Michał Piotrowski

Kierunek: Informatyka
Specjalność: Informatyka Stosowana
Numer albumu: 325585

Implementacja równoległego algorytmu genetycznego w języku Python

Praca magisterska
wykonana pod kierunkiem
dr hab. Tomasza M. Gwizdałły
w Katedrze Fizyki Ciała Stałego
WFiIS UŁ

Łódź 2018

Spis treści

1. Wstęp

1.1 Cel

Celem niniejszej pracy dyplomowej było pokazanie, iż możliwa jest implementacja równoległego algorytmu genetycznego za pomocą języka Python. W tym celu zrealizowano autorską implementację algorytmu, a także wykonano analizę porównawczą efektywności działania operatorów genetycznych w algorytmie genetycznym klasycznym oraz zrównoleglonym. Klasyczny algorytm napisano jako algorytm referencyjny, który posłużył za bazę porównawczą. Zrównoleglona wersja wykorzystywała zasoby oferowane przez nowoczesne, wielordzeniowe procesory, pozwalające na zrównoleglanie obliczeń.

1.2 Rodzina algorytmów genetycznych

Algorytmy genetyczne opierają się na mechanizmach doboru naturalnego oraz dziedziczności cech. Zostały rozwinięte przez Johna Hollanda [1], który uważany jest za pioniera w tej dziedzinie informatyki. Jego praca łączy teorię dziedziczenia opisaną przez Charlesa Darwina oraz genetykę zapoczątkowaną przez Gregora Mendla. Holland wyodrębnił operatory genetyczne imitujące mechanizmy zauważalne w przyrodzie. Operatory pracują na populacji wirtualnie utworzonych osobników. Osobniki można reprezentować na kilka sposobów, np. binarnie czy zmiennoprzecinkowo w zależności od zestawu cech. Binarny sposób reprezentacji to po prostu ciąg bitów charakteryzujący konkretnego osobnika. Zmiennoprzecinkowy natomiast do reprezentacji cech wykorzystuje liczby rzeczywiste, które w bardziej realny sposób opisują osobniki. Kolejność występowania operatorów również nie jest bez znaczenia. W początkowej fazie algorytmu dokonywana jest ocena każdego osobnika populacji. Funkcja oceny pozwala określić, które z osobników są najlepiej przystosowane do życia w danym środowisku. Kolejnym etapem algorytmu jest operator selekcji, który wybiera najlepiej przystosowane osobniki do nowej populacji. Ważnym operatorem w całym procesie jest operator krzyżowania, który wymienia informacje genetyczne (cechy) pomiędzy wyselekcjonowanymi osobnikami. Innymi słowy pozwala on na tworzenie nowych osobników, potomków, na podstawie jednostek z aktualnie rozpatrywanego pokolenia. Ostatnim

krokiem w algorytmie genetycznym jest wykorzystanie operatora mutacji. Operator mutacji modyfikuje geny osobników, dzięki czemu możliwy jest rozwój populacji. Istnieją jednak zagadnienia, dla których jego wykorzystanie nie jest konieczne, aby pozwolić danej populacji na eksplorację. Algorytmy genetyczne wykorzystuje się w zagadnieniach, których rozwiązanie w sposób tradycyjny jest niemożliwe bądź trudność takiego rozwiązania jest wysoka. Przykładem problemu, którego nie da się rozwiązać w sposób analityczny jest znajdowanie przybliżeń ekstremów funkcji.

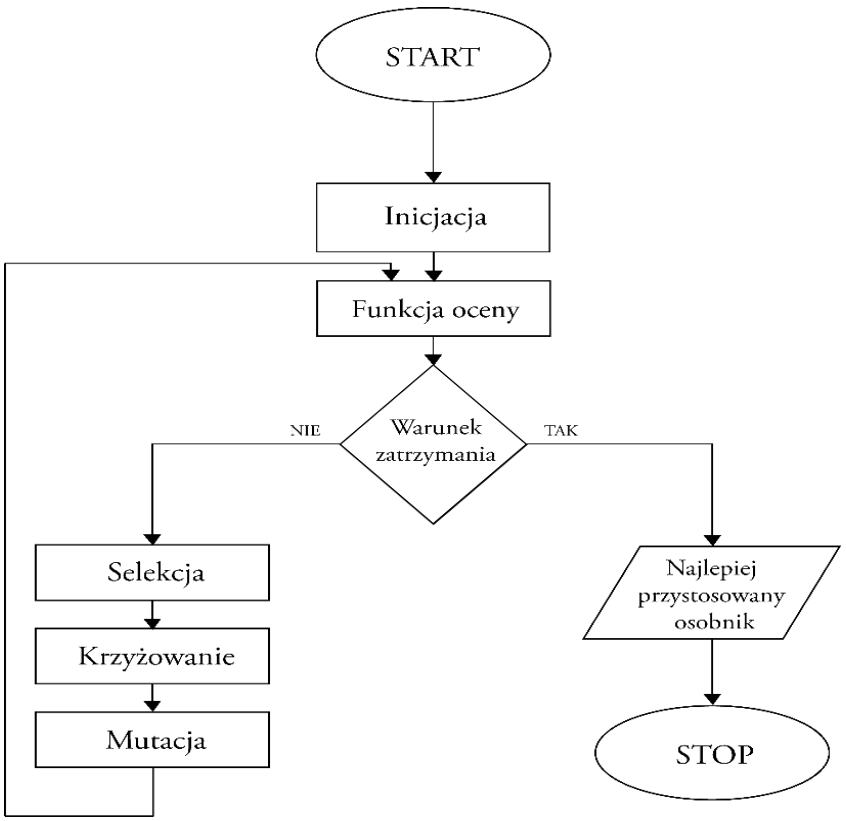
1.3 Wykorzystanie języka Python

Zastosowanie języka Python do zrównoleglania algorytmów genetycznych to dosyć nowe zagadnienie. O ile algorytmy ewolucyjne opisywane były już wiele razy, zrównoleglanie ich, dodatkowo przy użyciu Pythona, nie jest szeroko opisywane. Z pomocą przychodzi pakiet ***DEAP*** [2-4], będący platformą obliczeniową, którą można wykorzystać do szybkiego prototypowania algorytmów. Wykorzystuje on mechanizmy pozwalające na zrównoleglanie obliczeń z wykorzystaniem modułu ***multiprocessing*** czy ***SCOOP*** [5].

Niniejsza praca dyplomowa skupia się na autorskim podejściu do równoległości, również z wykorzystaniem kilku modułów języka Python. Przeanalizowano wady i zalety kilku rozwiązań, a także wybrano najbardziej optymalne w celu implementacji końcowego równoległego algorytmu genetycznego.

2. Podstawowy algorytm genetyczny

Algorytm genetyczny należy do grupy tzw. algorytmów ewolucyjnych. Jak zostało wcześniej wspomniane, sposób działania przypomina zjawisko ewolucji biologicznej występującej w przyrodzie. Problem definiuje się przez pewną populację osobników, z których każdy posiada pewien zbiór cech, określanych jako genotyp. Następnie osobniki są poddawane procesowi ewaluacji za pomocą funkcji oceny. W następnym kroku dokonuje się selekcji najlepiej przystosowanych osobników z aktualnej generacji i dokonuje się krzyżowania wyselekcjonowanych jednostek. By przyspieszyć eksplorację, dokonuje się mutacji pewnej części losowo wybranych osobników. Jeżeli najlepsza jednostka z nowej populacji nie spełnia kryterium funkcji oceny, cały proces powtarza się raz jeszcze. Istnieje oczywiście możliwość, iż algorytm nie będzie w stanie znaleźć szukanego optimum, dlatego dodatkowym zabezpieczeniem powinna być stała definiująca maksymalną ilość generacji, po których należy przerwać działanie algorytmu. Rysunek 1 pokazuje schemat działania algorytmu genetycznego.



Rysunek 1 Schemat działania algorytmu genetycznego.

2.1 Ewaluacja

Proces dostosowania poszczególnych osobników do populacji według jednego lub kilku kryteriów. Są one zazwyczaj zapisywane w postaci funkcji oceny. Algorytmy genetyczne dążą do rozwiązania szukanego zagadnienia. W przypadku niniejszej pracy dyplomowej zostały wybrane dwie funkcje testowe: wielowymiarowe implementacje funkcji Rosenbrocka oraz funkcji Griewanka, które zostały opisane w podrozdziale 3.2.

2.2 Selekcja

Selekcja jako jedyny proces w algorytmie genetycznym wymaga znajomości całej populacji, a właściwie informacji o dopasowaniu poszczególnych osobników do populacji jako całości. Na tym etapie zwykle selekcjonuje się osobniki najlepiej dostosowane, które mają największe szanse „na przeżycie" według szukanego kryterium. Istnieje oczywiście więcej sposobów selekcji, natomiast opisane metody są jednymi z najpopularniejszych metod stosowanych przy implementacjach algorytmów genetycznych.

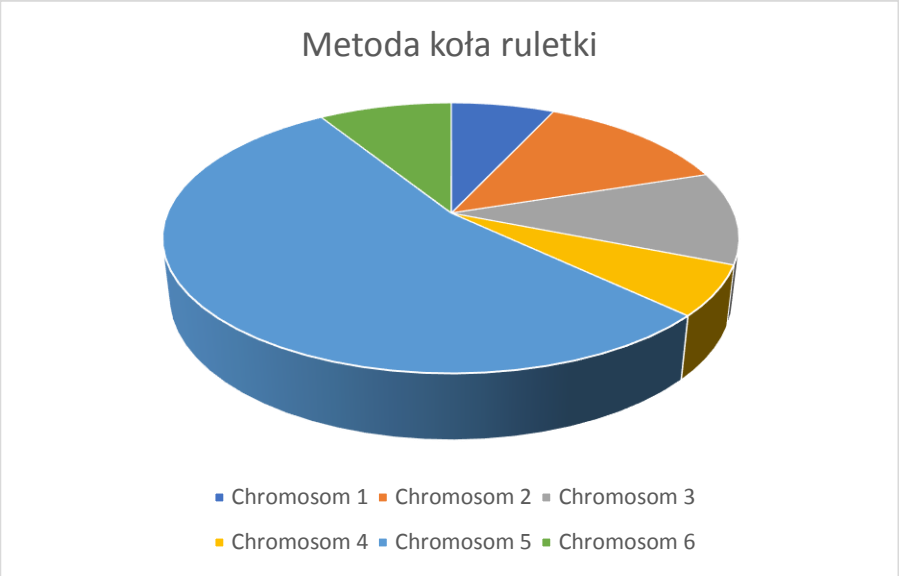
2.2.1 Metoda rankingowa

Jest to jedna z najprostszych dostępnych metod selekcji. Po wykonaniu ocen osobników z danej populacji, wystarczy ustawić je w szeregu od najlepiej do najmniej dostosowanego. Najlepiej k przystosowanych osobników z populacji dostaje prawo do krzyżowania, te najmniej pasujące zostają usunięte. Zaletą tej metody jest dobre rozróżnienie osobników pod względem dopasowania do populacji. Wadą tej metody jest pominięcie różnic pomiędzy genotypami osobników, przez co realne staje się, iż gorzej przystosowane jednostki będą krzyżować się tak samo często jak te lepiej przystosowane.

2.2.2 Metoda koła ruletki

Inną metodą selekcji może być metoda koła ruletki. Polega na zbudowaniu wirtualnego koła, którego poszczególne wycinki odpowiadają osobnikom. Im lepiej przystosowany osobnik do danej populacji, tym większy kawałek wirtualnego koła otrzymuje, przez co dostaje większą szansę na udział w procesie krzyżowania. Wadą takiego podejścia jest zwalnianie ewolucji z każdą kolejną iteracją algorytmu, ponieważ coraz bardziej podobne do

siebie osobniki otrzymują zbliżone wycinki koła. Metoda ta słabiej rozróżnia osobniki dobrze przystosowane od tych gorzej przystosowanych.



Rysunek 2 Wizualizacja wirtualnego kola ruletki.

2.2.3 Metoda turniejowa

W pierwszym kroku wybierane jest losowo N osobników z populacji P (bez zwracania). N oznacza rozmiar turnieju. Następnie wybierany jest najlepszy osobnik do populacji rodziców T. W następnym kroku zwracamy N osobników z powrotem do populacji P. Należy zorganizować turniej odpowiednią ilość razy, tak aby zapewnić żądaną wielkość populacji rodziców T, z której dokona się krzyżowanie. Zaletą tej metody jest to, iż nie wymaga informacji o każdym z osobników populacji, przez co jest dużo łatwiejsza w implementacji w algorytmach równoległych. Jest to powód, dla którego zdecydowano się na wybór tej metody selekcji przy implementacji projektu dyplomowego.

2.3 Krzyżowanie

Głównym zadaniem operatorów krzyżowania jest łączenie w losowy sposób cech osobników populacji wybranych do krzyżowania. Na tym etapie wybrane pary nie mają wpływu na inne wybrane pary osobników. W niniejszej pracy dyplomowej wybrano zastosowanie operatora krzyżowania liniowego [6]. W tej metodzie na podstawie informacji z dwójki

rodziców kreowane są trzy osobniki potomne, z których w każdym kolejnym kroku osobnik z najgorszą wartością funkcji oceny (najgorzej przystosowany) zostaje pominięty, a pozostała dwójka zastępuje w populacji swoich rodziców. Krzyżowanie odbywa się wg wzoru:

$$potomek_1 = 1.5 * rodzic_1 - 0.5 * rodzic_2$$
$$potomek_2 = 0.5 * rodzic_1 - 1.5 * rodzic_2$$
$$potomek_3 = 0.5 * rodzic_1 + 0.5 * rodzic_2$$

Wzór 1 Operator krzyżowania liniowego.

