

Aplikacje internetowe i rozproszone - projekt

Testowanie czy zadana liczba naturalna jest pierwsza.

Prowadzący: Dr inż. Henryk Maciejewski

Skład grupy:

Lider: Bartosz Miącz 200784

Tomasz Stankiewicz 180362

Stanisław Nowak 200761

Łukasz Sobolak 200666

Kamil Matusiak 200709

Paweł Brodziak 200749

1. Temat i cel projektu	3
2. Opis systemu.....	3
3. Architektura systemu	4
3.1. Architektura backendu	4
3.2. Architektura frontendu.....	6
4. Dokumentacja powykonawcza	7
4.1. Klaser	7
4.1.1. Konfiguracja wirtualnych maszyn	7
4.1.2. Ustawienia interfejsów sieciowych.....	7
4.1.3. SSH	9
4.1.4. NFS	10
4.1.5. MPICH	10
4.2. Testy	12
4.2.1. Sposób wykonania	12
4.2.2. Wyniki	13
4.2.3. Wnioski.....	19

1. Temat i cel projektu

Temat:

Testowanie czy zadana liczba naturalna jest pierwsza.

Cel:

Celem projektu jest stworzenie aplikacji, udostępniającej poprzez przeglądarkę WWW użytkownikowi klastra obliczeniowego. Zadaniem do wykonania jest przeprowadzenie testu czy zadana liczba naturalna jest pierwsza, za pomocą jednego z wielu znanych algorytmów wykonywanego na klastrze stacji roboczych w środowisku rozproszonym.

2. Opis systemu

System ma za zadanie przeprowadzeniu testu Millera-Rabina na liczbie podanej przez użytkownika. Składa się on z dwóch modułów

1. Frontend
2. Backend

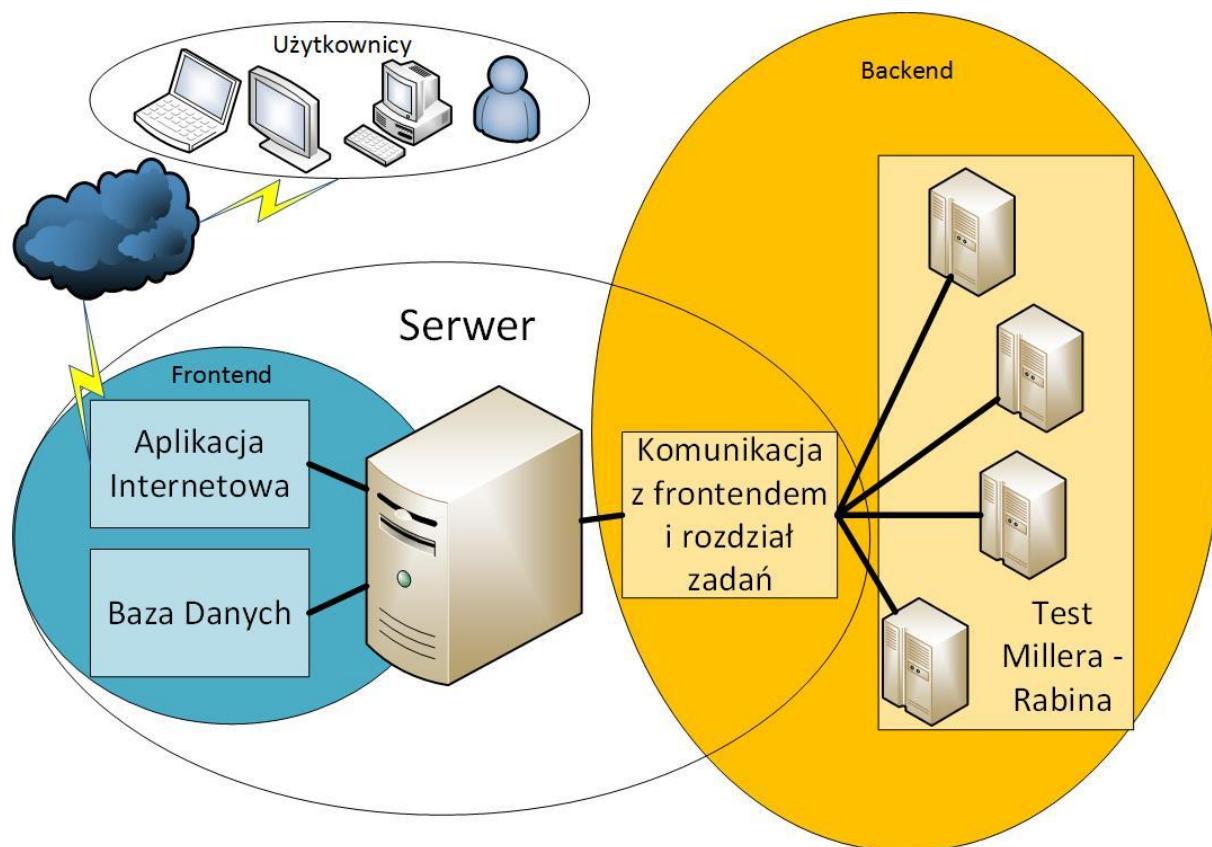
Frontend tworzą aplikacja internetowa oraz baza danych. Aplikacja umożliwia użytkownikowi wykorzystanie klastra obliczeniowego w celu przeprowadzeniu testu pierwszości liczby naturalnej. Aplikacja obejmuje:

- rejestrację w systemie oraz logowanie.
- zarządzanie zadaniami
- planowanie i kolejkovanie zadań
- komunikację z bazą danych

Baza danych przechowuje wyniki zadań oraz ich harmonogram i dostarcza tych informacji aplikacji.

