnej maszyny. Z tego powodu wprowadzono do biblioteki mechanizmy pozwalające na uruchomienie wielu wątków PCJ w ramach jednej wirtualnej maszyny.

Jedna wirtualna maszyna może zawierać wiele wątków PCJ, a każde dwa wątki PCJ są od siebie odseparowane, tak by komunikacja między nimi mogła zachodzić jedynie poprzez mechanizmy PCJ. Ma to szczególnie ważne znaczenie, gdy uruchamia się tę samą aplikację, z tą samą liczbą wątków z użyciem różnej liczby węzłów PCJ. Przykładowo, jeśli mechanizm separacji nie byłby zastosowany, to aplikacja korzystająca ze zmiennych statycznych mogłaby dawać inne wyniki w zależności od tego w ramach ilu węzłów PCJ została uruchomiona.

Biblioteka została tak stworzona, by uruchamianie aplikacji w ramach wielu wirtualnych maszyn jak i jednej wirtualnej maszyny nie wiązało się z tworzeniem specjalnych wersji kodu źródłowego do obsługi komunikacji wewnętrznej jak i zdalnej. Biblioteka ukrywa te szczegóły przed użytkownikiem i wykorzystuje mechanizm odpowiedni do określonej sytuacji.

Niestandardowe ładowarki klas

W celu separacji wątków wykorzystano niestandardowe ładowarki klas (ang. *class loader*). W języku Java dwie klasy są sobie równe jeśli mają tę samą nazwę, znajdują się w tym samym pakiecie (ang. *package*) oraz zostały załadowane przez tę samą

ładowarkę klas. Wystarczy by jeden z tych warunków nie był spełniony, by próba przypisania obiektu jednej klasy do zmiennej typu drugiej klasy zrzuciła wyjątek `java.lang.ClassCastException`, czyli błąd rzutowania klasy. Oczywiście jeśli klasa pierwsza jest nadklasą klasy drugiej z tej samej przestrzeni ładowarki klas, to wyjątek nie zostanie rzucony. Koncepcja różnych ładowarek klas wprowadza do języka Java mechanizm podobny do mechanizmu przestrzeni nazw znany między innymi z języka C++, z tą różnicą, że przestrzenie nazw z wykorzystaniem ładowarek klas tworzą się dynamicznie. Każda załadowana klasa z wykorzystaniem innej ładowarki klas posiada swoje własne zmienne statyczne.

Nie wszystkie klasy mogą zostać załadowane za pomocą niestandardowej ładowarki klas. Przykładem takich klas są klasy z pakietu, którego nazwa zaczyna się od `java.`, jak `java.lang.Math` czy `java.util.ArrayList`. Niestandardowe ładowarki klas muszą odfiltrowywać próby ładowania takich klas i delegować ich załadowanie do klasy nadrzędnej, która posiada stosowne uprawnienia. W przeciwnym wypadku zostanie wygenerowany wyjątek czasu wykonania `java.lang.SecurityException`.

W PCJ oprócz odfiltrowywania klas z pakietów z nazwą zaczynającą się od `java.` odfiltrowywane są również klasy z pakietów `javax.`, `sun.` oraz `pl.umk.mat.pcj.internal.` i ich podpakietów. Pozwala to na posiadanie przez klasy wspólnego rdzenia, który może być wykorzystywany wewnętrznie przez bibliotekę PCJ. Przykładem może być klasa `pl.umk.mat.pcj.Group`, która rozszerza klasę `pl.umk.mat.pcj.internal.InternalGroup`. Każdy wątek PCJ posiada swoją własną instancję klasy `Group`. Klasa `Group` zawiera identyfikator wątku w grupie, indywidualny dla każdego wątku. Natomiast inne dane, takie jak wielkość, identyfikator czy nazwa grupy, są wspólne dla wszystkich wątków i przechowywane są w `InternalGroup`.

Pamięć współdzielona

Zmienne w języku Java są zmiennymi, które przechowują referencje do obiektów, a nie same obiekty. Wyjątkiem od tej zasady są jedynie zmienne prymitywne, które przechowują wartości. W przypadku przypisania (np. `T b = a;`) kopiowana jest jedynie referencja, czyli dwie zmienne wskazują na dokładnie ten sam obiekt. Modyfikacja obiektu za pomocą pierwszej zmiennej (`b`) powoduje, że wprowadzone zmiany sa widoczne także w przypadku odczytania danych z obiektu z wykorzystaniem drugiej zmiennej (`a`).

Przedstawione powyżej działanie jest niepoprawne w przypadku przekazywania zmiennych między wątkami PCJ. W magazynie powinny znajdować się jedynie obiekty, które są w zgodnym stanie, to znaczy obiekty, których wszystkie pola mają stan zgodny z założeniami. Ma to na celu ochronę przed dostępem do obiektu, w którym jedne pole zostało zaktualizowane, a drugie jest w trakcie aktualizacji. Powodować by to mogło sytuację, w której wątek PCJ miałby dostęp do niepoprawnego obiektu.

Ze względu na domyślne przekazywanie obiektów jedynie przez kopiowanie referencji, każdy nielokalny zapis do magazynu i każdy nielokalny odczyt danych z magazynu za pomocą mechanizmów PCJ, tworzy kopię obiektu poprzez mechanizm serializacji i deserializacji. Serializacja to zamiana obiektu na postać strumienia bajtów, który przechowuje informacje o stanie obiektu. Deserializacja to operacja odwrotna do serializacji. W trakcie deserializacji następuje tworzenie nowego obiektu z wykorzystaniem odpowiedniej ładowarki klas.

Aktualnie lokalny zapis i odczyt kopiują jedynie referencję, czyli działają jak zwykłe przypisanie.

2.2.3 Węzły, wątki i grupy

Każdy wątek PCJ posiada swój unikatowy numer – jest to identyfikator wątku. Identyfikatorami są kolejne liczby naturalne począwszy od 0. Każdy wątek PCJ działa w ramach pewnej wirtualnej maszyny Java, czyli w ramach pewnego węzła PCJ. Podobnie jak wątek, węzeł również ma swój indywidualny numer zwany identyfikatorem fizycznym lub identyfikatorem węzła. Węzeł z numerem 0 nazywany jest *menadżerem* (ang. *Manager*) lub *node0*.

W trakcie startowania obliczeń przy korzystaniu z PCJ, każdy węzeł ma listę wszystkich węzłów biorących udział w obliczeniach. Domyślnie menadżerem jest węzeł, którego nazwa pojawia się na samym szczycie listy. Każdy węzeł przetwarza całą listę szukając wartości, które przyporządkowane są do jakiegokolwiek ich interfejsu sieciowego. Następnie, węzeł sprawdza, czy wartość zawierała numer portu sieciowego. Jeśli tak, to następuje porównanie podanego numeru portu z numerem portu z jakim uruchomiono PCJ. Gdy te wartości się zgadzają lub gdy nie podano numeru portu, zapamiętywany jest indeks przetwarzanego elementu listy. Po tej operacji każdy węzeł obliczeń zna swoją liczbę wątków i ich indeksy.

Menadżer odpowiedzialny jest za nadzorowanie działania systemu. To on odpowiada za nadawanie numerów pozostałym węzłom, wysyłanie sygnału do rozpoczęcia obliczeń, przechowywanie podstawowych informacji o utworzonych grupach i, w domyślnej konfiguracji, tylko on ma możliwość pisania na standardowe wyjście. Wśród podstawowych informacji o grupach są: nazwa, identyfikator grupy oraz identyfikator węzła odpowiedzialnego za grupę.

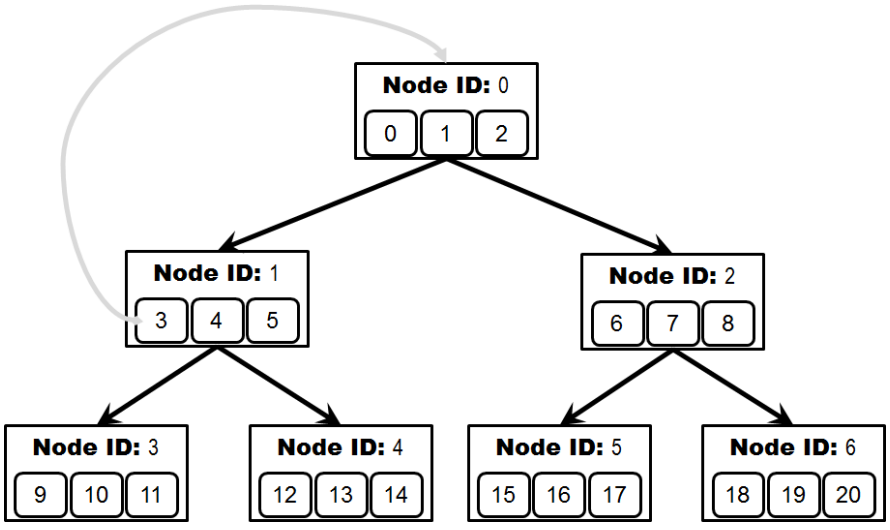
Wszystkie wątki PCJ znajdują sie w grupie globalnej. Wszystkie grupy, czyli grupa globalna i grupy, które zostały utworzone z wykorzystaniem odpowiednich mechanizmów PCJ, posiadają jeden wyróżniony węzeł zwany mistrzem grupy (ang. *Group Master*). Mistrz grupy odpowiedzialny jest za przydzielanie identyfikatorów wątkom w grupie, synchronizację wątków grupy i od niego rozpoczyna się mechanizm rozgłaszania. Rozgłaszanie wykorzystuje uporządkowanie węzłów do postaci pełnego drzewa binarnego. Mistrzem grupy zostaje węzeł, w ramach którego wątek PCJ pierwszy zgłosił się do

menadżera o dołączenie do grupy, która jeszcze nie istnieje. Menadżer wówczas tworzy grupę, nadaje jej identyfikator będący kolejną liczbą naturalną licząc od 1 i zapamiętuje, który węzeł stał się mistrzem grupy. Menadżer jest równocześnie mistrzem grupy globalnej. Aktualnie możliwe jest jedynie przyłączanie się przez wątek do grupy.

2.2.4 Rozgłaszanie po drzewie

Jak zostało to już wcześniej wspomniane, każdy wątek PCJ działa w ramach pewnej wirtualnej maszyny Java, czyli węzła PCJ. Ponadto pojedynczy węzeł PCJ może posiadać więcej niż jeden wątek PCJ. Oprócz tego, że każdy węzeł PCJ zawiera swój identyfikator, zawiera on również identyfikator swojego rodzica lub informację, że go nie ma, a także identyfikatory swoich dzieci. Identyfikatory rodzica i dzieci służą do przeprowadzania operacji rozgłaszania. Dzięki strukturze jaka tworzy się przez przesyłanie informacji o rodzicu i dzieciach do poszczególnych węzłów, operacja rozgłaszania wykonywana jest przy użyciu pełnego drzewa binarnego. Dzięki temu czas potrzebny na przesłanie komunikatu do wszystkich węzłów, zakładając, że przesłanie komunikatu od węzła A do B zajmuje jednostkę czasu, wykonywane jest w czasie $O(\log N)$, gdzie N oznacza liczbę węzłów.

Każda operacja rozgłaszania jest inicjowana przez pewien węzeł lub wątek. Komunikat przygotowany do rozgłoszenia przekazywany jest do węzła, który jest mistrzem grupy. Mistrz grupy po odczytaniu, iż komunikat należy rozgłosić, przesyła komunikat do swoich dzieci. Jego dzieci również przekazują komunikat do swoich dzieci, itd. Jeśli komunikat wymaga przekazania do wątków w ramach węzła, po rozgłoszeniu następuje liniowe przekazanie komunikatu do wątków występujących w danym węźle.



Rysunek 2.1: *Struktura pełnego drzewa binarnego i schemat rozgłaszania*

Przykład pokazany na rysunku 2.1 przedstawia strukturę pełnego drzewa binarnego. Duże prostokąty oznaczają węzły, a kwadraty wewnątrz węzłów oznaczają wątki. Numery przy napisie *Node ID* oznaczają identyfikator węzła, natomiast numery w kwadratach oznaczają numery wątków. W ramach obliczeń wykorzystanych jest 7 węzłów, w ramach których uruchomiono po 3 wątki. Wątek 3 z węzła 1 inicjuje rozgłaszanie przez przesłanie odpowiedniego komunikatu do węzła 0. Następnie węzeł 0 przesyła komunikat do swoich dzieci, czyli węzłów 1 i 2, po czym, jeśli tego wymaga komunikat, przekazuje go wątkom 0, 1 i 2. Węzeł 1 po odebraniu komunikatu przesyła go dalej do swoich dzieci, czyli do węzłów 3 i 4 a także, jeśli jest to konieczne, przekazuje go do wątków 3, 4 i 5. Operacja wykonywana jest przez wszystkie węzły, przy czym węzły na ostatnim poziomie będące liśćmi w drzewie, nie przesyłają komunikatu do kolejnych węzłów, ale jedynie przekazują komunikat do wątków uruchomionych w ramach danego węzła.

2.3 Protokół komunikacyjny w PCJ

W PCJ można rozróżnić dwa typy komunikacji. Pierwszy typ, to ten, z którego korzysta programista używający biblioteki – są to metody służące do przekazywania danych między magazynami i metody służące do operacji na grupach, czyli dołączanie do grup i synchronizacja. Drugi typ komunikacji to zakryty przed programistą, niskopoziomowy sposób komunikacji między poszczególnymi węzłami i wątkami PCJ – protokół komunikacyjny.

Komunikacja w bibliotece PCJ zaimplementowana została w oparciu o warstwę 7 modelu OSI, czyli warstwę aplikacji. Do łączenia się między węzłami wykorzystywany jest protokół TCP z warstwy transportowej oraz protokół IP z warstwy sieci.

Jeden wątek w węźle jest wątkiem przetwarzającym żądania sieciowe – nazwa tego wątku to *Selector*. *Selector* przetwarza żądania połączenia oraz odbiera i w szczególnych przypadkach wysyła dane. Sama komunikacja między węzłami następuje za pośrednictwem gniazd sieciowych. Każdy węzeł łączy się z każdym innym węzłem za pomocą nieblokującego kanału komunikacyjnego *SocketChannel*.

Wykorzystanie *Selectora* i nieblokujących kanałów komunikacyjnych pozwoliło na zastosowanie tylko jednego wątku do odbierania przychodzących danych – nie jest potrzebne tworzenie wątków dla każdego gniazda sieciowego jak to ma miejsce w przypadku komunikacji blokującej. *Selector* nasłuchuje na wszystkich kanałach i zostaje poinformowany, jeśli jakiś kanał zmienił swój stan. Gdy napłynęły dane do kanału, odczytuje je, przetwarza do postaci komunikatu i przekazuje do obsługi. Gdy z kanału ma zostać wysłana wiadomość, a kanał był pełny w momencie bezpośredniej wysyłki, to sprawdza, czy zwolniło się miejsce w kanale i jeśli tak, wysyła oczekujące dane.

Dane z *Selectora* przekazywane są do drugiego wątku o nazwie *Worker*. Worker zbiera odebrane komunikaty i wykonuje odpowiednie operacje z nimi związane.

Komunikaty są to wiadomości, które mają bezpośredni wpływ na działanie programów wykorzystujących bibliotekę. Przykładami komunikatów są: informacja o dotarciu do miejsca synchronizacji, przekazanie danych do zapisu w magazynie pewnego wątku czy prośba o odpowiedź z zawartością zmiennej z magazynu pewnego wątku. Każdy komunikat zawiera informację o typie wiadomości. Komunikaty zawierają ponadto inne dane specyficzne dla przesyłanej wiadomości.

W przypadku, gdy dane mają zostać przekazane między wątkami PCJ działającymi na jednym węźle PCJ nie jest używany *Selector*. Zamiast tego komunikat wędruje prosto do kolejki przetwarzania Worker.

Aktualnie w PCJ znajduje się 25 typów wiadomości. Są to:

- UNKNOWN,
- LOG,
- HELLO,
- HELLO_RESPONSE,
- HELLO_INFORM,
- HELLO_BONJOUR,
- HELLO_COMPLETED,
- HELLO_GO,
- FINISHED,
- FINISH_COMPLETED,
- SYNC_WAIT,
- SYNC_GO,
- NODE_SYNC,
- GROUP_JOIN_QUERY,
- GROUP_JOIN_ANSWER,
- GROUP_JOIN_REQUEST,
- GROUP_JOIN_RESPONSE,
- GROUP_JOIN_INFORM,
- GROUP_JOIN_BONJOUR,
- VALUE_ASYNC_GET_REQUEST,
- VALUE_ASYNC ↪_GET_REQUEST_INDEXES,
- VALUE_ASYNC_GET_RESPONSE,
- VALUE_PUT,
- VALUE_PUT_INDEXES,
- VALUE_BROADCAST.

Typ *UNKNOWN* używany jest w przetwarzaniu komunikatów, gdy przesłany typ jest nieznany.

W domyślnej konfiguracji zablokowane jest korzystanie ze standardowego strumienia wyjściowego i standardowego strumienia błędów ze wszystkich węzłów za wyjątkiem menadżera. Z tego powodu powstał typ *LOG*, który służy do przesyłania komunikatów, informacji i innych tekstów do wyświetlenia przez menadżera.

2.3.1 Startowanie PCJ

Komunikaty o nazwach zaczynających się od *HELLO_* wykorzystywane są na samym początku działania biblioteki. Pozwalają one na zainicjowanie wszystkich potrzebnych połączeń między węzłami, ustalenie identyfikatorów węzłów, przypisanie numerów wątków do węzłów i jednoczesne wystartowanie obliczeń na wszystkich węzłach.

Na samym początku każdy węzeł wysyła do menadżera komunikat *HELLO* z numerem portu, na którym nasłuchuje na połączenia przychodzące i listą zawierającą numery wątków do niego należących. Gdy wszystkie węzły wyślą komunikat *HELLO*, menadżer odpowiada każdemu komunikatem *HELLO_RESPONSE* zawierającym przypisany identyfikator węzła (identyfikator fizyczny) i identyfikator węzła rodzica (lub -1 w przypadku braku rodzica). Ponadto informuje wszystkie węzły o nowym węźle za pomocą rozgłoszenia komunikatu *HELLO_INFORM*. Komunikat ten zawiera takie dane jak identyfikator węzła, identyfikator fizyczny jego rodzica, listę zawierającą numery wątków przynależnych do węzła, jego nazwę hosta i port, na którym nasłuchuje. Węzły, które otrzymują *HELLO_INFORM* przesyłają je dalej do swoich dzieci, czyli do tych węzłów, których oni są rodzicami. W ten sposób wykorzystane jest ich uporządkowanie w postaci pełnego drzewa binarnego. Ponadto łączą się z nowym węzłem i wysyłają do niego komunikat *HELLO_BONJOUR* zawierający ich identyfikator fizyczny i numery ich wątków. Gdy nowy węzeł otrzyma komunikat *HELLO_BONJOUR* od wszystkich węzłów, które mają identyfikator fizyczny mniejszy od jego identyfikatora fizycznego, wysyła on komunikat *HELLO_COMPLETED* do menadżera. Gdy menadżer otrzyma komunikat *HELLO_COMPLETED* od wszystkich węzłów, inicjuje rozgłaszanie komunikatu *HELLO_GO*. Węzły po otrzymaniu tego komunikatu rozpoczynają wykonanie metody `main()` z punktu startowego.

Po skończonym każdym etapie startowania, czyli po wysłaniu *HELLO_COMPLETED* przez poszczególne węzły, wszystkie węzły z identyfikatorami nie większymi od identyfikatora węzła wysyłającego ten komunikat, są ze sobą połączone.

2.3.2 Kończenie wykonania w PCJ

Komunikaty *FINISHED* i *FINISH_COMPLETED* służą do informowania menadżera o zakończeniu obliczeń i jednoczesnego kończenia wykonywania operacji współbieżnych.

Gdy wszystkie wątki PCJ pewnego węzła zakończą swoje wykonanie, dany węzeł wysyła komunikat *FINISHED* do menadżera. Gdy wszystkie węzły poinformują o zakończeniu obliczeń, menadżer rozsyła do wszystkich, za pomocą mechanizmu rozgłaszania, komunikat *FINISH_COMPLETED* informując, że węzły mogą bezpiecznie zakończyć swoje działanie. Taka operacja jest wymagana, gdyż wątek, który jeszcze działa, może chcieć odczytać wartość zmiennej wątku, który już skończył swoje

wykonanie. Gdyby taki wątek zakończył swoje działanie, odczytanie wartości byłoby niemożliwe.

2.3.3 Dołączanie do grupy

Proces dołączania do grupy jest zbliżony do wcześniej omówionego procesu startowania PCJ. Wątek, aby dołączyć do grupy musi znać jej nazwę. Wysyła tę nazwę w komunikacie *GROUP_JOIN_QUERY* do menadżera z zapytaniem, kto jest mistrzem grupy. Menadżer sprawdza, czy dana grupa istnieje. Jeśli grupa nie istnieje, to tworzy ją nadając kolejny numer grupy i przypisuje identyfikator węzła wątku pytającego jako mistrza grupy. Następnie odpowiada wątkowi komunikatem *GROUP_JOIN_ANSWER* zawierającym identyfikator grupy i identyfikator mistrza grupy. Gdy wątek już wie, kto jest mistrzem grupy, wysyła do niego żądanie dołączenia do grupy *GROUP_JOIN_REQUEST*. Mistrz grupy nadaje nowemu wątkowi identyfikator i przesyła go z powrotem do nowego wątku w komunikacie *GROUP_JOIN_RESPONSE*. Informuje też wszystkie aktualnie występujące wątki w grupie o nowym wątku – o jego identyfikatorze globalnym i grupowym, a także o identyfikatorze węzła rodzica nowego wątku – za pomocą rozgłoszenia komunikatu *GROUP_JOIN_INFORM*. Po otrzymaniu tej wiadomości każdy wątek w grupie aktualizuje listę wątków w grupie i wysyła wiadomość *GROUP_JOIN_BONJOUR* do nowego wątku z danymi na temat siebie. Nowy wątek dodaje te informacje do swojej kopii informacji o grupie. W międzyczasie, jeśli węzeł jest rodzicem nowego węzła, następuje aktualizacja informacji o własnych dzieciach.

2.3.4 Synchronizacja wątków

Aktualnie w PCJ znajdują się dwie metody pozwalające na synchronizację wątków za pomocą mechanizmu bariery:

- jedna metoda służy do synchronizacji wszystkich wątków w grupie,
- druga metoda pozwala na synchronizację pary wątków.

Synchronizacja wszystkich wątków w grupie przebiega z wykorzystaniem komunikatów *SYNC_WAIT* i *SYNC_GO*. Komunikat *SYNC_WAIT* wysyłany jest do mistrza grupy przez węzeł, gdy wszystkie jego wątki należące do danej grupy wywołały metodę `barrier()`. Wątki po wywołaniu tej metody zatrzymują swoje działanie i czekają na sygnał do opuszczenia blokady. Gdy mistrz grupy otrzyma komunikaty *SYNC_WAIT* od wszystkich węzłów zawierających wątki w grupie, rozgłasza komunikat *SYNC_GO*, po którym wszystkie wątki są zwalniane z blokady i mogą dalej działać.

Nieco inaczej wygląda sprawa z synchronizacją tylko dwóch wątków. W tym przypadku wątek, który dochodzi do miejsca synchronizacji, zawsze wysyła do

drugiego wątku komunikat *NODE_SYNC* z parą reprezentującą identyfikatory synchronizowanych wątków. Następnie sprawdza swoją mapę zawierającą informacje o parach w synchronizacji. Jeśli w mapie znajduje się szukana para – usuwa ją z mapy. W przeciwnym wypadku oczekuje na otrzymanie komunikatu. Oczywiście mogłoby się zdarzyć, że zanim zdąży usunąć parę z mapy synchronizacji, przeciwny wątek znów będzie chciał wykonać synchronizację. By nie zgubić tej informacji, para przechowuje także liczba oczekujących synchronizacji. Stąd usuwanie pary z mapy następuje tylko wówczas, gdy liczba oczekujących synchronizacji, po odjęciu aktualnej, jest równa 0.

2.3.5 Przesyłanie danych między wątkami

Przesyłając dane-zmienne między wątkami w każdym przypadku używany jest magazyn wątku przeciwnego do wątku inicjującego połączenie, tzn. tego do którego dane są wysyłane lub od którego dane są pobierane. Istnieją trzy mechanizmy pozwalające na przekazywanie zmiennych między wątkami:

- pobieranie danych (*get*),
- wysyłanie danych (*put*),
- rozgłaszanie danych (*broadcast*).

Pobieranie danych polega na odczytaniu wartości zmiennej z magazynu przeciwnego wątku. Możliwe jest pobranie pełnych danych zmiennej lub, w przypadku tablic jedno- lub wielowymiarowych, pobranie jedynie pewnego elementu danej tablicy. Wątek chcący odebrać wartość zmiennej przesyła do odpowiedniego węzła zawierającego wątek przeciwny komunikat *VALUE_ASYNC_GET_REQUEST* lub *VALUE_ASYNC-_GET_REQUEST_INDEXES* zawierający nazwę zmiennej i identyfikator wątku żądającego danych i przeciwnego wątku; w przypadku chęci odczytania elementu tablicy wysyła również tablicę z indeksami wskazującymi na element. Węzeł, po odebraniu jednego z tych dwóch komunikatów, odczytuje dane z magazynu odpowiedniego wątku i przygotowuje komunikat *VALUE_ASYNC_GET_RESPONSE* zawierający wartość zmiennej i identyfikator wątku żądającego danych i wysyła odpowiedź do węzła zawierającego dany wątek. Identyfikator wątku żądającego danych jest wymagany, gdyż na tej podstawie wybierana jest odpowiednia ładowarka klas potrzebna do deserializacji przesłanych danych.

Innym mechanizmem jest wysyłanie danych z pamięci lokalnej do magazynu przeciwnego wątku, czyli wątku odbierającego dane. Jest to bardzo prosty mechanizm polegający na tym, że w komunikacie *VALUE_PUT* lub *VALUE_PUT_INDEXES* podaje się identyfikator wątku przeciwnego, nazwę zmiennej i jej wartość, a w przypadku wersji drugiej, komunikat zawiera także indeks tablicy, do którego daną wartość należy wstawić. Wątek odbierający taki komunikat deserializuje go za pomocą odpowiedniej ładowarki

klas, zależnej od identyfikatora wątku, i zamienia aktualną wartość w magazynie, z wartością odebraną w wiadomości. Ponadto zwiększa o jeden pewną wewnętrzną wartość przedstawiającą liczbę zmian dokonanych na zmiennej w magazynie.

Ostatnim sposobem na przesłanie danych między wątkami jest rozgłaszanie. Rozgłaszanie podobnie jak wysyłanie danych, jest inicjowane przez jeden wątek. Różnica między rozgłaszaniem a wysyłaniem danych polega na tym, że dane trafiają do wszystkich wątków w grupie, w której rozgłaszanie ma miejsce. Wątek inicjujący rozgłaszanie wysyła do mistrza grupy komunikat *VALUE_BROADCAST* zawierający identyfikator grupy, nazwę zmiennej i nową wartość zmiennej. Każdy węzeł po otrzymaniu komunikatu rozsyła go do węzłów będących jego dziećmi, po czym deserializuje przesłane dane z wykorzystaniem odpowiednich ładowarek klas i wstawia te dane do magazynów odpowiednich wątków zwiększając przy tym licznik zmian dokonanych na zmiennej w magazynie.

Rozdział 3

PCJ – z punktu widzenia programisty

W tym rozdziale przedstawione zostaną podstawy programowania z użyciem biblioteki PCJ.

3.1 Uruchamianie aplikacji

Uruchamianie aplikacji zapisanych w PCJ jest bardzo proste. Nie różni się zasadniczo niczym, w stosunku do uruchamiania zwykłych aplikacji stworzonych w języku Java. Potrzebne jest jedynie dołączenie biblioteki PCJ do ścieżki klas (ang. *class path*).

Tak jak to ma miejsce w programach napisanych w języku Java, na początku uruchamiana jest metoda `public static void main(String[] args)` klasy głównej (ang. *Main class*). Następnie następuje uruchomienie obliczeń równoległych z wykorzystaniem biblioteki PCJ. Do uruchomienia obliczeń równoległych w PCJ potrzebne jest sprecyzowanie klasy startowej, która implementuje interfejs `StartPoint`, a także klasy zwanej magazynem (ang. *Storage*), która rozszerza abstrakcyjną klasę `Storage`. Interfejs `StartPoint`, klasa `Storage` jak i inne klasy, które mogą być bezpośrednio wykorzystywane przez użytkownika znajdują się w pakiecie `pl.umk.mat.pcj`.

Interfejs `StartPoint` zawiera tylko jedną metodę: `public void main()`. Jest to metoda, która jest uruchamiana przez bibliotekę po początkowej fazie inicjalizacji, czyli po skomunikowaniu wszystkich używanych węzłów w danym obliczeniu, wystartowaniu wątków na każdym węźle, nadaniu im numerów i po początkowej synchronizacji. Metoda ta, jest początkową metodą biblioteki, podobnie jak metoda `public static void main(String[] args)` jest początkową metodą wirtualnej maszyny Java.

Klasa `Storage` służy do przechowywania deklaracji zmiennych współdzielonych. W najprostszym przypadku klasa rozszerzająca `Storage` może nie zawierać żadnych zmiennych współdzielonych.

```
1  import pl.umk.mat.pcj.PCJ;
2  import pl.umk.mat.pcj.StartPoint;
3  import pl.umk.mat.pcj.Storage;
4
5  public class MyStartPoint extends Storage
6                          implements StartPoint {
7
8      @Override
9      public void main() {
10         System.out.println("Hello!");
11     }
12
13     public static void main(String[] args) {
14         String[] nodes = new String[]{
15                 "localhost",
16                 "localhost"
17         };
18         PCJ.deploy(
19                 MyStartPoint.class, // StartPoint
20                 MyStartPoint.class, // Storage
21                 nodes
22         );
23     }
24 }
```

Listing 3.1: *Pojedyncza klasa wystarczająca do uruchomienia aplikacji równoległych w ramach biblioteki PCJ*

Aplikacja wykorzystująca bibliotekę PCJ może składać się z pojedynczej klasy, która zarówno implementuje interfejs `StartPoint` jak i rozszerza klasę `Storage`. Ta sama klasa może także być klasą główną zawierającą metodę główną języka Java: `public static void main(String[] args)`. Przykład przedstawiony na listingu 3.1 przedstawia taką klasę. Klasa ta, jak to ma miejsce w przypadku klas publicznych języka Java, powinna być zapisana w pliku o nazwie zgodnej z nazwą klasy, czyli `MyStartPoint.java`. Kompilacja i uruchomienie klasy nie różnią się od kompilacji i uruchomienia dowolnej aplikacji napisanej w języku Java:

```
javac -cp .:PCJ.jar MyStartPoint.java
java  -cp .:PCJ.jar MyStartPoint
```

Po poprawnym uruchomieniu, na ekranie powinien pojawić się wynik podobny do:

```
PCJ version 3.0.0.2 built on Sat, 15 Feb 2014 at 12:25:22 CET.
Starting MyStartPoint with 2 thread(s)...
Hello!
Hello!
```

Przykład wykorzystuje metodę `PCJ.deploy()`. Jest to metoda, która jako parametry przyjmuje zmienne określające klasę startową i magazyn, a także listę komputerów (węzłów), na których obliczenia mają być uruchomione. Lista ta zawiera informacje o wątkach PCJ. Każdy element listy zawiera nazwę komputera lub jego adres internetowy i, po dwukropku, może zawierać numer portu przypisany do danego węzła. W przypadku braku numeru portu, wybierany jest domyślny port, na którym działa biblioteka (8091). Lista jest przetwarzana w celu sprawdzenia, które jej elementy odnoszą się do:

- aktualnej wirtualnej maszyny Java – gdy nazwa komputera jest nazwą aktualnego węzła, a także numer portu jest taki sam jak domyślny numer portu,
- aktualnego węzła – gdy nazwa komputera jest nazwą aktualnego węzła, a numer portu jest różny od domyślnego,
- zdalnego węzła – gdy nazwa komputera nie jest nazwą aktualnego węzła.

Dla każdego elementu odnoszącego się do aktualnej wirtualnej maszyny Java, tworzony jest wątek PCJ. W przypadku, gdy element odnosi się do komputera, na którym uruchamiane są obliczenia, ale numer portu jest różny od domyślnego, uruchamiana jest nowa wirtualna maszyna Java, dla której podany numer portu stanie się domyślnym numerem portu. Natomiast, gdy element nie odnosi się do żadnej z wyżej wymienionych sytuacji, tworzone są połączenia SSH do zdalnych węzłów i uruchamiane są za ich pomocą nowe instancje wirtualnych maszyn Java z odpowiednim przełącznikiem ustalającym domyślny numer portu. Przełącznikiem tym jest `-Dpcj.port=`*`X`*, który przekazuje się w parametrach uruchamiania nowej wirtualnej maszyny Java.

Na listingu 3.1 przedstawiono uruchomienie aplikacji w PCJ, gdzie klasą początkową i magazynem jest klasa `MyStartPoint`, a lista komputerów składa się z dwukrotnie zapisanego słowa *localhost*, co oznacza, że lokalnie zostaną uruchomione dwa wątki.

Oprócz metody `PCJ.deploy()`, w bibliotece istnieje metoda `PCJ.start()`, która ma za zadanie również uruchomić obliczenia, ale jest przydatna zwłaszcza przy uruchamianiu obliczeń w PCJ za pomocą dostępnego na klastrze systemu planowania zadań, czyli systemu kolejkowego. W przypadku `PCJ.start()`, to system kolejkowy uruchamia wirtualne maszyny Java na różnych węzłach klastra, a biblioteka przechodzi od razu do etapu inicjalizacji.

Kolejność zapisu komputerów na liście węzłów ma znaczenie. Numer elementu na liście, zaczynając liczenie od 0, oznacza numer wątku jaki zostanie przypisany wątkowi odpowiedzialnemu za dany element. Każdy wątek ma możliwość odczytania swojego identyfikatora za pomocą polecenia:

```
int PCJ.myId()
```

Numery wątków są liczbami naturalnymi od 0 do $N - 1$, gdzie N to liczba wszystkich wątków biorących udział w obliczeniach. Wątki mogą sprawdzić tę wartość wywołując metodę:

```
int PCJ.threadCount()
```

3.2 Wyświetlanie tekstów

Domyślnie w bibliotece PCJ zablokowane jest przekazywanie tekstów do standardowych strumieni wyjścia i błędów z wątków, które nie znajdują się na węźle 0. Tylko wątki na węźle 0 mają taką możliwość. Innymi słowy gdy wątek, który nie znajduje się na węźle 0, wykona komendę:

```
1     System.out.println("Hello world!");
```

to tekst `Hello world!` zostanie porzucony. Natomiast, gdyby wątek znajdujący się na węźle 0 wykonał daną komendę, to wynik zostanie wyświetlony na ekranie lub przekazany do innego standardowego strumienia wyjścia.

W celu umożliwienia przekazywania tekstów do standardowego strumienia wyjścia, w bibliotece PCJ dodano metodę:

```
void PCJ.log(String text)
```

Korzystając z tej metody, każdy wątek ma możliwość przekazania tekstu do wyświetlenia. Każdy tekst wykorzystujący tę metodę, zostanie dodatkowo zaopatrzony identyfikatorem wątku, który dany tekst wysłał. Przykładowo, poniższy kod uruchomiony na wielu wątkach:

```
1     if (PCJ.myId() < 2) {
2         PCJ.log("Hallo world!");
3     }
```

spowoduje wypisanie (z dokładnością do kolejności linii):

```
0 > Hello world!
1 > Hello world!
```

Możliwe jest wyłączenie blokady standardowych strumieni wyjścia i błędów lub włączenie jej na węźle 0 za pomocą przełączników wirtualnej maszyny Java:

- `-Dpcj.redirect.out=`X,
- `-Dpcj.redirect.err=`X,
- `-Dpcj.redirect.node0=`X,

gdzie, gdy X wynosi 1, oznacza włączenie blokady, a każda inna wartość całkowita oznacza jej wyłączenie.

