

**Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie**

Katedra Informatyki



PRACA MAGISTERSKA

PIOTR KOZŁOWSKI

**PRZENOŚNY KLASTER OBliczeniowy oparty na
urządzeniach SoC**

PROMOTOR:
dr hab. inż. Aleksander Byrski

Kraków 2015

OŚWIADCZENIE AUTORA PRACY

OŚWIADCZAM, ŚWIADOMY ODPOWIEDZIALNOŚCI KARNEJ ZA POŚWIADCZENIE NIEPRAWDY, ŻE NINIEJSZĄ PRACĘ DYPLOMOWĄ WYKONAŁEM OSOBIŚCIE I SAMODZIELNIE, I ŻE NIE KORZYSTAŁEM ZE ŹRÓDEŁ INNYCH NIŻ WYMIESZCZONE W PRACY.

.....

PODPIS

AGH
University of Science and Technology in Krakow

Department of Computer Science



MASTER OF SCIENCE THESIS

PIOTR KOZŁOWSKI

MOBILE COMPUTING CLUSTER BASED ON SoC DEVICES

SUPERVISOR:
Aleksander Byrski Ph.D

Krakow 2015

Dziękuję mojemu promotorowi
za podjęcie się prowadzenia niniejszej pracy
oraz wszystkie wskazówki,
Karolinie za wsparcie oraz cierpliwość,
a także Rafałowi za cenne uwagi.

Spis treści

1.	Wstęp.....	6
2.	Obliczenia równoległe oraz rozproszone	8
2.1.	Model architektur równoległych Flynn'a.....	8
2.2.	Modele programowania równoległego	10
2.3.	Istniejące rozwiązania.....	13
2.4.	Scala i Akka	15
3.	Niekonwencjonalne metody realizacji obliczeń.....	18
3.1.	Architektura System on a Chip.....	18
3.2.	Internet of Things.....	21
3.3.	Ubiquitous Computing.....	22
4.	Mobilny kластer obliczeniowy	23
4.1.	Koncepcja	23
4.2.	Platforma	24
4.3.	Inne wykorzystane technologie	30
5.	Możliwości praktycznego zastosowania.....	32
5.1.	Równoległe algorytmy ewolucyjne	32
5.2.	Problemy benchmarkowe.....	34
5.3.	Ewolucyjna optymalizacja na platformie	34
5.4.	Wyniki testów	36
6.	Podsumowanie	51

1. Wstęp

Motywacja

Obliczenia równoległe a także systemy rozproszone od lat są wykorzystywane przez różne ośrodki badawcze, jak również przemysł do obliczeń wielkiej skali, a także wspomagania systemów wysokiej dostępności. Innymi obszarami zastosowań są różnego rodzaju symulacje, np. w biologii, chemii, astrofizyce lub medycynie, jak również przyśpieszenie krytycznych obliczeń w takich dziedzinach jak systemy obronne, przemysł komunikacyjny, meteorologia, astrodynamika, ponadto rozwiązywanie problemów optymalizacyjnych i przetwarzanie dużej ilości danych [79, 69].

Przyczyniło się to do silnego rozwoju poruszanej tematyki, lecz niektóre koncepcje opracowane już w połowie XX wieku kiedy trudno było przewidzieć skalę z jaką wzrasta ilość nowych urządzeń oraz danych do przetwarzania w dzisiejszych czasach napotykają pewne bariery. Cały czas kreowane są nowe idee w myśl rozwoju współczesnych technologii co sprawia, że tematyka niniejszej pracy nie traci na aktualności. Takie koncepcje jak BigData lub IoT motywują do rozwoju prac związanych z wydajnym przetwarzaniem dużej ilości danych [33, 63, 15].

Potrzeby rynku oraz zwiększcza podaż spowodowały spadek cen urządzeń mobilnych, a także obniżenie stosunku ceny do mocy obliczeniowej. Urządzenia SoC zaczęto wykorzystywać na większą skalę w edukacji jak i w domowych zastosowaniach [33, 17, 63]. Pomimo tego, że proces miniaturyzacji oraz zwiększania wydajności procesorów osiąga bariery technologiczne poprzez rozwój systemów wieloprocesorowych udaje się nadal zwiększać efektywność obliczeń [76]. Powstają aplikacje i technologie mające na celu połączenie zasobów w jedną całość oraz efektywne ich wykorzystanie. Zasoby te mogą być zgromadzone lokalnie lub rozproszone w różnych rejonach świata i połączone w sieci Internet. Natomiast dzięki elastycznym formom handlu, jak na przykład sprzedaż zużytej mocy obliczeniowej w Amazon AWS oraz popularyzacji rozwiązań chmur obliczeniowych udaje się w coraz łatwiejszy sposób docierać do klienta indywidualnego oraz mniejszych firm [11]. Możliwość użycia tego typu rozwiązań przez zwykłych użytkowników ma wpływ na tworzenie społeczności wspierających oraz testujących technologie o otwartym kodzie źródłowym.

Cel pracy

Celem niniejszej pracy jest przygotowanie rozwiązania potrafiącego koordynować pracę klastra obliczeniowego złożonego z urządzeń SoC działających w sieci lokalnej.

Do celów szczegółowych należy rozpoznanie tematyki obliczeń równoległych w środowisku rozproszonym, zbadanie oraz wybór odpowiednich technologii dzięki którym możliwe będzie zaprojektowanie architektury oraz implementacja platformy cechującej się mobilnością, skalowalnością, a także zapewniającej wsparcie w sytuacji awarii urządzeń pracujących w klastrze. Głównym zadaniem powstałej platformy jest zarządzanie obliczeniami prowadzonymi w klastrze oraz load-balancing. Ponadto jest ona również odpowiedzialna za komunikację urządzeń oraz monitoring stanu klastra czyli informowanie użytkownika jeżeli którykolwiek urządzenie opuści lub dołączy do klastra. Wynika z tego, że oprócz części platformy odpowiedzialnej za obliczenia i zarządzanie zadaniami oraz stanem klastra, wymagany jest interfejs użytkownika, dzięki któremu jest on w stanie zlecać zadania oraz mieć podgląd na stan działającego klastra. Dobór technologii oraz rozwiązań bierze pod uwagę ograniczone

możliwości obliczeniowe typowych urządzeń SoC oraz mobilność powstałej platformy, dlatego faworyzowane są multiplatformowe technologie wysokiego poziomu, lecz na tyle wydajne, aby możliwe było osiągnięcie zadowalającej efektywności obliczeń biorąc pod uwagę różne ograniczenia.

Ostatnim celem jest przetestowanie opracowanego rozwiązania poprzez implementację jednego z modeli programowania równoległego wykorzystanego do prowadzenia wymagających obliczeń matematycznych. Zbadana została skalowalność, wydajność oraz niezawodność. Zidentyfikowano również możliwości zastosowań, jak również perspektywy dalszego rozwoju powstałej platformy.

Opis rozdziałów

Na początku pracy zawarto motywacje oraz opis jej celów. Następnie zamieszczono wprowadzenie do tematyki obliczeń równoległych wraz z podziałem sprzętowych architektur oraz opisem kilku popularnych modeli programowania jak również istniejących implementacji wykorzystujących wymienione architektury oraz modele.

W kolejnym rozdziale opisano kilka popularnych koncepcji takich jak urządzenia SoC, Internet of Things oraz Ubiquitous Computing. Zawarto w nim również porównanie paru istniejących urządzeń SoC, przytoczono niektóre przykłady zastosowania wymienionych idei oraz pokazano skalę tych zjawisk powołując się na różne źródła.

Rozdział numer cztery został w całości poświęcony opisowi implementacji platformy. Zarysuje on koncepcję oraz problemy, którym stawiono czoła. Następnie przedstawia szczegóły implementacyjne zawierające m.in. schemat architektury czy protokołów komunikacji. Na końcu zawarto opis kilku innych wykorzystanych technologii.

Kolejny rozdział zawiera opis możliwości zastosowania przygotowanego rozwiązania wraz z przybliżeniem tematyki algorytmów ewolucyjnych w kontekście obliczeń równoległych. Następnie opisano użycie wspomnianych algorytmów do rozwiązania problemu testowego jakim jest optymalizacja funkcji wielomodalnej. Na końcu rozdziału zawarto opracowanie wyników testów przedstawiające uzyskaną skalowalność oraz wpływ na wydajność obliczeń różnych parametrów początkowych zadania.

Ostatni rozdział podsumowuje niniejszą pracę czyli przedstawia wyciągnięte wnioski oraz wskazanie kierunków dalszego rozwoju przygotowanej platformy.

2. Obliczenia równoległe oraz rozproszone

Dzięki rozwojowi systemów wieloprocesorowych oraz wielu technologii wspierających programowanie równoległe, jak na przykład języki funkcyjne takie jak Scala, które u podstaw swych założeń mają na celu dobrą skalowalność we współczesnych architekturach, obliczenia równoległe stały się tematem bardzo interesującym. Rozdział ten zawiera klasyfikację modelu architektur równoległych, a także przegląd kilku wybranych technologii mających zastosowanie przy obliczeniach równoległych oraz rozproszonych.

2.1. Model architektur równoległych Flynn'a

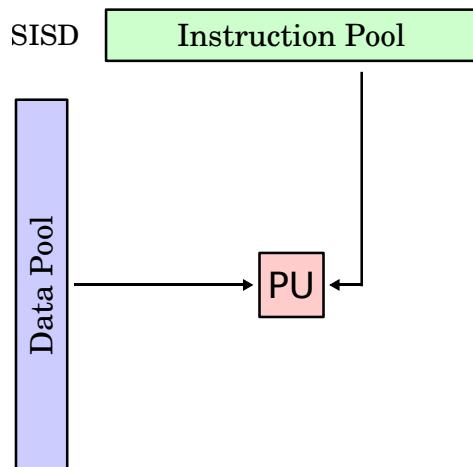
W latach 60-tych Michael Flynn sklasyfikował architektury komputerowe w czterech kategoriach biorąc pod uwagę ilość instrukcji oraz ilość strumieni danych przetwarzanych przez procesory. Poniżej opisano je w skrócie oraz pokazano schematy poglądowe.

SISD (ang. Single Instruction, Single Data)

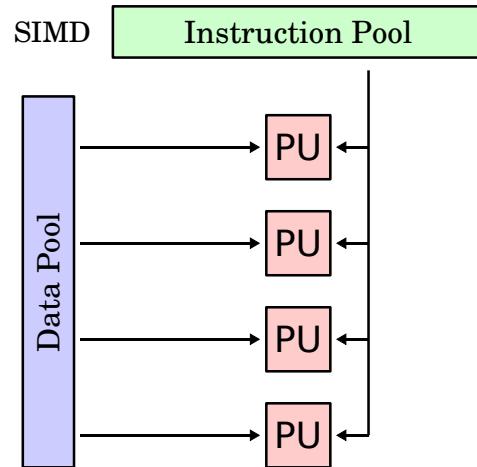
Jeden strumień danych jest przetwarzany sekwencyjnie przez jedną instrukcję w każdym kolejnym cyklu procesora (rysunek 2.1). Wyniki wykonania instrukcji są deterministyczne. Wykorzystany w komputerach skalarnych oraz typu mainframe. Przykład klasycznej architektury von Neumanna [13].

SIMD (ang. Single Instruction, Multiple Data)

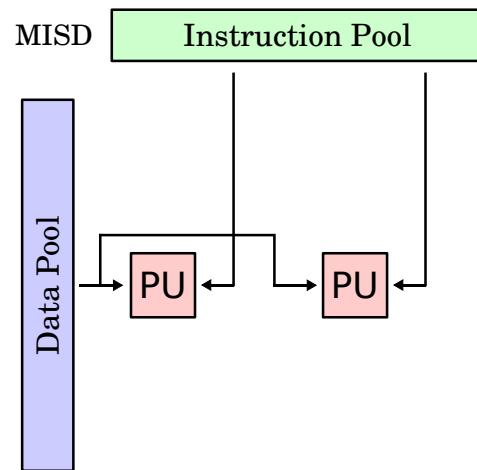
Ta sama instrukcja jest wykonywana równolegle na wielu procesorach używających oddzielnych strumieni danych (rysunek 2.2). Każdy procesor posiada swoją pamięć dla danych. Uzyskiwane wyniki są deterministyczne a instrukcje wykonywane są sekwencyjnie (ang. lockstep). Model ten dobrze pasuje do danych, które cechują się pewną regularnością, co sprawdza się np. przy przetwarzaniu obrazów. Jako



Rysunek 2.1: Schemat SISD [55].



Rysunek 2.2: Schemat SIMD [55].



Rysunek 2.3: Schemat MISD [55].

Przykład wykorzystania tej architektury można podać komputery wektorowe lub współczesne procesory GPU wykorzystywane m.in. w kartach graficznych.

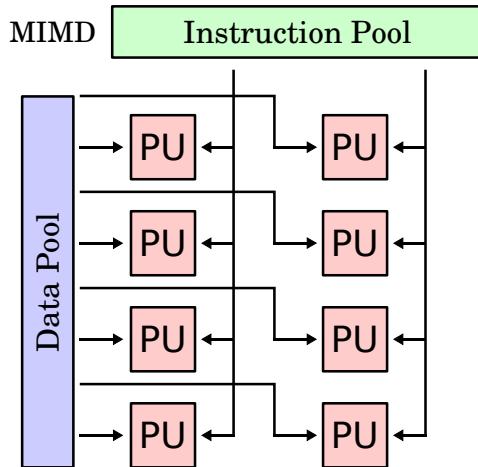
MISD (ang. Multiple Instruction, Single Data)

Wiele niezależnych instrukcji przetwarza równolegle ten sam strumień danych (rysunek 2.3). Model mający słabe zastosowanie komercyjne, mógłby zostać wykorzystany w przypadku potrzeby redundancji obliczeń, czyli wielokrotnego wykonywania tych samych obliczeń w celu minimalizacji błędów.

MIMD (ang. Multiple Instruction, Multiple Data)

Wiele procesorów wykonuje równolegle instrukcje przetwarzające niezależne strumienie danych (rysunek 2.4). Wykonanie może być synchroniczne jak i niesynchroniczne, jak również deterministyczne lub nie. Podejście bardziej elastyczne niż SIMD a co za tym idzie mające szersze zastosowanie. Jako przykład można tutaj podać systemy wieloprocesorowe, klastry lub gridy [74]. Większość współczesnych komputerów korzysta z tej architektury. Ta architektura może być stworzona z wielu komponentów SIMD.

Wyżej wymieniona klasyfikacja jest jedną z najstarszych oraz najbardziej ogólnych, będącą zazwyczaj punktem wyjścia do innych, bardziej szczegółowych lub biorących pod uwagę inne aspekty zrównoleglania obliczeń. W klasycznej pozycji książkowej [74] rozważono ponadto równoległość



Rysunek 2.4: Schemat MIMD [55].

na różnych poziomach abstrakcji, takich jak instrukcje (ang. Instruction-Level Parallelism), dane (ang. Data-Level Parallelism) oraz wątki (ang. Thread-Level Parallelism).

2.2. Modele programowania równoległego

Poniższy rozdział zawiera wstęp do kilku modeli programowania równoległego opartych na architekturze MIMD. Modele te istnieją jako abstrakcja od fizycznej architektury sprzętowej, co oznacza, że w niektórych przypadkach mogą być zaimplementowane na różnych architekturach sprzętowych.

SPMD (ang. Single Program, Multiple Data)

W tym modelu wiele procesorów wykonuje równolegle ten sam program przetwarzając różne struktury danych. Jest to najbardziej powszechny model programowania równoległego, który może być wykonywany na większości współczesnych procesorach powszechnego użytku. Zazwyczaj model ten nie wymaga synchronizacji tak jak w SIMD, gdyż procesory wykonują różne niezależne części programu i nie muszą się one wykonywać w tzw. lockstepie [35] tak jak w przypadku SIMD.

MPMD (ang. Multiple Program, Multiple Data)

Kolejny model programowania równoległego, który może mieć szersze zastosowanie aniżeli SPMD. Wiele niezależnych procesorów wykonuje równolegle co najmniej dwa niezależne programy.

Model SPMD implikuje kolejną klasyfikację ze względu na dostęp do pamięci biorąc pod uwagę pamięć współdzieloną oraz rozproszoną czyli Shared Memory oraz Distributed Memory.

Shared Memory (SM)

Procesy współdzierają jakiś obszar pamięci przechowujący dane, które odczytują lub zapisują asynchronicznie. Nie musi tutaj występować komunikacja pomiędzy procesami natomiast potrzebne są jakieś mechanizmy kontroli dostępu do danych, takie jak synchronizacja, semafory, blokady (ang. lock) lub bariery pamięci zapewniające poprawność wykonywania operacji przy zachowaniu spójności danych. W związku z wymienionymi problemami programy korzystające z pamięci współdzielonej mogą być niedeterministyczne.

Distributed Memory (DM)

Każdy proces posiada swoją własną przestrzeń pamięci, co oznacza, że nie ma bezpośredniego dostępu do pamięci innych procesów. Co za tym idzie wymagana jest komunikacja pomiędzy procedurami w celu wymiany informacji przez co mogą wystąpić jakieś opóźnienia w obliczeniach związane z narzutem komunikacji. Obszary pamięci nie muszą być podzielone sprzętowo, co oznacza, że ten model programowania może być zaimplementowany na sprzecie posiadającym pamięć współdzieloną podzieloną logicznie pomiędzy procesy. Problemami, które mogą wystąpić w opisywanym modelu jest load-balancing, podzielenie zadania na odpowiednie procesy, jak również łączenie wyników zadań po zakończeniu obliczeń.

Poniżej wymieniono kilka innych modeli programowania równoległego z uwzględnieniem tego czy korzystają one z pamięci współdzielonej lub nie.

Data Parallel

Jest jedna zorganizowana globalna przestrzeń danych, np. w postaci tablicy na której operują procesy wykonujące się równolegle. Procesy wykonują operacje na swojej wydzielonej części pamięci. Operacja może polegać na przykład na dodaniu konkretnej liczby do kilku elementów tablicy. Jeżeli tablica posiada sto elementów, a dziesięć różnych procesów wykonuje na niej jakieś operację, to każdy proces może mieć dostęp do dziesięciu elementów tablicy, co oznacza, że żaden proces nie będzie operował na tym samym elemencie tablicy. Wykorzystuje model SM.

Message Passing

W tym modelu procesy, zwane również zadaniami (ang. tasks) posiadają dane na których operują. Każdy zapisanie posiada unikalny identyfikator i komunikuje się z innymi zapisaniami przesyłając im jakieś informacje. Komunikacja musi być zazwyczaj skoordynowana, co oznacza, że po operacji wysłania musi nastąpić odbiór wiadomości w innym zapisaniu. Zapisania mogą wykonywać się równolegle, mogą być również tworzone dynamicznie. Każde zapisanie może wykonywać się na oddzielnym procesorze, może być również umieszczone na innej fizycznej maszynie. Implementacją tego modelu jest standard MPI, który zostanie opisany później. Model ten wykorzystuje DM.

Threads

W tym modelu jeden złożony proces może posiadać kilka wykonywanych wspólnie ścieżek, tzw. wątków. Dla przykładu założymy, że procesem jest program, który po starcie uruchamia kilka zadań wykonywanych równolegle przez różne wątki. Wykonywaniem równoległych wątków zajmuje się system operacyjny na którym ten program jest uruchomiony. Każdy wątek może posiadać swoje lokalne dane, które są prywatne i niewidoczne dla innych wątków ale może również współdzielić dane zaalokowane przez główny program z innymi wątkami, tzw. dane globalne. W przypadku gdy wątki potrzebują się ze sobą komunikować, mogą to zrobić właśnie przez dane globalne widziane z innych wątków. W takim przypadku jeśli chcemy uniknąć sytuacji, że dwa wątki próbują modyfikować ten sam obszar pamięci w tym samym czasie musi nastąpić synchronizacja dostępu do niego. Wątki mogą być tworzone i usuwane dynamicznie przez główny program i dzięki niemu mogą również współdzielić zasoby. Wykorzystuje model SM. Problemy, które mogą wystąpić w przypadku tego modelu, to zablokowanie wątków (ang. deadlock), zagłodzenie wątków (ang. starvation) oraz tzw. race condition [79, 66, 74].

W modelu Shared Memory jeżeli procesy operują na wspólnych danych utrudnieniem jest kontrola ich dostępu oraz zapewnienie zadowalającej wydajności obliczeń. Różne technologie oraz modele programowania równoległego zapewniają rozwiązań, które czasem mogą okazać się niewystarczające lub źle wpływać na proces zrównoleglania. W przypadku gdy współdzielony stan (pamięć) jest mutowalny, czyli może być modyfikowany przez wiele procesów, rozwiązaniem powinna być synchronizacja dostępu. Jeżeli jednak każdy proces posiada odizolowany stan, który nie może być mutowalny, to syn-

chronizacja jest niepotrzebna lecz tracimy również możliwość współdzielenia stanu. Jednym z rozwiązań tych problemów jest wykorzystanie podejścia z języków funkcyjnych takich jak Scala lub Erlang, czyli niemutowalnych obiektów. Modelem wykorzystującym takie obiekty do przesyłania komunikatów jest model aktorowy wykorzystany m.in. we wspomnianych wyżej językach funkcyjnych oraz w wielu innych do zrównoleglania obliczeń.

Model Aktorowy (ang. Actor Model)

W modelu aktorowym głównym bytem są aktorzy. Komunikują się oni ze sobą asynchronicznie poprzez wymianę wiadomości a więc są trochę podobni do zadań w modelu Message Passing. Po wysłaniu wiadomości nie muszą czekać na odpowiedź a kolejność dostarczania wiadomości jest nieistotna. Wiadomości są buforowane w skrzynkach odbiorczych. Każdy aktor przechowuje jakiś stan, który nie powinien być widoczny dla innych aktorów i może zostać zmieniony jedynie przez tego samego aktora. Model aktorowy wykorzystuje Shared Memory. Zaimplementowano go m.in. w Erlangu lub Scali. Zaletą tego modelu jest to, że nie trzeba się martwić o współdzielenie skomplikowanych stanów lecz z drugiej strony problem nie zawsze da się podzielić na mniejsze części i zaimplementować z wykorzystaniem aktorów, co można traktować jako wadę tego podejścia [2].