2.4 Mutacja

Mutacja to operacja losowa, która wprowadza do genotypu osobnika zmiany. Jej zadaniem jest urozmaicenie populacji. Prawdopodobieństwo mutacji powinno być ustalone na niskim poziomie, aby subtelnie różnicować osobniki. Zbyt duża wartość mutacji wpływa niekorzystnie na poszukiwanie dobrego rozwiązania. W przeciwieństwie do innych operatorów, mutacja ze względu na probabilistyczny charakter działania może nie wykonać żadnej operacji na danym osobniku. W pierwszym kroku losowany jest współczynnik prawdopodobieństwa ***factor*** (przedział 0-1). Jeżeli wylosowana wartość jest mniejsza niż określona stała ***p_mute***, następuje mutacja genów osobnika. W przeciwnym wypadku osobnik nie ulega zmianie.

3. Implementacja algorytmu genetycznego

Podczas implementacji kodu starano się wydzielić poszczególne etapy algorytmu do osobnych funkcji. Aby uzyskać możliwość wykorzystania tego samego kodu zarówno w programie iteracyjnym oraz w programie wykorzystującym wielozadaniowość, funkcje zostały zaimplementowane w sposób umożliwiający podanie zakresu populacji, na której należy wykonać odpowiednie operacje.

3.1 Reprezentacja osobników

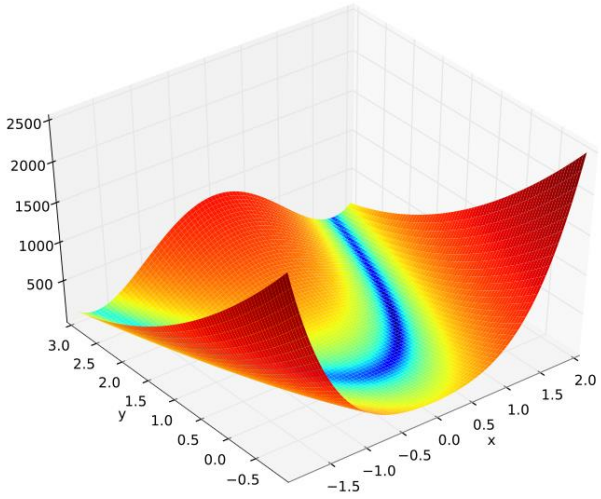
Istnieje kilka sposobów reprezentacji osobników, takie jak binarny czy zmiennoprzecinkowy. Binarny sposób reprezentowania osobników stosuje się w problemach, w których wystąpienie lub brak wystąpienia konkretnej, specyficznej cechy jest wystarczające do określenia osobnika. W implementacji algorytmu genetycznego do opisu cech osobnika została natomiast wybrana reprezentacja zmiennoprzecinkowa, ponieważ jest ona bardziej adekwatna dla badanych zagadnień optymalizacyjnych.

3.2 Funkcje testowe

Funkcje testowe zostały umieszczone w module ***optimization.py***. Dzięki modularnemu podejściu, w każdej chwili możliwe jest dodanie innych funkcji testowych, które działają na wielowymiarowych danych, mających ten sam sposób zapisu osobników. Do testów zostały wybrane dwie funkcje testowe: funkcja Rosenbrocka oraz funkcja Griewanka.

3.2.1 Funkcja Rosenbrocka

Niewypukła, jednomodalna funkcja, niezwykle często używana w algorytmach optymalizacyjnych ze względu na swoją charakterystykę i przewidywalne zachowanie. Jej kształt powoduje, iż często nazywana jest „Doliną Rosenbrocka” lub „Funkcją Bananową Rosenbrocka”. Minimum globalne funkcji można zlokalizować wewnątrz parabolicznego wgłębienia. Dla dwuwymiarowej funkcji Rosenbrocka, minimum lokalne znajduje się w punkcie ***(x,y) = (1,1)***.



Rysunek 3 Wykres funkcji Rosenbrocka dla dwóch zmiennych[1]

Zostało zaimplementowane wielowymiarowe rozwinięcie funkcji Rosenbrocka, które definiuje się wzorem:

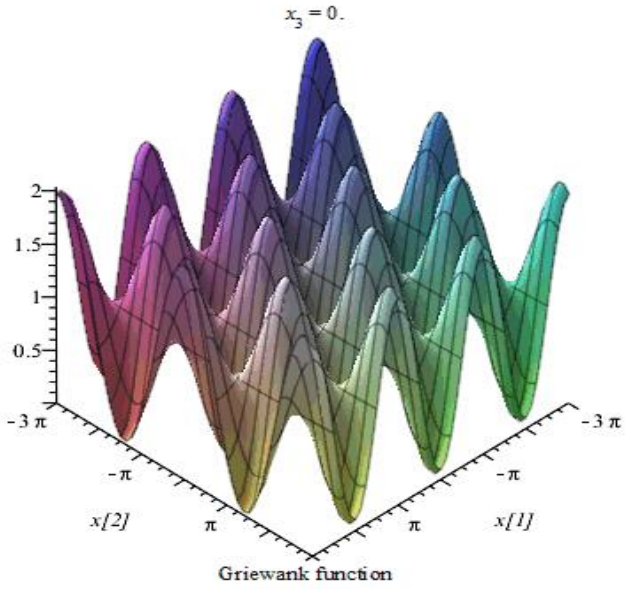
$$f(x) - \sum_{i=1}^{N-1} [(1 - x_i)^2 + 100(x_{i+1} - x_i^2)^2] \quad \forall_x \in \mathrm{R}^N$$

Wzór 2 Wielowymiarowe rozwinięcie funkcji Rosenbrocka.

3.2.2 Funkcja Griewanka

W przeciwieństwie do funkcji Rosenbrocka, funkcja Griewanka należy do funkcji wielomodalnych, przez co znalezienie rozwiązania optymalnego może przysparzać trudności. Jest ona równie często wykorzystywana w algorytmach optymalizacji. Poza prostymi operacjami mnożenia i dzielenia, Griewank zawiera także funkcje trygonometryczne, przez co wzrasta liczba obliczeń. Czasy obliczeń powinny być dłuższe niż przy funkcji Rosenbrocka.

[1] https://en.wikipedia.org/wiki/Rosenbrock_function



Rysunek 4 Trzeciorzędowa funkcja Griewanka[2]

Do testów została wykorzystana wielowymiarowa implementacja funkcji Griewanka, którą definiuje się wzorem:

$$1 + \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos(\frac{x_i}{\sqrt{i}})$$

Wzór 3 Wielowymiarowe rozwinięcie funkcji Griewanka.

3.3 Metodyka testów

Jak zostało wspomniane wcześniej, każdy etap algorytmu genetycznego został wyodrębniony do oddzielnej funkcji, aby ułatwić testowanie i porównywanie czasu wykonania poszczególnych implementacji. Pomiar czasu dla każdego etapu (ewaluacja, selekcja, krzyżowanie, mutacja) został wykonany dziesięć razy, natomiast czasy widoczne na wykresach to wynik uśrednienia tych wyników. Warto wspomnieć, iż analiza procesu ewaluacji była wstępnym kryterium dopuszczającym poszczególne techniki do dalszego wykorzystania. Oznacza to, że jeżeli któryś ze sposobów został uznany za nieefektywny, inne operatory genetyczne nie były już implementowane.

3.4 Platforma sprzętowa

Wszystkie testy przeprowadzone zostały na tej samej platformie sprzętowej. Do testów wykorzystano odpowiedni skonfigurowane środowisko języka

[2] https://en.wikipedia.org/wiki/Griewank_function

Python zainstalowane na serwerze VPS wykupionym w ramach usługi na jednym z polskich serwisów hostingowych. Konfiguracja wykupionego serwera prezentuje się następująco:

Zasoby serwera	
Procesor	Intel Xeon E5
vCore	4
RAM	8 GB
Powierzchnia dyskowa	100 GB
Rodzaj wirtualizacji	KVM

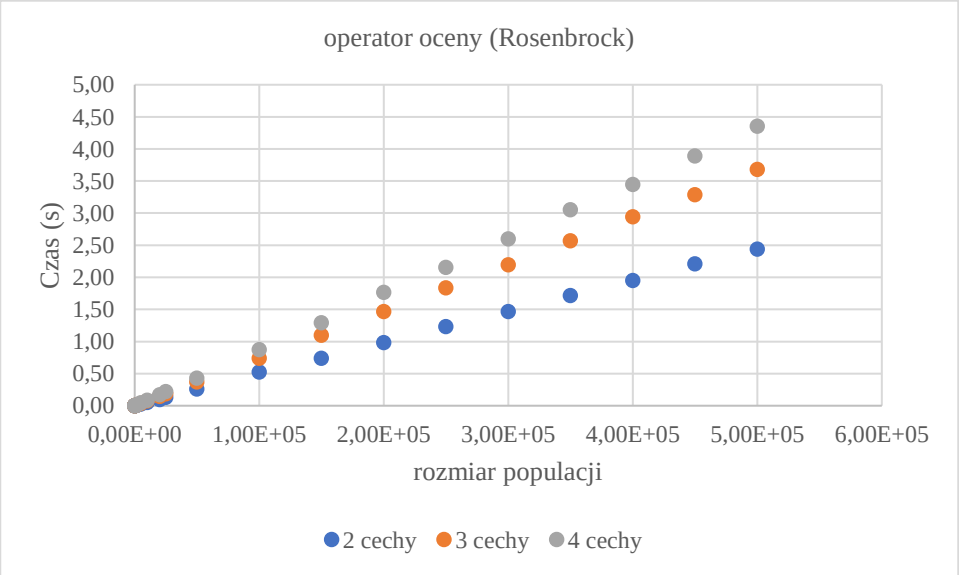
Tabela 1 Konfiguracja platformy sprzętowej. Serwer VPS

Na serwerze został zainstalowany system Ubuntu 16.04.3 LTS. Następnie zostało pobrane środowisko języka Python z oficjalnej strony projektu. Wersja środowiska użyta przy implementacji to Python 3.5.2 (default, Nov 23 2017, 16:37:01).

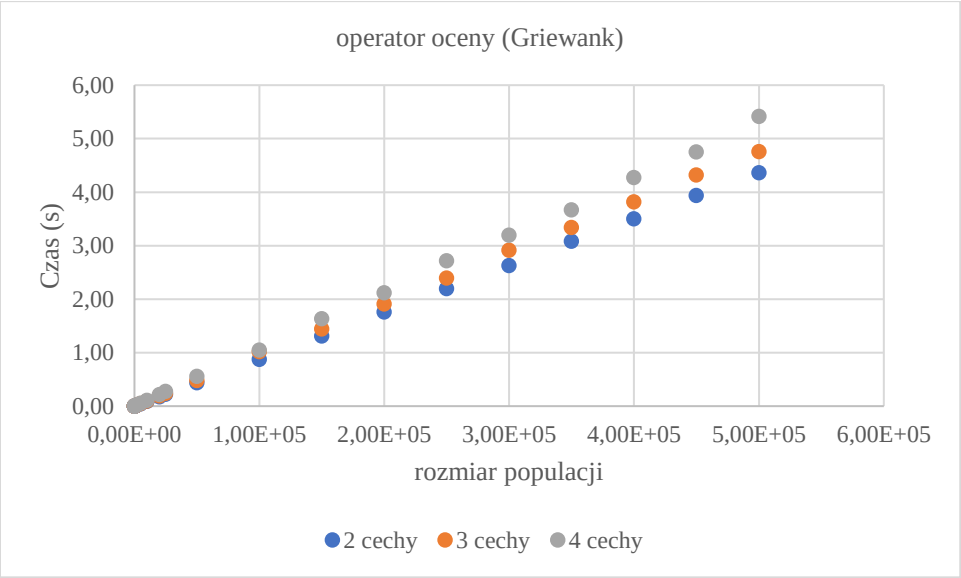
3.5 Testy

3.5.1 Ewaluacja – algorytm iteracyjny

W prostym algorytmie genetycznym, ewaluacja to po prostu wykonanie funkcji oceny dla całej populacji. Rysunek 5 oraz Rysunek 6 prezentują czasy wykonania operatorów oceny w zależności od wielkości populacji oraz wymiaru funkcji testowej (liczby cech genetycznych).



Rysunek 5 Czas wykonania funkcji Rosenbrocka w zależności od wielkości populacji.

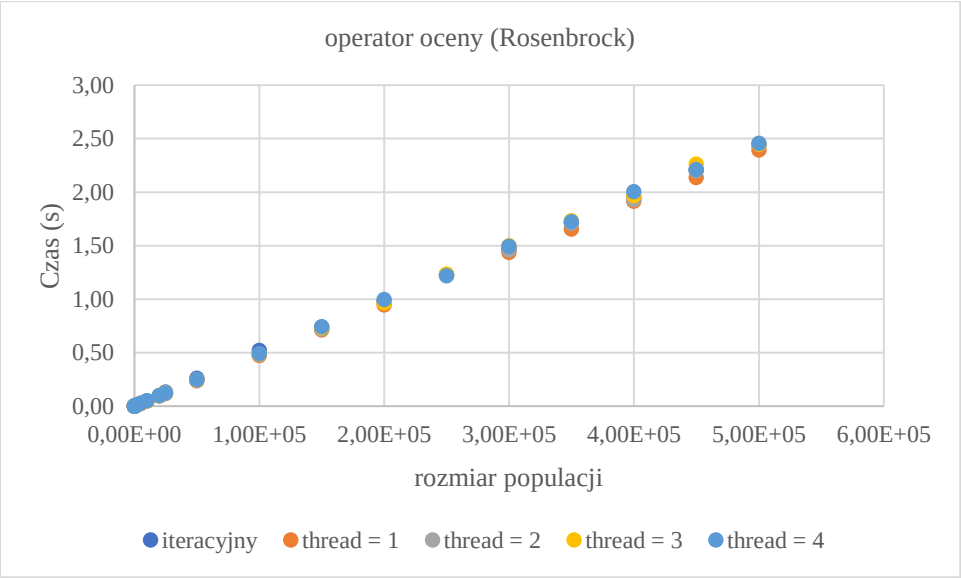


Rysunek 6 Czas wykonania funkcji Griewanka w zależności od wielkości populacji.

