Factorer MPI - dokumentacja projektowa

Rafał Duraj, Piotr Dowgiałło, Bartosz Janusz, Maciej Kolański

2016-06-07

Spis treści

1 Temat projektu

W dzisiejszych czasach, gdy właściwie wszystko co robimy w jakiś sposób połączone jest z Internetem bezpieczeństwo jest bardzo ważnym tematem.

Faktoryzacja jest to proces podczas którego dla zadanego obiektu odnajduje się inne obiekty, które spełniają to, że ich iloczyn równy jest oryginalnemu obiektowi, w związku z czym te znalezione czynniki są w pewnym sensie od niego prostsze.

Podstawowy algorytm faktoryzacji bazuje na próbowaniu podziału liczby do faktoryzacji n przez wszystkie liczby pierwsze od 2 do $\sqrt{n}$. Tego typu algorytm bardzo dobrze radzi sobie z początkiem faktoryzacji liczby, bo dowolna liczba ma czynnik zarówno małe jak i duże. Jak wiadomo połowa wszystkich liczb dzieli się przez dwa, jedna trzecia licz przez trzy i tak dalej, a więc z dużym prawdopodobieństwem można pozbyć się w prosty sposób niskich czynników.

RSA jest to jeden z pierwszych i też obecnie najpopularniejszych asymetrycznych algorytmów kryptograficznych gdzie klucz jest publiczny. Bezpieczeństwo szyfrowania przy pomocy tego algorytmu jest związane z trudnością faktoryzacji dużych liczb.

2 Zakres projektu

2.1 Cel

Celem projektu jest umożliwienie zlecenia zadania faktoryzacji dużej liczby (większych od $2^{64} - 1$). Jednym z głównych założeń jest udostępnienie prostego w obsłudze interfejsu użytkownika i zmaksymalizowanie elastyczności – system powinień być zdolny do współpracy z zadanymi komputerami, a instalacja wymaganego oprogramowania musi być prosta.

2.2 Funkcjonalność

Podstawowa

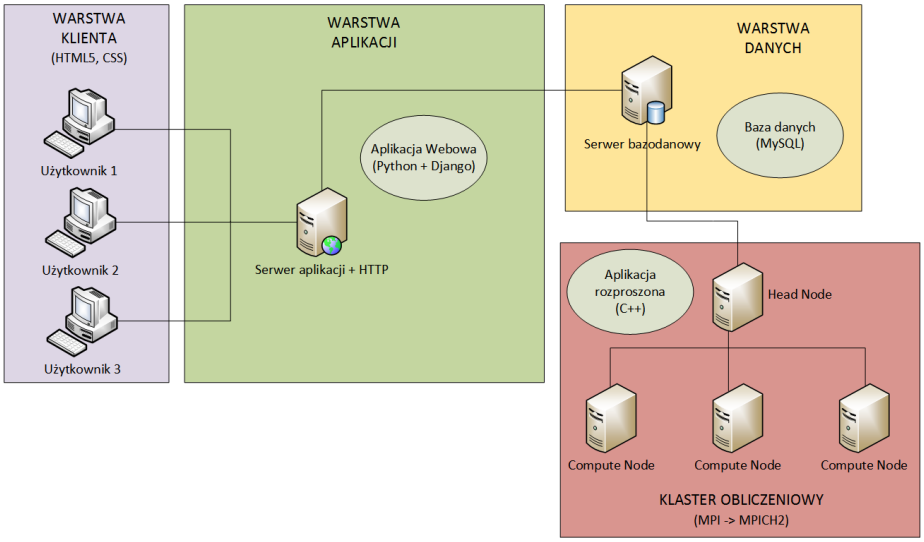
1. Udostępnienie mechanizmów rejestracji i logowania (konta użytkowników)
2. Zarządzanie zadaniami z poziomu interfejsu webowego
   (a) zlecanie zadań
   (b) przegląd historii
3. Możliwość rozbudowy klastra bez ingerencji w kod
4. Faktoryzacja metodą „brutalnej siły”
5. Faktoryzacja metodą CFRAC
6. Zachowywanie wyników pomyślanie wykonanych faktoryzacji

Rozszerzona

1. Łamanie szyfru RSA
2. Faktoryzacja metodą sita kwadratowego
3. Forma graficznej prezentacji wyników pomiarów

3 Narzędzia i technologie zastosowane w projekcie

3.1 Zastosowane technologie



Rysunek 1: Schemat blokowy systemu

Strona klienta - HTML5, CSS Interfejs z którego bedzie korzystal klient projektowanego systemu został napisany w jezyku skryptowym HTML5 z użyciem stylów CSS. HTML5 jest obecnie standardem przy tworzeniu stron internetowych i w wiekszosci wyparl HTML4, w którego specyfikacji bylo duzo niescislosci. Uzycie CSS z kolei pozwoli ujednolicic prezentacje zawartosci w różnych przegladarkach, oraz uprosci organizacje samego kodu.

Aplikacja Webowa - Python 3.4 + Django 1.8.7[1] Python jest jezykiem wysokiego poziomu ogólnego przeznaczenia, z kolei Django to framework webowy dla tego jezyka. Wybralismy ten zestaw z powodu wielu ulatwień przy

tworzeniu aplikacji webowych, które sa przezeń oferowane, np. dynamiczny interfejs bazy danych, automatyczny interfejst administracyjny. Dodatkowo czesc naszej grupy jest zaznajomiona z tymi technologiami, wiec nie ma potrzeby poznawania ich od zera. Istotne sa tutaj wersje srodowisk - najnowsze dystrybucje nie obsluguja MySQL, w zwiazku z tym wybralismy poprzednie.

Baza danych - MySQL SZBD rozwijany przez firme Oracle. Charakteryzuje sie wszystkimi najwazniejszymi funkcjonalnosciami bazy danych oraz prostota tworzenia takiej bazy. Rozważalismy zastosowanie systemu Oracle Database, jednakże jest on zbyt rozbudowany jak na nasze potrzeby, a co za tym idzie trudny w obsudze. Mamy również doswiadczenie w integracji bazy MySQL z aplikacjami napisanymi w Pythonie (Django).

