

POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: INFORMATYKA
SPECJALNOŚĆ: INŻYNIERIA INTERNETOWA
KURS: APLIKACJE INTERNETOWE I ROZPROSZONE

GRAVISIM
Dokumentacja projektowa

AUTORZY:

Adam Dłubak
Jakub Błaszczuk

PROWADZĄCY:

Dr inż. Marek Woda

WROCŁAW, 2017

Rozdział 1

Wprowadzenie

GRAVIsim to narzędzie umożliwiające przeprowadzanie astrofizycznej symulacji grawitacyjnej N -ciał zaimplementowane w systemie rozproszonym. Interfejsem symulacji jest aplikacja internetowa pozwalająca autoryzowanym członkom instytutu badawczego na zlecenie oraz analizowanie (interpretowanie) zadań badawczych.

1.1. Pomysł, motywacja, stan rynku

Pomysł stworzenia GRAVIsim zrodził się w głowach autorów poprzez ich zamięrowanie do astrofizyki. Operowanie na systemach rozproszonych doskonale wpasowuje się w tematykę przetwarzania dużej ilości informacji, potrzebnej do przeprowadzenia symulacji zderzeń kosmicznych.

Z racji, iż na ogólnodostępnym rynku brak jest aplikacji wykonującej podobne zadania, autorzy GRAVIsim mieli silną motywację do stworzenia narzędzia, które mogłoby zobrazować zjawisko zderzeń astrofizycznych występujących w kosmosie, jednak niedostrzegalnych dla człowieka.

Dzięki stworzeniu środowiska pozwalającego zarządzać symulacjami można lepiej poznać oraz zobrazować siły oddziaływania grawitacyjnego w makroskali. Wyniki takich badań mogą być wykorzystywane zarówno przez naukowców jak i w celach popularnonaukowych. Dzięki możliwości zobrazowania, w formie graficznej animacji, wyników przeprowadzonych badań narzędzie to może promować takie dziedziny nauki jak fizyka i astronomia wśród dzieci, młodzieży, ale również i osób dorosłych.

1.2. Zarys projektu

GRAVIsim to z jednej strony narzędzie pozwalające zarządzać klastrem obliczeniowym złożonym z dowolnej liczby komputerów klasy PC w celu wykonywania konkretnych obliczeń na sieci rozproszonej, z drugiej - jest to prosta i przyjazna w użytkowaniu aplikacja internetowa.

Zalogowany użytkownik może dzięki niej wykorzystać potencjał obliczeniowy klastra do symulowania zderzeń grawitacyjnych dowolnej liczby obiektów kosmicznych (np. planet, gwiazd, gromad, galaktyk). Aplikacja obsługuje cały proces takiego zadania zaczynając od utworzenia danych wejściowych, poprzez zlecenie pomiarów, jego monitorowanie, kończąc aż na analizie danych końcowych i jej graficznej wizualizacji.

1.3. Przeznaczenie projektu

Celem projektu jest stworzenie symulacji grawitacyjnej N -ciał wraz z interfejsem webowym pozwalającym na zlecanie badań dla różnych scen/obiektów i graficzna interpretacja otrzymanych wyników.

Projekt przeznaczony jest do celów naukowo-badawczych, obrazujący podstawową mechanikę oddziaływać grawitacyjnych. Skala i wartości obiektów symulacji (odległość, masa, siły) są rzeczywistym przekształceniem odpowiadających wartości świata rzeczywistego, co gwarantuje abstrakcyjnym obiektom symulacji faktyczną nośność informacji względem odpowiadających obiektom rzeczywistym.

Rozdział 2

Zakres projektu

W pierwszej fazie projektu (jeszcze przed oficjalnym Kick-Off) został ustalony cel projektu, który w ramach kursu chcieliśmy osiągnąć, a także funkcjonalności, które aplikacja GRAVIsim powinna posiadać. Funkcjonalności te zostały podzielone na podstawowe oraz rozszerzone, czyli takie które nie są krytyczne dla założonego funkcjonowania systemu jednak ich zaimplementowanie jest dodatkową wartością dla systemu i będzie dużym udogodnieniem (lub rozszerzeniem możliwości) dla osób z niego korzystających.

2.1. Cel projektu

Zbudowanie aplikacji internetowej, która poprzez interfejs użytkownika umożliwi obsługę zadań wykonywanych na systemie rozproszonym. Użytkowanie aplikacji możliwe jest poprzez dostęp do hermetycznego systemu, do którego użytkownik może dostać się jedynie poprzez poprawne jego zarejestrowanie w bazie przez administratora, a następnie zalogowanie do serwisu. Użytkownik systemu GRAVIsim po poprawnym zalogowaniu może wykonać następujące funkcjonalności:

- Generować dane wejściowe symulacji
- Zlecać zadania z wcześniej wygenerowanego pliku
- Zarządzać zadaniami – parametry takie jak: tytuł, opis, ilość iteracji, priorytet, status zadania
- Obserwować postęp prowadzonej symulacji
- Analizować wynik symulacji

2.2. Funkcjonalności podstawowe

- **Symulacja grawitacyjna N-ciał oparta na systemie rozproszonym**

Zaimplementowanie algorytmu dokładnego (pełnego) operującego na systemie rozproszonym, który wykonuje symulacje grawitacyjne N-ciał.

- **Interfejs webowy pozwalający na zlecenie oraz zarządzanie badaniami**

Stworzenie prostej, nowoczesnej i przyjaznej dla użytkownika aplikacji internetowej opartej o frameworki AngularJS oraz front-endowy Bootstrap. Serwis ma być intuicyjny oraz zapewniać dostęp do wszystkich możliwości jakie oferuje GRAVIsim. Użytkownik koniecznie musi mieć możliwość:

- Zlecać badania.
- Konfigurować parametry oraz opis zadania.
- Zarządzać statusem zadania (anulować, wstrzymywać, ustawiać w kolejce zadań).
- Jako rezultat pomiarów otrzymać wizualizację wyników w formie poklatkowej animacji.

- **System uwierzytelniania użytkownika**

Powinien zostać zaprojektowany system uwierzytelniania użytkowników w aplikacji wraz z zaimplementowane następujące funkcjonalności:

- Rejestracja nowego użytkownika.
- Logowanie w systemie.
- Zarządzanie innymi użytkownikami poprzez konto administratora.
- Automatyczne wygaszanie sesji w przypadku określonego czasu braku działań na koncie użytkownika.

- **Możliwość wprowadzenia danych wejściowych z poziomu aplikacji webowej**

Aplikacja powinna być wyposażona w formularz umożliwiający generowanie zadania z poziomu interfejsu oraz jego parametryzowanie poprzez:

- Wgrywanie pliku wejściowego z danymi.
- Nazywanie i opisywanie zadania.
- Ustawianie ilości iteracji.
- Wybieranie priorytetu wykonywalności.