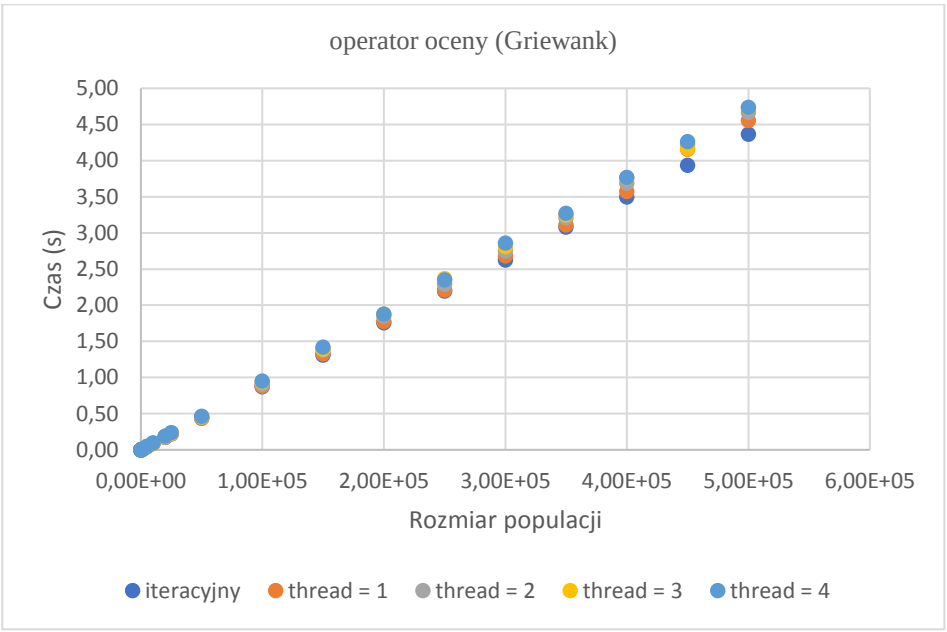
Zgodnie z przewidywaniami, uśrednione czasy wykonania funkcji Rosenbrocka oraz Griewanka rosną wraz ze wzrostem ilości cech genetycznych. Zauważyć można również różnicę w czasie wykonania pomiędzy obiema funkcjami testowymi. Czasy wykonania funkcji Griewanka są dłuższe niż czasy wykonania funkcji Rosenbrocka, z racji charakteru obu funkcji (więcej obliczeń w przypadku funkcji Griewanka).

3.5.2 Ewaluacja – algorytm równoległy

Pierwsze podejście do problemu równoległości dotyczyło użycia modułu ***threading***. Dzięki prostemu w użyciu API, uruchomienie wątków jest bardzo prostym zabiegiem. Aby poprawnie rozdzielić obliczenia na żądaną liczbę wątków, należało podać funkcji ewaluacji odpowiedni przedział danych, a następnie uruchomić wątek, wskazując na funkcję oceny. Rysunki 7 i 8 pokazują czasy wykonania operatorów oceny przy użyciu modułu threading. ***Thread*** widoczne w legendzie oznacza liczbę wątków, na których została uruchomiona funkcja oceny.



Rysunek 7 Czas wykonania funkcji Rosenbrocka w zależności od wielkości populacji. Porównanie czasu wykonania dwuwymiarowej funkcji testowej dla algorytmu iteracyjnego oraz zrównoleglonego za pomocą modułu threading.

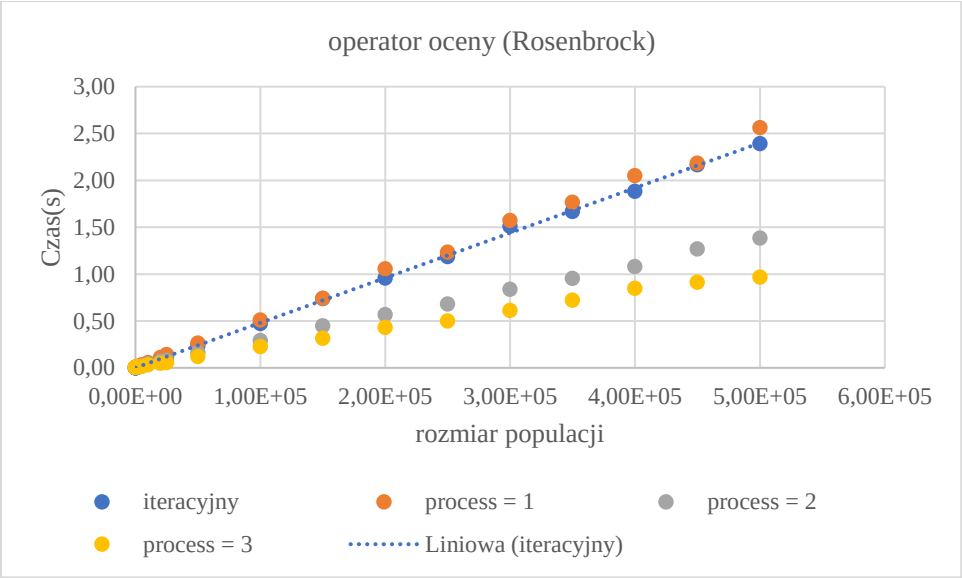


Rysunek 8 Czas wykonania funkcji Griewanka w zależności od wielkości populacji. Porównanie czasu wykonania dwuwymiarowej funkcji testowej dla algorytmu iteracyjnego oraz zrównoleglonego za pomocą modułu threading.

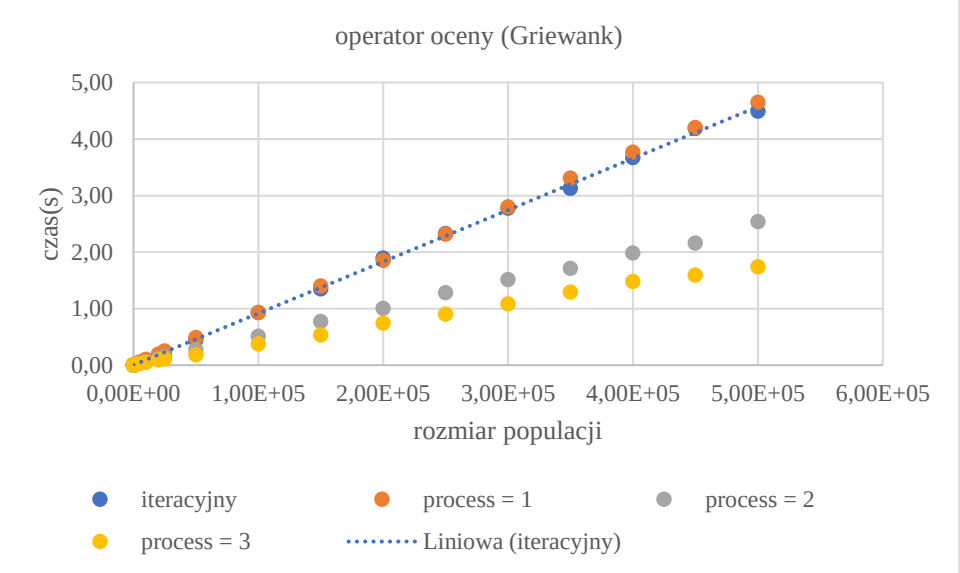
Można zauważyć, iż wykorzystanie modułu ***threading*** nie ma żadnego przełożenia na szybkość obliczeń. Na początku wydaję się to nieintuicyjne, natomiast wynika z implementacji języka Python, która została użyta przy pisaniu niniejszej pracy dyplomowej [6]. Global Interpreter Lock, opisany w podrozdziale 4.1.3, blokuje możliwość wykorzystania wielowątkowości przy pomocy tej biblioteki, przez co uzyskane czasy są niemal identyczne z algorytmem iteracyjnym (a nawet gorsze). W związku z powyższym moduł

threading okazał się niewystarczający dla założonego problemu, dlatego dalsze części algorytmu nie będą już testowane przy użyciu tego modułu.

Moduł ***multiprocessing*** pozwala na ominięcie ograniczeń spowodowanych przez mutex GIL oraz pełne wykorzystanie wielu rdzeni, które posiada procesor. API modułu jest bliźniaczo podobne do API, które posiadał moduł ***threading***, zatem implementacja obu kawałków kodu jest niemal identyczna. Multiprocessing pozwala również na dwojakie podejście do problemu, z wykorzystaniem klasy ***Process*** oraz ***Pool***. Używając **Pool** nie jest konieczne wskazanie przedziałów, na których poszczególne procesy powinny wykonać funkcję oceny, natomiast konieczna jest implementacja operatorów genetycznych, które operują na jednym osobniku. Zdecydowano, iż implementacja algorytmu z użyciem **Pool** nie będzie testowana, ponieważ zaburzyłaby spójność operatorów, co z kolei utrudniłoby przeprowadzenie rzeczowych testów. Podobnie jak przy module ***threading***, wykorzystując klasę ***Process*** z modułu ***multiprocessing***, należy podzielić obliczenia pomiędzy żądaną liczbę procesów. Zgodnie z oficjalna dokumentacją, zaleca się użycie ***n-1*** procesów, gdzie ***n*** oznacza liczbę procesorów dostępną na danej platformie sprzętowej.
Wykresy 9 i 10 przedstawiają czas wykonania operatorów oceny dla dwuwymiarowych funkcji Rosenbrocka oraz Griewanka.



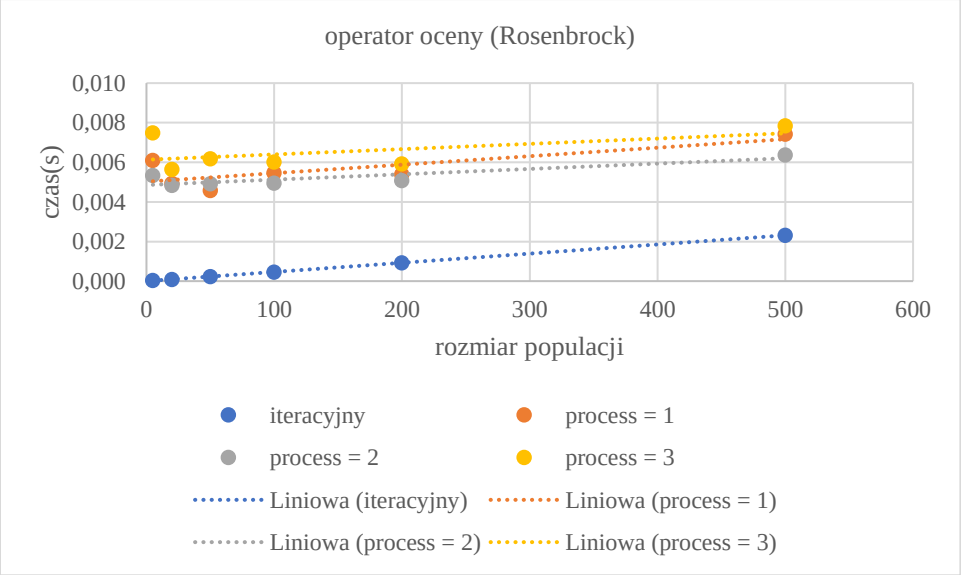
Rysunek 9 Czas wykonania funkcji Rosenbrocka w zależności od wielkości populacji. Porównanie czasu wykonania dwuwymiarowej funkcji testowej dla algorytmu iteracyjnego oraz zrównoleglonego za pomocą modułu multiprocessing.



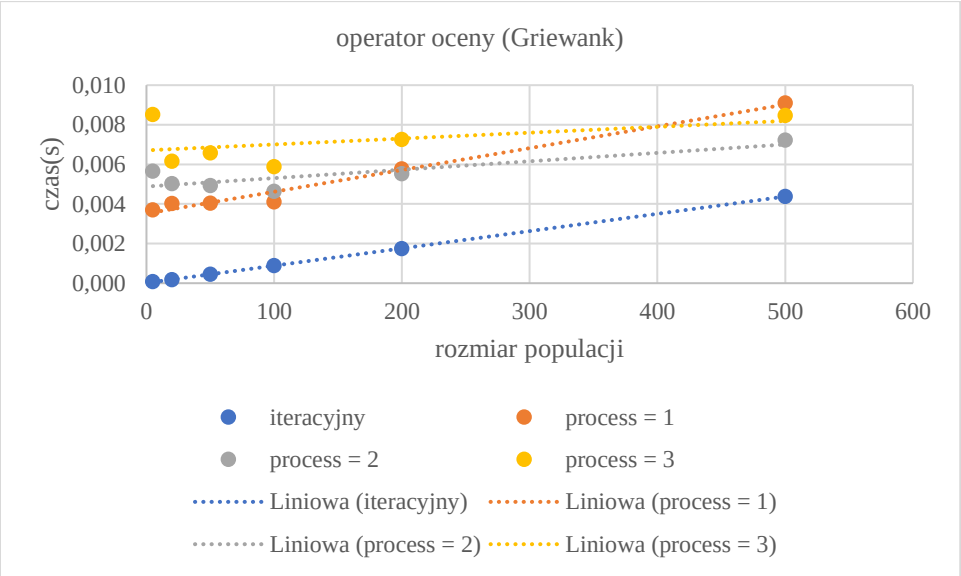
Rysunek 10 Czas wykonania funkcji Griewanka w zależności od wielkości populacji. Porównanie czasu wykonania dwuwymiarowej funkcji testowej dla algorytmu iteracyjnego oraz zrównoleglonego za pomocą modułu multiprocessing.