Klaster obliczeniowy - standard MPI[2] MPICH2 to darmowa implementacja standardu MPI dla systemów Linux. Umożliwa proste tworzenie klastrów obliczeniowych, rozdzielania zadań miedzy poszczególne wezly i zbierania wyników. Oferuje interfejs dla jezyka C++. Jest wykorzystywany w wiekszosci topowych urzadzen wieloprocesorowych i ta popularnosc znaczaco wplynelo na jego wybór.

Aplikacaja rozproszona - C++11 Wybór padl na ten jezyk ze wzgledu na jego znajomosc przez czlonków grupy oraz interfejs udostepniany przez srodowisko MPICH2.

3.2 Narzedzia wykorzystane w projekcie

- Aplikacja webowa PyCharm 5.0.4 - IDE do Pythona, obsuguje Django
- Aplikacja rozproszona - CodeLite 9.1 IDE do C++, wersja na system Linux
- Baza danych - developer do MySQL
- Organizacja pracy - Trello (`https://trello.com/`)
- Hosting plików - GitHub (`https://github.com/`)

4 Aktualny stan rynku[6]

GGNFS (GPL'd implementation of General Number Field Sieve) Aktywny rozwój. faktoryzacja liczb do 180 znaków, srednio do 140. Kilka wiekszych liczb tez. W wiekszosci przypadkow program GGNFS jest stabilny dla liczb skladajacych sie do 150-160 znakow. Posiada bugi. Nie jest czarna skrzynka, trzeba miec odpowiednia wiedze zeby go uzywać.

Cunningham Project Projekt faktoryzujacy liczby b^n+-1 dla b=2,3,5,6,7,10,11,12 i duze n.[3]

RSA Factoring Challenge Zawody zorganizowane przez RSA Security. Otwarte zawody dla wszystkich mające na celu zwiększyć zainteresowanie faktoryzacją liczb. Opublikowana została lista pseudopierwszych liczb (rozkładających się na dokładnie dwa czynniki), nazwanych liczbami RSA.Za rozłożenie niektórych z nich wyznaczono pieniężną nagrodę. Najmniejsza z nich, 100-cyfrowa liczba RSA-100 została rozłożona w ciągu kilku dni, ale większość do dziś pozostaje niezłamana.Zawody miały na celu śledzenie rozwoju możliwości komputerów w faktoryzacji. Jest to niezwykle istotne przy wyborze długości klucza w szyfrowaniu asymetrycznym metodą RSA. Postęp w łamaniu kolejnych liczb powinien zdradzać jakie długości klucza można jeszcze uznawać za bezpieczne.[4]

Poczatkowo rekordy były bite za pomoca algorytmu CFRAC, który ustapił dopiero w latach 8X algorytmowi sita kwadratowego. Za pomoca algorytmu QS sfaktoryzowano, miedzy innymi, liczby $RSA-100, RSA-110, RSA-120 i RSA-129$. Następnie na scene wkroczył algorytm sita ciałliczbowego, który do dzis pozostaje najszybszym ze znanych algorytmów faktoryzacji i dzierży rekordy - najpierw RSA-640 (640 bitów, 193 cyfry) i RSA-200 (200 cyfr) w 2005 roku, a następnie RSA-768(232 cyfry) w 2009 roku, co pozostaje niepobitym rekordem do dzis.

5 Projekt techniczny

5.1 Klaster obliczeniowy - Komunikacja

Program realizujący obliczenia został zrealizowany w oparciu o architekturę Master-Slave, a użytkownik systemu może uruchamiać kolejne zadania poprzez komunikację z węzłem Master. W tym celu określony został interface *FactorerCommunictorInterface* i jego dwie implementacje, z których jedna służy do komunikacji z klastrem przy użyciu konsoli (uruchomienie programu z parametrem cmd), a druga z bazą danych (stroną internetową). Metoda *getCommand* pobiera polecenie (od użytkównika lub bazy danych), a *algorithmFinished* przesyła wynik obliczeń. Takie podejście pozwala na abstrakcyjne podejście do tematu komunikacji - algorytm nie jest powiązany z bazą lub stroną internetową i z łatwością można dodać implementację pobierającą danę z innego źródła.

Listing 1: Interface wykorzystany do komunikacji

```
enum CommunicatorCommand{Quit, Algorithm};

class FactorerCommunicatorInterface
{
public:
    virtual ~FactorerCommunicatorInterface() {};
```

```
virtual CommunicatorCommand getCommand(MPIAlgorithm::AlgorithmsEnum
    &algorithm, std::string &number ) = 0;
virtual void algorithmFinished(const std::vector<std::string>
    &result) = 0;

static std::string commandAsString(CommunicatorCommand command)
```

Najważniejszym dla projektu sposobem komunikacji jest implementacje łącząca klaster z bazą danych - to ona umożliwia zlecanie zadań z poziomu strony internetowej. W trakcie realizacji projektu rozważano także podejście wykorzystujące bezpośrednie połączenie pomiędzy serwerem strony, a węzłem master, zdecydowano się jednak na wykorzystanie pośredniczącej bazy danych z następujących przyczyn:

- Użycie Django właściwie wymaga wykorzystania bazy danych do przechowywania danych, więc element musiał pojawić się w projekcie.
- Jednocześnie Django zapewnia bardzo dobre wsparcie w komunikacji z bazą danych.
- Podejście tego typu zapewnia modularność - do systemu można dodać klienta innego typu niż strona WWW (np. aplikację mobilną) z zerowym kosztem, dane byłby przesyłane do bazy dokładnie w ten sam sposób, sam klaster nie musi znać ich źródła.
- Liczba klastrów także staje się nieograniczony, każdy może pobierać wolne zadanie z bazy danych w dowolnej chwili.

5.2 Klaster obliczeniowy - Algorytm Brute Force

5.2.1 Opis algorytmu

Zaimplementowany algorytm przeglądu zupełnego polega na wyznaczeniu maksymalnej wartości czynnika - czyli $\sqrt{n}$ i określenie zestawu przedziałów. Następnie każdy węzeł wykonawczy przedział $[x, y], x > 1, y <= \sqrt{n}$ i testuje, czy któraś liczba z tego przedziału dzieli się bez reszty przez n. Jeśli tak, algorytm znalazł dwa czynniki określające szukaną liczbę, następuję przerwanie i rekurencyjne wywołanie algorytmu dla znalezionych elementów.