- **Graficzna interpretacja wyników**

Możliwość wizualnej interpretacji wyników wykonanego badania z poziomu szczegółowego widoku zadania w aplikacji. Okno animacji ma mieć możliwość:

- Wstrzymywania oraz ponowego uruchamiania animacji.
- Ustawiania szybkości wyświetlanej animacji (*FPS - Frame per second*).
- Uruchamiania animacji w trybie pełnoekranowym.
- Pobierania danych generujących animację (wynikowych badania).

2.3. Funkcjonalności rozszerzone

- **Historia badań**

Możliwość podglądu archiwalnych i obecnych zadań z pełną dostępnością informacji (parametry, logi, rezultaty) dla danego użytkownika oraz analogicznie dla wszystkich użytkowników jedynie z konta administratora.

- **Monitorowanie postępu prowadzonego badania**

Użytkownik ma mieć możliwość monitorowania postępu wykonywanego zadania w czasie rzeczywistym z poziomu aplikacji. Interfejs ma posiadać dane dotyczące stopnia wykonanego zadania (pasek postępu) wraz ze wszystkimi dodatkowymi informacjami w postaci logów systemowych.

- **Generator danych wejściowych**

Generator w formie graficznego narzędzia w przyjazny i intuicyjny dla użytkownika sposób ma generować dane wejściowe dla systemu badań grawitacyjnych. Generator ma umożliwiać:

- Tworzenie dwóch rodzajów próbek: pojedynczych punktów lub chmury punktów.
- Dla pojedynczej próbki wybór koloru oraz masy.
- Dla chmury próbek dodatkowo: rozmiar chmury oraz ilość pojedynczych próbek w chmurze.

2.4. Wymagania technologiczne

Aplikacja kliencka powinna zostać zbudowana w oparciu o technologię **AngularJS**, w formie aplikacji typu *One-Site-Page* w języku *Javascript*. Obsługa danych w aplikacji ma być zapewniona przez framework **Django**, napisany w *Python*. W celu zapewnienia aplikacji estetycznego wyglądu zastosowany zostanie framework **Bootstrap** wraz z *CSS3* oraz *HTML5*. Do wspomagania autoryzacji będą używane **tokeny JWT**. Moduł aplikacji wykonujący obliczenia rozproszone będzie wykonany przy użyciu **Apache Spark**, natomiast komunikacja pomiędzy aplikacją, a dokładniej **Django Web API** oraz modulem obliczającym będzie opierać się o dedykowany moduł **Job Scheduler’a** oraz **Loggera** (do przesyłania informacji o stanie badań) napisanych w języku *Python* oraz *Javascript*.

Rozdział 3

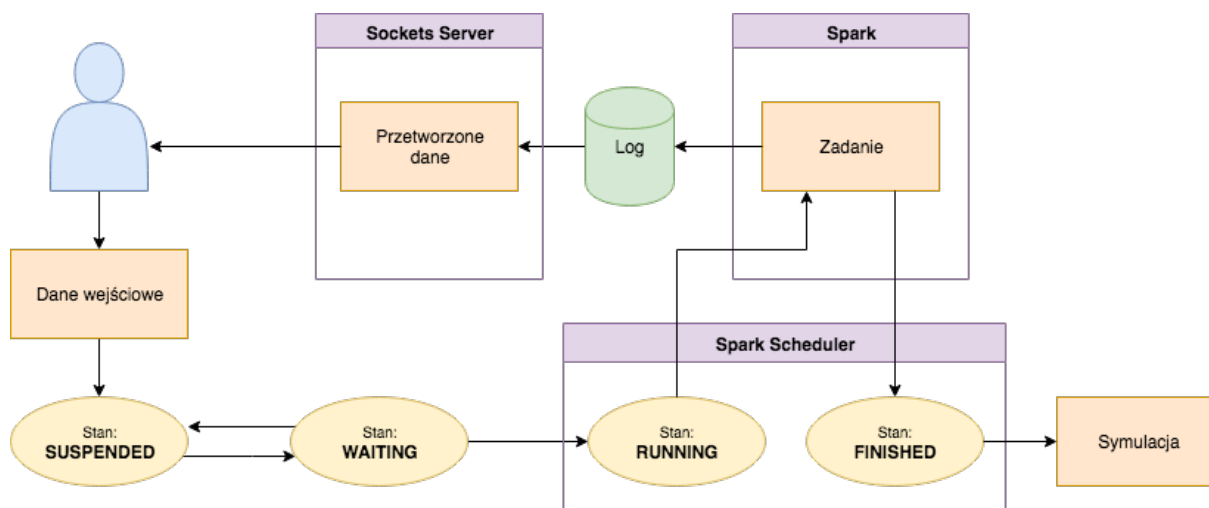
Charakterystyka projektu

3.1. Cykl życia aplikacji

Każde zadanie przetwarzane jest przez wszystkie 4 podsystemy aplikacji. Moduł serwerowy odpowiada za przetwarzanie takich zapytań jak tworzenie czy modyfikacja zleceń. Moduł Spark Schedulera zarządza oczekującymi zleceniami przetwarzając ich stan oraz przekazując do klastra. Moduł socketów pozwala na monitorowanie stanu zlecenia w czasie rzeczywistym. Ponadto, szczegółowo informuje o wszystkich ewentualnych błędach niezależnie od źródła ich wystąpienia. Zewnętrzny moduł Spark odpowiada za przetwarzanie otrzymanych zadań i generowanie symulacji.

Wynikiem pracy podsystemów jest gotowa symulacja w postaci pliku **JSON** wizualizowana w warstwie prezentacji aplikacji klienckiej.

Pełen cykl życia zadania przedstawia poniższy diagram:



Rys. 3.1: Schemat modułów aplikacji GRAVIsim

3.2. Technologia

Aplikacja podzielona została na kilka logicznych modułów, z których każdy pełni odrębną rolę w całym cyklu życia. Można wyróżnić 4 podsystemy:

- Serwer Django - Aplikacja
- Spark - Klaster
- Spark Scheduler - Zarządzanie
- Sockets server - Logger

3.2.1. Moduł aplikacji (Django)

Główny moduł aplikacji. Jego główną rolą jest przetwarzanie zapytań użytkowników, serwowanie statycznych zasobów oraz zarządzanie całym stanem aplikacji.

Serwer podzielony jest na 2 logiczne warstwy: serwerową oraz API. Pierwsza warstwa pełni interfejs pomiędzy warstwą prezentacji (frontend), a całą logiką systemu. Druga warstwa jest znacznie bardziej rozbudowana. Oparta na frameworku *Django REST framework* zajmuje się przetwarzaniem oraz interpretacją zapytań pozwalając na komunikację - odczyt bądź bezpieczny zapis danych - z bazą danych.

Warstwa API jest głównym mechanizmem generowania zleceń badawczych. Zaimplementowane mechanizmy serializacji dbają, aby użytkownik mógł być w stanie edytować tylko wybrane pola przy określonych warunkach - eliminuje to ryzyko utraty spójności danych.