Można zauważyć, iż wykorzystanie klasy ***Process*** z modułu ***multiprocessing*** pozwala ominąć mutex GIL, przez co zauważalne stają się różnice w czasie wykonania operatorów oceny na kilku procesach względem algorytmu iteracyjnego. W obu przypadkach najdłuższe czasy wykonania widoczne są dla algorytmu uruchomionego na jednym procesie. Jest to przewidywalna właściwość, ponieważ uruchomienie obliczeń na jednym procesie powoduje

dodatkowy narzut względem algorytmu iteracyjnego stąd przypuszczalnie czas wykonania powinien być gorszy względem implementacji referencyjnej. Zauważalna jest różnica czasu wykonania dla dużych populacji. Poniższe wykresy prezentują czas wykonania dla populacji do pięciuset osobników. Należy zwrócić uwagę, iż im większa liczba procesów jest uruchomiona, czas wykonania operatora oceny jest wyższy dla małych populacji. Jest to również zachowanie przewidywalne, ponieważ każde uruchomienie dodatkowego procesu generuje dodatkowy narzut czasowy wydłużając tym samym czas obliczeń.

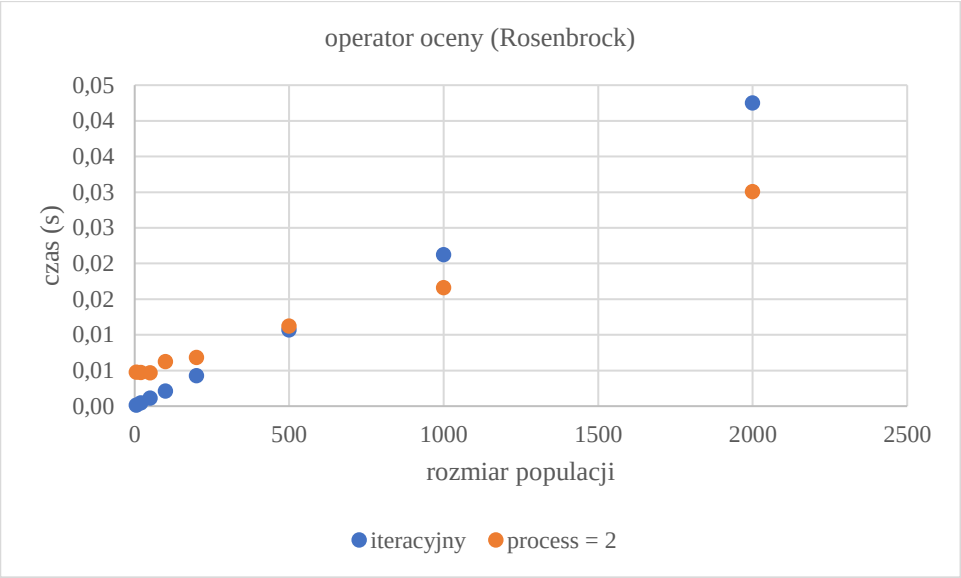


Rysunek 11 Czas wykonania funkcji Rosenbrocka w zależności od wielkości populacji. Porównanie czasu wykonania dwuwymiarowej funkcji testowej dla algorytmu iteracyjnego oraz zrównoleglonego za pomocą modułu multiprocessing.

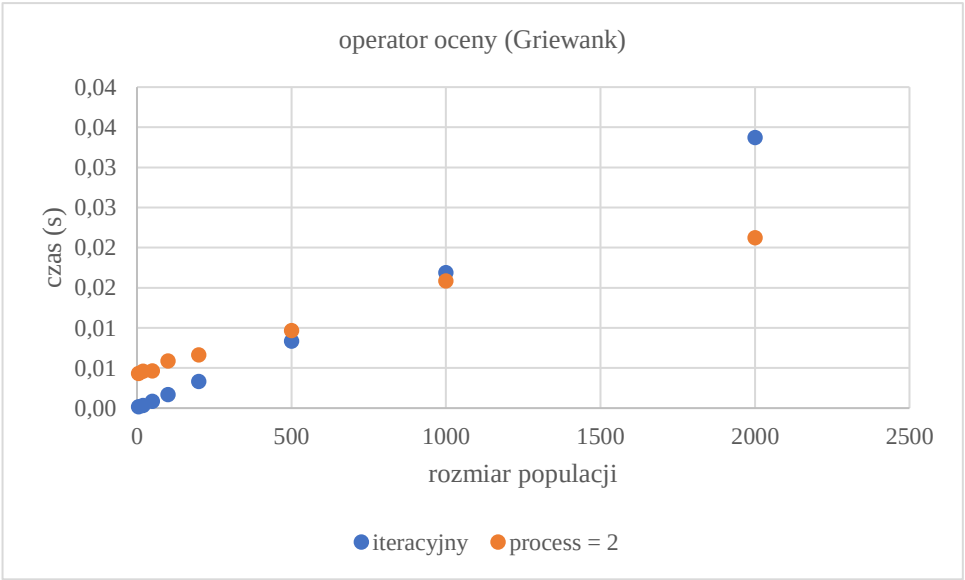


Rysunek 12 Czas wykonania funkcji Griewanka w zależności od wielkości populacji. Porównanie czasu wykonania dwuwymiarowej funkcji testowej dla algorytmu iteracyjnego oraz zrównoleglonego za pomocą modułu multiprocessing.

Na podstawie testów można wnioskować, iż problem optymalizacyjny powinien być odpowiednio skomplikowany obliczeniowo, tak aby uruchomienie kilku procesów nie było przyczyną większości narzutu czasowego podczas obliczeń. Kolejnym logicznym posunięciem wydaje się zatem sprawdzenie działania operatorów oceny dla funkcji Rosenbrocka i Griewanka o większym wymiarze. Wykonano testy porównujące czasy wykonania operatorów oceny dla algorytmu iteracyjnego oraz z użyciem klasy **Process** z modułu **multiprocessing**. Ilość procesów ustalono na dwa. Poniższe wykresy prezentują wartości zmierzone dla dziesięciowymiarowych funkcji testowych.



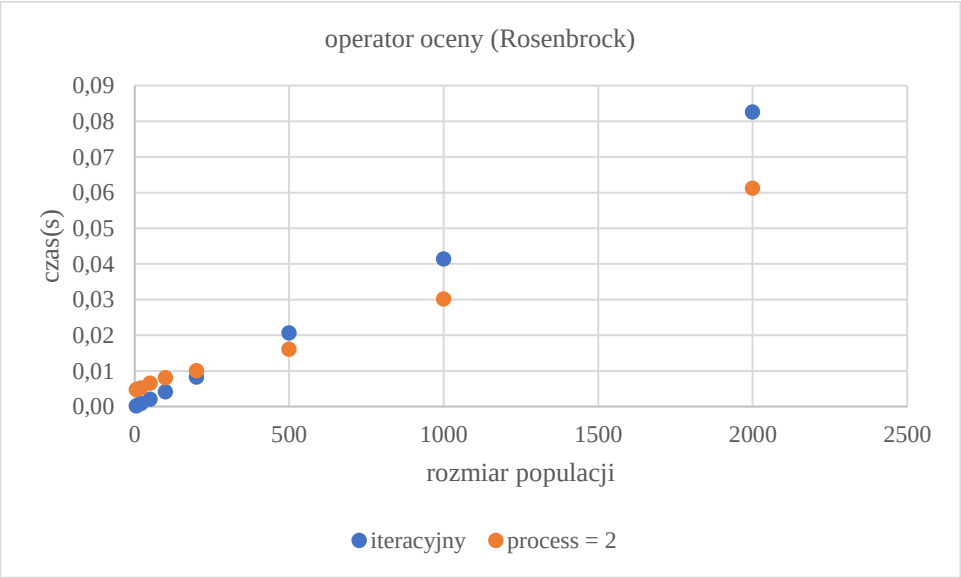
Rysunek 13 Czas wykonania funkcji Rosenbrocka w zależności od wielkości populacji. Porównanie czasu wykonania dziesięciowymiarowej funkcji testowej dla algorytmu iteracyjnego oraz zrównoleglonego za pomocą modułu multiprocessing, z wykorzystaniem dwóch procesów.



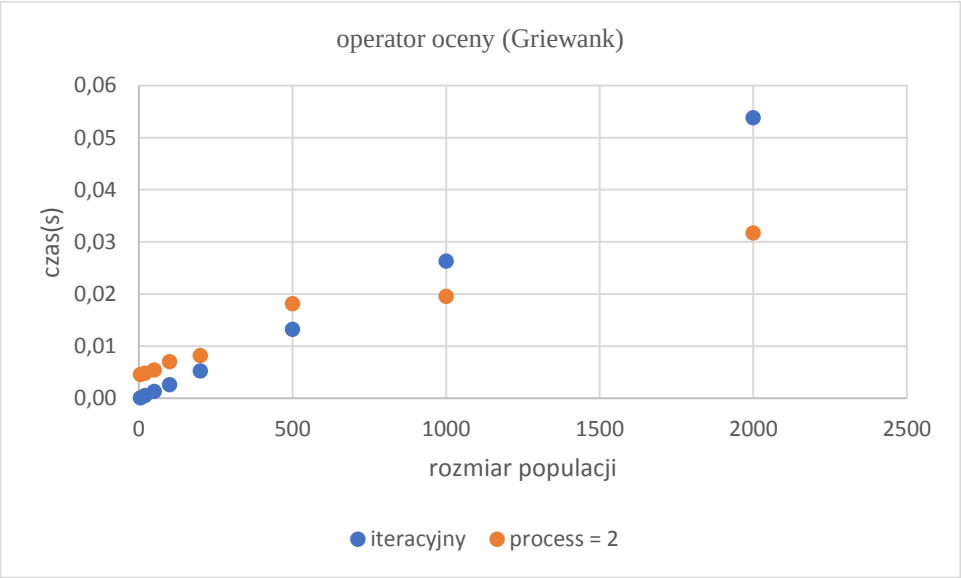
Rysunek 14 Czas wykonania funkcji Griewanka w zależności od wielkości populacji. Porównanie czasu wykonania dziesięciowymiarowej funkcji testowej dla algorytmu iteracyjnego oraz zrównoleglonego za pomocą modułu multiprocessing, z wykorzystaniem dwóch procesów.

Wyniki obu testów pokazały, iż operator oceny zaimplementowany w algorytmie równoległym dla dziesięciowymiarowych funkcji testowych osiąga lepsze wyniki dla populacji wielkości tysiąca osobników. Przy dwuwymiarowych funkcjach testowych granicą opłacalności były około dwustutysięczne populacje. Kolejne testy przeprowadzono dla funkcji o jeszcze większych wymiarach. Poniższe wykresy pokazują czasy

wykonania operatorów oceny dla dwudziestowymiarowych funkcji testowych.



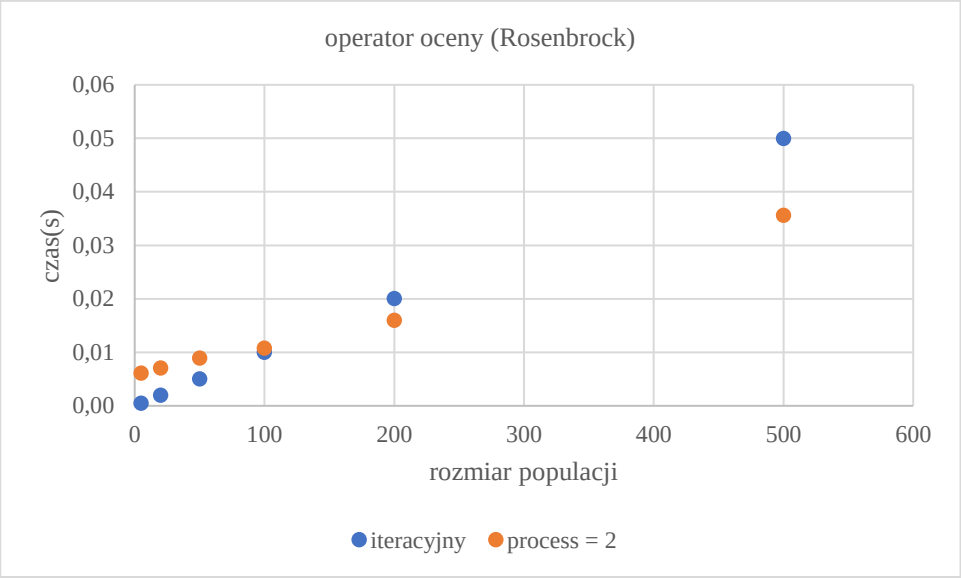
Rysunek 15 Czas wykonania funkcji Rosenbrocka w zależności od wielkości populacji. Porównanie czasu wykonania dwudziestowymiarowej funkcji testowej dla algorytmu iteracyjnego oraz zrównoleglonego za pomocą modułu multiprocessing, z wykorzystaniem dwóch procesów



