Piotr Mącznik, 209827
Mikołaj Malinowski, 218253
Karol Dudek, 226002
Jan Niewiarowski, 210828
Grzegorz Wykurz, 218571

Wrocław, 15.06.2018

Aplikacje internetowe i rozproszone

Projekt

Temat: Faktoryzacja dużej liczby metodą 'brutalnej siły' - stwierdzenie, że liczba jest pierwsza. Badanie skalowalności programu. Program powinien umożliwić faktoryzację liczb większych niż (2^64) -1

Prowadzący:
dr hab. inż. Henryk Maciejewski

Politechnika Wrocławska

Spis treści

1. Cel projektu

Celem naszego projektu jest napisanie programu do faktoryzacji dużych liczb, stwierdzenia czy podana liczba jest pierwsza. Na projekt składa się **frontend** – strona www będąca głównym panelem do wprowadzania i uzyskiwania wyniku zadanej liczby przez zalogowanego do systemu użytkownika oraz **backend** – baza danych (schemat przedstawiony w dalszej części dokumentacji), klaster obliczeniowy oraz elementy tj. weryfikacja czy liczba jest pierwsza, kolejka zadań obliczeń, postęp wykonywania obliczeń itd.

2. Wstęp teoretyczny

Na wstępie omówione zostanie czym jest tak naprawdę faktoryzacja.
Faktoryzacja lub rozkład na czynniki to proces, który dla danego x znajduje takie obiekty, że ich iloczyn jest równy x. Obiekty te nazywamy czynnikami, dzielnikami x lub faktorami.
Faktoryzacja liczby całkowitej x, czyli to co zwykle mamy na myśli mówiąc o faktoryzacji, to znalezienie takich liczb całkowitych y1, y2, ..., yn, że ich iloczyn jest równy danej liczbie: x = y1y2…yn, przy czym żadne z yi nie może być równe 1 lub x (tzw. faktoryzacja trywialna).
Najprostszy algorytm polega na próbie dzielenia faktoryzowanej liczby n przez wszystkie liczby pierwsze od 1 do pierwiastka z n. Algorytm ten dobrze nadaje się do tego, żeby zacząć faktoryzować liczbę – losowa liczba ma zarówno małe jak i duże czynniki. Połowa liczb dzieli się przez 2, co trzecia przez 3, co piąta przez 5 itd. Jeśli więc faktoryzowana liczba jest losowa, można z bardzo dużym prawdopodobieństwem pozbyć się szybko niskich czynników.

3. Wymagania funkcjonalne aplikacji

3.1 Użytkownik po wejściu na stronę internetową ma możliwość zalogowania się do aplikacji podając unikatowy login oraz własne hasło, które posłużą mu także w późniejszym logowaniu do serwera.

3.2 Użytkownik po wprowadzeniu dwóch poprawnych danych logowania(tj. nazwa użytkownika, hasło) będzie miał możliwość zarządzania swoimi zadaniami. Będą to:

a) wygenerowanie nowego zadania do aplikacji (np. podając liczbę lub wczytując dane z pliku),
b) sprawdzanie statusu zadań już wykonanych oraz wykonywanych w tym momencie przez aplikację,

3.3 Ponadto aplikacja będzie wyposażona w:

a) ukazanie postępu wykonywanego zadania poprzez procentowe wskazanie stopnia realizacji problemu,
b) algorytm pozwalający na sprawdzenie czy liczba jest pierwsza,
c) stały przydział pamięci dla każdego użytkownika – tylko 20 najnowszych zadań będzie przechowywanych na serwerze.

3.4 Przyjmujemy, iż problemy zadane przez użytkowników będą wykonywane przez aplikację szeregowo. Co za tym idzie wszystkie dostępne klastry w danym momencie działają nad rozwiązaniem danego problemu, a dopiero po jego wykonaniu przystępują do rozwiązywania kolejnego problemu. Wszystkie pozostałe niewykonane zadania będą kolejkowane i rozwiązane według daty dodania zadania.

4. Wymagania niefunkcjonalne

4.1 Środowiskiem pracy systemu jest aplikacja w przeglądarce.

4.2 Aplikacja zapewnia integralność z większością dostępnych przeglądarek internetowych.

4.3 Hasła użytkowników nigdy nie powinny być widoczne dla osób postronnych.

4.4 System powinien wytrzymać 100 jednoczesnych prób wykonywania czynności w aplikacji nie powodując przeciążenia.

4.5 Liczby wprowadzane ręcznie powinny być sprawdzane pod kątem błędów (występowania innych znaków niż dopuszczalne), tak samo jak zaimportowany plik do aplikacji, który nie może być pusty.