Backend tworzą dwie aplikacje zawarte w jednym pliku wykonywalnym. Pierwsza odpowiada za rozdzielanie zadań na pozostałe maszyny oraz komunikację z frontendem. Druga wykonuje test Millera-Rabina i zwraca wyniki.

3. Architektura systemu



3.1. Architektura backendu

Backend aplikacji do sprawdzania pierwszości liczb zrealizowany został z wykorzystaniem technologii MPICH która jest realizacją standardu MPI. Taki podejście pozwala na rozproszenie obliczeń w chmurze obliczeniowej. Do testowania pierwszości zadanych liczb zaimplementowany został algorytm Millera-Rabina.

Pseudokod dla powyższego algorytmu przedstawia się następująco:

Wejście

- p – liczba badana na pierwszość, $p \in \mathbb{N}$, $p > 2$, p jest nieparzyste
 n – ilość powtórzeń testu Millera-Rabina, $n \in \mathbb{N}$

Wyjście:

TAK, jeśli p jest pierwsze lub silnie pseudopierwsze z prawdopodobieństwem $(\frac{1}{4})^n$.
NIE, jeśli p jest liczbą złożoną

Elementy pomocnicze:

- s – wykładnik potęgi 2 w dzielniku $p - 1$. $s \in \mathbb{N}$

d	–	mnożnik potęgi 2 w dzielniku $p - 1$. $d \in \mathbb{N}$
i	–	zlicza wykonane testy Millera-Rabina, $i \in \mathbb{N}$
a	–	baza. $a \in \mathbb{N}$, $a \in \langle 2, p-2 \rangle$
x	–	wyraz ciągu Millera-Rabina, $x \in \mathbb{N}$
j	–	zlicza wyrazy ciągu Millera-Rabina, $j \in \mathbb{N}$

Lista kroków:

K01:	$d \leftarrow p - 1$; obliczamy s i d
K02:	$s \leftarrow 0$	
K03:	Dopóki $d \bmod 2 = 0$, wykonuj K04...K05	; usuwamy z $p - 1$ dzielniki 2 zliczając je w s
K04:	$s \leftarrow s + 1$	
K05:	$d \leftarrow d \text{ div } 2$	
K06:	Dla $i = 1, 2, \dots, n$, wykonuj K07...K15	; wykonujemy n testów Millera-Rabina
K07:	$a \leftarrow \text{Losuj}(2, p-2)$; losujemy bazę a
K08:	$x \leftarrow a^d \bmod p$; wyliczamy pierwszy wyraz ciągu Millera-Rabina
K09:	Jeśli $(x = 1) \vee (x = p - 1)$, to następny obieg K06	; jeśli x nie spełnia warunku, wybieramy inne a
K10:	$j \leftarrow 1$	
K11:	Dopóki $(j < s) \wedge (x \neq p - 1)$, wykonuj K12...K14	; rozpoczynamy generację kolejnych wyrazów ciągu Millera-Rabina
K12:	$x \leftarrow x^2 \bmod p$; obliczamy kolejny wyraz
K13:	Jeśli $x = 1$, to idź do K17	; tylko ostatni wyraz ciągu Millera-Rabina może mieć wartość 1!
K14:	$j \leftarrow j + 1$	
K15:	Jeśli $x \neq p - 1$, to idź do K17	; przedostatni wyraz ciągu Millera-Rabina musi być równy $p - 1$
K16:	Pisz "TAK" i zakończ	; pętla wykonała n testów i zakończyła się naturalnie.
K17:	Pisz "NIE" i zakończ	; liczba p nie przeszła testów Millera-Rabina

Kroki od K01 do K05 wykonywane są sekwencyjnie, natomiast pętla z kroku K06 może zostać w pełni zrównoleglona ponieważ każda jej iteracja jest niezależna.

Program został wykonany w klasycznej architekturze typu master-slave. Master wykonuje część sekwencyjną oraz oblicza losowy wektor niepowtarzających się baz „a”. Następnie master rozsyła dane wyliczone w części sekwencyjnej (można to zrobić raz dla każdego slave’a ponieważ dane te nie ulegają zmianie) oraz kolejne bazy obliczeń. Limitem pętli z kroku K06 jest minimum z ilości dostępnych baz oraz zadanej ilości obliczeń.

Po rozesłaniu pierwszej partii zadań master oczekuje na sygnały zakończenia przez danego slave’a. Slave wykonuje całą jedną iterację pętli z kroku K06. Jeżeli zwrócił wartość mówiącą że liczba jest pierwsza, dostaje on kolejne zadanie z puli. Po wyczerpaniu wszystkich zadań z puli, master oczekuje na zakończenie wszystkich sławów obliczając prawdopodobieństwo tego że liczba jest pierwsza. Jeżeli chociaż jeden slave zwróci wartość mówiącą że liczba nie jest pierwsza

rozsyłanie dalszych zadań jest zatrzymywane a master oczekuje na zakończenie pozostałych sławów. Następnie na ekran drukowany jest wynik.