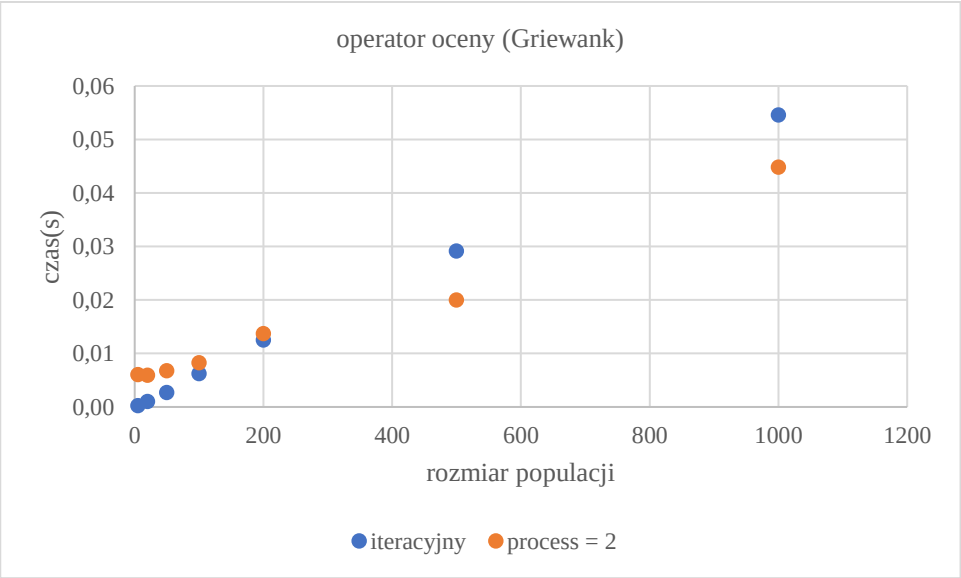
Rysunek 16 Czas wykonania funkcji Griewanka w zależności od wielkości populacji. Porównanie czasu wykonania dwudziestowymiarowej funkcji testowej dla algorytmu iteracyjnego oraz zrównoleglonego za pomocą modułu multiprocessing, z wykorzystaniem dwóch procesów

W przypadku funkcji testowej Rosenbrocka, zauważalny jest kolejny spadek wielkości populacji, przy której zastosowanie obliczeń zrównoleglonych wydaje się bardziej sensowne. Zwiększając rozmiar funkcji testowych czy rozmiar populacji na których dokonywane są obliczenia, początkowy narzut czasowy spowodowany uruchomieniem procesów zmniejsza swój udział

w ogólnym czasie obliczeń. W przypadku testu wykonanego przy użyciu funkcji Griewanka, nie zauważono poprawy, natomiast zauważalny jest pewien skok czasu wykonania funkcji dla populacji wielkości pięciuset osobników. Możliwe, iż jest to błąd statystyczny.



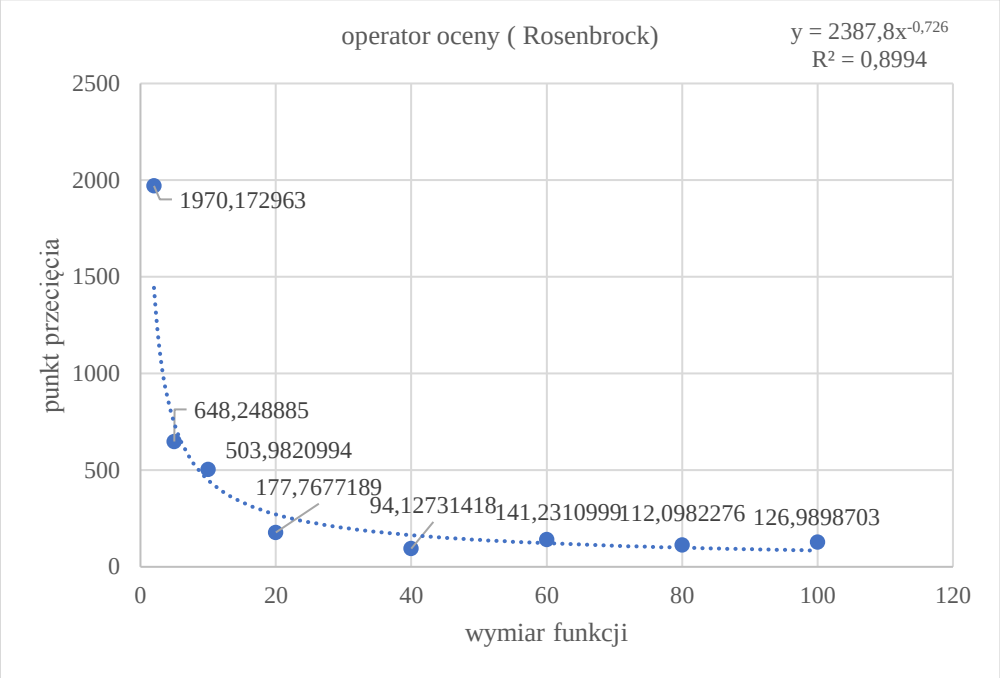
Rysunek 17 Czas wykonania funkcji Rosenbrocka w zależności od wielkości populacji. Porównanie czasu wykonania dwudziestowymiarowej funkcji testowej dla algorytmu iteracyjnego oraz zrównoleglonego za pomocą modułu multiprocessing, z wykorzystaniem dwóch procesów.



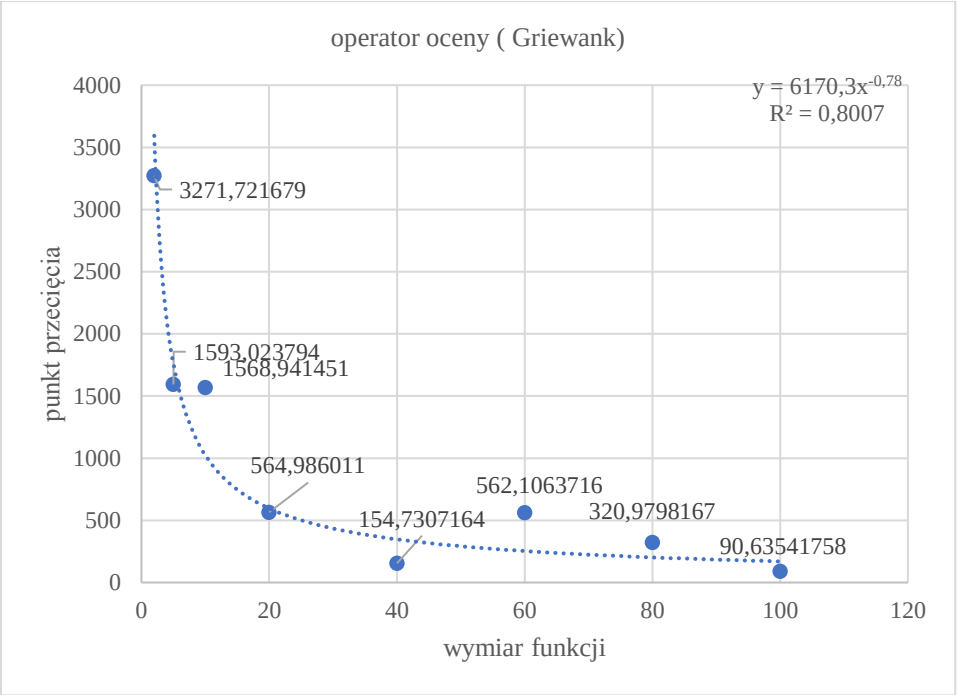
Rysunek 18 Czas wykonania funkcji Griewanka w zależności od wielkości populacji. Porównanie czasu wykonania dwudziestowymiarowej funkcji testowej dla algorytmu iteracyjnego oraz zrównoleglonego za pomocą modułu multiprocessing, z wykorzystaniem dwóch procesów.

Można zauważyć, iż zwiększenie rozmiaru funkcji Rosenbrocka do pięćdziesięciu wymiarów powoduje kolejny wzrost opłacalności dla algorytmu zaimplementowanego w sposób równoległy. Testy

z wykorzystaniem funkcji Griewanka wydają się w tym przykładzie wyraźnie lepsze dla populacji wielkości pięciuset osobników. Dla dwustu czas jest zbliżony, jednak ostatecznie gorszy niż implementacja iteracyjna. Rysunek 19 oraz Rysunek 20 pokazują dla jak dużych rozmiarów populacji w zależności od wymiaru funkcji testowej wykorzystanie dwóch procesów obliczeniowych przynosi lepsze rezultaty.

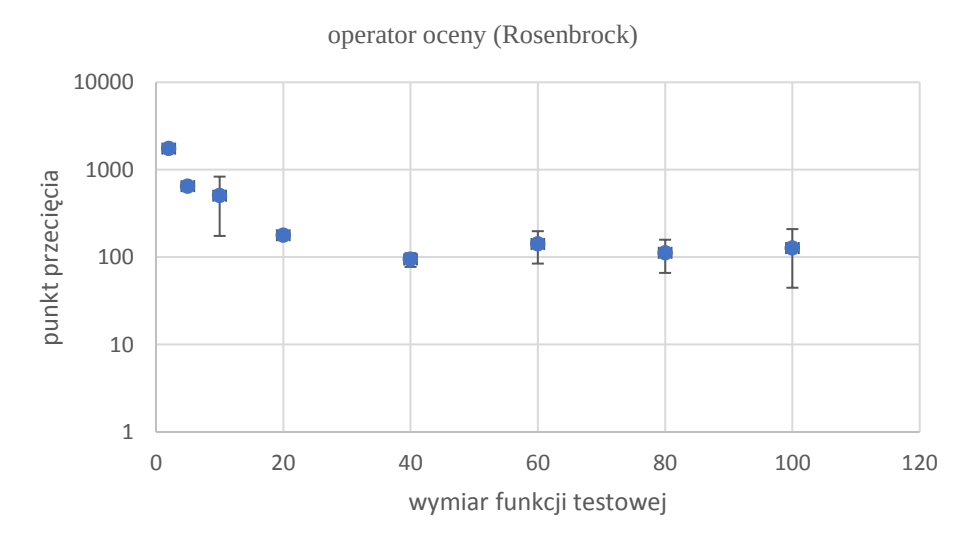


Rysunek 19 Opłacalność stosowania dwóch procesów obliczeniowych w zależności od wymiaru funkcji testowej.

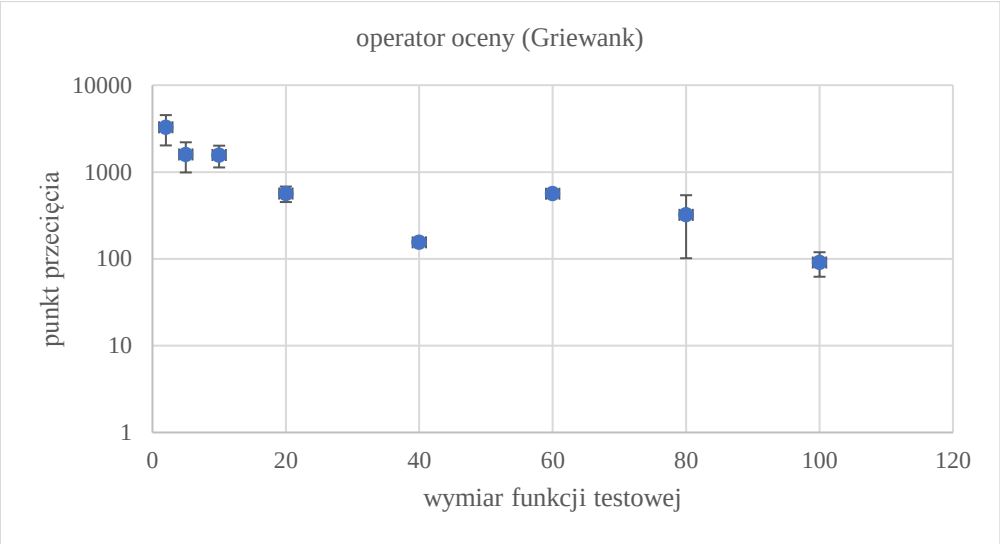


Rysunek 20 Opłacalność stosowania dwóch procesów obliczeniowych w zależności od wymiaru funkcji testowej.

Na podstawie wykonanych testów można wywnioskować, iż utworzenie procesów jest kosztowne. Koszt utworzenia procesów jest na tyle duży, iż równoległa implementacja algorytmu genetycznego nabiera sensu dopiero dla odpowiednio skomplikowanych problemów optymalizacyjnych. Im większa ilość cech genetycznych osobników, tym bardziej implementacja równoległa staje się sensowna. Poniższe wykresy prezentują wartości błędów dokonanych pomiarów.