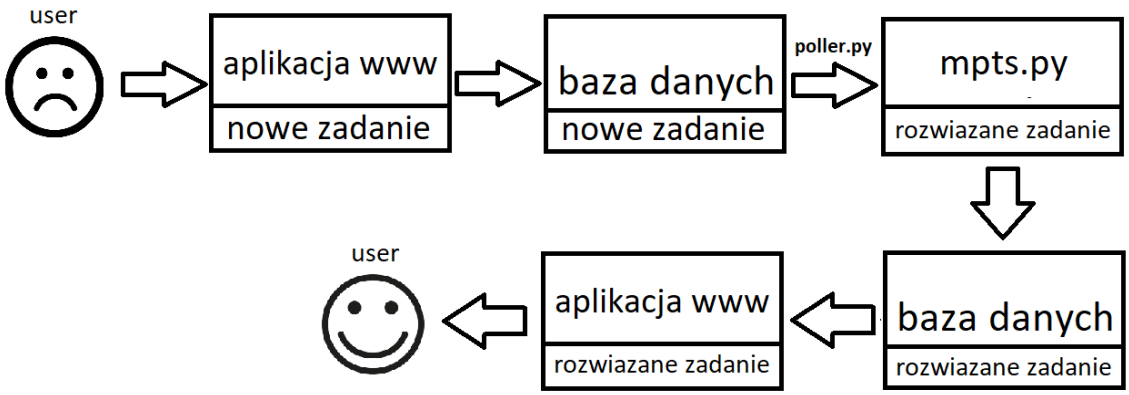
4.6 Aplikacja powinna być „user friendly” tj. łatwa w użytkowaniu dla każdej osoby mającej styczność z aplikacją.

4.7 W systemie zastosowany będzie protokół komunikacyjny (MPI) będący standardem przesyłania komunikatów pomiędzy procesami programów równoległych działających na jednym lub więcej komputerach.

4.8 System powinien pozwolić na wczytywanie dużych liczb (większych niż (2^64) -1.

5. Koncepcja i architektura

5.1 Logika działania



Screen 1. Schemat logiki działania aplikacji

Użytkownik aplikacji wprowadza zadanie poprzez aplikację www. Zadanie przesyłane jest do bazy danych. Poller.py sprawdza co dwie sekundy, czy w bazie danych znajduje się nierozwiązane zadanie. Gdy zauważy je w bazie danych uruchamia skrypt mpts.py, który rozwiązuje zadany problem. Następnie przesyła go do bazy danych. Dzięki aplikacji WWW użytkownik może sprawdzić w historii swoje rozwiązane zadanie.

5.2 Czym jest zadanie?

Zadaniem zleconym przez użytkownika jest sprawdzenie czy liczba należy do liczb pierwszych. Użytkownik ma możliwość wprowadzenia liczby ręcznie lub z pliku. Liczba musi być dodatnia oraz całkowita.

5.3 Czym jest ziarno podziału?

Ziarno podziału decyduje o tym, w jaki sposób zadania dzielona są pomiędzy procesami. Wyróżniamy grube ziarno, tj. do każdego procesu (klastra) przydzielona jest duży przedział liczb oraz małe ziarno, tj. do każdego procesu przydzielany jest mały przedział liczbowy. W aplikacji ziarno podziału dobierane jest automatycznie na podstawie wybranej liczby procesów.

5.4 Testy pierwszości

Testy pierwszości to algorytmy, które pozwalają nam na stwierdzenie, czy liczba jest złożona, czy (dla testów probabilistycznych – prawdopodobnie) pierwsza. Ważne jest tutaj to, że w testach probabilistycznych nigdy nie będziemy mieli całkowitej, stuprocentowej pewności, czy dana liczba jest liczbą pierwszą. Jednak odpowiednia ilość powtórzeń probabilistycznego testu pierwszości dla danej liczby może znacznie zmniejszyć prawdopodobieństwo błędu do bardzo małej liczby. Najprostszy test pierwszości to taki, który po prostu sprawdza, czy nasza testowana liczba n jest podzielna przez kolejne liczby od 2, 3, 4… do …n-1. Jeżeli żadna z tych liczb nie dzieli naszej liczby, to oczywiście jest ona pierwsza. Metoda ta jest zwana naiwną i jako jedyna daje nam 100% pewności, że liczba jest liczbą pierwszą.