STM (ang. Software Transactional Memory)

Wprowadza pojęcie atomicznych transakcji. Każdy dostęp do pamięci współdzielonej objęty jest transakcją, której idea jest podobna do transakcji bazodanowych. Jest to alternatywa do blokowania dostępu do pamięci poprzez synchronizację. Transakcja obejmuje serię odczytów i zapisów do pamięci współdzielonej a jej stan nie jest widoczny dla innych transakcji dopóki nie zostanie ona zakończona i zatwierdzona (ang. commit). Wyróżnia się tutaj sekcje krytyczne zwane blokami atomicznymi. Jeżeli wykonanie takiego bloku się nie powiedzie zmiany są wycofywane, co wymusza przechowywanie starszej wersji danych. Transakcje mogą wykonywać się równolegle. Model ten zaimplementowano w wielu popularnych językach programowania, m.in. w języku Scala [75, 9].

W literaturze można znaleźć jeszcze wiele innych modeli programowania równoległego wykorzystujących podobne koncepcje jak te opisane powyżej, jak również wprowadzające zupełnie inne podejścia. Ostatnim modelem, który zostanie opisany jest Master/Slave ze względu na szersze wykorzystanie go w niniejszej pracy.

Master/Slave

Master/Slave może być implementacją modelu SPMD. Polega on na tym, że jedno urządzenie lub proces zwane *master* kontroluje jedno lub więcej urządzeń lub procesów zwanych *slave* lub *worker* [36]. Obliczenia wykonywane przez *workers* odbywają się równolegle, natomiast *master* odpowiada za load-balancing oraz łączenie wyników w jedną całość.

Może okazać się przydatny w następujących przypadkach:

- gdy chcemy mieć jeden proces, tzw. single-point of responsibility, który podejmuje decyzje lub koordynuje akcje w celu zachowania spójności w systemie,
- potrzebujemy zapewnić jeden punkt dostępu do zewnętrznego systemu lub jeden punkt wejścia do systemu z zewnątrz,
- istnieje potrzeba stworzenia centralizowanego serwisu, np. zajmującego się routingu,
- istnieje możliwość podzielenia zadania na kilka części, z których każda może być wykonywana równolegle.

Rozwiązanie to ma niestety również kilka wad, jak na przykład istnieje niebezpieczeństwo utworzenia tzw. wąskiego gardła co może spowodować problemy z wydajnością lub być tzw. single-point of failure.

Jeśli urządzenie/proces posiadające rolę *master* ulega awarii, powinniśmy w jakiś sposób obsłużyć taką sytuację np. poprzez ponowne wystartowanie procesu/urządzenia z rolą Master, aby zapewnić poprawne i niezawodne działanie komunikacji. Sytuacje takie mogą wprowadzić pewne opóźnienia w trakcie odzyskiwania sprawności systemu [25].

2.3. Istniejące rozwiązania

Dla większości modeli programowania równoległego wymienionych w poprzednim rozdziale można znaleźć istniejące implementacje, niektóre z nich są komercyjne ale zdarzają się także takie o otwartym kodzie źródłowym. Poniżej zamieszczono krótką charakterystykę wybranych technologii.

OpenMP (ang. Open Multi-Processing)

Standard używany na skalę przemysłową, korzystający z modelu Threads i pamięci współdzielonej. Dostępne są implementacje w językach C/C++ oraz Fortran. Pozwala na wielowątkowe obliczenia w których wiele wątków pracuje na tym samym zestawie danych. Pierwsza wersja standardu została opublikowana w roku 1997 dla języka Fortran [40].

MPI (ang. Message Passing Interface)

Popularny standard szeroko stosowany w obliczeniach równoległych, jak również w przemyśle, powstał w roku 1994. Korzysta z modelu Message Passing i Distributed Memory. Najbardziej znane implementacje to OpenMPI i MPICH dla języków C oraz Fortran. Zapewnia komunikację przez sieć dla danych przetwarzanych na różnych fizycznych maszynach [37].

CUDA (ang. Compute Unified Device Architecture)

Jest to technologia opracowana w 2007 roku przez firmę NVIDIA, producenta kart graficznych. Wykorzystuje układy GPU a co za tym idzie architekturę SIMD. Pozwala wykonywać obliczenia równolegle. Dostarcza SDK oraz API pozwalające na pisanie aplikacji z wykorzystaniem języka C/C++ lub Fortran. Ma zastosowanie w przetwarzaniu video, biologii obliczeniowej i chemii, astrofizyce oraz różnego rodzaju symulacjach lub analizach [38].

OpenCL (ang. Open Computing Language)

Technologia o otwartym kodzie źródłowym w przeciwieństwie do CUDA, rozwijana przez grupę Khronos od 2009 roku. Wykorzystująca do obliczeń również procesory graficzne. Dodatkowo wspierająca obliczenia równolegle na zwykłych procesorach CPU. Podczas gdy CUDA wykorzystuje karty graficzne firmy NVIDIA, OpenCL wspiera również platformy AMD, Intel, ARM oraz wiele innych. Technologia jest oparta o język C++ [39].

Java Concurrency Framework

Większość współczesnych obiektowych języków programowania wspiera pojęcie współbieżności poprzez wątki, które operują na pamięci współdzielonej. Nie inaczej jest z Javą, w której od wersji 5 wprowadzono wiele usprawnień do wsparcia wielowątkowości m.in. relację happens-before, która wymusza, że jeden dostęp do pamięci musi nastąpić przed innymi. Poza wieloma mechanizmami wspomagającymi różne poziomy synchronizacji wprowadzono również wysokopoziomowe API, wiele kolekcji oraz typów danych z wbudowanymi mechanizmami wspierającymi współbieżność i ułatwiającymi programistom implementowanie obliczeń równoległych [34].

Poza frameworkami wymienionymi powyżej, których użycie oraz przygotowanie końcowego rozwiązania często wymaga dużo czasu, powstało również kilka gotowych systemów oferujących

sprawdzone implementacje niektórych problemów i będące punktem wyjścia dla bardziej skomplikowanych rozwiązań.

Apache Hadoop

Jest to framework open-source napisany w Javie i wspierający obliczenia równoległe prowadzone na dużych ilościach danych w środowisku rozproszonym, takim jak klastry komputerowe. Pierwsza wersja została opublikowana w 2005 roku. Jego dwoma głównymi komponentami są HDFS (Hadoop Distributed File System) wykorzystujący technologię HBase i służący do przechowywania danych oraz MapReduce służący do przetwarzania danych. Kolejnym istotnym komponentem jest YARN pomagający zarządzać zasobami dostępnymi w klastrze oraz wykorzystywany do harmonogramowania zadań. Zamierzeniem twórców było stworzenie narzędzia dobrze skalowalnego oraz zapewniającego wysoką dostępność poprzez wykrywanie stanu urządzeń pracujących w klastrze [12].

HTCondor

Jest systemem służącym do zarządzania zasobami oraz kolejkowaniem zadań w architekturach rozproszonych. Jest to projekt Open-source rozwijany przez Uniwersytet Wisconsin-Madison. Powstał w latach 80-tych. Może być uruchamiany na wielu platformach takich jak Windows czy Linux. Dostarcza mechanizmy zabezpieczeń takie jak autentykacja, autoryzacja i szyfrowanie danych, jak również mechanizmy do wspomagania administrowania systemem. Może być użyty do rozłożenia obciążenia na komputerach pracujących w klastrze, a także do wykorzystania wolnych zasobów podłączonych do niego komputerów zwykłych użytkowników w celu wykonywania obliczeń dużej skali wykorzystując czas gdy komputery są nieobciążone. Wspiera równoległe wykonywanie zadań napisanych w standardzie MPI. Może również zarządzać zasobami znajdującymi się w chmurze obliczeniowej lub gridzie [30].

Wymienione wyżej narzędzia implementują różne modele programowania równoległego. Warto również wspomnieć o takich architekturach jak systemy gridowe lub klastry komputerowe, które są polem do testowania wspomnianych technologii oraz w dużym stopniu przyczyniły się do ich rozwoju.

Systemy klastrowe

Klaster komputerowy jest zbiorem wielu komputerów połączonych zazwyczaj w sieci LAN i mających na celu rozwiązanie złożonego problemu obliczeniowego oraz współpracujących ze sobą. Zazwyczaj są używane do zwiększenia wydajności oraz dostępności konkretnych usług. W porównaniu do gridów komputery pracujące w klastrze mają podobną architekturę oraz zazwyczaj zajmują się wykonywaniem podobnych, skoordynowanych zadań. Mogą posiadać dostęp poprzez sieć do magazynu danych, który można traktować jako pamięć współdzioną. W typowym wykorzystaniu klastra istnieje dodatkowo maszyna klienta, która dystrybuuje zadania do klastra i zbiera wyniki.

Systemy gridowe

Systemy, które integrują heterogeniczne zasoby będące pod kontrolą różnych domen i połączone siecią komputerową WAN. Zapewniają narzędzia do zarządzania zasobami, uwierzytelniania oraz autoryzacji w celu rozwiązywania problemów dużej skali. Gridy są rozszerzeniem idei klastrów. Mogą zapewniać takie usługi jak globalny magazyn danych, aplikacje sieciowe, Software as a Service czyli pojęcia znane z chmur obliczeniowych [79, 69].

2.4. Scala i Akka

Technologiom Scala i Akka został poświęcony osobny rozdział, ze względu na wykorzystanie ich w części praktycznej niniejszej pracy. Scala posiada wsparcie dla dwóch modeli programowania równoległego takich jak STM oraz Actor Model, który został wykorzystany dzięki użyciu frameworka Akka. Z kolei rozszerzenia Akka, takie jak Akka Cluster wspierają użycie tego frameworka w klastrach komputerowych.

Scala

Scala jest językiem funkcyjnym rozwijanym od 2001 roku przez firmę Typesafe, której współtwórcą jest Martin Odersky na co dzień pracujący na Uniwersytecie w Lozannie. Głównym aspektem przemawiającym na korzyść tego języka jest to, że do jego działania wystarczy maszyna wirtualna Javy. Jest to język obiektowy podobnie jak Java, ale łączy również zalety języków funkcyjnych, które w ostatnim czasie stają się coraz bardziej popularne. Zamiarem twórców było stworzenie języka eleganckiego oraz zwięzłego syntaktycznie. Mimo, że Scala jest językiem dynamicznie typowanym zapewnia tzw. type safety [57]. Wprowadza również wiele innowacji oraz ciekawych rozwiązań do konstrukcji języka jak np. case classes, currying, zagnieżdżanie funkcji, DSL, słowo kluczowe lazy lub trait, konstrukcja podobna do interfejsu z języka Java ale mogąca posiadać częściową implementację. Scala preferuje obiekty niemutowalne. Obsługuje również funkcje wyższego rzędu oraz pozwala zwięźle definiować funkcje anonimowe, kładzie również duży nacisk na skalowalność. Dodatkową zaletą tego języka jest to, iż jest on w pełni kompatybilny z językiem Java, co oznacza, że możemy w nim używać bibliotek lub frameworków napisanych w Javie bez żadnych dodatkowych deklaracji. Ostatecznie program napisany w Scali jest komplikowany do kodu bajtowego Javy [60, 65, 23, 78].

Akka

Jednym z kluczowych frameworków wykorzystanych w niniejszej pracy jest Akka. Akka jest projektem Open-source na licencji Apache 2 powstały w roku 2009. Posiada wersję przeznaczoną dla języka Java, jak również dla języka Scala. Stał się częścią implementacji języka Scala od wersji 2.10. Projekt Akka pomimo stosunkowo niedługiej obecności na rynku jest w pełni gotowy do zastosowań produkcyjnych. Cechuje się obecnością wielu interesujących z punktu widzenia niniejszej pracy rozszerzeń, które zostały opisane w jednym z następnych podrozdziałów, wyczerpującej dokumentacji oraz dużej i aktywnej społeczności rozwijającej ten produkt, a także wsparciem komercyjnych firm. Akka używa modelu aktorowego aby zwiększyć poziom abstrakcji i oddzielić logikę biznesową od niskopoziomowego zarządzania wątkami oraz operacjami I/O [71, 22].

Podstawowym bytem w technologii Akka są aktorzy. Aktor zapewnia wysokopoziomową abstrakcję dla lepszej współprzeźności oraz zrównoleglenia operacji. Jest on lekkim, wydajnym oraz sterowanym zdarzeniami procesem, który komunikuje się z innymi aktorami za pomocą asynchronicznych i niemutowalnych wiadomości przechowywanych w skrzynkach odbiorczych, które posiada każdy aktor. Aktor enkapsuluje pewien stan i zachowania, które realizują określone zadania [22].

Każdy aktor w systemie może być identyfikowany na kilka różnych sposobów, głównym z nich jest unikalny adres aktora dzięki któremu może zostać znaleziony przez innych aktorów np. w celu wysłania wiadomości. Aktorzy działają w obrębie systemu zwanego Actor System opisanego w pozycji [1]. System aktorowy można traktować jako pewną strukturę z zaalokowaną pulą wątków, stworzoną w ramach jednej logicznej aplikacji. Konfiguracją puli wątków zajmuje się Akka. Pewne ustawienia mogą być zmienione programatycznie lub w pliku *application.conf*, który jest głównym plikiem konfiguracyjnym systemu aktorowego. System aktorowy zarządza dostępnymi zasobami i może mieć uruchomione wiele aktorów, instancja każdego z aktorów zajmuje zaledwie 300 bajtów pamięci.

Akka Framework dostarcza wszystkich zalet programowania reaktywnego [48] oraz zapewnia między innymi:

- współprzecieność, dzięki zaadaptowaniu modelu aktorowego, programista może zatem skupić się na logice biznesowej zamiast zajmować się problemami współprzezeń,
- skalowalność, asynchroniczna komunikacja pomiędzy aktorami dobrze skaluje się w systemach multiprocesorowych,
- odporność na błędy, framework Akka zapożyczył podejście z języka Erlang, co pozwoliło wykorzystać model *let it crash* [61] (zwany też *fail fast*) do zapewnienia niezawodnego działania systemu i skrócenia jego niedostępności,
- architekturę sterowaną zdarzeniami,
- transakcyjność,
- ujednolicony model programowania dla potrzeb wielowątkowości oraz obliczeń rozproszonych,
- Akka wspiera zarówno API języka Java jak i języka Scala [71].

Zastosowania projektu Akka są bardzo szerokie, można je odnaleźć w następujących dziedzinach:

- analiza danych,
- bankowość inwestycyjna,
- eCommerce,
- symulacje,
- media społecznościowe,
- batch processing,
- transaction processing (online gaming, social media, telecom, finanse, zakłady),
- aplikacje real-time,
- serwisy REST, SOAP, message hub.

Systemy w których potrzebujemy uzyskać wysoką przepustowość oraz małe opóźnienia są dobrym kandydatem do wykorzystania framework'a Akka [22].

Akka Cluster

Jest to rozszerzenie projektu Akka pozwalające zaimplementować komunikację pomiędzy urządzeniami działającymi w obrębie klastra. Zapewnia ono pewien poziom abstrakcji dla protokołu TCP/IP [8].

Dostarcza również odporny na awarie oraz zdecentralizowany serwis członkostwa (ang. membership service) oparty na protokole Gossip [43] oraz automatycznej detekcji niedziałających węzłów (ang. failure detector).

Pojęcia:

- węzeł (ang. node) - logiczny członek klastra, może istnieć wiele węzłów na jednej fizycznej maszynie, identyfikowany krotką: nazwa_hosta:port:uid,
- klaster (ang. cluster) - grupa węzłów zarejestrowana w serwisie członkostwa,
- lider (ang. leader) - węzeł odpowiedzialny za kluczowe akcje pozwalające zachować odpowiedni stan klastra w przypadku dołączania nowych lub awarii istniejących węzłów [22].

Lider jest rolą jaką posiada dany węzeł, każdy węzeł może zostać liderem oraz każdy węzeł jest w stanie w sposób deterministyczny wyznaczyć lidera. Węzły mogą też posiadać inne role, które mogą się przydać np. w ograniczeniu zasięgu komunikacji.

Protokół Gossip wspomniany powyżej pozwala rozpropagować stan klastra do wszystkich jego węzłów tak, aby każdy węzeł posiadał takie same informację o pozostałych członkach klastra. Protokół ten pomaga uzyskać zbieżność stanu klastra we wszystkich jego węzłach w skończonym czasie.

Członkostwo w klastrze jest rozpoczętym komendą *join* wysydaną do jednego z aktywnych członków klastra, gdy każdy węzeł klastra otrzyma informację o dołączającym węźle dzięki protokołowi Gossip, taki węzeł może zostać uznany za osiągalny. Może on jednak utracić ten stan gdy np. wystąpią jakieś problemy z siecią. Stan nieosiągalny może trwać jedynie określony czas po upływie którego jeśli węzeł nie powróci do pełnej sprawności utraci on status członka klastra. Każdy węzeł poza nazwą hosta oraz portem jest identyfikowany dodatkowo unikalnym identyfikatorem, tzw. uid, co pozwala na uruchomienie kilku węzłów klastra na jednej fizycznej maszynie. Jeżeli członek klastra ulegnie awarii lub opuści klasztro na własne życzenie, nie może on ponownie dołączyć do klastra dopóki system aktorowy uruchomiony na nim nie zostanie zrestartowany, co pozwoli na wygenerowanie nowego uid. Każdy węzeł może zmienić swój stan członkostwa lub może on zostać zmieniony dzięki automatycznej detekcji niedziałających węzłów. Ustawienia detekcji niedziałających węzłów oraz protokołu Gossip są konfigurowalne, tzn. że można np. ustalić czas po którym węzeł zostanie uznany za nieosiągalny. Zdarzenia związane ze zmianą stanu węzłów klastra mogą być subskrybowane przez istniejących członków, co ułatwia implementację monitoringu stanu klastra. Więcej szczegółów na temat idei członkostwa w klastrze oraz dostępnych stanów węzłów można znaleźć w pozycji [22].

Kolejną ciekawą opcją w rozszerzeniu Akka Cluster jest wykorzystanie bibliotek Sigar [53] oraz integracja z nimi. Pozwalają one na dostęp do informacji systemowych takich jak zużycie CPU, wykorzystanie pamięci RAM czy stan sieci. Mogą zostać wykorzystane do implementacji load-balancingu lub monitorowania obciążenia urządzeń klastra [4, 3].

Cluster Singleton

Cluster Singleton jest rozszerzeniem projektu Akka zapewniającym jedną instancję aktora danego typu w obrębie klastra lub w obrębie węzłów z wybraną rolą. Może zostać wykorzystane do zaimplementowania modelu Master/Slave opisanego w jednym z poprzednich podrozdziałów. Rozszerzenie to dostarcza implementacji menedżera, który pozwala zarządzać instancjonowaniem aktora typu singleton. Dostęp do działającej instancji jest możliwy z każdego urządzenia działającego w klastrze i odbywa się poprzez aktora pośrednika, tzw. proxy [25].

Distributed Publish Subscribe in Cluster

Rozszerzenie projektu Akka, które umożliwia komunikację między aktorami bez posiadania informacji na których konkretnie urządzeniach poszczególni aktorzy są uruchomieni, czyli lokacja aktora jest transparentna z punktu widzenia komunikacji. Dostarcza ono aktora pełniącego rolę mediatora, który zarządza rejestracją innych aktorów do konkretnych kanałów komunikacji którymi są zainteresowani. Zachowanie to można porównać z subskrypcją RSS [85]. Wiadomość opublikowana w danym kanale powinna zostać dostarczona do wszystkich aktorów, którzy zostali w nim poprzednio zarejestrowani. Rozszerzenie to pozwala również wysłać wiadomość do jednego lub większej ilości aktorów pasujących do określonego wzorca [19].

3. Niekonwencjonalne metody realizacji obliczeń

Miniaturyzacja oraz rozwój procesorów zwiększa możliwości ich zastosowania. Dzięki temu współczesne systemy komputerowe przybierają coraz bardziej nietypowe formy, co stwarza możliwości ich wykorzystania do niekonwencjonalnych metod obliczeń, takich jak Ubiquitous Computing opisane w poniższym rozdziale. Rozdział zawiera również charakterystykę urządzeń SoC oraz opis idei Internet of Things.

3.1. Architektura System on a Chip

Urządzenia SoC (System-on-Chip) są to układy, które oprócz głównego procesora opartego np. na architekturze ARM, zawierają cyfrowe i analogowe moduły składające się w jeden system elektroniczny. Poszczególne moduły zazwyczaj są tworzone przez różnych producentów od których firmy produkujące urządzenia SoC zamawiają sprzęt. W porównaniu do mikrokontrolerów charakteryzuje się one posiadaniem większej ilości pamięci RAM oraz CPU o stosunkowo dużej mocy obliczeniowej, która jest wystarczająca do uruchomienia systemów operacyjnych podobnych do tych uruchamianych na komputerach PC. Charakteryzuje się również posiadaniem interfejsów umożliwiających podpinanie urządzeń peryferyjnych tych samych co do zwykłych komputerów PC. Urządzenia SoC zyskały swą popularność wśród zwykłych użytkowników dopiero w ostatnich latach, głównie za sprawą taniego mini komputera jakim jest Raspberry Pi. Mają one zastosowanie m.in. jako systemy wbudowane, w dziedzinie Home Automation jak również posiadają wiele innych multimedialnych zastosowań ze względu na możliwości zbliżone do zwykłych PC oraz niewielkie rozmiary. Z założenia powinny mieścić się na jednej płycie drukowanej, być energooszczędne oraz tanie w produkcji seryjnej.

Typowe urządzenie SoC powinno zawierać:

- procesor CPU,
- pamięć RAM, ROM, EEPROM lub FLASH,
- układy czasowo-licznikowe,
- kontrolery transmisji szeregowej lub równoległej,
- przetworniki analogowo-cyfrowe lub cyfrowo-analogowe,
- obwody zarządzania zasilaniem.

Spotykane są również urządzenia zawierające procesory GPU oprócz jednostki CPU [54, 68]. Poniżej wymieniono kilka popularnych urządzeń SoC.

Parallel

Urządzenie posiada dwurdzeniowy procesor ARM o taktowaniu 800MHz, wielordzeniowy procesor Epiphany o architekturze RISC oraz 1GB pamięci RAM. Dodatkowo posiada złącze kart MicroSD, USB 2.0, port Ethernet, HDMI oraz możliwość uruchomienia systemów operacyjnych z rodziny Linux.

Urządzenie jest wielkości karty kredytowej, a jego cena oscyluje w granicach 100\$. Istnieje kilka różnych specyfikacji tego urządzenia różniących się ilością rdzeni procesora Epiphany się oraz ceną. Dzięki obecności procesora Epiphany urządzenie dobrze sprawdza się w obliczeniach równoległych, które mogą zostać zaimplementowane przy wykorzystaniu takich technologii jak OpenMP, MPI lub OpenCL. Projekt został zapoczątkowany na platformie Kickstarter w 2012 roku [41].