3.3 Interakcje między wątkami

Oprócz uruchamiania wątków w sposób równoległy, PCJ zawiera mechanizmy pozwalające na komunikację i interakcję między wątkami. Najważniejszymi operacjami są:

- operacja bariery,
- przekazywanie danych.

3.3.1 Bariera

Operacja bariery pozwala na synchronizację wątków. Dzięki niej programista może mieć pewność, że wszystkie wątki są na tym samym etapie obliczeń. Wątki wywołujące operację bariery zatrzymują swoje działanie do czasu, aż wszystkie wątki ją wywołają, czyli wszystkie wątki dojdą do miejsca synchronizacji. Operacja bariery nie zakańcza ani nie czeka na zakończenie operacji asynchronicznych. Operację bariery wywołuje się przez uruchomienie metody:

```
void PCJ.barrier()
```

Oprócz operacji bariery przedstawionej powyżej, która angażuje wszystkie wątki, istnieje operacja bariery pozwalająca na synchronizację między dwoma wątkami. Jej wywołanie jest analogiczne:

```
void PCJ.barrier(int id)
```

Identyfikator `id` jest identyfikatorem wątku przeciwnego. Przykładowo, aby wykonać operację bariery między wątkami o numerach 4 i 5, oba wątki muszą wywołać powyższą metodę. Wątek 4 wywołuje tę metodę z parametrem 5 (`PCJ.barrier(5)`), natomiast wątek 5 wywołuje tę metodę z parametrem 4 (`PCJ.barrier(4)`). Wątek wywołujący metodę barier zatrzymuje swoje działanie, aż przeciwny wątek wywoła analogiczną operację.

W PCJ synchronizacja wykonywana jest z wykorzystaniem struktury wątków, które uporządkowane są w postaci pełnego drzewa binarnego.

3.3.2 Zmienne współdzielone

Magazyn jest miejscem służącym do przechowywania zmiennych współdzielonych. Magazynem jest klasa rozszerzająca klasę abstrakcyjną `Storage` z pakietu `pl.umk.mat.pcj`. W ramach magazynu deklaruje się zmienne współdzielone (ang. *shared variables*).

Zmiennymi współdzielonymi mogą być jedynie niefinalne pola klasy, a więc wszystkie zmienne, które nie zawierają w swojej deklaracji modyfikatorów `static` ani `final`. Ponadto zmienne współdzielone muszą być typu, który można *serializować*, czyli muszą być albo typem prostym, albo implementować interfejs `java.io.Serializable`. W trakcie kompilacji klasy przechowującej zmienne współdzielone, kompilator, a dokładniej procesor adnotacji biblioteki PCJ, sprawdza poprawność deklaracji zmiennych współdzielonych i w przypadku niezgodności z założeniami zgłasza odpowiedni błąd. Błąd zgłaszany jest również, gdy zmienna współdzielona zadeklarowana jest w klasie nierozszerzającej klasy `Storage`.

Pole klasy oznacza się jako zmienna współdzielona przez dodanie adnotacji `@Shared`. Adnotacja może, ale nie musi, być zadeklarowana z parametrem. Parametr oznacza nazwę zmiennej współdzielonej, która będzie wykorzystywana w trakcie przekazywania danych między wątkami. W przypadku pominięcia parametru, nazwa zmiennej współdzielonej jest tożsama z nazwą pola klasy. Nazwy zmiennych współdzielonych w ramach jednego magazynu nie mogą się powtarzać. Gdyby tak się stało, kompilator wyświetli stosowny komunikat w trakcie kompilacji. Na listingu 3.2 przedstawiono sposób deklarowania dwóch zmiennych współdzielonych.

```
1    @Shared
2    int sum;
3
4    @Shared("array")
5    double[] tablica;
```

Listing 3.2: *Deklaracja zmiennych współdzielonych*

Pierwsza zmienna współdzielona zadeklarowana w liniach 1-2 ma nazwę *sum* i jest typu całkowitoliczbowego (`int`). Druga zmienna współdzielona zadeklarowana w liniach 4-5 nosi nazwę *array* i jest typu tablicowego, gdzie elementem tablicy jest typ zmiennoprzecinkowy (`double`). Należy zwrócić uwagę, że nazwa tej zmiennej jest inna w kodzie aplikacji – aby odwołać się do niej jako pola klasy należy skorzystać z nazwy *tablica*.

3.3.3 Dostęp do danych

Przekazywanie i dostęp do danych współdzielonych wiąże się z ich przechowywaniem, przetwarzaniem a także z przesyłaniem ich pomiędzy wątkami.

W PCJ dostępne są dwa sposoby transportowania danych między wątkami. Pierwszy sposób polega na odczytaniu danych z pamięci wątku, drugi sposób polega na wysłaniu danych do pamięci wątku. Ponadto istnieją dwie metody służące do odczytywania i wstawiania danych do lokalnego magazynu, czyli magazynu aktualnego wątku.

Pobieranie danych

Pobieranie danych z przeciwnego wątku działa w oparciu o nieblokującą komunikację jednostronną. Wątek chcący pobrać dane z magazynu drugiego wątku wywołuje operację:

```
FutureObject<T> PCJ.getFutureObject(int id, String name)
```

Jako parametry należy podać identyfikator przeciwnego wątku i nazwę zmiennej do odczytania. Wątek, z którego dane będą pobierane, nie musi wykonywać żadnej operacji związanej z przesłaniem danych, nie musi przerywać swoich obliczeń – przetworzeniem komunikatu zajmie się biblioteka PCJ.

Metoda `PCJ.getFutureObject()` zwraca obiekt typu `FutureObject<T>`. Jest to obiekt, który będzie przechowywał wynik operacji pobierania danych. Pozwala on wątkowi wywołującemu metodę na dalsze prowadzenie obliczeń, sprawdzenie, czy odpowiedź z danymi jest gotowa lub po prostu zatrzymać swoje działanie, aż dane zostaną odebrane. Dane są odczytywane z magazynu przeciwnego wątku i zapamiętywane nie w magazynie, ale w lokalnej pamięci.

```
1    if (PCJ.myId() == 2) {
2        FutureObject<double[]> fo;
3        fo = PCJ.getFutureObject(5, "array");
4        while (fo.isDone() == false) {
5            doSomeCalculations();
6        }
7        double[] array = fo.get();
8        doProcess(array);
9    }
```

Listing 3.3: *Pobieranie danych za pomocą metody* `PCJ.getFutureObject()`

Na listingu 3.3 przedstawiono przykładowy kod, w którym wątek 2 chce odebrać wartość zmiennej *array* z wątku o numerze 5. W tym celu wątek 2 wywołuje metodę

`PCJ.getFutureObject()` i zapisuje wynik tej operacji do zmiennej `fo`. Korzystając z tej zmiennej wątek 2 może wykonywać pewne obliczenia, dopóki dane nie zostaną odebrane. Gdy odpowiedź jest gotowa, wątek 2 odczytuje przesłaną wartość i zapisuje w zmiennej `array`. Na koniec przetwarza odczytaną wartość zmiennej.

Kolejną metodą pobierania danych z innego wątku jest metoda:

```
T PCJ.get(int id, String name)
```

Jest ona podobna do `PCJ.getFutureObject()`, z tą różnicą, że jest to metoda blokująca – po jej wywołaniu program zatrzymuje swoje działanie, aż zostanie odebrana odpowiedź z wartością zmiennej.

```
1    if (PCJ.myId() == 2) {
2        double[] array = PCJ.get(5, "array");
3        doProcess(array);
4    }
```

Listing 3.4: *Pobieranie danych za pomocą metody `PCJ.get()`*

Listing 3.4 przedstawia implementację pobierania wartości zmiennej *array* z wątku o numerze 5 przez wątek 2. Kod jest analogiczny do tego przedstawionego na listingu 3.3, z tą różnicą, że aktualnie nie ma możliwości wykonania innych obliczeń w trakcie czekania na zakończenie pobierania danych.

Metoda `PCJ.get()` przedstawiona na listingu 3.4 oznacza dokładnie to samo, co następujące wywołanie:

```
double[] array = PCJ.<double[]>getFutureObject(5, "array").get();
```

W celu odczytywania wartości zapisanej w lokalnym magazynie należy użyć następującej metody:

```
T PCJ.getLocal(String name)
```

Metoda ta zwraca referencję na obiekt przechowywany w magazynie lub wartość prymitywną, jeśli typem odczytywanej zmiennej współdzielonej jest typ prymitywny. Można wykorzystać metodę `PCJ.getFutureObject()` lub `PCJ.get()` do odczytania wartości z lokalnego magazynu, ale w takim wypadku obiekt przechowywany w magazynie będzie sklonowany i zostanie zwrócona jego kopia.

Wysyłanie danych

Wysyłanie danych do pamięci wątku wiąże się bezpośrednio z modyfikacją danych w magazynie wątku docelowego. Wysyłanie danych jest wykonywane w sposób nieblokujący za pomocą komunikacji jednostronnej – węzeł odbierający dane nie musi przerywać obliczeń, by dane uzyskać. W celu wysłania danych wykorzystuje się metodę:

```
void PCJ.put(int id, String name, Object value)
```

Parametrami metody są: identyfikator wątku, nazwa zmiennej współdzielonej oraz nowa wartość zmiennej.

Dzięki temu, że zmienne współdzielone zdefiniowane w magazynie są takie same u wszystkich wątków, wątek wysyłający dane ma możliwość sprawdzenia, czy przesyłana wartość ma typ zgodny ze zmienną współdzieloną, której wartość będzie zmieniana. W przypadku, gdyby tak nie było, wątek wysyłający zostanie o tym natychmiast poinformowany.

Natomiast, aby wstawić nową wartość do lokalnego magazynu należy skorzystać z metody:

```
void PCJ.putLocal(String name, Object value)
```

Ta metoda nie klonuje obiektu, ale ustawia referencję na wstawiany obiekt w zmiennej współdzielonej. Podobnie jak miało to miejsce w przypadku pobierania danych, możliwe jest skorzystanie z ogólnej metody do wysyłania danych `PCJ.put()`, ale wówczas obiekt zostanie sklonowany i jego kopia będzie przechowywana w magazynie.

```
1     if (PCJ.myId() == 0) {
2         PCJ.put(1, "array", new double[2]);
3         PCJ.barrier(1);
4     } else if (PCJ.myId() == 1) {
5         PCJ.barrier(0);
6         doSomething(PCJ.getLocal("array"));
7     }
```

Listing 3.5: *Wysłanie danych i bariera*

Ze względu na to, że metoda `PCJ.put()` jest całkowicie asynchroniczna i nieblokująca może dojść do sytuacji, w której jeden wątek zlecił wysłanie danych, a drugi wątek jeszcze nie zdążył ich odebrać lub przetworzyć, pomimo że, oba wątki w międzyczasie wykonały operację bariery. Przykładowy kod ilustrujący problem przedstawiony jest na listingu 3.5. Dzieje się tak, gdyż operacja bariery nie sprawdza stanu aktualnych połączeń asynchronicznych, a metoda `PCJ.put()` pozwala na kontynuację obliczeń bezpośrednio po przygotowaniu danych do wysyłki, bez czekania na odebranie danych ani na ich przetworzenie i włożenie do magazynu przez wątek odbierający dane.

Może się zdarzyć, że odczytanie wartości zmiennej współdzielonej *array* zwróci jej poprzednią wartość. Z tego względu wprowadzono do PCJ metody pozwalające na monitorowanie stanu zmiennej współdzielonej:

- `void PCJ.monitor(String name)`,
- `void PCJ.waitFor(String name, int count)`.

Pierwsza metoda (`PCJ.monitor()`) informuje bibliotekę, że programista jest zainteresowany zebraniem informacji o liczbie zmian w zmiennej współdzielonej, resetuje

licznik zmian. Wywołanie drugiej metody (`PCJ.waitFor()`) powoduje oczekiwanie na zmianę wartości w zmiennej współdzielonej odpowiednią liczbę razy (`count`). W przypadku, gdy zmienna została zmodyfikowana odpowiednią liczbę razy przed wywołaniem tej metody, nie następuje oczekiwanie, a jedynie zmniejszany jest licznik o odpowiednią wartość. Ponadto istnieje jeszcze jedna metoda, z krótszą liczbą parametrów, mająca na celu oczekiwanie na tylko jedną zmianę wartości zmiennej współdzielonej:

```
void PCJ.waitFor(String name)
```

Na początku działania aplikacji, wszystkie zmienne współdzielone mają wyzerowany licznik zmian. Każde wywołanie metody `PCJ.put()` lub `PCJ.putLocal()` zwiększa wartość licznika zmian odpowiedniej zmiennej współdzielonej. Każde wyjście z oczekiwania na zmianę wartości zmiennej współdzielonej, czyli z jednej z metod `PCJ.waitFor()`, powoduje zmniejszenie wartości licznika.

```
1     if (PCJ.myId() == 0) {
2         PCJ.put(1, "array", new double[2]);
3     } else if (PCJ.myId() == 1) {
4         PCJ.waitFor("array");
5         doSomething(PCJ.getLocal("array"));
6     }
```

Listing 3.6: *Wysłanie danych i monitorowanie zmiennych*

Listing 3.6 przedstawia poprawną implementację oczekiwania na dane wysłane za pomocą metody `PCJ.put()`. Wątek 1 nie wykona dalszych kroków, póki nie dostanie informacji, że zmienna *array* została zmieniona. Dopiero po odebraniu tej wiadomości zostanie odczytana wartość zmiennej współdzielonej *array*.

Rozgłoszenie danych

Rozgłoszenie danych jest specjalnym typem operacji wysyłania danych do wątków. Operację tą uruchamia się przez wywołanie metody:

```
void PCJ.broadcast(String name, Object value)
```

Operacja rozgłaszania działa w podobny sposób do wysyłania danych do pojedynczego wątku. Podobnie jak operacja `PCJ.put()` wykorzystuje ona nieblokującą komunikację jednostronną, z tą różnicą, że wykorzystywany jest mechanizm rozgłaszania. Wątek wysyłający dane, wysyła je do węzła, który jest mistrzem grupy, który z kolei rozsyła je do swoich dzieci, a te do swoich dzieci, itd. Dodatkowo, węzeł po rozgłoszeniu danych, wstawia do magazynów, czyli pamięci wątków, nowe dane oraz zwiększa licznik modyfikacji odpowiedniej zmiennej współdzielonej.

Kod źródłowy przedstawiony na listingu 3.7 przedstawia poprawny sposób wykonania operacji rozgłoszenia danych w PCJ. Na początek wszystkie wątki zaznaczają,

```
1     PCJ.monitor("array");
2     PCJ.barrier();
3
4     if (PCJ.myId() == 3) {
5         PCJ.broadcast("array", new double[1024]);
6     }
7
8     PCJ.waitFor("array");
```

Listing 3.7: *Rozgłaszanie danych z monitorowaniem zmian zmiennej współdzielonej*

że chcą monitorować zmienną współdzieloną *array*. Następnie następuje operacja bariery, by mieć pewność, że wszystkie wątki zdążyły wyzerować licznik zmian zmiennej *array* przed rozpoczęciem rozgłaszania. Rozgłoszenie jest zainicjowane przez wątek o numerze 3 i tylko on wywołuje metodę `PCJ.broadcast()`. Następnie wszystkie wątki wywołują metodę `PCJ.waitFor()` i oczekują na otrzymanie nowych danych.

Więcej na temat rozgłaszania można przeczytać w podrozdziale 2.2.4 na stronie 51.

Przekazywanie elementów tablic

Do tej pory przedstawione było przekazywanie całych zmiennych, tzn. zastępowanie starych wartości nowymi. W przypadku typu tablicowego, możliwe jest również przekazywanie poszczególnych elementów tablicy. Takie przekazywanie dostępne jest za pomocą metod:

- `FutureObject<T> PCJ.getFutureObject(int id, String name, int... indexes)`,
- `T PCJ.get(int id, String name, int... indexes)`,
- `void PCJ.put(int id, String name, Object value, int... indexes)`,
- `T PCJ.getLocal(String name, int... indexes)`,
- `void PCJ.putLocal(String name, Object value, int... indexes)`.

Są to te same metody co przedstawione wcześniej, ale mają dodatkowy element – tablicę zawierającą odwołanie do indeksu tablicy będącej zmienną współdzieloną. W przypadku, gdy typ tablicowy nie posiada wystarczającej liczby wymiarów (np. gdy jest jednowymiarowy, a tablica indeksów posiada więcej niż 1 indeks), zrzucany jest wyjątek `java.lang.ArrayIndexOutOfBoundsException`. Dodatkowo, każde wywołanie `PCJ.put()`, czy to z odniesieniem do indeksu tablicy, czy bez niego, traktowane jest jako zmiana i zwiększa licznik modyfikacji zmiennej współdzielonej.

Znając przedstawione w tym miejscu wersje metod, można bardzo prosto stworzyć kod, który będzie zbierać wyniki od wszystkich wątków do jednego, wybranego wątku (listing 3.8) a także będzie rozdzielał dane między różne wątki (listing 3.9).

```
1     if (PCJ.myId() == 0) {
2         PCJ.putLocal("array", new double[PCJ.threadCount()]);
3         PCJ.monitor("array");
4     }
5     PCJ.barrier();
6
7     PCJ.put(0, "array", Math.sin((double) PCJ.myId()),
          ↪ PCJ.myId());
8
9     if (PCJ.myId() == 0) {
10        PCJ.waitFor("array", PCJ.threadCount());
11        int i = 0;
12        for (double d : (double[]) PCJ.getLocal("array")) {
13            System.out.println(i++ + "> " + d);
14        }
15    }
```

Listing 3.8: *Zbieranie danych ze wszystkich wątków do tablicy wątku 0*

Sposób na zebranie danych ze wszystkich wątków przedstawiony jest na listingu 3.8. Wątek 0 tworzy nową tablicę długości takiej, jaka jest liczba wątków w trakcie obliczeń i ustawia jej monitorowanie. Następnie wszystkie wątki wykonują barierę, tak by żaden nie próbował pisać do tablicy, zanim wątek 0 ją zainicjalizuje. Po tej operacji, wszystkie wątki wkładają do tablicy wątku 0, w miejscu skojarzonym z ich identyfikatorem, wartość przez siebie obliczoną. Dalej, wątek 0 czeka na dane ze wszystkich wątków, po czym wypisuje uzyskane przez wątki wyniki.

Listing 3.9 przedstawia sposób, w jaki można wykonać operację rozdzielenia danych pomiędzy wszystkie wątki biorące udział w obliczeniach. W nim wątek 0 tworzy tablicę długości takiej, jaka jest całkowita liczba wątków w obliczeniach i ją wypełnia. Warto zwrócić uwagę na wykorzystanie metody `PCJ.putLocal()`, dzięki której, wątek 0 ma w dalszej części kodu możliwość modyfikowania zmiennej lokalnej i zmiany te widoczne są poprzez zmienną współdzieloną zapisaną w magazynie. Po wypełnieniu tablicy przez wątek 0, wszystkie wątki wykonują barierę i każdy wątek pobiera odpowiedni fragment tablicy do zmiennej lokalnej. Następnie wszystkie wątki przekazują do wypisania odczytaną przez siebie wartość.

```
1     if (PCJ.myId() == 0) {
2         double[] array = new double[PCJ.threadCount()];
3         PCJ.putLocal("array", array);
4         for (int i = 0; i < array.length; ++i) {
5             array[i] = Math.sin((double) i);
6         }
7     }
8     PCJ.barrier();
9
10    double d = PCJ.get(0, "array", PCJ.myId());
11
12    PCJ.log("" + d);
```

Listing 3.9: *Rozdzielenie danych przez wątek 0 do wszystkich wątków*

3.3.4 Grupy

Domyślnie w PCJ, wszystkie operacje wykonywane są na grupie wszystkich wątków, jakie biorą udział w obliczeniach. Ta grupa nazywana jest *grupą globalną*. Istnieje możliwość wykonywania operacji na podgrupach wątków. Grupy rozróżniane są nazwami. Nie ma ograniczenia na liczbę grup ani na liczbę wątków w grupie. Każdy wątek może stworzyć lub dołączyć do każdej grupy – do tego celu służy metoda:

```
Group PCJ.join(String name)
```

Jeśli wątek wywołujący metodę nie znajduje się w grupie, to dołączy do niej, dostanie swój identyfikator grupowy, a wartość zwrócona przez metodę to zmienna przedstawiająca grupę. Korzystając z tej zmiennej wątek może wykonywać operacje na danej grupie. Jeśli wątek już znajdował się w grupie, zwrócona zostanie od razu zmienna przedstawiająca grupę. Należy pamiętać, że operacja dołączania do grupy jest blokująca, ale asynchroniczna. To, że operacja jest blokująca oznacza, że wątek dołączający się do grupy oczekuje na zakończenie tej operacji. Asynchroniczność w tym przypadku oznacza, że dołączanie do grupy nie wymaga bezpośredniego udziału innych wątków grupy.

By mieć pewność, że inne wątki grupy widzą nowego jej członka, należy po etapie dołączania do grupy wywołać barierę na grupie zawierającej aktualne wątki grupy i nowy wątek, co w wersji bezpiecznej oznacza synchronizację grupy globalnej, a w wersji minimalistycznej, barierę pomiędzy dołączającym się wątkiem, a wątkiem, który znajduje się aktualnie w grupie. Wersja minimalistyczna może być szczególnie przydatna, gdy w danym etapie obliczeń do grupy dołącza tylko 1 wątek. Wersja bezpieczna może być bezpiecznie użyta, gdy w danym etapie do grupy dołącza więcej niż jeden wątek lub gdy nie mamy pewności, który wątek już znajduje się w grupie.

Listing 3.10 przedstawia sposób dołączania do grupy. W jego ramach wykorzystana jest metoda `PCJ.getPhysicalNodeId()`. Jest to metoda, która zwraca identyfikator

```
1     Group group = PCJ.join("group:" +
          ↪ PCJ.getPhysicalNodeId());
2     PCJ.barrier();
3
4     PCJ.log(group.myId() + "/" + group.threadCount() + " " +
          ↪ group.getGroupName());
```

Listing 3.10: *Dołączanie do grupy*

nadany przez bibliotekę PCJ wirtualnej maszynie Java, na której działają wątki PCJ. Identyfikatory te, to kolejne liczby naturalne licząc od 0. Po wykonaniu operacji dołączania do grupy następuje operacja bariery. Po niej jest pewne, że wszystkie wątki zdążyły zakończyć procedurę dołączania do grup. Na koniec przesyłany jest tekst do wyświetlenia zawierający identyfikator wątku w grupie, liczbę wątków w grupie i nazwę grupy.

Gdyby uruchomić kod 3.10 na 6 wątkach, gdzie 4 znajdują się na pierwszym, a 2 na drugim węźle, wynik wyglądałby podobnie do (bez uwzględniania kolejności linii):

```
0 > 0/4 group:0
4 > 1/4 group:0
2 > 2/4 group:0
5 > 3/4 group:0
1 > 0/2 group:1
3 > 1/2 group:1
```

Oprócz metod `group.myId()`, `group.threadCount()` i `group.getGroupName()` przedstawionych na listingu 3.10, w grupie dostępne są następujące metody:

- `void Group.barrier()`,
- `void Group.barrier(int id)`,
- `FutureObject<T> Group.getFutureObject(int id, String name)`,
- `FutureObject<T> Group.getFutureObject(int id, String name, int...`
  `↪ indexes)`,
- `T Group.get(int id, String name)`,
- `T Group.get(int id, String name, int... indexes)`,
- `Group.put(int id, String name, Object value)`,
- `Group.put(int id, String name, Object value, int... indexes)`,
- `Group.broadcast(String name, Object value)`.

Sposób działania tych metod został już wcześniej przedstawiony, z tą różnicą, że wcześniejsze operacje działały na grupie globalnej, zawierającej wszystkie wątki, a przy korzystaniu z metod grupowych, zaangażowane są tylko wątki w grupie i są używane grupowe a nie globalne identyfikatory wątków.

Rozdział 4

Metody oceny rozwiązań równoległych

W celu oceny rozwiązań równoległych należy wziąć pod uwagę różne czynniki. Weryfikacja rozwiązania powinna zawierać następujące elementy:

- weryfikacja teoretyczna,
- weryfikacja pod kątem wydajności obliczeniowej,
- weryfikacja uwzględniająca skalowalność,
- weryfikacja łatwości programowania.

4.1 Weryfikacja teoretyczna

Biblioteka PCJ została stworzona w oparciu o paradygmat PGAS. Jego uzasadnienie teoretyczne można znaleźć w pracy [67], w związku z tym, w niniejszej pracy nie ma potrzeby osobnego przeprowadzania weryfikacji użytego modelu. Dodatkowo model PGAS, jak i sama biblioteka PCJ jest bardzo skomplikowana i przeprowadzenie pełnej teoretycznej weryfikacji wydaje się być zadaniem bardzo pracochłonnym, które stanowi osobne zagadnienie. Warto też wspomnieć, iż inne rozwiązania, takie jak CAF, UPC czy Titanium, zostały opracowane i były wykorzystywane zanim opublikowane zostało formalne uzasadnieniu modelu PGAS wykorzystywanego przez te rozwiązania [68], co jest praktyką stosowaną w rozwiązaniach HPC.

Nie ma w zasadzie również możliwości formalnego porównania biblioteki PCJ z innymi ugruntowanymi rozwiązaniami jak biblioteki MPI i OpenMP. Biblioteka PCJ i biblioteka MPI a także OpenMP są rozwiązaniami budowanymi w oparciu o bardzo odmienne paradygmaty [40]. Uzasadnione za to jest porównanie wydajności poszczególnych rozwiązań, co zresztą jest dokonywane powszechnie w literaturze naukowej. Podejście takie przyjęto również w rozprawie.

4.2 Weryfikacja wydajności obliczeniowej

Porównywanie wydajności obliczeniowej z innymi rozwiązaniami jest powszechnie stosowane w literaturze naukowej. W szczególności porównywanie następuje z rozwiązaniami zaimplementowanymi w języku C z wykorzystaniem biblioteki MPI. Stąd w rozprawie następuje porównanie biblioteki PCJ z rozwiązaniami opartymi o MPI. Należy jednak pamiętać, że MPI jest biblioteką rozwijaną od ponad 20 lat z udziałem wielu grup badawczych oraz firm komercyjnych przez co jest bardzo dobrze zoptymalizowana i uważana za *de facto* wzorzec w zakresie efektywności rozwiązań. W przypadku podstawowych testów zostało wykonane również porównanie z biblioteką ProActive.

Zrezygnowano z porównywania PCJ z innymi rozwiązaniami, w szczególności opartymi o język Java (w szczególności opisanymi w pracy). Większość tych rozwiązań ma status oprogramowania dostarczanego w formie *as is* z wąskim zakresem stosowalności, a ich wykorzystanie wiązało się z dużym nakładem pracy zarówno ze strony autora jak też administratorów systemów HPC, z których korzystano. W pracy skupiono się na oprogramowaniu, ugruntowanym w HPC, które jest dostępne na szeroką gamę systemów i jest wykorzystywane praktycznie np. do prognozowania pogody.

Podczas tworzenia biblioteki PCJ nie przeprowadzono zaawansowanej optymalizacji obliczeniowej. Poszczególne implementacje MPI były przez wiele lat optymalizowane pod katem różnych zastosowań. W efekcie w zależności od wielkości danych MPI stosuje kilka różnych algorytmów kompresji i przesyłania danych. Wykorzystywane algorytmy i zakres ich stosowania zostały ustalone praktycznie na bazie doświadczeń w wykorzystaniu MPI. Przy projektowaniu biblioteki PCJ priorytetem była wydajność dla dużych danych i skalowalność rozwiązań. W związku z tym proces optymalizacji dla różnych rozmiarów danych nie został jeszcze przeprowadzony. Problem ten jest generalnie szerszy i związany z wydajnością natywnych rozwiązań *Java Concurrency* wykorzystywanych przez bibliotekę PCJ. W chwili obecnej prowadzone są prace nad podniesieniem wydajności wirtualnej maszyny Java w tym zakresie (wprowadzenie usprawnień planowane jest w wirtualnej maszynie Java wersji 9 dostarczanej przez Oracle jak również OpenJDK) [69].

W celu wyeliminowania testowania przebiegów aplikacji przed wykonaniem podstawowych mechanizmów kompilacji JIT (*just-in-time*) wykorzystano metodę rozgrzewania (ang. *warm-up*) wirtualnej maszyny Java przez wielokrotne uruchomienie powtórzeń głównej pętli. Jako wynik testu zawsze przyjmowano najbardziej optymistyczny wariant uruchomienia, mianowicie minimalny czas wykonania jednego przebiegu pętli głównej lub maksymalną otrzymywaną przepustowość. W ramach głównej pętli wykonywano wiele powtórzeń danego testu, stąd po wybraniu minimalnego czasu wykonania głównej pętli, następowało uśrednianie czasu wykonania pojedynczego testu. W ten sposób starano się również uwzględnić możliwe wstrzymania działania aplikacji

związane z odśmiecaniem pamięci (ang. *garbage collection*, GC). Zastosowano taką metodę ze względu na to, iż obliczenia wykonywane są na wielu wirtualnych maszynach Java, w ramach których procesy JIT oraz GC wykonują się niezależnie na różnych instancjach maszyny wirtualnej. Dodatkowo w nietestowym uruchomieniu aplikacji, procesy związane z JIT i GC mogą również mieć kluczowy wpływ na wydajność aplikacji, stąd całkowite wyeliminowanie tych mechanizmów mogłoby powodować uzyskanie zawyżonych wyników w stosunku do wydajności możliwej do osiągnięcia w rzeczywistości.

4.2.1 Testy wydajnościowe

Tradycyjnie w przypadku rozwiązań HPC stosuje się powszechnie weryfikację rozwiązań pod kątem czasu wykonywania. W przypadku rozwiązań programistycznych takich jak PCJ wykonuje się proste testy podstawowych operacji jak komunikacja, rozgłaszanie, czy bariera.

Testy te mają różną charakterystykę. Testy komunikacyjne zwane *ping-pongiem* wymagają użycia dwóch wątków. Natomiast większość innych wykonywana jest dla różnej liczby wątków. W pracy przyjęto, by osobne jednostki obliczeń nazywać *wątkami*, ponieważ w modelu PGAS mamy raczej do czynienia z wątkami niż z procesami, gdyż każdy wątek obliczeń ma swoją lokalną pamięć, którą mają i wątki i procesy, ale również pamięć współdzieloną, którą posiadają tylko wątki wchodzące w skład procesu.

Dla języka Java zbiór testów został opracowany przez *Java Grande Forum* i jest wzorowany na testach dostępnych dla innych rozwiązań w zakresie programowania równoległego. Przykładowe testy wchodzą w skład *Java Grande Forum Benchmark Suite* [3].

4.2.2 Standardowe *kernele* obliczeniowe

Najlepszym przykładem standardowych *kerneli* obliczeniowych dla systemów równoległych jest *HPC Challenge Benchmark Suite*. Składa się on z siedmiu testów, z których do najważniejszych, powszechnie stosowanych należą cztery aplikacje:

- EP STREAM (Triad),
- Global RandomAccess,
- Global HPL,
- Global FFT.

Ich opis dostępny jest w pracy [70, 71]. Implementacja tych testów jest rozbudowanym zagadnieniem. Była ona przeprowadzana równolegle do niniejszej pracy. Implementacja testów *HPC Challenge Benchmark Suite* została zaprezentowana na

konferencji *Supercomputing 2014* i uzyskała nagrodę *Most elegant solution* w kategorii *HPC Challenge Class 2* [18].

4.2.3 Wydajność przykładowych aplikacji

Przydatność bibliotek i narzędzi do programowania równoległego jest ostatecznie weryfikowane przez uzyskiwaną wydajność aplikacji zrównoleglonych z wykorzystaniem konkretnego narzędzia. W przypadku niniejszej pracy jako przykładowe aplikacji wybrano:

- Pi,
- RayTracer,
- MapReduce.

Równoległe obliczanie przybliżenia liczby π jest klasyczną aplikacją, którą stosuje się do testowania wszystkich rozwiązań równoległych. W pracy zaprezentowano dwa znacząco różniące się skalowalnością sposoby obliczania liczby π.

RayTracer jest przykładową aplikacją posiadającą większe zapotrzebowanie na pamięć operacyjną i jednocześnie często znajduje się w typowych bibliotekach do testowania rozwiązań równoległych. W pracy zaimplementowano aplikację *RayTracer* wzorując się na implementacji dostępnej w *Java Grande Forum Benchmark Suite* [3].

Aplikacja *MapReduce* nie znajduje się w tradycyjnych zestawach aplikacji równoległych. Została ona wybrana ze względu na jej rosnące znaczenie i popularność w obszarze analizy dużych danych.

4.3 Skalowalność

Kolejnym ważnym parametrem jest skalowalność rozwiązań. Nowoczesne rozwiązania programistyczne powinny skalować się do setek tysięcy, a nawet milionów jednostek obliczeniowych. Dobra wydajność obliczeniowa dla niewielkiej liczby rdzeni obliczeniowych, rzędu kilkudziesięciu, nie gwarantuje odpowiedniej wydajności dla setek czy tysięcy jednostek obliczeniowych.

Zanim zostanie opisana skalowalność, zostaną zdefiniowane prostsze miary wydajności obliczeń równoległych jakimi są przyśpieszenie i efektywność. Przyśpieszenie algorytmu równoległego definiujemy jako:

$$S_p = \frac{T_1}{T_p} ,$$

gdzie S_p oznacza przyśpieszenie obliczeń na p procesorach, T_1 oznacza czas rozwiązania zadania algorytmem sekwencyjnym, na pojedynczym procesorze, a T_p oznacza czas rozwiązania zadania algorytmem równoległym przy wykorzystaniu p procesorów. Wartość przyśpieszenia wskazuje ile razy rozwiązanie jest szybsze przy wykorzystaniu p procesorów od rozwiązania korzystającego z 1 procesora.

Efektywność zrównoleglenia opisuje się wzorem:

$$E_p = \frac{S_p}{p} ,$$

gdzie E_p oznacza efektywność obliczeń, S_p to przyśpieszenie, a p to liczba procesorów.

Skalowalność jest miarą bardziej ogólną. Istnieje wiele różnych metryk przedstawiających skalowalność [72, 73]. Ogólnie mówiąc, skalowalność mówi jak zmienia się wydajność w zależności od liczby procesorów. Podobnie jak w algorytmach sekwencyjnych wykorzystuje się złożoność czasową i pamięciową do oceny poszczególnych algorytmów przy zmieniającym się rozmiarze problemu wejściowego, tak w algorytmach równoległych można mówić o skalowalności jako mierze oceny wykorzystanych algorytmów i narzędzi, przy stałym rozmiarze problemu wejściowego i zmieniającej się liczbie jednostek wykonujących obliczenia.

W wielu przypadkach, przy porównywaniu biblioteki PCJ z innymi rozwiązaniami, skalowalność jest najbardziej istotnym parametrem. Testy skalowalności zostały przeprowadzone do 1024-2048 jednostek obliczeniowych, a liczba ta była ograniczona dostępnymi zasobami obliczeniowymi. Testy były przeprowadzane na systemach klastrowych ICM UW w ramach standardowych grantów obliczeniowych, w związku z czym niemożliwe było uzyskanie większej liczby procesorów.