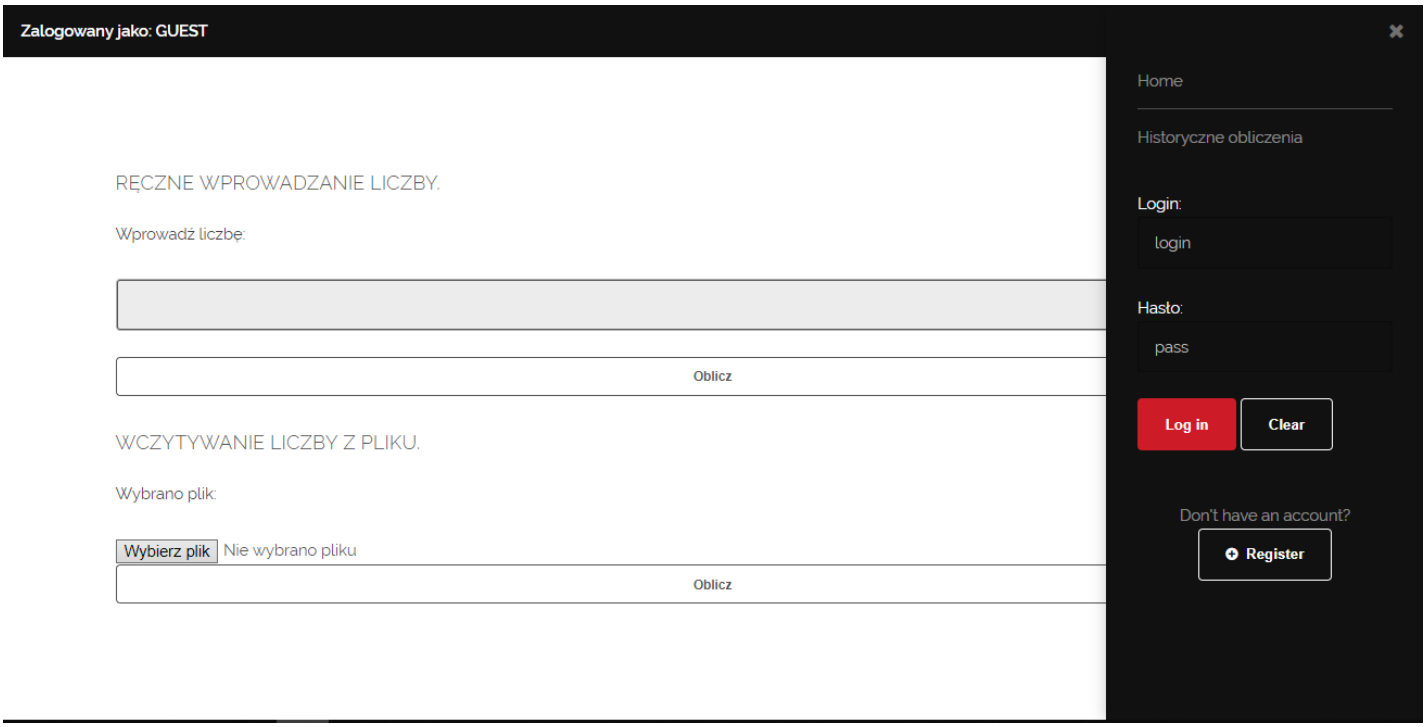
5.5 Mechanizm komunikacji komponentów

Nasze programy komunikują się ze sobą wykorzystując wspólną bazę danych.

5.6 Opis dodatkowej aplikacji pobierającej wynik

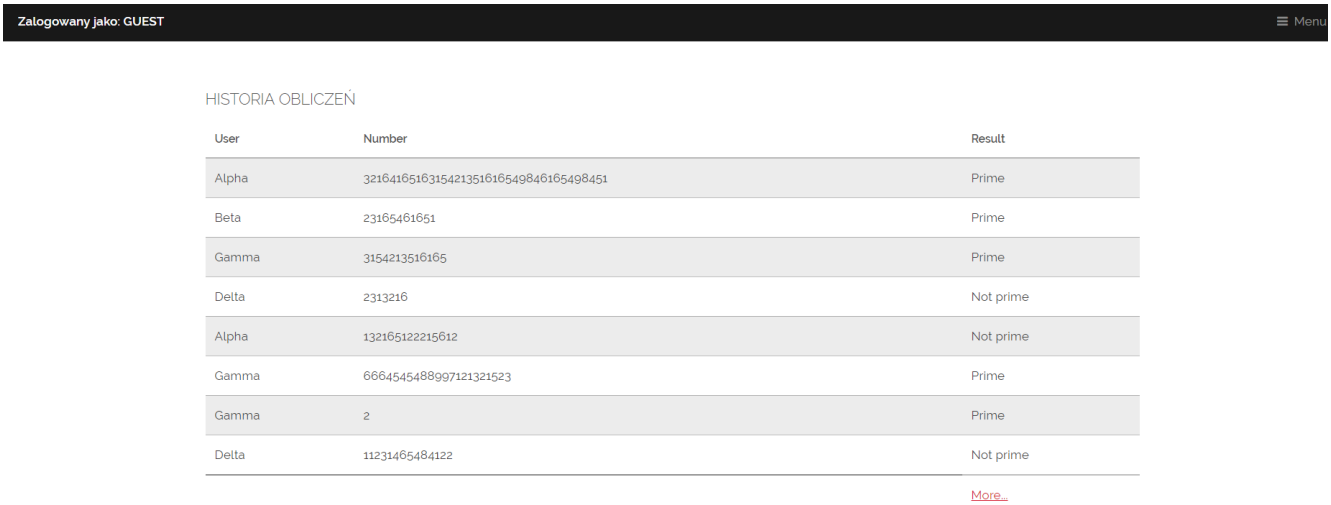
Aplikacja Poller.py co 2 sekundy sprawdza czy w danym momencie nie ma nowego zadania do rozwiązania. Jeżeli w bazie danych jest zadanie, które czeka na rozwiązanie wtedy aplikacja Poller.py uruchamia skrypt wykonujący to zadanie. Tym samym zawiesza się na czas obliczeń.