3.2.2. Moduł zarządzający (Spark-Scheduler)

To główny moduł zarządzający wykonywanymi zadaniami. Podobnie jak serwer socketów uruchamiany jest jako komenda głównego serwera. Jego ideą jest przekazywanie oczekujących zadań zgodnie z ustaloną kolejnością priorytetów do obliczeń w klastrze. Moduł odpowiedzialny jest za zachowanie spójności zadań (klaster równocześnie może obsłużyć tylko 1 zadanie) oraz ciągłość pracy (po zakończeniu 1 zadania lub w stanie bezczynności, klaster automatycznie otrzymuje nowe zadanie).

Zasada działania *Spark Scheduler'a* oparta jest na cyklicznym odpytywaniu bazy danych w poszukiwaniu zadań oczekujących na dostępność klastra. Gdy w systemie znajdują się takie zadania, moduł szereguje je zgodnie z ustaloną kolejką priorytetów (im wyższy priorytet - tym szybciej zadanie zostanie obsłużone) oraz w przypadku równoznacznych priorytetów - zgodnie z czasem oczekiwania (eliminacja zjawiska zagłodzenia). Zadanie znajdujące się na początku

kolejki jest blokowane poprzez zmianę stanu na ***RUNNING*** oraz przekazane do klastra.

Przekazanie zadania do obliczenia w klastrze odbywa się poprzez wywołanie komendy ***spark-submit*** z odpowiednimi parametrami, przy użyciu funkcji *Popen*. Idea tego rozwiązania pozwala na przekierowanie całego wyjścia komendy - zarówno zwykłych komunikatów jak i błędów systemowych czy własnych informacji jak stan czy postęp zadania - do określonego pliku zewnętrznego służącego jako log zadania. Tak przekazane zadanie może być asynchronicznie śledzone przez inne moduły oraz użytkowników końcowych.

Po zakończeniu komendy - zarówno pozytywnego jak i w przypadku wystąpienia dowolnego błędu - zadanie jest finalizowane poprzez ustawienie stanu na ***FINISHED***. Rezultat zadania przekazany jest automatycznie w logu zadania oraz w przypadku sukcesu - plik symulacyjny jest zapisany na dysku serwera z ustaloną konwencją nazewnictwa gwarantującą unikalność oraz dostępność.

3.2.3. Moduł asynchroniczny (Socket-Server)

Asynchroniczny moduł uruchamiany jako komenda głównego serwera. Pozwala on na komunikację aplikacji klienckiej z serwerem za pośrednictwem otwartego na oddzielnym porcie gniazda. Jego główną ideą jest umożliwienie asynchronicznego śledzenia stanu i postępu zleconych badań oraz ewentualnych komunikatów błędów i ostrzeżeń.

Główna zasada działania modułu opiera się na cyklicznym odczycie zadanego pliku z logiem zadania i przekazywaniu przetworzonej informacji do socketa. Bardzo ważnym aspektem było zapewnienie nieblokującego przetwarzania - w przeciwnym wypadku serwer byłby w stanie równocześnie obsłużyć tylko i wyłącznie 1 użytkownika.

Zapytanie jest przetwarzane wg. poniższego schematu:

1. Utworzenie połączenia i określenie identyfikatora monitorowanego zadania.
2. Sprawdzenie, czy plik z logiem zadania istnieje na serwerze: jeśli nie istnieje następuje wysłanie komunikatu, że zadanie się jeszcze nie rozpoczęło, następnie przechodzi w tryb nieblokującego oczekiwania przez 1000 ms i ponawia krok 2.
3. Ustalenie znacznika na początek pliku.
4. Odczyt do bufora wszystkich linii od znacznika do końca pliku.
5. Ustalenie znacznika na końcu pliku.
6. Wysłanie do gniazda komunikatu z buforem linii.
7. Nieblokujące oczekiwanie przez 500 ms i powrót do kroku 4.

Połączenia po stronie serwera zamykane są automatycznie po zamknięciu ich w aplikacji klienckiej.

Rozdział 4

Implementacja

4.1. Aplikacja

Aplikacja została podzielona na 3 logiczne warstwy: aplikację użytkową, serwer oraz warstwę API. Frontend został zrealizowany z wykorzystaniem frameworka AngularJS oraz wzorca projektowego MVC. Wykorzystanie mechanizmów routingu po stronie aplikacji klienckiej pozwoliło na stworzenie aplikacji SPA (Single Page Application), dzięki czemu strona wczytywana jest dynamicznie przez obciążania połączenia z serwerem. Zgodnie z tą ideą - wszystkie niezbędne zasoby wczytywane są w dopiero wtedy, gdy są potrzebne a nie wszystkie przy każdym ładowaniu strony.

Główną ideą warstwy serwerowej jest udostępnianie statycznych zasobów oraz pośredniczenie w przetwarzaniu zapytań i odpowiedzi (serwer jest jedynym punktem dostępowym aplikacji). Wszelka logika samego procesu przetwarzania oraz komunikacji z bazą danych odbywa się tylko i wyłącznie w warstwie API, która została zabezpieczona biblioteką JWT. Każde zapytanie musi przechowywać w nagłówku odpowiedni token (otrzymany uprzednio w trakcie logowania) aby zapytanie było przetworzone. W przeciwnym wypadku, biblioteka JWT znajdująca się na styku warstwy serwerowej i API, odrzuci zapytanie i odeśle odpowiedź z kodem błędu 403.

W warstwie API można rozróżnić 2 główne kategorie klas obiektów: widoki oraz serializery. Pierwsze z nich odpowiadają za obsługę zapytań: GET, POST, DELETE. Ponadto, mają zaimplementowane mechanizmy bezpieczeństwa oraz ogólną konfigurację sposobu generowania odpowiedzi. Widoki dziedziczą po generycznej klasie, mającej zdefiniowane metody dostępu.

```
1 class DefaultsMixin(viewsets.ModelViewSet):
2     authentication_classes = (
3         authentication.SessionAuthentication,
4         JSONWebTokenAuthentication
5     )
6     permission_classes = (
7         permissions.IsAuthenticated,
8     )
9     filter_backends = (
10        filters.DjangoFilterBackend,
11        filters.SearchFilter,
12        filters.OrderingFilter
13    )
14
15 class ReadOnlyDefaultsMixin(viewsets.ReadOnlyModelViewSet):
```

```

16 authentication_classes = (
17     authentication.SessionAuthentication,
18     JSONWebTokenAuthentication
19 )
20 permission_classes = (
21     permissions.IsAuthenticated,
22 )
23 filter_backends = (
24     filters.DjangoFilterBackend,
25     filters.SearchFilter,
26     filters.OrderingFilter
27 )

```

Listing 4.1: Generyczne klasy widoków API

Parametr `authentication_classes` przechowuje krotkę klas odpowiadających za schemat autoryzacyjny. Obie klasy generyczne autoryzują zapytania przy użyciu tokenów. Parametr `permission_classes` odpowiada za warunkowe przyznanie dostępu do danych realizowanych w ramach konkretnej instancji widoku. Z racji, że nie ma rozróżnienia na administratorów wśród użytkowników, dostęp do danych jest ujednolicony i wymaga jedynie autoryzacji. Ostatni parametr `filter_backends` odpowiada za mechanizm filtrowania (selekcji) oraz sortowania rekordów wśród kolekcji danych.