Intel Galileo

Urządzenie stworzone przez firmę Intel bazujące na procesorze tej samej firmy klasy x86 Pentium o taktowaniu 400MHz (32 bity) wraz z 256MB pamięci RAM. Jest kompatybilne z systemami operacyjnymi takimi jak Windows, Linux lub Mac OS, a także z wieloma rozszerzeniami czy bibliotekami platformy Arduino [14]. Cena oraz rozmiary urządzenia są podobne jak w przypadku urządzeń Parallel. Urządzenie to posiada również złącze Ethernet, MicroSD oraz mini PCI Express. Nie posiada natomiast karty dźwiękowej ani procesora graficznego [31].

Sharks Cove

Urządzenie promowane przez firmę Microsoft i kompatybilne z Windowsem 10 oraz systemem Android. Pracuje pod kontrolą procesora Intel Atom o taktowaniu 1.33GHz, wyposażone jest również w zintegrowany procesor graficzny Intel HD Graphics, 1GB pamięci RAM, kartę dźwiękową oraz m.in. w złącze MicroSD, HDMI, USB 2.0 oraz Ethernet. Posiada trochę większe rozmiary od innych opisywanych tutaj urządzeń. Cena urządzenia również jest wyższa, bo wynosi około 300\$ [52].

Raspberry Pi

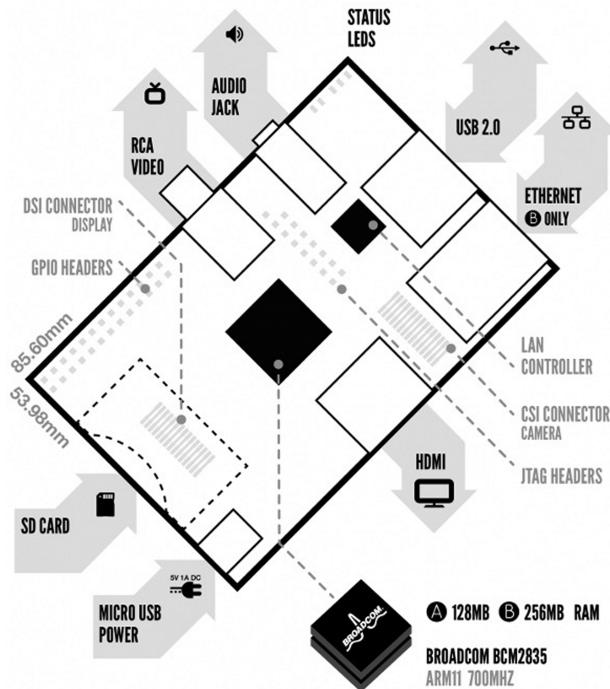
Raspberry Pi jest urządzeniem SoC wykorzystywany w niniejszej pracy. Jest to urządzenie stworzone przez fundację non-profit Raspberry Pi Foundation, oparte na architekturze ARM. Pierwsza jego wersja została wydana w roku 2012. Na rysunku 3.1 przedstawiono schemat poglądowy urządzenia.

Urządzenie Raspberry Pi w wersji B posiada jednostkę CPU o taktowaniu 700MHz oraz 256MB pamięci RAM. Poza procesorem CPU jest również wyposażone m.in. w jednostkę GPU, dwa porty USB, złącze kart SD, złącze Ethernet 100Mb/s oraz złącze MicroUSB wykorzystane do zasilania. W trakcie obciążenia pobiera około 4W prądu. Na tym urządzeniu może być zainstalowana m.in. jedna z wielu dystrybucji Linuxa. Urządzenie to może służyć zarówno jako platforma deweloperska jak i centrum domowej rozrywki. Znalazło również zastosowanie w budowie klastrów obliczeniowych. Przykładem może być praca opisana w pozycji [62], której podgląd jest widoczny na rysunku 3.2. Pełną specyfikację urządzenia można znaleźć w pozycji [26].

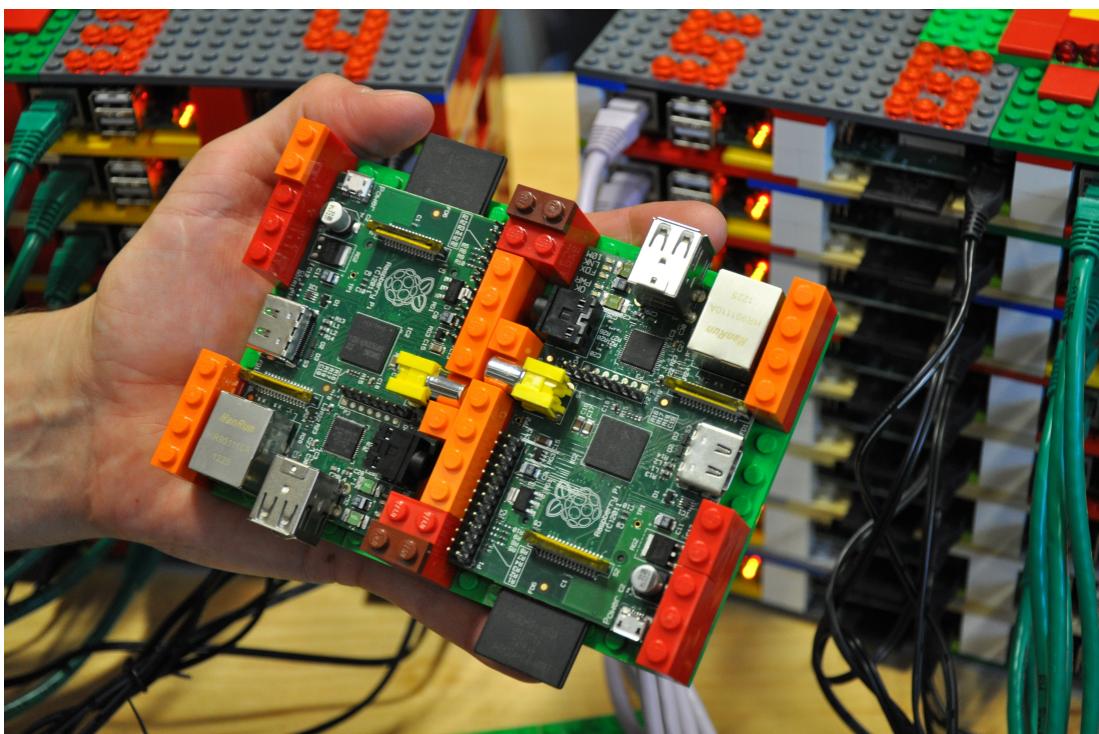
W roku 2015 został udostępniony do dystrybucji nowy model urządzenia oznaczony wersją 2, który różni się od poprzedniej wersji m.in. tym, że posiada 1GB pamięci RAM oraz czterordzeniowy procesor ARM o taktowaniu 900MHz a także możliwość uruchomienia poza dystrybucjami Linuxa również systemu Windows w wersji 10. Cechuje się także mniejszym rozmiarem - wielkości karty kredytowej. Cena tej wersji urządzenia wynosi 35\$ [46].

Arduino

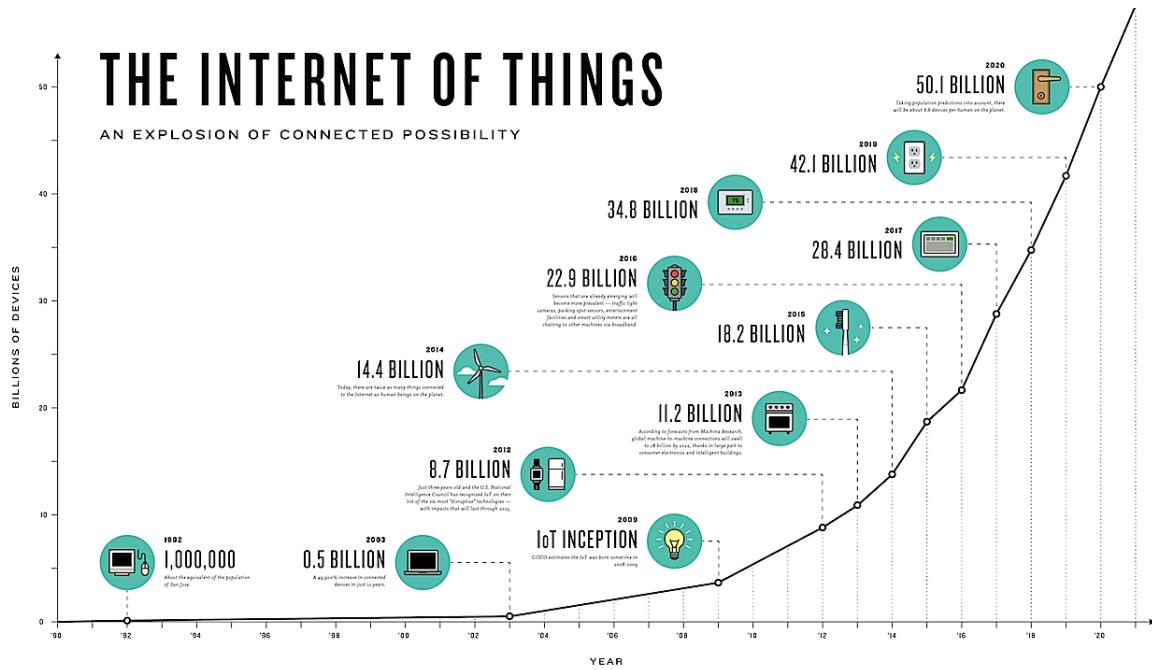
Jest to rodzina mikrokontrolerów bazująca na technologii Atmel AVR i bardzo popularna wśród użytkowników. Nie są to co prawda urządzenia SoC, ponieważ posiadają znacznie mniej pamięci, słabsze procesory oraz mniejszą ilość wbudowanych modułów ale są oparte na otwartej platformie, która jest łatwo programowalna, posiada dużą społeczność zainteresowanych osób przyczyniających się do jej rozwoju oraz wiele darmowych rozszerzeń, a także bibliotek ułatwiających ich użycie oraz rozszerzających możliwości tych urządzeń. Dzięki temu są one często wykorzystywane w połączeniu z urządzeniami SoC lub do sterowania innymi urządzeniami np. w Home Automation [14].



Rysunek 3.1: Schemat Raspberry Pi model B [26].



Rysunek 3.2: Klaster Raspberry Pi [47].



Rysunek 3.3: Prognozy ilości urządzeń podłączonych do Internetu [16].

Dzięki popularyzacji, miniaturyzacji oraz obniżeniu cen urządzeń SoC oraz mikrokontrolerów rozwinała się idea Internet of Things. Firmy które wypuszczają nowe urządzenia na rynek starają się aktywizować społeczności wokół tych urządzeń, np. poprzez udostępnianie darmowych urządzeń we wczesnej fazie promocji zainteresowanym użytkownikom lub uczelniom. Dla przykładu można podać akcję Intelu w przypadku urządzenia Galileo, gdzie aż 50 tys. egzemplarzy trafiło do tysiąca uczelni na całym świecie a wśród nich także do 18 polskich placówek [32].

3.2. Internet of Things

Współczesne systemy komputerowe stają się coraz mniejsze oraz tańsze, co skutkuje obecnością komputerów w prawie każdej dziedzinie życia. Dzięki rozwojowi urządzeń mobilnych, tanich sensorów, bezprzewodowej transmisji danych oraz technologiom działającym w chmurze zasięg Internetu zostaje zwiększyły. Skutkuje to tym, że ilość danych do przetwarzania ciągle wzrasta jak również wzrasta skala komunikacji tych urządzeń. Do tego typu urządzeń zaliczają się już nie tylko komputery osobiste i urządzenia mobilne takie jak smartfony, tablety czy inteligentne zegarki ale również urządzenia gospodarstwa domowego takie jak telewizory, lodówki, artykuły oświetleniowe i grzewcze oraz wiele innych urządzeń posiadających różnego rodzaju sensory i zbierających dane również na skalę przemysłową.

Koncepcja IoT (ang. Internet Of Things) rozwinięła się wokół wszystkich urządzeń podłączonych do Internetu. Zapoczątkował ją Kevin Ashton w 1999 roku. Zauważono tutaj wiele nowych możliwości jak również tendencję do wzrostu liczby urządzeń połączonych z Internetem. Firma Cisco szacuje, że do roku 2020 liczba takich urządzeń może przekroczyć 50 miliardów [17]. Na rysunku 3.3 przedstawiono poglądowy wykres tego zjawiska.

Warto również wspomnieć o istnieniu podobnego pojęcia zwanego IoE (ang. Internet of Everything) i zapoczątkowanego przez firmę Cisco, które czasem jest używane zamiennie do IoT. Ciekawostką jest fakt, że licznik pokazujący przybliżoną liczbę urządzeń podłączonych do Internetu jest aktualizowany na bieżąco oraz zamieszczony na stronie internetowej firmy Cisco. Aktualnie wskazuje na ponad 16 miliardów urządzeń i cały czas rośnie [17].

Wiele firm komercyjnych zauważa w tychże prognozach sposób na zysk, dlatego tanie, miniaturowe platformy komputerowe takie jak urządzenia SoC są bardzo mocno promowane.

Internet oraz wiele współcześnie używanych technologii zostało wymyślonych o wiele wcześniej aniżeli koncepcja IoT więc nie wszystkie są gotowe na taką ilość urządzeń oraz danych. Pozostawia to miejsce do badań nad nowymi rozwiązaniami oraz istniejącymi problemami, których ilość będzie rosła wraz ze wzrostem skali tego zjawiska [33, 63].

3.3. Ubiquitous Computing

Koncepcją powiązaną z ideą IoT jest Ubiquitous Computing, czyli wszechobecne obliczenia. Po raz pierwszy tej nazwy użył Mark Weiser w 1991 roku (czyli parę lat wcześniej przed wymyśleniem nazwy IoT) w odniesieniu do urządzeń działających w Internecie.

Idea ta określa sytuację gdzie systemy komputerowe staną się integralną częścią otaczającego nas świata, stając się bardziej transparentne dla otoczenia. Miliony małych bezprzewodowych urządzeń wykonujących różne obliczenia staną się niewidoczne dla użytkowników dzięki miniaturyzacji i integracji ze środowiskiem, ale będą objawiać swą obecność we wszystkich dziedzinach życia. Środowisko w którym żyjemy stanie się bardziej inteligentne.

Istotą tej idei jest wsparcie ze strony takich technologii jak m.in. Internet, urządzenia mobilne, mikroprocesory, urządzenia SoC, sensory czy sieci bezprzewodowe. Temat ten dotyczy również wiele innych powiązanych dziedzin jak obliczenia rozproszone, obliczenia zależne od kontekstu (ang. context-aware computing), protokoły transmisji, geolokalizacja, sztuczna inteligencja (ang. AI) oraz interakcja użytkownika z komputerem [58, 83].

4. Mobilny klaster obliczeniowy

Idee opisane w poprzednich rozdziałach a w szczególności urządzenia SoC oraz obliczenia równoległe znalazły odzwierciedlenie w głównej części niniejszej pracy. W poniższym rozdziale opisano przygotowaną platformę pozwalającą na zestawienie klastra obliczeniowego wykorzystującego urządzenia SoC oraz technologie Scala i Akka. Znalazły się tutaj m.in. zarys początkowej koncepcji, szczegółowe implementacyjne gotowego rozwiązania oraz opis wykorzystanych technologii.

4.1. Koncepcja

Na rysunku 4.1 zaprezentowano początkową koncepcję przedstawiającą sieć lokalną i działający w niej klaster obliczeniowy w skład którego wchodzą urządzenia Raspberry Pi, gdzie jedno z nich pełni rolę *master* natomiast pozostałe pełnią rolę *slave*. W klastrze znajduje się również uruchomiony serwer WWW służący do komunikacji z użytkownikiem za pomocą przeglądarki internetowej.

W klastrze uruchomione są aplikacje oparte o technologie JVM, które komunikują się ze sobą za pomocą protokołu TCP/IP.

Na urządzeniach klastra wykonywane są obliczenia równolegle, które są rozpoczynane przez klienta klastra a następnie dystrybuowane przez węzeł posiadający rolę *master* pomiędzy pozostałe węzły posiadające rolę *slave*, które zajmują się właściwymi obliczeniami.

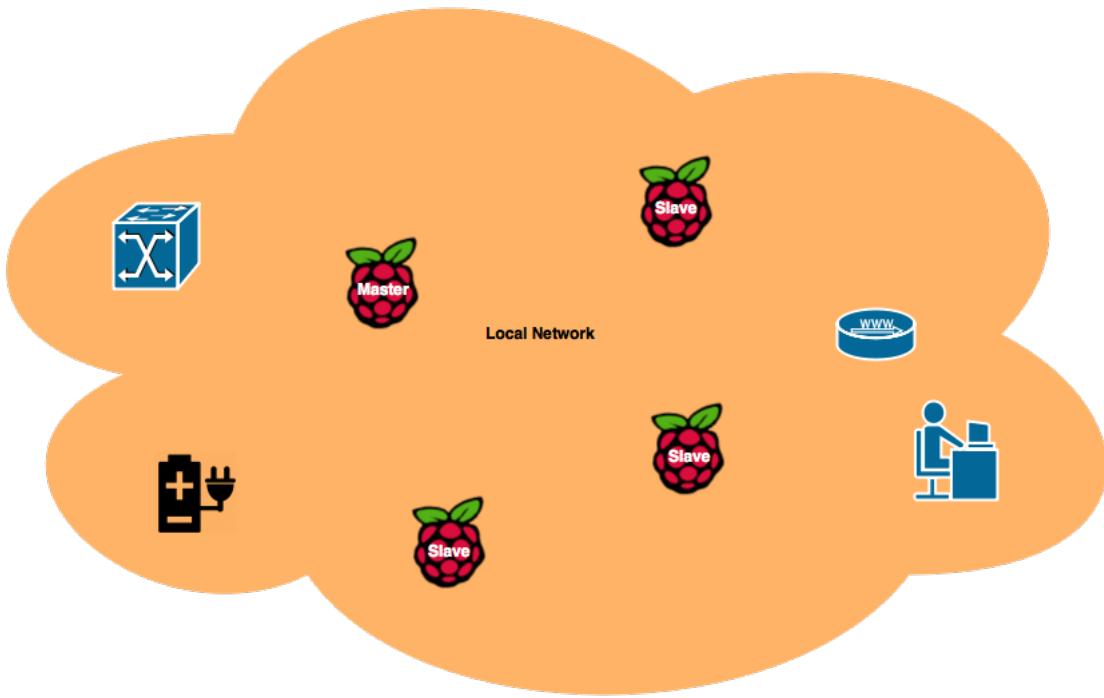
Powinna istnieć tylko jedna instancja *mastera* w obrębie klastra a w przypadku awarii urządzenia na którym znajduje się ta instancja powinna zostać uruchomiona inna na jednym z pozostałych urządzeń, które będzie w stanie odtworzyć stan poprzedniej instancji. Co wymaga, aby co określony czas stan *mastera* był zapisywany. Stan ten jest istotny ze względu na przechowywane przez niego informacje o wynikach zadania, które nie dotarły jeszcze do klienta lub informacje o zadaniach rozpoczętych przez klienta ale nie wysłanych jeszcze do żadnego *workera*. Komunikacja na linii Master-Worker powinna być zaimplementowana w taki sposób, aby *master* nie stanowił tzw. „wąskiego gardła” klastra. Wyzwaniem jest również wybór algorytmu losowania nowego *mastera*, jak również load-balancing zadań, który powinien być zaimplementowany w sposób, aby obciążenie *workerów* było równomierne i powinien on wyeliminować sytuacje gdy *worker* jest w stanie bezczynności mimo, że istnieje zakolejkowane zadanie.

Wyniki zadań odsyłane są do *mastera*, który następnie wysyła je klientowi. W przypadku dugo trwających zadań przesyłane są wyniki częściowe co jakiś określony czas, aby poinformować klienta o postępie prac.

W trakcie trwania obliczeń częściowe wyniki co jakiś określony czas są zapisywane w bazie danych po to, aby możliwe było ich odzyskanie w sytuacji awarii któregoś urządzenia.

Interesującą kwestią jest wybór tego jakie dane wymagają persystencji oraz jak często powinna się ona odbywać, a także gdzie będzie przechowywany magazyn danych. Jeżeli podjęta zostanie decyzja, że na urządzeniu, które może ulec potencjalnej awarii, to będzie wymagana replikacja, aby zapewnić bezpieczeństwo danych.

Poza wykonywaniem zadań platforma ma za zadanie monitorować pracę urządzeń, informować o dołączaniu nowych oraz opuszczaniu klastra przez urządzenia będące jego członkiem – rozróżniane są sytuacje gdy urządzenie opuści klaster na własne życzenie albo ulegnie awarii, czyli przestanie brać



Rysunek 4.1: Koncepcja klastra złożonego z urządzeń SoC

udział w komunikacji z pozostałymi urządzeniami i w takim przypadku zostanie uznane za niedostępne oraz usunięte z klastra po jakimś określonym czasie.

4.2. Platforma

Niniejszy podrozdział omawia wybrane aspekty przygotowanej platformy, takie jak architektura, protokoły komunikacji oraz sposób przydzielania zadań. Ponadto zawiera opis dystrybucji przygotowanego rozwiązania.

Architektura

Węzeł klastra jest to instancja aplikacji niezwiązana z fizycznym urządzeniem. Co oznacza, że na jednym urządzeniu może zostać uruchomionych kilka węzłów, czyli kilka aplikacji. W skład platformy wchodzą właściwie dwie aplikacje nazwane *frontend* oraz *backend*. Posiadają one różne odpowiedzialności jak i różne rodzaje uruchomionych na nich aktorów, których dokładniejszy opis znajduje się w kolejnym podrozdziale.

Aplikacja *frontend* odpowiada za interakcję z użytkownikiem dzięki takim technologiom jak Play Framework, AngularJS, WebSocket oraz protokołowi HTTP. Komunikuje się ona z aplikacją *backend* w celu zlecenia wykonania zadania oraz odebrania wyników. Posiada ona interfejsowy. Instancja aplikacji jest traktowana jako osobny węzeł klastra.

Aplikacja *backend* nie posiada interfejsu użytkownika. Zajmuje się realizacją zleconych zadań. Instancja tej aplikacji również odpowiada jednemu węzlowi klastra. W klastrze może być uruchomionych wiele takich instancji, po jednej na każdym urządzeniu lub jeśli urządzenie posiada wystarczająco dużo mocy obliczeniowej może posiadać ich kilka. Każda uruchomiona aplikacja posiada uruchomionego

workerą oraz tylko jedna z nich posiada również uruchomionego *mastera*, gdyż w klastrze powinna być tylko jedna aktywna instancja *mastera*.