6. Frontend

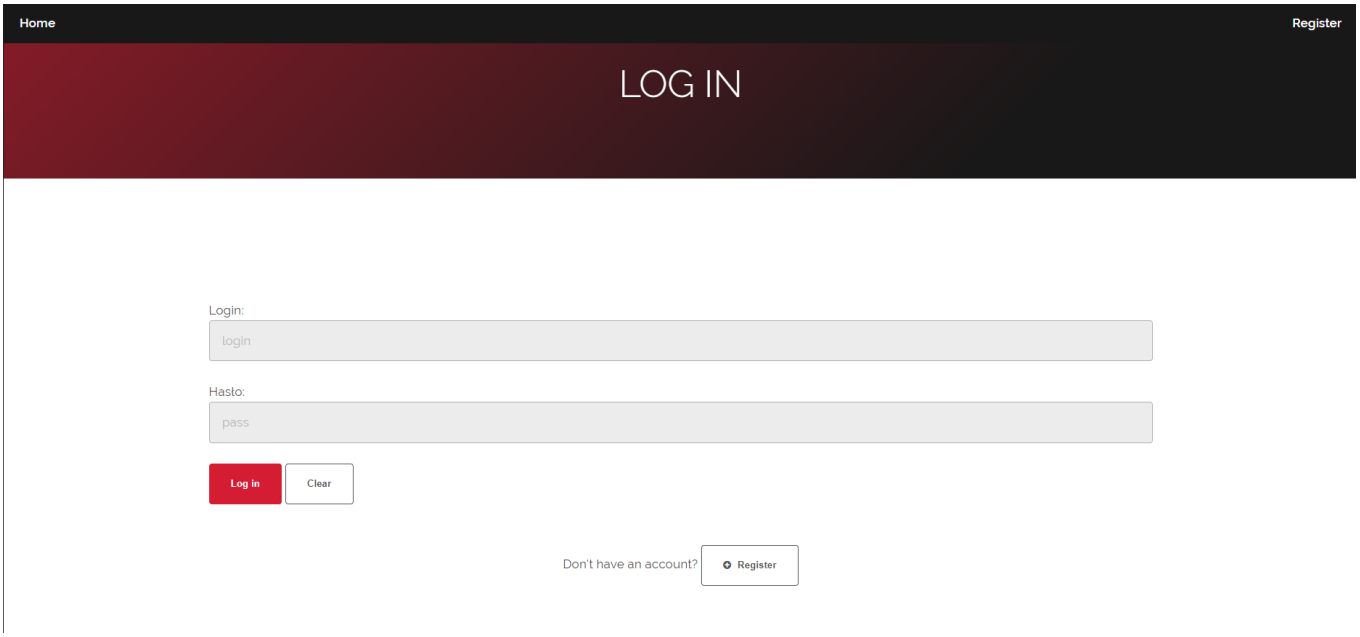


Screen 2. Widok głównego panelu na stronie www

Aplikacja webowa została napisana w HTML'u z wykorzystaniem CSS (cascading style sheets) oraz skryptów JavaScript. Tak jak zakładaliśmy podczas pierwotnego określania funkcjonalności zdołaliśmy zrealizować panel główny strony na którym użytkownik ma możliwość wprowadzenia liczby do sprawdzenia (ręcznie lub z pliku), rejestrację użytkowników oraz logowanie a także podgląd historii wykonanych obliczeń.