4.4 Łatwość zrównoleglania aplikacji

Łatwość zrównoleglania aplikacji jest to czynnik często decydujący o upowszechnieniu konkretnego rozwiązania – biblioteki czy paradygmatu. Niestety ten czynnik jest trudny do zdefiniowania w sposób mierzalny i obiektywny.

W literaturze znanych jest szereg prób [74, 75] oceniających łatwość programowania. Są próby, które oceniają łatwość programowania poprzez czas niezbędny dla implementacji programu z wykorzystaniem różnych rozwiązań. Inne próby oceniają łatwość programowania w oparciu o opinie programistów. Niestety wszystkie te podejścia obarczone są dużą dozą subiektywizmu i trudno je stosować jako obiektywną miarę.

W niektórych sytuacjach porównuje się liczbę linii kodu przy czym miara ta ma jedynie sens jedynie wówczas, gdy porównywane rozwiązania zostały napisane z wykorzystaniem tego samego języka programowania.

Kolejnym miernikiem może być liczba wywołań funkcji znajdujących się w bibliotece, czy liczba dyrektyw sterujących wykonaniem aplikacji. Tutaj ponownie problemem jest porównywanie rozwiązań napisanych różnych paradygmatach jak OpenMP, MPI czy PGAS.

W przypadku narzędzi do zrównoleglania aplikacji HPC rolę miary łatwości zrównoleglania aplikacji pełni *HPC Challenge Benchmark Suite – Class 2: Most Productivity*, w którym grono ekspertów ocenia zastosowane rozwiązania programistyczne. W 2014 roku biblioteka PCJ zyskała uznanie ekspertów i otrzymała nagrodę: *Most elegant*. W przeszłości nagrodę w tej kategorii uzyskały inne rozwiązania, w tym X10 i Chapel opisane w niniejszej pracy, w rozdziale 1.

Rozdział 5

PCJ – testy wydajnościowe

Testy wydajnościowe (ang. *benchmarks*) biblioteki PCJ przedstawione w niniejszej pracy zostały wykonane na dużych systemach komputerowych znajdujących się w Centrum Nauk Obliczeniowych Interdyscyplinarnego Centrum Modelowania Matematycznego i Komputerowego Uniwersytetu Warszawskiego. Sprawdzają one przede wszystkim szybkość wykonania podstawowych operacji biblioteki PCJ. Z tego względu testy te nazywa się mikrotestami wydajnościowymi (ang. *micro benchmarks*). Testy te nie sprawdzają wydajności złożonych aplikacji, ale oddzielnie testują szybkości wykonania poszczególnych operacji.

5.1 Systemy komputerowe

Do wykonania testów zawartych w niniejszej pracy doktorskiej wykorzystano następujące systemy komputerowe:

- *halo2*,
- *boreasz*,
- *hydra*.

Dostęp do tych systemów był możliwy dzięki wsparciu Interdyscyplinarnego Centrum Modelowania Matematycznego i Komputerowego Uniwersytetu Warszawskiego.

5.1.1 System *halo2*

System *halo2* [76] jest systemem klastrowym składającym się z węzłów, czyli oddzielnych serwerów kasetowych. W skład każdego węzła wchodzą 4 procesory składające się z 4 rdzeni, co daje łącznie 16 jednostek obliczeniowych. Wykorzystane procesory są procesorami 64-bitowymi (architektura x86_64) typu AMD Opteron.

Węzły udostępniają każdemu procesowi od 16 do 32 GB pamięci operacyjnej. Pamięć wykorzystuje architekturę ccNUMA (*cache coherent Non-Uniform Memory Access*). W tej architekturze rdzenie współdzielą pamięć podręczną trzeciego poziomu, a procesory na jednym węźle współdzielą pamięć główną, do której mają niejednolity dostęp. Istnieją również tzw. *grube* węzły składające się z procesorów mających łącznie 64 rdzenie z dostępem do 512 GB pamięci operacyjnej.

Węzły połączone są za pomocą wysokowydajnej sieci Infiniband, w ramach której pracuje mechanizm rDMA (*remote Direct Memory Access*) umożliwiający na bezpośredni dostęp do pamięci innego węzła. Ze względu na brak dysków lokalnych w węzłach, połączenie przez sieć Infiniband wykorzystywane jest również do operacji wejścia/wyjścia na rozproszonym systemie plików Lustre.

System *halo2* działa pod kontrolą systemu operacyjnego z rodziny GNU/Linux.

5.1.2 System *boreasz*

System *boreasz* [77] podobnie jak system *halo2* jest systemem klastrowym. Składa się on z 76 węzłów obliczeniowych. Każdy węzeł obliczeniowy zawiera 4 procesory typu POWER7 o architekturze ppc64 (architektura 64-bitowa) firmy IBM. Pojedynczy procesor posiada 8 rdzeni, a na każdym rdzeniu może działać do 4 wątków fizycznych. Daje to łącznie 32 rdzenie fizyczne składające się łącznie ze 128 rdzeni logicznych.

Każdy węzeł ma dostęp do 128 GB pamięci operacyjnej. Ponadto system umożliwia traktowanie pamięci w wielu węzłach w sposób jednolity w ramach wspólnej przestrzeni adresowej.

Węzły połączone są za pomocą dedykowanej opatentowanej sieci IBM HFI (*Host Fabric Interface*) [78]. Sieć ta pozwala na wydajne wykonywanie operacji na globalnej współdzielonej pamięci. Ponadto każdy węzeł ma dostęp do współdzielonego systemu plików poprzez GPFS.

Systemem operacyjnym zainstalowanym na maszynie *boreasz* jest system AIX. Stąd też jedyną dostępną wersją wirtualnej maszyny Java jest przygotowana specjalnie przez firmę IBM wersja *IBM J9 VM* działająca w oparciu o środowisko *IBM Java SDK, Version 7.0 SR4*.

5.1.3 System *hydra*

System *hydra* [79] jest również systemem klastrowym. Jest to komputer ogólnego przeznaczenia. Składa się zarówno z procesorów firmy AMD jak i z procesorów firmy Intel. Wszystkie procesory na tym klastrze są procesorami 64-bitowymi. Jest to najbardziej

zróżnicowany pod względem budowy klaster spośród systemów użytych do testów. Jego węzły można podzielić na cztery typy zależne od typu procesora:

istanbul – 96 węzłów z 2 procesorami AMD Opteron zawierającymi po 6 rdzeni obliczeniowych; każdy węzeł posiada dostęp do 32 GB pamięci operacyjnej; węzły połączone są między sobą za pomocą sieci Infiniband DDR i za pomocą sieci Ethernet o przepustowości do 1 Gb;

magnycours – 30 węzłów z 4 procesorami AMD Opteron zawierającymi po 12 rdzeni obliczeniowych; każdy węzeł posiada dostęp do 256 GB pamięci operacyjnej; węzły połączone są między sobą za pomocą sieci Ethernet o przepustowości do 10 Gb;

interlagos – 16 węzłów z 4 procesorami AMD Opteron zawierającymi po 16 rdzeni obliczeniowych; każdy węzeł posiada dostęp do 512 GB pamięci operacyjnej; węzły połączone są między sobą za pomocą sieci Ethernet o przepustowości do 10 Gb; schematyczna budowa tego procesora przedstawiona jest na rysunku 5.1;

westmere – 120 węzłów z 2 procesorami Intel Xeon zawierającymi po 6 rdzeni obliczeniowych; każdy węzeł posiada dostęp do 24 GB pamięci operacyjnej; węzły połączone są między sobą za pomocą sieci Infiniband QDR a także za pomocą sieci Ethernet o przepustowości do 1 Gb.

Łącznie system *hydra* posiada 262 węzły obliczeniowe dające łącznie ponad 5000 rdzeni obliczeniowych.

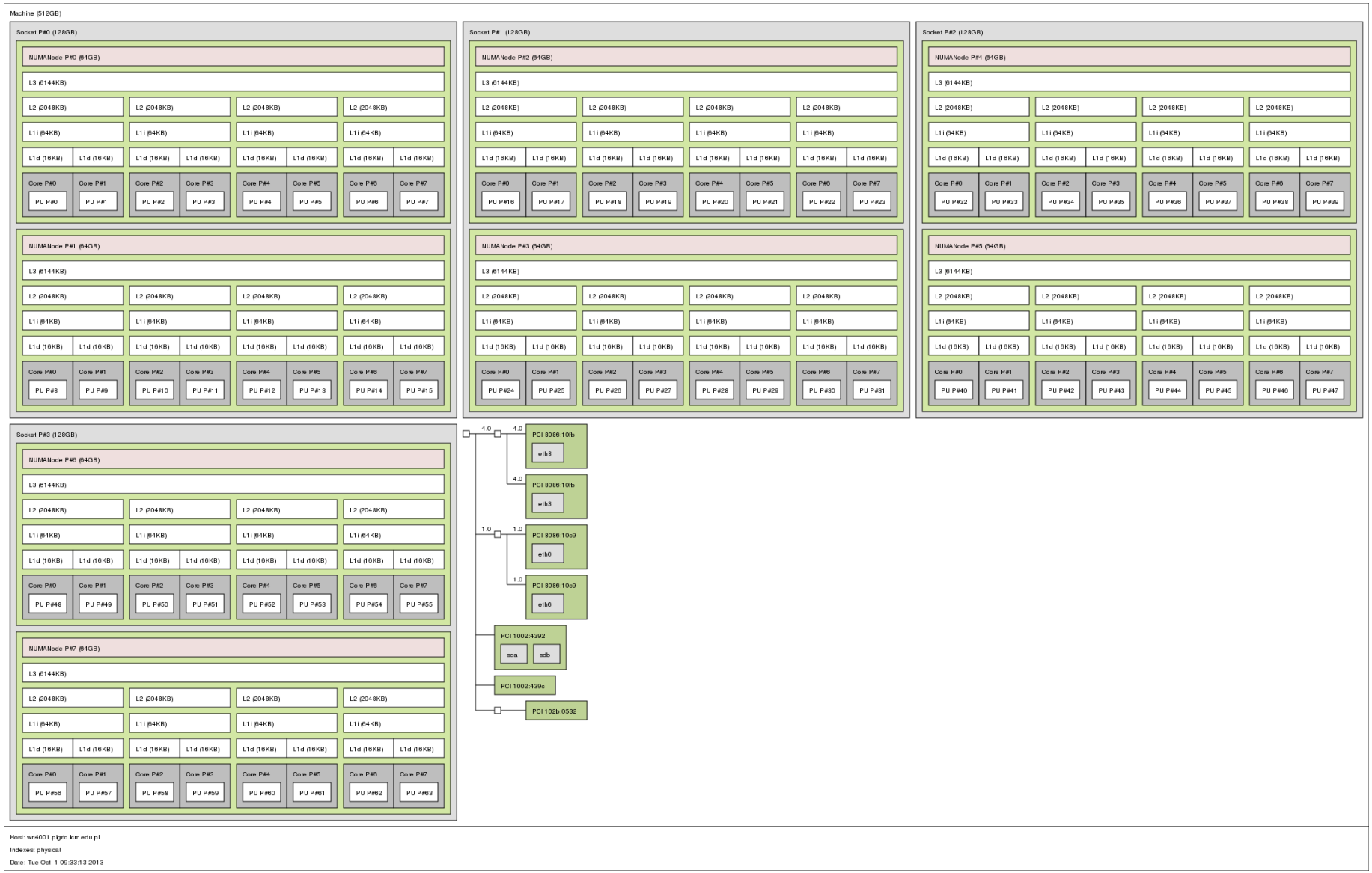
Dostęp do danych współdzielonych na dyskach sieciowych również jest heterogeniczny. W zależności od lokalizacji, do której proces ma zamiar uzyskać dostęp, zastosowany będzie inny system plików: NFS lub Lustre.

Węzły obliczeniowe klastra *hydra* działają pod kontrolą systemu operacyjnego z rodziny GNU/Linux.

5.2 Testy wydajnościowe

W niniejszej pracy zostaną przedstawione wyniki wydajnościowe uzyskane dla poniższych typów mikrotestów wydajnościowych:

- Barrier,
- Ping-pong,
- Broadcast,
- Reduce.



Rysunek 5.1: *Schemat budowy procesora* AMD Opteron™ Processor 6272 (Interlagos)
(źródło: `https://www.icm.edu.pl/kdm_wiki/images/0/01/Hydra-interlagos.png` *[dostęp: 9.03.2014])*

W tej sekcji zostaną przedstawione wyniki małych testów wydajnościowych zebrane z wykorzystaniem biblioteki PCJ. Testy uruchamiano na opisanych wcześniej systemach klastrowych. Do porównania wydajności PCJ wykorzystano ugruntowane w świecie HPC (*High-Performance Computing*) rozwiązanie bazujące na protokole MPI w implementacji OpenMPI. Kod programów w MPI bazowany był na odpowiednich kodach źródłowych programów zapisanych w języku Java z wykorzystaniem biblioteki PCJ. W większości przypadków PCJ uruchamiano z wykorzystaniem wirtualnej maszyny Java w implementacji Oracle w wersji 1.7.0.

Wykonano również testy wydajnościowe z wykorzystaniem biblioteki programistycznej ProActive Programming w wersji 6.0.1 dostępnej w postaci kodu źródłowego [62]. W tym przypadku skorzystano z wirtualnej maszyny Java w implementacji Oracle w wersji 1.8.0. Wykorzystano również domyślny protokół komunikacyjny ProActive jakim jest protokół RMI. W celu uproszczenia opisu uzyskanych wyników, aktywne obiekty w ProActive utożsamiano z wątkami obliczeń.

Wszystkie testy zliczały czas 100 powtórzeń głównej pętli, po czym wynik uśredniał czas wykonania, tak by uzyskać czas wykonania tylko jednej iteracji. Operację tą wykonywano pięciokrotnie i jako wynik wybierano najmniejszą uzyskaną w ramach wszystkich powtórzeń wartość czasu.

Wynik zbierano dla różnej liczby węzłów oraz różnej liczby wątków na węzeł. Liczba węzłów zawsze była potęgą dwójki, czyli: 1, 2, 4, 8 i 16 węzłów. Liczba wątków na węźle również była potęgą dwójki i wynosiła odpowiednio: 1, 2, 4, 8, 16 i 32 wątki. W przypadku, gdy liczba dostępnych rdzeni na wykorzystywanym węźle nie była potęgą dwójki, uruchomiano również tyle wątków aplikacji ile znajdowało się rdzeni na węźle. Na podstawie liczby węzłów (n_{nodes}) i liczby wątków na węźle ($n_{theards}$) obliczono całkowitą liczbę wątków w trakcie obliczeń (n):

$$n = n_{nodes} \cdot n_{threads}$$

Jak widać, niektóre wartości można uzyskać na wiele sposobów, np. $n = 64$ można uzyskać w następujący sposób:

- $n_{nodes} = 2, \quad n_{threads} = 32,$
- $n_{nodes} = 4, \quad n_{threads} = 16,$
- $n_{nodes} = 8, \quad n_{threads} = 8,$
- $n_{nodes} = 16, \quad n_{threads} = 4.$

Z tego powodu wyniki pogrupowano według całkowitej liczby wątków i wybrano dla nich jedynie najmniejszą uzyskaną eksperymentalnie wartość czasu lub największą szybkość, czyli wynik najbardziej optymistyczny.

W przypadku testów w PCJ jak również z wykorzystaniem biblioteki ProActive, do mierzenia czasu, który upłynął, zastosowano metodę `System.nanoTime()`, która zwraca wartość czasu liczoną w nanosekundach od pewnego dowolnie wybranego, ale stałego dla danego uruchomienia wirtualnej maszyny Java, momentu. Listing 5.1 przedstawia sposób wyznaczania upływu czasu.

```
1    long start = System.nanoTime();
2
3    /* ... some time consuming operations ... */
4
5    long stop = System.nanoTime();
6    double elapsedTime = (stop - start) * 1e-9;
```

Listing 5.1: *Obliczenie upływu czasu w sekundach w języku Java*

Liczenie upływu czasu w przypadku testów napisanych w MPI było analogiczne – zastosowano funkcję `MPI_Wtime()` zwracającą liczbę sekund, która upłynęła od pewnego momentu w przeszłości. Wartość ta jest wyrażana w sekundach, ale jest to wartość wyrażona typem zmiennoprzecinkowym, a sama metoda jest w zamierzeniu wysokorozdzielcza. Kod przedstawiony na listingu 5.2 pokazuje sposób wyznaczania czasu, który upłynął na wykonywanie pewnych operacji.

```
1    double start = MPI_Wtime();
2
3    /* ... some time consuming operations ... */
4
5    double stop = MPI_Wtime();
6    double elapsedTime = stop - start;
```

Listing 5.2: *Obliczenie upływu czasu w sekundach z wykorzystaniem MPI*

W wynikach uwzględniono istotny rozmiar danych przeliczony na podstawie liczby przesyłanych elementów i ich wielkości, bez uwzględnienia nagłówków wiadomości. Przykładowo, przesłanie tablicy zawierającej 100 elementów typu double, to przesłanie wiadomości o rozmiarze 800 B. W rzeczywistości rozmiar przesłanej wiadomości jest większy.

5.2.1 Barrier

Barrier, czyli bariera, to pierwszy omawiany mikrotest wydajnościowy. Jego zadaniem jest zliczenie czasu wykonania pojedynczej operacji bariery w zależności od liczby wątków obliczeniowych biorących udział w barierze. Na listingu 5.3

przedstawiono kod źródłowy programu służącego do badania wydajności operacji bariery z wykorzystaniem biblioteki PCJ.

```
1     @Override
2     public void main() {
3         int number_of_tests = 5;
4         int ntimes = 100;
5
6         PCJ.barrier();
7
8         double tmin = Double.MAX_VALUE;
9         for (int k = 0; k < number_of_tests; k++) {
10            long rTime = System.nanoTime();
11
12            for (int i = 0; i < ntimes; i++) {
13                PCJ.barrier();
14            }
15
16            rTime = System.nanoTime() - rTime;
17            double t = (rTime / (double) ntimes) * 1e-9;
18
19            if (tmin > t) {
20                tmin = t;
21            }
22
23            PCJ.log(PCJ.threadCount() + " " + t);
24            PCJ.barrier();
25        }
26
27        if (PCJ.myId() == 0) {
28            System.out.format(Locale.FRANCE,
29                    "%5d \t time %7f%n",
30                    PCJ.threadCount(), tmin);
31        }
32    }
```

Listing 5.3: *Kod źródłowy testu* Barrier

Przedstawiona metoda `public void main()` jest punktem startowym instrukcji dla uruchomienia PCJ. Metoda ta oznaczona jest adnotacją `@Override` pokazującą, że nadpisuje, a w zasadzie implementuje, ona deklarację tej metody z interfejsu `pl.umk.mat.pcj.StartPoint`. Zawiera ona dwie pętle. Zewnętrzna pętla jest odpowiedzialna za powtórzenie testu odpowiednią liczbę razy. Pętla wewnętrzna wywołuje operację bariery stukrotnie. Czas na jeden test liczony jest jako uśredniony czas jednej

bariery. Jako wynik brany jest minimalny czas jaki potrzeba na wykonanie bariery. Każdy wątek zaangażowany w test wyświetla również swój czas na ekranie przekazując wiadomość do wątku 0 za pomocą metody `PCJ.log()`. Po zakończeniu pojedynczego testu następuje bariera przed kolejnym wywołaniem testu. Na sam koniec wątek 0 wypisuje na ekran najlepszy wynik korzystając z francuskich ustawień regionalnych, czyli ustawień, w których znakiem rozdzielającym część całkowitą od części ułamkowej liczby jest przecinek.

Metoda `public static void main(String args[])`, przedstawiona na listingu 5.4, jest standardową metodą startową dla programów napisanych w języku Java. W niniejszym przykładzie przedstawiono w jaki sposób, w jednym programie uruchomić wiele razy obliczenia z wykorzystaniem PCJ z różną liczbą użytych węzłów i wątków na każdy węzeł. Założono przy tym, że publiczna klasa `PcjMicroBenchmarkBarrier`, rozszerza klasę `pl.umk.mat.pcj.Storage` i implementuje interfejs `pl.umk.mat.pcj.StartPoint`.

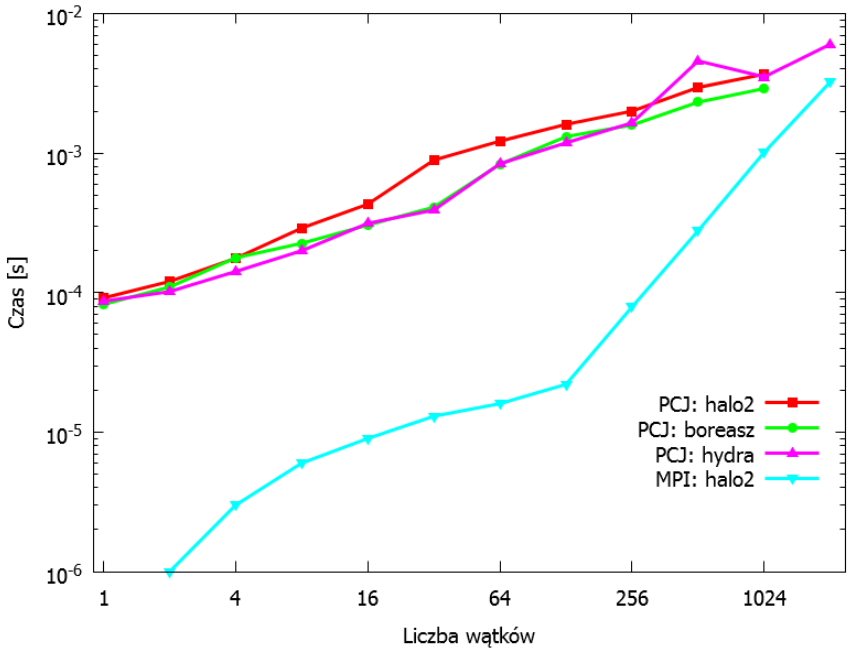
Tablica `threads` przechowuje informacje o liczbie wątków na węzeł w poszczególnych uruchomieniach. Następnie następuje wczytanie opisów możliwych do wykorzystania węzłów z pliku *nodes.txt*. Jeśli plik nie istnieje, to wyświetlany jest stosowny komunikat i program kończy działanie z odpowiednim kodem wyjścia oznaczającym błąd. W przypadku odczytania opisów węzłów, następują dwie pętle. Zewnętrzna pętla iterująca z wykorzystaniem zmiennej `nn` informuje o liczbie węzłów do użycia w trakcie testu, natomiast wewnętrzna pętla służy do zmiany liczby wątków jakie mają być uruchomione na pojedynczym węźle. W wewnętrznej pętli ma miejsce kopiowanie danych do tablicy `nodes`, czyli do tablicy przechowującej nazwy węzłów do użycia w obliczeniach. Po zakończeniu wewnętrznej pętli następuje uruchomienie zadania na podanych węzłach z odpowiednią liczbą wątków na węzeł. Dzięki temu mechanizmowi możliwe jest uruchomienie jednej aplikacji, która w zależności od sytuacji może uruchomić różną liczbę wątków PCJ.

Wykresy przedstawione na rysunku 5.2 pokazują czas wykonywania pojedynczej operacji bariery w zależności od systemu klastrowego użytego do testów. Na osi poziomej wyrażono liczbę użytych wątków, natomiast oś pionowa oznacza czas wykonania operacji bariery. Obie osie przedstawione są w skali logarytmicznej – pozioma z podstawą 2, pionowa z podstawą 10. Do rysunku dodano wykres przedstawiający czas wykonania pojedynczej bariery w MPI. W MPI, podobnie jak w PCJ, uruchamiano odpowiednią operację w pętli i powtarzano wielokrotnie. Odpowiedni fragment kodu przedstawiony jest na listingu 5.5.

Na rysunku 5.2 widać, że czas pojedynczej bariery dla PCJ jest o kilka rzędów wielkości większy niż w przypadku MPI, zwłaszcza dla małej liczby wątków. Jednak wraz ze wzrostem liczby wątków różnica ta znacząco maleje i widać wyraźny trend, który sugeruje, że dalej zwiększając liczbę wątków, bariera w wykonaniu PCJ będzie wykonywać się szybciej, niż miałoby to miejsce w przypadku MPI. Ponadto widać, że czas wykonania bariery w PCJ jest niezależny od architektury klastra. Można również zauważyć, różne

```
1   public static void main(String[] args) {
2       int[] threads = {1, 2, 4, 8, 16, 32};
3
4       Set<String> nodesSet = new LinkedHashSet<>();
5       try (Scanner s = new Scanner(new File("nodes.txt"))) {
6           while (s.hasNextLine()) {
7               String node = s.nextLine();
8               nodesSet.add(node);
9           }
10      } catch (IOException ex) {
11          Logger.getLogger(
12                  PcjMicroBenchmarkBarrier.class.getName())
13                  .log(Level.SEVERE,
14                          "Unable to load descriptor file",
15                          ex);
16          System.exit(1);
17      }
18
19      String[] nodesUniq = nodesSet.toArray(new String[0]);
20
21      for (int nn = nodesUniq.length; nn > 0; nn = nn / 2) {
22          for (int nt : threads) {
23              String[] nodes = new String[nt * nn];
24              System.out.printf(" Start deploy nn=%d nt=%d",
25                      nn, nt);
26              int ii = 0;
27              for (int i = 0; i < nn; i++) {
28                  for (int j = 0; j < nt; j++, ii++) {
29                      nodes[ii] = nodesUniq[i];
30                  }
31              }
32              PCJ.deploy(PcjMicroBenchmarkBarrier.class,
33                      PcjMicroBenchmarkBarrier.class,
34                      nodes);
35          }
36      }
37  }
```

Listing 5.4: *Kod źródłowy metody startowej dla przykładu* Barrier



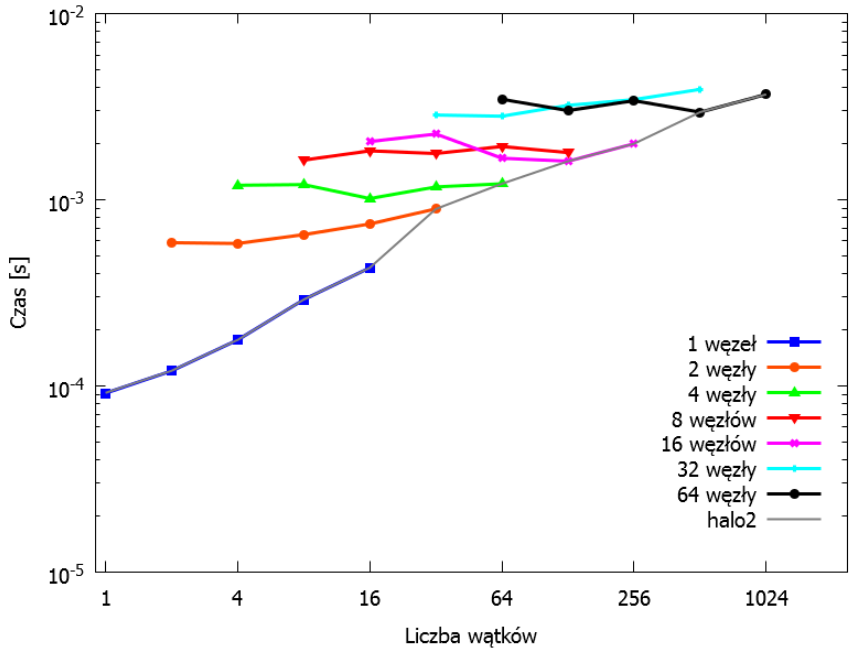
Rysunek 5.2: *Czas wykonywania pojedynczej operacji bariery w zależności od liczby wątków*

```
1     for (j = 0; j < ntimes; j++) {
2         MPI_Barrier(MPI_COMM_WORLD);
3     }
```

Listing 5.5: *Wykonywanie operacji bariery w MPI*

skalowanie czasu wykonania w przypadku MPI dla liczby wątków mniejszej niż 128 i liczby wątków większej od 128. Dla liczby wątków większej niż 128, czas wykonania skaluje się liniowo w stosunku do liczby wątków, a dla liczby wątków mniejszej niż 128 skalowanie jest lepsze, to znaczy: początkowo przy szesnastokrotnym wzroście liczby wątków (pomiędzy 4 a 64 wątkami), czas działania operacji bariery wzrósł ponad pięciokrotnie, a dalej, przy zachowaniu wzrostu liczby wątków, to jest pomiędzy 8 a 128 wątkami, czas wykonywania operacji bariery wzrósł jedynie czterokrotnie. Natomiast PCJ skaluje się generalnie liniowo, ale wzrost jest zdecydowanie słabszy niż w przypadku MPI, więc skalowanie w przypadku PCJ jest lepsze niż w przypadku MPI.

Rysunek 5.3 przedstawia wynik testu porównującego czas wykonania pojedynczej operacji bariery w zależności od liczby węzłów na klastrze halo2. Obie osie mają takie same znaczenie jak poprzednio – oś pozioma oznacza liczbę wątków biorących udział w teście, w skali logarytmicznej z podstawą 2, a na osi pionowej wyrażono czas operacji bariery w skali logarytmicznej przy podstawie 10. Każda seria danych obrazuje uruchomienie testu dla różnej liczby węzłów. Widać, że przy takiej samej liczbie



Rysunek 5.3: *Czas wykonywania pojedynczej operacji bariery w zależności od liczby węzłów*

wątków, operacja bariery działa szybciej, jeśli jest wykonana z wykorzystaniem mniejszej liczby węzłów, co jest zgodne z oczekiwaniami – komunikacja i synchronizacja w ramach pojedynczego węzła, dla pojedynczej maszyny wirtualnej Java, jest szybsza niż pomiędzy węzłami.

Biblioteka ProActive nie wspiera operacji bariery znanej z innych rozwiązań opartych o SPMD. Jednak stosując mechanizmy `@ImmediateService` i SPMD dostępne w ProActive oraz `java.util.concurrent.CyclicBarrier` z platformy Java, można zasymulować blokującą operację bariery.

```
1  public class ActiveObject {
2      private CyclicBarrier barrier;
3
4      public BooleanWrapper setupBarrier() {
5          if (PASPMD.getMyRank() == 0) {
6              barrier = new CyclicBarrier(
7                      PASPMD.getMySPMDGroupSize());
8              return new BooleanWrapper(true);
9          }
10         return new BooleanWrapper(false);
11     }
12
```

Ciąg dalszy na następnej stronie

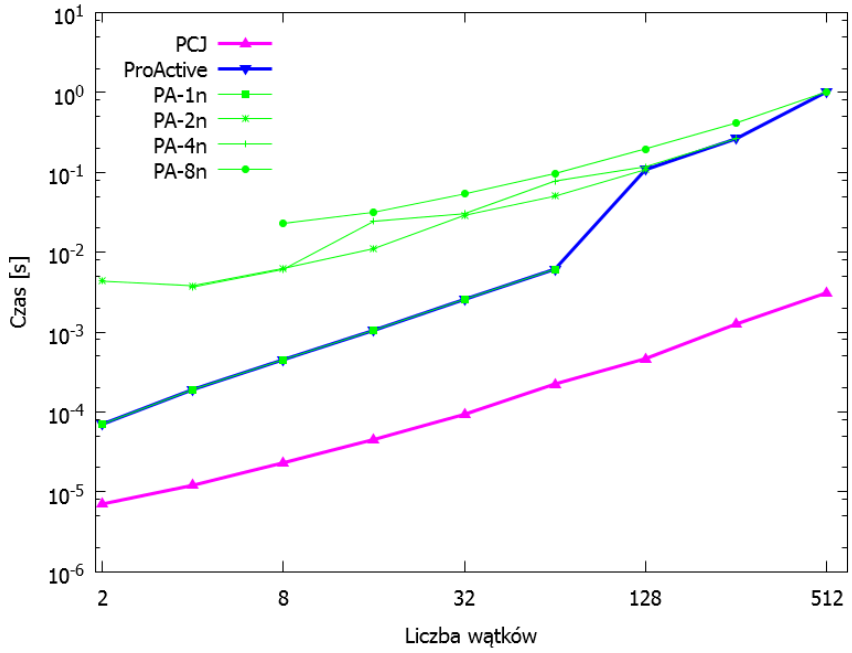
Ciąg dalszy z poprzedniej strony

```
13      @ImmediateService
14      public void barrier() throws Exception {
15          barrier.await();
16      }
17
18      public DoubleWrapper run() throws Exception {
19              ActiveObject[] activeObjects =
20                      PAGroup.getGroup(PASPMD.getSPMDGroup())
21                              .toArray(new ActiveObject[0]);
22
23          final int ntimes = 100;
24          final int number_of_tests = 5;
25          double tmin_barrier = Double.MAX_VALUE;
26
27          for (int k = 0; k < number_of_tests; k++) {
28
29              long time = System.nanoTime();
30              for (int i = 0; i < ntimes; i++) {
31                  activeObjects[0].barrier();
32              }
33              time = System.nanoTime() - time;
34
35              double dtime = (time / (double) ntimes) * 1e-9;
36
37              if (PASPMD.getMyRank() == 0
38                      && tmin_barrier > dtime) {
39                  tmin_barrier = dtime;
40              }
41          }
42
43          return new DoubleWrapper(tmin_barrier);
44      }
45  }
```

Listing 5.6: *Aktywny obiekt pozwalający na symulowanie blokującej operacji bariery*

Listing 5.6 przedstawia przykładowy aktywny obiekt, który może być wykorzystywany do wykonania blokującej operacji bariery. Na początek wyznaczony wątek, w tym przypadku wątek o numerze 0, wywołując metodę `setupBarrier()`, tworzy instancję obiektu `CyclicBarrier` przekazując w argumencie konstruktora rozmiar grupy. Następnie, w momencie, gdy następuje czas na wykonanie operacji bariery, każdy wątek wywołuje metodę `barrier()` na wątku o numerze 0. Dzięki temu, iż jest to metoda,

która może rzucać *sprawdzany wyjątek* (ang. *checked exception*), wywołanie tej metody następuje w sposób synchroniczny. Gdy wszystkie wątki wykonają tę metodę, to zostanie zwolniona blokada i wszystkie wątki będą mogły pracować dalej.



Rysunek 5.4: *Porównanie czasu wykonywania pojedynczej operacji bariery dla PCJ i ProActive (hydra)*

Rysunek 5.4 przedstawia porównanie czasu wykonania pojedynczej operacji bariery dla różnej liczby wątków przy korzystaniu z bibliotek: PCJ i ProActive. Wykorzystano różne węzły klastra hydra (1 węzeł typu *istanbul*, 3 węzły typu *westmere*, 2 węzły typu *magnycours* oraz 2 węzły typu *interlagos*), a na każdym uruchomiono po 1, 2, 4, 8, 16, 32 i 64 wątki wykonujące operację bariery. W przypadku biblioteki ProActive zastosowano operację bariery zasymulowaną za pomocą kodu widocznego na listingu 5.6. Na wykresie widać, że czas działania dla obu bibliotek skaluje się generalnie liniowo, ze zbliżonym współczynnikiem kierunkowym, czyli z tym samym nachyleniem prostej. Jednak czas działania pojedynczej operacji bariery w przypadku PCJ jest dziesięciokrotnie mniejszy niż w przypadku ProActive niezależnie od liczby wątków. Dodatkowo w przypadku wyników ProActive, występuje jeden skok pomiędzy 64 a 128 wątkami. Ten skok nie występuje przy korzystaniu z PCJ. Jest to spowodowane przejściem pomiędzy wynikami uzyskanymi dla 2 i 4 węzłów. Na wykresie zaznaczono również czasy wykonania operacji bariery dla różnej liczby węzłów (1, 2, 4 i 8 węzłów; oznaczone na wykresie jako, odpowiednio: *PA-1n*, *PA-2n*, *PA-4n* i *PA-8n*) przy wykorzystaniu biblioteki ProActive, dzięki czemu łatwiej można wywnioskować czym spowodowany jest skok, jak również utwierdzić się w konkluzji mówiącej o tym, że przy takiej samej liczbie wątków,

operacja bariery działa szybciej, jeśli jest wykonywana z wykorzystaniem mniejszej liczby węzłów.