Każdy z widoków odwołuje do konkretnej klasy serializera. Druga kategoria klas API odpowiada za sposób prezentacji oraz walidację danych konkretnej instancji modelu (adekwatnie: rekordu relacji). Każdy widok czy serializer może się odwoływać do innych serializerów – zależnie od potrzeb sposobu obsługi danych.

4.2. **Logger**

Mechanizm śledzenia postępu zadań został zrealizowany w formie komendy serwera głównego. Strumień wyjściowy każdego uruchomionego zadania jest przekierowywany do zewnętrznego pliku o unikalnej nazwie, utworzonego na dysku serwera. Każda informacja związana z zadaniem - niezależnie czy to komunikat o stanie, czy zmiana stanu zadania w klastrze lub błąd systemowy skutkujący przerwaniem wykonywania zadania - jest zapisywana do pliku. Pozwala to na skuteczne i bardzo dokładne monitorowanie przebiegu symulacji oraz wykrywania i eliminacji źródeł ewentualnych błędów.

Sposób przekierowania strumienia zadań do pliku logującego przedstawia poniższy fragment:

```

1 path = os.path.join(LOGS_DIR, "{}.log".format(instance.id))
2 with open(path, 'wb') as log:
3     p = Popen(command, stdout=log, stderr=log, shell=True)
4     p.wait()

```

Listing 4.2: Tworzenie logu

Wszystkie wewnętrzne komunikaty - np. informacja o symulacji, zmiana stanu realizacji - formatowane są według ustalej reguły tokenów, co pozwala na jednoznaczną identyfikację i adekwatne parsowanie wiadomości. Dostępny jest poniższy zestaw komunikatów:

- **D** - data - komunikat traktowany jako parametr symulacji i jest zapisywany do słownika

- **P** - progress - informacja o bieżącym postępie oraz estymowanym czasie zakończenia zadania
- **X** - status - słowny komunikat informujący o stanie zadanie
- **W** - ostrzeżenie - w monitorze zadania wiadomość będzie wyróżniona
- **E** - error - wiadomość krytyczna dla zadania
- **S** - success - informacja o prawidłowym zakończeniu zadania
- **I** - info - istotny komunikat informacyjny

Zawartość logu przesyłana jest asynchronicznie za pomocą serwera socketów. Serwer odpowiada za pierwszą fazę selekcji oraz parsowania komunikatów. Cała logika prezentacji znajduje się w dyrektywnie w aplikacji klienckiej.

4.3. Zarządzanie zadaniami

W celu sprawnego zarządzania zasobami klastrami napisany został dedykowany serwer - spark scheduler - uruchamiany jako komenda serwera głównego. Głównymi zadaniami schedulera są: zapewnienie spójności pracy klastra (jednocześnie tylko jedno zadanie może być realizowane) oraz zapewnienie ciągłości pracy (zadanie oczekujące w kolejce jest przekazywane do klastra niezwłocznie po zakończeniu poprzedniego zadania lub natychmiastowo w przypadku bezczynności klastra).

Spark scheduler zrealizowany został jako nieskończona pętla, cyklicznie sprawdzająca bazę danych w poszukiwaniu i sortowaniu zadań oraz ich realizacji pod kątem umożliwienia pracy loggera:

```

1 def job(self, *args, **options):
2     jobs = SparkJob.objects\
3         .filter(state=SparkJobStates['WAITING'])\
4         .order_by('-priority')\
5         .order_by('id')
6     if jobs.count() == 0:
7         self.log("No jobs in queue: waiting...")
8
9     while(jobs.count() == 0):
10         sleep(SLEEP_DELAY)
11
12     instance = jobs.first()
13     instance.state = SparkJobStates['RUNNING']
14     instance.started = datetime.now()
15     instance.save()
16
17     self.log(self.style.WARNING("Starting job: #{}".format(instance.id)))
18     self.process_job(instance, *args, **options)

```

Listing 4.3: Główna pętla Spark Schedulera

Pierwszym elementem pętli jest sortowanie zadań: wszystkie oczekujące zadanie (i tylko takie) szeregowane są wg. malejącego priorytetu, a w przypadku jednakowej wartości - pierwszeństwo zyskuje zadanie z mniejszym id (czyli zadanie starsze). Następnie wątek zasypia i cyklicznie werfykuje kolejkę aż do znalezienia przynajmniej jednego zadania gotowego do przekazania. Skorzystano tutaj w atrybutu leniwej-ewaluacji `QuerySet`ów, dzięki czemu przygotowane

pod zmienną `jobs` zapytanie jest wywoływane każdorazowo przy użyciu metody `count`. Następnie zadanie ma zmieniany status na aktywny oraz przekazywane do wywołania w klastrze. Po zakończeniu obliczeń, zadanie wraca do scheduler gdzie ma ustawiany status na zakończony i pętla główna rozpoczyna kolejną iterację.

4.4. Symulacja

Symulacja napisana została w języku Python i interpretowana jest przez Apache Sparka z modulem PySpark. Cała symulacja - z logicznego punktu widzenia - spełnia dogmat czystej funkcji, czyli jej wyjście zależy tylko i wyłącznie od wejścia (eliminacja stanu wewnętrznego). Symulacja przyjmuje 3 parametry wejściowe: nazwę pliku wejściowego, identyfikator zadania (zgodny z id rekordu zadania w bazie danych) oraz ilość iteracji. Dla tak przedstawionych danych zwrócony zostanie plik wynikowy pełnej symulacji, zapisany na dysku serwera z nazwą przekazanego identyfikatora zadania.

Aby symulacja mogła być prawidłowo przeprowadzona, konieczne jest zapewnić pliku wejściowego wg. poniższego schematu:

```

1 [
2   {
3     "mass": <Number range(1, 8)>,
4     "colour": <String as "RRGGBB">,
5     "x": <Number range(-800, 800)>,
6     "y": <Number range(-800, 800)>,
7   },
8   { ... }
9 ]

```

Listing 4.4: Plik wejściowy symulacji

Symulacja podzielona została na kilka etapów:

1. inicjalizacja parametrów
2. odczyt danych
3. przygotowanie struktury wynikowej
4. obliczanie pojedynczej klatki
 1. zrównoleglenie danych
 2. normalizacja
 3. obliczenie przyspieszenia ciał
 4. obliczenie prędkości ciał
 5. przemieszczenie ciał
 6. zapis danych klatki w strukturze
5. zapis pełnej struktury do pliku wyjściowego

Każdy z etapów został wydzielony do odrębnej metody (operacje do odrębnej klasy) co pozwoliło zachować przejrzystość kodu i łatwość ewentualnego rozwoju. Pełen cykl pojedynczej iteracji symulacji przedstawia poniższy fragment kody:

```

1 for frame in range(1, self.iterations):

```

```

2 progressBar.print(frame, self.iterations)
3 rdd = context.parallelize(data)
4 rdd = self.normalize(rdd)
5 rdd = ParticlesOperations.updateAccelerations(rdd)
6 rdd = ParticlesOperations.updateVelocity(rdd)
7 rdd = ParticlesOperations.updatePosition(rdd)
8 frame, data = self.saveStep(rdd)
9 result['timeline'].append(frame)

```

Listing 4.5: Plik wejściowy symulacji

Mimo iż symulacja opiera się na obliczeniu sił każdego ciała względem wszystkich pozostałych (co daje złożoność kwadratową) symulację zrównoleglono liniowo. W trakcie projektowania zweryfikowano 2 metody: rozkładu jedno-poziomowego ciał na zadania oraz dokładnego jako iloczyn kartezyjski ciał. Z racji, iż same obliczenia sił mają małą złożoność obliczeniową, nakład opóźnienia komunikacyjnego znacząco spowolnił symulację tworzoną jako iloczyn kartezyjski.