5.3 Klaster obliczeniowy - Algorytm CFRAC

5.3.1 Opis algorytmu[5][6]

Metoda CFRAC jest algorytmem faktoryzacji liczb całkowitych. Jest to uniwersalny algorytm będący w stanie rozłożyć na czynniki każdą liczbę, nie polegając na żadnych ograniczeniach czy warunkach. Został on opisany przez D.H.

Lehmer'a oraz R. E. Powers'a w 1931 roku, oraz wdrożony na komputery pierwszy raz przez Michael'a A. Morisson'a oraz John'a Brillhart'a w 1975 roku.

Algorytm ten bazuje na metodzie faktoryzacji Diaxona. Metoda Diaxona polegała na losowaniu kolejnych liczb 'a' takie, że: $sqrt[n] < a < n$ (gdzie n to liczba, którą chcemy sfaktoryzować) i sprawdzamy(używając algorytmu naiwnego), czy $b^2 = a^2 mod(n)$ jest liczbą B-gładką, dla ustalonego B. Jeżeli tak to dodajemy znalezioną parę do zbioru (liczba B-gładka to taka liczba, której wszystkie dzielniki pierwsze są mniejsze bądź równe dla ustalonego B).

W algorytmie CFRAC idea pozostaje bez zmian, definiowany jest natomiast sposób wybierania par. Zamiast losowania ich wykorzystywany jest ciąg rozwinięcia sqrt(n) w ułamek łańcuchowy. Złożoność obliczeniowa algorytmu CFRAC jest rzędu $O(e^{\sqrt{2*\log n * \log\log n}})$.

5.3.2 Implementacja[7]

Program generuję rekurencyjnie elementy tablicy, której elementy są ze sobą ściśle powiązane, następującymi wzorami:

$$
\begin{aligned}
Q[n] &= Q[n-2] + q[n-1] * (r[n-1] - r[n-2]) \\
G[n] &= 2g - r[n-1] \\
q[n] &= \frac{G[n]}{Q[n]} \\
r[n] &= G[n] - q[n] \\
A[n] &= q[n] * A[n-1] + A[n-2] mod N
\end{aligned}
$$

gdzie:

$$
\begin{aligned}
Q[-1] &= N \\
Q[0] &= 1 \\
q[0] &= g \\
r[-1] &= g \\
r[0] &= 0 \\
A[-1] &= 1 \\
A[0] &= g \\
g &= \sqrt{N}
\end{aligned}
$$

Dla powyższych reguł generowane są rekurencyjnie kolejne rekordy. Przy wygenerowaniu każdego kolejnego "zestawu" wyników badany był element Q[i]. Jeżeli był on możliwy do spierwiastkowania(reszta z pierwiastka kwadratowego

jego wartości była równa zeru), to następnym krokiem w celu uzyskania szukanego faktora było obliczenie następującej wartości:
$temp = A[i-1] - \sqrt{Q[i]}$

Ostatnim krokiem było obliczenie NWD (największego wspólnego dzielnika)pomiędzy uzyskaną wartością (temp), a liczbą dla której szukamy faktora(N). Wykorzystanym algorytmem do obliczania NWD był algorytm euklidesa, który jest aktualnie najefektywniejszym algorytmem wykorzystywanym do tej operacji.

5.3.3 Podział zadań (Master Slave)

Z powodu rekurencyjnego generowania wyników, efektywny podział pracy pomiędzy różne maszyny(slave'y) był bardzo ciężki do zrealizowania. Ostatecznie udało nam się wymyślić sposób podziału prac, który może nie jest najlepszy ale zapewnia w pewnym stopniu poprawe wydajności przy wykorzystaniu wielu maszyn.

Polega on na wprowadzeniu do algorytmu parametru K. Dla każdej maszyny oprócz danej liczby do faktoryzacji wysyłamy również indywidualny dla niej parametr, przez który mnoży on otrzymaną liczbę do faktoryzacji. Dla nowej uzyskanej liczby k*N, algorytm znajduję wyniki w innym czasie. Wadą tego rozwiązania jest nie zawsze poprawny wynik, dlatego po jego odebraniu trzeba sprawdzić jego poprawność (np odrzucić wyniki, które dzielą się przez k)

Należy też wspomnieć, że CFRAC jest efektywnym algorytmem tylko dla dużych liczb, dlatego w celu znacznego skrócenia czasu operacji mniejsze liczby faktoryzowane są prostym algorytmem naiwnym przez maszyne Master.

Algorytm podziału zadań przez maszyna Master wygląda w następujący sposób:

Listing 2: Sekwencja instrukcji algorytmu CFRAC

```
1    Utworz kolejke na wynik
2    Wrzuc pierwsza liczbe (N) do kolejki liczb do faktoryzacji
3    Dopoki kolejka nie jest pusta
4        Dopoki nie znalazles ostatniej liczby pierwszej
5            Wyciagnij liczbe A z kolejki
6            Jezeli A jest liczba duza //(efektywny CFRAC)
7                Zainicjuj maszyny (Slaves)
8                Rozeslij zadania wraz z indywidualnym parametrem K
9                Wyrzuc aktualna liczbe A z kolejki
10               Wykonuj dopoki nie odbierzesz wynikow od wszystkich maszyn
11                   Czekaj na wynik
12                   Sprawdz jego poprawnosc
13                   Jezeli wynik jest poprawny:
```

```
14                  Jezeli jest liczba pierwsza, wrzuc do kolejki
                        wynikow
15                  Jezeli nie jest liczba pierwsza wrzuc go oraz jego
                        iloraz z liczba A do kolejki liczb do
                        faktoryzacji
16              Jezeli zadna maszyna nie zwrocila poprawnego wyniku,
                    zmien parametr k i wrzuc z powrotem liczbe A do
                    kolejki liczb do faktoryzacji
17          Dla malej liczby uzyj algorytmu naiwnego (bruteforce)
```

5.4 Strona internetowa

Dostępna w sieci Internet strona WWW finalnie umożliwia rejestrację i logowanie użytkowników oraz zlecanie zadań, czyli liczb do faktoryzacji. Do tego użytkownik ma wgląd w historię zleconych przez niego zadań wraz z ich rozwiązaniami. Poza funkcjonalnościami dla zwykłego użytkownika strona umożliwia także działania administracyjne poprzez panel administracyjny. Do zadań tych należą takie rzeczy jak usuwanie zleconych zadań, nadawanie im priorytetu czy też zarządzanie użytkownikami. Strona współpracuje z bazą danych MySQL. Jest ona dostępna pod adresem `http://156.17.235.48/`.