5.2.2 Ping-pong

Test typu *ping-pong* jest najprostszym typem testu sprawdzającym szybkość przesyłania danych pomiędzy dwoma wątkami obliczeń w zależności od rozmiaru przesyłanych danych. W test ping-pong zaangażowane są jedynie dwa wątki – w najprostszej postaci jeden wątek jest wątkiem wysyłającym dane, a drugi wątek jest wątkiem odbierającym dane. W przypadku testów wykonywanych do niniejszej pracy przekazywana była tablica zmiennych typu `double`.

Wykorzystując bibliotekę PCJ test ping-pong można wykonać na wiele sposobów. W pracy przedstawiono trzy możliwości:

- jednostronne blokujące pobieranie danych od przeciwnego wątku obliczeń,
- jednostronne nieblokujące wysyłanie danych do przeciwnego wątku obliczeń,
- naprzemienne blokujące wysyłanie danych do przeciwnego wątku obliczeń.

Kod przedstawiony na listingu 5.7 przedstawia szkielet programu wykorzystanego do przeprowadzenia testów ping-pong. We wszystkich przypadkach wykonano 5 testów (zmienna `number_of_tests`). Każdy test składał się ze 100 powtórzeń (zmienna `ntimes`) przesyłania tablicy. Tablica `transmit` przechowuje rozmiar tablicy do przesłania między węzłami w teście. Wynikiem testu jest uśredniona wartość czasu potrzebnego na wykonanie wszystkich powtórzeń wewnętrznej pętli. Po zakończeniu obliczeń i zebraniu średnich wyników, wybierana jest najmniejsza wartość czasu i drukowana jest na ekranie wraz z rozmiarem przesłanych danych liczonych w kB:

$$rozmiar = \frac{n * 8\text{B}}{1024} = \frac{n\text{B}}{128},$$

gdzie 8B oznacza ilość pamięci potrzebnej do przechowania wartości zmiennej typu double, czyli 8 bajtów, gdyż typ `double` jest typem 64-bitowym.

Kod odpowiedzialny za jednostronne pobieranie danych od przeciwnego wątku obliczeń przedstawiono na listingu 5.8. Pominięto w nim sekcję importów użytych klas. W pierwszej części opisanej przez komentarz `prepare data` następuje przygotowanie, przez wątek 1, tablicy długości n, po czym wypełniana jest ona liczbami naturalnymi zaczynając od 1. Następnie we fragmencie rozpoczynającym się od komentarza `test body` przedstawiony jest główny kod testu. Jeśli aktualnym wątkiem jest wątek 0, wykonuje on `ntimes` powtórzeń pętli, w której następuje wywołanie metody `PCJ.get()` pobierającej z wątku 1 wartość zmiennej `a` przechowywanej w magazynie. Metoda ta jest blokująca – wątek 0 nie ma możliwości wykonania kolejnej instrukcji, dopóki nie otrzyma odpowiedzi od wątku 1.

```
7  public class PcjMicroBenchmarkPingPongTemplate extends Storage
8          implements StartPoint {
9      @Shared
10     double[] a;
11
12     @Override
13     public void main() {
14         int[] transmit = {1, 10, 100, 1_024, 2_048, 4_096,
15             8_192, 16_384, 32_768, 65_536, 131_072, 262_144,
16             524_288, 1_048_576, 2_097_152};
17         int ntimes = 100;
18         int number_of_tests = 5;
19
20         for (int n : transmit) {
21             /* ... prepare data ... */
22
23             PCJ.barrier();
24
25             double tmin = Double.MAX_VALUE;
26             for (int k = 0; k < number_of_tests; k++) {
27                 long rTime = System.nanoTime();
28
29                 /* ... test body ... */
30
31                 rTime = System.nanoTime() - rTime;
32                 double t = ((double) rTime / ntimes) * 1e-9;
33
34                 if (tmin > t) tmin = t;
35                 PCJ.barrier();
36             }
37
38             if (PCJ.myId() == 0) {
39                 System.out.format(Locale.FRANCE,
40                   "%5d \t size %10f KB \t time %7f%n",
41                   PCJ.threadCount(), (double) n / 128, tmin);
42             }
43         }
44     }
45 }
```

Listing 5.7: *Szkielet programu ping-pong*

```
1     /* ... prepare data ... */
2     if (PCJ.myId() == 1) {
3          a = new double[n];
4          for (int i = 0; i < n; i++) {
5              a[i] = i + 1;
6          }
7     }
8     /* ... test body ... */
9     if (PCJ.myId() == 0) {
10         for (int i = 0; i < ntimes; i++) {
11             double[] b = PCJ.get(1, "a");
12         }
13    }
```

Listing 5.8: *Kod jednostronnego blokującego pobierania danych od przeciwnego wątku obliczeń*

Jednostronne nieblokujące wysyłanie danych do przeciwnego wątku obliczeń przedstawione jest na listingu 5.9. W części odpowiedzialnej za przygotowanie danych deklarowana jest zmienna `b`, która w wątku 0 przyjmuje za wartość tablicę n-elementową wypełnioną kolejnymi liczbami naturalnymi począwszy od 1. Ponadto w tej części wątek 1 czyści status modyfikacji zmiennej współdzielonej `a`. Natomiast w części głównej testu następuje `ntimes` powtórzeń pętli, w której wątek 0 wysyła do magazynu wątku 1, do zmiennej współdzielonej `a`, tablicę `b`. W tym czasie wątek 1 oczekuje na otrzymanie tych danych. Oczekiwanie nie ma wpływu na pracę wątku 0, więc wysyłanie danych odbywa się w sposób nieblokujący, asynchroniczny. Wątek 0 niezależnie od tego, czy wątek 1 zdążył odebrać dane czy nie, inicjuje kolejny transfer danych. W ten sposób możliwe jest nakładanie się odbioru danych w wątku 1. Policzony czas nie uwzględnia czasu odbioru danych przez wątek 1.

Ostatnim przykładem jest modyfikacja nieblokującego wysyłania danych, w którym oba wątki naprzemiennie wysyłają dane. Dzięki temu transfery danych nie nakładają się na siebie, ale następuje blokowanie postępu pętli do czasu odebrania danych przez przeciwny wątek. Najważniejsze fragmenty kodu odpowiedzialnego za ten test przedstawiono na listingu 5.10. Oba wątki tworzą tablicę `b` i wypełniają ją kolejnymi liczbami naturalnymi począwszy od 1, a także zerują status monitorowania zmiennej współdzielonej `a`. Następnie w głównym kodzie testu, następuje `ntimes` powtórzeń pętli, gdzie wątki, w zależności od numeru powtórzenia stają się stroną wysyłającą i odbierającą. Wątek 0 jest stroną wysyłającą dla `i` parzystego a wątek 1 dla `i` nieparzystego. W przypadku `ntimes` parzystego, a taka sytuacja ma miejsce dla niniejszych testów, w ostatnim powtórzeniu pętli, wątek 0 oczekuje na odebranie danych

```
1     /* ... prepare data ... */
2     double[] b = null;
3     if (PCJ.myId() == 0) {
4         b = new double[n];
5         for (int i = 0; i < n; i++) {
6             b[i] = i + 1;
7         }
8     } else {
9         PCJ.monitor("a");
10    }
11    /* ... test body ... */
12    for (int i = 0; i < ntimes; i++) {
13        if (PCJ.myId() == 0) {
14            PCJ.put(1, "a", b);
15        } else {
16            PCJ.waitFor("a");
17        }
18    }
```

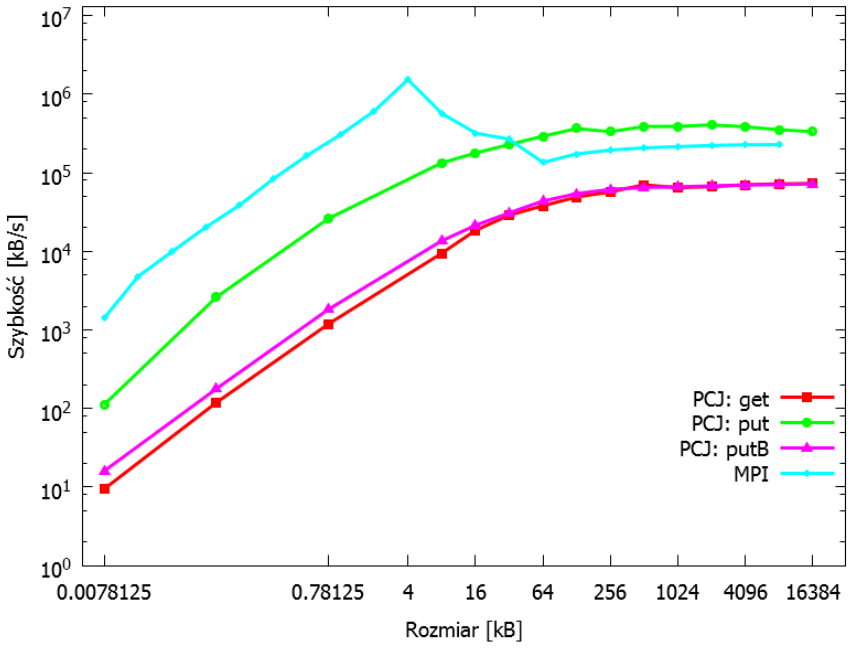
Listing 5.9: *Kod jednostronnego nieblokującego wysyłania danych do przeciwnego wątku obliczeń*

przesłanych przez wątek 1, stąd policzony czas jest czasem pełnych `ntimes` powtórzeń pętli.

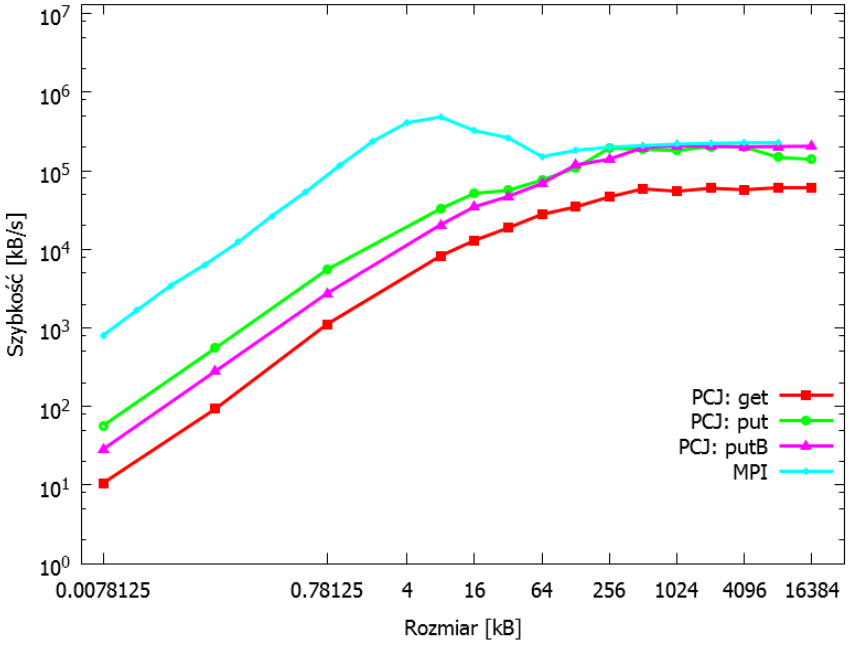
```
1      /* ... prepare data ... */
2      double[] b = new double[n];
3      for (int i = 0; i < n; i++) {
4          b[i] = i + 1;
5      }
6      PCJ.monitor("a");
7      /* ... test body ... */
8      for (int i = 0; i < ntimes; i++) {
9          if (PCJ.myId() == i % 2) {
10             PCJ.put((i + 1) % 2, "a", b);
11         } else {
12             PCJ.waitFor("a");
13         }
14     }
```

Listing 5.10: *Kod naprzemiennego blokującego wysyłania danych do przeciwnego wątku obliczeń*

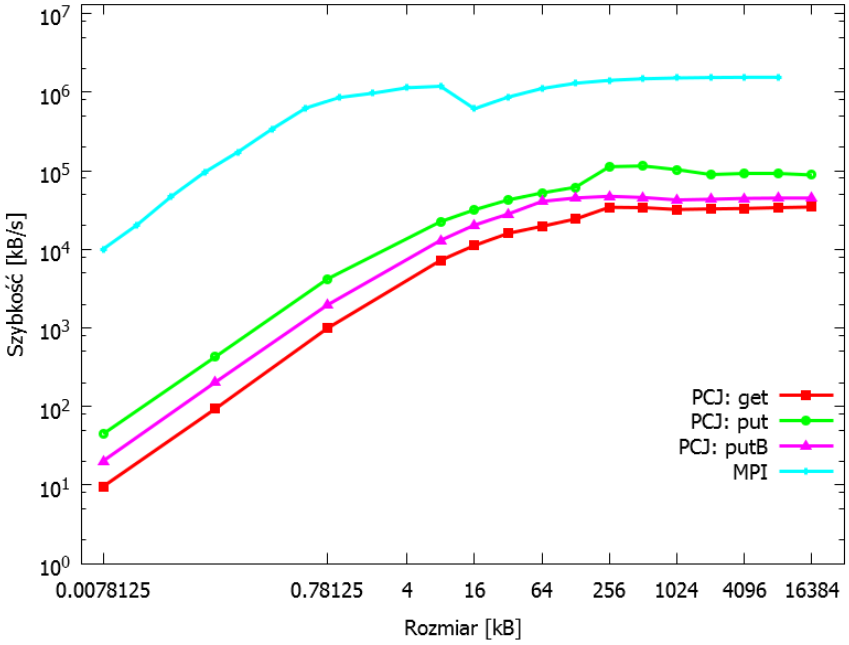
Dwa węzły po jednym wątku



(a) *hydra – węzeł typu* westmere



(b) *hydra – węzeł typu* istanbul



(c) *halo2*

Rysunek 5.5: *Szybkość przesyłania danych między dwoma wątkami na dwóch węzłach w zależności od rozmiaru wiadomości i wykorzystanego mechanizmu*

Wykresy przedstawione na rysunku 5.5 przedstawiają szybkość przesyłania danych między dwoma wątkami znajdującymi się na dwóch węzłach. Każdy wykres zawiera trzy serie danych odpowiadające użytym metodom:

get – oznacza jednostronne blokujące pobieranie danych od przeciwnego wątku obliczeń;

put – oznacza jednostronne nieblokujące wysyłanie danych do przeciwnego wątku obliczeń;

putB – oznacza naprzemienne blokujące wysyłanie danych do przeciwnego wątku obliczeń.

Na wykresach widać, że najszybszą metodą na przekazywanie danych pomiędzy wątkami na dwóch węzłach, jest metoda *put*. Wiąże się to z tym, że w przypadku tej metody, wątek wysyłający dane nie zwraca uwagi na to, czy wątek odbierający dane zdążył je odebrać i przetworzyć, czy też nie. Metoda *get* jest najwolniejsza, gdyż zawsze wiąże się z przesłaniem zapytania, otrzymaniem i przetworzeniem odpowiedzi a także z mechanizmami synchronizacji wątków wewnątrz wirtualnej maszyny Java. Metoda *putB* jest metodą pośrednią, w której wątki naprzemiennie są wątkami odbierającymi i wysyłającymi dane.

W przypadku przesyłania danych o wielkości powyżej 256 KB uzyskano maksymalną prędkość zarówno dla MPI jak i dla PCJ. W przypadku PCJ prędkość ta wynosi nawet do 400 MB/s dla metody *put* na klastrze *hydra* w przypadku rozmiaru wiadomości wynoszącemu 2 MB.

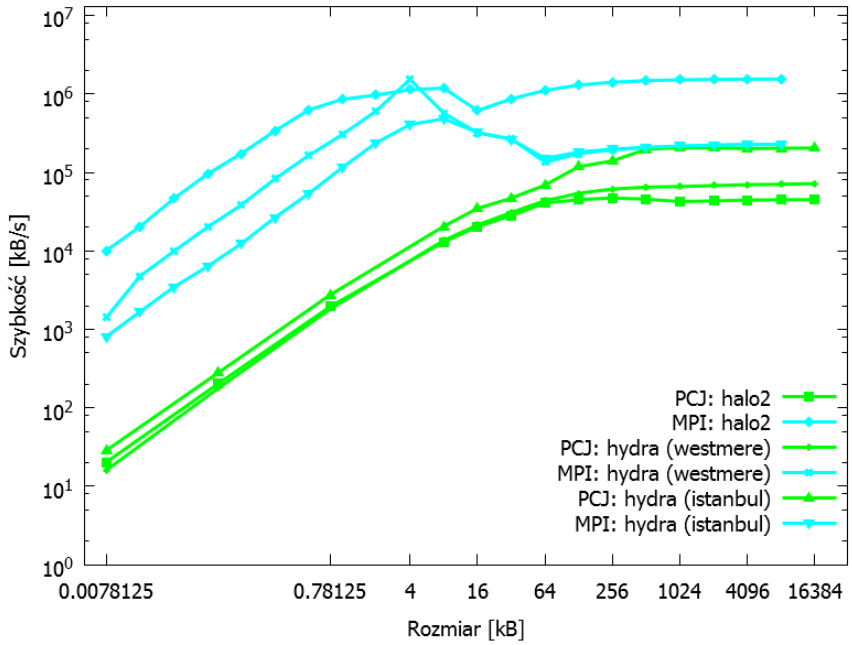
Widać również, że w zależności od wykorzystanego klastra obliczeniowego, a także w zależności od typu węzła, metoda *putB* osiąga różne wyniki. Na klastrze *hydra* wykorzystując węzły wyposażone w procesor *westmere* metoda ta ma osiągi zbliżone do metody *get*, natomiast na węzłach posiadających procesor *istanbul* wyniki są podobne do metody *put*. Dodatkowo na klastrze *halo2* metoda *putB* osiąga wyniki pośrednie.

Porównanie wyników zebranych na różnych architekturach dla metody *putB* z wynikami uzyskanymi za pomocą MPI przedstawione jest na rysunku 5.6. Kod głównej pętli wykorzystujący MPI znajduje się na listingu 5.11. Działa on w podobny sposób jak naprzemienne przesyłanie i odbieranie danych między wątkami w przypadku testu *putB* z PCJ.

```
1     for (j = 0; j < ntimes; j++) {
2         if (rank == j % 2) {
3             MPI_Send(buf, n, MPI_DOUBLE,
4                 (j + 1) % 2, k, MPI_COMM_WORLD);
5         } else {
6             MPI_Recv(buf, n, MPI_DOUBLE,
7                 j % 2, k, MPI_COMM_WORLD, &status);
8         }
9     }
```

Listing 5.11: *Test ping-pong w MPI*

Osiągane szybkości dla PCJ są znacząco zależne od systemu klastrowego. W przypadku naprzemiennego przesyłania danych (metoda *putB*), dla klastra *halo2*



Rysunek 5.6: *Szybkość przesyłania danych między dwoma wątkami na dwóch węzłach w zależności od rozmiaru wiadomości*

maksymalna prędkość wynosi blisko 50 MB/s, dla klastra *hydra* dla węzłów typu *westmere* maksymalna prędkość to niecałe 80 MB/s, a dla węzłów typu *istanbul* maksymalna prędkość wynosi ponad 200 MB/s. W przypadku MPI maksymalna uzyskiwana prędkość również zależy od użytego systemu klastrowego. Maksymalna prędkość dla systemu *halo2* oscyluje w okolicy 1 GB/s. W przypadku klastra *hydra*, zarówno dla węzłów składających się z procesorów Intel jak i AMD maksymalna prędkość wynosi około 200 MB/s.

Różne wyniki związane są z różnymi architekturami procesorów oraz z różną technologią stosowaną do połączenia procesorów, co dotyczy nie tylko klastrów, ale także ich poszczególnych części. Należy pamiętać, że MPI w zależności od wielkości rpzesyłanych danych stosuje różne strategie ich pakowania, kompresji i przesyłania, natomiast PCJ korzysta z jednakowego mechanizmu niezależnie od długości przesyłanych danych. Wyniki wskazują na konieczność optymalizacji biblioteki PCJ w zakresie przesyłania danych pomiędzy różnymi węzłami za pomocą sieci. Trzeba jednak pamiętać, że obsługa nowych interfejsów sieciowych, takich jak Infiniband, została wprowadzona do platformy Java niedawno.

Podobne testy wykonano z wykorzystaniem biblioteki ProActive. W przypadku tej biblioteki możliwe było wykonanie testu *ping-pong* na dwa sposoby: z wykorzystaniem jednego oraz dwóch aktywnych obiektów.

```
1  static public class ActiveObject {
2      private double[] data;
3
4      public BooleanWrapper prepareData(int size) {
5          data = new double[size];
6          for (int i = 0; i < size; i++) {
7              data[i] = (double) i + 1;
8          }
9          return new BooleanWrapper(true);
10     }
11
12     public double[] getData() {
13         return data;
14     }
15
16     public BooleanWrapper setData(double[] data) {
17         this.data = data;
18         return new BooleanWrapper(true);
19     }
20
21     public void setDataVoid(double[] data) {
22         this.data = data;
23     }
24 }
```

Listing 5.12: *Klasa reprezentująca aktywny obiekt w sposobie z jednym aktywnym obiektem*

Pierwszy sposób polegał na stworzeniu jednego aktywnego obiektu, do którego odwoływał się główny program. Na listingu 5.12 przedstawiona jest wykorzystana klasa reprezentująca aktywny obiekt. Aktywny obiekt odbierał dane przekazywane jako argument wywołanej metody (`BooleanWrapper setData(double[] data)` i `void setDataVoid(double[] data)`) oraz wysyłał dane jako wartość zwracaną metody bezargumentowej (`double[] getData()`). Przed pobieraniem danych, jednorazowo dla danej wielkości przesyłanych danych, tworzono tablicę odpowiednich rozmiarów zawierającą zmienne typu `double` i wypełniano ją kolejnymi liczbami naturalnymi (`BooleanWrapper prepareData(int size)`).

Dla tego sposobu wykorzystano kilka różnych metod: *get*, *put*, *putB* oraz *putVoid*. Metody te oznaczają:

get – jednostronne blokujące pobieranie danych od aktywnego obiektu (listing 5.13);

put – jednostronne blokujące wysyłanie danych do aktywnego obiektu (listing 5.14);

putB – naprzemienne blokujące pobieranie wartości z aktywnego obiektu i wysyłanie danych do aktywnego obiektu (listing 5.15);

putVoid – jednostronne nieblokujące, asynchroniczne wysyłanie danych do aktywnego obiektu (listing 5.16);

```
1    for (int i = 0; i < ntimes; i++) {
2        tmp = activeObject.getData();
3    }
```

Listing 5.13: *Synchroniczne pobieranie danych z aktywnego obiektu*

```
1    for (int i = 0; i < ntimes; i++) {
2        PAFuture.waitFor(activeObject.setData(data));
3    }
```

Listing 5.14: *Wysyłanie danych do aktywnego obiektu z oczekiwaniem na zakończenie operacji*

```
1    for (int i = 0; i < ntimes; i++) {
2        if (i % 2 == 0) {
3            PAFuture.waitFor(activeObject.setData(data));
4        } else {
5            data = activeObject.getData();
6        }
7    }
```

Listing 5.15: *Naprzemienne wysyłanie danych do aktywnego obiektu i odbieranie danych z aktywnego obiektu*

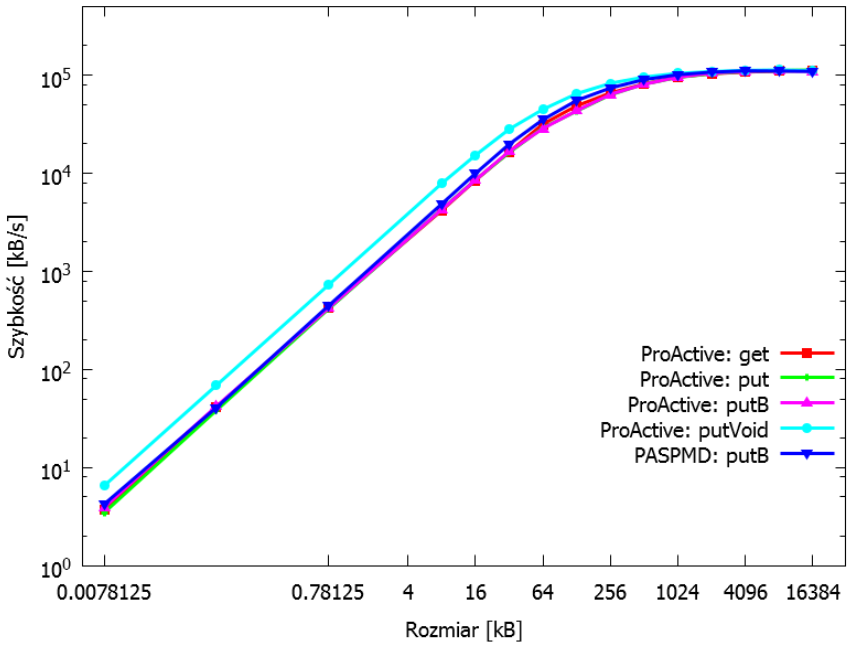
W drugim sposobie program główny tworzył dwa aktywne obiekty, które przekazywały argumenty między sobą, a jako zwracaną wartość przekazywały wyniki testu. Sposób ten wykorzystywał mechanizm SPMD.

Wykonano jedynie testy naprzemiennego wysyłania danych między aktywnymi obiektami oznaczane wcześniej jako metoda *putB*. W celu wyeliminowania sytuacji, gdy dwa obiekty równocześnie wysyłają do siebie dane, zastosowano dostępną od wersji Java 5 klasę reprezentującą semafor (`java.util.concurrent.Semaphore`). Każdy aktywny obiekt posiada dokładnie semafor, który początkowo jest opuszczony. Aktywny obiekt o identyfikatorze równym 0 wysyła dane do aktywnego obiektu z identyfikatorem równym 1. Ten po otrzymaniu wiadomości podnosi semafor i wysyła dane do aktywnego obiektu z identyfikatorem równym 0, który w tym przebiegu pętli oczekiwał na podniesienie semafora.

```
1     for (int i = 0; i < ntimes; i++) {
2         activeObject.setDataVoid(data);
3     }
```

Listing 5.16: *Asynchroniczne wysyłanie danych do aktywnego obiektu*

Najważniejsze fragmenty klasy reprezentującej aktywne obiekty przedstawiono na listingu 5.17. Dodanie adnotacji `@ImmediateService` powoduje, że metoda ta zostanie wykonywana nawet wówczas, gdy aktualnie aktywny obiekt jest w trakcie wykonywania innej operacji – w niniejszym przykładzie w momencie wykonywania ciała metody `StringWrapper run()` będącej główną metodą testów.



Rysunek 5.7: *Szybkość przesyłania danych z wykorzystaniem biblioteki ProActive (hydra – węzeł typu* westmere*)*

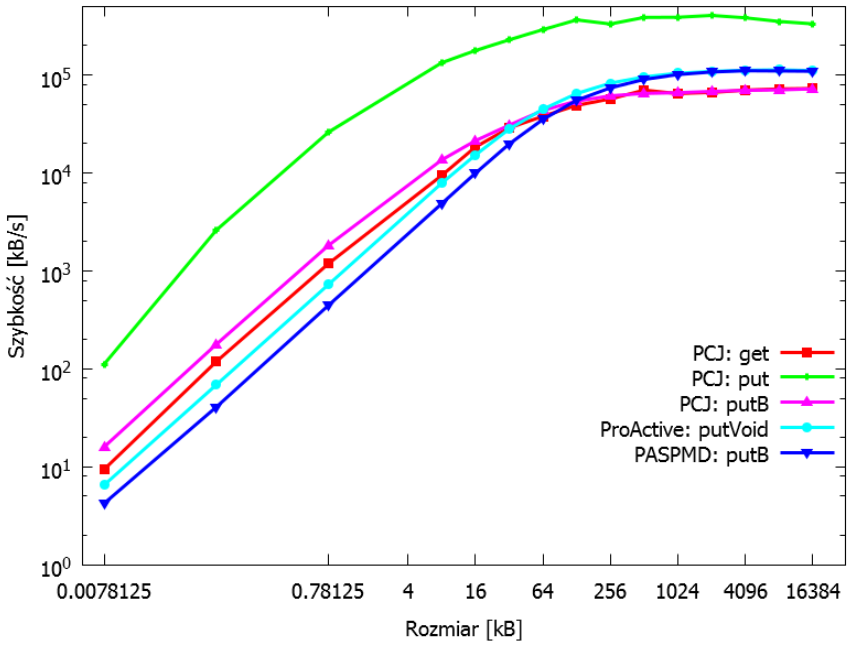
Porównanie różnych metod wykonania operacji *ping-pong* z wykorzystaniem biblioteki ProActive znajduje się na rysunku 5.7. Testy wykonano na klastrze *hydra* na węzłach *westmere*.

Z uzyskanych wyników wynika, że osiągana szybkość wykonywania operacji *ping-pong* jest zbliżona niezależnie od użytej metody. Jednocześnie widać, że w pełni asynchroniczny mechanizm, oznaczony na wykresie jako *ProActive: putVoid*, zachowuje się zgodnie z przewidywaniami i jest nieznacznie szybszy od pozostałych metod.

Na rysunku 5.8 znajduje się porównanie uzyskanej szybkości w teście *ping-pong* dla bibliotek PCJ i ProActive. Testy dla PCJ zostały ponownie wykonane z wykorzystaniem

```
1  static public class ActiveObject {
2      private final Semaphore sem = new Semaphore(0, true);
3      public double[] localData;
4
5      @ImmediateService
6      public void setData(double[] data) {
7          localData = data;
8          sem.release();
9      }
⋮                                  ...
17     public StringWrapper run() {
18         ActiveObject[] activeObjects =
19                 PAGroup.getGroup(PASPMD.getSPMDGroup())
20                     .toArray(new ActiveObject[0]);
⋮                                  ...
26         /*  ... prepare data ... */
27         double[] data = new double[n];
28         for (int i = 0; i < n; i++) {
29             data[i] = (double) i + 1;
30         }
⋮                                  ...
41         /* ... test body ... */
42         for (int i = 0; i < ntimes; i++) {
43             if (PASPMD.getMyRank() == i % 2) {
44                 activeObjects[(i + 1) % 2].setData(data);
45             } else {
46                 sem.acquire();
47             }
48         }
⋮                                  ...
58     }
59 }
```

Listing 5.17: *Klasa reprezentująca aktywne obiekty w ProActive z wykorzystaniem modelu SPMD*

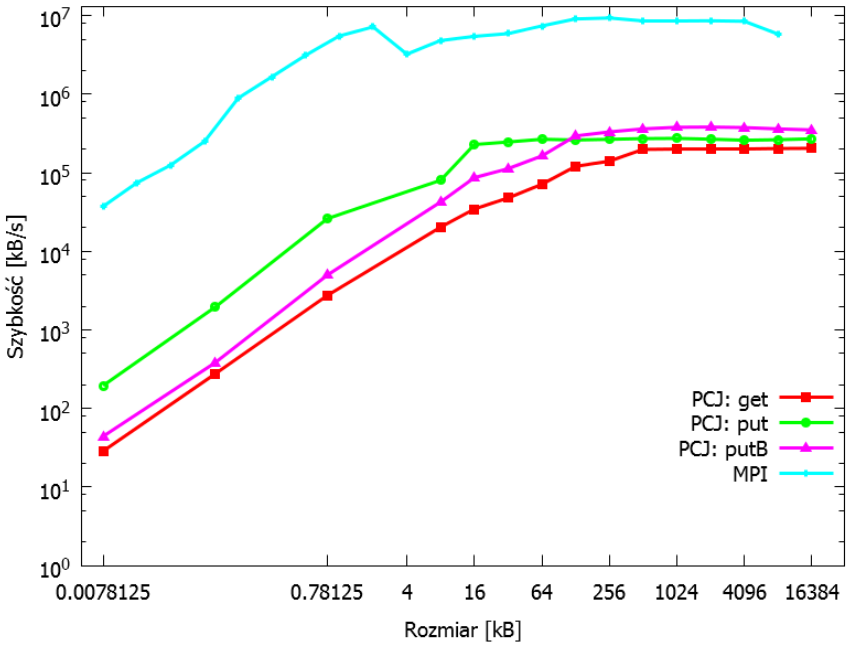


Rysunek 5.8: *Porównanie szybkości przesyłania danych z wykorzystaniem bibliotek PCJ i ProActive (hydra – węzeł typu* westmere*)*

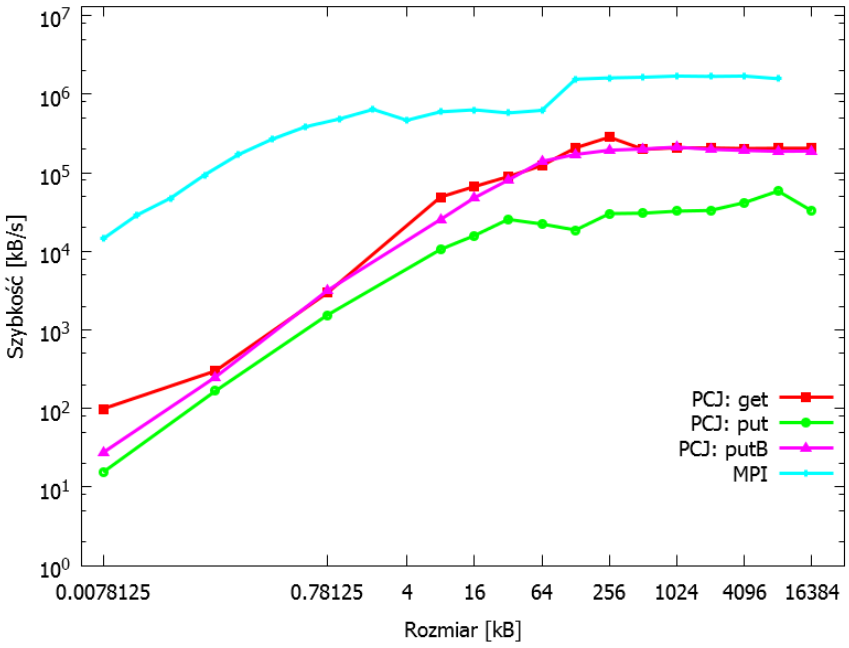
tej samej wersji maszyny wirtualnej Java, której użyto do uzyskania wyników dla ProActive, czyli dla wersji 1.8.0. Testy wykonano na klastrze *hydra* na węzłach *westmere*.

Na podstawie wykresów można stwierdzić, że biblioteka PCJ działa wyraźnie szybciej dla metody typu *put* i *putB* od biblioteki ProActive na danych mniejszych niż 256 kB, a dla danych większych oraz przy wykorzystaniu metody *get* biblioteka PCJ ma osiągi zbliżone do biblioteki ProActive. Ponadto można zauważyć, że przy większych danych osiąga się maksymalną szybkość sięgającą ponad 300 MB/s dla PCJ oraz około 110 MB/s dla ProActive.