Na rysunku 4.2 przedstawiono podgląd architektury, gdzie zielonym kolorem zaznaczono aplikację *frontend* natomiast niebieskim aplikację *backend*. Założono, że każda aplikacja jest uruchomiona na osobnym urządzeniu pracującym w klastrze, są dwa urządzenia wykonujące obliczenia oraz jedno urządzenie klienckie. Na wspomnianym rysunku pokazano także kilka aktorów uruchomionych na wybranych urządzeniach. Założono również, że urządzenia pracują w sieci lokalnej z wykorzystaniem technologii Ethernet. Użytkownik komunikuje się za pomocą protokołu HTTP z urządzeniem klienckim, które z kolei przesyła odpowiednie żądania do urządzenia posiadającego aktywnego *mastera*. Urządzenie klienckie nie musi posiadać informacji o tym gdzie dokładnie jest aktywny *master*, ponieważ dostęp do *mastera* odbywa się poprzez proxy. Natomiast aplikacja kliencka powinna mieć podany adres IP przynajmniej jednego urządzenia pracującego w klastrze. Może mieć informację o kilku adresach IP urządzeń pracujących w klastrze, co pozwala uniknąć problemów gdy jedno z urządzeń nie odpowiada.

Urządzenia wykonujące obliczenia w klastrze komunikują się ze sobą za pomocą protokołu TCP/IP, którym zarządza Akka. Odkąd aktywny *master* posiada listę wszystkich zarejestrowanych *workerów*, potrafi on w łatwy sposób nawiązywać z nimi kontakt oraz informować ich o nadchodzących zadaniach od klienta. Z kolei komunikacja w drugą stronę, np. gdy *worker* skończył wykonywać zadanie i chce wysłać zgromadzone wyniki odbywa się poprzez proxy, ponieważ *worker* nie wie na którym urządzeniu znajduje się aktywny *master*. Gdy *master* otrzyma wyniki zadania od realizującego je *workerą* odsyła je do klienta za pomocą rozszerzenia *Distributed Publish Subscribe in Cluster* opisanego w jednym z poprzednich rozdziałów. Co oznacza, że nie musi posiadać informacji o adresie IP klienta, ponieważ opublikowane wyniki trafią tylko do zainteresowanych odbiorców.

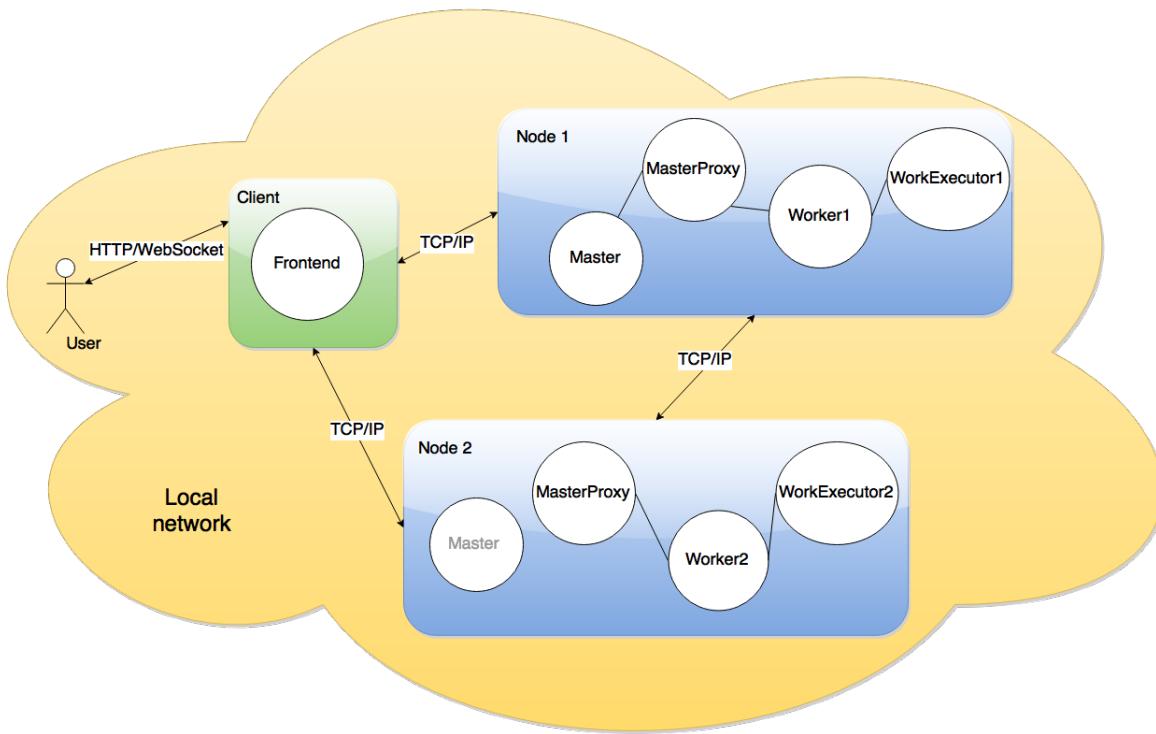
Na rysunku 4.2 pokazano jedynie kilka najważniejszych aktorów działających na urządzeniach będących częścią klastra. Zostali oni opisani razem z pozostałymi, którzy również są aktywni podczas działania platformy w następującym podrozdziale wraz z podziałem na aplikacje w których są wykorzystywani.

Wybrani aktorzy

Poniżej wymieniono kilka aktorów działających w klastrze z uwzględnieniem tego na jakiej aplikacji wchodzącej w skład przygotowanej platformy są oni uruchomieni.

Aplikacja *backend*:

- *ClusterSingletonManager*, aktor dostarczony wraz z rozszerzeniem *Akka Cluster Singleton*, który odpowiada za to, aby w klastrze była aktywna tylko jedna instancja *mastera*. Zostaje on uruchomiony na wszystkich węzłach lub tylko na węzłach z wybraną rolą. Instancjonuje aktora typu singleton na najstarszym węźle, czyli takim, który dołączył do klastra najwcześniej. Monitoruje on również stan dostępności węzła na którym uruchomiony jest aktor typu Singleton i w przypadku awarii startuje nową instancję *mastera* na innym węźle klastra.
- *ClusterSingletonProxy*, aktor współpracujący z poprzednim opisany aktorem i będący pośrednikiem (ang. proxy) zapewniającym dostęp do aktora typu singleton czyli w tym przypadku *mastera*. Jest on uruchomiony na każdym węźle, który potrzebuje komunikować się z *masterem* i działa jako router przekierowujący wszystkie przychodzące wiadomości do aktora typu singleton. Jeśli aktor ten jest niedostępny np. ze względu na awarię lub jest w trakcie odzyskiwania sprawności, zadaniem aktora *ClusterSingletonProxy* jest przechowywanie wszystkich odebranych wiadomości aż do momentu ponownego nawiązania komunikacji [25].
- *DistributedPubSubMediator*, aktor ten zostaje uruchomiony na wszystkich węzłach klastra lub tylko na węzłach z wybraną rolą. Zarządza rejestracją innych aktorów do wybranych kanałów komunikacji oraz replikuje tą wiedzę pośród inne instancje pracujące na pozostałych urządzeniach klastra tak, aby zachować spójność tych danych w obrębie klastra. Zajmuje się również wysyłaniem wiadomości z każdego urządzenia klastra do zarejestrowanych aktorów pracujących na jakimkolwiek urządzeniu w obrębie klastra [19].

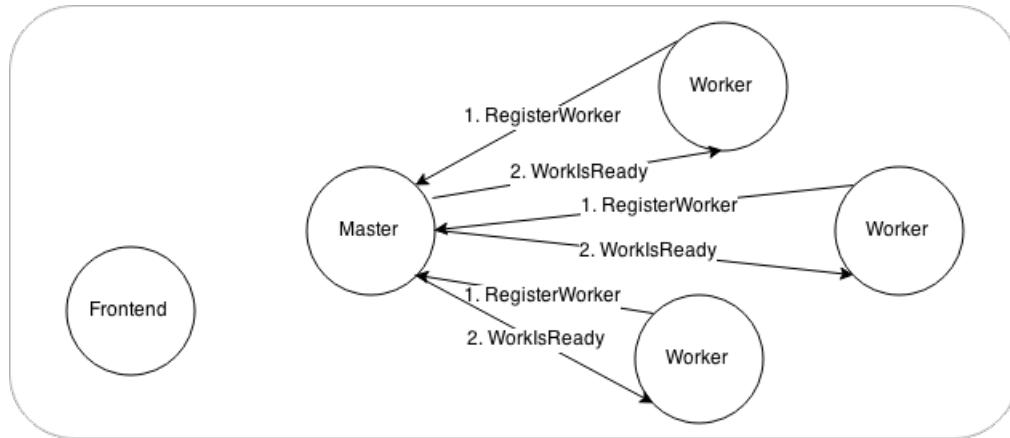


Rysunek 4.2: Podgląd architektury.

- Master, jest to aktor typu singleton. Oznacza to, że w obrębie klastra powinna być aktywna tylko jedna instancja tego aktora. Jest to zapewnione poprzez aktora ClusterSingletonManager opisanego powyżej. Dostęp do *mastera* uzyskuje się poprzez proxy, czyli aktora ClusterSingletonProxy z każdego urządzenia pracującego w klastrze. Głównymi zadaniami tego aktora są: przydzielanie zadań *workerom*, zarządzanie ich rejestracją, odbieranie wyników i przesyłanie ich do klienta, reagowanie w sytuacji awarii jednego z *workerów* oraz zapisywanie wyników do bazy.
- Worker, jest to aktor realizujący zadania otrzymane od *mastera* po uprzedniej rejestracji w serwisie *mastera*. Każdy Worker uruchamia jednąinstancję aktora wykonującego właściwe obliczenia zwanego WorkExecutor po to, aby zachować responsywność podczas komunikacji z *masterem*. Zajmuje się on również procesem migracji, gdzie znowu jest wykorzystywany mechanizm publikowania w wybranym kanale dostarczony wraz z rozszerzeniem *Distributed Publish Subscribe*.
- WorkExecutor, aktor wykonujący właściwe obliczenia czyli realizujący zadanie zlecone przez klienta. Jest on instancjonowany przez aktora Worker i komunikuje się tylko z nim.

Aplikacja *frontend*:

- Frontend, aktor z którym komunikuje się użytkownik. Odbiera on wyniki zadań od *mastera* oraz przesyła mu zadania do wykonania. Do komunikacji z *masterem* używa proxy, natomiast wyniki odbiera dzięki rozszerzeniu *Distributed Publish Subscribe* oraz faktem bycia zarejestrowanym w wybranym kanale komunikacji.
- Metrics, aktor subskrybujący zdarzenia związane ze zmianą różnych metryk klastra, np. obciążenia CPU lub pamięci RAM. Dzięki połączeniu WebSocket przesyła te informacje na warstwę widoku.



Rysunek 4.3: Rejestracja workerów.

Tablica 4.1: Opis komunikatów - rejestracja workerów.

Nazwa komunikatu	Opis
RegisterWorker	Komunikat rejestrujący <i>worker</i> a o danym ID w serwisie <i>mastera</i> .
WorkIsReady	Informacja o tym, że <i>master</i> posiada zakolejkowane zadanie po które dany <i>worker</i> może się zgłosić.

- Monitor, aktor obserwujący zdarzenia związane z członkostwem w klastrze. Jeżeli jakiś węzeł dołącza do klastra lub go opuszcza taka informacja jest aktualizowana na warstwie widoku w aplikacji klienckiej.

Poza wyżej wymienionymi aktorami działającymi w aplikacji *frontend*, posiada ona również kilka podobnych aktorów co aplikacja *backend*, m.in. ClusterSingletonProxy oraz DistributedPubSubMediator, którzy służą do komunikacji z *masterem* oraz odbieraniem wyników realizowanych zadań.

Protokoły komunikacji

Na rysunkach 4.3 oraz 4.4 przedstawiono dwa najważniejsze protokoły definiujące kolejność oraz rodzaje przesyłanych wiadomości podczas komunikacji na linii Frontend-Master-Worker-WorkExecutor. Widoczni są tam również aktorzy biorący lub nie udział we wspomnianej komunikacji.

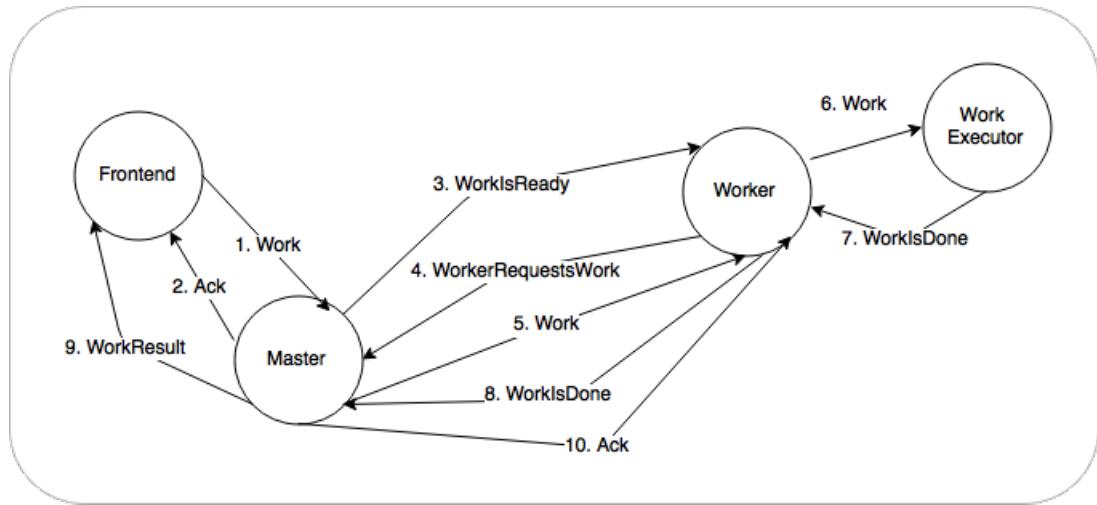
Na rysunku 4.3 przedstawiono proces rejestracji workerów w serwisie *mastera* po to, aby *master* mógł ich poinformować o tym, że posiada jakieś zadanie, które trzeba zrealizować.

W tabeli 4.1 przedstawiono opis komunikatów zawartych na rysunku 4.3.

Na rysunku 4.4 pokazano przepływ informacji podczas zlecania oraz realizacji zadania. Dla uproszczenia przedstawiono przypadek zadania zrealizowanego bez przesyłania częściowych wyników. W przypadku zadań trwających długo lub gdzie czas trwania nie jest z góry określony (tzw. *long-running jobs*) wprowadzono również komunikat *WorkInProgress*, który jest przesyłany cyklicznie w trakcie wykonywania zadania i zawiera jego częściowe wyniki, można go umieścić pomiędzy komunikatami *Work* a *WorkIsDone*.

W tabeli 4.2 przedstawiono opis komunikatów przesyłanych w trakcie procesu realizacji zadań.

Do odpowiedniego rozłożenia obciążenia pomiędzy pracującymi workerów początkowo rozważano podejście gdzie to *master* decyduje, któremu *workerowi* przydzielić zadanie bazując między innymi na metrykach takich jak obciążenie CPU lub zajętość pamięci RAM. Jednak okazało się ono nieefektywne, ponieważ te metryki zmieniały się zazwyczaj z opóźnieniem pozostawiając *worker*a w stanie bezczynności przez nieokreślony czas. Zdecydowano się zatem zaimplementować podejście zwane *Work Pulling Pattern* [59] w którym to wolny *worker* zgłasza się po zadanie, a nie oczekuje na akcję *mastera*.

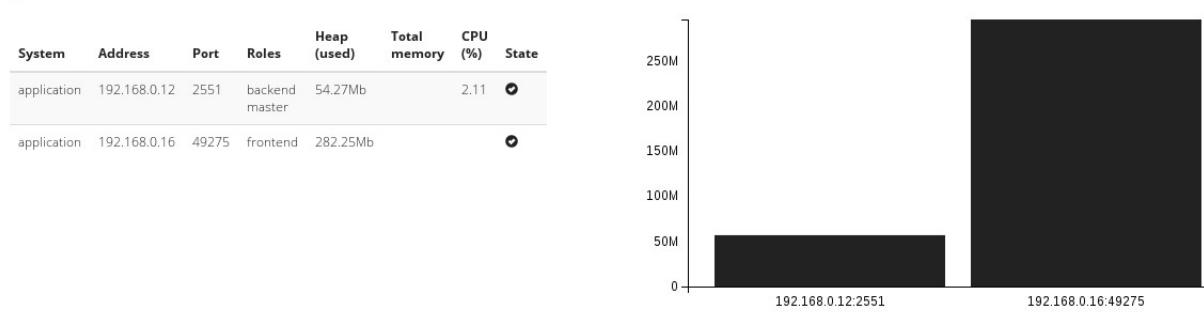


Rysunek 4.4: Protokół Master/Worker.

Tablica 4.2: Opis komunikatów przesyłanych w trakcie realizacji zadania.

Nazwa komunikatu	Opis
Work	Komunikat przechowujący parametry konfiguracyjne zadania, jak również jego ID.
Ack	Potwierdzenie odebrania poprzedniego komunikatu.
WorkIsReady	Informacja o tym, że <i>master</i> posiada zakolejkowane zadanie po które dany <i>worker</i> może się zgłosić.
WorkerRequestsWork	Komunikat przechowujący ID <i>workera</i> i informujący <i>mastera</i> , że dany <i>worker</i> jest wolny i może zająć się kolejnym zadaniem.
WorkIsProgress	Komunikat przechowujący częściowe wyniki zadania, jak również jego ID oraz ID <i>workera</i> .
WorkIsDone	Komunikat przechowujący WorkResult oraz ID <i>workera</i> .
WorkResult	Komunikat przechowujący końcowe wyniki zadania jak i jego ID.

Dashboard akka cluster



Rysunek 4.5: Podgląd stanu klastra.

Tablica 4.3: Lista zdarzeń sterujących stanem zadania.

Nazwa stanu	Opis
WorkAccepted	Stan zadania po przyjęciu go przez <i>mastera</i> oraz wysłaniu potwierdzenia do klienta.
WorkStarted	Stan zadania przekazanego do realizacji dla <i>workerów</i> .
WorkInProgress	Stan zadania w trakcie trwania obliczeń, przechowujący częściowe wyniki zadania.
WorkCompleted	Stan zadania otrzymany po zakończeniu obliczeń, przechowujący końcowe wyniki.
WorkerFailed	Stan zadania zakończonego niepowodzeniem.
WorkerTimedOut	Stan zadania ustawiany jeśli <i>worker</i> pracujący nad danym zadaniem nie odeśle wyników w określonym czasie.

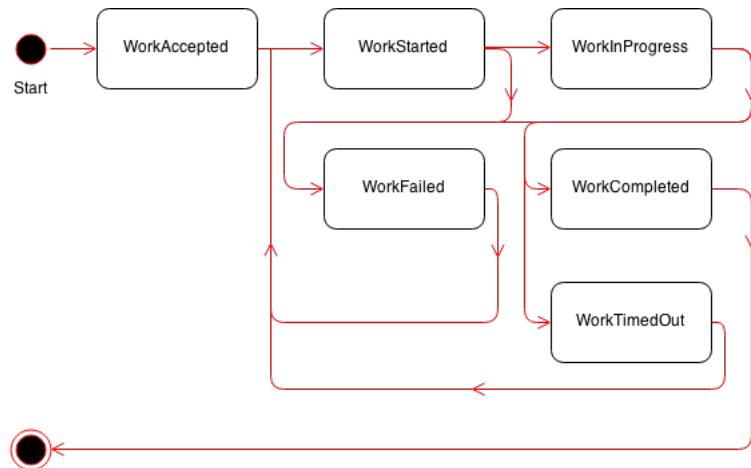
Monitoring

Na rysunku 4.5 przedstawiono interfejs klienta prezentujący obecny stan klastra. Wylistowano na nim również listę urządzeń pracujących w klastrze, z uwzględnieniem aktywnego *mastera*, adresu IP wraz z portem, dostępności urządzenia oraz kilku metryk, m.in. zużycia CPU oraz pamięci RAM.

Stan klastra propagowany jest za pomocą protokołu *Gossip* opisanego w poprzednim rozdziale. Jeżeli jakieś urządzenie nie odpowiada przez określony czas w wyniku np. problemów z siecią, jest ono uznawane jako niedostępne, czyli nie może już przyjmować żadnych zadań ale może jeszcze powrócić do pracy w klastrze jeżeli odzyska sprawność w ciągu najbliższych kilku sekund. Jeżeli nie, w takim przypadku jest ono usuwane z listy urządzeń pracujących w klastrze. Po usunięciu urządzenia z klastra nie może ono dołączyć ponownie do klastra dopóki aplikacja uruchomiona na tym urządzeniu nie zostanie zrestartowana. Pozwala to zapobiec sytuacji, gdy w trakcie problemów z siecią, dane urządzenia odłączy się na chwilę od klastra aktywując własnego *mastera* a następnie po powrocie do klastra będą aktywne dwa urządzenia *master*. Dzieje się tak dlatego, że w przypadku gdy urządzenie utraci kontakt z pozostałymi urządzeniami, nie może ono z pewnością stwierdzić czy to ono utraciło połączenie lub czy inne urządzenia uległy awarii więc zakładając, że jest jedynym pozostały urządzeniem w klastrze.

Stan *mastera* jest zapisywany w bazie danych, co pozwala na jego odtworzenie w przypadku awarii urządzenia z aktywnym *masterem*. Do bazy danych persystowane są zserializowane zdarzenia zgodnie ze wzorcem Event Sourcing opisanym w pozycjach [27, 28]. Zdarzenia te odzwierciedlają stan przyjętego zadania. Listę zdarzeń oraz ich opis przedstawiono w tabeli 4.3.

Dla przykładu, jeżeli klient zadał zadanie i zmienił ono stan na *WorkAccepted* (czyli zostało zaakceptowane ale nie przesłane jeszcze do żadnego *worker*) a w tym samym czasie nastąpiła jakas awaria przerywająca pracę *mastera*, kolejna instancja *mastera* zostanie aktywowana na innym urządzeniu i odt-



Rysunek 4.6: Przejścia pomiędzy stanami zadania.

worzy stan *mastera*, który uległ awarii, co oznacza, że klient nie będzie musiał ponownie rozpoczynać zadania, lecz jego poprzednio rozpoczęte zadanie zostanie przekazane do realizacji do wolnego workera.

Diagram prezentujący dopuszczalne zmiany stanów zadania przedstawiono na rysunku 4.6.

Dystrybucja

Do łatwej dystrybucji platformy pomiędzy różnymi urządzeniami wykorzystano dodatek projektu Akka o nazwie Microkernel oraz rozszerzenie SBT o nazwie sbt-native-packager.