Rysunek 21 Wartości błędów dokonanych pomiarów dla funkcji testowej Rosenbrocka



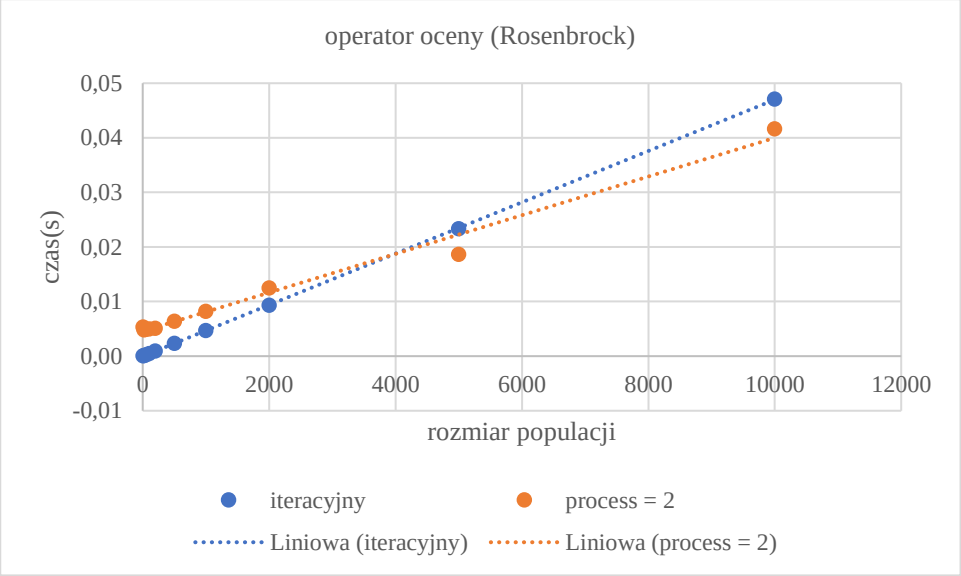
Rysunek 22 Wartości błędów dokonanych pomiarów dla funkcji testowej Griewanka

Wykresy pokazują niepewność pomiaru, zarówno dla funkcji testowej Rosenbrocka oraz Griewanka. O ile zauważalne jest, iż dla funkcji testowych o wyższych wymiarach punkty przecięcia krzywych dla algorytmu iteracyjnego i równoległego przesuwają się w pożądanym kierunku, o tyle istnieje dość duże prawdopodobieństwo uzyskania kompletnie różnych czasów (a zatem również większych odchyleń) w każdym uruchomionym teście. Powodem uzyskania niepewności pomiaru może być zarówno zbyt mała próba wykonanych testów, jak również liczba uruchomionych procesów. Im większa liczba procesów, tym większy narzut początkowy, przez co opłacalność stosowania implementacji równoległej osiąga się dla większych populacji. Kolejnym istotnym czynnikiem jest platforma sprzętowa na której dokonuje się pomiaru. Lepsze konfiguracje sprzętowe prawdopodobnie pozwolą uzyskać lepsze czasy algorytmu iteracyjnego, przez co opłacalność stosowania algorytmu równoległego może się zmienić. Aby upewnić się, iż na innej platformie testowej, uzyskane wyniki mogą odbiegać od tych opisanych w niniejszej pracy dyplomowej, przeprowadzono dodatkowe testy etapu ewaluacji na innej konfiguracji testowej. Tym razem testy przeprowadzono na zwykłym domowym komputerze, z systemem Windows 7. Konfigurację komputera, na którym wykonano dodatkowe testy prezentuje Tabela 1.

HP EliteBook 840 G3	
Procesor	Intel Core i5-6300U CPU 2.40Ghz
vCore	4
RAM	8 GB
Powierzchnia dyskowa	240 GB

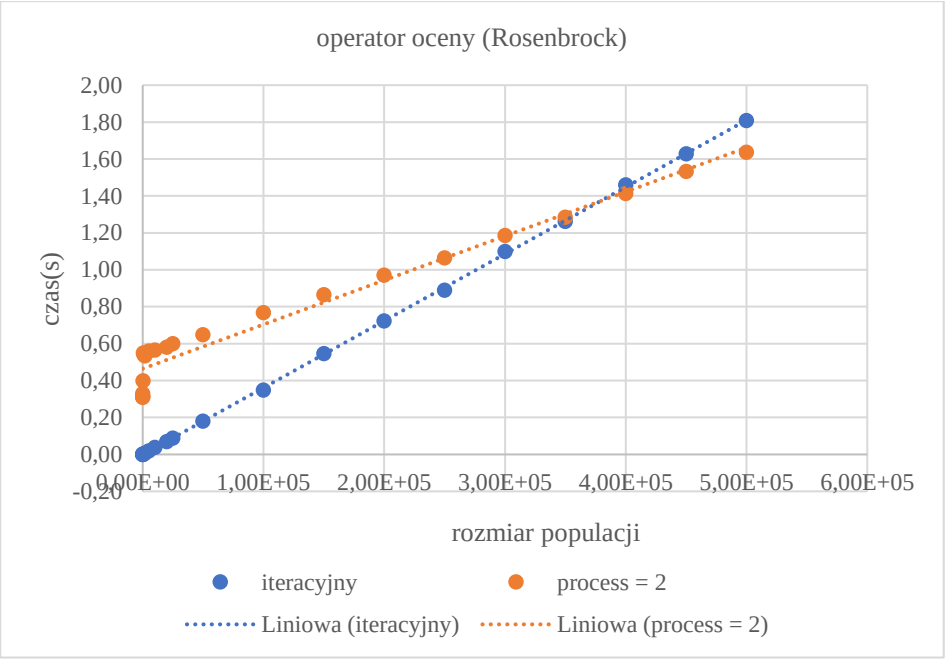
Tabela 2 Konfiguracja komputera testowego HP EliteBook 840 G3.

Danymi referencyjnymi były wyniki uzyskane na serwerowej platformie sprzętowej (patrz podrozdział 3.4). Poniższy wykres prezentuje czas wykonania dwuwymiarowej funkcji Rosenbrocka w algorytmie iteracyjnym oraz zrównoleglonym dla dwóch procesów.



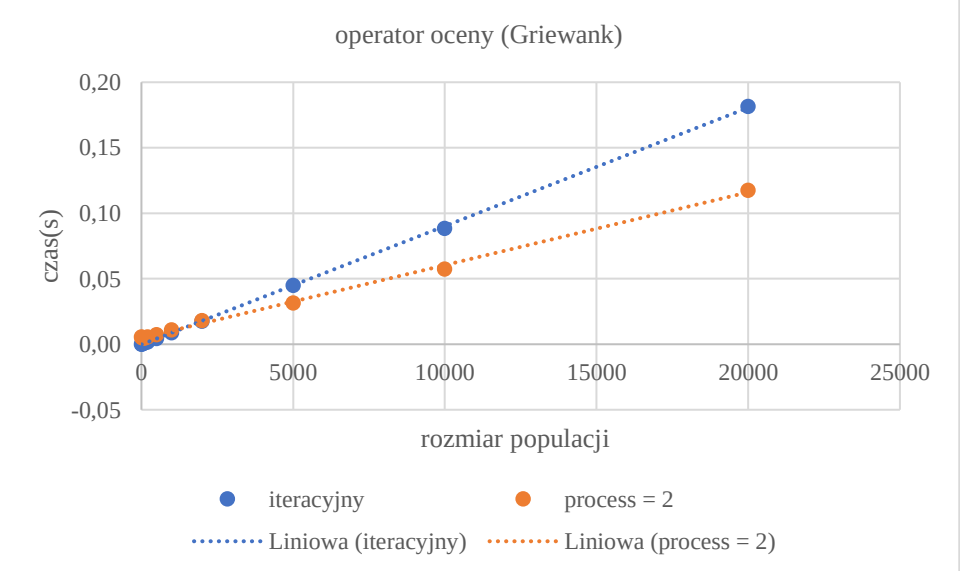
Rysunek 23 Czas wykonania funkcji Rosenbrocka w zależności od wielkości populacji. Porównanie czasu wykonania dwuwymiarowej funkcji testowej dla algorytmu iteracyjnego oraz zrównoleglonego za pomocą modułu multiprocessing, z wykorzystaniem dwóch procesów. Platforma sprzętowa Server VPS.

Na powyższym wykresie zauważalne jest, iż dla populacji wielkości pięciu tysięcy osobników uzyskany czas pomiaru jest lepszy dla algorytmu uruchomionego na dwóch procesach niż dla algorytmu referencyjnego. Następnie wykonano dokładnie ten sam test, tym razem na platformie sprzętowej HP EliteBook 840 G3.

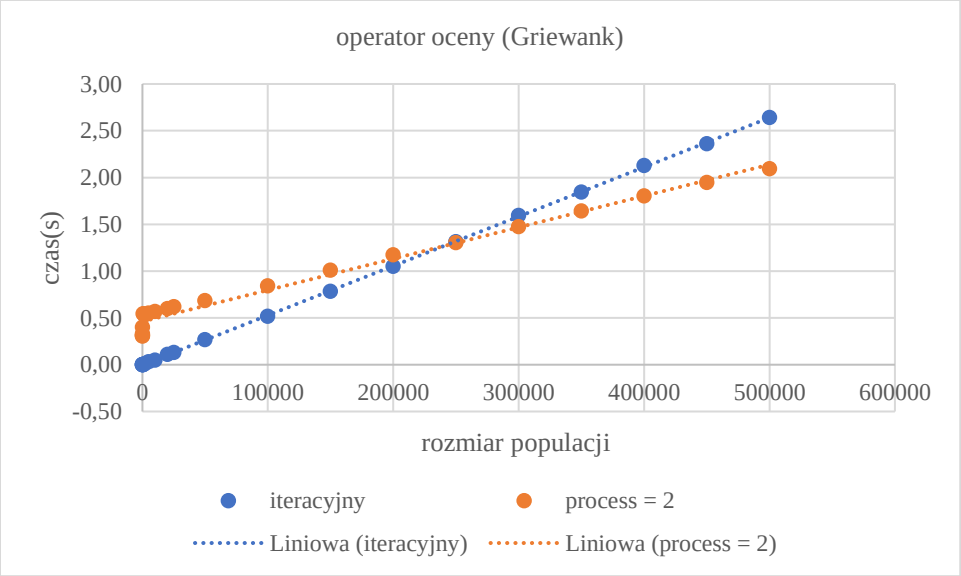


Rysunek 24 Czas wykonania funkcji Rosenbrocka w zależności od wielkości populacji. Porównanie czasu wykonania dwuwymiarowej funkcji testowej dla algorytmu iteracyjnego oraz zrównoleglonego za pomocą modułu multiprocessing, z wykorzystaniem dwóch procesów. Platforma sprzętowa HP EliteBook 840 G3.

Na platformie sprzętowej HP EliteBook 840 G3 dwa procesy przeliczyły funkcję oceny szybciej dla populacji wielkości czterystu tysięcy. Pomimo, iż algorytm iteracyjny wykonał się szybciej na tej platformie testowej, uruchomiony na dwóch procesach uzyskał gorsze wyniki, przez co zasadność wykorzystania algorytmu równoległego maleje. Poniżej prezentowane są wyniki identycznych testów, tym razem dla funkcji testowej Griewanka.



Rysunek 25 Czas wykonania funkcji Griewanka w zależności od wielkości populacji. Porównanie czasu wykonania dwuwymiarowej funkcji testowej dla algorytmu iteracyjnego oraz zrównoleglonego za pomocą modułu multiprocessing, z wykorzystaniem dwóch procesów. Platforma sprzętowa Server VPS.



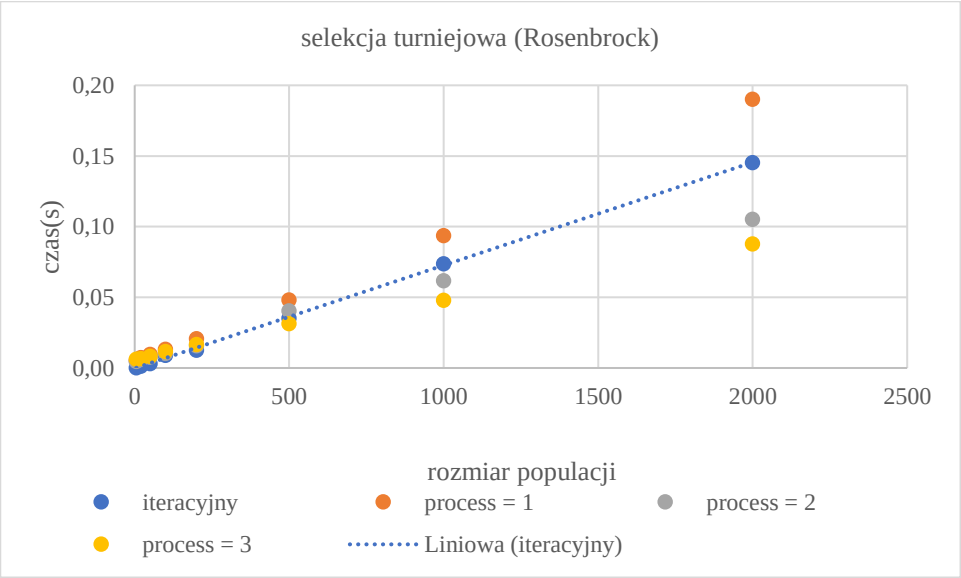
Rysunek 26 Czas wykonania funkcji Griewanka w zależności od wielkości populacji. Porównanie czasu wykonania dwuwymiarowej funkcji testowej dla algorytmu iteracyjnego oraz zrównoleglonego za pomocą modułu multiprocessing, z wykorzystaniem dwóch procesów. Platforma sprzętowa HP EliteBook 840 G3.

W przypadku funkcji Griewanka, pomimo uzyskania lepszych czasów wykonania zarówno dla algorytmu iteracyjnego jak i równoległego, punkt przecięcia przesunął się również w prawo, zmniejszając zasadność stosowania algorytmu w wersji zrównoleglonej.