W wyniku działania symulacji utworzony zostaje plik wg. poniższego schematu przedstawiającego fragment funkcji generującej szkic struktury wynikowej:

```

1 def prepareResult(self, data):
2     return {
3         'particles': map(lambda p: {
4             'mass': p['mass'],
5             'colour': p['colour'],
6         }, data),
7         'timeline': [map(lambda p: [
8             p['x'],
9             p['y']
10          ], data)],
11     }

```

Listing 4.6: Wynik symulacji

Rozdział 5

Instalacja i uruchomienie aplikacji

W celu uruchomienia aplikacji GRAVIsim ze wszystkimi jej funkcjonalnościami, konieczna jest instalacja dodatkowych zależności, z których główną jest Apache Spark zajmujący się obsługą klastra obliczeniowego.

5.1. Proces instalacji

W pierwszej kolejności należy pobrać i zainstalować następujące oprogramowanie:

1. **Java SE Development Kit (SDK) w wersji 8 lub nowszej** - [Link do pobrania](#)

Aby zweryfikować poprawność instalacji, należy otworzyć konsolę (`cmd`), wpisać `java` i kliknąć *Enter*. Jeśli pojawi się wiadomość:

```
Java is not recognized as an internal or external command.
```

lub podobna, należy:

Windows: skonfigurować zmienne środowiskowe: `JAVA_HOME` (podając ścieżkę do miejsca zainstalowania *Java JDK*) oraz `PATH` jako `%JAVA_HOME%\bin`

1. Przejdź do `Control Panel\System and Security\System`

2. Wybierz `Advance System Settings`

3. Wybierz `Environment Variables`

4. W polu `user variables` dodaj nową zmienną środowiskową np. `SCALA_HOME` podając jako wartość ścieżkę do miejsca zainstalowania *Scala*.

5. W polu `user variables` edytuj zmienną `PATH` dodając nowy klucz np. `%SCALA_HOME%\bin`

Linux: dodać ścieżkę z plikiem wykonywalnym do `PATH`

2. **Scala** - [Link do pobrania](#)

Gdy Scala zostanie zainstalowana, należy ustawić zmienne środowiskowe `SCALA_HOME` do folderu, gdzie zainstalowano *Scala* i `PATH` jako `%SCALA_HOME%\bin`.

5.2. Tworzenie klastra

Tworzenie klastra rozpoczyna się od uruchomienia głównego węzła na maszynie mającej być węzłem zarządzającym. Odbywa się to przy użyciu komendy:

```
start-master.sh
```

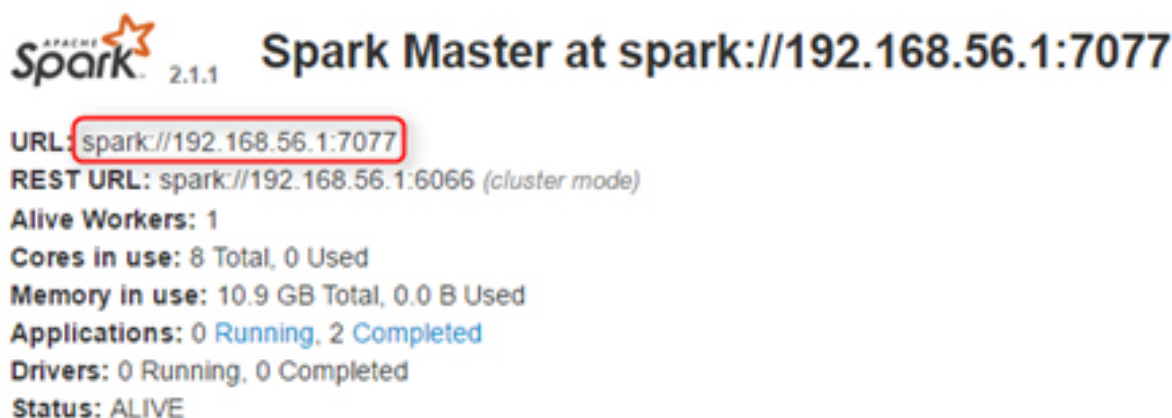
znajdującej się w folderze `sbin` katalogu głównego *Apache Spark'a*.

W przypadku problemów z uruchomieniem powyższej komendy (szczególnie w systemie *Windows*, należy w konsoli systemowej wpisać:

```
Spark-class org.apache.spark.deploy.master.Master
```

Po uruchomieniu konieczne jest udanie się pod adres `localhost:8080`, gdzie powinien zostać aktywowany główny panel webowy klastra. Ukazany tam adres URL węzła zarządzającego będzie adresem docelowym wszystkich węzłów obliczeniowych, np.

```
spark://192.168.56.1:7077
```



Rys. 5.2: Spark Master - Adres URL węzła zarządzającego klastrem

Kolejnym krokiem jest aktywacja wszystkich pozostałych maszyn jako węzły obliczeniowe. Wykonuje się to poprzez użycie komendy znajdującej się jak uprzednio w folderze `sbin` katalogu głównego *Apache Spark'a*:

```
start-slave.sh -c <max-cores> -m <max-memory> <master-url>, np.
```

```
start-slave.sh -c 4 -m 2G spark://192.168.56.1:7077
```

Jeśli wystąpią problemy z powyższą komendą, należy skorzystać z:

```
Spark-class org.apache.spark.deploy.worker.Worker  
-c <max-cores> -m <max-memory> <master-url>
```

Jeśli węzły prawidłowo skomunikują się z głównym węzłem, będzie to widoczne w panelu webowym głównej maszyny. **Klaster obliczeniowy został uruchomiony poprawnie.**

5.3. Uruchomienie GRAVIsim

Uruchamianie aplikacji należy rozpocząć od aktywowania właściwej wersji *Python'a* (3.5) najlepiej przy użyciu narzędzia *Virtual-Env*. Dla tak przygotowanej konfiguracji należy zainstalować wszystkie wymagane zależności *Python'a* poprzez komendę:

```
pip install -r requirements.txt
```

oraz wszystkie zależności aplikacji klienckiej poprzez komendę:

```
bower install
```

Kolejnym krokiem jest uruchomienie klastra (instrukcja w poprzednim rozdziale) - bardzo ważne jest aby komendy tworzenia klastra wywołane były w konfiguracji *Python 3.5* (aktywnym wirtualnym środowisku).