Akka Microkernel oraz sbt-native-packager dostarczają mechanizmu archiwizowania dzięki któremu można udostępniać aplikację jako pojedynczy plik bez potrzeby uruchamiania lub instalowania dodatkowych aplikacji, dostarczania zależności lub ręcznego tworzenia skryptów startowych. Pozwalają również na wystartowanie systemu aktorowego używając klasy z metodą statyczną main [5, 50].

Umożliwia to łatwą instalację oraz uruchomienie platformy na różnych urządzeniach klastra. Do uruchomienia platformy na danym urządzeniu wystarczy aby posiadało ono zainstalowany system wraz z wirtualną maszyną Javy oraz archiwum z platformą, które po rozpakowaniu pozwala uruchomić odpowiednią aplikację za pomocą jednego skryptu wykonywalnego. Żadne dodatkowe oprogramowanie nie jest wymagane.

4.3. Inne wykorzystane technologie

Akka Persistence

Rozszerzenie Akka Persistence umożliwia zapisywanie stanu aktorów w bazie danych, co pozwala na odnowienie stanu aplikacji w przypadku awarii i zapewnia w pewnym stopniu niezawodność działania systemu [6]. Dostępnych jest wiele różnych dodatków dedykowanych dla różnych technologii bazodanowych. W niniejszej pracy wykorzystano dodatek przeznaczony do pracy z bazą MongoDB [7].

MongoDB

MongoDB jest bazą NoSQL opartą na dokumentach typu JSON, co pozwala na elastyczność oraz przejrzystość w przechowywaniu danych. Jest to technologia Open Source. Jako, że jest to baza NoSQL posiada ona elastyczny schemat danych. Wspiera replikację oraz sharding, a także map reduce. Technologia ta została wykorzystana ze względu na jej łatwe użycie, dobrą skalowalność oraz wydajność przy wykorzystaniu stosunkowo niewielkich zasobów obliczeniowych [21].

Kolejnymi argumentami przemawiającymi na korzyść tej technologii są łatwa dostępność dokumentacji oraz integracja z innymi technologiami wykorzystanymi w niniejszej pracy.

Play Framework i AngularJS

Technologie Play Framework oraz AngularJS zostały wykorzystane do przygotowania aplikacji internetowej będącej interfejsem klienta dzięki któremu może sterować on działaniem platformy, monitorować ją oraz oglądać wyniki obliczeń.

Play Framework jest to technologia server-side, działająca zarówno z językiem Java jak i Scala. Integruje ona komponenty oraz API potrzebne do stworzenia aplikacji klienckiej opartej o wzorzec MVC. Bazuje on na lekkiej, bezstanowej architekturze, niskim zużyciu zasobów, wysokiej skalowalności oraz programowaniu reaktywnym. Do komunikacji wykorzystuje metody protokołu HTTP oraz pozwala na implementację serwisów w technologii REST. Play zapewnia również wsparcie dla technologii WebSocket wykorzystanej w niniejszej pracy oraz łatwą integrację z frameworkiem Akka [24, 80].

Z kolei AngularJS jest to technologia stworzona z użyciem języka JavaScript, ze wsparciem komercyjnych organizacji takich jak Google. Rozszerza możliwości języka HTML oraz CSS a ponadto pozwala tworzyć tzw. SPA (ang. Single Page Applications). W niniejszej pracy wykorzystano ją do prezentowania informacji odebranych z aplikacji serwerowej oraz interakcji z użytkownikiem za pomocą przeglądarki internetowej [20, 67].

Wykorzystane szablony

Platforma Typesafe w której skład wchodzą technologie wykorzystane w niniejszej pracy, takie jak Play Framework, Akka oraz Scala dostarcza bardzo wyczerpującą dokumentację wraz z wieloma przykładami oraz szablonami na licencji *Public domain* [45].

Rozwiązanie stworzone na potrzeby niniejszej pracy bazowało początkowo na dwóch szablonach, pierwszy zawierał przykład monitoringu urządzeń podłączonych do klastra natomiast drugi dystrybucję zadań pomiędzy urządzenia [42, 56].

5. Możliwości praktycznego zastosowania

Dzięki zaimplementowaniu algorytmów ewolucyjnych w modelu Master/Slave przygotowana platforma może zostać wykorzystana m.in. do symulacji, rozwiązywania problemów optymalizacyjnych, przyśpieszenia obliczeń krytycznych, przeszukiwania dużych zbiorów rozwiązań, szeregowania zadań oraz wszelkich obliczeń, gdzie zadanie może zostać podzielone na kilka niezależnych części wykonywanych równolegle na każdym z workerów. Dzięki wykorzystaniu urządzeń SoC, które pobierają stosunkowo niewielką ilość prądu, zestawienie platformy jest możliwe w warunkach, gdzie dostęp do energii elektrycznej jest mocno ograniczony poprzez wykorzystanie np. zasilania baterijnego.

Działanie platformy przetestowano dzięki optymalizacji funkcji wielomodalnej Rastrigina. Poniższy rozdział zawiera wyniki eksperymentów z uwzględnieniem takich cech stworzonej platformy jak m.in. skalowalności czy wydajność obliczeń.

5.1. Równoległe algorytmy ewolucyjne

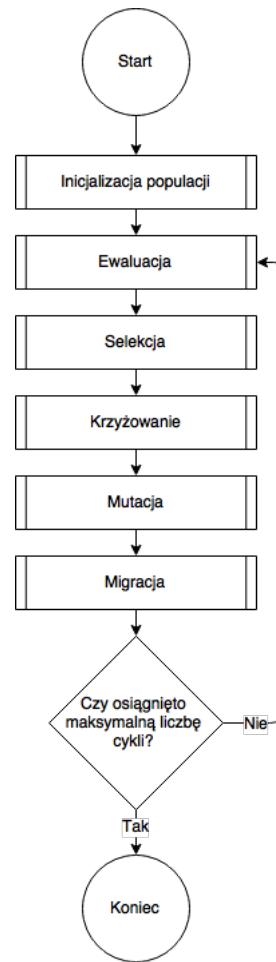
W rozdziale opisano odmianę algorytmów ewolucyjnych wykorzystaną do obliczeń równoległych na przygotowanej platformie.

Algorytmy ewolucyjne

Algorytmy ewolucyjne zostały opracowane przez Johna Hollanda w latach siedemdziesiątych i są inspirowane teorią ewolucji Darwina tedy ewolucją biologiczną [64]. Istnieje wiele różnych prac traktujących o wykorzystaniu algorytmów ewolucyjnych, różnych ich odmianach czy modyfikacjach oraz badaniu ich efektywności. Przykładem może być praca magisterska zawarta w pozycji [64] lub trochę starsza, klasyczna pozycja książkowa [70]. Dokładny opis implementacji algorytmów ewolucyjnych można znaleźć w pozycji [84].

Pojęcia

- Gen - cecha mająca wpływ na jakość rozwiązania.
- Chromosom - indywidual, osobnik prezentujący jedno z rozwiązań problemu, składa się z genów.
- Populacja - zbiór chromosomów, prezentuje zbiór rozwiązań danego problemu.
- Migracja - wymiana osobników pomiędzy populacjami.
- Selekcja - algorytm wyboru osobników pozostałych w danej populacji w następnym cyklu.
- Krzyżowanie - tworzenie osobników potomnych w trakcie trwania ewolucji na podstawie innych, wybranych osobników - rodziców.
- Mutacja - modyfikacja osobnika zawierająca element losowości, mająca na celu próbę eksploracji nowych obszarów rozwiązań.
- Funkcja przystosowania (ang. fitness) - zwana też funkcją celu, określa jakość danego rozwiązania.



Rysunek 5.1: Schemat blokowy algorytmu ewolucji.

- Ewaluacja - wyliczenie wartości funkcji fitness dla każdego osobnika populacji czyli inaczej ocena jakości znalezionych rozwiązań.
- Ewolucja - powtarzający się cykl w trakcie którego następuje rozwój populacji oraz w skład którego wchodzi m.in. ewaluacja, selekcja, krzyżowanie, mutacja oraz migracja.
- Generacja - jeden cykl algorytmu.

Na rysunku 5.1 przedstawiono schemat blokowy ewolucji.

Zastosowanie

Algorytmy ewolucyjne sprawdzają się tam gdzie nie jest dobrze znana przestrzeń rozwiązań ale został określony sposób oceny jakości rozwiązania. Znajdują one zastosowanie przy rozwiązywaniu problemów NP, np. problemu komiwojażera w którym trzeba znaleźć najkrótszą drogę łączącą wszystkie miasta tak, aby odwiedzić każde miasto co najmniej raz. Innymi zastosowaniami mogą być chociażby poszukiwanie ekstremów funkcji, których nie da się obliczyć analitycznie lub przeszukiwanie dużych przestrzeni rozwiązań jak np. w przypadku problemu grupowania. Algorytmy ewolucyjne są mniej zależne od wstępnej inicjalizacji zadania oraz mniej skłonne do znajdywania rozwiązań lokalnych zamiast optymalnych [10, 64].

Model wyspowy

Jest to odmiana algorytmów ewolucyjnych przystosowana do obliczeń równoległych oraz rozproszonych. Zamiast jednej dużej populacji rozważamy tutaj kilka podpopulacji zwanych wyspami. Każda wyspa może być traktowana jako osobna populacja, ponieważ ewolucja na niej zachodzi w izolacji od pozostałych wysp. Co pewien okres populacje znajdujące się na wyspach wymieniają się między sobą pewną ilością osobników w procesie migracji. Pozwala to przyśpieszyć znajdywanie rozwiązania. Istotna jest tutaj topologia połączeń pomiędzy wyspami pozwalająca na wymianę osobników, metoda selekcji migrujących osobników, ich wielkość oraz odstęp pomiędzy kolejnymi migracjami. Badania różnych parametrów tych zjawisk można znaleźć m.in. w pracy [64].

5.2. Problemy benchmarkowe

Algorytmy ewolucyjne ze względu na kilka ich cech, m.in. możliwość znalezienia wielu optymalnych rozwiązań są często wykorzystywane do optymalizacji wielomodalnej.

W problemach wielomodalnych zdarza się, że funkcja celu nie uwzględnia wszystkich elementów mających wpływ na jakość rozwiązania. Spowodowane jest to zazwyczaj pewnymi uproszczeniami, które trzeba przyjąć implementując model algorytmów ewolucyjnych oraz istnieniem cech niemierzalnych. Dlatego istotnym jest, aby algorytm optymalizacyjny był w stanie znaleźć wiele równie dobrych rozwiązań. Odbywa się to dzięki wprowadzeniu czynnika losowego, który pozwala utrzymywać różnorodność populacji i zapobiega znajdywaniu rozwiązań optymalnych lokalnie zamiast globalnie [81].

Istnieje grupa funkcji, które są wykorzystywane jako tzw. benchmarki w optymalizacji wielomodalnej, jedną z nich jest wielowymiarowa funkcja Rastrigina [82].

Algorytmy ewolucyjne opisane w poprzednim podrozdziale zostały wykorzystane do znalezienia minimum funkcji Rastrigina.

Funkcja Rastrigina jest funkcją niewypukłą. Ze względu na to, że posiada wiele minimów lokalnych oraz jedno globalne dla $x = 0$, $f(x) = 0$ bywa trudna w optymalizacji, ponieważ algorytmy optymalizacyjne mogą utknąć w którymś z lokalnych minimów tracąc szansę na znalezienie minimum globalnego. Funkcja ta często znajduje zatem zastosowanie jako funkcja testująca algorytmy optymalizacyjne. Została ona również użyta jako problem testowy w niniejszej implementacji.

Wzór funkcji Rastrigina przedstawiono poniżej

$$f(x) = An + \sum_{i=1}^n [x_i^2 - A\cos(2\pi x_i)], \quad (5.1)$$

gdzie $A = 10$, $x_i \in [-5, 12]$, n oznacza wymiar funkcji [72].

Wykres funkcji Rastrigina przedstawiono na rysunkach 5.2 oraz 5.3.

5.3. Ewolucyjna optymalizacja na platformie

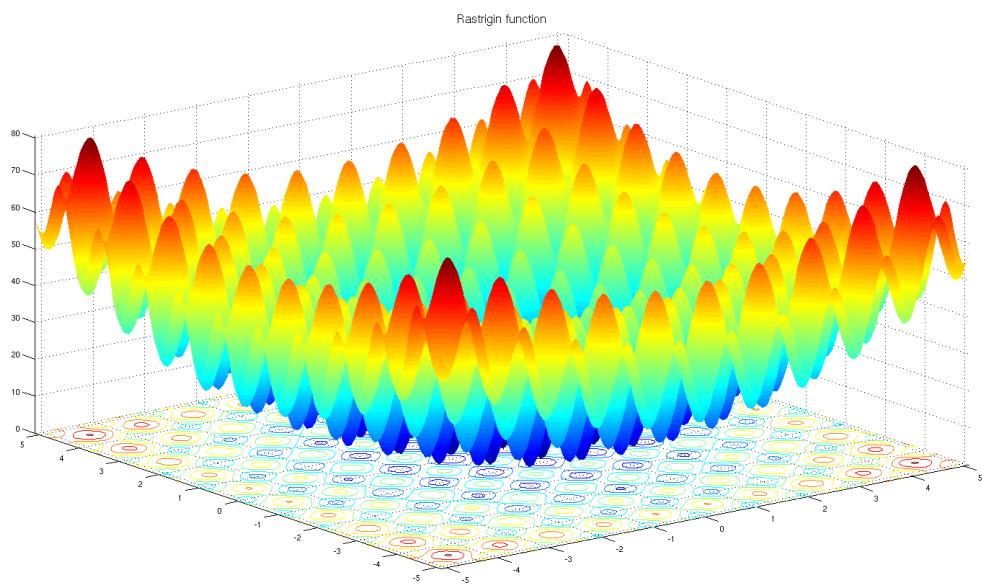
Rozwiązanie stworzone na potrzeby niniejszej pracy przetestowano dzięki zaimplementowaniu algorytmów ewolucyjnych w odmianie wyspowej opisanych w poprzednim rozdziale.

Implementacja ta została oparta na tej przedstawionej w pozycji [77] oraz dostosowana do optymalizacji funkcji ciągłej.

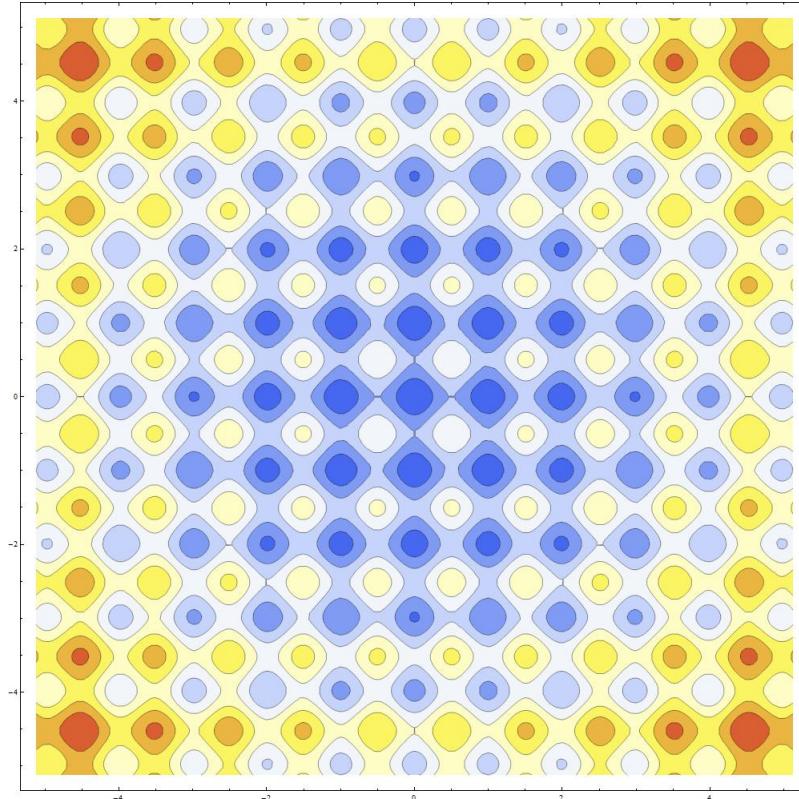
W przypadku problemu Rastrigina gen odpowiada liczbie zmienoprzecinkowej a chromosomy zawierają listę genów, czyli obrazują punkty w przestrzeni rozwiązań. W przypadku dwuwymiarowej funkcji Rastrigina jest to lista dwuelementowa. Natomiast populacja zawiera listę chromosomów.

Zadanie jest konfigurowane parametrami przedstawionymi w tabeli 5.1.

W trakcie awarii urządzenia podczas wykonywania obliczeń dzięki występowaniu migracji nie tracimy wszystkich wyników pracy a jedynie te uzyskane od poprzedniej migracji. Migracja odbywa



Rysunek 5.2: Wykres funkcji Rastrigina 3D [29].



Rysunek 5.3: Wykres funkcji Rastrigina 2D [29].

Tablica 5.1: Parametry zadania.

Nazwa parametru	Opis
Dimension	Wymiar optymalizowanej funkcji ciągłej.
Cycles	Maksymalna ilość cykli ewolucji.
Initial population size	Początkowy rozmiar populacji (podczas krzyżowania i mutacji liczba osobników populacji wzrasta).
Maximum population size	Maksymalny rozmiar populacji.
Snapshot frequency	Częstotliwość persystowania wyników obliczeń (w tym przypadku osobników populacji). *
Migration frequency	Częstotliwość występowania migracji. *
Mutation parameter	Parametr mutacji, określa wpływ mutacji na geny.
Migration factor	Procent migrującej populacji.

* częstotliwość jest określana przez ilość cykli

się poprzez wybranie pewnej ilości najlepszych osobników populacji oraz wysłanie ich do innego węzła klastra. Wykonuje się ona co określoną liczbę cykli konfigurowaną w parametrach zadania. Do wyboru są dwa rodzaje selekcji węzła do którego wysyłani są migrujący osobnicy. Pierwszy to jest wybór losowy a drugi metodą round-robin, czyli wysyłanie migrującej części populacji na zmianę do każdego kolejnego węzła [49]. Migracja odbywa się bez użycia *mastera*, do jej działania wykorzystano rozszerzenie framework'a Akka o nazwie *Distributed Publish Subscribe in Cluster* opisane w jednym z poprzednich rozdziałów.

Inicjalizacja populacji odbywa się w każdym węźle klastra osobno. W przypadku problemu Rastrigina opisanego w kolejnym podrozdziale, początkowa populacja składa się z osobników zainicjalizowanych losowymi liczbami z przedziału $[-5.12, 5.12]$.

Na rysunku 5.4 zaprezentowano wygląd interfejsu użytkownika pozwalającego na podgląd wyników rozpoczętych zadań oraz rozpoczęcie nowych. Widoczne jest tutaj każde z urządzeń wykonujących obliczenia w klastrze identyfikowane po adresie IP oraz ID *worker'a*.

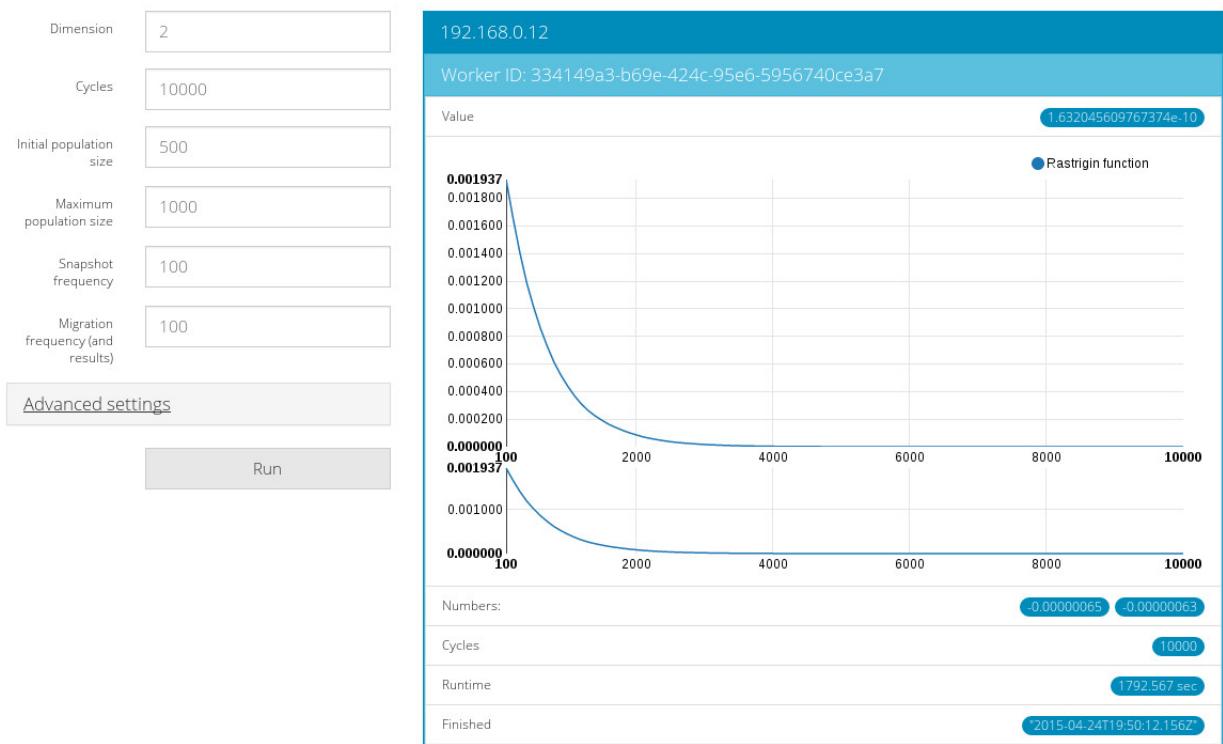
Wyniki prezentowane są w czasie rzeczywistym przy użyciu technologii WebSocket [44]. Dla każdego zadania rysowany jest wykres przedstawiający wartość funkcji Rastrigina zmieniającą się w trakcie trwania ewolucji, czyli w kolejnych cyklach algorytmu. Wykres narysowano za pomocą biblioteki D3.js [18].

5.4. Wyniki testów

Do przeprowadzenia testów posłużono się czterema urządzeniami Raspberry Pi. Jako problem testowy wykorzystano optymalizację wielowymiarowej funkcji Rastrigina opisanej w poprzednim podrozdziale.

Ze względu na ograniczoną precyzję obliczeń nie udało się uzyskać wyniku równego poszukiwanemu minimum, czyli $f(x) = 0$. Najlepsze uzyskane wyniki oscylowały w okolicach $2e - 30$. Ta precyzja odpowiada typowi Double w języku Scala, czyli podwójna precyzja zapisana na 64 bitach [78].