Number	User	Algorithm	Date	State	Priority
128	admin	CFRAC	May 24, 2016	2	0
123458756	admin	Brute Force	May 24, 2016	2	0
76348732	admin	Brute Force	May 24, 2016	2	0
126	test	Brute Force	May 24, 2016	2	0
21212121	test	Brute Force	May 23, 2016	2	0
21212121	test	Brute Force	May 23, 2016	2	0
1000000000	test	Brute Force	May 23, 2016	2	0
3727481741	test	CFRAC	May 23, 2016	2	0
3727481741	test	Brute Force	May 23, 2016	2	0
1111	test	Brute Force	May 23, 2016	2	0

Login	Data dołączenia
admin	May 4, 2016, 7:49 p.m.
test	May 4, 2016, 7:50 p.m.

Rysunek 2: Strona główna.

Stronę wykonano przy użyciu technologii Python Django[1] i edytora tekstowego Sublime[8] wraz z dodatkiem Anaconda Python IDE[9]. Do komunikacji z bazą danych wykorzystano standardową bibliotekę do połączenia z MySQL dostępną w Django. Baza danych została stworzona przy pomocy ORM[10] także standardowego dla Django. Przykładowy kod dla encji Task poniżej.

Listing 3: Przykładowa implementacja encji w kodzie Python

```
1  class Task(models.Model):
2      CANCELLED_STATUS = -1
3      UNDONE_STATUS = 0
4      WORKING_STATUS = 1
5      DONE_STATUS = 2
6      STATUS_CHOICES = (
7          (CANCELLED_STATUS, "Cancelled"),
8          (UNDONE_STATUS, "Undone"),
9          (WORKING_STATUS, "Working"),
10         (DONE_STATUS, "Done")
11     )
12
13     number_to_factor = models.CharField(max_length=200)
14     job_date = models.DateField(default=timezone.now)
15     state = models.IntegerField(choices=STATUS_CHOICES,
           default=UNDONE_STATUS)
16     priority = models.IntegerField(default=0)
```

```
    result = models.CharField(max_length=200)
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    algorithm = models.ForeignKey(Algorithm, on_delete=models.CASCADE)

    def __str__(self):
        return str(self.number_to_factor)
```

Kolejne kolumny danej tabeli w bazie danych są tworzone poprzez definiowanie pól klas dziedziczących po models.Model[11], tak jak powyżej na przykład kolumna result jest polem tekstowym o długości maksymalnej 200 znaków.

W celu utworzenia nowej podstrony definiuje się klasy dziedziczące po klasie View z Django. Poniżej znajduje się kod dla widoku podstrony z zadaniami użytkownika.

Listing 4: Klasa widoku obsługująca podstronę

```
class UserView(LoggedInMixin, View):
    template_name = 'FactorerMain/userview.html'

    def get(self, request, *args, **kwargs):
        tasks = Task.objects.filter(user=request.user.id)[::-1]

        context = {'tasks': tasks}

        return render(request, self.template_name, context)
```

W tym przypadku UserView dziedziczy również po klasie LoggedInMixin odpowiadającej za autoryzację, dzięki czemu do takiej podstrony ma dostęp tylko użytkownik zalogowany. W zmiennej template_name wskazano na szablon strony opisany w języku HTML. Następnie w metodzie get pobierane są wyniki z bazy danych przy pomocy metody Task.objects.filter(user=request.user.id)[::-1]. Zwróci to z encji Task wyniki należące do aktualnego użytkownika posortowane w od najnowszych do najstarszych. Na końcu zwracany jest szablon oraz kontekst, w tym przypadku zadania użytkownika, poprzez metodę render.

Można tak pobrane dane z bazy użyć w HTML i wyświetlić je na stronie. Kod wygląda następująco:

Listing 5: Szablon HTML używający odczytane dane z bazy

```
<h2>Moje zadania:</h2>
{% if tasks %}
    {% for task in tasks %}
        <details>
        <summary><p>Liczba <b>{{ task.number_to_factor }}</b> przy
            uzyciu {{ task.algorithm }} dnia {{ task.job_date }} jest w
            stanie {{ task.state }}</p></summary>
        {% ifequal task.state 2 %}
```

```
            {{ task.result }}
        </p>
        {% endifequal %}
        </details>
    </br></br>
    {% endfor %}
{% else %}
    <p>No tasks to show.</p>
{% endif %}
```

Pierw sprawdzane jest czy w kontekście została przekazana zmienna tasks. Następnie przy pomocy pętli for przechodzi się przez kolejne elementy tabeli i drukuje jej elementy przy pomocy składni [tabela].[element]. Przykład uzyskanych z bazy danych wyników na stronie pokazano na rysunku poniżej.

Rysunek 3: Podstrona z zadaniami użytkownika.

W Django zazwyczaj definiuje się jeden bazowy szablon w postaci pliku HTML, a następnie na podstronach zastępuje się tylko wyróżnione bloki strony w sposób wyspecyfikowany dla danej podstrony. Uzyskuje się to przez rozszerzanie szablonów. Dla przykładu w główny pliku HTML znajduje się następujący fragment:

Listing 6: Fragment głównego szablonu strony

```
<div id="portfolio-samples">
```

```
    <div class="wrap clearfix">
        {% block gallery %}
        {% endblock %}
    </div><!-- // end .wrap -->
</div><!-- // end #portfolio-samples -->
```

W tym kodzie zdefiniowano blok gallery, można go zastąpić przy pomocy poniższego fragmentu na innej stronie.