Aby zapewnić prawidłowe działanie *logger'a* zaleca się zmianę ustawień logowania *Sparka* poprzez zmianę poziomu logowania z **INFO** na **WARN**. W tym celu należy w pliku:

```
%SPARK_HOME/conf/log4j.properties
```

Jeśli plik nie istnieje, należy powielić plik:

```
%SPARK_HOME/conf/log4j.properties.template
```

oraz zmienić jego nazwę na `log4j.properties`

dokonać modyfikacji następującego kodu:

```
Log4j.rootCategory=INFO, console, na
```

```
Log4j.rootCategory=WARN, console
```

Następnie należy uruchomić wszystkie moduły:

1. Serwer: `python manage.py runserver`
2. Sockets-Server: `python manage.py sockets-server`
3. Spark-Scheduler: `python manage.py spark-scheduler <master-url>`

Komendy są blokujące w związku z czym zaleca się użycie kilku terminali lub uruchomieniu komend w trybie odłączonym i przekierowaniu wyjścia do pliku zewnętrznego.

Dodatkowo - na wszystkich maszynach oprócz maszyny głównej - należy podmontować folder `media` przy użyciu skryptu `mount.sh` dostarczonym z aplikacją lub ręcznie przy użyciu komendy `sshfs`.

Aplikacja dostępna jest pod adresem hosta głównego na porcie **8000**, np.

```
http://localhost:8000
```

Rozdział 6

Instrukcja użytkownika

Rozdział 7

Badanie złożoności obliczeniowej

Badania przeprowadzono w politechnicznym laboratorium komputerowym z wykorzystaniem 9 identycznych komputerów, z których jeden służył za główny serwer aplikacji oraz zarządcę klastra, a pozostałe 8 za węzły obliczeniowe (równolegle od 1 do 8 - zależnie od specyfikacji testu).

7.1. Zestaw konfiguracyjny

Testy przeprowadzono dla wybranych zestawów konfiguracyjnych, gdzie parametrami były:


- Ilość węzłów obliczeniowych
- Wielkość pliku wejściowego (ilość punktów masy)
- Ilość iteracji symulacji

Każdy z węzłów obliczeniowych otrzymywał na wyłączność badań:

- 2 rdzenie (procesor i7 3.4MHz)
- 2.0 GB pamięci RAM (DDR3)

W trakcie badania parametry oraz zasoby poszczególnych węzłów nie były zmieniane.

Symulacja ma charakter liniowy (liniowa charakterystyka iteracji) dlatego dla większych wartości istnieje możliwość dokładnej estymacji czasu obliczania w oparciu o empiryczne wyniki pomiarów dla wartości mniejszych.



2.1.1

Spark Master

URL: spark://gravisim:7077

REST URL: spark://gravisim:6066 (cluster mode)

Alive Workers: 8

Cores in use: 16 Total, 16 Used

Memory in use: 16.0 GB Total, 8.0 GB Used

Applications: 1 Running, 7 Completed

Drivers: 0 Running, 0 Completed

Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20170529100618-156.17.41.50-40529	156.17.41.50:40529	ALIVE	2 (2 Used)	2.0 GB (1024.0 MB Used)
worker-20170529110143-156.17.41.18-39437	156.17.41.18:39437	ALIVE	2 (2 Used)	2.0 GB (1024.0 MB Used)
worker-20170529110531-156.17.41.60-35279	156.17.41.60:35279	ALIVE	2 (2 Used)	2.0 GB (1024.0 MB Used)
worker-20170529110806-156.17.41.51-40762	156.17.41.51:40762	ALIVE	2 (2 Used)	2.0 GB (1024.0 MB Used)
worker-20170529110841-156.17.41.17-45451	156.17.41.17:45451	ALIVE	2 (2 Used)	2.0 GB (1024.0 MB Used)
worker-20170529111009-156.17.41.48-44214	156.17.41.48:44214	ALIVE	2 (2 Used)	2.0 GB (1024.0 MB Used)
worker-20170529111243-156.17.41.52-46199	156.17.41.52:46199	ALIVE	2 (2 Used)	2.0 GB (1024.0 MB Used)
worker-20170529113413-156.17.41.15-44760	156.17.41.15:44760	ALIVE	2 (2 Used)	2.0 GB (1024.0 MB Used)

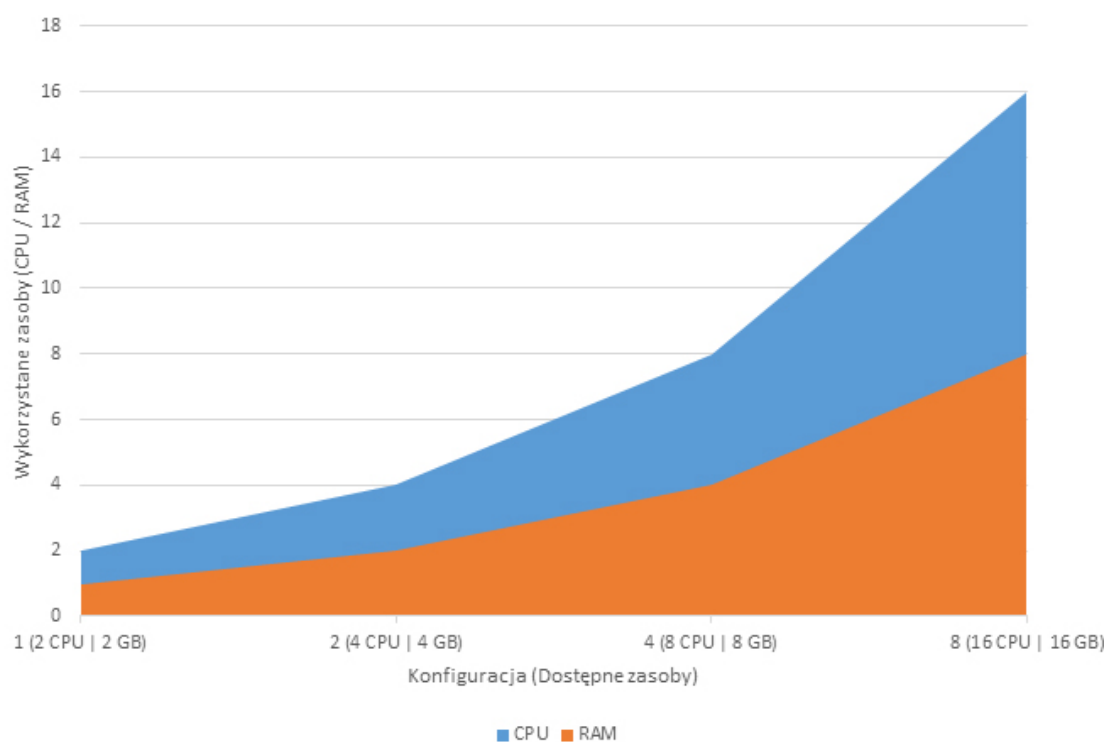
Rys. 7.1: Konfiguracja klastra wykorzystywanego do wykonywania pomiarów