Na poniższych wykresach przedstawiono zmieniającą się wartość funkcji Rastrigina najlepszego osobnika w populacji w trakcie trwania ewolucji oraz wpływ różnych ustawień początkowych zadania na uzyskiwane wyniki. Rozważono tutaj zarówno problem dla jednego urządzenia, jak również wielu urządzeń pracujących równolegle. Dla zwiększenia czytelności wykresów i pokazania szybkości uzyskiwania zbieżności algorytmu w większości przypadków użyto skali logarytmicznej. Domyslnie współczynnik mutacji mu wynosił 0,96. Rozważono różne wielkości populacji, wymiary funkcji oraz ilości urządzeń. Zbadano również wpływ migracji, poprzez zmianę współczynnika migracji określającego ilość migrującej populacji jak i częstotliwość jej występowania. Częstotliwość wysyłania wyników częstociowych odpowiada częstotliwości występowania migracji.



Rysunek 5.4: Rozpoczęcie zadania.

Tablica 5.2: Parametry zadania do rysunku 5.5.

Nazwa parametru	Wartość
Dimension	10
Cycles	1000
Initial population size	10, 100 *
Maximum population size	10, 100 *
Snapshot frequency	10
Migration frequency	10
Mutation parameter	0, 96
Migration factor	0

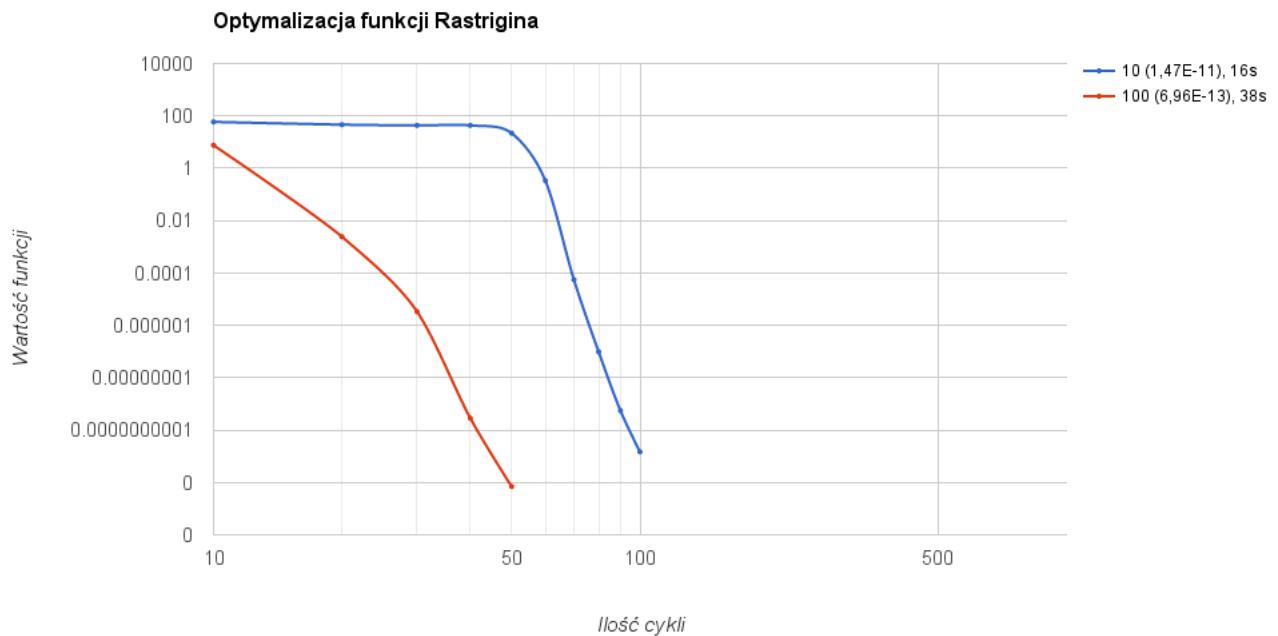
* parametry zmienne podczas eksperymentu

Na wykresach wskazano również czas wykonania algorytmu oraz znalezione minimum. Maksymalna ilość cykli była zazwyczaj ustawiona na 1000. Na legendzie wykresów jako pierwszą liczbę podano wielkość populacji jeżeli była ona zmienna w trakcie eksperymentu oraz numer urządzenia jeżeli w trakcie eksperymentu każde urządzenie posiadało tą samą wielkość populacji a na wykresie przedstawiono razem kilka z nich, z kolei obok w nawiasie podano znalezione minimum. Następnie podano również czas wykonania algorytmu wyrażony w sekundach. Dla potrzeb testów założono, że jedno urządzenie odpowiada jednej wyspie w algorytmie ewolucyjnym.

Wykresy na rysunkach 5.5 oraz 5.6 pokazują, że większa populacja sprzyja znajdywaniu minimum.

Na rysunku 5.7 przedstawiono wpływ współczynnika mutacji na wyniki ewolucji. Najlepsze wyniki uzyskano gdy współczynnik mutacji był maksymalny czyli wynosił 1.0, co odzwierciedlało duży wpływ wartości losowej na osobników populacji w kolejnych cyklach ewolucji.

Wykresy na rysunkach 5.8 oraz 5.9 pokazują wpływ braku migracji na znajdywane wyniki. Gdy migracja nie występuje mimo, że jedno z urządzeń zacznie ewoluować dużo szybciej od innych, nie wpłynie to w żaden sposób na ewolucję odbywającą się na innych urządzeniach.



Rysunek 5.5: Jedno urządzenie. Wymiar funkcji 10. Wyniki co 10 cykli.

Tablica 5.3: Parametry zadania do rysunku 5.6.

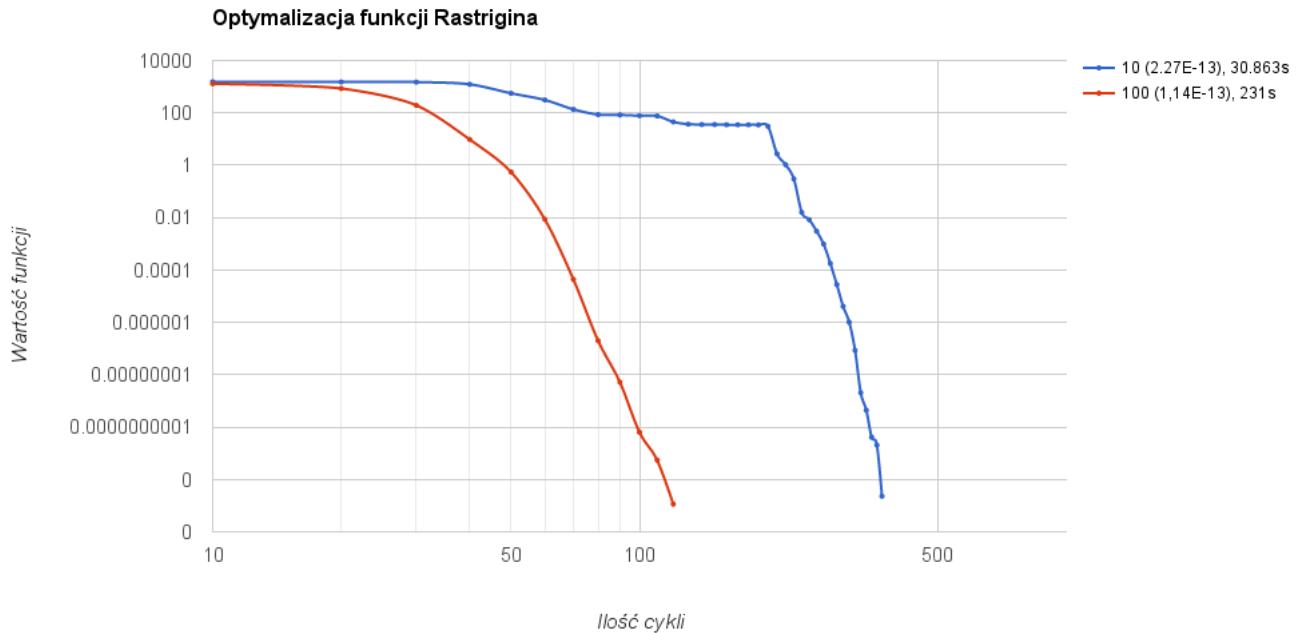
Nazwa parametru	Wartość
Dimension	100
Cycles	1000
Initial population size	10, 100 *
Maximum population size	10, 100 *
Snapshot frequency	10
Migration frequency	10
Mutation parameter	0, 96
Migration factor	0

* parametry zmienne podczas eksperymentu

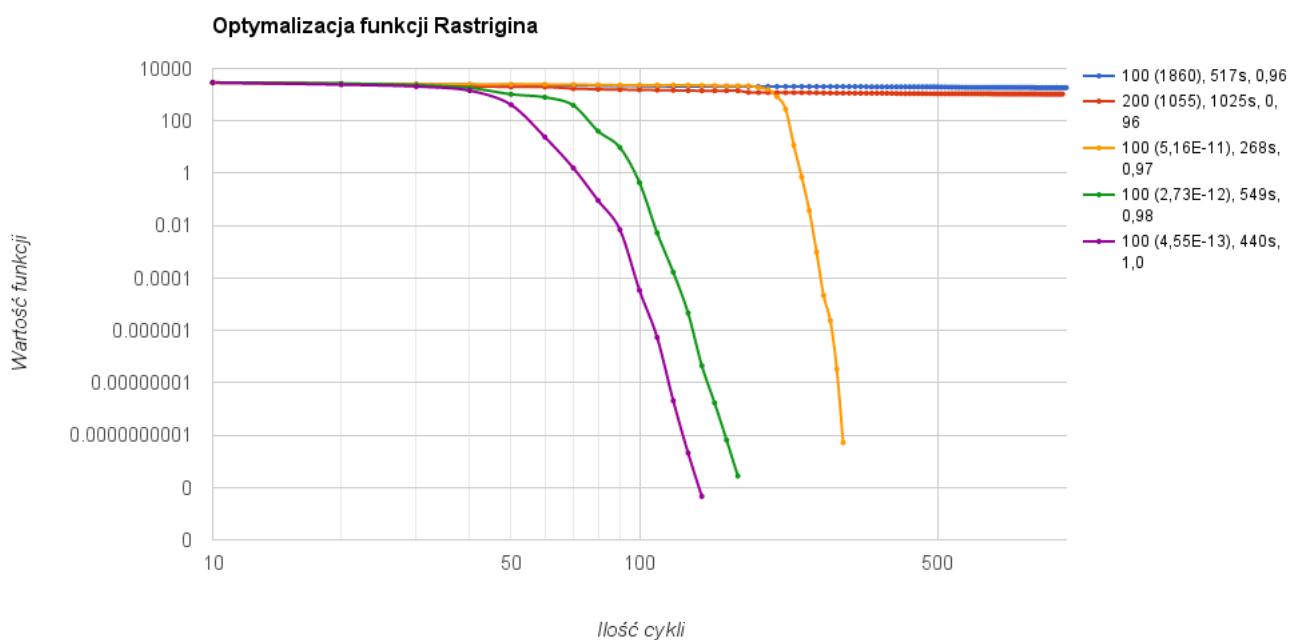
Tablica 5.4: Parametry zadania do rysunku 5.7.

Nazwa parametru	Wartość
Dimension	200
Cycles	1000
Initial population size	100, 200 *
Maximum population size	100, 200 *
Snapshot frequency	10
Migration frequency	10
Mutation parameter	0, 96; 0, 97; 0, 96; 1, 0 *
Migration factor	0

* parametry zmienne podczas eksperymentu



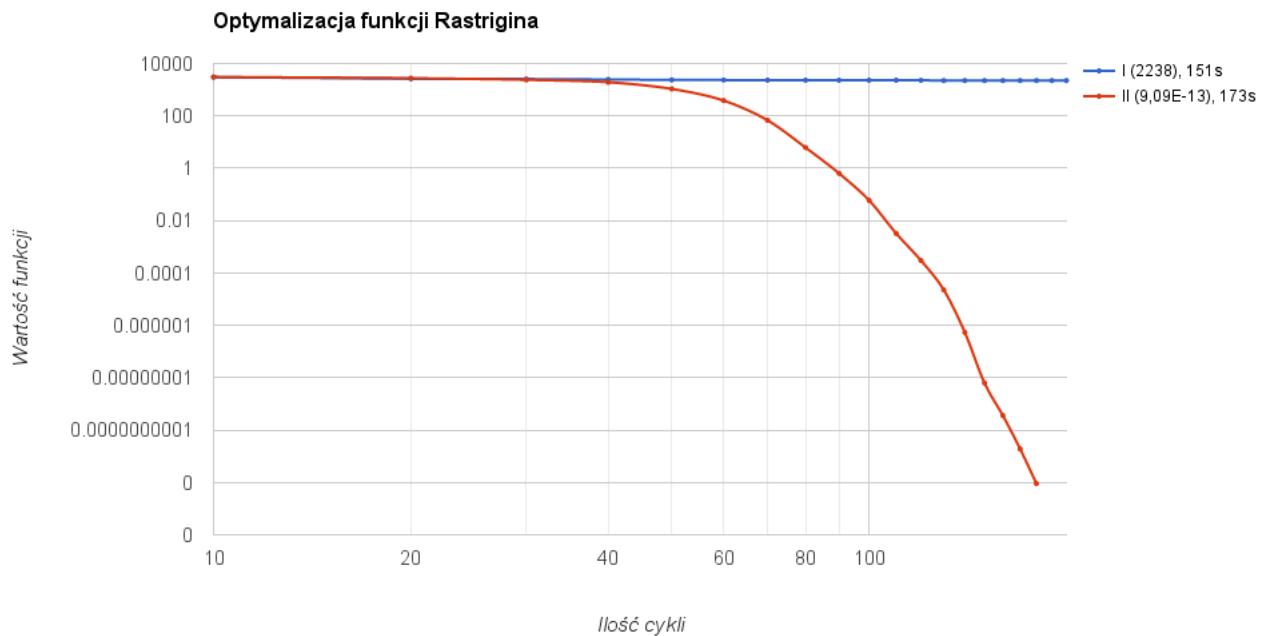
Rysunek 5.6: Jedno urządzenie. Wymiar funkcji 100. Wyniki co 10 cykli.



Rysunek 5.7: Jedno urządzenie. Zmienny wsp. mutacji. Wymiar funkcji 200. Wyniki co 10 cykli.

Tablica 5.5: Parametry zadania do rysunku 5.8.

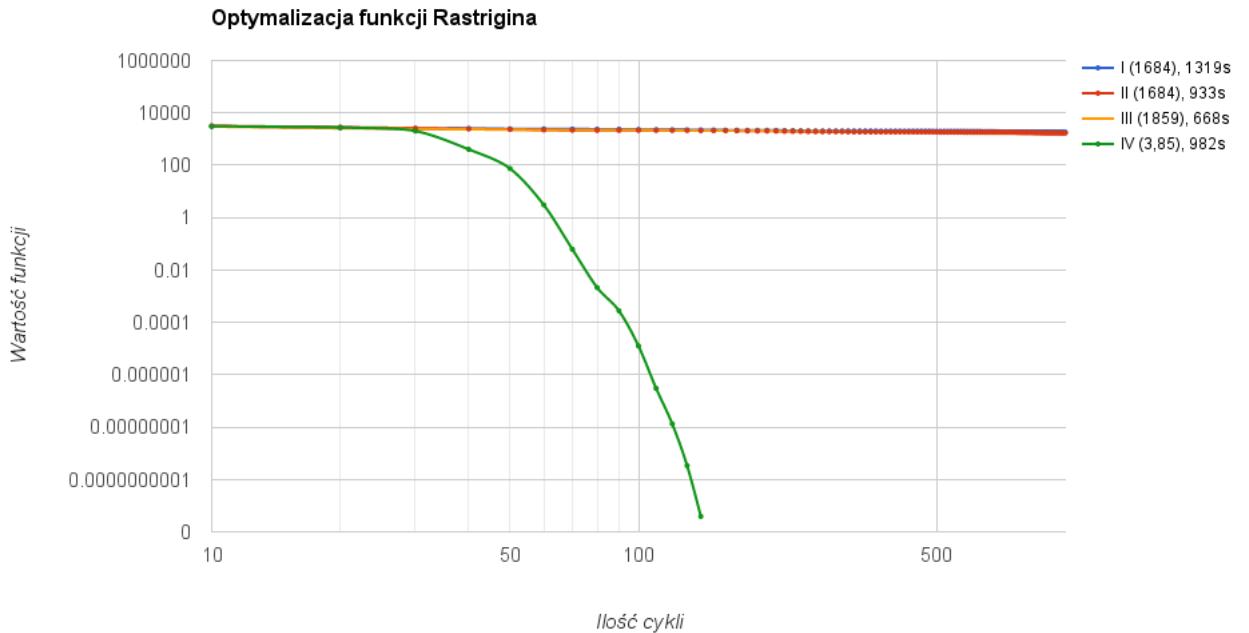
Nazwa parametru	Wartość
Dimension	200
Cycles	200
Initial population size	100
Maximum population size	100
Snapshot frequency	10
Migration frequency	10
Mutation parameter	0,96
Migration factor	0



Rysunek 5.8: Dwa urządzenia. Wymiar funkcji 200. Wyniki co 10 cykli. Brak migracji.

Tablica 5.6: Parametry zadania do rysunku 5.9.

Nazwa parametru	Wartość
Dimension	200
Cycles	1000
Initial population size	100
Maximum population size	100
Snapshot frequency	10
Migration frequency	10
Mutation parameter	0,96
Migration factor	0



Rysunek 5.9: Cztery urządzenia. Wymiar funkcji 200. Wyniki co 10 cykli. Brak migracji.

Tablica 5.7: Parametry zadania do rysunku 5.10.

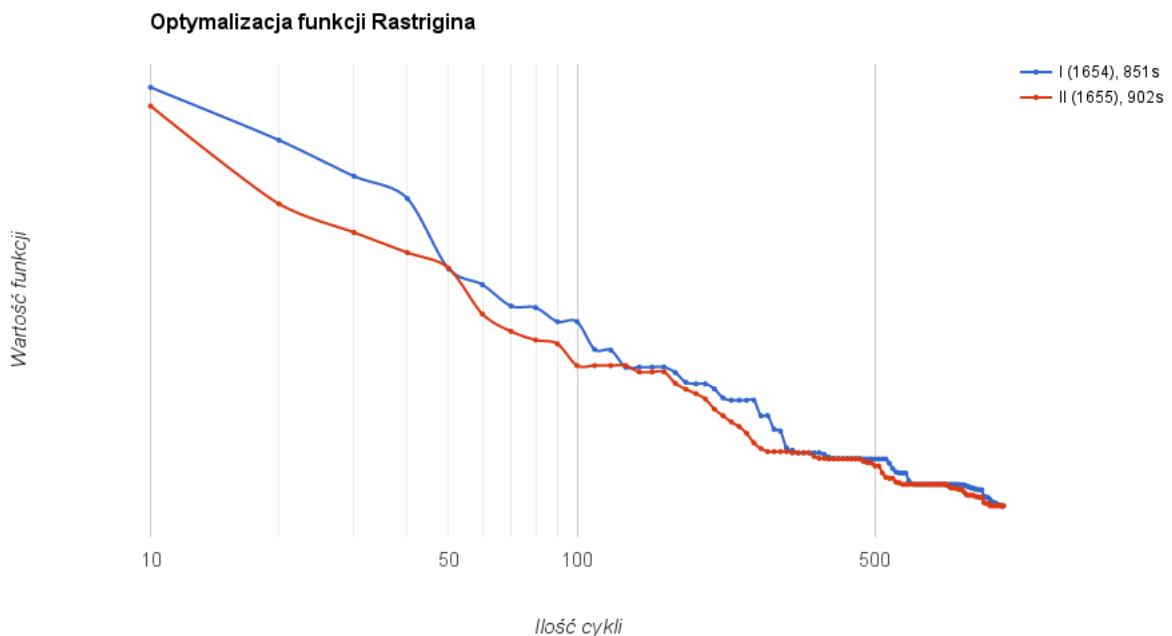
Nazwa parametru	Wartość
Dimension	200
Cycles	1000
Initial population size	100
Maximum population size	100
Snapshot frequency	10
Migration frequency	10
Mutation parameter	0, 96
Migration factor	10

Na rysunku 5.10 przedstawiono dwa urządzenia ze współczynnikiem migracji wynoszącym 10. Oznacza to, biorąc pod uwagę fakt, że maksymalna wielkość populacji wynosi 100 a częstotliwość przesyłania wyników 10, iż co każde dziesięć cykli dziesięciu najlepszych osobników populacji zostanie wysłanych do sąsiedniej wyspy. Duża ilość migrujących osobników ma wpływ na brak zróżnicowania populacji na wyspach, co może skutkować małą zbieżnością algorytmu. Wpływ osobników migrujących do sąsiedniej wyspy jest przez to bardzo duży, co obrazują podobne wykresy wartości funkcji zmieniającej się w trakcie trwania ewolucji na obu urządzeniach.

Wykres na rysunku 5.11 przedstawia dwa eksperymenty z mniejszym współczynnikiem migracji niż poprzednio i wynoszącym 1. Pierwszy eksperyment (zaznaczony odcieniami czerwonego), gdzie w ewolucji biorą udział dwa urządzenia uzyskał gorsze wyniki aniżeli kolejny (zaznaczony odcieniami niebieskiego), w którym udział biorą cztery urządzenia. Sytuacja ta pokazuje pozytywny wpływ większej ilości urządzeń na ewolucję.

Wykresy na rysunkach 5.12 oraz 5.13 pokazują, że poprzez mniej częstą migrację (co 50 cykli) zmiany w sąsiednich wyspach zachodzą wolniej.