Listing 7: Fragment podstrony rozszerzającej bazowy szablon

```
{% extends 'base.html' %}

{% block gallery %}
<div class="portfolio-item fl col-2">
  <a href={% url 'bruteforce' %}><img src="static/images/al1.jpg"
      alt="thumb" /></a>
      <div class="info">Brute force</div>
  </div><!-- // end .portfolio-item -->
  <div class="portfolio-item fl push-2 col-2">
      <a href={% url 'cfrac' %}><img src="static/images/al2.jpg"
          alt="thumb" /></a>
      <div class="info">CFRAC</div>
  </div><!-- // end .portfolio-item -->
{% endblock %}
```

Poza blokiem gallery reszta strony będzie taka jak zdefiniowano w pliku base.html. Strona projektu składa się z jednego bazowego szablonu i rozszerzających go kilku podstron.

W celu połączenia wszystkiego należy zdefiniować jeszcze w pliku urls.py odwzorowania adresów url dla odpowiednich stron, dzięki czemu Django uruchamia odpowiednie klasy widoków dla zadanego adresu.

Listing 8: Definicja adresów URL

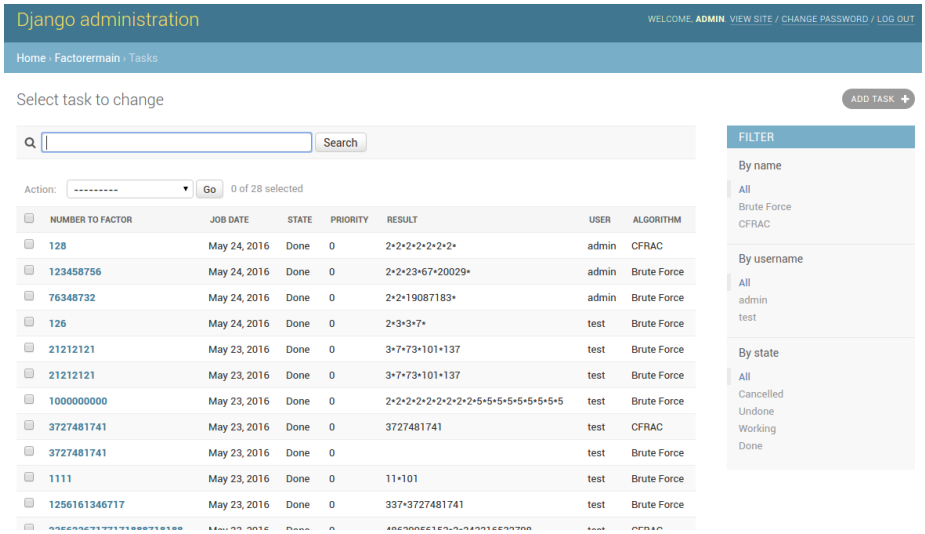
```
urlpatterns = [
    url(r'^$', views.IndexView.as_view(), name='index'),
    url(r'^userview/?$', views.UserView.as_view(), name='userview'),
    url(r'^about/?$', views.AboutView.as_view(), name='about'),
    url(r'^creators/?$', views.CreatorsView.as_view(), name='creators'),
    url(r'^bruteforce/?$', views.BruteforceView.as_view(),
        name='bruteforce'),
    url(r'^cfrac/?$', views.CFRACView.as_view(), name='cfrac'),
    url(r'^login/?$', 'django.contrib.auth.views.login'),
    url(r'^logout/?$', 'django.contrib.auth.views.logout'),
    url(r'^register/?$', views.register, name='register'),
    url(r'^success_register/?$', views.SuccessRegisterView.as_view(),
        name='success_register'),
    url(r'^admin/?', include(admin.site.urls)),
```

```
14 ]
```

Przy tak zdefiniowanych adresach odnosi się do nich w HTML poprzez następującą linię kodu:

```
<a href="{% url 'index' %}">Strona glowna</a>}
```

Do zadań administracyjnych został wykorzystany standardowy panel Django administration skonfigurowany dla potrzeb projektu.



Rysunek 4: Panel administracyjny Django.

Panel modyfikuje się poprzez edycję pliku admin.py.

Listing 9: Klasa konfigurująca panel administratora

```
1 class TaskAdmin(admin.ModelAdmin):
2     list_display = ('number_to_factor', 'job_date', 'state', 'priority',
          'result', 'user', 'algorithm')
3     search_fields = ['user__username']
4     list_filter = ('algorithm__name', 'user__username', 'state')
5
6 admin.site.register(Task, TaskAdmin)
```

Klasa musi dziedziczyć po klasie admin.ModelAdmin, następnie można zdefiniować sposób wyświetlania danych, pola, po których ma być realizowane wyszukiwanie, czy też listę filtrów. Na końcu trzeba zarejestrować taką klasę dla danej encji.

6 Dokumentacja powykonawcza (instalacyjna)

6.1 Uruchomienie klastra

6.1.1 Wymagania systemowe

System posiada następujące wymagania programowe:

- system operacyjny Ubuntu 14.04 LTS
- zainstalowana biblioteka MPICH2 w wersji 3.0.4
- klient SSH, np. openssh (1:6.6p1-2ubuntu2.7)
- serwer systemu wymiany plików NFS, np. nfs-kernel-server (1:1.2.8-6ubuntu1.2) - na głównej maszynie klastra (master)
- klient systemu wymiany plików NFS, np. nfs-common (1:1.2.8-6ubuntu1.2) - na pozostałych maszynach klastra (slaves)

Minimalne wymagania sprzętowe nie zostały sprecyzowane, system powinien się uruchomić na dowolnej konfiguracji z zainstalowanym powyższym oprogramowaniem. Niezbędne jest, aby wszystkie maszyny tworzące klaster były umieszczone w jednej sieci LAN pracującej w technologii FastEthernet lub wyższej. Dodatkowo węzeł główny musi posiadać połączenie z Internetem w celu komunikacji z bazą danych.