Jak widać z powyższego opisu instancja mastera podczas głównych obliczeń, praktycznie nic nie robi dzięki czemu może ona być odpalona na maszynie na której jest również serwer frontendu ponieważ praktycznie nie będzie obciążać tej maszyny.

Ponieważ potęgowanie z kroku K08 może odbyć się na bardzo dużych liczbach, zastosowano algorytm szybkiego potęgowania oraz bibliotekę pozwalającą przeprowadzać operacje arytmetyczne na bardzo dużych liczbach całkowitych.

Algorytm szybkiego potęgowania a^n został zrealizowany według poniższego schematu:

```
wynik = 1
potęga = n
baza = a
```

Dopóki potęga > 0

Jeżeli potęga jest nieparzysta:

```
wynik = wynik*baza
baza = baza*baza
potęga = potęga/2
```

3.2. Architektura frontendu

Frontend zawiera system autoryzacji użytkowników oparty na wbudowanych mechanizmach Django w wersji 1.8. Umożliwia on rejestrację oraz bezpieczną pracę wielu użytkowników. Użytkownik systemu dodaje zadania do zrealizowania na klastrze obliczeniowym. Dla każdego z nich wprowadza dane wejściowe, które są przetwarzane. Zadanie fizycznie jest rekordem bazy danych o następujących atrybutach:

- ID
- Number - badana liczba
- result - tutaj będzie wynik true albo false (dopuszczalna wartość null)
- probability - prawdopodobieństwo (dopuszczalna wartość null)
- create_date - data utworzenia zadania (dopuszczalna wartość null)
- start_date - data odpalenia zadania przez backend (dopuszczalna wartość null)
- finish_date - data ukończenia zadania (dopuszczalna wartość null)
- user - klucz obcy do użytkownika (dopuszczalna wartość null)
- status - są cztery {In_queue, In_progress, Finished, Aborted}

Komunikacja pomiędzy frontend'em a backend'em odbywa się poprzez bazę danych SQLite w wersji 3. Aplikacja zawiera skrypt napisany w języku Python w wersji 3, który co 60 sekund wysyła zapytanie do bazy o jeden wpis, którego status to In_queue, który został najdawniej dodany do bazy. Następnie przekazuje zadanie do backendu, czeka na wynik i aktualizuje wpis w bazie. Przesłanie rekordu do backendu polega na odczytaniu wybranych atrybutów rekordu zadania z

task-managera i wywołanie komendy na masterze. Task-manager aktualizuje te dane (null-available) w trakcie oraz po ukończeniu wykonywania testu.

4. Dokumentacja powykonawcza

Poniższy rozdział dokumentacji będzie opisywała sposób w jaki można przygotować klastry do pracy. Pokazuje co jest potrzebne do ich działania oraz przebiega proces ich konfiguracji. W dalszej części dokumentu przedstawione zostaną testy wydajnościowe aplikacji zajmującej się sprawdzaniem czy liczba jest pierwsza.

4.1. Klaser

W celu wykonania testów należało postawić klaster dzięki, któremu maszyny będą mogły korzystać z MPI. Ogólny proces tworzenia klastra składał się z:

- (1) postawienia wirtualnych maszyn,
- (2) stworzeniu interfejsów sieciowych na których będzie miało miejsce połączenie,
- (3) stworzenia bez hasłowego połączenia SSH,
- (4) stworzenia sieciowego systemu plików NFS,
- (5) wgranie do niego części backend'owej aplikacji którą opisuje ten dokument.

4.1.1. Konfiguracja wirtualnych maszyn

Do postawienia maszyn wirtualnych użyto programu **VirtualBox 4.3.28**. Jako bazę wybrano system **Ubuntu w wersji 15.04**. Do obsługi węzła master wykorzystano jego pełną, natomiast dla slave'ów ograniczono się do wydania MinimalCD, pozbawionej między innymi GUI. Dzięki temu zaoszczędzono dużo miejsca, gdyż taka wersja waży jedynie kilkadziesiąt MB. Każdej maszynie przydzielono 1 rdzeń procesora o prędkości 2.5GHz. Instalacja wszystkich systemów przebiegała na ustawieniach domyślnych z wyjątkiem wybrania opcji instalowania OpenSSH na slave'ach.

4.1.2. Ustawienia interfejsów sieciowych

W celu możliwości komunikacji między maszynami należy skonfigurować interfejs na którym będzie dochodziło do łączenia. W celu sprawdzenia obecnych interfejsów, można w terminalu użyć polecenia:

Ifconfig

```

mpiu@master-VirtualBox:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr 08:00:27:c4:63:7d
          inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fec4:637d/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:29 errors:0 dropped:0 overruns:0 frame:0
          TX packets:85 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:3305 (3.3 KB)  TX bytes:11216 (11.2 KB)

```

Rysunek 1: Wynik polecenia ifconfig

Aby zmodyfikować interfejsy należy otworzyć plik `/etc/network/interfaces` w dowolnym edytorze np. `pico`:

`sudo pico /etc/network/interfaces`

W celu dodania interfejsu trzeba dopisać treść przedstawioną na rysunku 2.

```

auto eth1
iface eth1 inet static

address 192.168.100.100
netmask 255.255.255.0
network 192.168.100.0
broadcast 192.168.100.255
gateway 10.0.2.15

```

Rysunek 2: Dodanie interfejsu

W tym momencie warto przedstawić przykładową adresację maszyn która została użyta podczas testów.

