AKADEMIA GÓRNICZO-HUTNICZA

IM. STANISŁAWA STASZICA W KRAKOWIE



Wydział Inżynierii Metali i Informatyki Przemysłowej



PRACA DYPLOMOWA INŻYNIERSKA

pt.

„**Implementacja procedur optymalizacji przystosowanych   
do obliczeń w środowiskach masowo równoległych**”

Imię i nazwisko dyplomanta: **Jakub Karamański**

Kierunek studiów: **Informatyka Stosowana**

Nr albumu: **268958**

Promotor: dr inż Łukasz Rauch

Recenzent: dr inż. Piotr Kustra

Podpis dyplomanta: Podpis promotora:

Kraków 2017

„Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia

4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90,

poz. 631 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do

autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega

grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej

karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy

utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo

publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub

nadanie.”, a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211

ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r.

poz. 572, z późn. zm.) „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny

uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed

komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym

dalej "sądem koleżeńskim"”, oświadczam, że niniejszą pracę dyplomową

wykonałem(-am) osobiście i samodzielnie i że nie korzystałem(-am) ze źródeł innych niż

wymienione w pracy.”

Kraków, dnia …… Podpis dyplomanta…………….

**Spis treści**

[1. Wstęp 1](#_Toc471108408)

[2. Opis wybranych algorytmów 2](#_Toc471108409)

[**2.1.** **Metoda Hook’a-Jeevesa** 2](#_Toc471108410)

[**2.2.** **Metoda Nelder’a-Mead’a** 3](#_Toc471108411)

[3. Projekt oprogramowania 7](#_Toc471108412)

[**3.1.** **Projekt ..** 7](#_Toc471108413)

[**3.2.** **Architektura oprogramowania** 7](#_Toc471108414)

[**3.3.** **Przegląd metod** 8](#_Toc471108415)

[4. Realizacja oprogramowania 9](#_Toc471108416)

[**4.1.** **Idea zrównoleglenia wybranych metod** 9](#_Toc471108417)

[**4.1.1.** **Zrównoleglenie metody Hook’a-Jeeves’a** 9](#_Toc471108418)

[**4.1.2.** **Zrównoleglenie metody Nelder’a–Mead’a** 11](#_Toc471108419)

[**4.2.** **Szczegóły implementacji** 12](#_Toc471108420)

[**4.2.1.** **Implementacja metody Hook’a–Jeeves’a** 13](#_Toc471108421)

[**4.2.2.** **Implementacja metody Nelder’a–Mead’a** 15](#_Toc471108422)

[5. Wyniki 17](#_Toc471108423)

[**5.1.** **Specyfikacja sprzętu** 17](#_Toc471108424)

[**5.2.** **Funkcje Testowe** 17](#_Toc471108425)

[**5.2.1.** **Funkcja testowa Rastrigin** 17](#_Toc471108426)

[**5.2.2.** **Funkcja testowa Ackley’s** 19](#_Toc471108427)

[**5.2.3.** **Funkcja testowa Baley’s** 21](#_Toc471108428)

[**5.2.4.** **Funkcja testowa Booth’s** 22](#_Toc471108429)

[**5.3.** **Podsumowanie** 24](#_Toc471108430)

[**5.4.** **Problem praktyczny** 25](#_Toc471108431)

[6. Wnioski 26](#_Toc471108432)

[7. Literatura 27](#_Toc471108433)

# Wstęp

Wraz z wzrostem techniki pojawiło się zapotrzebowanie na środowiska,  
 które oferowały bardzo dużą moc obliczeniową, którą można wykorzystać przy analizie danych czy obliczeniu skomplikowanych równań. Środowiska masowo równoległe są tworzone i wykorzystywane m.in. do obliczeń wielkoskalowych, symulacji numerycznych za pomocą złożonych modeli nieliniowych czy też wykonywania obliczeń wielo—iteracyjnych.

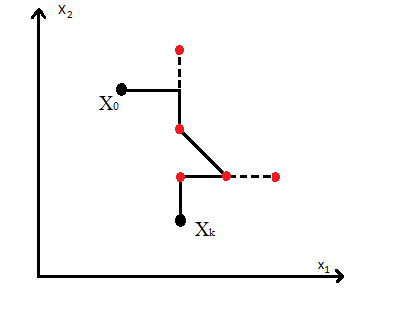
W tej pracy podjęto problem adaptacji procedur optymalizacji, które w swojej naturze nie są równoległe, czyli metod, które w pojedynczym kroku wykonują obliczenia tylko dla jednego punktu. Przystosowanie algorytmów do pracy z serwerami obliczeniowymi przyniosłoby znaczny wzrost wydajności. Algorytmy takie jak genetyczny czy rój cząstek są w naturze równoległe więc łatwo przystosować je do obliczeń   
na maszynach wielordzeniowych lecz w tej pracy zostały przedstawione takie, które są trudne do zrównoleglenia ale przy zastosowaniu odpowiednich modyfikacji mogą okazać się bardzo skuteczne, co zostało również zbadane w tej pracy.