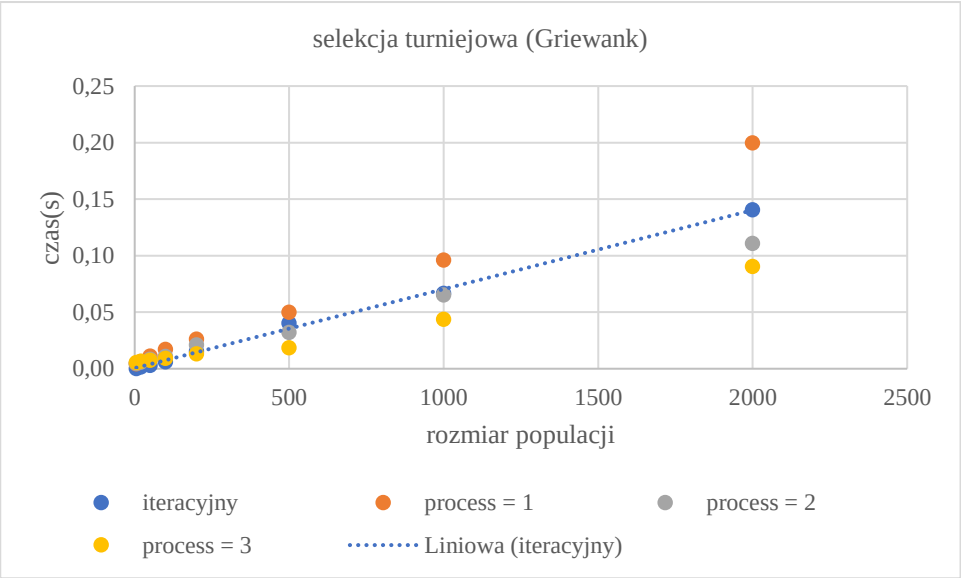
Przeprowadzone testy pokazały, iż bardzo dużo zależy również od platformy sprzętowej. Im lepszy czas wykonania algorytmu iteracyjnego, tym trudniej algorytmowi w zrównoleglonej wersji przezwyciężyć narzut uruchomienia na wielu procesach.

Kolejne etapy algorytmu genetycznego były testowane tylko przy użyciu modułu ***multiprocessing***, który pozwolił na ominięcie ***GILa***. Kolejnym założeniem było posiadanie przez osobniki dokładnie stu cech genetycznych, tak aby umożliwić przeprowadzenie testów na jak najmniejszych populacjach. Ostatnim założeniem było użycie od jednego do trzech procesów przy porównywaniu czasów wykonania.

3.5.3 Selekcja



Rysunek 27 Czas wykonania operatora selekcji w zależności od wielkości populacji. Porównanie czasu wykonania stuwymiarowej funkcji testowej dla algorytmu iteracyjnego oraz zrównoleglonego za pomocą modułu multiprocessing, z wykorzystaniem jednego, dwóch oraz trzech procesów.

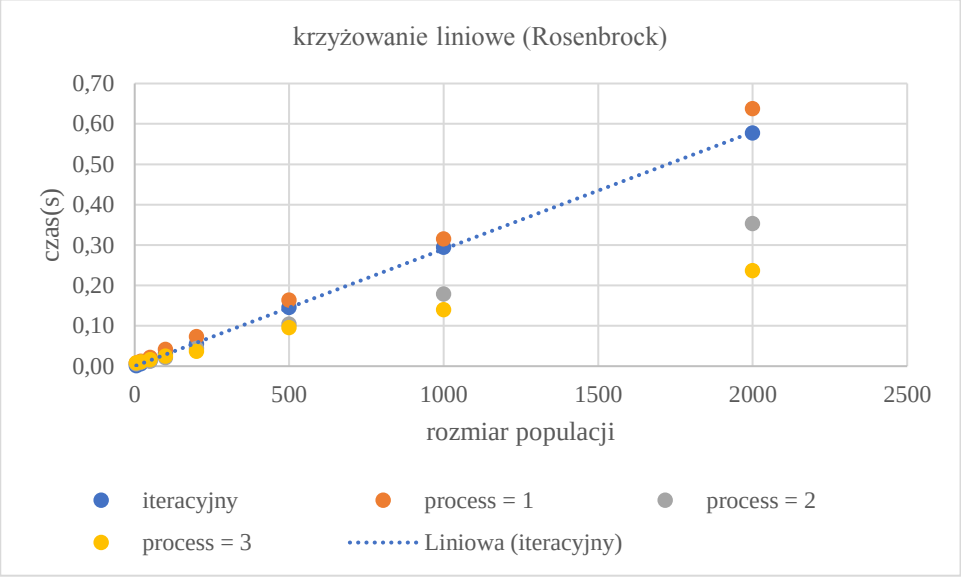


Rysunek 28 Czas wykonania operatora selekcji w zależności od wielkości populacji. Porównanie czasu wykonania stuwymiarowej funkcji testowej dla algorytmu iteracyjnego oraz zrównoleglonego za pomocą modułu multiprocessing, z wykorzystaniem jednego, dwóch oraz trzech procesów.

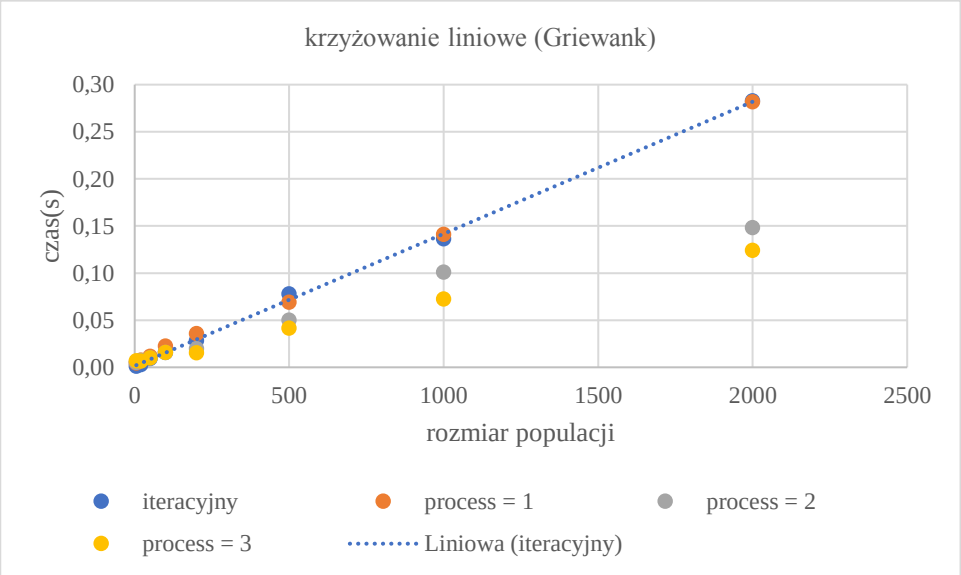
Charakterystyka wykresów dla testów operatorów selekcji jest podobna jak w przypadku operatorów oceny. Zauważalny jest narzut, jaki generują kolejno dodawane procesy. W przypadku funkcji Rosenbrocka, opłacalność stosowania równoległej implementacji pojawia się w przypadku selekcjonowania populacji o wielkości pięciuset osobników. W przypadku funkcji Griewanka, opłacalność zauważyć można dla populacji liczącej dwustu osobników. Należy jednak pamiętać, iż przy tak krótkich czasach

wykonania operatorów, nawet uśrednione wyniki z dziesięciu testów mogą nieco odbiegać od realnych wyników. Niemniej jednak, z przeprowadzonych testów można wyciągnąć wnioski bliźniaczo podobne do tych uzyskanych przy poprzednim etapie algorytmu. Równoległa implementacja prostego algorytmu genetycznego staje się opłacalna dopiero przy odpowiednio trudnych problemach oraz przy odpowiednio dużej grupie osobników.

3.5.4 Krzyżowanie



Rysunek 29 Czas wykonania operatora krzyżowania w zależności od wielkości populacji. Porównanie czasu wykonania stuwymiarowej funkcji testowej dla algorytmu iteracyjnego oraz zrównoleglonego za pomocą modułu multiprocessing, z wykorzystaniem jednego, dwóch oraz trzech procesów.

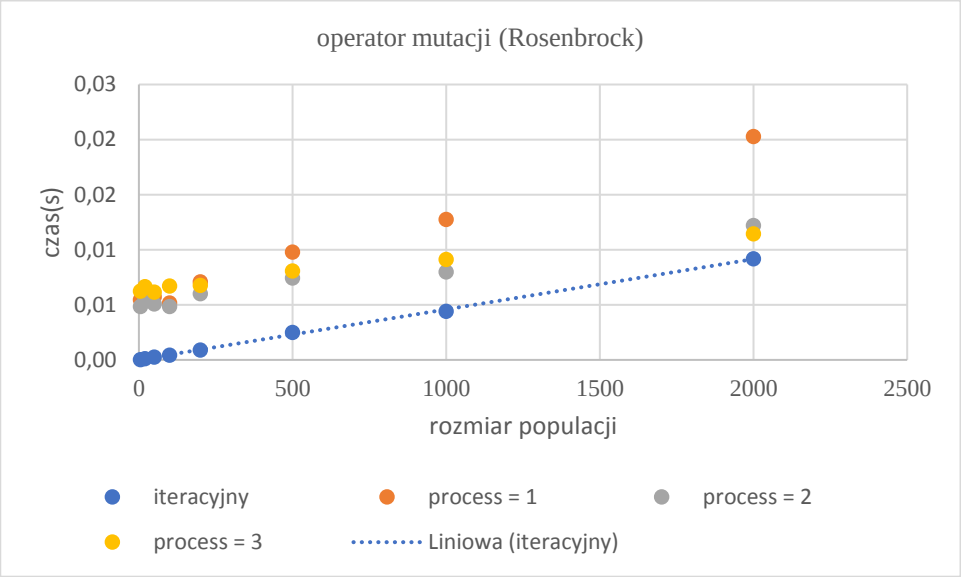


Rysunek 30 Czas wykonania operatora krzyżowania w zależności od wielkości populacji. Porównanie czasu wykonania stuwymiarowej funkcji testowej dla algorytmu iteracyjnego oraz zrównoleglonego za pomocą modułu multiprocessing, z wykorzystaniem jednego, dwóch oraz trzech procesów.

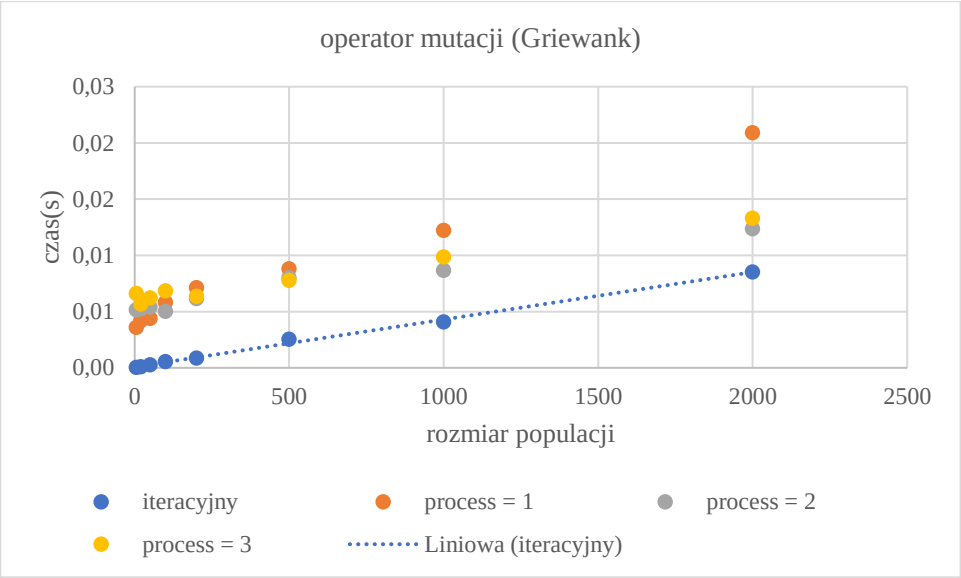
Jak łatwo zauważyć, operator krzyżowania zachowuje podobną charakterystykę jak poprzednie dwa operatory. Najlepszy wynik uzyskano uruchamiając trzy procesy, ale dopiero dla dwustu osobników. Przypuszczalnie zwiększanie ilości genów osobników lub zmiana problemu optymalizacyjnego, powinno dać lepsze wyniki dla jeszcze mniejszych populacji.

3.5.5 Mutacja

Ostatnim etapem prostego algorytmu genetycznego jest mutacja osobnika. Czasy wykonania operatora mutacji widoczne są na Rysunkach 31-32.



Rysunek 31 Czas wykonania operatora mutacji w zależności od wielkości populacji. Porównanie czasu wykonania stuwymiarowej funkcji testowej dla algorytmu iteracyjnego oraz zrównoleglonego za pomocą modułu multiprocessing, z wykorzystaniem jednego, dwóch oraz trzech procesów.

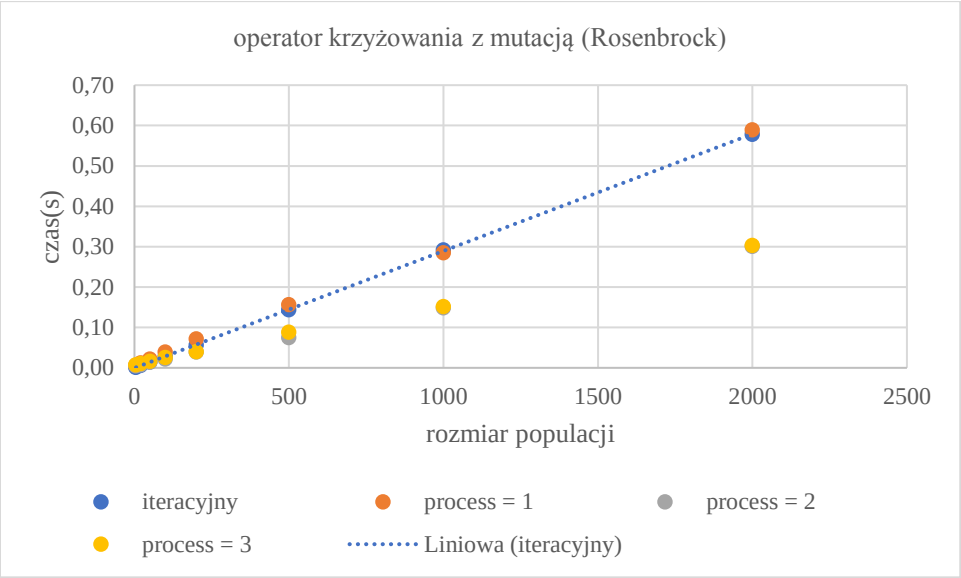


Rysunek 32 Czas wykonania operatora mutacji w zależności od wielkości populacji. Porównanie czasu wykonania stuwymiarowej funkcji testowej dla algorytmu iteracyjnego oraz zrównoleglonego za pomocą modułu multiprocessing, z wykorzystaniem jednego, dwóch oraz trzech procesów.