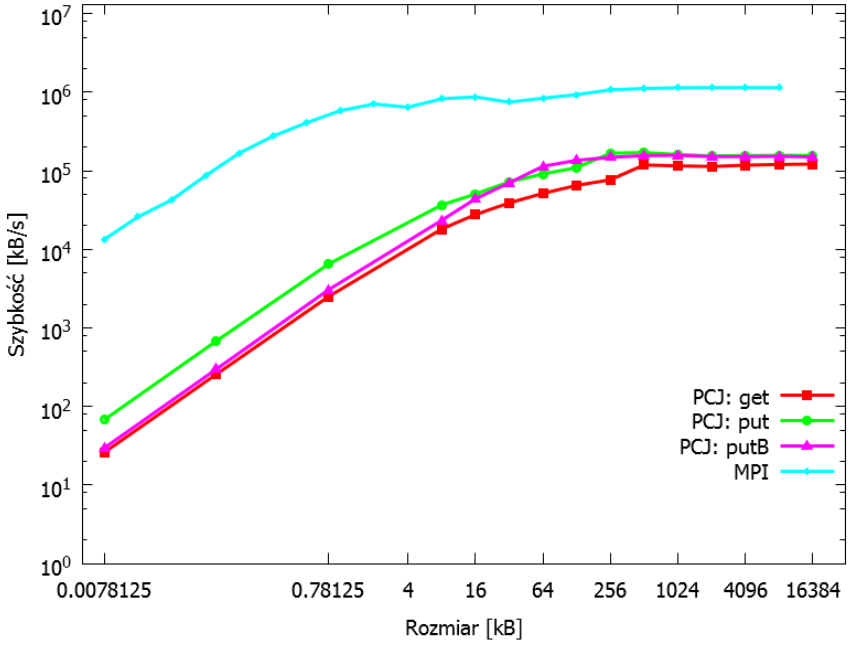
Jeden węzeł z dwoma wątkami



(a) *hydra – węzeł typu* westmere



(b) *hydra – węzeł typu* istanbul



(c) *halo2*

Rysunek 5.9: *Szybkość przesyłania danych między dwoma wątkami na jednym węźle w zależności od rozmiaru wiadomości i wykorzystanego mechanizmu*

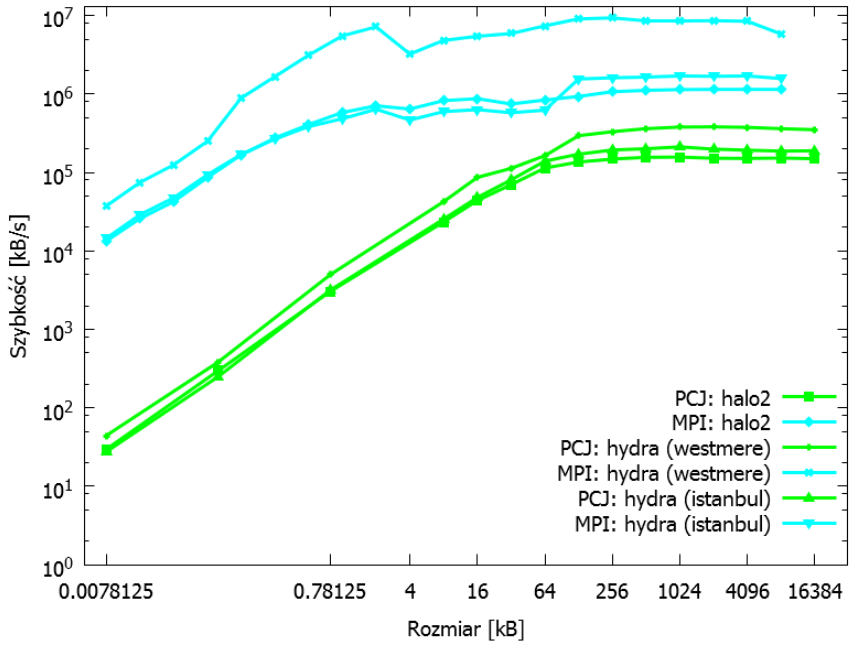
Podobne wykresy do tych przedstawionych na rysunku 5.5, ale na podstawie danych zebranych z wykorzystaniem tylko jednego węzła, na którym uruchomiono dwa wątki, znajdują się na rysunku 5.9. W tym przypadku dane nie były przesyłane z wykorzystaniem gniazd, tylko przetwarzane wewnątrz jednej wirtualnej maszyny Java. W związku z tym najszybszą metodą przesyłania danych, na różnych architekturach maszyn, okazała się metoda pośrednia *putB*.

Wykresy przedstawione na rysunku 5.10 porównują szybkość wykonywania operacji przekazywania danych między dwoma wątkami znajdującymi się na jednym węźle pomiędzy PCJ i MPI. W przypadku PCJ wybrano mechanizm *putB*. Uzyskane wyniki wskazują, że biblioteka PCJ jest konkurencyjna względem MPI.

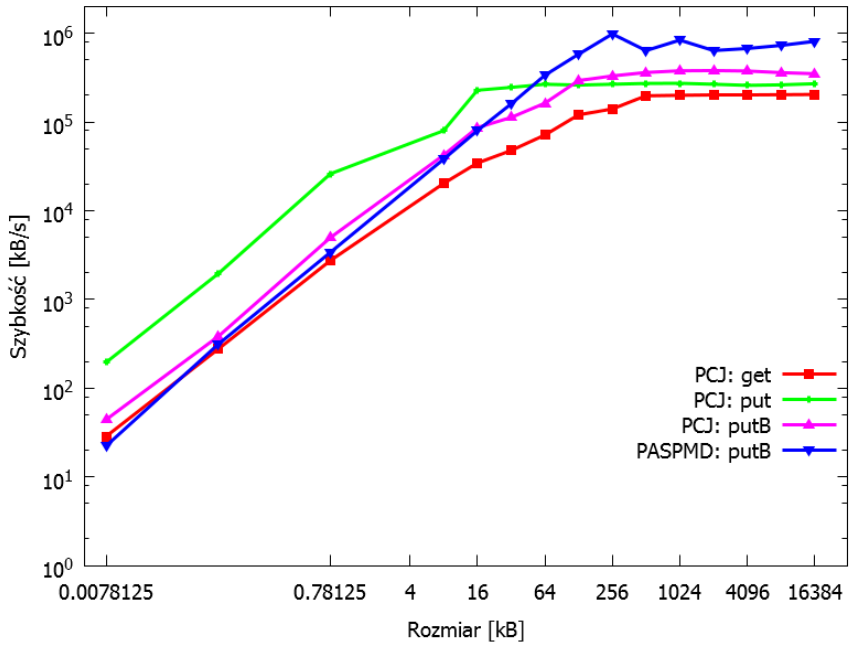
W przypadku przekazywania danych pomiędzy dwoma wątkami wewnątrz jednego węzła, maksymalna prędkość jest osiągalna już dla danych o rozmiarze 256 KB. Dla PCJ prędkość ta oscyluje wokół wartości 150-200 MB/s, natomiast dla MPI maksymalna zaobserwowana prędkość dochodziła do 1 GB/s.

Ze względu na sposób uruchamiania aktywnych obiektów w bibliotece ProActive nie było możliwe wykonanie testu *ping-pong* w ramach jednej wirtualnej maszyny Java z jednym aktywnym obiektem przy wykorzystaniu metod *get* i *put*. Wykonano natomiast testy wykorzystujące `PASPMD`, w których na jednej wirtualnej maszynie Java uruchomiono dwa aktywne obiekty i wykorzystano metodę *putB*.

Porównanie szybkości przesyłania danych z wykorzystaniem bibliotek PCJ i ProActive między dwoma wątkami w ramach jednej wirtualnej maszyny Java są



Rysunek 5.10: *Szybkość przesyłania danych między dwoma wątkami na jednym węźle w zależności od rozmiaru wiadomości*



Rysunek 5.11: *Porównanie szybkości przesyłania danych z wykorzystaniem bibliotek PCJ i ProActive między dwoma wątkami w ramach jednej wirtualnej maszyny Java (hydra – węzeł typu* westmere*)*

widoczne na rysunku 5.11. Wyniki zostały zebrane na klastrze *hydra* na węźle typu *westmere*. Widać, że dla małych rozmiarów przesyłanych danych w jednym komunikacie, czyli poniżej 64 kB, szybkość przesyłania danych z wykorzystaniem ProActive jest taka sama jak z wykorzystaniem PCJ, zwłaszcza dla mechanizmu *putB*. Jedynie przy wykorzystaniu mechanizmu *put* w PCJ, prędkość przesyłania danych jest widocznie wyższa. Jednak po przekroczeniu granicy 64 kB, szybkość transferu danych osiągana dla ProActive jest wyższa niż dla PCJ.

5.2.3 Broadcast

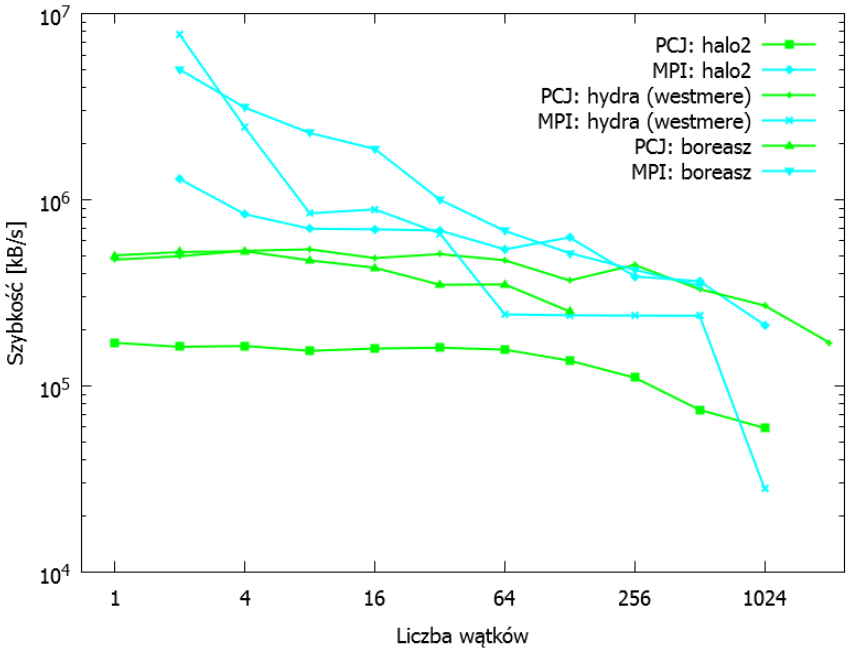
Test typu *broadcast*, czyli rozgłoszenie, polega na wysłaniu danych z jednego wątku do wszystkich pozostałych wątków. W niniejszej pracy wysyłana była różnej długości tablica zmiennych typu `double`. Wyniki dla rozgłaszania zależne są od liczby użytych fizycznych węzłów, wątków PCJ i wielkości tablicy.

Rozgłaszanie nie różni się zasadniczo od jednostronnego nieblokującego wysyłania danych do przeciwnego wątku obliczeń z wykorzystaniem metody `PCJ.put()`, z tym, że dane zamiast być wysłane tylko do jednego wątku, są wysyłane do wszystkich wątków. Ponadto wysyłanie następuje z wykorzystaniem struktury drzewa binarnego. Z tego powodu przykład na listingu 5.18 przedstawia jedynie zmodyfikowaną część *test body*, gdyż pozostała część jest taka sama jak w przypadku wersji z *ping-pong*. Na przykładzie widać, że wątek o numerze 0 rozsyła swoją wartość zmiennej `b` zlecając, by wartość ta była przechowana w zmiennej współdzielonej o nazwie `a`. Następnie każdy wątek, w tym także wątek 0, czeka na otrzymanie nowej wartości zmiennej współdzielonej `a`. Dopiero po tym następuje przejście do kolejnej iteracji pętli.

```
1     /* ... test body ... */
2     for (int i = 0; i < ntimes; i++) {
3         if (PCJ.myId() == 0) {
4             PCJ.broadcast("a", b);
5         }
6         PCJ.waitFor("a");
7     }
```

Listing 5.18: *Kod nieblokującego rozgłaszania danych do wszystkich wątków obliczeń*

Broadcast, czyli rozgłaszanie, jest ostatnim wykonanym małym testem wydajnościowym. Rysunek 5.12 zawiera porównanie szybkości rozgłaszania 1 MB danych (tablicy zawierającej 131072 elementy typu `double`) w zależności od wykorzystanego systemu klastrowego. Rysunek ten zawiera także wyniki uzyskane dla MPI na tych samych systemach klastrowych. Główny kod użyty w rozwiązaniu korzystającym z MPI



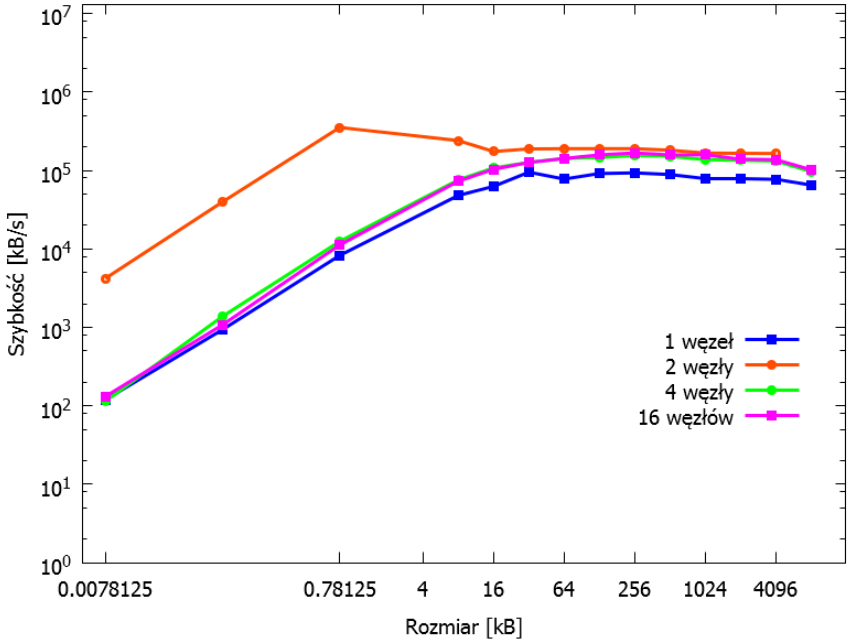
Rysunek 5.12: *Szybkość rozgłaszania 1 MB danych w zależności od liczby użytych wątków i systemu klastrowego*

znajduje się na listingu 5.19. W MPI każdy proces biorący udział w obliczeniach zobligowany jest do wykonania operacji `MPI_Bcast`, w celu otrzymania danych od procesu wysyłającego. Jest to *implicite* punkt synchronizacji w programie. W przypadku PCJ inicjalizacja rozgłaszania następuje z inicjatywy tylko jednego wątku, a samo rozgłaszanie jest asynchroniczne.

```
1     for (j = 0; j < ntimes; j++) {
2         MPI_Bcast(buf, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
3     }
```

Listing 5.19: *Test broadcast w MPI*

Na rysunku 5.12 widać wyraźnie, że rozgłaszanie w MPI jest szczególnie zoptymalizowane w przypadku wykonywania tej operacji w ramach jednego węzła. Szybkość rozgłaszania w MPI maleje liniowo ze wzrostem liczby procesorów. W przypadku PCJ rozgłaszanie jest wolniejsze, ale do kilkudziesięciu węzłów praktycznie niezależne od ich ilości. Dla większej liczby wątków wydajność spada, jednak znacząco wolniej niż w przypadku MPI. Skalowalność rozgłaszania zaimplementowanego w PCJ jest więc lepsza niż w MPI, pomimo wyższej wydajności tego drugiego. Dla liczby wątków większej niż 128, wyniki uzyskiwane przez PCJ są konkurencyjne w stosunku do MPI.

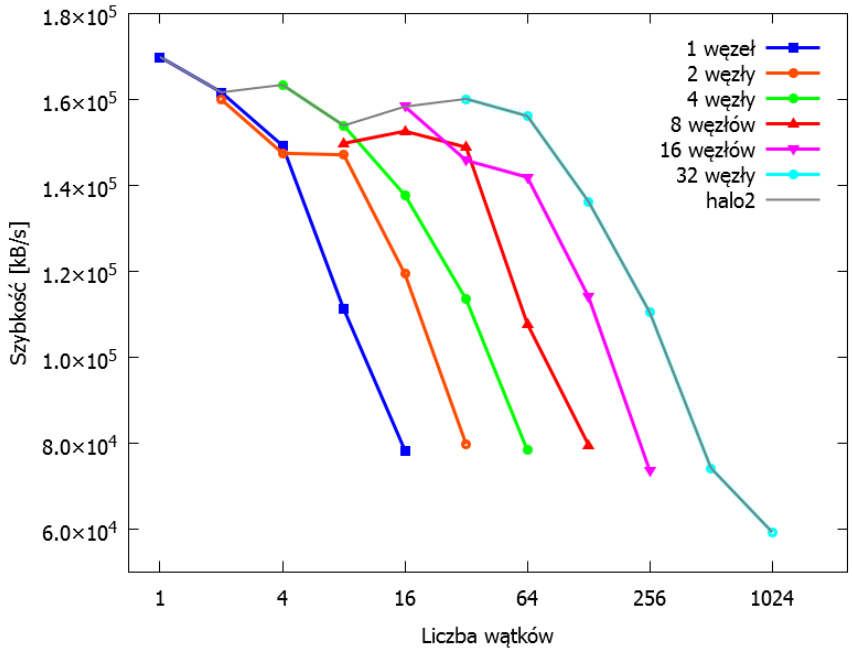


Rysunek 5.13: *Szybkość rozgłaszania w ramach 16 wątków w zależności od rozmiaru przesyłanej tablicy i liczby węzłów (halo2)*

Rysunek 5.13 przedstawia szybkość pojedynczej operacji rozgłaszania w przypadku wykorzystania łącznie 16 wątków PCJ. Poszczególne serie danych oznaczają uruchomienie:

- 16 wątków na 1 węźle,
- po 8 wątków na 2 węzłach,
- po 4 wątki na 4 węzłach,
- po 1 wątku na 16 węzłach.

Na wykresie widać, że osiągane szybkości są zbliżone, a najwyższą wydajność uzyskuje się dla 16 węzłów po 1 wątku na węzeł. Dzieje się tak dlatego, że dane są przesyłane po drzewie binarnym i wszystkie operacje mogą zachodzić równolegle. W przypadku odwrotnym, gdzie na 1 węźle znajduje się 16 wątków, sam mechanizm wysyłania komunikatu jest prawie natychmiastowy, natomiast węzeł traci znaczną część czasu na przekazanie danych do odpowiednich deserializatorów, gdzie następuje przetworzenie komunikatu w postać obiektu z wykorzystaniem odpowiedniej dla poszczególnych wątków ładowarek klas. Przekazanie danych do deserializatorów wykonywane jest liniowo. Stąd najgorszy wynik rozgłaszania występuje w ramach dokładnie jednego węzła.



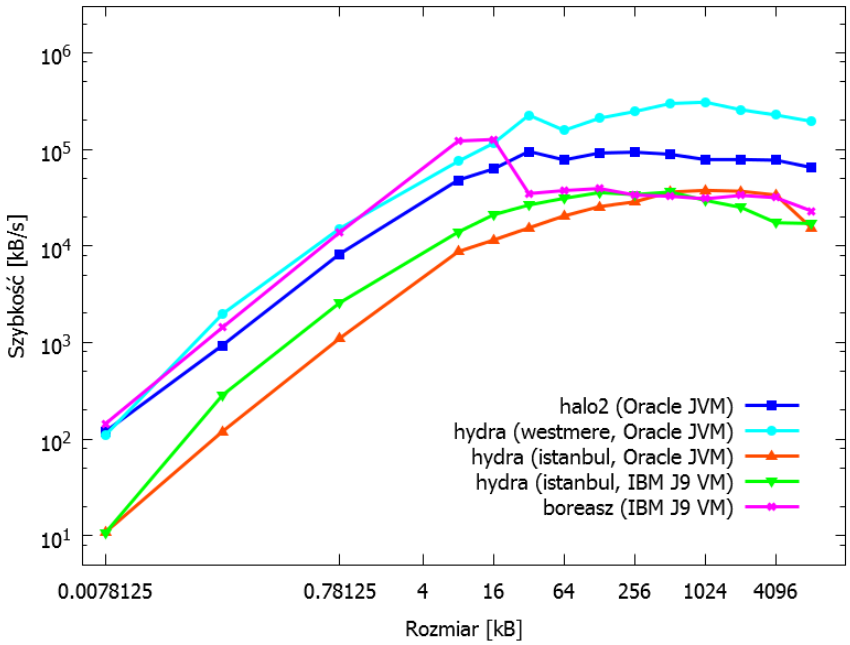
Rysunek 5.14: *Szybkość rozgłaszania 1 MB danych w zależności od liczby węzłów i wątków (halo2)*

Podobne wnioski można wysnuć analizując rysunek 5.14. Przedstawia on szybkość wykonywania operacji rozgłaszania 1 MB danych w zależności od liczby użytych wątków. Różne linie na wykresie oznaczają różną użytą liczbę węzłów. Przykładowo patrząc na 16 wątków na osi poziomej, widać, że najwolniej wykonuje się rozgłaszanie wśród 16 wątków dla 1 węzła, następnie dla 2, 4, 8 i 16 węzłów. Wyniki są analogiczne dla większej liczby wykorzystanych wątków.

Porównanie szybkości rozgłaszania w zależności od rozmiaru wiadomości i systemu klastrowego w przypadku wykorzystania 1 węzła z 16 wątkami przedstawiono na rysunku 5.15. Dodatkowo w tym przypadku wykonano test z wykorzystaniem implementacji wirtualnej maszyny Java firmy IBM: *IBM J9 VM* (środowisko: *IBM SDK, Java Technology Edition, Version 7, Service Refresh 6*). Widać, że wykorzystanie różnych wirtualnych maszyn wiąże się z różnicami w wydajności. W tym przypadku wirtualna maszyna Java stworzona przez IBM wydaje się być szybsza od wirtualnej maszyny Java firmy Oracle. Widoczny gwałtowny spadek wydajności w systemie boreasz, dla rozgłaszanych danych o rozmiarze powyżej 16 kB, można tłumaczyć innym sposobem alokacji pamięci w zależności od rozmiaru danych.

Rozgłaszanie z czekaniem

Oprócz testów samego rozgłaszania, wykonano testy rozgłaszania z czekaniem, czyli rozgłaszania z barierą. W tym przypadku po każdej operacji rozgłaszania, wszystkie



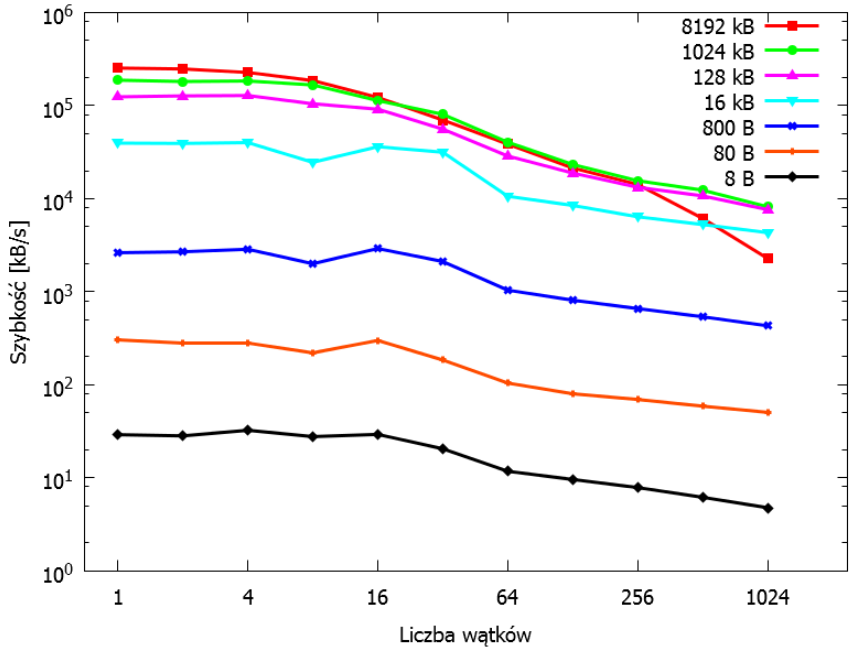
Rysunek 5.15: *Szybkość rozgłaszania w zależności od rozmiaru przesyłanej tablicy i systemu klastrowego*

wątki oczekują na otrzymanie zmian danych, po czym wykonują operację bariery. Listing 5.20 przedstawia kod tej operacji. Policzona w testach szybkość rozgłaszania uwzględnia również czas wykonania operacji bariery.

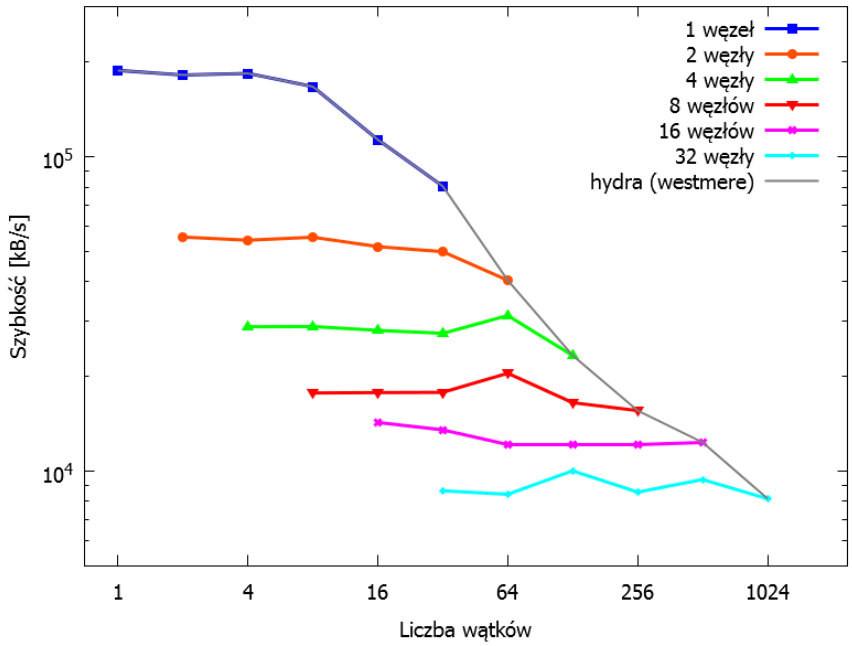
```
1      /* ... test body ... */
2      for (int i = 0; i < ntimes; i++) {
3          if (PCJ.myId() == 0) {
4              PCJ.broadcast("a", b);
5          }
6          PCJ.waitFor("a");
7          PCJ.barrier();
8      }
```

Listing 5.20: *Kod rozgłaszania z czekaniem*

Na rysunku 5.16 przedstawiono szybkość rozgłaszania z uwzględnieniem operacji bariery w zależności od rozmiaru przesyłanej tablicy. Widać zależność między wielkością wiadomości a szybkością rozgłaszania. Widać również, że maksymalna uzyskiwana szybkość to około 200 MB/s dla małej liczby wątków, a przy 1024 wątkach to około 15 MB/s. Ponadto maksymalna prędkość dla danej wielkości wiadomości jest stała dla pierwszych kilkunastu wątków, wśród których wiadomość jest rozsyłana. Jest to spowodowane szybkością rozsyłania i bariery w ramach jednego węzła.



Rysunek 5.16: *Szybkość rozgłaszania wraz z operacją bariery w zależności od rozmiaru przesyłanej tablicy (hydra – węzeł typu* westmere*)*



Rysunek 5.17: *Szybkość rozgłaszania 1 MB danych wraz z operacją bariery w zależności od liczby węzłów i wątków (hydra – węzeł typu* westmere*)*

Rysunek 5.17 przedstawia szybkość operacji rozgłaszania uwzględniającą czas wykonania operacji bariery przy wysyłaniu 1 MB danych. Dla 1 węzła, szybkość

rozgłaszania i bariery jest największa. Wiąże się to z tym, że nie ma tu do czynienia z przesyłaniem komunikatów w sieci, a operacja bariery wykorzystuje jedynie wbudowane mechanizmy synchronizacji. Dodatkowo po przekroczeniu bariery 12 wątków, czyli łącznej liczby rdzeni procesorów na węźle, szybkość drastycznie maleje. Jest to efekt współzawodnictwa o procesor w celu deserializacji otrzymanych danych. Co ciekawe, efekt ten nie jest już tak dobrze widoczny dla większej liczby wątków, prawdopodobnie ze względu na straty spowodowane przez komunikację międzywęzłową.

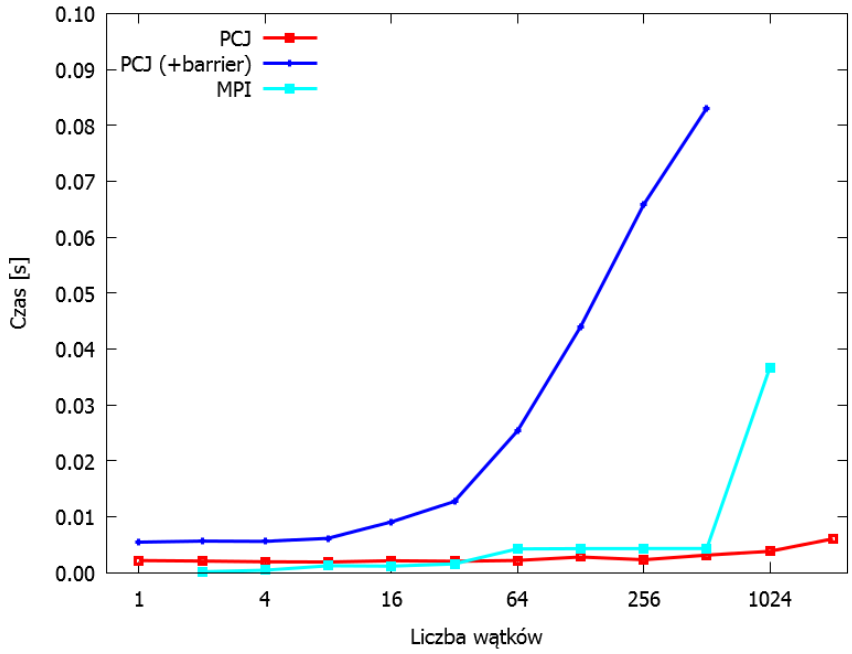
Wykresy na rysunku 5.18a porównują czas potrzebny na rozgłoszenie 1 MB danych w przypadku samego rozgłoszenia i rozgłoszenia wraz z barierą w zależności od liczby wątków. Ponadto dla porównania wykres przedstawia również czas wykonania operacji rozgłoszenia z wykorzystaniem rozwiązania stworzonego w MPI. Rozgłoszenie bez bariery jest rozgłoszeniem nieblokującym i asynchronicznym. Dzięki temu czas wykonania jest bardzo krótki, niezależnie od liczby wątków. W przypadku rozgłaszania z barierą, czas potrzebny na wykonanie pojedynczej operacji rośnie z pierwiastkiem względem liczby wątków.

Wykresy na rysunku 5.18b przedstawiają dokładnie te same dane, co rysunek 5.18a, z tą różnicą, że zamiast czasu, na osi pionowej są wartości uzyskanych szybkości. Dla danych zebranych z użyciem MPI widać wyraźne dwa schodki. Pierwszy, pomiędzy 32 a 64 wątkami, oznacza wyjście poza na jeden węzeł w trakcie rozgłaszania. Drugie załamanie, pomiędzy 512 a 1024 wątkami, oznacza zwiększenie liczby węzłów z 16 do 32, kiedy to oprócz wykonywania operacji rozgłaszania między węzłami, które fizycznie są blisko siebie, następuje przekazanie danych między dwoma oddzielnymi fragmentami klastra.

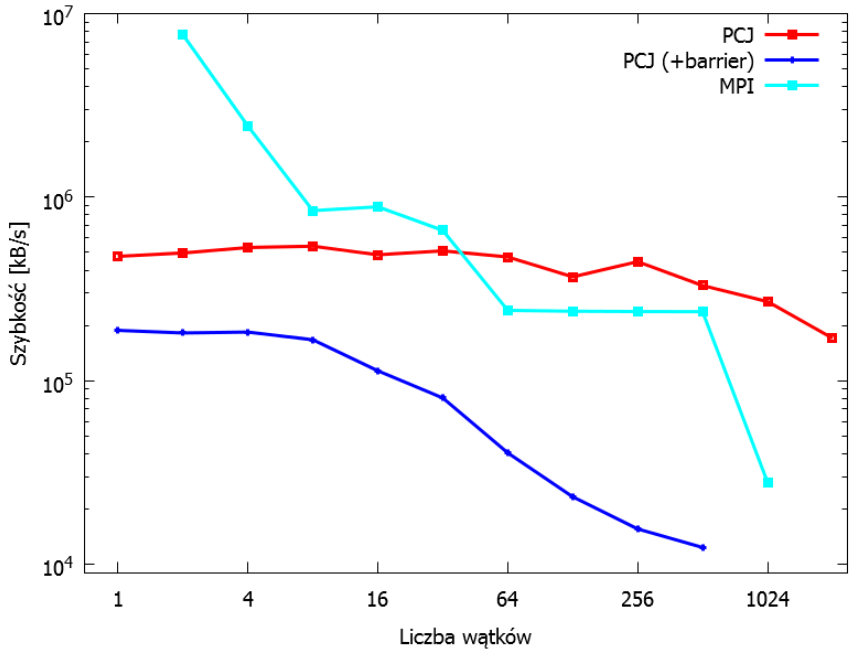
Podobnie jak w przypadku testu wydajnościowego *ping-pong* wykonano również testy wydajnościowe rozgłaszania biblioteki ProActive. Rozgłaszanie występowało wraz z operacją czekania, czyli *de facto* z operacją bariery podobnie jak miało to miejsce w rozwiązaniu opartym o PCJ.

Listing 5.21 przedstawia fragment programu służącego do przeprowadzenia testu rozgłaszania w bibliotece ProActive. W tym fragmencie tworzona jest grupa aktywnych obiektów i przygotowywane są dane do wysyłki, a następnie pobierany jest aktualny znacznika czasu dużej precyzji, który wykorzystywany jest do obliczenia czasu wykonywania głównej pętli testu. W głównej pętli testu wykonywana jest metoda `setData(double[] data)` na grupie aktywnych obiektów, czyli następuje rozgłaszanie przygotowanych wcześniej danych. Metoda `setData(double[] data)` wchodzi w skład klasy `ActiveObject`. Klasa ta wygląda analogiczne do klasy przedstawionej na listingu 5.12 wykorzystywanej w teście *ping-pong* z tą różnicą, że nie zawiera niewykorzystywanych metod.