W drugim rozdziale zostały opisane wybrane algorytmy, które zostały zrównoleglone. Trzeci rozdział to opis realizacji oprogramowania oraz została przedstawiona idea zrównoleglania algorytmów. Został także zawarty projekt oprogramowania oraz szczegóły implementacji. W rozdziale czwartym wykonano również analizę zmodyfikowanych algorytmów z wykorzystaniem kilku standardowych problemów (funkcji) testowych. Dodatkowo wykonano testy optymalizacji z wykorzystaniem środowisk masowo równoległych, gdzie problemem praktycznym było projektowanie procesu produkcyjnego. Porównanie czasu działania i dokładności obliczeń zostały zawarte w podsumowaniu

# Opis wybranych algorytmów

## **Metoda Hook’a-Jeevesa**

Algorytm Hook’a-Jeeves’a zalicza się do iteracyjnych metod optymalizacji. Składa się z sekwencji ruchów z punktu bazowego – krok próbny, a następnie z ruchów – krok roboczy, które zapewniają następny punkt bazowy do zbadania. W kroku próbnym wykonuje lokalne wyszukiwanie w jednym kierunku poprzez zmianę parametru lub parametrów punktu bazowego. Jeśli wartość funkcji w tym punkcie jest lepsza niż wartość poprzednia to algorytm wybiera ten nowy punkt jako punkt bazowy. W przeciwnym razie algorytm wykonuje wyszukiwanie w przeciwnym kierunku i jeśli wynik jest lepszy od poprzedniego to nowy punkt zostaje punktem bazowym. Algorytm cały czas przechowuje wartość początkową i sprawdza czy wartość jest lepsza czy gorsza. Gdy wszystkie parametry zostały zbadane algorytm przechodzi do kroku roboczego. W kroku roboczym każdy z parametrów zostaje zwiększony o stałą. Celem jest przesunięcie punktu bazowego w kierunku, który poprawiłby wynik z kroku próbnego. Następnie funkcja zostaje oceniana w nowym miejscu. Jeśli wartość jest lepsza to nowy punkt staje się punktem bazowym w przeciwnym razie przemieszczenie jest ignorowane, a stała zostaje zredukowana, żeby zmniejszyć przemieszczenie układu [1]. Kroki te są powtarzane aż do momentu, kiedy nie zostanie spełniony warunek stopu



**Rys. 2.** Graficzne przedstawienie działania metody Hook’a – Jeevesa

Źródło : własne opracowanie

Na rysunku 2 można zobaczyć graficzne przedstawienie działania metody Hook’a-Jeevesa. Punkt początkowy znajduje się w *xo*, z którego czarną linią zostały zaznaczone kroki pomyślne czyli takie, które dały mniejszą wartość funkcji, a linie przerywane wraz   
z zaznaczonymi na czerwono punktami to kroki próbne, gdzie wartość funkcji w tych punktach była większa, te kroki uznajemy za niepomyślne. Kroki próbne sprawdzane są w dwóch kierunkach: góra i dół, aby uznać, który będzie najlepszy. Etap roboczy wykonywany jest dopiero wtedy, gdy przynajmniej jeden z kroków etapu próbnego zakończył się sukcesem, czyli osiągnięcia wartości funkcji mniejszej niż w punkcie bazowym. Skoki próbne oraz skoki robocze zawsze wykonywane są, aż do momentu znalezienia najmniejszej wartości funkcji lub w przypadku jej nie znalezienia do określonego warunku stopu.

## **Metoda Nelder’a-Mead’a**

Algorytm simpleksów Neldera-Meada (ang. Downhill Simplex, Amoeba), jest metodą wykorzystywaną do wyznaczenia ekstremum funkcji . Główną zaletą algorytmu Neldera-Meada jest bardzo niska złożoność obliczeniowa w przypadku niewielu rozbudowanych funkcji celu. Algorytm polega na utworzeniu w przestrzeni En+1 *n*-wymiarowego simpleks o *n*+1 wierzchołkach. Wykonuje się to po to, aby można było go wpisać w powierzchnię reprezentująca badaną funkcję celu. Pierwszym krokiem, jaki należy wykonać, to wyliczanie punktów wierzchołkowych simpleksu *Qj*, przy czym zakładana jest pewną odległość pomiędzy wierzchołkami – tak zwany krok. W kolejnych iteracjach dokonuje sie wymiany „najgorszego” wierzchołka i zostaje budowany nowy simpleks, aż odległość pomiędzy jego wierzchołkami w pobliżu poszukiwanego ekstremum funkcji będzie mniejsza od założonej dokładności obliczeń. Kryterium zbieżności to dokładność obliczeń w tej metodzie [2]. Podczas kolejnych iteracji wykonujemy operacje, które mają na celu osiągnięcie najniższej wartości funkcji celu:

1. Odbicie punktu *Qh*względem *Q’*,

Początkowo należy wybrać najgorszy punkt (punkt, w którym funkcja ma największą wartość) i odbić go względem punktu *Q’*, który wyznaczamy z wzoru:

|  |  |  |
| --- | --- | --- |
|  |  | (1) |

Następnie zostaje obliczona odległość pomiędzy punktami *Q’*, a punktem *Qh*:

|  |  |  |
| --- | --- | --- |
|  |  | (2) |

Ostatnim etapem jest obliczenie punktu *Q\**

|  |  |  |
| --- | --- | --- |
|  |  | (3) |

gdzie: *α* – współczynnik odbicia (*α* > 0)

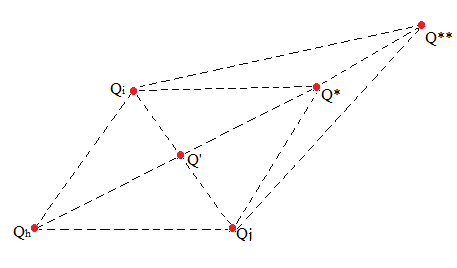
|  |  |
| --- | --- |
|  |  |
| **Rys. 3.** Graficzne przedstawienie odbicia punktu  gdzie współczynniki odbicia α = 1  Źródło : własne opracowanie | **Rys. 4.** Graficzne przedstawienie odbicia punktu  gdzie współczynnik odbicia α = 0.5  Źródło : własne opracowanie |

Na rysunku 3 i 4 zostało graficznie pokazane odbicie punktu *Qh* względem punktu *Q’*.

1. Ekspansja punktu *Q\*\** względem *Q’*,

Jeśli poprzedni krok został zakończony pomyślnie czyli wartość funkcji w punkcie *Q\** jest najmniejsza wykonujemy ekspansję czyli przejście o zadaną odległość   
w tym samym kierunku co w poprzednim kroku:

|  |  |  |
| --- | --- | --- |
|  |  | (4) |



**Rys. 5.** Graficzne przedstawienie ekspansji punktu

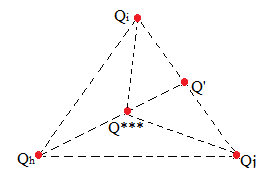
Źródło : własne opracowanie

Gdzie:  
*γ* – współczynnik ekspansji (*γ* > 0)

1. Kontrakcja punktu *Qh* względem *Q’*,

Jeśli odbicie nie zostało wykonane wykonujemy kontrakcję – nowy punkt otrzymamy wykonując odbicie do środka simpleksu względem punktu *Q’*.

|  |  |  |
| --- | --- | --- |
|  |  | (5) |

****

**Rys. 6.** Graficzne przedstawienie kontrakcji punktu

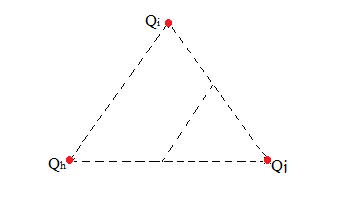
Źródło : własne opracowanie

Gdzie:  
*β* – współczynnik kontrakcji (0 < *β* < 1)

1. Redukcja simpleksu,

Redukcja simpleksu została wprowadzona po to aby zapobiec upadkom algorytmu, obliczaniu złych wartości oraz poprawienie dokładności obliczeń.

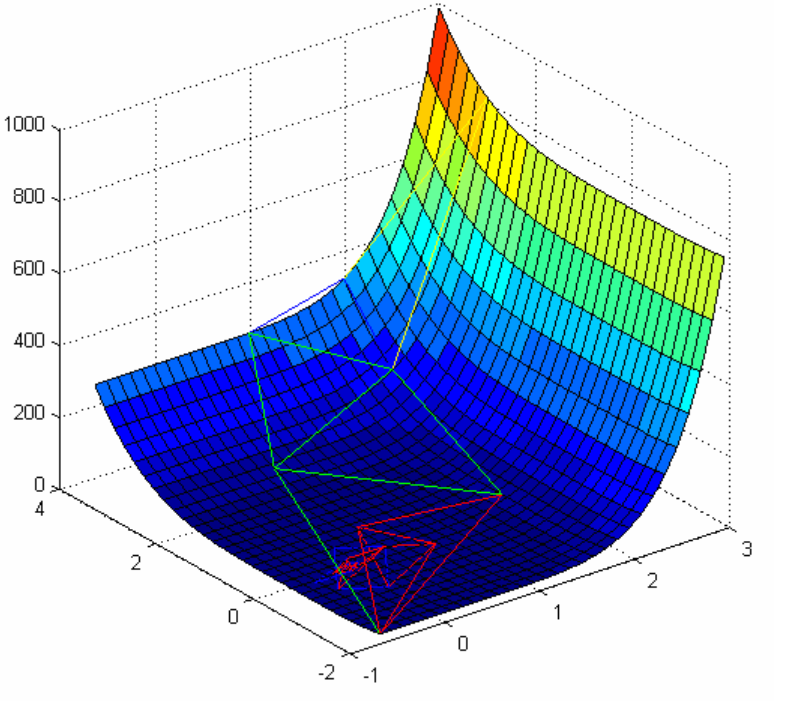
|  |  |  |
| --- | --- | --- |
|  |  | (6)  (7) |