Tabela 1: Adresacja maszyn

Node	Adres
master	192.168.100.100
slave 0	192.168.100.101
slave 1	192.168.100.102
slave 2	192.168.100.103
slave 3	192.168.100.104
slave 4	192.168.100.105
slave 5	192.168.100.106
slave 6	192.168.100.107

Komunikację między maszynami można sprawdzić za pomocą wykorzystania w terminalu polecenia:

`ping adres_ip`


```
mpiu@master-VirtualBox:~$ ping 192.168.100.101
PING 192.168.100.101 (192.168.100.101) 56(84) bytes of data.
64 bytes from 192.168.100.101: icmp_seq=1 ttl=64 time=1.14 ms
64 bytes from 192.168.100.101: icmp_seq=2 ttl=64 time=0.354 ms
```

Rysunek 3: Sprawdzanie połączenia pomiędzy master'em i slave0 za pomocą komendy ping.

Ewentualne błędy w połączeniu mogą być spowodowane źle skonfigurowanym firewall'em.

4.1.3. SSH

Przed instalacją SSH warto stworzyć na wszystkich nodach user'a o takiej samej nazwie oraz posiadającego wszystkie prawa. W konsoli należy wpisać:

adduser nazwa_usera

Po wybraniu domyślnych opcji wpisujemy:

visudo

Pod linijką root ALL=(ALL:ALL) ALL dopisujemy:

nazwa_usera ALL=(ALL:ALL) ALL

Teraz należy przelogować się na nowo stworzonego użytkownika.

Kolejnym krokiem jest instalacja serwera SSH na nodzie master. Można tego dokonać za pomocą polecenia

sudo apt-get install openssh-server

Do stworzenia bez hasłowego połączenia będzie potrzebny folder .SSH. Najłatwiej stworzyć go poprzez jednorazowe połączenie się master'em do każdego slave i odwrotnie. Dzięki temu na każdej maszynie otrzymamy folder .SSH zawierający plik *known_hosts*. Wchodzimy do folderu .ssh za pomocą *cd*, a następnie na master nodzie tworzymy klucz publiczny za pomocą polecenia

ssh-keygen -t dsa

Po wybraniu opcji domyślnych w naszym katalogu stworzył się plik *id_dsa.pub*, który należy wysłać do każdego slave'a na przykład za pomocą komendy *scp*:

scp id_dsa.pub nazwa_usera@192.168.100.10x:~/.ssh/id_dsa.pub

Na slave nodach w katalogu .ssh pojawił się plik *id_dsa.pub*, który otwieramy:

cat id_dsa.pub >> authorized_keys

Połączenie do innej maszyny wykonuje się za pomocą polecenia

ssh 192.168.100.10x (*x* – w zależności od numeru slave'u, tabela 1)

4.1.4. NFS

Na master nodzie instalujemy serwer NFS za pomocą komendy:

```
sudo apt-get install nfs-kernel-server portmap
```

Natomiast na slave'ach wystarczy komenda:

```
sudo apt-get install nfs-common portmap
```

Na master nodzie tworzymy folder, który udostępnimy oraz zmieniamy jego prawa na nienależące do nikogo i do żadnej grupy.

```
sudo mkdir /mirror
```

```
sudo chown nobody:nogroup /mirror
```

W pliku /etc/exports dopisujemy na samym dole „/mirror *(rw,sync)”. Włączamy serwer:

```
sudo service nfs-kernel-server start
```

Folder jest gotowy do udostępnienia. Na slave'ach analogicznie tworzymy folder /mirror i montujemy go za pomocą polecenia:

```
sudo mount 192.168.100.100:/mirror /mirror
```

Poprawność można sprawić za pomocą polecenia `df -h`

```
mpiu@slave0:~$ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1        6.8G  1.3G  5.2G  19% /
none            4.0K    0   4.0K   0% /sys/fs/cgroup
udev            485M  4.0K  485M   1% /dev
tmpfs           100M  412K   99M   1% /run
none            5.0M    0   5.0M   0% /run/lock
none            497M    0  497M   0% /run/shm
none            100M    0  100M   0% /run/user
192.168.100.100:/mirror 14G  4.8G  8.2G  37% /mirror
```

Rysunek 4: Sprawdzenie zamontowania folderu NFS.

Teraz każdy plik umieszczony w folderze /mirror jest widoczny na wszystkich nodach. W nim umieszczamy aplikację backend'ową.