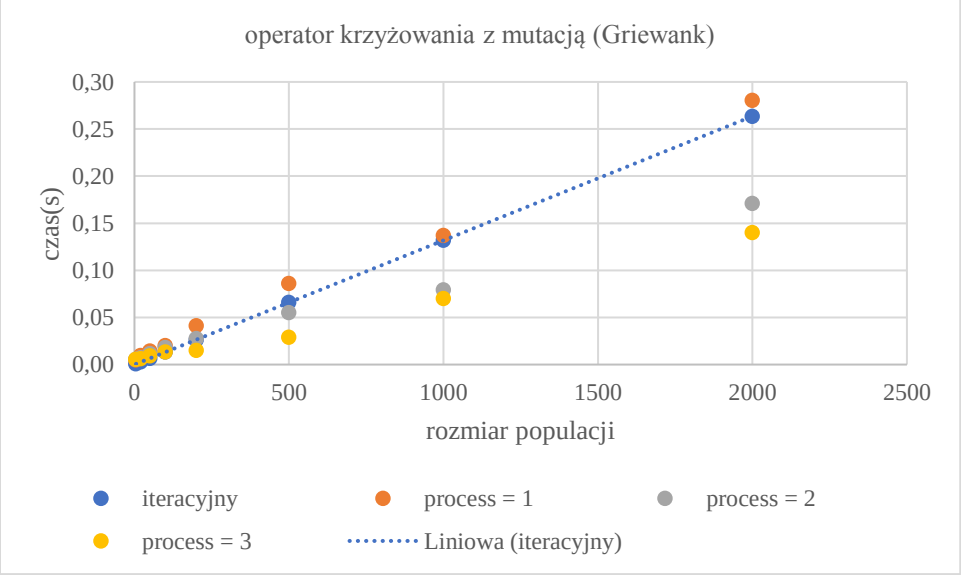
W przeprowadzonych testach operatorów mutacji nie osiągnięto satysfakcjonującego wyniku, chociaż charakterystyka wykresów jest podobna, jak chociażby przy bliźniaczo podobnym operatorze oceny. Z racji probabilistycznej charakterystyki samego operatora, mutacja nie odbywa się na każdym osobniku, a jedynie na losowej próbie. Powoduje to zmniejszenie opłacalności użycia równoległej implementacji tego etapu algorytmu genetycznego.

3.5.6 Krzyżowanie z mutacją

Ciekawym rozwiązaniem może być również włączenie operatora mutacji do etapu krzyżowania. Nie zwiększyłoby to znacząco czasu wykonania operatora krzyżowania, natomiast zapobiegłoby to niepotrzebnemu tworzeniu dodatkowego narzutu spowodowanego uruchamianiem kolejnych procesów, które w przypadku operatora mutacji nie są w pełni wykorzystywane. W tym celu w module ***algorithm*** zaimplementowano dodatkową funkcję, ***linear_crossing_with_mutation***, która łączy obie te funkcjonalności. Wyniki wykonanych pomiarów prezentują Rysunki 33 i 34.



Rysunek 33 Czas wykonania operatora krzyżowania z mutacją w zależności od wielkości populacji. Porównanie czasu wykonania stuwymiarowej funkcji Rosenbrocka dla algorytmu iteracyjnego oraz zrównoleglonego za pomocą modułu multiprocessing, z wykorzystaniem jednego, dwóch oraz trzech procesów. Punkty dla dwóch i trzech procesów biegną niemal współliniowo.



Rysunek 34 Czas wykonania operatora krzyżowania z mutacją w zależności od wielkości populacji. Porównanie czasu wykonania stuwymiarowej funkcji Griewanka dla algorytmu iteracyjnego oraz zrównoleglonego za pomocą modułu multiprocessing, z wykorzystaniem jednego, dwóch oraz trzech procesów.

Porównując wykresy dla operatorów krzyżowania oraz operatorów krzyżowania z mutacją, można zauważyć, iż czasy wykonania są niemal identyczne. Potwierdza to założenie, iż mutacja nie wpływa zauważalnie na czas wykonania operacji krzyżowania. Wydaje się, iż implementacja operatora mutacji łącznie z operatorem krzyżowania, pozwoli na uzyskanie lepszych czasów końcowych algorytmu, zwłaszcza w zrównoleglonej wersji algorytmu genetycznego. Nieuruchamianie procesów dla rzadko

występującej operacji mutacji pozwala na zmniejszenie czasu wykonania oraz ograniczenie narzutu spowodowanego przez uruchomienie nowych procesów.

4. Narzędzia i technologie użyte w pracy

4.1 Język Python

4.1.1 Instalacja środowiska

Aby zainstalować środowisko Python należy w pierwszej kolejności skorzystać z sekcji **Downloads** na głównej stronie projektu (https://www.python.org/). Strona powinna automatycznie wykryć system operacyjny i zaproponować instalator w odpowiedniej wersji. Po pobraniu należy zainstalować środowisko Python zgodnie z poleceniami widocznymi w instalatorze. Należy się jednak upewnić, iż wraz ze środowiskiem instalujemy **pip** oraz dodajemy je do **zmiennych środowiskowych** systemu.

4.1.2 Pip

Jest to swojego rodzaju system narzędzi pozwalający na automatyczną instalację, aktualizację czy usuwanie poszczególnych modułów czy bibliotek. Dzięki prostocie użycia, jest to bardzo potężne narzędzie pozwalające na konfigurację lokalnej instalacji Pythona. Wszystkie moduły zewnętrzne zostały zainstalowane przy użyciu tego narzędzia. Opis instalacji poszczególnych modułów można znaleźć w podrozdziale 4.2.

4.1.3 Global Interpreter Lock (GIL)

Jest to mutex podtrzymywany przez wątek interpretera języka, który uniemożliwia współbieżne wykonywanie kodu. Każdy proces interpretera posiada dokładnie jedną globalną blokadę ów interpretera. Według dokumentacji użycie GIL zmniejsza konkurencję pomiędzy wątkami uruchomionymi na tym samym procesie interpretera, co uniemożliwia natomiast uzyskanie większej wydajności z racji posiadania kilku rdzeni. Co ciekawe, GIL jest obecny tylko w „klasycznej” implementacji języka Python tzw. ***CPython***. Istnieją alternatywne implementacje środowiska, jak ***Jython*** (napisany w Javie) czy ***IronPython*** (środowisko zaimplementowane na platformie .NET), w których GIL nie występuje. Informacja o GIL wpłynęła znacząco na próbę implementacji równoległego algorytmu genetycznego, której opis można znaleźć we wcześniejszej części pracy.

4.2 Moduły użyte w implementacji

4.2.1 Numpy

Jest to biblioteka dodająca wsparcie dla przetwarzania dużych, wielowymiarowych tablic i macierzy, wraz z wieloma wbudowanymi funkcjami mogącymi na nich operować. **NumPy** to tzw. projekt **Open Source**, więc bez przeszkód można go używać we własnym kodzie, a nawet dołączyć do osób odpowiedzialnych za rozwijanie kodu.
Aby zainstalować **NumPy**, należy upewnić się, iż mamy zainstalowaną najnowszą wersję pip. Następnie wystarczy uruchomić w wierszu poleceń poniższą komendę tekstową:

pip install numpy

4.2.2 Threading

Moduł ten dostarcza, zgodnie z oficjalną dokumentacją, wysoko poziomowy interfejs nad modułem ***thread***. Moduł ten zapewnia niskopoziomowe prymitywy, dzięki którym możliwa jest praca z wieloma wątkami. W **CPython**, w wyniku działania GILa, tylko jeden wątek może być użyty na raz w danym momencie. W tej sytuacji polecany jest moduł multiprocessing, natomiast threading dalej może być dobrym wyborem, jeżeli potrzeba jest uruchomienia obliczeń symultanicznie. Moduł threading jest dostarczony automatycznie wraz z biblioteką standardową języka, zatem nie jest konieczna instalacja modułu z użyciem narzędzia pip.

4.2.3 Multiprocessing

Ten moduł wspiera uruchamianie procesów w podobny sposób, w jaki jest to możliwe w module threading. Dzięki użyciu tego modułu możliwe jest ominięcie GILa poprzez używanie podprocesów zamiast wątków. Takie podejście zapewnia programiście możliwość skorzystania ze wszystkich rdzeni procesora. Moduł może być użyty zarówno na systemie Windows oraz Linux. Podobnie jak moduł threading, moduł multiprocessing jest dostarczony razem z biblioteką standardową języka Python, przez co nie jest konieczna manualna instalacja.

5. Podsumowanie

Prosty algorytm genetyczny w klasycznej postaci jest bardzo efektywnym sposobem na optymalizację zadanych zagadnień. O ile podstawowa wersja algorytmu jest bardzo prosta w implementacji i daje się łatwo przeanalizować, o tyle analiza zrównoleglonej wersji algorytmu nie jest trywialna. Wybrane środowisko pozwala w prosty sposób zaprogramować obie implementacje, niestety posiada swoje ograniczenia, które znacząco wpływają na końcowy rezultat.

Pierwszym napotkanym problemem była blokada spowodowana przez mutex ***GIL***. Z racji wyboru klasycznej implementacji języka Python, nieopłacalne stało się użycie modułu ***threading***, ponieważ uniemożliwia on wykorzystanie wielu rdzeni do obliczeń. Z pomocą przyszedł natomiast bliźniaczo podobny moduł ***multiprocessing***. Z jego pomocą udało się efektywnie zaimplementować równoległy algorytm genetyczny.

Z racji ograniczeń języka Python czy dodatkowego narzutu obliczeniowego spowodowanego uruchomieniem procesów, przeprowadzone testy pokazały, iż opłacalność stosowania algorytmu genetycznego zauważalna była dla obliczeń, w których narzut spowodowany uruchomieniem procesu nie stanowił większości czasu wykonania. Na podstawie przeprowadzonych badań można ekstrapolować problem, iż zwiększając odpowiednio trudność zagadnienia, można zaobserwować korzyści płynące z równoległej implementacji.

Użyty w pracy język Python jest bardzo prostym narzędziem, pozwalającym na szybką i efektywną implementację algorytmów. Kolejnym krokiem mogłoby być przetestowanie obu rozwiązań w środowisku Python, ale zaimplementowanym w innej technologii, w której nie ma ograniczeń związanych z mutexem GIL, np. z użyciem ***IronPython***, czyli implementacji języka Python na platformie **.Net**.

Niezależnie od wybranych technologii, zrównoleglanie obliczeń za pomocą wątków czy procesów staje się coraz bardziej popularne oraz opłacalne. Dzięki zastosowaniu nowoczesnych technologii informatycznych możliwe staje się dokonywanie obliczeń dla problemów o wysokim stopniu skomplikowania, gdzie klasyczne podejście do zagadnień jest nieefektywne.

Warto wspomnieć o pracy magisterskiej [11], w której dokonano analizy porównawczej działania algorytmu genetycznego iteracyjnego oraz równoległego, ale zaimplementowanych przy użyciu języka C++ oraz biblioteki CUDA firmy

NVIDIA. Z opisu wynika, iż sporo czasu poświęcono na komunikację międzyprocesową, próbując w optymalny sposób przekazywać wymagane dane. Implementacja z użyciem języka Python nie wymagała dodatkowych czynności, stąd wydaje się, iż implementacja z językiem Python jest dużo prostsza. Zauważono jednak, iż zrównoleglanie algorytmu staje się opłacalne dopiero od momentu, w którym narzut spowodowany uruchomieniem wątków czy procesów przestaje mieć zauważalny wpływ na czasy wykonania operatorów.

6. Bibliografia

[1] Holland J.H., Adaptation in Natural and Artificial Systems, 1975 The University of Michigan

[2] Distributed Evolutionary Algorithms in Python, https://github.com/DEAP/deap

[3] Fortin, Félix-Antoine; F.-M. De Rainville; M-A. Gardner; C. Gagné; M. Parizeau (2012). "DEAP: Evolutionary Algorithms Made Easy". Journal of Machine Learning Research. 13: 2171–2175.

[4] De Rainville, François-Michel; F.-A Fortin; M-A. Gardner; C. Gagné; M. Parizeau (2012). "DEAP: A Python Framework for Evolutionary Algorithms", In Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation, Philadelphia, PA.

[5] Scalable Concurrent Operations in Python, https://github.com/soravux/scoop/

[6] Gwiazda T.D., Algorytmy Genetyczne Kompendium Tom 1, 2007 Wydawnictwo Naukowe PWN

[7] The official home of the Python Programming Language, https://www.python.org/

[8] Melanie Mitchel, An Introduction to Genetic Algorithms, 1999 Massachusetts Institute of Technology

[9] Gwizdałła T.M., Własności algorytmów genetycznych do zagadnień magnetyzmu, 2010 Wydawnictwo Uniwersytetu Łódzkiego

[10] Forbes E., Learning Concurrency in Python, 2017 Packt Publishing Ltd.

[11] Kacprzak M, Implementacja i analiza działania równoległego algorytmu genetycznego, 2016 Uniwersytet Łódzki