6.1.2 Konfiguracja systemu

Aby uruchomić system na docelowej grupie maszyn należy wykonać następujące kroki:

- zmapować adresy ip maszyn w pliku systemowym */etc/hosts*
- utworzyć na każdej maszynie nowego użytkownika *mpiuser*
- utworzyć folder współdzielony w sieci za pomocą systemu NFS
- zapewnić bezhasłowe połączenie SSH pomiędzy węzłem głównym, a każdym węzłem obliczeniowym
- skopiować plik wykonywalny programu do folderu współdzielonego
- uruchomić plik za pomocą komendy *mpirun*

Edycja pliku *hosts* Plik znajduje się w katalogu /etc. Należy edytować go w dowolnym edytorze tekstu i przypisać lokalne adresy ip do nazw hostów. Przykładowy plik:

```
127.0.0.1  localhost
#127.0.1.1 linux0

156.17.41.15  master
156.17.41.60  slave1
156.17.41.61  slave2
156.17.41.62  slave3

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

Tworzenie nowego użytkownika Aby utworzyć nowego użytkownika należy wykonać następującą komendę z uprawnieniami administratora:

```
$ sudo adduser mpiuser
```

Krok należy powtórzyć na każdym węźle.

Utworzenie folderu współdzielonego i udostępnienie go w sieci Po zalogowaniu na konto *mpiuser* należy dokonać następujących kroków:

Na głównym węźle:

Utworzenie folderu cloud:

```
$ mkdir cloud
```

Edycja pliku /etc/exports i umieszczenie w nim wpisu:

```
/home/mpiuser/cloud *(rw,sync,no_root_squash,no_subtree_check)
```

Wyeksportowanie zmian:

```
$ exportfs -a
```

Na węzłach obliczeniowych:
Utworzenie folderu cloud:

```
$ mkdir cloud
```

Zamontowanie folderu współdzielonego do lokalnego systemu plików:

```
$ sudo mount -t nfs master:/home/mpiuser/cloud ~/cloud
```

Sprawdzenie, czy folder został poprawnie zamontowany:

```
$ df -h
Filesystem               Size  Used Avail Use% Mounted on
master:/home/mpiuser/cloud 49G 15G  32G  32% /home/mpiuser/cloud
```

Tworzenie bezhasłowego połączenia SSH Poniższe kroki należy wykonać na każdym węźle.
Generacja pary kluczy - publicznego i prywatnego:

```
$ ssh-keygen -t dsa
```

Przesłanie klucza publicznego z węzła głównego na obliczeniowe i na odwrót:

```
$ ssh-copy-id master (na wezlach obliczeniowych)
$ ssh-copy-id slave[i] (na wezle glownym)
$ ssh-copy-id localhost (dodatkowo na kazdej maszynie)
```

Wyłączenie konieczności podawania hasła przy logowaniu SSH:

```
$ eval `ssh-agent`
$ ssh-add ~/.ssh/id_dsa
```

Należy przetestować połączenie pomiędzy węzłem głównym i każdym z obliczeniowych za pomocą próby zalogowania:

```
$ ssh master (slave[i])
```

Uruchomienie pliku wykonywalnego Uruchomienie pliku wykonywalnego Factorer dokonuje się za pomocą komendy mpirun z określonymi parametrami. Przykładowe wywołanie:

```
$ mpirun -np 4 -hosts master,slave1 ./Factorer
```

Komenda przyjmuje następujące parametry:

- -np *liczba* - zbiorcza liczba wątków wykonywanych na klastrze. Przykład: System składa się z 3 jednakowych maszyn obliczeniowych i węzła głów-

nego. Każda stacja posiada 2 rdzenie procesora. W celu równomiernego zrównoważenia obciążenia procesór wartość parametru powinna wynosić 3*2 + 2 = 8 wątków.

- -hosts *hostname1,hostname2 ...* - nazwy węzłów na których ma zostać uruchomiony program oddzielone przecinkami (bez spacji). Lista musi zawierać węzeł główny (master) oraz może zawierać dowolną liczbę węzłów obliczeniowych (slave).

6.2 Uruchamianie strony

W celu uruchomienia strony potrzebny jest serwer wraz z zainstalowaną powłoką Python 3.4. Do tego należy zainstalować moduł Django w wersji 1.8.7. Oprócz powyższych należy też mieć serwer bazy danych, w przypadku tego projektu jest to serwer MySQL. Po spełnieniu tych wymagań można przejść do uruchamiania strony.

Pierwszym krokiem jest ustawienie w opcjach strony w pliku settings.py danych dostępowych do bazy danych.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'factorDB',
        'USER': 'projekt',
        'PASSWORD': 'projekt',
        'HOST': '156.17.235.48',
        'PORT': '3306',
    }
}
```