4.1.5. MPICH

Aby zainstalować MPICH należy ściągnąć jego wersję source, rozpakować, skonfigurować oraz zbuildować. W master nodzie wchodzimy do folderu /mirror i wpisujemy polecenie:

```
sudo wget http://www.mpich.org/static/downloads/3.0.4/mpich-3.0.4.tar.gz
```

```
sudo tar xvf mpich-3.0.4.tar.gz
```

```
cd mpich-3.0.4
```

```
./configure --prefix=/mirror/mpich2 --disable-f77 --disable-fc
```

sudo make

sudo make install

Następnie należy ustawić ścieżki:

export PATH=/mirror/mpich2/bin:\$PATH

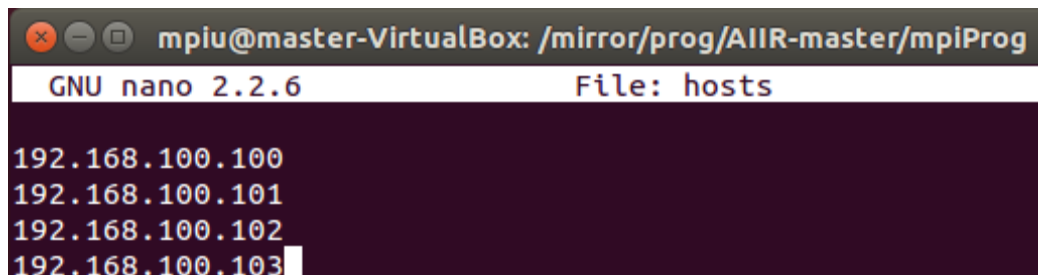
export LD_LIBRARY_PATH="/mirror/mpich2/lib:\$LD_LIBRARY_PATH"

Ostatnim krokiem jest dodanie ścieżki do /etc/environent

```
PATH="/mirror/mpich2/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
```

Rysunek 5: Dodanie ścieżki

W miejscu pliku wykonywalnego backend'u trzeba stworzyć plik zawierający listę slave'ów.



```
mpi@master-VirtualBox: /mirror/prog/AllR-master/mpiProg
GNU nano 2.2.6 File: hosts
192.168.100.100
192.168.100.101
192.168.100.102
192.168.100.103
```

Rysunek 6: Plik z hostami

Wykonanie pliku odbywa się poprzez komendę:

mpiexec -n A -f B ./Main C D

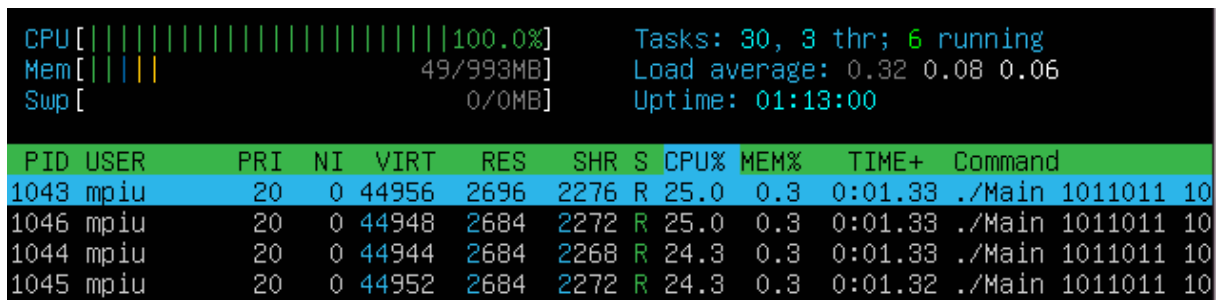
A – liczba procesów

B – nazwa pliku z hostami

C – kandydat na liczbę pierwszą

D – liczba testów

Poprawne działanie MPI można zaobserwować poprzez program htop uruchomiony na slave'ach, który podczas wykonywania zadania zleconego na master nodzie pokazuje użycie procesora oraz procesy jak na rysunku 7.



Rysunek 7: Wykaz z htop na slave0.

4.2. Testy

4.2.1. Sposób wykonania

Pomiarów dokonano za pomocą funkcji *clock()*, z biblioteki *time.h*. Na listingu 1 pokazano koncept według którego robiono pomiary.

```
clock_t start,end;

start = clock();

    //sprawdzanie liczby pierwszej

end = clock();

std::cout << (double)(end-start)/CLOCKS_PER_SEC << endl;
```

Listing 1: Pomiar czasu

Tak dokonane pomiary dają dokładność rzędu jednej mikrosekundy. Testy przeprowadzono dla 10 liczb. Połowa z nich była pierwsza. Liczby dobrano tak, aby sprawdzić wyniki dla liczb rzędu setek, tysięcy, dziesiątek tysięcy itp. Wybrane liczby zostały przedstawione w tabeli 2.

Tabela 2: Testowane liczby

Liczba	Pierwsza
11	tak
21	nie
569	tak
189	nie
2557	tak
5859	nie
41843	tak
52731	nie
211543	tak
474579	nie

4.2.2. Wyniki

Wyniki testów dla klastra składającego się z jednego mastera i jednego slave'a przedstawiają tabele 3-6.

Tabela 3: 1 slave, 2 testy

1 master 1 slave: 2 testy		
Liczba	Czas	Pierwsza
11	0,009	tak
21	0,054	nie
569	0,34	tak
189	1,085	nie
2557	0,42	tak
5859	1,105	nie
41843	121	tak
52731	74,25	nie
211543	1920	tak
474579	1238	nie

Tabela 4: 1 slave, 4 testy

1 master 1 slave: 4 testy		
Liczba	Czas	Pierwsza
11	0,0081	tak
21	0,054	nie
569	0,306	tak
189	0,9765	nie
2557	0,378	tak
5859	1,105	nie
41843	108,9	tak
52731	81,675	nie
211543	2112	tak
474579	1238	nie