Rysunek 5.19 przedstawia porównanie czasu wykonywania pojedynczej operacji rozgłaszania wraz z operacją bariery w przypadku korzystania z biblioteki PCJ i biblioteki



(a) *porównanie czasu*



(b) *porównanie szybkości*

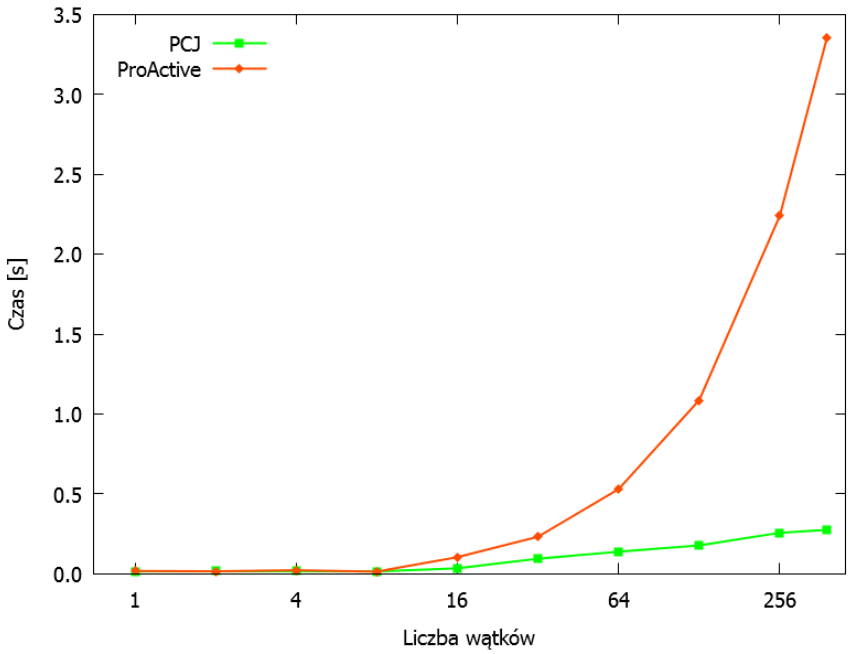
Rysunek 5.18: *Porównanie czasu i uzyskanej szybkości rozgłaszania 1 MB danych w zależności od liczby wątków (hydra – węzeł typu* westmere*)*

ProActive w zależności od liczby użytych wątków. Testy wykonano na klastrze *hydra* na węzłach typu *istanbul* posiadających 12 rdzeni obliczeniowych.

Na rysunku widać, że dla ilości wątków mieszczących się na jednym węźle, czyli do 8 wątków, PCJ i ProActive zachowują się w podobny sposób – w przybliżeniu operacja

```
1     ActiveObject activeObjectsGroup
2             = (ActiveObject) PAGroup.newGroup(
3                     ActiveObject.class.getName(),
4                     new Object[]{}, nodes);
⋮                                 ...
12    /*  ... prepare data ... */
13    double[] data = new double[n];
14    for (int i = 0; i < n; i++) {
15        data[i] = (double) i + 1;
16    }
⋮                                 ...
25    /* ... test body ... */
26    long time = System.nanoTime();
27    for (int i = 0; i < ntimes; i++) {
28        PAGroup.waitAll(activeObjectsGroup.setData(data));
29    }
30    time = System.nanoTime() - time;
```

Listing 5.21: *Klasa wykonująca test rozgłaszania z czekaniem*



Rysunek 5.19: *Czas rozgłaszania 1 MB danych wraz z operacją bariery w zależności od liczby wątków (hydra – węzeł typu* istanbul*)*

rozgłaszania trwa tyle samo czasu. Wraz ze wzrostem liczby wątków, czyli wyjściu komunikacji z jednego węzła, czas rozgłaszania zwiększa się. W przypadku PCJ czas ten

zwiększa się w sposób logarytmiczny w zależności od liczby wątków, co jest spowodowane zastosowaniem drzewiastej struktury wątków w trakcie rozgłaszania. W przypadku ProActive czas rozgłaszania rośnie dużo szybciej. Dla 384 wątków uruchomionych na 32 węzłach czas rozgłaszania w przypadku ProActive (3,35 s) jest dwunastokrotnie większy niż czas rozgłaszania z wykorzystaniem biblioteki PCJ (0,27 s).

5.2.4 Redukcja

Redukcja jest w pewnym sensie przeciwieństwem rozgłaszania. Celem redukcji jest zebranie danych ze wszystkich wątków i wykonanie na nich pewnej operacji redukującej. Przykładowymi operacjami redukującymi mogą być: wartość maksymalna lub minimalna, suma czy iloczyn. Wszystkie one redukują zbiór danych do postaci pojedynczej wartości. W przypadku rozgłaszania, pojedyncza wartość jest rozsyłana do wszystkich wątków.

W aktualnej wersji biblioteki PCJ nie istnieje wbudowany mechanizm wykonywania operacji redukcji. Programista sam musi napisać odpowiedni fragment kodu. Dzięki temu możliwe jest dokładne zapanowanie nad redukcją i przeplatanie komunikacji z obliczeniami w celu osiągnięcia najwyższej wydajności.

W niniejszej pracy przedstawione zostanie kilka sposobów realizacji redukcji. W każdym przypadku wykonywaną operacją będzie suma. Inne operacje mają implementację analogiczną.

Każdy z wątków biorących udział w redukcji posiada swoją kopię zmiennej współdzielonej do zredukowania. W przykładach ta zmienna będzie typu `double` i nazywać się będzie `a`. Po zakończonej redukcji wątek o numerze 0 będzie posiadać zredukowaną wartość, czyli sumę wartości zmiennej współdzielonej `a` ze wszystkich wątków.

Redukcja liniowa

Pierwszą zaprezentowaną implementacją jest redukcja liniowa. W jej przypadku wątek 0 pobiera dane z każdego wątku po kolei i sumuje poszczególne wartości. Listing 5.22 przedstawia fragment kodu odpowiedzialny za redukcję liniową.

Początkowo wszystkie wątki mają wartość sumy równą *NaN* (*nie-liczba* – ang. *Not-a-Number*). Wątek 0 przypisuje do tej zmiennej wartość zmiennej współdzielonej `a` przechowywaną lokalnie. Następnie pobiera od wszystkich pozostałych wątków ich wartość zmiennej `a` i sumuje te wartości. W przykładzie wykorzystana jest metoda `PCJ.get()`, która jest blokująca – program nie zacznie wykonywać kolejnej instrukcji, dopóki odpowiedź nie nadejdzie.

```
1      double sum = Double.NaN;
2      if (PCJ.myId() == 0) {
3          sum = PCJ.getLocal("a");
4          for (int p = 1; p < PCJ.threadCount(); p++) {
5              sum = sum + (double) PCJ.get(p, "a");
6          }
7      }
```

Listing 5.22: *Redukcja liniowa*

Redukcja liniowa z obiektem `Future`

Pewnym udoskonaleniem redukcji liniowej jest redukcja z obiektem `Future` przedstawiona na listingu 5.23.

```
1      double sum = Double.NaN;
2      if (PCJ.myId() == 0) {
3          FutureObject[] aF;
4          aF = new FutureObject[PCJ.threadCount()];
5          for (int p = 1; p < PCJ.threadCount(); p++) {
6              aF[p] = PCJ.getFutureObject(p, "a");
7          }
8
9          sum = PCJ.getLocal("a");
10         for (int p = 1; p < PCJ.threadCount(); p++) {
11             sum = sum + (double) aF[p].get();
12         }
13     }
```

Listing 5.23: *Redukcja liniowa z obiektem `Future`*

W tym przypadku wątek 0 korzysta z nieblokującej metody `PCJ.getFutureObject()`. Dzięki zastosowaniu tej metody, odpowiedzi mogą spływać do wątku 0 w trakcie wysyłania kolejnych żądań i przetwarzania odpowiedzi. Blokowanie może, ale nie musi, nastąpić dopiero w momencie, gdy odpowiedź jest potrzebna do obliczenia sumy. Blokowanie nie następuje, gdy odpowiedź zdążyła spłynąć przed wywołaniem metody `Future.get()`.

Możliwe jest także zebranie wyników w lekko zmodyfikowany sposób przedstawiony na listingu 5.24. Sposób ten niepotrzebnie zabiera czas procesora, gdyż pętle sprawdzające, czy są dostępne wyniki, działają bez przerwy, w kółko. Z drugiej strony, dzięki zastosowaniu tej metody, możliwe jest szybsze przetwarzanie danych – przetwarzane są odpowiedzi, które zdążyły już napłynąć, niezależnie od miejsca ich występowania w tablicy obiektów `FutureObject`, jak to miało miejsce w listingu 5.23.

```
1     double sum = Double.NaN;
2     if (PCJ.myId() == 0) {
3         Set<FutureObject> futureSet = new HashSet<>();
4         for (int p = 1; p < PCJ.threadCount(); p++) {
5             futureSet.add(PCJ.getFutureObject(p, "a"));
6         }
7
8         sum = PCJ.getLocal("a");
9         while(futureSet.isEmpty() == false) {
10            Iterator<FutureObject> it = futureSet.iterator();
11            while(it.hasNext()) {
12                FutureObject future = it.next();
13                if (future.isDone()) {
14                    sum = sum + (double) future.get();
15                    it.remove();
16                }
17            }
18        }
19    }
```

Listing 5.24: *Redukcja liniowa z obiektem* `Future` *w pętli*

Redukcja liniowa z wydelegowanym sumowaniem

Redukcja liniowa z wydelegowanym sumowaniem opiera się na przekazywaniu wartości sumy pomiędzy wątkami p i $p+1$. W tym przypadku oprócz zmiennej współdzielonej `a` potrzebna jest jeszcze jedna zmienna przechowująca aktualną wartość sumy dla wątku – zmienna współdzielona `sum`.

Listing 5.25 przedstawia sposób implementacji redukcji liniowej z wydelegowanym sumowaniem z wykorzystaniem pobierania danych z wątków. Załóżmy, że w obliczeniach bierze udział N wątków (N=`PCJ.threadCount()`) o identyfikatorach od 0 do $N-1$. Początkowo, wartość sumy w ostatnim wątku obliczeń o numerze $N-1$ jest równa wartości zmiennej `a`. Następnie wątek $N-2$ dodaje lokalną wartość zmiennej `a` do wartości zmiennej `sum` na wątku $N-1$. Obliczenia są synchronizowane za pomocą operacji `PCJ.barrier()` między dwoma wątkami. W kolejnych krokach wątek p synchronizuje się z wątkiem $p+1$, gdy ten ma już policzoną wartość sumy, i oblicza jej nową wartość. Obliczenia wykonywane są tak długo, aż wątek 0 wykona operację bariery z wątkiem 1 i zsumuje swoją wartość zmiennej `a` z wartością `sum` z wątku 1. Po zakończeniu całej procedury, wątek 0 zawiera w zmiennej `sum` wartość sumy zmiennej współdzielonej `a` ze wszystkich wątków.

Kod redukcji liniowej z wydelegowanym sumowaniem można również zapisać za pomocą mechanizmu wysyłania danych. Przykładowa implementacja przedstawiona jest

```
1     if (PCJ.myId() == PCJ.threadCount() - 1) {
2         PCJ.putLocal("sum", PCJ.getLocal("a"));
3     } else {
4         PCJ.barrier(PCJ.myId() + 1);
5         PCJ.putLocal("sum",
6                 (double) PCJ.getLocal("a")
7                 + (double) PCJ.get(PCJ.myId() + 1, "sum"));
8     }
9
10    if (PCJ.myId() > 0) {
11        PCJ.barrier((PCJ.myId() - 1) % PCJ.threadCount());
12    }
```

Listing 5.25: *Redukcja liniowa z wydelegowanym sumowaniem (wersja z pobieraniem danych)*

na listingu 5.26. W tym przypadku zamiast pobierania danych przez wątek p od wątku $p+1$, to wątek $p+1$ wysyła policzoną przez siebie sumę do wątku p. Gdy wątek p otrzyma wartość, wysyła sumę do wątku $p-1$. Na końcu wątek 0, po otrzymaniu wartości sumy od wątku 1, ustawia wartość `sum` jako sumę wartości zmiennej `a` na pozostałych wątkach (aktualna wartość zmiennej `sum`) i własnej wartości zmiennej `a`.

```
1     if (PCJ.myId() == PCJ.threadCount() - 1) {
2         PCJ.put(PCJ.myId() - 1, "sum", PCJ.getLocal("a"));
3     } else if (PCJ.myId() > 0) {
4         PCJ.waitFor("sum");
5         PCJ.put(PCJ.myId() - 1, "sum",
6                 (double) PCJ.getLocal("sum")
7                 + (double) PCJ.getLocal("a"));
8     } else {
9         PCJ.waitFor("sum");
10        PCJ.putLocal("sum",
11                (double) PCJ.getLocal("sum")
12                + (double) PCJ.getLocal("a"));
13    }
```

Listing 5.26: *Redukcja liniowa z wydelegowanym sumowaniem (wersja z wysyłaniem danych)*

Redukcja z tablicą

Podobnym sposobem redukcji jak redukcja liniowa, jest redukcja z tablicą. W tym przypadku wątek 0 posiada zmienną współdzieloną o nazwie `tab` typu tablicowego

rozmiaru N, gdzie N oznacza liczbę wątków w obliczeniach. Każdy wątek wysyła do wątku 0, do odpowiedniej komórki tablicy, swoją wartość zmiennej `a`. Po otrzymaniu wszystkich wartości, wątek 0 iteruje po elementach tej tablicy i je sumuje. Przykładowa implementacja przedstawiona jest na listingu 5.27.

```
1     PCJ.put(0, "tab", PCJ.getLocal("a"), PCJ.myId());
2
3     if (PCJ.myId() == 0) {
4         PCJ.waitFor("tab", PCJ.threadCount());
5     }
6
7     double sum = Double.NaN;
8     if (PCJ.myId() == 0) {
9         sum = 0.0;
10        for (double d : (double[]) PCJ.getLocal("tab")) {
11            sum = sum + d;
12        }
13    }
```

Listing 5.27: *Redukcja z tablicą*

Redukcja po drzewie binarnym z tablicą

Redukcja po kompletnym drzewie binarnym jest udoskonaleniem wcześniej przedstawionych implementacji redukcji: redukcji liniowej z wydelegowanym sumowaniem (wersji z wysyłaniem danych) i redukcji z tablicą. Implementacja redukcji z tablicą ma pewną wadę – wymaga stworzenia w jednym wątku tablicy posiadającej liczbę elementów zależną od liczby wątków, natomiast w redukcji liniowej z wydelegowanym sumowaniem przetwarzany jest pojedynczy element przez pojedynczy wątek i dopiero wówczas kolejny wątek może działać.

Redukcja po drzewie binarnym z tablicą usuwa te wady. Zakłada ona, że zmienna współdzielona `tab` jest tablicą składającą się z dwóch elementów i jest zadeklarowana na wszystkich wątkach. Ponadto komunikacja i przetwarzanie danych odbywają się z wykorzystaniem reprezentacji listy wątków w postaci drzewa binarnego. Każdy wątek przedstawiony został jako wierzchołek drzewa binarnego. Wszystkie wątki, z wyjątkiem wątku 0, posiadają dokładnie jednego rodzica. Wszystkie wątki, które nie są liśćmi oczekują na odebranie wyników sumowania od swoich dzieci. Natomiast wszystkie liście takiego drzewa wysyłają do swoich rodziców swoją wartość zmiennej współdzielonej `a`. Po odebraniu wyników następuje sumowanie wartości przechowywanej w wątku z wartością sumy dzieci i przekazanie danych do rodzica. Korzeń drzewa, czyli wątek 0, po zakończeniu całej procedury, zawiera zredukowaną wartość.

```
1     double sum = PCJ.getLocal("a");
2     if (PCJ.myId() >= PCJ.threadCount() / 2) {
3         PCJ.put((PCJ.myId() - 1) / 2, "tab",
4                 sum, (PCJ.myId() + 1) % 2);
5     } else {
6         int childCount = 2;
7         if ((PCJ.myId() + 1) * 2 == PCJ.threadCount()) {
8             childCount = 1;
9         }
10        PCJ.waitFor("tab", childCount);
11        for (int i = 0; i < childCount; ++i) {
12            sum += tab[i];
13        }
14        if (PCJ.myId() > 0) {
15            PCJ.put((PCJ.myId() - 1) / 2, "tab",
16                    sum, (PCJ.myId() + 1) % 2);
17        }
18    }
```

Listing 5.28: *Redukcja po drzewie binarnym z tablicą*

Przykładowa implementacja redukcji po drzewie binarnym z tablicą znajduje się na listingu 5.28.

Wyniki

Zebrane wyniki testu *Reduce* zależą od typu użytej operacji, liczby fizycznych węzłów i wątków w obliczeniach. W każdym przypadku redukowano za pomocą operacji sumy pojedynczą zmienną typu `double` przechowywaną na wszystkich wątkach a ostateczny wynik zapamiętywany był w wątku o numerze 0. Podobnie jak miało to miejsce w przypadku małych testów wydajnościowych, każdy test wykonywano pięciokrotnie na różnej liczbie wątków na jeden węzeł i zawsze wybierano najbardziej optymalny wynik dla sumarycznie tej samej liczby wątków.

Wyniki porównywano z operacją redukcji zaimplementowaną w MPI przedstawioną na listingu 5.29. Pierwszy parametr oznacza adres pamięci przechowujący zmienną do redukcji, drugi parametr to adres pamięci do przechowania wyniku operacji redukcji, kolejny informuje o liczbie elementów danych, następny o ich typie. Dalszymi parametrami są: informacja o typie operacji redukcji, numer procesora, w którym wynik ma być przechowany i uchwyt do komunikatora (ang. *communicator handle*), w ramach którego wykonywana jest operacja redukcji – w tym przypadku redukcja wykonywana jest w ramach wszystkich procesów.

```
1     double sum = NAN;
2     MPI_Reduce(&a, &sum, 1, MPI_DOUBLE,
3                   MPI_SUM, 0, MPI_COMM_WORLD);
```

Listing 5.29: *Operacja redukcji z wykorzystaniem MPI*

Rysunek 5.20 przedstawia czas wykonania pojedynczej operacji redukcji w zależności od liczby wątków na systemie *hydra* (5.20a) i *halo2* (5.20b). Poszczególne serie danych oznaczają przedstawione powyżej metody:

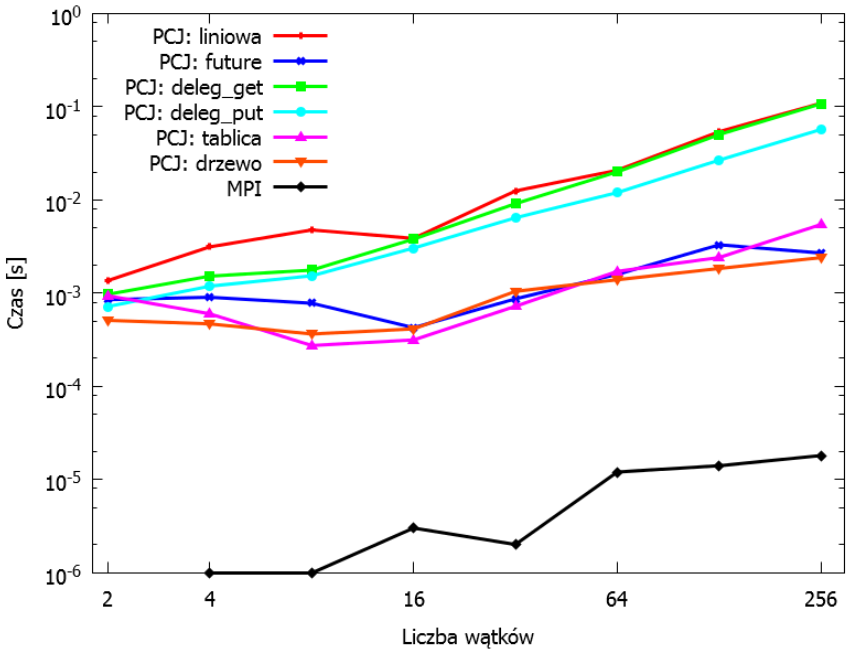
- **liniowa** – redukcja liniowa (listing 5.22),
- **future** – redukcja liniowa z obiektem `Future` w pętli (listing 5.24),
- **deleg_get** – redukcja liniowa z wydelegowanym sumowaniem (wersja z pobieraniem danych) (listing 5.25),
- **deleg_put** – redukcja liniowa z wydelegowanym sumowaniem (wersja z wysyłaniem danych) (listing 5.26),
- **tablica** – redukcja z tablicą (listing 5.27),
- **drzewo** – redukcja po drzewie binarnym z tablicą (listing 5.28),
- **MPI** – operacja redukcji wykonana za pomocą MPI (listing 5.29).

Na wykresach widać, że redukcja liniowa jest najwolniejszym sposobem redukcji. Jest to spowodowane tym, że wątek 0 pobiera dane z kolejnych wątków czekając za każdym razem aż uzyska dane – dopiero wówczas pobiera dane z kolejnego wątku.

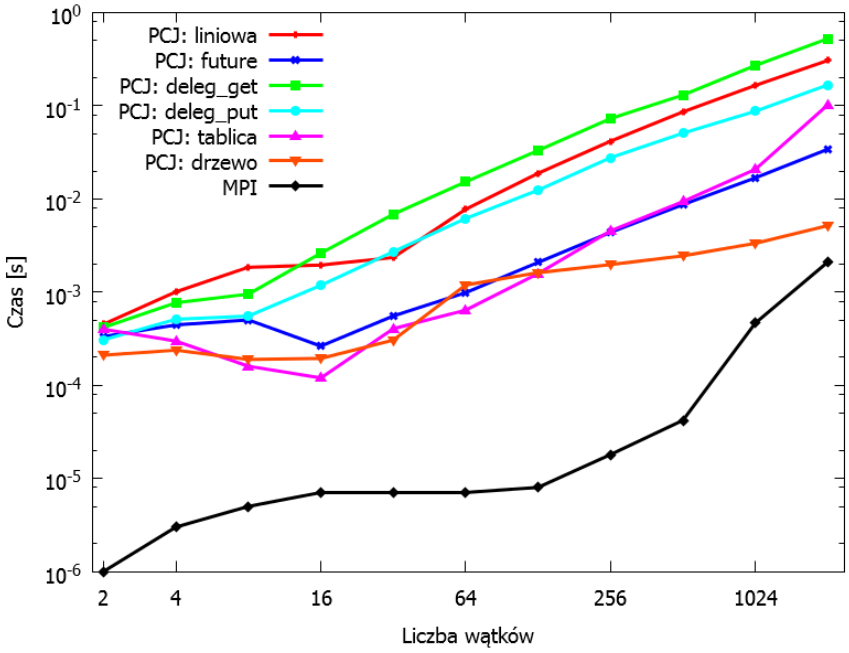
Podobna szybkość uzyskiwana jest dla redukcji liniowej z wydelegowanym sumowaniem, zarówno z pobieraniem jak i wysyłaniem danych. Wiąże się to z oczekiwaniem przez poszczególne wątki na wyniki obliczeń w innych wątkach, a dodatkowo tych momentów jest liniowo wiele w zależności od liczby wątków. Wersja z wysyłaniem danych jest szybsza od wersji z pobieraniem danych, bo nie wymaga korzystania z operacji bariery.

Szybszymi operacjami są: redukcja liniowa z obiektem `Future`, redukcja z tablicą i redukcja po drzewie binarnym. Redukcja liniowa z obiektem `Future` polega na odpytaniu wszystkich wątków o dane przez wątek 0 bez oczekiwania na uzyskanie danych. Oczekiwanie na dane następuje dopiero po zainicjowaniu wszystkich zapytań, a wątek 0 przetwarza każdą odpowiedź oddzielnie.

Podobnie do redukcji liniowej z obiektem `Future` działa redukcja z tablicą, z tą różnicą, że wątek 0 nie wysyła żądań o dane do pozostałych wątków, a jedynie odczytuje przesłane dane z lokalnej tablicy – pozostałe wątki wysyłają dane od razu, gdy tylko



(a) *hydra*



(b) *halo2*

Rysunek 5.20: *Czas wykonywania pojedynczej operacji redukcji w zależności od liczby wątków*

są one gotowe. Wątek 0 musi jedynie obsłużyć wszystkie polecenia zapisania danych do tablicy w sposób bezpieczny, co może powodować krótkie, ale częste wchodzenie w sekcję synchronizacji.

Najszybszą operacją redukcji okazuje się być redukcja po drzewie binarnym, która jest skrzyżowaniem operacji redukcji z tablicą i operacji redukcji z wydelegowanym sumowaniem. Operacja redukcji po drzewie binarnym jest najszybsza, ponieważ każdy wątek przetwarza wyniki maksymalnie tylko trzech wątków – swój i swoich dzieci. Ponadto, w przeciwieństwie do operacji redukcji z wydelegowanym sumowaniem, oczekiwanie na dane jest równoległe i w danym momencie nie czeka tylko jeden wątek, ale czeka ich więcej, w zależności od poziomu drzewa, na którym odbywa się przetwarzanie danych.

Operacja redukcji w przypadku korzystania z MPI dla małej liczby wątków jest znacznie szybsza niż w przypadku PCJ. Jednak od 256 wątków czas wykonywania operacji redukcji szybko rośnie. W tym zakresie skalowalność implementacji MPI jest zdecydowanie gorsza niż skalowalność PCJ. W efekcie czas wykonania redukcji dla 2048 wątków staje się porównywalny, a dla większej liczby wątków należy spodziewać się przewagi PCJ.

5.3 Podsumowanie

Przedstawione wyniki testów pokazują, że obliczenia równoległe i rozproszone wykonane z użyciem biblioteki PCJ pozwalają na uzyskanie wydajności porównywalnej z MPI. Szczególnie dobre wyniki uzyskiwane są dla dużych danych, co związane jest z bardzo dobrą (często lepszą niż w przypadku MPI czy ProActive) skalowalnością rozwiązań opartych o PCJ.

Na podstawie wyników można także wysnuć wniosek, że bardziej optymalnym działaniem jest maksymalne wykorzystanie rdzeni procesorów dostępnych na węźle niż korzystanie z takiej samej liczby wątków, ale podzielonych na wiele węzłów. Co więcej, na przedstawionych wykresach widać, że szybkość wykonania może zależeć nie tylko od sposobu rozdzielenia wątków aplikacji na węzły, ale również od wykorzystanej metody przesyłania danych. Widać także różnice w wydajności w zależności od wykorzystanej wirtualnej maszyny Java.

Wyniki wykonanych testów pokazują również, że nadal istnieją miejsca działania biblioteki, które można ulepszyć. Wśród nich główny nacisk powinien zostać postawiony na zwiększenie wydajności komunikacji wewnątrz węzła. Ponadto wszelkie mechanizmy, które w swoim działaniu wykorzystują przesyłanie małych porcji danych, również wymagają dokładniejszego spojrzenia i udoskonalenia.

Rozdział 6

PCJ – Przykładowe aplikacje

W tym rozdziale zostaną przedstawione przykładowe fragmenty aplikacji wraz z testami ich skalowalności. Omówione zostaną dwa sposoby wyliczania wartości liczby Pi, a także zostaną przedstawione wyniki uzyskane w teście *RayTracer* z pakietu testów Java Grande Forum Benchmark [3]. Ponadto zostanie przedstawione porównanie operacji *MapReduce* w aktualnie oficjalnej wersji języka Java 8 z analogiczną wersją zapisaną z wykorzystaniem biblioteki PCJ, a także porównanie tej samej operacji zapisanej przy użyciu biblioteki PCJ oraz biblioteki ProActive.

6.1 Pi

Wyznaczenie wartości liczby π jest kolejną przykładową aplikacją zrównoleglonąą z wykorzystaniem biblioteki PCJ. Przedstawione zostaną dwa sposoby przybliżania wartości liczby π:

- za pomocą metody Monte Carlo,
- za pomocą całkowania metodą prostokątów.

6.1.1 Metoda Monte Carlo

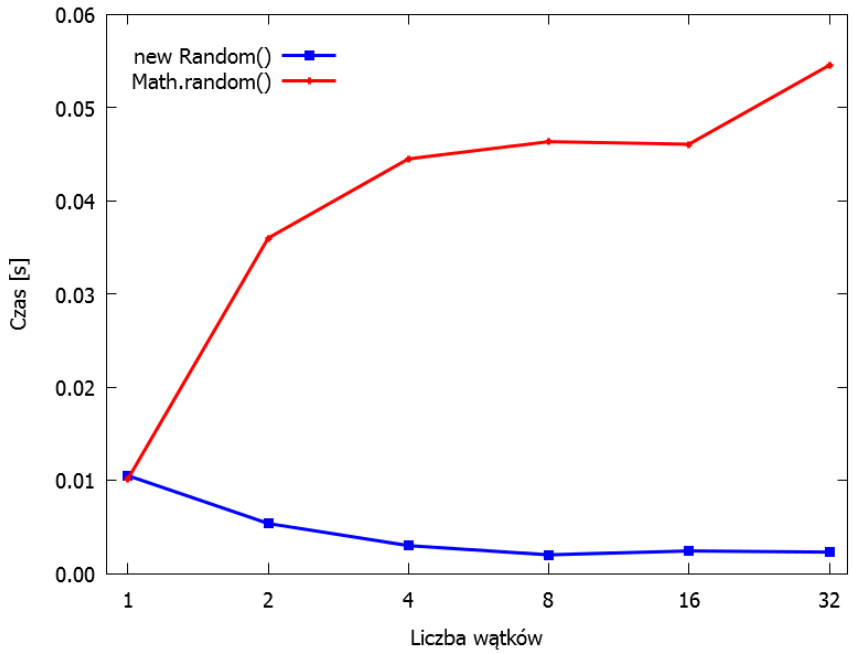
Metoda Monte Carlo obliczenia przybliżonej wartości liczby π przedstawiona w niniejszej pracy polega na wyborze losowych punktów z rozkładu jednostajnego w kwadracie $[-1..1] \times [-1..1]$ i wykonaniu sprawdzenia ile punktów należy do koła o środku w punkcie (0,0) i promieniu 1. Stosunek liczby punktów należących do koła do liczby wszystkich wylosowanych punktów przemnożony przez 4, daje przybliżoną wartość liczby π. Na listingu 6.1 przedstawiono implementację obliczania liczby π z zastosowaniem metody Monte Carlo.

```
1     Random random = new Random();
2     long nAll = 1_280_000_000;
3     long n = nAll / PCJ.threadCount();
4
5     long myCircleCount = 0;
6
7     for (long i = 0; i < n; ++i) {
8         double x = 2.0 * random.nextDouble() - 1.0;
9         double y = 2.0 * random.nextDouble() - 1.0;
10        if ((x * x + y * y) <= 1.0) {
11            myCircleCount++;
12        }
13    }
14    PCJ.putLocal("count", myCircleCount);
15    PCJ.barrier();
16
17    if (PCJ.myId() == 0) {
18        FutureObject cL[] =
19                new FutureObject[PCJ.threadCount()];
20        for (int p = 0; p < PCJ.threadCount(); p++) {
21            cL[p] = PCJ.getFutureObject(p, "count");
22        }
23        long globalCircleCount = 0;
24        for (FutureObject fo : cL) {
25            globalCircleCount = globalCircleCount
26                    + (long) fo.get();
27        }
28        return 4.0 * (double) globalCircleCount
29                / (double) (n * PCJ.threadCount());
30    }
31
32    return Double.NaN;
```

Listing 6.1: *Implementacja obliczania liczby π z zastosowaniem metody Monte Carlo*

W przykładowej implementacji nie zastosowano statycznej metody `random()` z klasy `java.lang.Math`, ale wykorzystano klasę `java.util.Random`, której każdy wątek posiada swoją własną instancję. Można również wykorzystać, wprowadzoną w wersji 7 języka Java, klasę `java.util.concurrent.ThreadLocalRandom`, która przy prawidłowym użyciu, zapewnia, że każdy wątek otrzyma swój własny, odizolowany od innych, generator liczb pseudolosowych.

Wykorzystanie statycznej metody `random()` powodowałoby znaczne spowolnienie działania programu przy wykorzystaniu więcej niż jednego wątku na węzeł. Działoby się tak dlatego, że klasy w pakiecie `java.lang` są wspólne dla wszystkich wątków PCJ. Stąd wywołanie metody `random` powodowałoby wykorzystanie statycznego, wspólnego dla wszystkich wątków, generatora liczb pseudolosowych. Generator ten, w celu obliczenia kolejnej liczby pseudolosowej, wykorzystuje w pętli operację atomową o nazwie *porównaj i zamień* (CAS, ang. *compare and swap*), by wygenerować nową wartość ziarna losowości (ang. *seed*). Wiele wątków współzawodniczyłoby w uaktualnieniu ziarna losowości, co powodowałoby wstrzymanie działania do czasu sukcesu tej operacji w wątku.



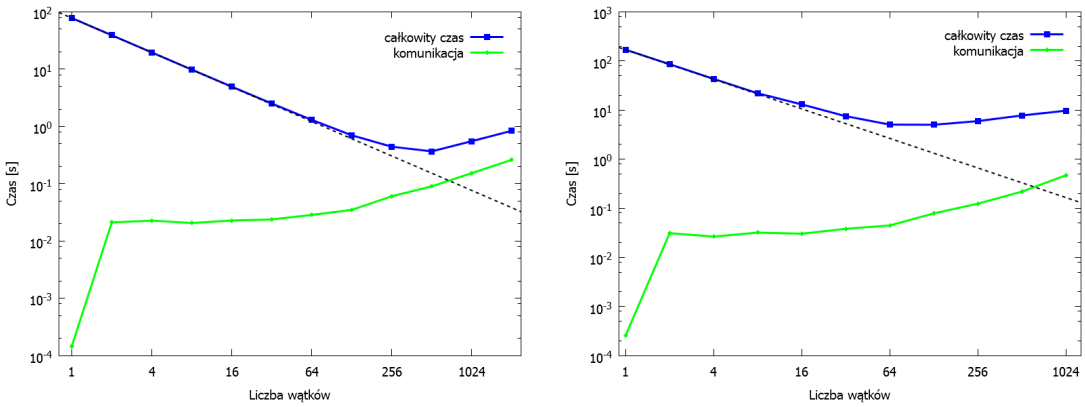
Rysunek 6.1: *Porównanie szybkości działania obliczenia wartości liczby π w metodzie Monte Carlo (*hydra*)*

Operacja CAS jest niepodzielna i przyjmuje trzy parametry. Pierwszym parametrem jest miejsce w pamięci, które będzie odczytywane i modyfikowane, drugim jest wartość w zależności od której pamięć będzie zmieniana, natomiast trzecim jest nowa wartość pamięci. Działanie atomowej, czyli niepodzielnej, operacji CAS polega na porównaniu wartości pamięci z drugim parametrem operacji a następnie, w przypadku gdy obie wartości są równe, modyfikacji pamięci wykorzystując trzeci

parametr. W przypadku, gdy wartość pamięci jest różna niż drugi parametr, czyli z reguły gdy pamięć została zmodyfikowana pomiędzy pobraniem wartości dla drugiego parametru a wykonaniem samej operacji CAS, operacja ta zwraca błąd. Mechanizm ten jest bezpieczny z punktu widzenia wielowątkowości.

Porównanie szybkości działania obliczenia wartości liczby π w przypadku wykorzystania statycznej metody `random()` (*Math.random()*) i instancji klasy `java.util.Random` (*new Random()*) w ramach jednego węzła przedstawiono na rysunku 6.1. W obu przypadkach losowano współrzędne 200 000 punktów, a wyniki zostały zebrane na systemie *hydra*. Na rysunku widać dużą różnicę czasu wykonania w zależności od zastosowanej metody wyznaczania kolejnej liczby pseudolosowej.

Każdy wątek wyznacza własną liczbę trafień w koło. Po skończonych obliczeniach, następuje operacja bariery, po której wątek o numerze 0 wykonuje redukcję z operacją sumy i zwraca obliczoną wartość.



(a) *halo2* **(b)** *boreasz*

Rysunek 6.2: *Czas obliczania przybliżonej wartości liczby π metodą Monte Carlo*

Na rysunku 6.2 przedstawiono skalowalność obliczenia przybliżonej wartości liczby π za pomocą metody Monte Carlo na systemie *halo2* i *boreasz*. W przypadku obu systemów, każdy test polegał na wylosowaniu łącznie 1 280 000 000 punktów przez wszystkie wątki biorące udział w obliczeniach. Przerywaną linią zaznaczono teoretyczne idealne skalowanie aplikacji. W przypadku systemu *halo2* skalowalność realna zachowana jest aż do 256 wątków obliczeń, natomiast w przypadku systemu *boreasz* wyniki są bliskie idealnym aż do wykorzystania 64 wątków. W obu przypadkach widać, że odchylenia związane są z czasem potrzebnym na wykonanie komunikacji. Początkowo, przy małej liczbie wątków, czas potrzebny na wykonanie obliczeń był wysoki w stosunku do czasu komunikacji, dzięki czemu komunikacja nie odgrywała dużej roli w skalowalności. Wraz ze wzrostem liczby wątków, zmniejszyła się ilość obliczeń na pojedynczym wątku, przez co komunikacja zaczęła być bardziej znacząca.