Wykresy na rysunkach 5.13 oraz 5.14 pokazują, że optymalizacja funkcji Rastrigina o wymiarze 100



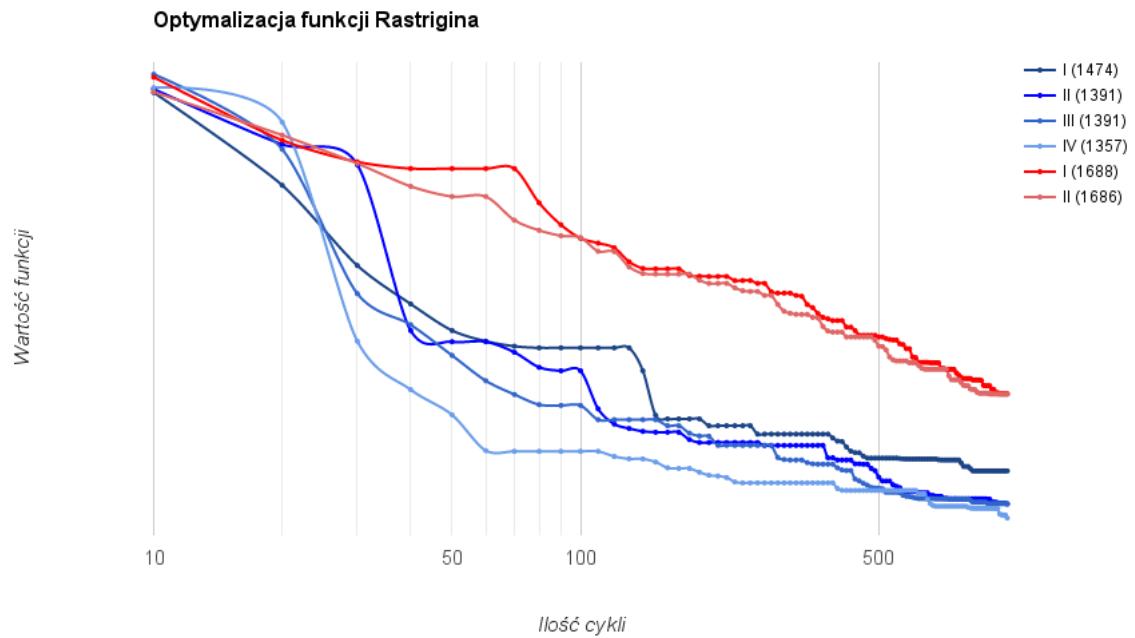
Rysunek 5.10: Dwa urządzenia. Wymiar funkcji 200. Wielkość populacji 100. Wyniki co 10 cykli. Współczynnik migracji 10.

Tablica 5.8: Parametry zadania do rysunku 5.11.

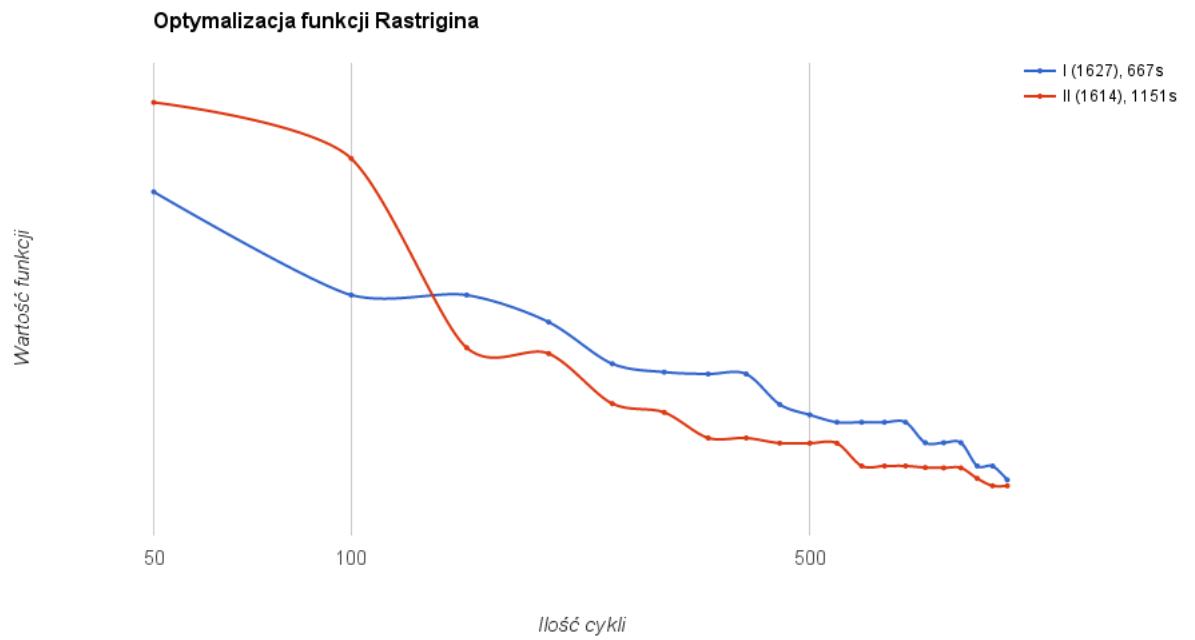
Nazwa parametru	Wartość
Dimension	200
Cycles	1000
Initial population size	100
Maximum population size	100
Snapshot frequency	10
Migration frequency	10
Mutation parameter	0,96
Migration factor	1

Tablica 5.9: Parametry zadania do rysunku 5.12.

Nazwa parametru	Wartość
Dimension	200
Cycles	1000
Initial population size	100
Maximum population size	100
Snapshot frequency	50
Migration frequency	50
Mutation parameter	0,96
Migration factor	1



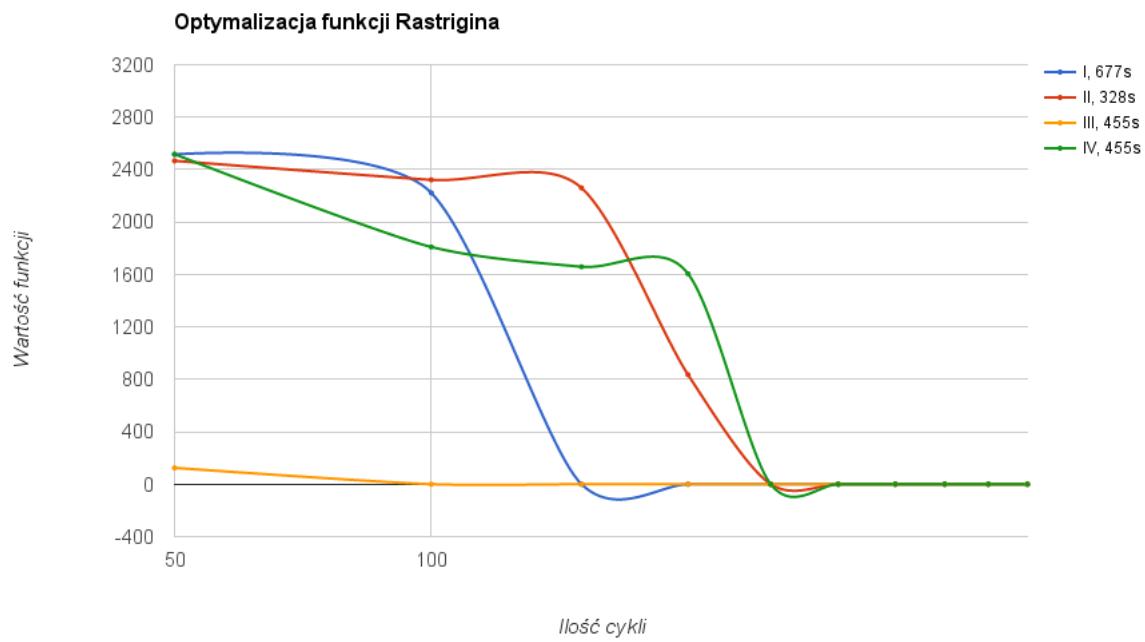
Rysunek 5.11: Cztery urządzenia. Wymiar funkcji 200. Wielkość populacji 100. Wyniki co 10 cykli. Współczynnik migracji 1.



Rysunek 5.12: Dwa urządzenia. Wymiar funkcji 200. Wielkość populacji 100. Wyniki co 50 cykli. Współczynnik migracji 1.

Tablica 5.10: Parametry zadania do rysunku 5.13.

Nazwa parametru	Wartość
Dimension	200
Cycles	500
Initial population size	100
Maximum population size	100
Snapshot frequency	50
Migration frequency	50
Mutation parameter	0,96
Migration factor	1



Rysunek 5.13: Cztery urządzenia. Wymiar funkcji 200. Wielkość populacji 100. Wyniki co 50 cykli. Współczynnik migracji 1.

Tablica 5.11: Parametry zadania do rysunkach 5.14 oraz 5.15.

Nazwa parametru	Wartość
Dimension	100
Cycles	200
Initial population size	100
Maximum population size	100
Snapshot frequency	10
Migration frequency	10
Mutation parameter	0, 96
Migration factor	1

została rozwiązana bardzo szybko, bo w zaledwie stu cyklach. Po zwiększeniu ilości urządzeń z dwóch do czterech, ilość cykli potrzebna do zoptymalizowania funkcji zmniejszyła się prawie o połowę.

Podsumowując, zauważono, że większy współczynnik mutacji przyśpiesza znajdywanie nowych rozwiązań oraz pomaga uzyskać lepszą zbieżność algorytmu. Szybkość znajdywania nowych rozwiązań przyśpiesza również wraz ze wzrostem wielkości populacji.

Dzięki zastosowaniu migracji, gdy do klastra zostaje włączone nowe urządzenie i zaczyna wykonywać obliczenia, to po odebraniu osobników migrujących z innej wyspy, którzy z dużym prawdopodobieństwem będą zawierali lepszej jakości rozwiązania aniżeli nowo zainicjalizowani osobnicy na aktualnej wyspie, ewolucja takiej wyspy zostaje przyśpieszona. Przy wysokiej częstości migracji oraz dużym procencie migrującej populacji istnieje ryzyko, że wyspy zostaną zdominowane przez osobników o podobnych cechach, co może prowadzić do odnalezienia jedynie lokalnego minimum zamiast minimum globalnego oraz wolnej zbieżności.

Warto dodać, że podczas przeprowadzania testów w klastrze pracowały dwa typy urządzeń Raspberry Pi, model B oraz B plus, który jest nowszy od modelu B oraz posiada większą ilość zainstalowanej pamięci RAM [26]. Dostępna w urządzeniach pamięć RAM ma wpływ na czas obliczeń. Czas uzyskiwany przez nowszy model urządzenia, który posiada więcej pamięci był zazwyczaj parę procent lepszy. Również warto zauważać, że urządzenie posiadające rolę *master* było zazwyczaj trochę wolniejsze od pozostałych w związku z większym obciążeniem jakim była potrzeba uruchomienia dodatkowych aktorów.

Skalowalność

W ramach sprawdzenia skalowalności platformy uruchomiono obliczenia z populacją o wielkości 400 osobników na jednym urządzeniu, następnie rozmieszczono po 200 osobników na dwóch urządzeniach działających równolegle, a na końcu zmierzono czas wykonania takiej samej ilości cykli na czterech urządzeniach zawierających po 100 osobników w populacji.

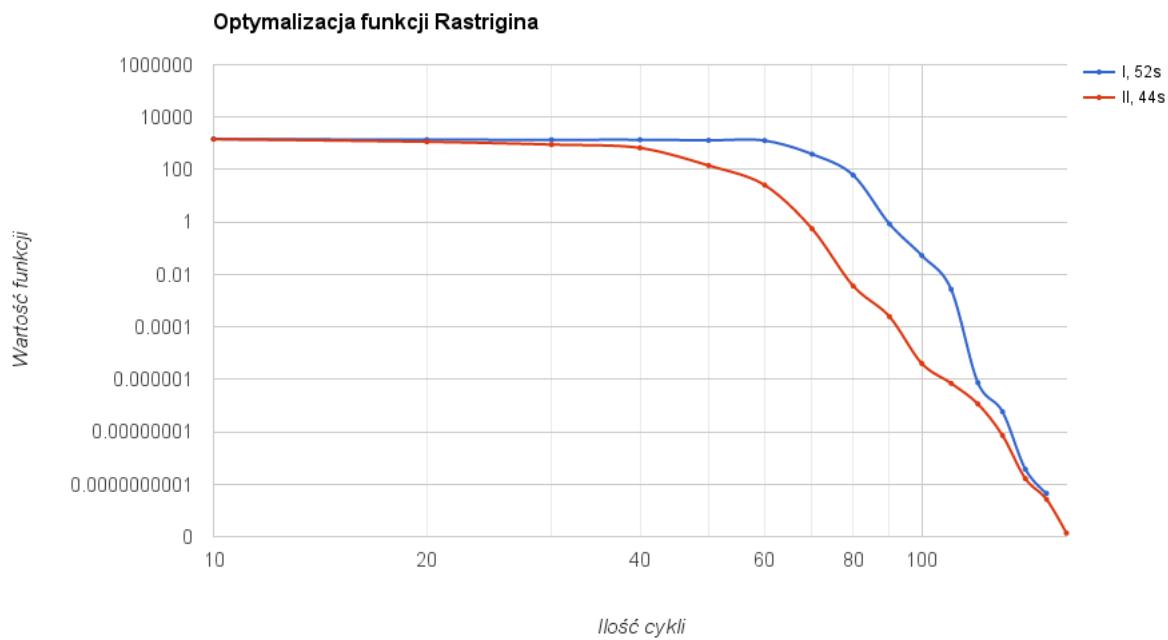
Na rysunkach 5.16 oraz 5.17 pokazano czasy wykonania poszczególnych wariantów. Na drugim wykresie zwiększo wielkość problemu poprzez podwojenie wymiaru funkcji Rastrigina, przez co uzyskano jeszcze większe przyśpieszenie.

Poniżej przedstawiono wzory na skuteczność oraz przyśpieszenie, według których oceniono skalowalność powstałej platformy w kontekście pracy na wielu urządzeniach.

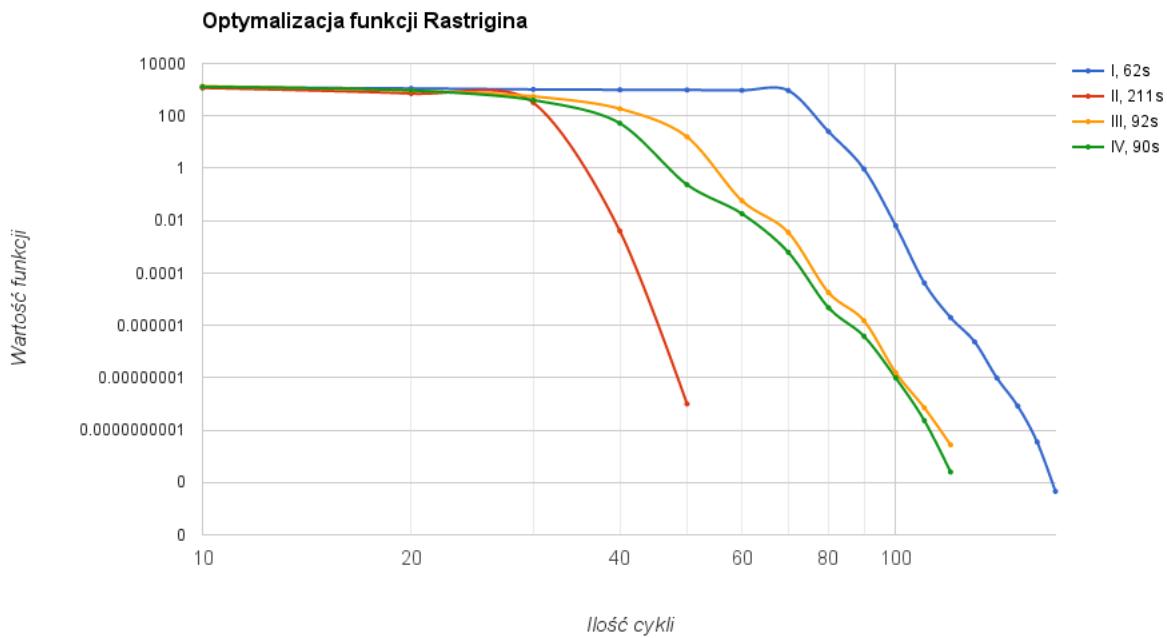
$$S = \frac{T_s}{T_p}, \quad (5.2)$$

gdzie S to jest przyśpieszenie, T_s oznacza czas wykonania programu w wersji sekwencyjnej a T_p w wersji równoległej.

$$E = \frac{S}{P}, \quad (5.3)$$



Rysunek 5.14: Dwa urządzenia. Wymiar funkcji 100. Wielkość populacji 100. Wyniki co 10 cykli. Współczynnik migracji 1.



Rysunek 5.15: Cztery urządzenia. Wymiar funkcji 100. Wielkość populacji 100. Wyniki co 10 cykli. Współczynnik migracji 1.

Tablica 5.12: Parametry zadania do rysunkach 5.16 oraz 5.17.

Nazwa parametru	Wartość
Dimension	100
Cycles	100
Initial population size	100, 200, 400 *
Maximum population size	100, 200, 400 *
Snapshot frequency	10
Migration frequency	10
Mutation parameter	0, 96
Migration factor	0, 1 *

* parametry zmienne podczas eksperymentu

gdzie E oznacza skuteczność, S przyśpieszenie, natomiast P ilość użytych procesorów, w tym przypadku urządzeń.

Zgodnie z prawem Amdahla skuteczność powinna wynosić między 0 a 1, czyli dla dwóch urządzeń nie powinniśmy uzyskać większego przyśpieszenia niż dwukrotne. Wyjątkiem są przypadki gdy brakuje pamięci operacyjnej i istnieje potrzeba użycia pamięci *cache* [74].

Dla pierwszego przypadku i problemu dwóch urządzeń oraz wymiaru funkcji 100 skuteczność wyniosła 0,86, a przyśpieszenie 1,73, co jest dobrym wynikiem i oznacza, że szybkość wykonania tysiąca cykli algorytmu wzrosła prawie dwukrotnie po dołożeniu drugiego urządzenia, a przyśpieszenie było prawie liniowe. Połączeniu kolejnych dwóch urządzeń przyśpieszenie wyniosło 2,73, a skuteczność 0,68, co jest wynikiem troszkę gorszym, oznacza to, że cztery urządzenia były tylko około dwa i pół razy lepsze od jednego. Spowodowane jest to zapewne tym, że problem wybrany w tym przypadku nie był zbytnio wymagający, dlatego do następnego eksperymentu zdecydowano się zwiększyć rozmiar funkcji do 200.

W kolejnym eksperymencie, którego wyniki zaprezentowano na rysunku 5.17, dla dwóch urządzeń przyśpieszenie wyniosło 2,33 a skuteczność 1,16 a dla czterech przyśpieszenie wyniosło 4,24 natomiast skuteczność 1,06. Co oznacza, że wraz ze wzrostem liczby urządzeń przyśpieszenie nieznacznie maleje, natomiast uzyskano tutaj przyśpieszenie ponad liniowe, co jest wynikiem trochę zaskakującym biorąc pod uwagę prawo Amdahla [74], lecz można tłumaczyć go faktem, iż w przypadku z tylko jednym urządzeniem komunikacja nie była obecna.

Wykorzystanie zasobów sprzętowych

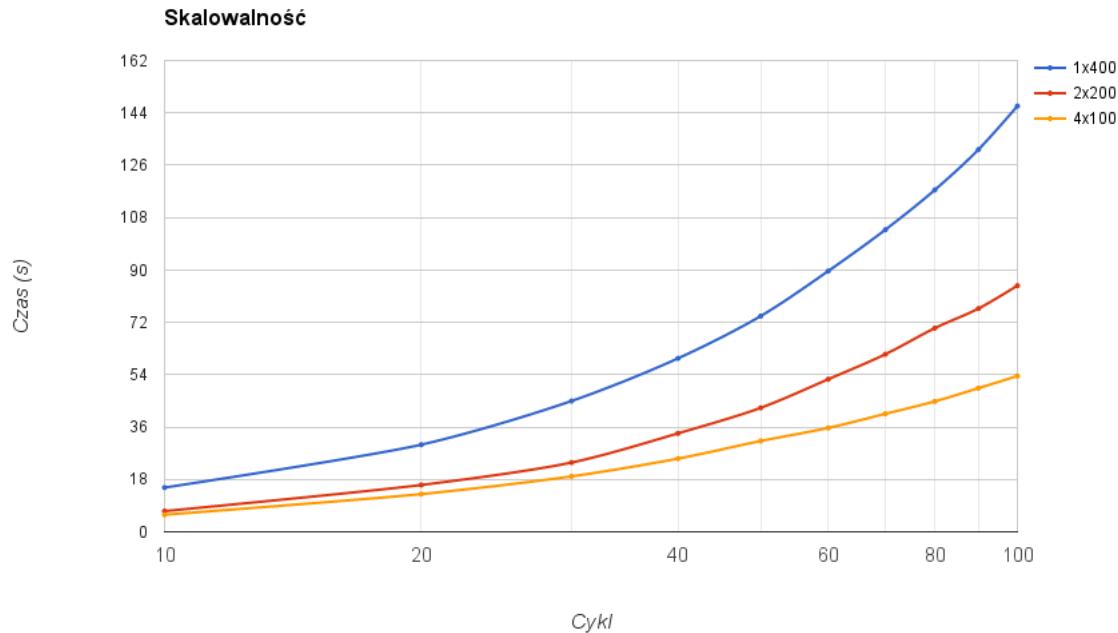
Na wykresie 5.18 zaprezentowano zużycie pamięci RAM oraz obciążenie procesora w trakcie wykonywania obliczeń oraz w stanie bezczynności.

W trakcie obciążenia procesor był wykorzystywany prawie w stu procentach, natomiast w odwrotnej sytuacji jego obciążenie oscylowało w okolicy piętnastu procent. Zwracając uwagę na pamięć RAM można zauważać, że w trakcie wykonywania zadania zużycie pamięci rosło do 70%, cykliczne spadki zużycia były spowodowane zapewne włączaniem się GC. W trakcie bezczynności również widać pewną okresowość w zwalnianiu użytych zasobów związaną zapewne z aktywnością GC.

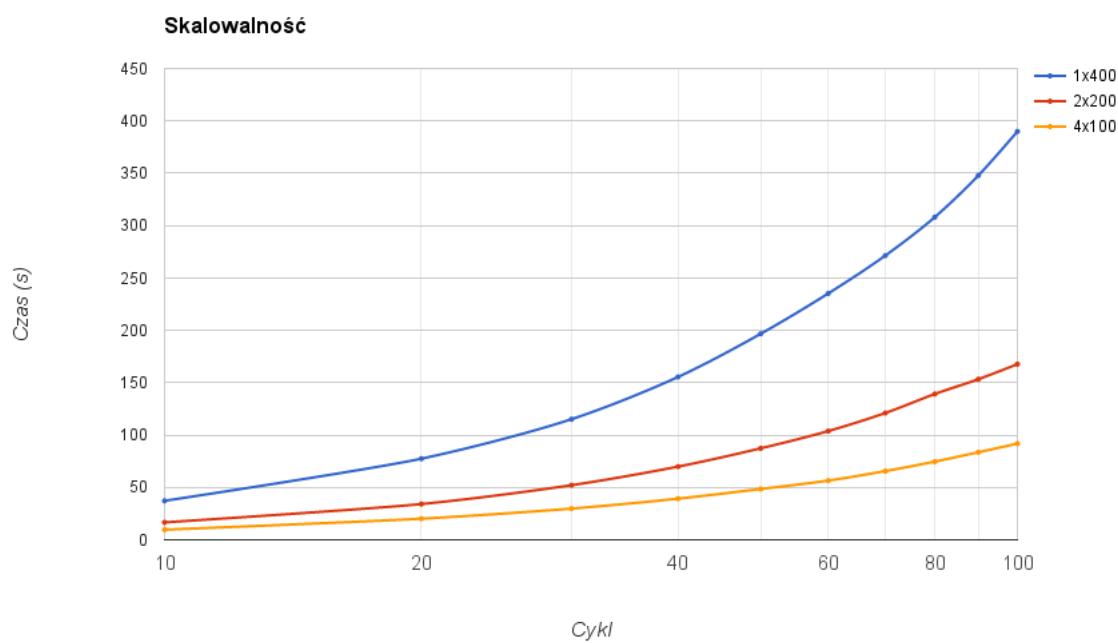
Narzut komunikacji

Na rysunku 5.19 przedstawiono wykres zależności częstotliwości migracji oraz wielkości migrującej populacji. Przetestowano współczynnik migracji wynoszący 1 oraz 10 a także częstotliwość przesyłania wyników wynoszącą 10 oraz 100.