Następnym krokiem jest wypełnienie bazy danych odpowiednimi strukturami stworzonymi wcześniej w Python używając rozwiązania typu ORM. Służy do tego skrypt manage.py znajdujacy się w folderze z plikami źródłowymi strony. Należy zrobić migrację bazy danych dla naszej aplikacji(nazwa to FactorerMain). Osiąga sie to przez wywołanie **python manage.py makemigrations FactorerMain**. Następnie koniecznym jest wykonać utworzone migrację poprzez **python manage.py migrate**. Dzięki tym dwóm komendom baza danych zostanie wypełniona odpowiednio zdefiniowanymi strukturami. Ostatnim krokiem jest uruchomienie serwera, osiąga się to przez użycie kolejnej komendy skryput manage.py - **python manage.py runserver**.

```
$ python manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).

June 08, 2016 - 16:24:25
```

```
Django version 1.8.7, using settings 'Factorer.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Jak widać na powyższym wyciągu z terminala serwer został uruchominy.

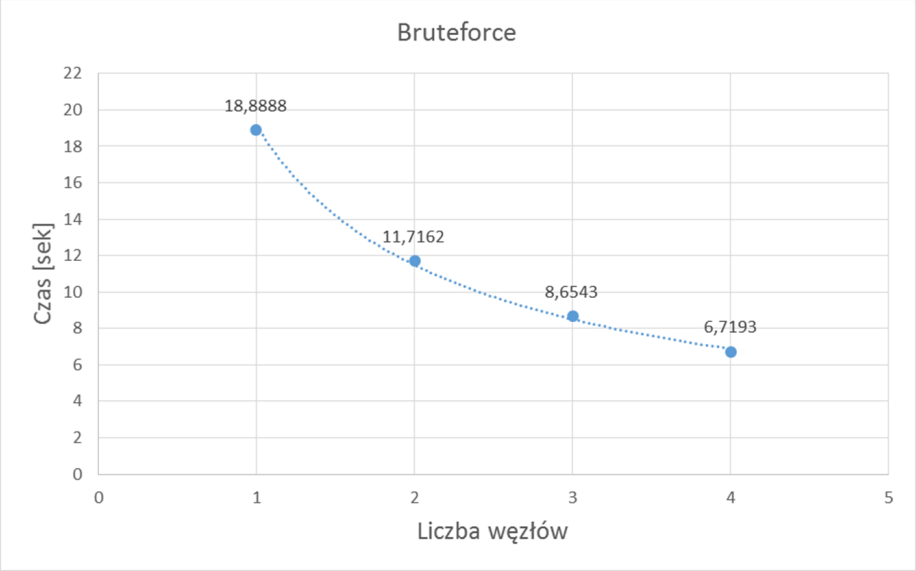
7 Przykładowe wyniki badań efektywności programu równoległego

Wydajność została przetestowana dla następujących parametrów:

- faktoryzowana liczba - 1234567891011121314
- ilość powtórzeń pomiarów - 20, wyniki uśredniono
- ilość rdzeni na węzeł - 2
- ilości węzłów obliczeniowych - 1, 2, 3, 4

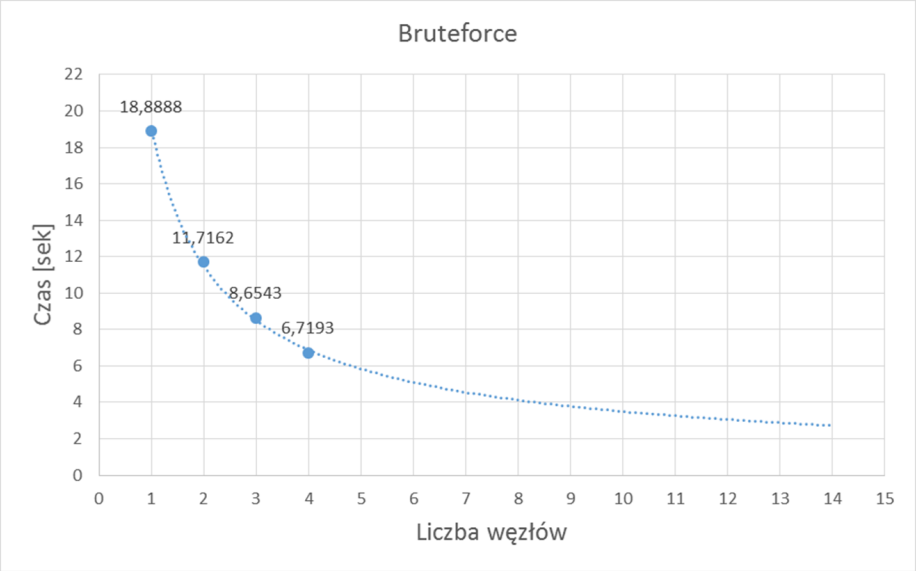
7.1 Testy wydajności dla algorytmu Bruteforce

Wyniki w formie wykresu dla algorytmu Bruteforce:



Rysunek 5: Bruteforce

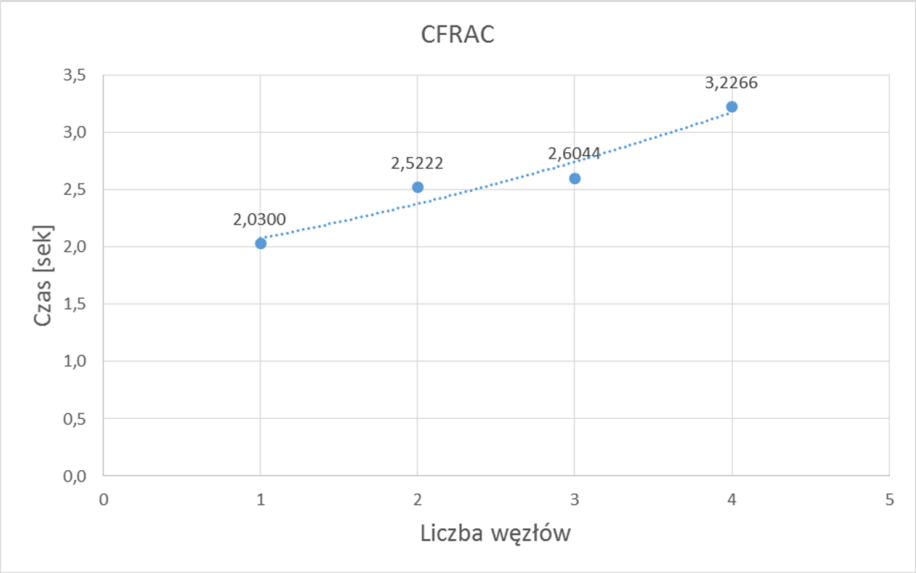
Algorytm faktoryzacji bruteforce posiada niekorzystną złożoność obliczeniową, przez co czas jego wykonania jest długi. Na wykresie widać, że dołączanie kolejnych maszyn powoduje skrócenie czasu obliczeń. Zależność czasu od ilości węzłów nie jest jednak liniowa. Wykorzystując narzędzia arkusza kalkulacyjnego można zasymulować prognozowaną linię trendu dla większej ilości maszyn. Widzimy zatem, że wzrost wydajności następuje tylko do pewnej ilości dołączanych węzłów obliczeniowych. Powyżej pewnej wartości zwiększanie ilości węzłów nie będzie powodowało przyspieszenia obliczeń.



Rysunek 6: Bruteforce - prognozowane wyniki dla 14 maszyn

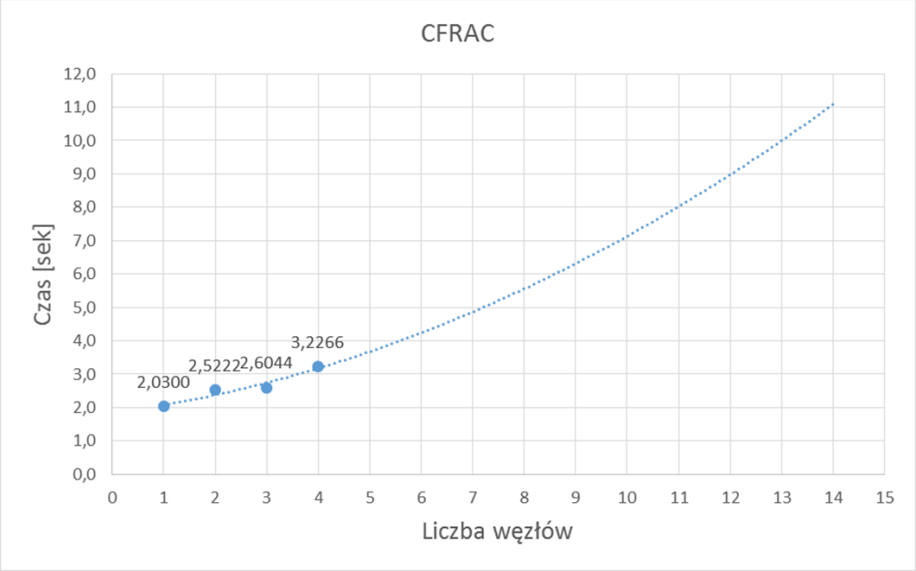
7.2 Testy wydajności dla algorytmu CFRAC

Wyniki w formie wykresu dla algorytmu CFRAC:



Rysunek 7: CFRAC

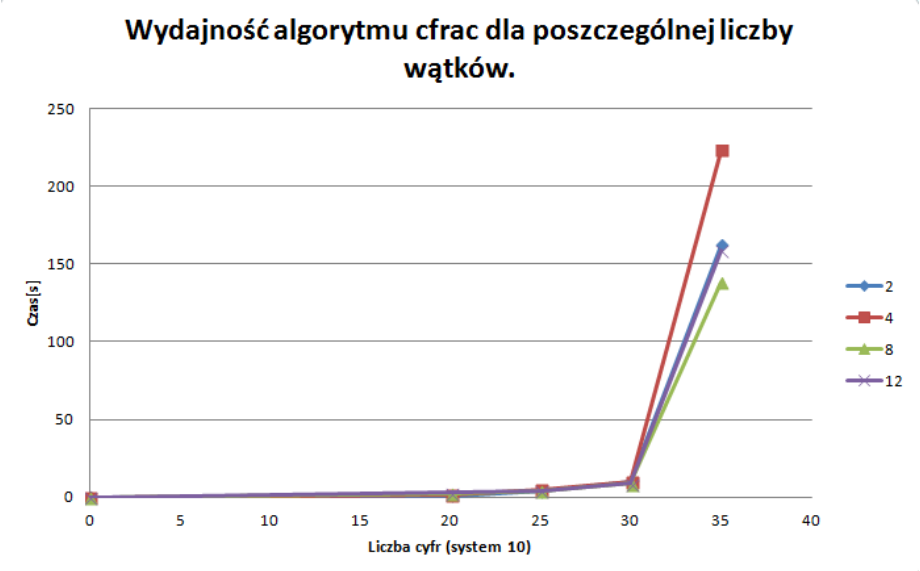