6.1.2 Całkowanie metodą prostokątów

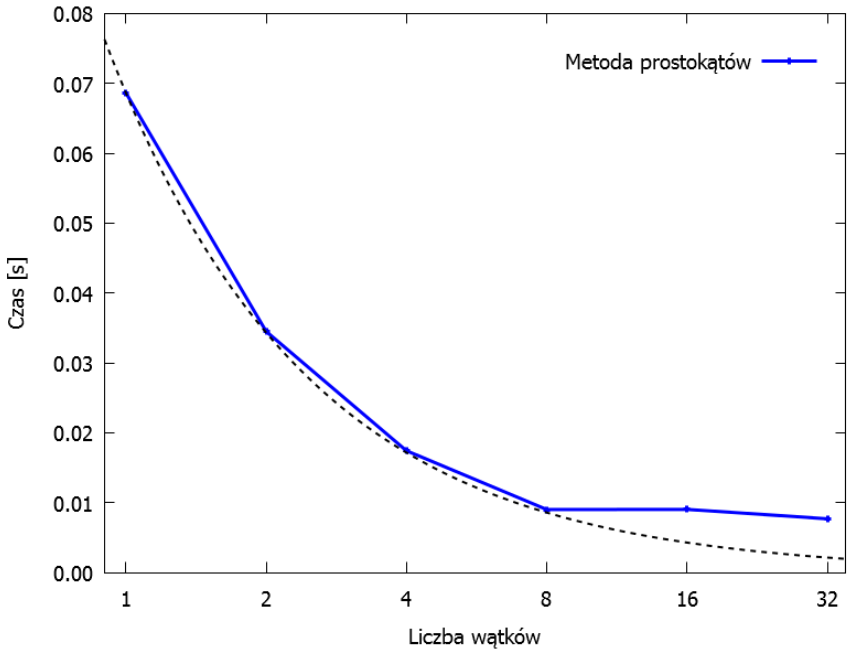
Dużo szybszym i bardziej dokładnym sposobem obliczenia wartości liczby π jest przybliżenie wartości całki:

$$\pi = \int_0^1 \frac{4}{1+x^2}\,\mathrm{d}x \approx \sum_{i=1}^{N} \frac{4}{1+\left(\frac{i-\frac{1}{2}}{N}\right)^2}$$

```
1     private double f(double x) {
2         return (4.0 / (1.0 + x * x));
3     }
4
5     private double calculateIntegralPi() {
6         long n = 10_000;
7         double w = 1.0 / (double) n;
8
9         double mySum = 0.0;
10        for (int i = PCJ.myId() + 1; i <= n;
11                  i += PCJ.threadCount()) {
12            mySum = mySum + f(((double) i - 0.5) * w);
13        }
14        mySum = mySum * w;
15        PCJ.putLocal("sum", mySum);
16        PCJ.barrier();
17
18        double gSum = 0.0;
19        if (PCJ.myId() == 0) {
20            FutureObject[] sL =
21                    new FutureObject[PCJ.threadCount()];
22            for (int i = 0; i < PCJ.threadCount(); ++i) {
23                sL[i] = PCJ.getFutureObject(i, "sum");
24            }
25            for (FutureObject fo : sL) {
26                gSum = gSum + (double) fo.getObject();
27            }
28        }
29
30        return gSum;
31    }
```

Listing 6.2: *Implementacja obliczania liczby π z zastosowaniem całkowania metodą prostokątów*

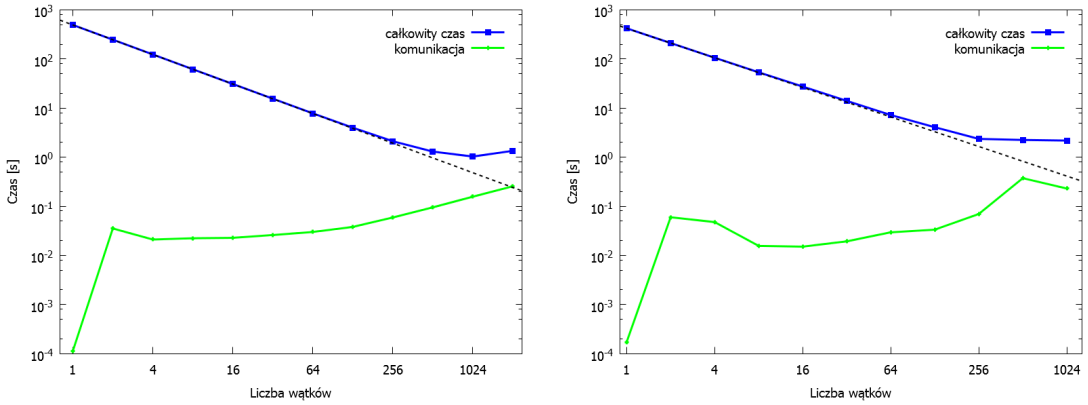
Listing 6.2 zawiera przykładową implementację pozwalającą na obliczenie wartości liczby π. Odcinek, na którym liczona jest wartość całki, jest dzielony na n równych części w zależności od liczby wątków. Każdy wątek oblicza pole prostokątów, o podstawie długości $\frac{1}{n}$ i boku wyznaczonym jako wynik wykonania metody `f()` w odpowiednich punktach odcinka, po czym sumuje wyliczone pola. Na koniec następuje operacja bariery i redukcja sumująca wyniki obliczeń każdego wątku.



Rysunek 6.3: *Wykres szybkości działania obliczenia wartości liczby π za pomocą całkowania metodą prostokątów (*hydra*)*

Czas potrzebny na wykonanie całkowania metodą prostokątów korzystając z 10 000 000 punktów przedstawione są na rysunku 6.3. Wyniki te zostały zebrane na systemie *hydra* z wykorzystaniem tylko jednego węzła. Linią przerywaną zaznaczono idealne skalowanie. Dla 16 i 32 wątków uzyskane wyniki odbiegają od idealnych rezultatów, gdyż przekroczono granicę liczby rdzeni na węźle.

Wykonano również testy dla większej liczby wątków. Wykresy na rysunku 6.4 przedstawiają czas potrzebny na wyliczenie przybliżonej wartości liczby π za pomocą przybliżenia całki metodą prostokątów. Przedział całkowania podzielono na 1 280 000 000 równych części. Linią przerywaną zaznaczono teoretyczny idealnie skalowalny całkowity czas obliczeń. Widać bardzo dobrą skalowalność rozwiązań do setek wątków w przypadku systemu *halo2* jak i systemu *hydra*. Podobnie jak miało to miejsce w przypadku wykorzystania metody Monte Carlo, skalowalność znacznie zaczęła odbiegać od idealnej wraz ze wzrostem czasu potrzebnego na komunikację.

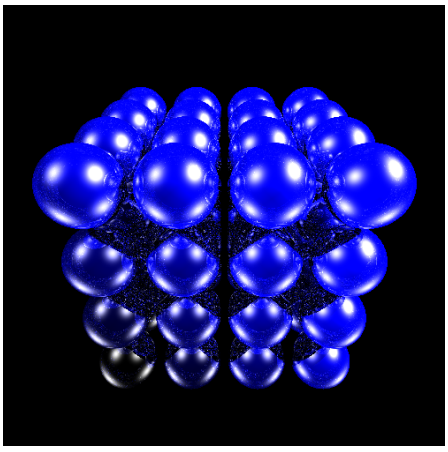


(a) *halo2* **(b)** *hydra*

Rysunek 6.4: *Czas obliczania przybliżonej wartości liczby π za pomocą przybliżania wartości całki metodą prostokątów*

6.2 RayTracer

RayTracer jest testem wchodzącym w skład Java Grande Forum Benchmark [3]. Jest to jeden z trzech testów sekcji aplikacji dużej skali. Test ten mierzy wydajność renderowania obrazu trójwymiarowej sceny za pomocą techniki śledzenia promieni. Na scenie znajdują się 64 kule ułożone w sześcian 4x4x4 i 5 źródeł światła, a generowany na płaszczyźnie rzutni obraz jest wymiaru $N \times N$ pikseli. Rysunek 6.5 przedstawia obraz sceny renderowanej w tym teście.

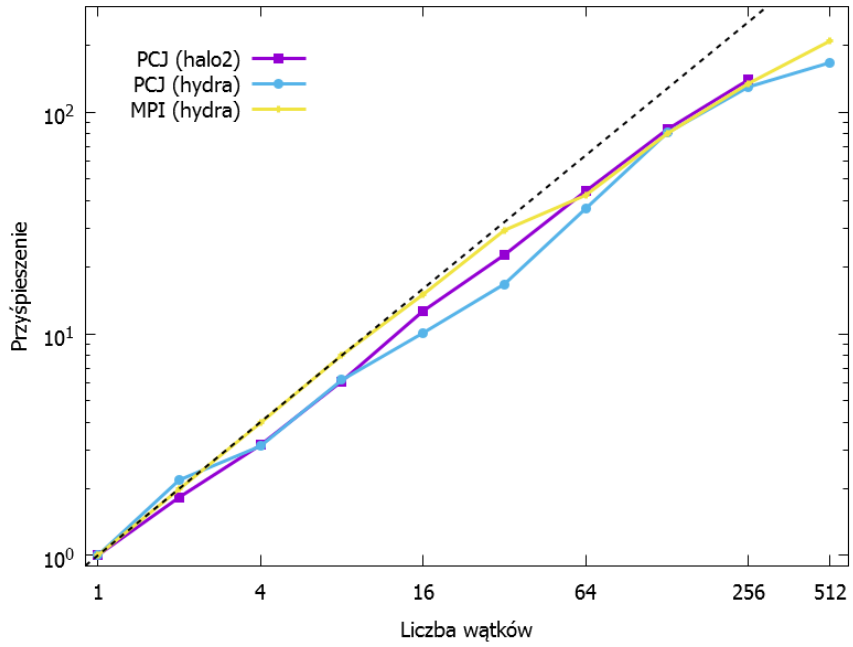


Rysunek 6.5: *Wygenerowany obraz sceny*

Z punktu obserwatora wyprowadzane są promienie pierwotne w każdy punkt rzutni, a dalej w stronę obiektów sceny. Wyznaczany jest najbliższy punkt przecięcia promienia z obiektem na scenie do obserwatora. Dla tego punktu wyznaczana jest jasność na podstawie wszystkich źródeł światła wraz z uwzględnieniem przesłaniania światła przez obiekty sceny. Dodatkowo test uwzględnia odbicie światła od kul wysyłając

promienie wtórne i w sposób rekurencyjny, z maksymalnym poziomem zagłębienia równym 6, wyznacza ostateczny kolor.

Równoległość w teście *RayTracer* uzyskuje się bardzo prosto, gdyż wszystkie promienie pierwotne i wtórne są niezależne od siebie. Jedynym momentem wymiany danych jest przesyłanie kolorów pikseli obrazu do wątku 0, a także przesłanie sumy kontrolnej.

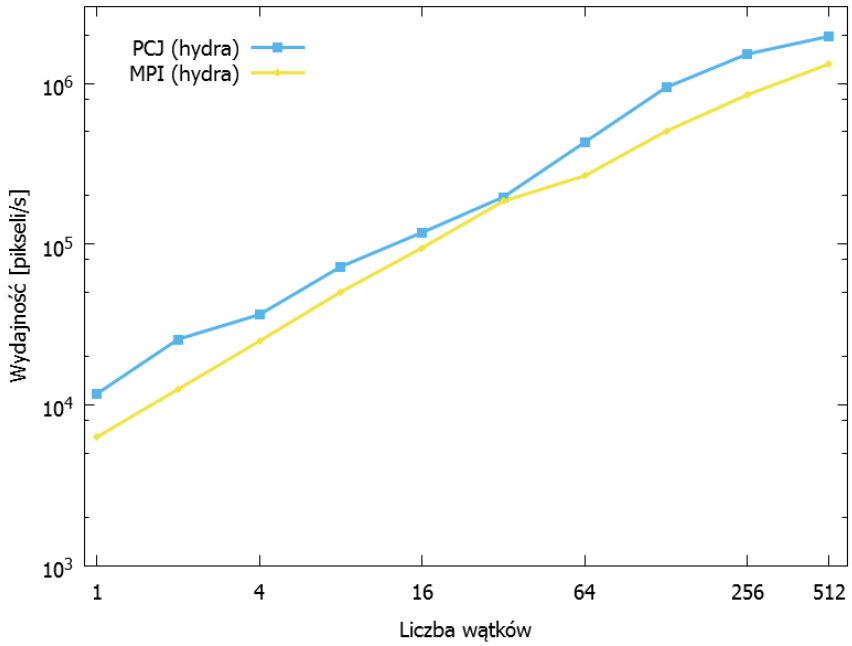


Rysunek 6.6: *Uzyskane przyśpieszenie względem liczby wątków*

Wykres przedstawiony na rysunku 6.6 pokazuje uzyskane przyśpieszenie dla testu *RayTracer* przy wymiarze sceny 2500 × 2500 pikseli z wykorzystaniem PCJ i MPI na systemie *halo2* zawierającym 16 rdzeni obliczeniowych na każdym węźle, a także na systemie *hydra* przy wykorzystaniu węzłów zawierających procesory *interlagos*, gdzie na każdym węźle znajdują się 64 rdzenie obliczeniowe. Linią przerywaną zaznaczono idealne skalowanie.

Rozwiązanie w MPI bezpośrednio oparto na kodzie z Java Grande Forum Benchmark [3]. Kod z języka Java przetłumaczono do C++ a nie do C, by jak najbardziej wykorzystać gotowy kod obiektowy. Tłumaczenie polegało głównie na zmianie pewnych słów kluczowych i poprawieniu składni, by była zgodna z językiem C++. Rozwiązanie w języku Java *implicite* korzysta z odśmiecacza pamięci (GC, ang. *Garbage Collector*) do zwalniania pamięci przez usunięcie z niej nieużywanych obiektów, czego nie udało się przenieść do rozwiązania w C++ i w związku z tym w pewnych sytuacjach tworzone są obiekty, które nie są usuwane z pamięci.

Na rysunku 6.6 widać, że wszystkie rozwiązania bardzo dobrze się skalują, a uzyskane przyśpieszenia są zbliżone do idealnego. Rozwiązanie oparte na MPI również skaluje się bardzo dobrze, a dla liczby wątków większej niż 64, skalowalność rozwiązania dla MPI jest taka sama jak dla PCJ. Dla mniejszej liczby wątków skalowalność dla MPI jest niemal idealna. Słabszy wynik dla PCJ związany jest z mechanizmami przekazywania danych wewnątrz węzła, a także z zaskakująco dobrymi osiągami dla obliczeń z wykorzystaniem jednego wątku, gdzie rozwiązanie oparte o język Java jest blisko dwukrotnie bardziej wydajne od wersji stworzonej w języku C++.



Rysunek 6.7: *Wydajność względem liczby wątków*

Rysunek 6.7 przedstawia wykres zależności wydajności obliczeń od liczby wątków. Na osi OY oznaczono uzyskaną wydajność w jednostce *piksele na sekundę*. Na podstawie tego wykresu można powiedzieć, że wydajność obliczeń w przypadku wykonania testu na jednym wątku jest zdecydowanie lepsza dla PCJ niż dla MPI. Szybsze wykonanie, czyli wyższa wydajność, utrzymuje się dla każdego rozmiaru zbioru wątków, poczynając od 1 wątku, gdzie rozwiązanie stworzone w języku Java wykonuje *raytracing* z szybkością ponad 11 500 pikseli/s, podczas gdy rozwiązanie powstałe w języku C++ uzyskuje niecałe 6 300 pikseli/s, przez 32 wątki (powyżej 196 000 pikseli/s dla wersji z wykorzystaniem PCJ oraz niecałe 185 000 pikseli/s dla wersji MPI), do 512 wątków, gdzie rozwiązanie w języku Java oparte o bibliotekę PCJ uzyskuje szybkość wyższą niż 1 950 000 pikseli/s, a rozwiązanie w C++ wykorzystujące MPI uzyskuje niecałe 1 320 000 pikseli/s. Wynik ten może być bardzo zaskakujący, gdyż w powszechnym przekonaniu aplikacje napisane w języku Java są wolniejsze od analogicznych aplikacji stworzonych w języku C++.

6.3 MapReduce

MapReduce to nazwa modelu przetwarzania danych w ramach obliczeń równoległych i rozproszonych. Składa się z dwóch następujących kroków:

1. Map – w tym kroku następuje filtrowanie i zwrócenie zmapowanych, przetworzonych danych wejściowych jako danych wejściowych do kroku *Reduce*,
2. Reduce – krok ten zbiera wyniki uzyskane w trakcie kroku *Map* i wykonuje operację redukcji, podsumowania.

Przykładowo, chcąc podać średni wiek użytkowników pewnego systemu komputerowego, krok *Map* zmapuje rekord użytkownika na wiek, natomiast krok *Reduce* wykona operację sumy na wieku. Następnie wyróżniony wątek, z reguły wątek główny, otrzymuje wynik operacji redukcji, czyli sumę wieku wszystkich użytkowników, i wylicza średni wiek.

Pętla *for-each*

```
1     long sum = 0;
2     for (User user : users) {
3         sum += user.getAge();
4     }
5     double average = (double) sum / users.size();
```

Listing 6.3: MapReduce *zapisany w języku Java 7*

Kod źródłowy zapisany w aktualnie ogólnie dostępnej wersji języka Java przedstawiony jest na listingu 6.3. Jest to kod sekwencyjny, możliwy do uruchomienia przez jeden wątek. Wykorzystuje pętle *for-each*, która *de facto* dostępna jest w języku Java od wersji 5.

Wykorzystanie puli `ForkJoinPool`

Listing 6.4 przedstawia implementację operacji *MapReduce* z wykorzystaniem puli `java.util.concurrent.ForkJoinPool`, która została dodana do języka Java w wersji 7. W ramach tej puli dostępny jest mechanizm *podkradania zadań* (ang. *work-stealing*), w którym wszystkie wątki w ramach puli wykonują zadania, które zostały *explicite* wysłane do niej lub zostały stworzone przez inne aktywne zadania. Implementacja ta wykorzystuje wielowątkowość komputera, na którym została uruchomiona.

Klasa `ForkMapReduce` reprezentuje pojedyncze rekurencyjne zadanie zwracające wynik typu `Long`. W konstruktorze klasy przekazywana jest lista użytkowników do

```
1  class ForkMapReduce extends RecursiveTask<Long> {
2
3      final private static int threshold = 16_384;
4      final private List<User> list;
5
6      protected ForkMapReduce(List<User> list) {
7          this.list = list;
8      }
9
10     @Override
11     protected Long compute() {
12         int length = list.size();
13
14         if (length < threshold) {
15             long sum = 0;
16             for (User u : list) {
17                 sum += u.getAge();
18             }
19             return sum;
20         }
21
22         int split = length / 2;
23
24         ForkMapReduce left =
25                 new ForkMapReduce(list.subList(0, split));
26         ForkMapReduce right =
27                 new ForkMapReduce(list.subList(split, length));
28
29         invokeAll(left, right);
30         return left.getRawResult() + right.getRawResult();
31     }
32 }
:                              ...
38         ForkMapReduce fmr = new ForkMapReduce(users);
39         ForkJoinPool.commonPool().invoke(fmr);
40         long result = fmr.getRawResult();
41         double average = (double) result / users.size();
```

Listing 6.4: MapReduce *zapisany z wykorzystaniem metody* fork/join

przetworzenia. Metoda `compute()` wykonywana jest przez wątki puli `ForkJoinPool`. W ramach tej metody, gdy długość listy jest mniejsza niż zadany próg, liczona jest suma wieku użytkowników w sposób sekwencyjny. W przeciwnym przypadku zadanie sumowania wieku jest dzielone na dwa podzadania (każde zawierające połowę listy) i zwrócenie sumy wyników uzyskanych przez podzadania. Linie 38-41 przedstawiają sposób rozpoczęcia wykonywania zadania w ramach puli wątków. Linia 38 tworzy nowy obiekt-zadanie `fmr` przesyłający pełną listę użytkowników. Linia 39 wykorzystuje wspólną dla wirtualnej maszyny Java pulę wątków i uruchamia na niej zadanie `fmr`. Po zakończeniu wykonania, czyli po policzeniu sumy wieku użytkowników, w linii 40 następuje odczytanie wyniku zadania `fmr`, a w linii 41 wyliczany jest średni wiek użytkowników.

Strumienie równoległe

Java 8 wprowadza do języka Java wiele nowości. Wśród nich można wymienić strumienie, wyrażenia lambda, interfejsy funkcyjne i referencje do metod. W związku z tymi zmianami, kod odpowiedzialny za obliczenie średniej wieku można zapisać w nowy sposób.

```
1     long sum = users.parallelStream()
2                     .map(u -> (long) u.getAge())
3                     .reduce(Long::sum)
4                     .get();
5     double average = (double) sum / users.size();
```

Listing 6.5: *Ogólna wersja* MapReduce *zapisana w języku Java 8*

Listing 6.5 zawiera przykładową implementację wykorzystującą strumienie równoległe (linia 1), wyrażenia lambda (linia 2) i referencje do metod (linia 3). Strumienie przetwarzane są przez operacje *leniwe* lub *pośrednie* (ang. *lazy/intermediate operation*), aż do *operacji końcowej* (ang. *terminal operation*), wymagającej rzeczywistego przetworzenia strumienia (linia 4). Strumienie równoległe domyślnie przetwarzają elementy kolekcji wykorzystując tyle wątków ile jest dostępnych procesorów i rdzeni na maszynie. Możliwe jest wykorzystanie innej liczby wątków, ale zmiana tego parametru jest jednorazowa i ma ona wpływ na całą wirtualną maszynę Java.

Zapis przedstawiony na listingu 6.5 jest ogólny i prawie niezależny od użytego typu danych. Z tego względu nie jest najszybszą możliwą implementacją. Możliwe jest zwiększenie wydajności przez wykorzystanie bardziej precyzyjnych typów danych. Listing 6.6 przedstawia taką właśnie zmianę.

Można również zastosować istniejącą metodę `sum()` do zsumowania elementów strumienia, co zostało przedstawione na listingu 6.7.

```
1     long sum = users.parallelStream()
2                     .mapToLong(u -> u.getAge())
3                     .reduce(Long::sum)
4                     .getAsLong();
5     double average = (double) sum / users.size();
```

Listing 6.6: *Bardziej szczegółowa wersja* MapReduce *zapisana z wykorzystaniem składni języka Java 8*

```
1     long sum = users.parallelStream()
2                     .mapToLong(u -> u.getAge())
3                     .sum();
4     double average = (double) sum / users.size();
```

Listing 6.7: *Zastosowanie metody* `sum()` *jako operacji redukcji w* MapReduce

```
1     double average = users.parallelStream()
2                     .mapToLong(u -> u.getAge())
3                     .average().orElse(Double.NaN);
```

Listing 6.8: *Zastosowanie metody* `average()` *jako operacji redukcji w* MapReduce

Strumienie zawierają także metody, które pozwalają policzyć wartość średnią bezpośrednio z elementów strumienia co pokazuje listing 6.8. Metoda `orElse(Double.NaN)` zwraca uzyskany wynik lub, gdy strumień jest pusty, zostanie zwrócona wartość *NaN*.

```
1     AtomicLong sum = new AtomicLong(0);
2     users.parallelStream()
3             .forEach(u -> sum.addAndGet(u.getAge()));
4     double average = (double) sum.get() / users.size();
```

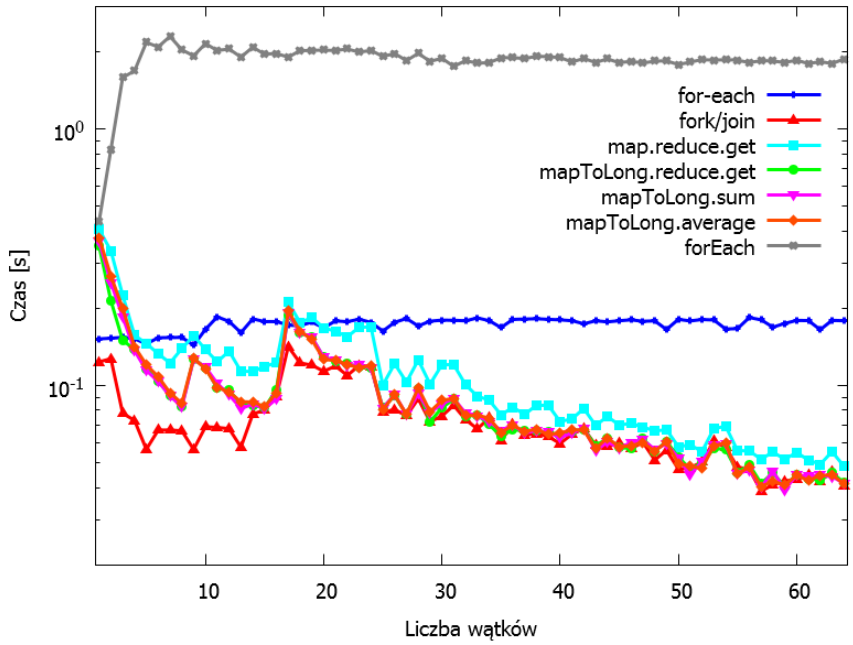
Listing 6.9: *Zastosowanie metody `forEach` i zmiennej atomowej do policzenia wartości sumy w* MapReduce

Ostatni sposób policzenia wartości sumy, przedstawiony na listingu 6.9, nie jest szybszy niż pozostałe, ale ze względu na to, iż pokazuje inną możliwość wykorzystania strumieni, jest bardzo ciekawy. Strumienie nie mogą modyfikować lokalnych zmiennych metod, a wszystkie użycia takich zmiennych mogą być wykonywane tylko przy założeniu, że dana zmienna jest *efektywnie finalna* (ang. *effective final*), co w skrócie można opisać jako zmienną, która może zostać zadeklarowana z modyfikatorem `final` nie doprowadzając do błędu kompilacji. Przykładem takiej zmiennej jest zmienna `sum`, która nie zmienia swojej wartości, gdyż zawsze wskazuje na tę samą instancję obiektu. Jednakże już wewnątrz tego obiektu mogą występować zmiany i w ten sposób możliwe jest sumowanie wartości wieku w zmiennej `sum`. Ze względu na to, że strumień jest równoległy, należało zastosować taki obiekt, który pozwalałby na wątkowo-bezpieczne modyfikowanie przechowywanej wartości. Na listingu przedstawiono klasę `AtomicLong`, ale równie dobrze możliwe byłoby stworzenie własnej klasy, która pozwoliłaby każdemu wątkowi modyfikować jego własną wartość sumy, po czym, na samum końcu, następowałoby sumowanie wyników z każdego wątku.

Porównanie metod

Różnice szybkości w zależności od zastosowanej metody obliczeń znajdują się na rysunku 6.8. Kompilację wykonano i wyniki zebrano z wykorzystaniem pakietu JDK 8 Build b128 Early Access (Release Candidate 1). Wyniki zebrano na klastrze *hydra* na węzłach *interlagos* zawierających procesory mające łącznie 64 rdzenie. Poszczególne serie danych oznaczają różne sposoby wykonania operacji *MapReduce*:

- **for-each** – *MapReduce* zapisany z wykorzystaniem pętli *for-each* (listing 6.3),
- **fork/join** – *MapReduce* zapisany w języku Java 7 z wykorzystaniem metody *fork/join* (listing 6.4),
- **map.reduce.get** – ogólna wersja *MapReduce* zapisana w języku Java 8 (listing 6.5),



Rysunek 6.8: *Czas działania różnych implementacji operacji* MapReduce

- **mapToInt.reduce.get** – bardziej szczegółowa wersja *MapReduce* zapisana z wykorzystaniem składni języka Java 8 (listing 6.6),
- **mapToInt.sum** – zastosowanie metody `sum()` jako operacji redukcji w *MapReduce* (listing 6.7),
- **mapToInt.average** – zastosowanie metody `average()` jako operacji redukcji w *MapReduce* (listing 6.8),
- **forEach** – zastosowanie metody `forEach` i zmiennej atomowej, do policzenia sumy w *MapReduce* (listing 6.9).

Ze względu na to, iż `parallelStream` korzysta ze wspólnej dla całej wirtualnej maszyny Java instancji klasy puli `ForkJoinPool`, która jest tworzona w zależności od liczby dostępnych procesorów i rdzeni w systemie, zastosowano mechanizmy ograniczające możliwości wykorzystania rdzeni procesora w trakcie testów. Pierwszym mechanizmem było uruchomienie aplikacji przy użyciu dostępnego z poziomu systemu Linux narzędzia `taskset`, które pozwala na przypisanie procesorów i ich rdzeni do uruchamianego zadania. Drugim mechanizmem było uruchomienie wirtualnej maszyny z parametrem:

`-Djava.util.concurrent.ForkJoinPool.common.parallelism=`X,

który oznacza, że wspólna pula wątków w klasie `ForkJoinPool` będzie stworzona z równoległością ustaloną na X, czyli z wykorzystaniem $X + 1$ rdzeni procesora.

Oś pozioma oznacza liczbę wątków biorących udział w przetwarzaniu danych, z tą różnicą, że dla wersji w starym stylu programowania (*for-each*), każdy punkt

jest wyliczony tylko dla 1 wątku obliczeń, gdy wirtualna maszyna ma możliwość wykorzystania większej liczby rdzeni. Oś pionowa oznacza pełen czas wykonania obliczenia średniej wieku 12 000 000 osób.

Słaby wynik najbardziej ogólnej wersji strumieniowej (*map.reduce.get*) spowodowany jest tym, że wszystkie operacje wykonywane są na typach obiektowych, stąd częsta potrzeba wykonywania operacji *autoboxing* i *unboxing* pomiędzy typem obiektowym (`java.lang.Long`) a typem prymitywnym (`long`).

Przy wykorzystaniu pętli *for-each* dane nie zostały podzielone między wątki, stąd stały czas wykonywania obliczeń. Ponadto jest on wyższy niż wersji z równoległym strumieniem, gdyż pojedynczy wątek musi przejść po wszystkich elementach listy. Natomiast przy braku równoległości lub bardzo małej liczbie możliwych do wykorzystania wątków, wyniki są konkurencyjne, gdyż nie ma narzutu na tworzenie strumienia i dzielenie go pomiędzy wątki.

Wersje strumienia, które mapują wartości bezpośrednio do typu prymitywnego `long` działają szybciej od wersji obiektowej. W stosunku do siebie działają jednak podobnie szybko. Widać także ich skalowalność wraz ze wzrostem ilości wątków.

Bardzo dobrze spisuje się wersja *fork/join*. Już przy wykorzystaniu dwóch wątków uzyskiwany jest najlepszy czas. Jest to spowodowane lepszym podziałem danych do przetworzenia na mniejsze fragmenty, niż miało to miejsce w przypadku strumieni równoległych.

Najgorszy wynik sposobu opartego na operacji *forEach* i zmiennej atomowej nie dziwi, gdyż wątkowo-bezpieczny dostęp do tej zmiennej powoduje, że każdy wątek musi oczekiwać na możliwość uaktualnienia przechowywanej wartości przed przejściem do kolejnego elementu strumienia. Ma tu także miejsce omawiana wcześniej operacja atomowa o nazwie *porównaj i zamień*.

Na wykresach wersji *fork/join* i strumieniowych, z wyjątkiem zastosowania metody `forEach`, występują widoczne schodki w okolicy 8-9 i 16-17 wątków. Dzieje się tak ze względu na budowę procesora *AMD Opteron™Processor 6272 (Interlagos)*, na którym testy były wykonywane. Testy były uruchamiane na rdzeniach od 0 do 63. Stąd przy wykorzystaniu 9 rdzeni, nastąpiło wyjście poza jeden węzeł NUMA. W przypadku kolejnego skoku, czyli przy 17 wątkach, nastąpiło wyjście poza jeden procesor.

6.3.1 Porównanie z PCJ

Istnieje możliwość zapisu algorytmu *MapReduce* także z wykorzystaniem biblioteki PCJ. Kod źródłowy nie różni się bardzo od aktualnego stylu zapisu kodu źródłowego z wykorzystaniem pętli *for-each*.

Listing 6.10 zawiera przykładową implementację *MapReduce* z wykorzystaniem PCJ. Każdy wątek przetwarza co N-ty element listy użytkowników, gdzie N to liczba

```
1     @Shared
2     long sum;
:                                  ...
13    int i = 0;
14    long s = 0;
15    for (User u : users) {
16        if (i++ % PCJ.threadCount() == PCJ.myId()) {
17            s += u.getAge();
18        }
19    }
20    PCJ.putLocal("sum", s);
21    PCJ.barrier();
22    s = pcj_reduce("sum");
23    if (PCJ.myId() == 0) {
24        double average = (double) s / users.size();
25    }
```

Listing 6.10: MapReduce *w PCJ (pełna lista)*

wątków. Na koniec następuje operacja redukcji. Może ona mieć formę przedstawioną w podrozdziale 5.2.4 na stronie 120.

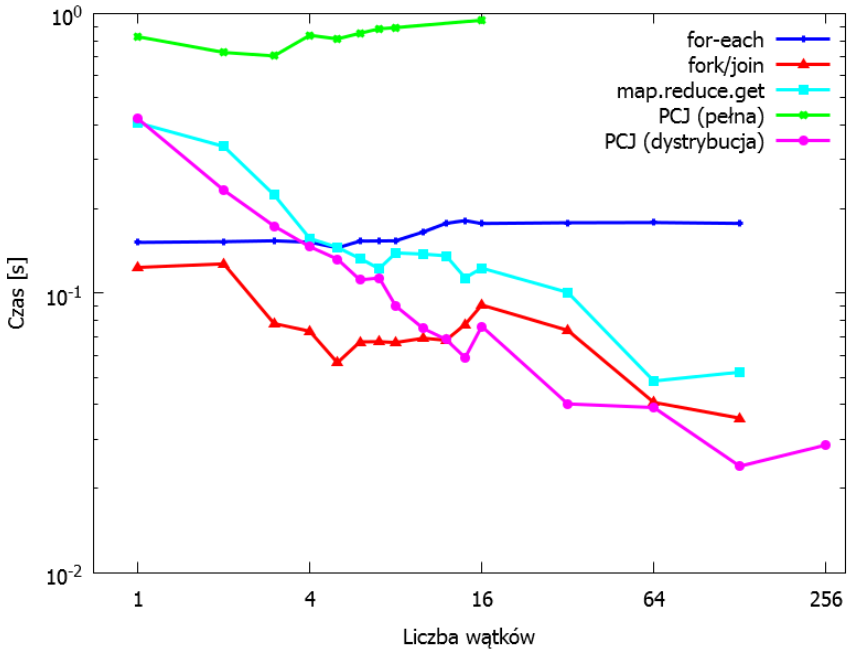
Na listingu 6.10 założono, że każdy wątek zawiera pełną kopię listy użytkowników, co może powodować problemy z brakiem pamięci. Lepszym rozwiązaniem jest rozdzielenie listy użytkowników we wcześniejszym etapie, na przykład w momencie wczytywania danych, i przetwarzanie przez każdy wątek tylko tych użytkowników, którzy do danego wątku należą. Wówczas kod znacznie się upraszcza co widać na listingu 6.11. Z drugiej strony kod ten wymaga dodatkowo przesłania liczby użytkowników z każdego wątku i wykonania operacji redukcji tej liczby, by policzyć średnią.