7.2. Wyniki badań

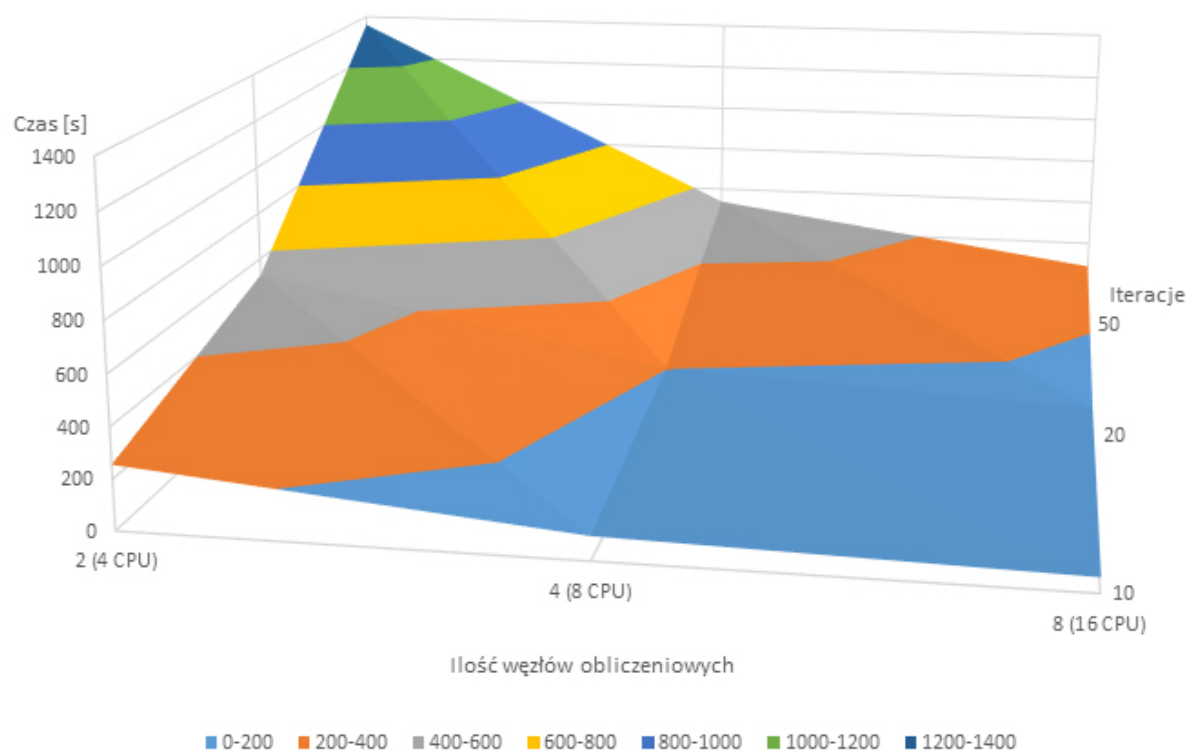
Wyniki badań oraz analizy przedstawia poniższe zestawienie tabelaryczne oraz wykresy.

Id	Węzły	CPU	RAM [GB]	Ilość punktów	Ilość iteracji	Wynik [s]
1	1	2	2.0	100	500	215.67
2	1	2	2.0	100	1000	421.42
3	1	2	2.0	100	2000	821.24
4	2	4	4.0	5000	10	256.46
5	2	4	4.0	5000	20	528.44
6	4	8	8.0	5000	10	94.43
7	4	8	8.0	5000	20	201.78
8	8	16	16.0	100	200	121.90
9	8	16	16.0	100	500	238.60
10	8	16	16.0	100	1000	537.77
11	8	16	16.0	5000	10	58.99
12	8	16	16.0	5000	20	116.29
13	8	16	16.0	5000	500	2704.34

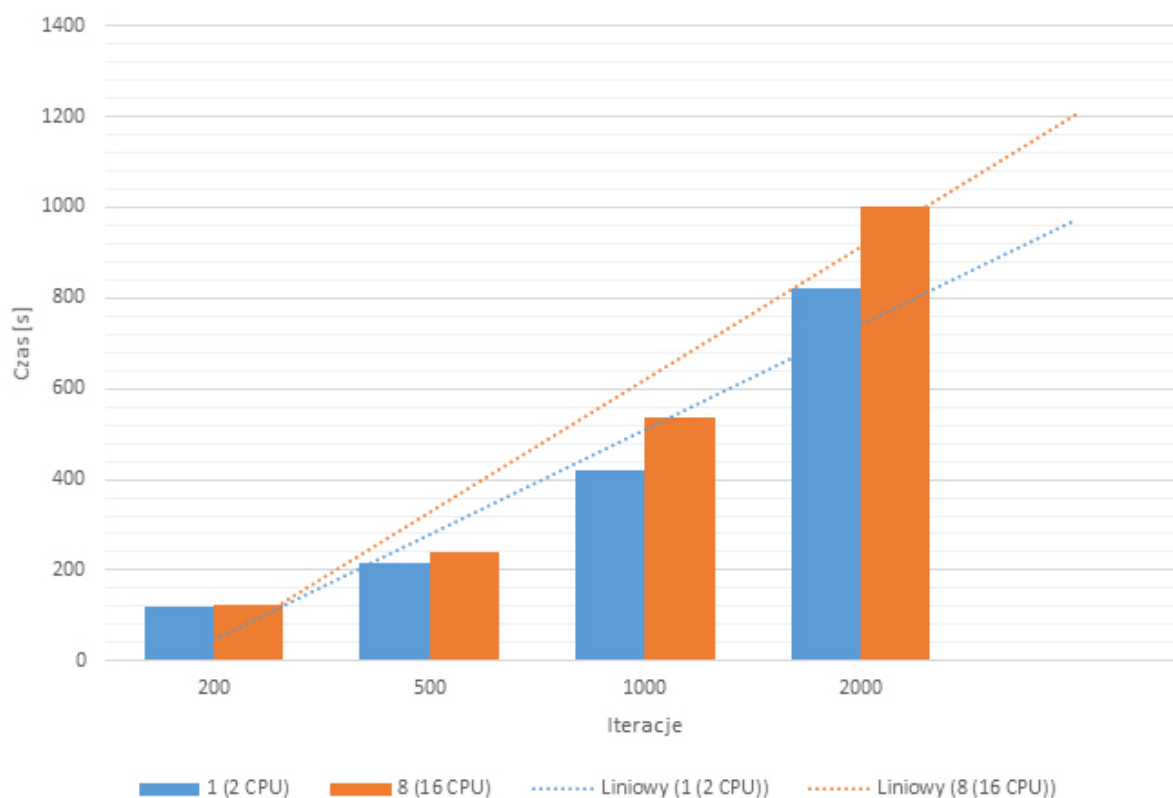
Tab. 7.1: Wyniki czasowe pomiarów wydajności klastra obliczeniowego