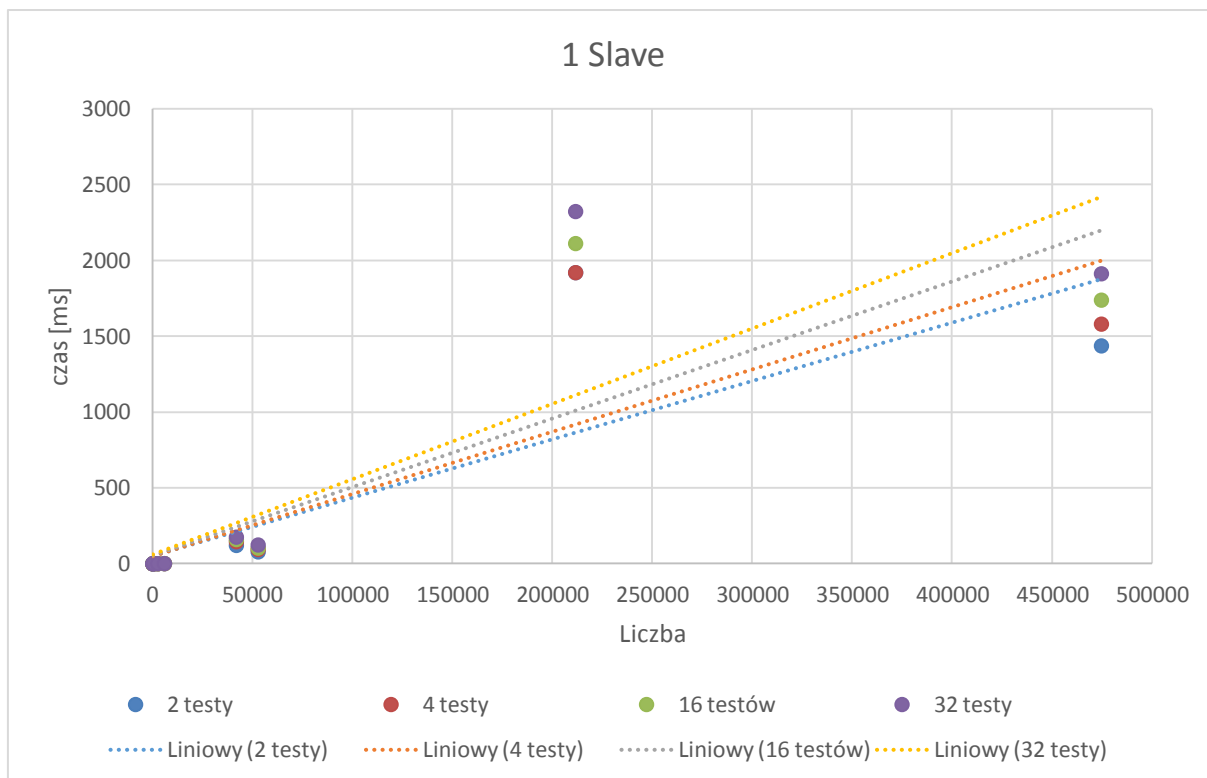
Tabela 5: 1 slave, 16 testów

1 master 1 slave: 16 testów		
Liczba	Czas	Pierwsza
11	0,01134	tak
21	0,081	nie
569	0,459	tak
189	1,1718	nie
2557	0,4536	tak
5859	1,4365	nie
41843	141,57	tak
52731	98,01	nie
211543	2534,4	tak
474579	1609,4	nie

Tabela 6: 1 slave, 32 testy

1 master 1 slave: 32 testów		
Liczba	Czas	Pierwsza
11	0,018144	tak
21	0,1215	nie
569	0,6426	tak
189	2,3436	nie
2557	0,95256	tak
5859	2,44205	nie
41843	311,454	tak
52731	186,219	nie
211543	4815,36	tak
474579	2575,04	nie

Powyższe wyniki zestawiono w wykres dodając do niego regresję liniową. Można zauważyć, że dla dwu-nodowego klastra zwiększenie ziarna podziału zwiększa czas wykonywania algorytmu jednak różnice nie przekraczają 30%.



Wykres 1: 1 Master 1 Slave dla różnej ilości testów

Wyniki testów dla 3 slave'ów przedstawiają tabele 7-10.

Tabela 7: 3 slave'y, 2 testy

1 master 3 slave: 2 testy		
Liczba	Czas	Pierwsza
11	0,009	tak
21	0,0486	nie
569	0,34	tak
189	0,9765	nie
2557	0,42	tak
5859	1,105	nie
41843	121	tak
52731	74,25	nie
211543	1728	tak
474579	1286	nie

Tabela 8: 3 slave'y, 4 testy

1 master 3 slave: 4 testy		
Liczba	Czas	Pierwsza
11	0,0063	tak
21	0,04158	nie
569	0,2244	tak
189	0,68355	nie
2557	0,294	tak
5859	0,69615	nie
41843	79,86	tak
52731	57,1725	nie
211543	1036,8	tak
474579	694,44	nie