Screen 3. Widok historii obliczeń na stronie www

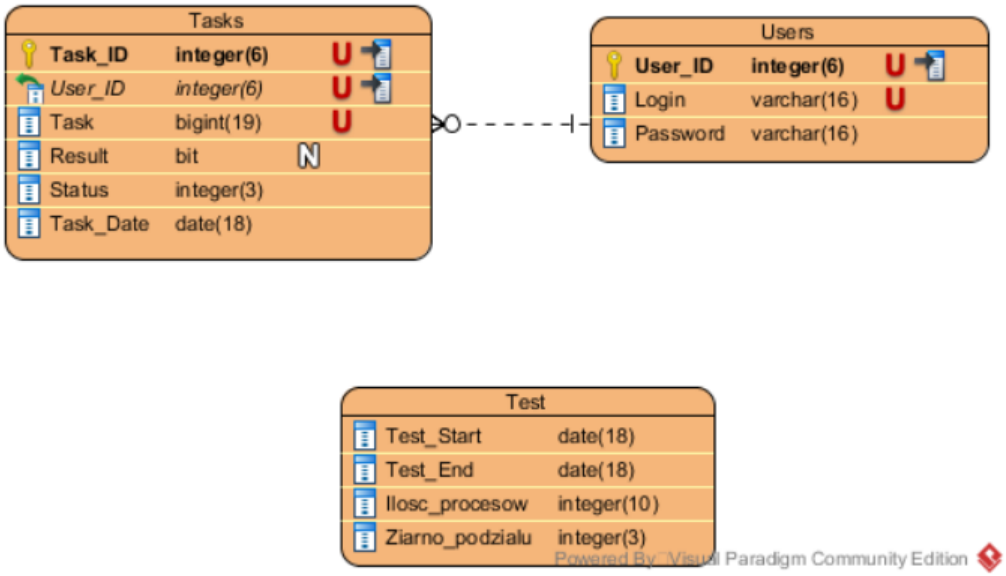


Screen 4. Widok logowania/rejestracji na stronie www

7. Backend

Na backend aplikacji składają się dwa główne komponenty. Jednym z nich jest baza danych, w której zapisywane są wszystkie informacje potrzebne do rozpatrzenia problemów zadanych przez użytkownika aplikacji. Drugim z nich jest właściwy algorytm wyznaczający czy liczba jest pierwsza. Dodatkowo algorytm wykorzystuje protokół MPI, gwarantując nam obsługę klastrów komputerowych.

7.1 Baza danych



Screen 5. Diagram ERD

Do utworzenia bazy danych, potrzebnej do realizacji naszego zadania wykorzystano język zapytań SQL. Powyżej znajduje się diagram ERD obrazujący relacje w naszej bazie danych.

7.2 Algorytm i wykorzystanie protokołu MPI

Do rozwiązania zadanego nam problemu, a mianowicie sprawdzenia czy zadana liczba jest liczbą pierwszą wykorzystaliśmy metodę Brute Force. Liczba, którą wprowadził użytkownik jest dzielona na przedziały. W kolejnym kroku te właśnie przedziały, wraz z wprowadzoną liczbą zostają wysłane do istniejących klastrów komputerowych za pomocą protokołu MPI. Problem ten realizowany jest na zasadzie relacji klient-serwer (do stworzenia wyżej wymienionej aplikacji rozproszonej wykorzystaliśmy język programowania Python). Gdy każdy klaster obliczeniowy otrzyma zlecone zadanie przystępuje do działania. W tym wypadku polega to na podzieleniu zadanej liczby, przez wszystkie kolejne liczby w danym przedziale. W ten sposób sprawdzane jest czy któraś z liczb w przedziale jest dzielnikiem naszej wejściowej liczby. Gdy klaster obliczeniowy skończy wykonywać zadanie, zwraca wynik do serwera - który po zebraniu wszystkich danych od poszczególnych komputerów stwierdza czy zadana liczba jest pierwsza.

7.3 Flask

Nasze backend'owe rozwiązania zawarte w projekcie zostały napisane w Pythonie, przez co zdecydowaliśmy się na skorzystanie z mikro-framework'a Flask. Python znakomicie nadaje się do tworzenia aplikacji internetowych dzięki takim rozszerzeniom jak micro-framework Flask. Upraszcza on projektowanie zapewniając przejrzysty schemat łączenia adresów URL, źródła danych, widoków i szablonów. Domyślnie dostajemy również deweloperski serwer WWW, nie musimy instalować żadnych dodatkowych narzędzi typu LAMP (WAMP). Flask traktowany jest jako „mikro" framework ponieważ nie wymaga on specjalnych narzędzi czy bibliotek. Dzięki Flask'owi jesteśmy w stanie rozdzielić zadania na poszczególne procesy wykonywane na serwerze.

```python
from flask import Flask, render_template, request
from multiprocessing import Process
import os;



def run_mpi_script(num, proc):
    os.system("mpiexec -n " + proc + " python mpts.py " + num)


app = Flask(__name__)

@app.route('/index.html', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        if __name__ == '__main__':
            num = request.form['number']
            proc = "4"
            p = Process(target=run_mpi_script, args=(num, proc))
            p.start()
        return render_template("index.html")
    else:
        return render_template("index.html")


@app.route('/calc_history.html')
def hist():
    return render_template("calc_history.html")

@app.route('/login.html')
def login():
    return render_template("login.html")


if __name__ == "__main__":
    app.run()
```