Algorytm CFRAC implementuje o wiele wydajniejszą metodę factoryzacji. Wyniki uzyskane na pojedynczym węźle obliczeniowym są około 9 razy mniejsze od wyników bruteforce dla tej samej liczby. Widać tutaj również ciekawe zjawisko. Przy dodawaniu kolejnych węzłów obliczeniowych czas wykonania zadania nie maleje, a wręcz nieznacznie wzrasta. Wydłużenie czasu jest wynikiem konieczności zużycia zasobów na rozdzielenie oraz rozesłanie podzadań do węzłów obliczeniowych, a następnie pobranie wyników cząstkowych i skompletowanie ostatecznego. Czas jest również uzależniony od prędkości przesyłu danych pomiędzy węzłami. W testach wykorzystywane było łącze FastEthernet o maksymalnej przepustowości 100 Mbit/s. Możliwe, że przy większej przepustowości dołączanie kolejnych węzłów nie powodowałoby takiego opóźnienia. W tym wypadku również można zasymulować prognozę dla większej ilości maszyn, jednakże należy ją traktować z większym dystansem niż tą dla bruteforce. Wyniki mają większe wahania, przez co trudniej było dobrać odpowiednią funkcję trendu.



Rysunek 8: CFRAC - prognozowane wyniki dla 14 maszyn

7.3 Algorytm CFRAC - badania dodatkowe

Badania przeprowadzone zostały na pojedyńczym komputerze wykorzystując symulację MPI dla danej ilości wątków.



Rysunek 9: Wydajność algorytmu CFRAC dla poszczególnej liczby wątków

		Liczba wątków				
liczba bitów	Liczba cyfr(sys. 10)	2	4	8	12	
64-66	20	1,1	1,5	2,5	3,1	Czas[s]
79-82	25	3,9	4,7	4,2	4,2	
96-99	30	10,2	10,1	8,9	8,8	
112 - 116	35	162,2	223,2	138,2	158,7	

Rysunek 10: Dane pomiarów wydajność algorytmu CFRAC

8 Wnioski

Literatura

[1] *Dokumentacja Django*. https://docs.djangoproject.com/en/1.8/, 2016

[2] *Dokumentacja MPICH2*. http://www.mpich.org/documentation/guides/, 2016.

[3] *Cunningham Project*. http://homes.cerias.purdue.edu/~ssw/cun/

[4] *RSA Factoring Challenge*. http://pl.wikipedia.org/wiki/RSA_Factoring_Challenge

[5] *CFRAC opis*. https://en.wikipedia.org/wiki/Continued_fraction_factorization

[6] *CFRAC zastosowanie*. Mateusz Niezabitowski http://ki.agh.edu.pl/sites/default/files/usefiles/172/theses/mateusz.niezabitowski.algorytmy.faktoryzacji.w.zastosowaniach.kryptograficznych.v1.0-final.pdf

[7] *CFRAC implementacja*. https://math.dartmouth.edu/~carlp/PDF/implementation.pdf

[8] *Sublime Text Editor*. https://www.sublimetext.com/

[9] . *Anaconda Python IDE* http://damnwidget.github.io/anaconda/

[10] *ORM - object-relational mapping*. https://pl.wikipedia.org/wiki/Mapowanie_obiektowo-relacyjne

[11] *Django Models documentation*. https://docs.djangoproject.com/en/1.9/topics/db/models/