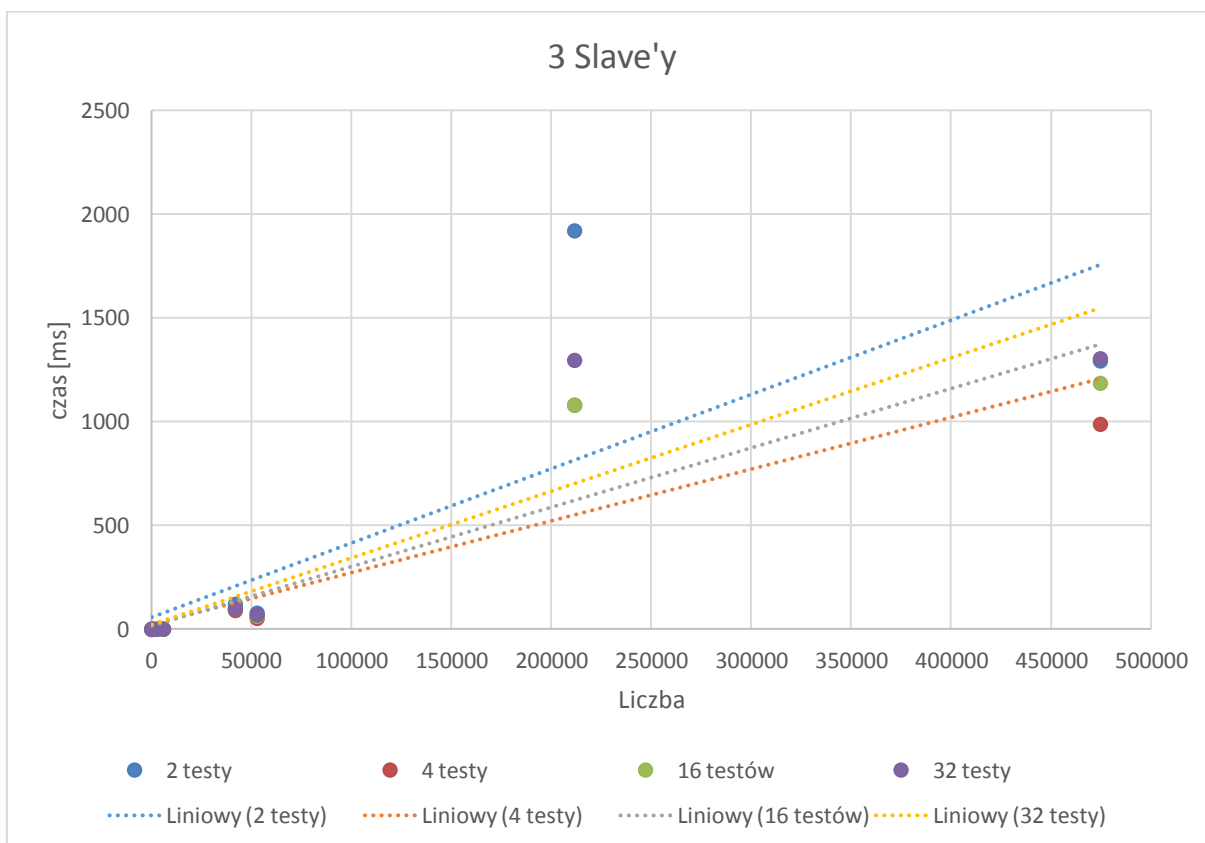
Tabela 9: 3 slave'y, 16 testów

1 master 3 slave: 16 testów		
Liczba	Czas	Pierwsza
11	0,006804	tak
21	0,04536	nie
569	0,1836	tak
189	0,751905	nie
2557	0,27216	tak
5859	0,663	nie
41843	91,476	tak
52731	58,806	nie
211543	1626,24	tak
474579	953,26	nie

Tabela 10: 3 slave'y, 32 testy

1 master 3 slave: 32 testów		
Liczba	Czas	Pierwsza
11	0,0088452	tak
21	0,06804	nie
569	0,2754	tak
189	1,203048	nie
2557	0,435456	tak
5859	0,8619	nie
41843	146,3616	tak
52731	94,0896	nie
211543	1788,864	tak
474579	1048,586	nie

Na wykresie drugim widać, że przy zwiększeniu liczby nodów do czterech najbardziej optymalny podział to 4 oraz 16, natomiast najgorszy 2. Przy odpowiednim podziale zyskujemy około 50% lepszy wynik w stosunku do najgorszego.

**Wykres 2:** 1 Master 3 Slave dla różnej ilości testów

Wyniki testów dla 7 slave'ów przedstawiają tabele 11-14.

Tabela 11: 7 slave'y, 2 testy

1 master 7 slave: 2 testy		
Liczba	Czas	Pierwsza
11	0,0081	tak
21	0,054	nie
569	0,306	tak
189	1,085	nie
2557	0,42	tak
5859	1,105	nie
41843	98,01	tak
52731	66,825	nie
211543	1920	tak
474579	1114,2	nie

Tabela 12: 7 slave'y, 4 testy

1 master 7 slave: 4 testy		
Liczba	Czas	Pierwsza
11	0,005103	tak
21	0,04158	nie
569	0,1836	tak
189	0,68355	nie
2557	0,20412	tak
5859	0,5967	nie
41843	83,853	tak
52731	49,005	nie
211543	1478,4	tak
474579	953,26	nie

Tabela 13: 7 slave'y, 16 testów

1 master 7 slave: 16 testów		
Liczba	Czas	Pierwsza
11	0,0040824	tak
21	0,031752	nie
569	0,12852	tak
189	0,451143	nie
2557	0,13608	tak
5859	0,3315	nie
41843	45,738	tak
52731	35,2836	nie
211543	1138,368	tak
474579	476,63	nie

Tabela 14: 7 slave'y, 32 testy

1 master 7 slave: 32 testów		
Liczba	Czas	Pierwsza
11	0,0057154	tak
21	0,0381024	nie
569	0,179928	tak
189	0,5864859	nie
2557	0,149688	tak
5859	0,4641	nie
41843	59,4594	tak
52731	49,39704	nie
211543	1366,0416	tak
474579	524,293	nie

Na wykres 3 widać, że można zaobserwować wyraźny podział na dwa poziomy czasowe. Przy podziale na 2 oraz 4 testy czas jest około 40% gorszy w stosunku do podziału na 16 i 32.