Screen 5. Zapisu kodu mikro-framwork'a Flask

8. Komunikacja backendu z frontendem

Aby nasza aplikacja w pełni spełniała wymagania wymienione w punktach 3 i 4 zdecydowaliśmy się na wykorzystanie framework'u Booststrap. Do komunikacji z bazą danych, która została stworzona w SQL'u, wykorzystujemy jedną z podstawowych funkcji php, a mianowicie mysql_connect, co umożliwia użytkownikom aplikacji swobodą rejestrację oraz logowanie na stronie. Kolejną funkcją php, którą wykorzystujemy do komunikacji pomiędzy aplikacją, a bazą danych jest funkcja session_start(), która pozwala na odczytanie sesji aktualnie zalogowanego użytkownika. Pozwala to na rozróżnienie poszczególnych zadań zleconych serwerowi.

```python
from mpi4py import MPI
import sys
import time
import os

if (len(sys.argv) != 2):
    prime = 0
else:
    prime = int(sys.argv[1])


milli_sec = int(round(time.time() * 1000))
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
primes = int(prime ** (1/2))
proc = comm.Get_size()
prime_range = int(primes/proc)
diffrence = primes - prime_range*proc
result = 0
plik = open("result.txt", "w")
if rank == 0:
    for i in range((prime_range*proc)+1, primes+1, 1):
        if ((i != 1) and (i != prime)):
            if (prime % i) == 0:
                result = 1
                break
for i in range((rank*prime_range)+1, ((rank+1)*prime_range)+1, 1):
    if ((i != 1) and (i != prime)):
        if (prime % i) == 0:
            result = 1
            break
data = comm.gather(result, root=0)

if rank == 0:
    for r in data:
        if r == 1:
            result = 1
            break
    milli_sec = int(round(time.time() * 1000)) - milli_sec
    if prime == 0:
        result = 1
    if result == 1:
        print("number:", prime, "is not prime")
        print("execution time:", milli_sec, "milliseconds")
        open("result.txt", "w").write("number: " + str(prime) + " is not prime" +
                "       execution time: " + str(milli_sec) + " milliseconds")
    else:
        print("number:", prime, "is prime")
        print("execution time:", milli_sec, "milliseconds")
        open("result.txt", "w").write("number: " + str(prime) + " is prime" +
                "       execution time: " + str(milli_sec) + " milliseconds")
```

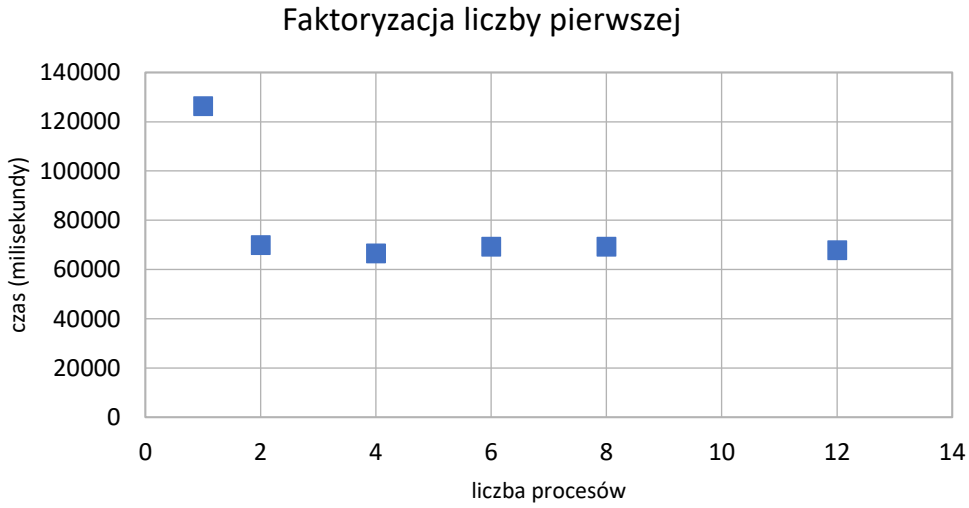
Screen 6. Zapisu kodu algorytmu z protokołem MPI

9. Badania

Nasze testy zostały przeprowadzone dla liczby pierwszej o wartości: 9224305113841897 oraz liczbie procesów równej kolejno: 1, 2, 4, 6, 8 oraz 12. Zmierzyliśmy czas wykonywania poszczególnych obliczeń, a uśrednione wyniki przedstawiamy w poniższej tabeli i wykresie.

liczba procesów	czas (milisekundy)
1	126350
2	69941
4	66470
6	69277
8	69261
12	67768

Tabela 1. Czas wykonywania obliczeń w zależności od liczby procesów



Wykres 1. Zależność czasu od ilości procesów dla wykonywanych obliczeń

Wyniki przeprowadzonego eksperymentu wskazują dużą poprawę wydajności przy rozdzieleniu zadania na ilość procesów równej liczbie fizycznych rdzeni procesorów. Przy dalszym podziale zadania nie widać znacznej poprawy czasu wykonywania zadania. Przewidujemy dalszą poprawę wydajności przy rozdzieleniu zadania na większą liczbę fizycznych rdzeni procesora. Aby zaobserwować wykładniczy spadek czasu wykonywania

obliczeń konieczne jest przeprowadzenie testów na maszynie z dostępem do większych zasobów, czego z powodu niewystarczającego czasu na zajęciach niestety nie udało się zrealizować.