Rysunek 6.9 przedstawia porównanie czasu działania implementacji operacji *MapReduce* z wykorzystaniem pętli *for-each*, strumieni (*map.reduce.get*) i PCJ. W każdym przypadku baza użytkowników zawierała 12 000 000 elementów. Testy wykonano na węzłach klastra *hydra* z 64 rdzeniami obliczeniowymi, gdzie wykorzystano ponad 250 wątków, które konkurowały ze sobą o zasoby. W przypadku rozwiązań opartych na pętli *for-each* i strumieniach (*map.reduce.get*) możliwe było wykorzystanie tylko 1 węzła do obliczeń, natomiast wyniki w przypadku PCJ zebrano z wykorzystaniem 1, 2 i 4 węzłów.

Na wykresie przedstawione są dwie serie danych związanych z biblioteką PCJ. *PCJ (pełna)* oznacza wersję, w której każdy wątek zawiera pełną kopię listy użytkowników, stąd wyniki są dostępne dla nie więcej niż 16 wątków – przy większej liczbie wątków, aplikacja nie działała prawidłowo ze względu na wyczerpanie pamięci. *PCJ (dystrybucja)* oznacza implementację, w której każdy wątek zawiera listę jedynie tych użytkowników,

```
1       @Shared
2       long sum;
3       @Shared
4       int usersCount;
⋮                                   ...
15      myUsers = loadUsers(PCJ.myId());
16      long s = 0;
17      for (User u : myUsers) {
18          s += u.getAge();
19      }
20      PCJ.putLocal("sum", s);
21      PCJ.putLocal("usersCount", myUsers.size());
22      PCJ.barrier();
23      s = pcj_reduce("sum");
24      int count = pcj_reduce("usersCount");
25      if (PCJ.myId() == 0) {
26          double average = (double) s / count;
27      }
```

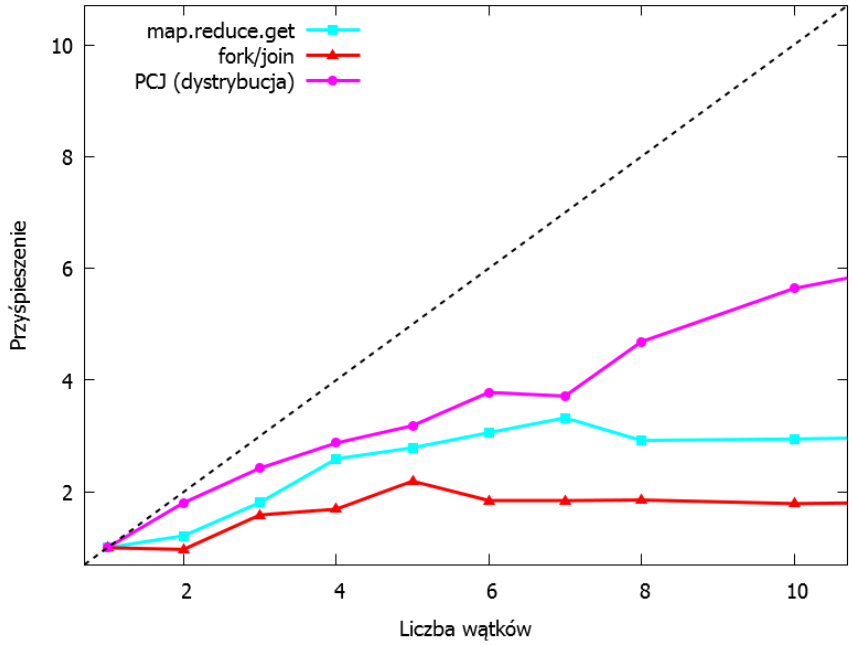
Listing 6.11: MapReduce *w PCJ (lista rozdystrybuowana)*



Rysunek 6.9: *Porównanie czasu działania implementacji operacji* MapReduce *z wykorzystaniem pętli* for-each*, strumieni i PCJ*

na których ma dokonać obliczeń. Innymi słowy pełna lista użytkowników została rozdystrybuowana pomiędzy wątki PCJ.

Na podstawie przedstawionych danych widać, że przetwarzanie dużej liczby elementów powoduje, że wykorzystanie strumienia równoległego jest bardziej opłacalne niż stosowanie pętli *for-each* już dla 4 wątków. Ponadto widać, że w momencie, gdy przekroczona zostanie bariera 8 wątków rozwiązanie oparte o PCJ zaczyna być szybsze od wersji z równoległymi strumieniami, a powyżej 12 wątków szybsze nawet od wersji *fork/join*.



Rysunek 6.10: *Przyśpieszenie uzyskane przy użyciu strumieni i PCJ*

Rysunek 6.10 przedstawia porównanie przyśpieszenia uzyskiwanego w przetwarzaniu listy użytkowników przy użyciu strumieni i PCJ. Przyśpieszenie, które daje się uzyskać przy wykorzystaniu strumieni jest trzykrotne, dwukrotne przyśpieszenie jest możliwe przy zastosowaniu metody *fork/join*, natomiast blisko sześciokrotne przyśpieszenie jest możliwe do uzyskania przy zastosowaniu PCJ. Ponadto przy małej liczbie wątków, rozwiązanie oparte o PCJ skaluje się prawie zgodnie z teoretycznie idealnym skalowaniem.

6.3.2 Porównanie PCJ z ProActive

W tym podrozdziale zostanie zaprezentowane porównanie operacji *MapReduce* zapisanej w bibliotece PCJ z analogiczną operacją zapisaną z wykorzystaniem biblioteki ProActive.

Kod źródłowy zapisany z użyciem biblioteki PCJ został przedstawiony w podrozdziale 6.3.1, natomiast listingi 6.12 i 6.13 zawierają fragmenty kodu źródłowego użytego do implementacji operacji *MapReduce* z wykorzystaniem biblioteki ProActive.

```
1  public static class ActiveObject {
2      final private int id;
3      private List<User> users = new ArrayList<>();
4
5      public ActiveObject() { this(-1); }
6      public ActiveObject(int id) { this.id = id; }
7      public BooleanWrapper prepareUsers(int threadCount) {
8          for (int i = 0; i < 12_000_000; i++) {
9              if (i % threadCount == id) {
10                 users.add(new User(i, "N" + i, "S", 40 + i));
11             }
12         }
13         return new BooleanWrapper(true);
14     }
15     public LongWrapper calculateAgeSum() {
16         long sum = 0;
17         for (User u : users) {
18             sum += u.getAge();
19         }
20         return new LongWrapper(sum);
21     }
22     public IntWrapper getCount() {
23         return new IntWrapper(users.size());
24     }
25 }
```

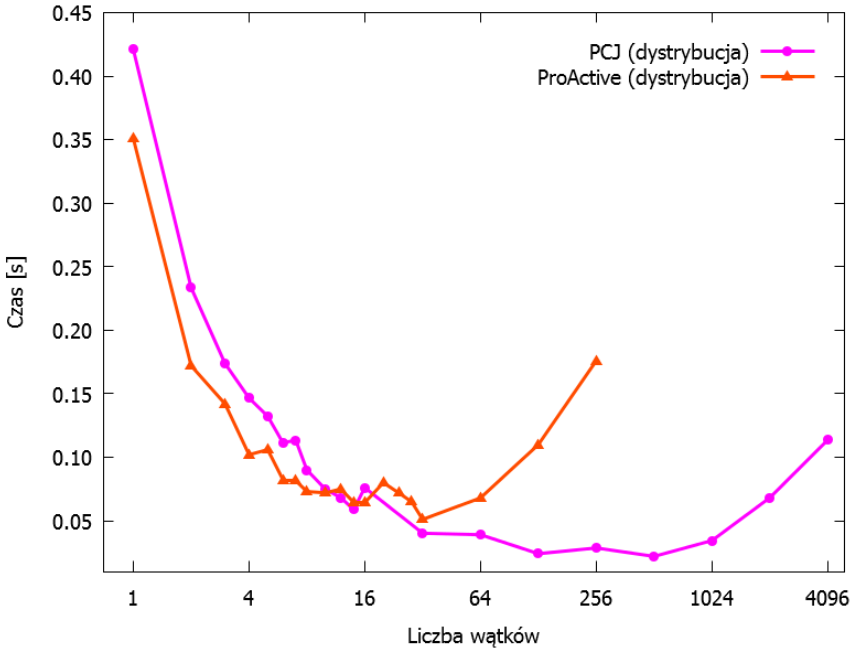
Listing 6.12: *Klasa reprezentująca aktywny obiekt dla ProActive*

Listing 6.12 przedstawia klasę reprezentującą aktywny obiekt. Klasa ta zawiera bezargumentowy konstruktor wymagany przez ProActive, jak również konstruktor przyjmujący jeden argument będący liczbą całkowitą, oznaczającą identyfikator aktywnego obiektu. Ponadto znajdują się w klasie trzy metody `BooleanWrapper prepareUsers(int threadCount)`, `LongWrapper calculateAgeSum()` oraz `IntWrapper getCount()`. Pierwsza metoda służy do przygotowania listy z użytkownikami do przetworzenia. W przykładzie użytkownicy są tworzeni w pętli, ale równie dobrze mogą być wczytywani z pliku. Druga metoda ma za zadanie policzyć sumę wieku wszystkich użytkowników przechowywanych w lokalnej liście z użytkownikami. Ostatnia metoda zwraca liczbę użytkowników na liście.

```
1     /* ... preparing active objects group ... */
2     Object[][] params = new Object[nodes.length][1];
3     for (int i = 0; i < nodes.length; ++i) params[i][0] = i;
4     ActiveObject aoGroup = (ActiveObject) PAGroup.newGroup(
5             ActiveObject.class.getName(), params, nodes);
6
7     /* ... prepare data ... */
8     PAGroup.waitAll(aoGroup.prepareUsers(nodes.length));
9
10    /* ... test body ... */
11    LongWrapper sumGroup = aoGroup.calculateAgeSum();
12    IntWrapper countGroup = aoGroup.getCount();
13
14    long sum = 0;
15    while (PAGroup.size(sumGroup) > 0) {
16        LongWrapper sumW = (LongWrapper) PAGroup
17                .waitAndGetOneThenRemoveIt(sumGroup);
18        sum += sumW.getLongValue();
19    }
20
21    int count = 0;
22    while (PAGroup.size(countGroup) > 0) {
23        IntWrapper countW = (IntWrapper) PAGroup
24                .waitAndGetOneThenRemoveIt(countGroup);
25        count += countW.getIntValue();
26    }
27    double average = (double) sum / count;
```

Listing 6.13: *Kod głównego wątku zlecający zadania do grupy aktywnych obiektów i przetwarzający ich odpowiedzi*

Na listingu 6.13 znajduje się kod głównego wątku wykonującego obliczenia z wykorzystaniem aktywnych obiektów. Na początku wątek ten tworzy tablicę zawierającą parametry wywołania konstruktorów, a następnie wykorzystuje ją w trakcie tworzenia grupy aktywnych obiektów. Po stworzeniu grupy, każdy aktywny obiekt w grupie przygotowuje dane do przetwarzania. W przykładzie są one tworzone w pętli, ale mogłyby być one wczytywane z pliku. Po zakończeniu tej operacji przez wszystkie aktywne obiekty w grupie, następuje główna część programu. Mierzony jest czas wykonania tej części programu. Na początek wywoływane są asynchroniczne metody grupy, zwracające obiekty typu *Future* zawierające wynik operacji. Ze względu na to, że operacja została wywołana przez grupę, wynik również jest grupą. Następnie odczytywane są wszystkie dane z z grupy zawierającej wynik sumowania wieku. Po ich zsumowaniu przeliczana jest również jest łączna liczba użytkowników. Po przetworzeniu tych dwóch danych możliwe jest policzenie średniej. Bezpośrednio po policzeniu średniej wyliczany jest czas jaki upłynął od momentu poprzedzającego wysłanie żądań do grupy aktywnych obiektów do momentu zakończenia obliczania średniej wieku, czyli czas wykonywania operacji *MapReduce*.



Rysunek 6.11: *Czas potrzebny na policzenie średniej wieku użytkowników z wykorzystaniem PCJ i ProActive*

Rysunek 6.11 przedstawia czas potrzebny na policzenie średniej wieku użytkowników w zależności od liczby wątków przy wykorzystaniu biblioteki PCJ i biblioteki ProActive. Test został wykonany na klastrze *hydra* na 1, 2 i 4 węzłach typu *interlagos* zawierających po 64 rdzenie obliczeniowe. Podobnie jak w opisie testów

wydajnościowych, tutaj również w celu uproszczenia opisu uzyskanych wyników, aktywne obiekty w ProActive utożsamiano z wątkami obliczeń.

Na wykresie widać, że dla małej liczby wątków, czyli nie większej niż 8 wątków, ProActive wykonuje się nieznacznie szybciej od PCJ. Dla liczby wątków między 10 a 32, czas obliczeń dla ProActive i PCJ jest zbliżony, natomiast dla liczby wątków większej niż 32 rozwiązanie w ProActive staje się coraz wolniejsze. Dla 256 wątków ProActive jest około sześciokrotnie wolniejszy od PCJ dla tej samej liczby wątków.

W obu przypadkach, przy większej liczbie wątków, czas wykonywania operacji *MapReduce* zaczyna się zwiększać. Jest to spowodowane coraz większym udziałem czasu komunikacji w stosunku do czasu potrzebnego na wykonywanie obliczeń.

Niestety nie udało się wykonać testów dla ProActive w przypadku, gdy próbowano stworzyć większą liczbę aktywnych obiektów niż dostępna liczba rdzeni na węźle, tak jak wykonano testy z wykorzystaniem PCJ, ze względu na rzucanie błędu:

```
org.objectweb.proactive.ActiveObjectCreationException:
   ↪ org.objectweb.proactive.core.ProActiveException:
   ↪ java.lang.reflect.InvocationTargetException
    at org.objectweb.proactive.api
       .PAActiveObject.newActive(PAActiveObject.java:455)
    at org.objectweb.proactive.api
       .PAActiveObject.newActive(PAActiveObject.java:218)
    at org.objectweb.proactive.api
       .PAGroup.newGroup(PAGroup.java:424)
    at org.objectweb.proactive.api
       .PAGroup.newGroup(PAGroup.java:666)
                          ⋮
Caused by: java.lang.OutOfMemoryError: unable to create new
   ↪ native thread
    at java.lang.Thread.start0(Native Method)
    at java.lang.Thread.start(Thread.java:713)
    at org.objectweb.proactive.core.body
       .ActiveBody.startBody(ActiveBody.java:220)
    at org.objectweb.proactive.core.body
       .ActiveBody.<init>(ActiveBody.java:130)
    ... 26 more
```

Mimo iż sama nazwa wyjątku może sugerować, że jest to problem z ilością pamięci, to nie jest to prawda – zadeklarowano możliwość użycia do 128 GB pamięci RAM przez wirtualną maszynę Java, a także zarezerwowania 138 GB pamięci na węzeł w systemie kolejkowym. Wyjątek ten oznacza, że limity na liczbę utworzonych wątków zostały przekroczone – administratorzy systemów klastrowych, z których korzystano, ustalili limit na maksymalną liczbę procesów, z których użytkownik może korzystać na 1024. Niestety w tym przypadku ProActive tworzył zdecydowanie za dużo wątków. Przyczyną

przekroczenia limitu może być problem z kończeniem działania wątków związanych z aktywnymi obiektami nawet po wymuszeniu zakańczania działania aktywnych obiektów na wszystkich aktywnych obiektach grupy:

```
1     Iterator<ActiveObject> it =
2             PAGroup.getGroup(activeObjectsGroup).iterator();
3     while (it.hasNext()) {
4         PAActiveObject.terminateActiveObject(it.next(), true);
5         it.remove();
6     }
```

Inną przyczyną przekroczenia limitu może być wykorzystanie mechanizmów wykonywania zadań natychmiastowo, czyli z adnotacją `@ImmediateService`, co powoduje, że dla każdego żądania tworzony jest osobny wątek.

Podsumowanie

W niniejszej pracy przedstawiono bibliotekę PCJ przeznaczoną do zrównoleglania w modelu PGAS aplikacji napisanych w języku Java. Należy podkreślić, że biblioteka została zaprojektowana i zaimplementowana w ramach niniejszej pracy oraz została udostępniona wraz z dokumentacją (`http://pcj.icm.edu.pl`).

Opisana w niniejszej pracy biblioteka PCJ oferuje nowe możliwości zrównoleglania aplikacji. Biblioteka oferuje metody do pracy w środowisku rozproszonym, synchronizacji węzłów, przesyłania i rozgłaszania (*broadcasting*) wartości zmiennych wykorzystując asynchroniczną jednostronną komunikację. Biblioteka dostarcza również metody służące do monitorowania zmian zmiennych.

Przedstawione w pracy wyniki testów wydajnościowych i testów aplikacji pokazują, że obliczenia równoległe i rozproszone wykonane z użyciem biblioteki PCJ pozwalają na uzyskanie wydajności porównywalnej do uzyskanej przy wykorzystaniu MPI. Szczególnie dobre wyniki wydajnościowe i bardzo dobra skalowalność uzyskiwana jest dla dużych danych, co otwiera możliwość zastosowania biblioteki w obszarze analizy dużych danych (*Big Data*).

Uzyskane wyniki pokazują, iż rozwiązania stworzone w języku Java mogą być równie szybkie, a nawet szybsze od rozwiązań zbudowanych w oparciu o języki niższego poziomu typu C czy C++ z wykorzystaniem tradycyjnych bibliotek takich jak MPI. Jak się okazuje, biblioteka PCJ jest również konkurencyjna w stosunku do nowych rozwiązań pojawiających się w języku Java (*Java 8 Parallel Streams*). Co więcej, widać, że szybkość wykonania może zależeć od sposobu implementacji algorytmu przesyłania danych pomiędzy węzłami. Przedstawione wyniki pokazują także różnice wydajności w zależności od wykorzystanej wirtualnej maszyny Java.

Dodatkowo należy zwrócić uwagę, iż wykorzystane algorytmy powinny zawierać jak najmniej punktów synchronizacji. Ich zbyt duża liczba jest przyczyną zmniejszonej wydajności, a co za tym idzie, gorszej skalowalności aplikacji. W tym kontekście bardzo ważne staje się wykorzystywanie asynchronicznej komunikacji jednostronnej (ang. *one sided communication*) zaimplementowanej w bibliotece PCJ. Pozwala ona na łatwą implementację szerokiej gamy algorytmów oraz pozwala na jednoczesne wykonywanie obliczeń i komunikacji.

Należy podkreślić, że oprócz algorytmów przedstawionych w niniejszej pracy, biblioteka PCJ została wykorzystana do zrównoleglenia mnożenia macierzy [80], przykładowych algorytmów grafowych [81] oraz aplikacji będących częścią *HPC Challenge Benchmark Suite*, których implementacja została zaprezentowana na konferencji *Supercomputing 2014* i uzyskała nagrodę *Most elegant solution* w kategorii *HPC Challenge Class 2* [18].

Uzyskane wyniki testów pokazują, że nadal istnieją obszary, w których działanie biblioteki można ulepszyć, a które nie zostały zrealizowane ze względu na ograniczenia czasowe związane z przygotowywaniem pracy. Analiza wyników wskazuje, że ulepszenia wymaga zwłaszcza komunikacja wewnątrz węzła, realizowana w oparciu o natywne rozwiązania języka Java. Mechanizmy synchronizacji i przekazywania wiadomości między węzłami również mogą być ulepszone. Należy podkreślić, że ich efektywność jest i tak zaskakująca dobra biorąc pod uwagę fakt, że zostały wykorzystane proste mechanizmy oparte na komunikację z wykorzystaniem gniazd TCP/IP.

W aktualnej wersji biblioteki PCJ, nie istnieją zaawansowane techniki wychodzenia ze stanów awarii ani nawet nie istnieje system notyfikacji związany z awarią węzłów. Takie mechanizmy powinny zostać zaimplementowane, aby biblioteka PCJ mogła być szerzej używana do rozwoju aplikacji równoległych i rozproszonych w języku Java. Problem ten jest jednak na tyle skomplikowany, że może stanowić temat dla odrębnej pracy doktorskiej.

Biblioteka PCJ była tworzona z myślą uwzględnienia awarii węzłów, jednak rozwiązanie problemu zostało odłożone na później. Kwestia uwzględnienia częściowych awarii w PCJ jest między innymi tematem 3-letniego projektu HPDCJ (konkurs CHIST ERA UE).

Bazując na kierunkach rozwoju konkurencyjnych rozwiązań a także na własnych doświadczeniach związanych z rozwojem aplikacji równoległych z wykorzystaniem biblioteki PCJ, biblioteka ta mogłaby wzbogacić się miedzy innymi o implementacje asynchronicznej bariery, pełnej synchronizacji i mechanizmów notyfikacji o zakończonych działaniach. Asynchroniczna bariera powinna pozwalać na notyfikowanie o dojściu do miejsca synchronizacji. Pełna synchronizacja, oprócz działania podobnego do operacji bariery, miałaby na uwadze również wszelkie asynchroniczne zadania wysłane przez watek. Mechanizm notyfikacji o zakończonych działaniach pozwoliłby na sprawdzenie czy zakończyły się operacje typu *put* i *broadcast*. Brak takich operacji wynika jedynie z ograniczenia funkcjonalności biblioteki PCJ do operacji niezbędnych do programowania w paradygmacie PGAS i nie ogranicza w żaden sposób możliwości wykorzystania biblioteki PCJ. Zaawansowane mechanizmy synchronizacji czy scenariusze komunikacji mogą być zaimplementowane z wykorzystaniem dostępnych w PCJ metod.

Bibliografia

[1] J2SE™5.0 New Features. `http://docs.oracle.com/javase/1.5.0/docs/relnotes/features.html#concurrency`. Dostęp: 12.03.2013.

[2] Alan Kaminsky. Parallel Java: A Unified API for Shared Memory and Cluster Parallel Programming in 100% Java. Long Beach, redaktor, *21th International Parallel and Distributed Processing Symposium (IPDPS 2007)*, strony 381–388, Long Beach, California, USA, 2007. IEEE International on Cluster Computing.

[3] Java Grande Project: benchmark suite. `http://www.epcc.ed.ac.uk/research/java-grande/`. Dostęp: 12.03.2013.

[4] Titanium Project Home Page. `http://titanium.cs.berkeley.edu`. Dostęp: 12.03.2013.

[5] Java™ Remote Method Invocation API (Java RMI). `http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/index.html`. Dostęp: 12.03.2013.

[6] Christian Nester, Michael Philippsen, Bernhard Haumacher. A more efficient RMI for Java. *Proceedings of the ACM 1999 conference on Java Grande (JAVA '99)*, strony 152–159, New York, NY, USA, 1999. ACM.

[7] Denis Caromel, Christian Delbé, Alexandre Di Costanzo, Mario Leyton, i in. ProActive: an Integrated platform for programming and running applications on grids and P2P systems. *Computational Methods in Science and Technology*, 12(1):69–77, 2006.

[8] MPI-Forum. MPI: A Message-Passing Interface Standard. `http://mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`, Wrzesie/n 21 2012. Dostęp: 1.03.2014.

[9] Bryan Carpenter, Vladimir Getov, Glenn Judd, Anthony Skjellum, Geoffrey C Fox. MPJ: MPI-like Message Passing for Java. *Concurrency: Practice and Experience*, 12(11):1019–1038, 2000.

[10] Mark Baker, Bryan Carpenter, A Shaft. MPJ Express: towards thread safe Java HPC. *Cluster Computing, 2006 IEEE International Conference on*, strony 1–10. IEEE, 2006.

[11] Guillermo L. Taboada, Juan Touriño, Ramón Doallo. F-MPJ: scalable Java message-passing communications on parallel systems. *The Journal of Supercomputing*, 60(1):117–140, 2012.

[12] Wenzhang Zhu, Cho-Li Wang, Francis C.M. Lau. Lightweight Transparent Java Thread Migration for Distributed JVM. *2003 International Conference on Parallel Processing (ICPP'03)*, strony 465–472. IEEE, 2003.

[13] Wenzhang Zhu, Cho-Li Wang, Francis C.M. Lau. JESSICA2: a distributed Java Virtual Machine with transparent thread migration support. *IEEE International Conference on Cluster Computing*, strony 381–388. IEEE, 2002.

[14] Jonas Bonér, Eugene Kuleshov. Clustering the Java virtual machine using aspect-oriented programming. *AOSD '07: Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, 2007.

[15] Marek Nowicki, Piotr Bała. Parallel computations in Java with PCJ library. *High Performance Computing and Simulation (HPCS)*, strony 381–387, 2012.

[16] Marek Nowicki, Piotr Bała. New Approach for Parallel Computations in Java. P. Manninen, redaktor, *PARA 2012: State-of-the-Art in Scientific and Parallel Computing*, wolumen 7782 serii *Lecture Notes in Computer Science*, strony 115–125, 2013.

[17] Concurrency Utilities Overview. `http://docs.oracle.com/javase/7/docs/technotes/guides/concurrency/overview.html`. Dostęp: 13.03.2014.

[18] HPC Challenge Award Competition. `http://www.hpcchallenge.org/custom/index.html?lid=103&slid=272`. Dostęp: 15.01.2015.

[19] Damián A. Mallón, Guillermo L. Taboada, Carlos Teijeiro, Juan Touriño, Basilio B. Fraguela, Andrés Gómez, Ramón Doallo, J. Carlos Mouriño. Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures. M. Ropo, J. Westerholm, J. Dongarra, redaktorzy, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, wolumen 5759 serii *Lecture Notes in Computer Science*, strony 174–184. Springer Berlin / Heidelberg, 2009.

[20] Berkeley Unified Parallel C (UPC) Project Homepage. `http://upc.lbl.gov/`. Dostęp: 12.03.2013.

[21] Co-Array Fortran Homepage. `http://www.co-array.org`. Dostęp: 28.02.2014.

[22] The Chapel Parallel Programming Language Homepage. `http://chapel.cray.com/`. Dostęp: 12.03.2013.

[23] X10 Homepage. `http://x10-lang.org/`. Dostęp: 12.03.2013.

[24] Project Fortress Homepage. http://projectfortress.java.net/. Dostęp: 12.03.2013.

[25] Intel®Cilk™ Plus Homepage. http://software.intel.com/en-us/articles/intel-cilk-plus/. Dostęp: 12.03.2013.

[26] Threading Building Blocks Homepage. http://threadingbuildingblocks.org. Dostęp: 12.03.2013.

[27] Rich Cook, Evi Dube, Ian Lee, Lee Nau, Charles Shereda, Felix Wang. Survey of Novel Programming Models for Parallelizing Applications at Exascale. Raport instytutowy LLNL-TR-515971, Lawrence Livermore National Laboratory, 2011.

[28] Leonardo Dagum, Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[29] OpenMP Application Program Interface. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, Lipiec 2013. Dostęp: 1.03.2014.

[30] Mark Baker, Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, Sang Lim. mpiJava: An object-oriented Java interface to MPI. *Parallel and Distributed Processing*, strony 748–762. Springer, 1999.

[31] Tim Stitt. *An introduction to the Partitioned Global Address Space (PGAS) programming model*. Connexions, Rice University, 2009.

[32] Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, Olivier Tardieu. The asynchronous partitioned global address space model. *Proceedings of The First Workshop on Advances in Message Passing*, 2010.

[33] GNU Unified Parallel C (GNU UPC) Homepage. http://www.gccupc.org. Dostęp: 12.03.2013.

[34] Dan Bonachea. Gasnet Specification, version 1.1. Raport instytutowy UCB/CSD-02-1207, U.C. Berkeley Tech Report, 2002.

[35] William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, Karen Warren. *Introduction to UPC and language specification*. Center for Computing Sciences, Institute for Defense Analyses, 1999.

[36] Tarek El-Ghazawi, François Cantonnet, Yiyi Yao, Ram Rajamony. Developing an optimized upc compiler for future architectures. Raport instytutowy, Technical report, IDA Center for Computing Sciences, 2005.

[37] Bradford L. Chamberlain, David Callahan, Hans P Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[38] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, Daniel Chavarría-Miranda. An evaluation of global address space languages: Co-Array Fortran and Unified Parallel C. *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, strony 36–47. ACM, 2005.

[39] Damian Alvarez Mallón. *Design of scalable PGAS collectives for NUMA and manycore systems*. Praca doktorska, Universidade da Coruña, 2014.

[40] Robert W. Numrich, John Reid. Co-Array Fortran for parallel programming. *ACM SIGPLAN Fortran Forum*, wolumen 17, strony 1–31. ACM, 1998.

[41] Robert W. Numrich, John Reid. Co-arrays in the next Fortran Standard. *ACM SIGPLAN Fortran Forum*, wolumen 24, strony 4–17. ACM, 2005.

[42] Piotr Bała, Terry Clark, L. Ridgway Scott. Application of Pfortran and Co-Array Fortran in the parallelization of the GROMOS96 molecular dynamics module. *Scientific Programming*, 9(1):61–68, 2001.

[43] Javier Diaz, Camelia Munoz-Caro, Alfonso Nino. A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1369–1386, 2012.

[44] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phillip Colella, i in. Titanium: A high-performance Java dialect. *Concurrency Practice and Experience*, 10(11-13):825–836, 1998.

[45] Paul N. Hilfinger, Dan Oscar Bonachea, Kaushik Datta, David Gay, Susan L. Graham, Benjamin Robert Liblit, Geoffrey Pike, Jimmy Zhigang Su, Katherine A Yelick. Titanium Language Reference Manual, version 2.19. Raport instytutowy, UC Berkeley Tech Rep. UCB/EECS-2005-15, 2005.

[46] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Acm SIGPLAN Notices*, 40(10):519–538, 2005.

[47] The Fortress Language Specification, Version 1.0. `http://www.ccs.neu.edu/home/samth/fortress-spec.pdf`. Dostęp: 12.03.2014.

[48] Fortress Wrapping Up. `https://blogs.oracle.com/projectfortress/entry/fortress_wrapping_up`. Dostęp: 12.03.2014.

[49] James Gosling, Henry McGilton. *The Java language environment*, wolumen 2550. Sun Microsystems Computer Company, 1995.

[50] C. Enrique Ortiz, Eric Giguère. *Mobile information device profile for Java 2 MicroEdition: professional developer's guide*, wolumen 15. John Wiley & Sons, 2001.

[51] Aidan Fries. *The use of Java in large scientific applications in HPC environments*. Praca doktorska, Universitat de Barcelona, 2013.

[52] WJ Cody, i in. *IEEE standards 754 and 854 for Floating-Point Arithmetic*, 1984.

[53] *IEEE Standard for Floating-Point Arithmetic*, 2008.

[54] James Gosling, Bill Joy, Guy L. Steele Jr, Gilad Bracha, Alex Buckley. *The Java Language Specification*. Addison-Wesley, 2013.

[55] Brian Göetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea. *Java concurrency in practice*. Addison-Wesley, 2006.

[56] Java SE 6 Performance White Paper. `http://www.oracle.com/technetwork/java/6-performance-137236.html`. Dostęp: 7.03.2014.

[57] Oracle Completes Acquisition of Sun. `http://www.oracle.com/us/corporate/press/044428`. Dostęp: 8.03.2014.

[58] Concurrency Utilities Enhancements in Java SE 7. `http://docs.oracle.com/javase/7/docs/technotes/guides/concurrency/changes7.html`. Dostęp: 7.03.2014.

[59] Tomasz Nurkiewicz. Java 8 – najbardziej rewolucyjna wersja w historii. *Programista: magazyn programistów i liderów zespołów IT*, 21(2), 2014. ISSN: 2084:9400.

[60] Guillermo López Taboada. *Design of Efficient Java Communications for High Performance Computing*. Praca doktorska, University of A Coruna, Spain, 2009.

[61] Rafał Metkowski, Piotr Bała. Parallel computing in Java: looking for the most effective RMI implementation for clusters. *Parallel Processing and Applied Mathematics*, strony 272–277. Springer, 2006.

[62] ProActive Programming library – programming in OW2 ProActive - Gitorious. `http://gitorious.ow2.org/ow2-proactive/programming`. Dostęp: 3.01.2014.

[63] Łukasz Mikulski. Rozwój równoległych algorytmów dynamiki molekularnej. Praca magisterska, Uniwersytet Mikołaja Kopernika w Toruniu, 2006.

[64] Jarosław Mederski. Asynchroniczny algorytm dynamiki molekularnej i jego obiektowa implementacja. Praca magisterska, Uniwersytet Mikołaja Kopernika w Toruniu, 2006.

[65] Przemysław Fusik. Aplikacje rozproszone w środowisku ProActive. Praca magisterska, Uniwersytet Mikołaja Kopernika w Toruniu, 2010.

[66] Douglas Kramer. The Java Platform. *White Paper, Sun Microsystems, Mountain View, CA*, 1996.

[67] Georgel Calin, Egor Derevenetc, Rupak Majumdar, Roland Meyer. A theory of Partitioned Global Address Spaces. *arXiv preprint arXiv:1307.6590*, 2013.

[68] Ami Marowka. Execution model of three parallel languages: OpenMP, UPC and CAF. *Scientific Programming*, 13(2):127–135, 2005.

[69] JEP 143: Improve Contended Locking. `http://openjdk.java.net/jeps/143`. Dostęp: 15.01.2015.

[70] Piotr Luszczek, Jack J Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John McCalpin, David Bailey, Daisuke Takahashi. Introduction to the hpc challenge benchmark suite. *Lawrence Berkeley National Laboratory*, 2005.

[71] Piotr Luszczek, Jack Dongarra, Jeremy Kepner. Design and implementation of the hpc challenge benchmark suite. *CT Watch Quarterly*, 2(4A), 2006.

[72] Kumar, Vijay P. and Gupta, Anshul. Analyzing scalability of parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, 22(3):379–391, 1994.

[73] Parallel Scalability – Osman Sarood. `https://courses.engr.illinois.edu/cs420/fa2012/ParallelScalability.pptx`. Dostęp: 15.01.2015.

[74] Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000.

[75] Ken Kennedy, Charles Koelbel, Robert Schreiber. Defining and measuring the productivity of programming languages. *International Journal of High Performance Computing Applications*, 18(4):441–448, 2004.

[76] Halo2 – Centrum Nauk Obliczeniowych, ICM Uniwersytet Warszawski. `http://www.icm.edu.pl/kdm/Halo2`. Dostęp: 9.12.2013.

[77] Boreasz – Centrum Nauk Obliczeniowych, ICM Uniwersytet Warszawski. `http://www.icm.edu.pl/kdm/Boreasz`. Dostęp: 9.12.2013.

[78] Lakshminarayana B. Arimilli, Ravi K. Arimilli, Robert S. Blackmore, Chulho Kim, Ramakrishnan Rajamony, William J. Starke, Hanhong Xue. Host Fabric Interface (HFI) to Perform Global Shared Memory (GSM) Operations, Sierpie/n 6 2009. US Patent App. 12/024,397.

[79] Hydra – Centrum Nauk Obliczeniowych, ICM Uniwersytet Warszawski. `http://www.icm.edu.pl/kdm/Hydra`. Dostęp: 9.12.2013.

[80] Marek Nowicki, Łukasz Górski, Patryk Grabarczyk, Piotr Bała. PCJ – Java library for high performance computing in PGAS model. *High Performance Computing and Simulation (HPCS)*, strony 202–209. IEEE, 2014.

[81] Magdalena Ryczkowska, Piotr Bała. Evaluating PCJ Library for Graph Problems – Graph500 in PCJ. *High Performance Computing and Simulation (HPCS)*, strony 1005–1007. IEEE, 2014.