Częstotliwość migracji, która odpowiada również częstotliwości przesyłania wyników częściowych do klienta (podczas których wykonywane są również zapisy do bazy danych) ma widoczny wpływ na czas trwania ewolucji. Wielkość migrującej populacji ma natomiast nieznaczny wpływ na czas ewolucji, ponieważ migracja odbywa się asynchronicznie i zazwyczaj odbywa się stosunkowo rzadko i w małych



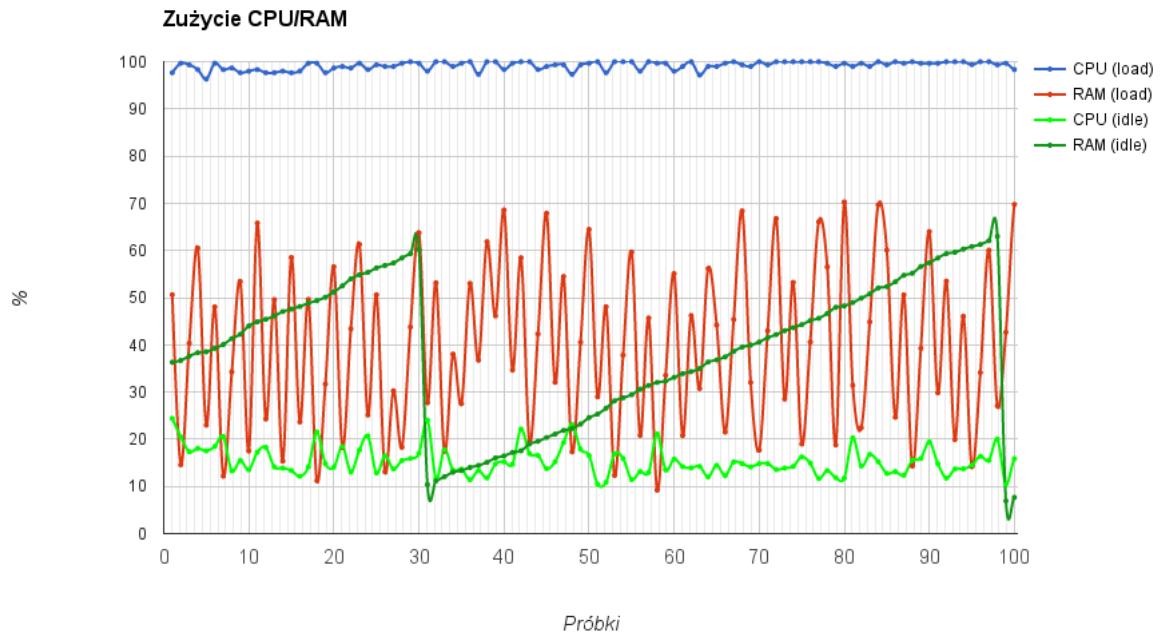
Rysunek 5.16: Wymiar funkcji 100.



Rysunek 5.17: Wymiar funkcji 200.

Tablica 5.13: Parametry zadania do rysunku 5.18.

Nazwa parametru	Wartość
Dimension	100
Cycles	100
Initial population size	100
Maximum population size	100
Snapshot frequency	10
Migration frequency	10
Mutation parameter	0, 96
Migration factor	0

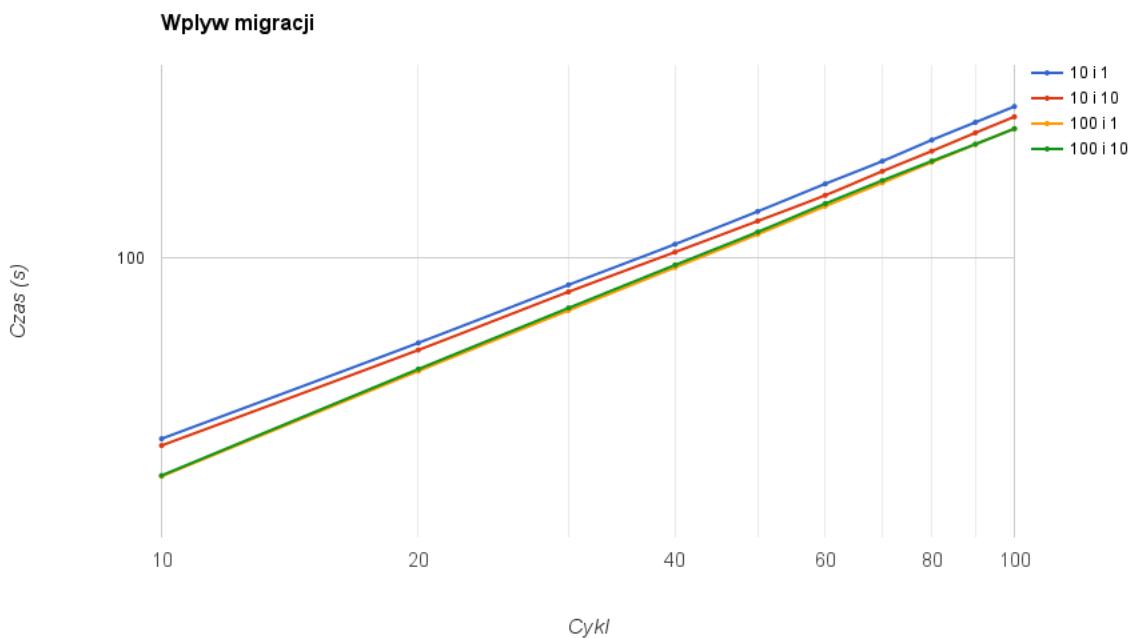


Rysunek 5.18: Wykorzystanie pamięci RAM oraz CPU.

Tablica 5.14: Parametry zadania do rysunku 5.19.

Nazwa parametru	Wartość
Dimension	100
Cycles	1000
Initial population size	100
Maximum population size	100
Snapshot frequency	10, 100 *
Migration frequency	10, 100 *
Mutation parameter	0, 96
Migration factor	1, 10 *

* parametry zmienne podczas eksperymentu



Rysunek 5.19: Zmienne parametry migracji Wymiar funkcji 100.

ilościach więc narzut przesyłania takich wiadomości jest mały. Z kolei całkowita wielkość populacji oraz wymiar funkcji mają duży wpływ na wielkość przesyłanych wiadomości pomiędzy aktorami.

6. Podsumowanie

W ramach niniejszej pracy udało się spełnić początkowe założenia i przygotować dobrze skalowalną platformę oferującą możliwość uruchomienia obliczeń równoległych na wielu urządzeniach pracujących w klastrze. Dostarcza ona narzędzi do monitoringu, rozwiązuje problem load-balancingu zadań, a także zapewnia wsparcie w sytuacji awarii poprzez reorganizację oraz odzyskanie sprawności, czyli powrót do stanu klastra sprzed wystąpienia awarii. Platforma pozwala również na prowadzenie efektywnych obliczeń mimo ograniczonej mocy obliczeniowej urządzeń SoC. Komunikacja, która występuje na platformie odbywa się asynchronicznie, przez co nie powoduje zbyt dużego obciążenia. Wykorzystanie urządzeń oraz technologii współpracujących z maszyną wirtualną Javy zapewnia dużą mobilność tego rozwiązania.

Technologia Akka okazała się kluczowa z punktu widzenia niniejszej pracy i idealnie wkomponowała się w wybraną tematykę. Dzięki rozwiązaniom wysokiego poziomu ułatwia ona tworzenie nowych komponentów oraz dostarczyła pewnych strategii oraz wzorców zwiększąc tym samym przejrzystość jak i możliwości dalszego rozwoju przygotowanego rozwiązania.

Niniejsza praca pozwoliła również autorowi na zapoznanie się z tematyką problemów występujących w klastrach oraz rozwiązaniami stosowanymi w celu zrównoleglenia lub rozproszenia obliczeń. Kolejnymi ciekawymi aspektami niniejszej pracy, które pozwoliły na eksplorację nowych obszarów wiedzy z punktu widzenia autora jest względnie nowa dziedzina w programowaniu, a mianowicie programowanie reaktywne i technologie z platformy Typesafe w połączeniu z urządzeniami SoC oraz Cluster Computing.

Rozwiązanie stworzone na potrzeby niniejszej pracy posiada kilka ciekawych aspektów nad którymi badania mogą być kontynuowane. Kilka z nich zostało opisanych poniżej.

Wiele aplikacji klienckich

W testowanym rozwiążaniu rozważono tylko jedną aplikację kliencką, będącą jedynym źródłem zadań w klastrze. W kolejnych etapach prac nad stworzoną platformą warto byłoby przetestować przypadek gdzie klientów jest więcej i w różnym stopniu obciążają oni urządzenia pracujące w klastrze. Wymagałoby to zaimplementowania sposobu dysponowania wolnymi węzłami tak, aby jeden klient nie zajął wszystkich dostępnych węzłów, podczas gdy reszta nie byłaby w żaden sposób obsługiwana.

Sieć rozległa

Na potrzeby niniejszej pracy rozważono działanie urządzeń klastra w sieci lokalnej lecz przygotowane rozwiązanie mogłyby się również sprawdzić w sieci globalnej Internet, gdzie węzły klastra mogłyby być uruchomione na różnego typu urządzeniach znaczaco oddalonych od siebie i mających do siebie dostęp za pomocą protokołu TCP/IP. Ciekawym aspektem tutaj byłaby zróżnicowana przepustowość połączeń w sieci, ponieważ w takim przypadku należałoby dobrać odpowiednie ustawienia detekcji awarii a arbitralne wybranie dozwolonego czasu opóźnień mogłyby być niełatwym zadaniem.

Wykorzystanie GPU

Do obliczeń wykonywanych w obrębie platformy wykorzystano jedynie procesory CPU lecz większość urządzeń SoC posiada również jednostki GPU, które są w tym czasie niewykorzystywane. Warto byłoby się zastanowić nad potencjalnym wykorzystaniem tych jednostek np. przy użyciu takich bibliotek jak ScalaCL (opartej o OpenCL) [51], które pozwalają na wykonywanie kodu napisanego w Scala na procesorach graficznych i równoległym prowadzeniu obliczeń na różnych architekturach. W roku 2011 Martin Odersky współtwórca języka Scala oraz jego zespół pracujący na uniwersytecie w Lozannie uzyskali wielomilionowy grant od Europejskiej Rady ds. Badań Naukowych (ERBN), dzięki któremu mogli kontynuować pracę nad problemami zrównoleglania obliczeń w różnych modelach programowania jak np. OpenMP, MPI, CUDA, OpenCL [73] co potwierdza, że prace nad podobnymi rozwiązaniami są prowadzone już od wielu lat.

Inne algorytmy

Kolejną kwestią którą można rozważyć w planach dalszego rozwoju jest zaimplementowanie innych algorytmów, które będą w stanie równie dobrze wykorzystać potencjał stworzonej platformy. Warto byłoby również przeprowadzić eksperymenty z różnymi modyfikacjami algorytmów ewolucyjnych, takimi jak inne metody selekcji lub krzyżowania, aby uzyskać jeszcze większą efektywność obliczeń.

Optymalizacja

Urządzenia SoC na których testowano przygotowaną platformę posiadają ograniczone możliwości obliczeniowe, co oznacza, że uruchamiane na nich aplikacje powinny być zoptymalizowane pod tym kątem. Wykorzystanie pamięci RAM oraz procesora jest tutaj bardzo istotne i może w znaczny sposób wpływać na szybkość obliczeń. Nie można zatem pozwolić sobie na zbędne operacje i nadmiernązęjętość pamięci, co może wydłużyć wykonywanie obliczeń. Warto byłoby użyć jakiegoś narzędzia do profilowania JVM, aby sprawdzić czy nie ma żadnych wycieków pamięci, a także przeanalizować zrzut pamięci (ang. heap dump) pod kątem instancjonowanych obiektów oraz zajętości pamięci. Można byłoby również wykonać tuning GC (ang. Garbage Collector) oraz spróbować dobrać optymalne parametry GC.

IoT

W ostatnich latach coraz bardziej wzrasta znaczenie urządzeń podłączonych do Internetu co zostało już wspomniane w rozdziale trzecim. Coraz więcej sprzętu codziennego użytku posiada zainstalowane układy procesorowe o wystarczającej mocy obliczeniowej, aby móc obsłużyć systemy mobilne takie jak Android. Stwarza to zatem możliwości wykorzystania stworzonego rozwiązania w wersji bardziej rozproszonej z użyciem komunikacji bezprzewodowej oraz innych urządzeń.

Bibliografia

- [1] Actor Systems. <http://doc.akka.io/docs/akka/snapshot/general/actor-systems.html>.
- [2] Akka Actor Model. <http://doc.akka.io/docs/akka/snapshot/general/actors.html>.
- [3] Akka Cluster Specification. <http://doc.akka.io/docs/akka/2.3.9/common/cluster.html>.
- [4] Akka Cluster Usage. <http://doc.akka.io/docs/akka/2.3.9/scala/cluster-usage.html>.
- [5] Akka Microkernel. <http://doc.akka.io/docs/akka/2.3.9/scala/microkernel.html>.
- [6] Akka Persistence. <http://doc.akka.io/docs/akka/2.3.9/scala/persistence.html>.
- [7] Akka Persistence Mongo. <https://github.com/ironfish/akka-persistence-mongo>.
- [8] Akka Remoting. <http://doc.akka.io/docs/akka/2.3.9/scala/remoting.html>.
- [9] Akka STM. <http://doc.akka.io/docs/akka/snapshot/general/jmm.html>.
- [10] Algorytm genetyczny. http://pl.wikipedia.org/wiki/Algorytm_genetyczny.
- [11] Amazon AWS HPC. <http://aws.amazon.com/hpc>.
- [12] Apache Hadoop. <https://hadoop.apache.org>.
- [13] Architektura von neumannia. https://pl.wikipedia.org/wiki/Architektura_von_Neumanna.
- [14] Arduino. <https://www.arduino.cc>.
- [15] Big Data. https://en.wikipedia.org/wiki/Big_data.
- [16] Broadband by the numbers. <https://www.ncta.com/broadband-by-the-numbers>.
- [17] Cisco - IoE. <http://newsroom.cisco.com/ioe>.
- [18] D3.js. <http://d3js.org>.
- [19] Distributed Publish Subscribe in Cluster.
<http://doc.akka.io/docs/akka/2.3.9/contrib/distributed-pub-sub.html>.
- [20] Dokumentacja AngularJS. <https://docs.angularjs.org/api>.
- [21] Dokumentacja bazy MongoDB. <http://docs.mongodb.org/manual>.
- [22] Dokumentacja frameworka Akka dla języka Scala. <http://doc.akka.io/docs/akka/2.3.9/scala.html>.
- [23] Dokumentacja języka Scala. <http://www.scala-lang.org/documentation>.
- [24] Dokumentacja Play Framework dla języka Scala.
<https://www.playframework.com/documentation/2.3.x/ScalaHome>.

- [25] Dokumentacja projektu Akka Cluster Singleton.
<http://doc.akka.io/docs/akka/2.3.9/contrib/cluster-singleton.html>.
- [26] Dokumentacja Raspberry Pi. <https://www.raspberrypi.org/documentation>.
- [27] Event Sourcing. <http://martinfowler.com/eaaDev/EventSourcing.html>.
- [28] Event Sourcing Pattern. <https://msdn.microsoft.com/en-us/library/dn589792.aspx>.
- [29] Funkcja Rastrigina. http://en.wikipedia.org/wiki/Rastrigin_function.
- [30] HTCondor. <http://research.cs.wisc.edu/htcondor>.
- [31] Intel Galileo.
<http://www.intel.com/content/www/us/en/embedded/products/galileo/galileo-overview.html>.
- [32] Intel Galileo idzie na studia. <http://iq.intel.pl/intel-galileo-idzie-na-studia>.
- [33] IoT. https://en.wikipedia.org/wiki/Internet_of_Things.
- [34] Java Concurrency Framework. <http://docs.oracle.com/javase/tutorial/essential/concurrency>.
- [35] Lockstep. [https://en.wikipedia.org/wiki/Lockstep_\(computing\)](https://en.wikipedia.org/wiki/Lockstep_(computing)).
- [36] Master/slave. [http://en.wikipedia.org/wiki/Master/slave_\(technology\)](http://en.wikipedia.org/wiki/Master/slave_(technology)).
- [37] MPI. <http://www.mpi-forum.org/docs>.
- [38] Nvidia CUDA. <http://www.nvidia.pl/object/cuda-parallel-computing-pl.html>.
- [39] OpenCL. <https://www.khronos.org/opencl>.
- [40] OpenMP. <http://openmp.org/wp>.
- [41] ParallelA Board. <https://www.parallelia.org/board>.
- [42] Play Akka cluster sample. <https://typesafe.com/activator/template/play-akka-cluster-sample>.
- [43] Protokół Gossip. http://en.wikipedia.org/wiki/Gossip_protocol.
- [44] Protokół WebSocket - RFC6455. <https://tools.ietf.org/html/rfc6455>.
- [45] Public domain. <https://creativecommons.org/licenses/publicdomain>.
- [46] Raspberry Pi 2. <https://www.raspberrypi.org/products/raspberry-pi-2-model-b>.
- [47] Raspberry Pi at Southampton. <http://www.southampton.ac.uk/~sjc/raspberrypi>.
- [48] Reactive Manifesto. <http://www.reactivemanifesto.org>.
- [49] Round-robin. http://en.wikipedia.org/wiki/Round-robin_scheduling.
- [50] SBT Native Packager Plugin. <http://www.scala-sbt.org/sbt-native-packager>.
- [51] ScalaCL. <https://code.google.com/p/scalacl>.
- [52] Sharks Cove. <http://www.sharks Cove.org>.
- [53] Sigar. <https://support.hyperic.com>.
- [54] SoC. http://en.wikipedia.org/wiki/System_on_a_chip.

- [55] Taksonomia Flynn'a. https://pl.wikipedia.org/wiki/Taksonomia_Flynn'a.
- [56] Template Akka distributed workers.
<https://typesafe.com/activator/template/akka-distributed-workers>.
- [57] Type safety. http://en.wikipedia.org/wiki>Type_safety.
- [58] Ubiquitous Computing. <http://www.ubiq.com/hypertext/weiser/UbiHome.html>.
- [59] Work Pulling Pattern. <http://www.michaelpollmeier.com/akka-work-pulling-pattern>.
- [60] T. Alexandre. *Scala for Java Developers*. Packt Publishing, 2014.
- [61] J. Armstrong. *Programming Erlang: Software for a Concurrent World (Pragmatic Programmers)*. The Pragmatic Bookshelf, 2013.
- [62] S. J. Cox. Iridis-pi: a low-cost, compact demonstration cluster. *Cluster Computing*, Czerwiec 2014.
- [63] F. daCosta. *Rethinking the Internet of Things: A Scalable Approach to Connecting Everything*. Apress Open, 2014.
- [64] M. Dudek. Badanie efektywności wielopopulacyjnego algorytmu ewolucyjnego. Akademia Górnictwo-Hutnicza w Krakowie, 2009.
- [65] B. Eckel D. Marsh. *Atomic Scala*. Mindview LLC, Crested Butte, CO, 2013.
- [66] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, 1995.
- [67] A. Freeman. *Pro AngularJS*. Apress, 2014.
- [68] S. Furber. *ARM System-on-Chip Architecture*. Pearson Education, 2000.
- [69] F. Gebali. *Algorithms and Parallel Computing*. A John Wiley & Sons, Inc., Publication, 2011.
- [70] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Longman Publishing Co., Styczeń 1989.
- [71] M. K. Gupta. *Akka Essentials*. Packt Publishing, 2012.
- [72] D. S. H. Mühlenbein J. Born. The Parallel Genetic Algorithm as Function Optimizer. *Parallel Computing*, 17:619–632, 1991.
- [73] C. Hassan, D. Zach, M. Adriaan, R. Tiark, S. Arvind, H. Pat, O. Martin, O. Kunle. Language Virtualization for Heterogeneous Parallel Computing. <http://infoscience.epfl.ch/record/148814>, 2010.
- [74] J. L. Hennessy D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, 2012.
- [75] M. Herlihy N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.
- [76] J. Hruska. The death of cpu scaling: From one core to many - and why we're still stuck. <http://www.extremetech.com/computing/116561-the-death-of-cpu-scaling-from-one-core-to-many-and-why-were-still-stuck>, 2012.
- [77] P. R. Nicolas. *Scala for Machine Learning*. Packt Publishing, Grudzień 2014.

- [78] M. Odersky, L. Spoon, B. Venners. *Programming in Scala, Second Edition*. Artima, Grudzień 2010.
- [79] P. S. Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011.
- [80] J. Richard-Foy. *Play Framework Essentials*. Packt Publishing, 2014.
- [81] A. Słowik. WŁaściwości i zastosowania algorytmów ewolucyjnych w optymalizacji. *Metody Informatyki Stosowanej, Kwartalnik Komisji Informatyki Polskiej Akademii Nauk Oddział w Gdańsku*, 2, 2007.
- [82] P. N. Suganthan, N. Hansen, J. J. Liang, K. Deb, Y. P. Chen, A. Auger, S. Tiwari. Problem definitions and evaluation criteria for the cec 2005 special session on real-parameter optimization. *IEEE Congress on Evolutionary Computation*, 2005.
- [83] M. Weiser. The Computer for the Twenty-First Century. *Scientific American*, pp. 94–10, 1991.
- [84] D. Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4:65–85, 1994.
- [85] Wikipedia. RSS. <http://pl.wikipedia.org/wiki/RSS>.