10. Dokumentacja powykonawcza

10.1 Wymagania

- Linux Ubuntu 16.04
- Python 3.6.5
- MPI 3.2
- Flask 1.0.2
- MySQL ver 14.14 Distrib 5.7.22 for Linux (x86_64)

10.2 Działanie

Aby nasza aplikacja działa poprawnie uruchamiamy bazę danych mySQL. Następnie w terminalu Linux uruchamiamy skrypt poller.py za pomocą komendy: sudo python3 poller.py. Kolejnym krokiem jest instalacja wszystkich potrzebnych komponentów za pomocą komendy: sudo pip3 [komponent]. Aby uruchomić naszą aplikację korzystamy z komendy: FLASK_APP = flaskhello.py flash run. Wszystkie operacje wykonywane są w terminalu, tam wpisujemy wszystkie instrukcje.

Komendy w terminalu:

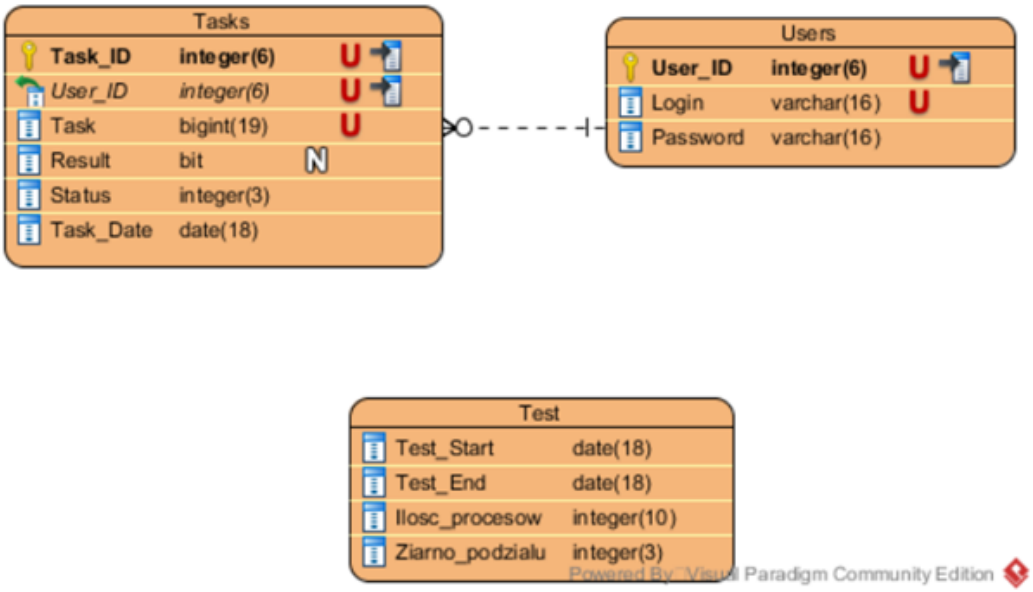
sudo python3 poller.py

sudo pip3 [komponent]