Wykres 3: 1 Master 7 Slave'ów dla różnej ilości testów

Dokonano również pomiarów dla stałej ilości testów, a różnej ilości slave'ów. Wyniki przedstawione zostały na wykresach 4-7.

Dla dwóch testów ilość nodów nie ma większego znaczenia i różnice w pomiarach można uznać za błąd pomiarowy.



Wykres 4: 2 testy dla różnej ilości nodów

Przy 4 testach widać wyraźną poprawę dla klastrów 4 i 8 nodowych w porównaniu do 2 nodowego. Poprawa jest prawie dwukrotna.



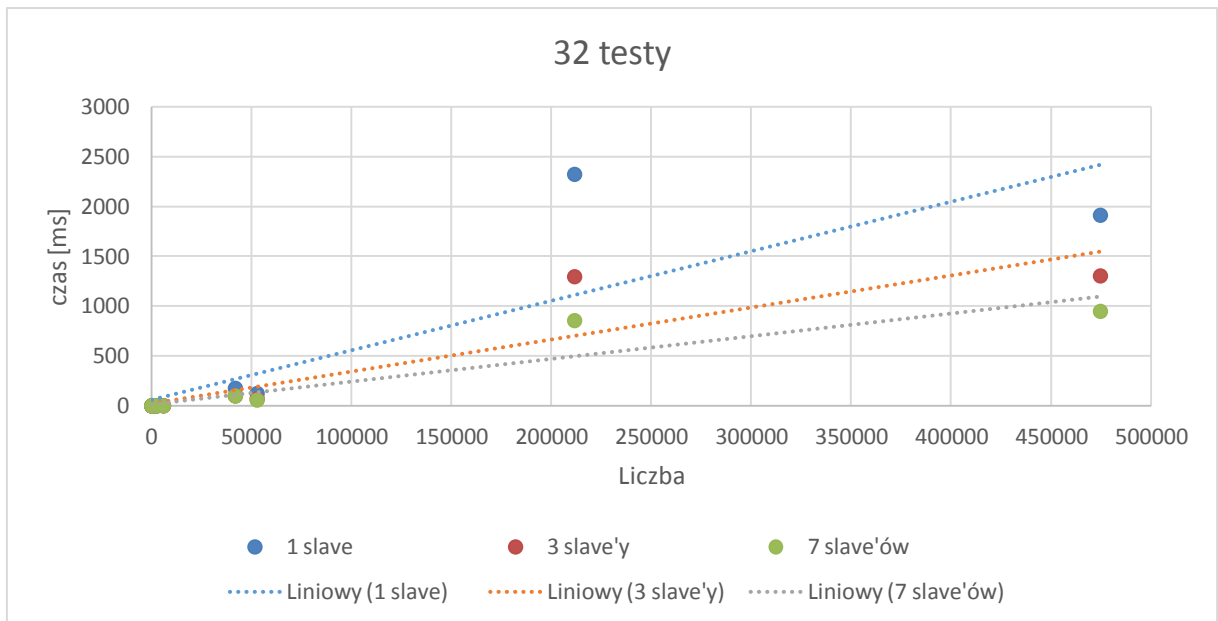
Wykres 5: 4 testy dla różnej ilości nodów

Przy 16 testach wyraźnie widać trzy poziomy tworzone przez klastery 2, 4 i 8 nodowy. Każdy kolejny poziom powoduje polepszenie czasu o około 40%.



Wykres 6: 16 testów dla różnej ilości nodów

Wykres 7 jest analogiczny do wykresu 6, jednak różnice na poszczególnych poziomach są jeszcze bardziej widoczne, wynoszą około 45%.



Wykres 7: 32 testy dla różnej ilości nodów

4.2.3. Wnioski

Po wykonaniu testów można zauważyć następujące zależności. Wzrost ilości nodów nigdy nie pogarsza czasu wykonywania się programu, natomiast poprawia go przy mniejszym ziarnie podziału (wykres 2 i 3). Dzieje się tak, bo przy małej ilości testów kilka nodów może być bezczynnych (sytuacja na wykresie 5), natomiast przy dużej ilości testów każdy node coś robi (wykres 6 i 7). Gdy zwiększamy dwukrotnie ilość nodów i testów otrzymujemy dwukrotną poprawę (widać to między innymi w tabeli 4 i 8). Tak więc optymalnym rozwiązaniem jest ilość testów nieznacznie przekraczająca ilość nodów. Za małe ziarno podziału powoduje zwiększenie czasu wykonaniu algorytmu co spowodowane jest czasem potrzebnym na przesyłanie informacji przez MPI.