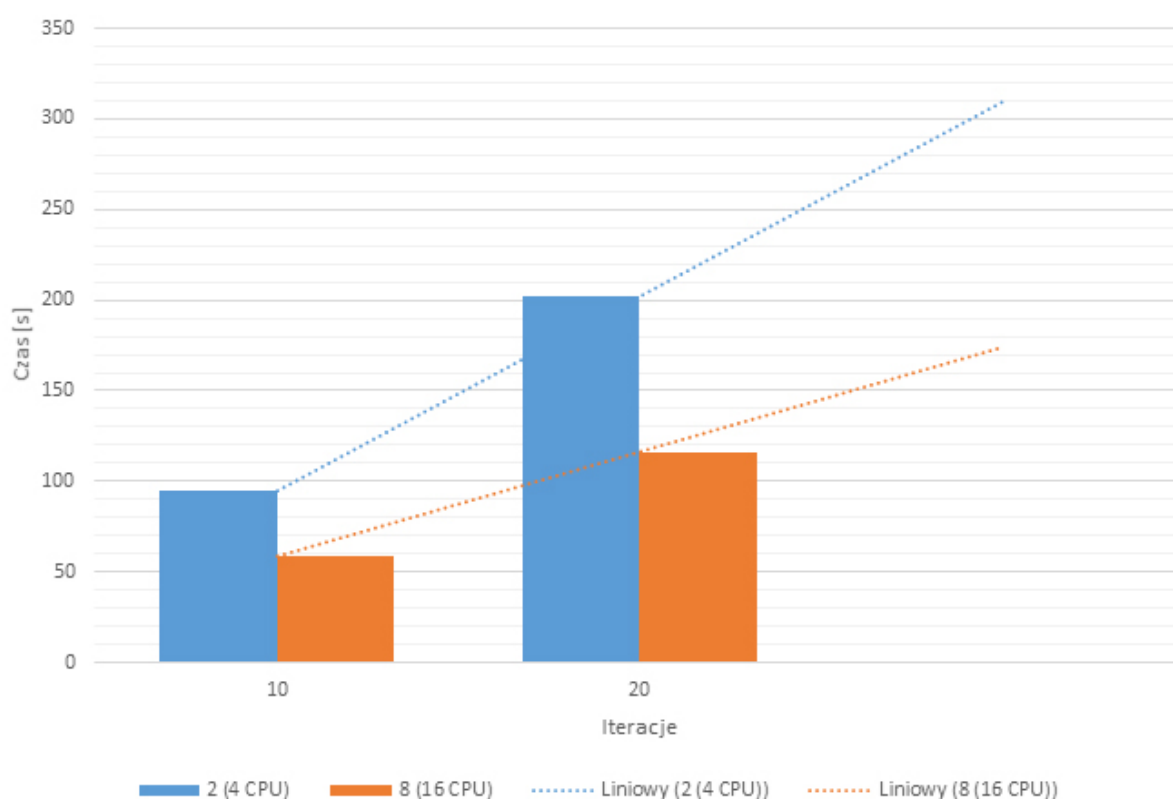
Rys. 7.2: Zużycie zasobów dla różnej konfiguracji klastra



Rys. 7.3: Złożoność czasowa dla symulacji wielu punktów zależnie od ilości węzłów oraz ilości iteracji



Rys. 7.4: Porównanie złożoności czasowych symulacji małej ilości punktów dla skrajnych konfiguracji klastra



Rys. 7.5: Porównanie złożoności czasowych symulacji dużej ilości punktów dla skrajnych konfiguracji klastra

7.3. Analiza wyników

Na podstawie wyników przeprowadzonych badań można zaobserwować, iż wszystkie 3 badane parametry mają znaczący wpływ na czas wykonywania zadania. Klaster obliczeniowy opiera się na komunikacji poszczególnych węzłów, która odbywając się poprzez sieć skutkuje znaczącym opóźnieniem w wykonywaniu zadania. Ma to ogromny wpływ dla zadań liczonych stosunkowo szybko (mała ilość punktów masy w symulacji), stąd też zaobserwowano, iż klaster o większej ilości węzłów obliczył to samo zadanie w czasie o 10-15% dłuższym niż klaster składający się z 1 węzła. Dla większych zadań, narzut komunikacyjny staje się znikomo mały i czas tego samego zadania zależy tylko od dostępności zasobów.

Rozdział 8

Podsumowanie i wnioski

Projekt GRAVIsim okazał się trudnym i skomplikowanym przedsięwzięciem, które wymagało bardzo dużej ilości czasu, a także wiedzy. W czasie realizacji poszczególnych zadań konieczne okazało się poznanie i zrozumienie wielu rozwiązań, z którymi wcześniej nie mieliśmy do czynienia.

Proces tworzenia projektu nie obył się bez problemów i trudności, z którymi należało się zmierzyć. Były to między innymi problemy personalne, wskutek czego ostatecznie zespół projektowy liczy jedynie 2 członków. Pojawiły się również problemy technologiczne jak np. multiplatformowość napisanego przez nas *Job-Schedulera*. Początkowo kod utworzony na systemie *Unixowym* nie działał poprawnie na komputerach z zainstalowanym systemem *Windows*. Ostatecznie jednak udało się zmodyfikować kod w taki sposób aby spełniał swoje zadanie na obydwu klasach systemów.

Analiza przeprowadzonych pomiarów wykazała, iż narzut transmisji danych w klastrze obliczeniowych dla małych zadań jest na tyle duży, że nieoptymalne jest jego wykorzystanie. Przy zadaniach stosunkowo większych, wzrastają możliwości i zalety wykorzystywania *Apache-Spark'a*. Narzut informacji transmitowanych między węzłami obliczeniowymi staje się coraz mniejszy, a moc obliczeniowa wielu węzłów pozwala stanowczo przyspieszyć realizację powierzonych zadań, co potwierdza zasadność korzystania z rozproszonych rozwiązań obliczeniowych.

Podsumowując projekt można stwierdzić, iż wszystkie założone funkcjonalności, zarówno podstawowe jak i rozszerzone zostały zaimplementowane i wdrożone. **GRAVIsim** umożliwia pełną obsługę: przygotowanie, zlecenie oraz analizowanie symulacji grawitacyjnej N-ciał wykonywanej na systemie rozproszonym.

Spis rysunków

3.1. Schemat modułów aplikacji GRAVISim	9
5.1. Komunikat informujący o poprawnie zakończonej instalacji Apache Spark	18
5.2. Spark Master - Adres URL węzła zarządzającego klastrem	19
7.1. Konfiguracja klastra wykorzystywanego do wykonywania pomiarów	22
7.2. Zużycie zasobów dla różnej konfiguracji klastra	23
7.3. Złożoność czasowa dla symulacji wielu punktów zależnie od ilości węzłów oraz ilości iteracji	24
7.4. Porównanie złożoności czasowych symulacji małej ilości punktów dla skrajnych konfiguracji klastra	24
7.5. Porównanie złożoności czasowych symulacji dużej ilości punktów dla skrajnych konfiguracji klastra	25