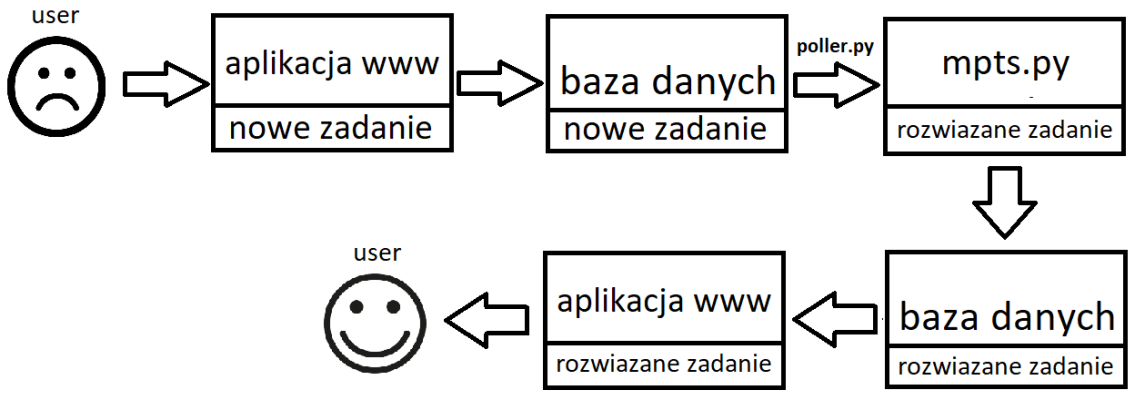
FLASK_APP = flaskhello.py flash run

10.3 Architektura systemu

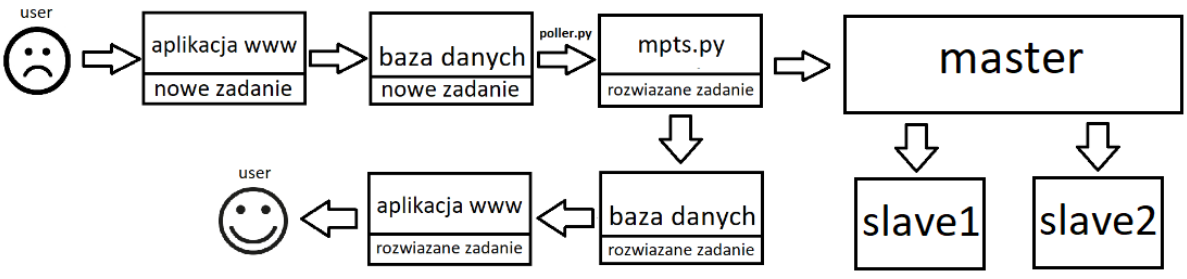
Schemat bazy danych:



Logika działania:



Logika działania z uwzględnionym Masterem i Slave'mi:



Opis schematu:

Użytkownik, który chce sprawdzić czy zadana liczba jest pierwsza loguje się na stronie WWW. Następnie zleca nowe zadanie, problem do rozwiązania przechowywany jest we wspólnej bazie danych. Skrypt pooler.py zauważa nowe, nierozwiązane zadanie w bazie danych i uruchamia skrypt odpowiedzialny za obliczenia. Podczas działania tego procesu przedziały są równo dzielone dla każdego slave'a, tj. każdy slave dostaje taki sam przedział liczb. W tym momencie nasza wejściowa liczba jest dzielona przez każdą liczbę z przedziału, w ten sposób sprawdzane jest czy zadana liczba należy do liczb pierwszych. Następnie wyniki przekazywane są do Mastera, który je analizuje i przekazuje do bazy danych, jako rozwiązane zadanie. Jest to wspólna baza danych, dlatego też użytkownik dzięki aplikacji WWW jest w stanie zobaczyć rezultat działania naszej aplikacji.

10.4 Konfiguracja master'a i slave'ów

Aby skonfigurować komputer do roli slave'a należy zmienić zawartość pliku hosts, a następnie postępować zgodnie z instrukcjami:

```
$ cat /etc/hosts
127.0.0.1       localhost
172.50.88.34    master //gdzie zamiast powyższego adresu należy wprowadzić adres ip master'a
```

Dodać użytkownika:

```
$ sudo adduser mpiuser
```

Skonfigurować połączenie ssh między dwoma, lub większą ilością komputerów:

```
$ sudo apt-get install openssh-server
$ su - mpiuser
$ ssh-keygen -t dsa
$ ssh-copy-id master
$ eval `ssh-agent`
$ ssh-add ~/.ssh/id_dsa
$ ssh master
```

Skonfigurować klienta NFS:

```
$ sudo apt-get install nfs-common
$ mkdir cloud
$ sudo mount -t nfs master:/home/mpiuser/cloud ~/cloud
$ cat /etc/fstab
```

Aby skonfigurować komputer do roli master'a należy zmienić zawartość pliku hosts, a następnie postępować zgodnie z instrukcjami:

```
$ cat /etc/hosts
127.0.0.1       localhost
172.50.88.34    client //gdzie zamiast powyższego adresu należy wprowadzić adres ip slave'a
```

Dodać użytkownika:

```
$ sudo adduser mpiuser
```

Skonfigurować połączenie ssh między dwoma, lub większą ilością komputerów:

```
$ sudo apt-get install openssh-server
$ su - mpiuser
$ ssh-keygen -t dsa
$ ssh-copy-id client
$ eval `ssh-agent`
$ ssh-add ~/.ssh/id_dsa
$ ssh client
```

Skonfigurować serwer NFS:

```
$ sudo apt-get install nfs-kernel-server
$ mkdir cloud
$  cat /etc/exports/home/mpiuser/cloud *(rw,sync,no_root_squash,no_subtree_check)
$ exportfs -a
$ sudo service nfs-kernel-server restart
```

Po wykonaniu powyższych kroków można uruchomić aplikację. W tym celu należy skopiować pliki wykonywalne aplikacji do folderu cloud. Następnie należy za pomocą komendy uruchomić aplikację:

```
$ mpirun -np 5 -hosts client,localhost ./cpi
```

Gdzie 5 jest maksymalną liczbą rdzeni procesora jaką chcemy wykorzystywać (zwykle maksymalna liczba fizycznych dostępnych rdzeni procesora z mastera i slave'ów)

11. Źródła aplikacji

Kod źródłowy aplikacji jest przechowywany w repozytorium publicznym, poniżej znajduje się odnośnik do tego repozytorium:

https://github.com/wojkd/AIiR