**Rys. 7.** Graficzne przedstawienie redukcji simplexu

Źródło : własne opracowanie

W algorytmie Neldera-Meada dobiera się wartości współczynników w następujący sposób: *α* = 1, *β* = 0.5, *γ* = 2. Na rysunku 8 można zobaczyć obraz zminimalizowanej funkcji 2D:



**Rys. 8.** Graficzne przedstawienie działania metody Nelder’a-Mead’a  
dla funkcji dwuwymiarowej

Źródło : [3]

Podsumowując w metodzie bezgradientowej, jaką jest algorytm Neldera-Meada   
na dokładność rezultatów oraz szybkość wpływają:

1. skomplikowanie funkcji celu,
2. tolerancja algorytmu, która gwarantuje zakończenie obliczeń.

Więcej informacji na temat tego algorytmu znajduje się w [4].

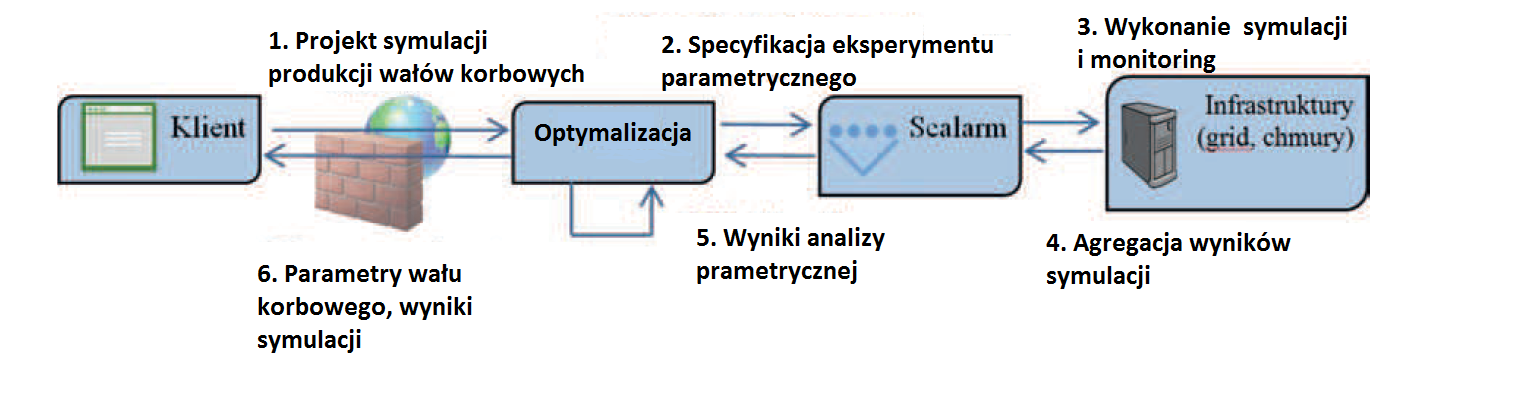
# Projekt oprogramowania

## **Projekt ..**

Opisac jak wygląda biblioteka, klasy.

## **Architektura oprogramowania**

Środowiska masowo równoległe powstają coraz częściej – skomplikowane obliczenia potrzebują coraz większej mocy obliczeniowej. Polska infrastruktura obliczeniowa ( PL – Grid) umożliwia przeprowadzenie badań naukowych, które opierają się na symulacjach i obliczeniach wykorzystując klastry komputerowe. Jednym z klastrów jest superkomputer Zeus, który znajduje się w Akademickim Centrum Komputerowym Cyfronet Akademii Górniczo Hutniczej w Krakowie. W niniejszej pracy wykorzystano moc obliczeniową Zeusa przy pomocy platformy Scalarm. Scalarm to platforma wspierająca wykonywanie eksperymentów „data farming” lub badań parametrycznych – badań wymagających uruchomienia tego samego kodu z różnymi wartościami parametrów wejściowych.

  
**Rys. 1.** Architektura oprogramowania

Źródło : własne opracowanie

Na rysunku 1 widzimy architekturę oprogramowania, jako klient tworzymy projekt, wiele projektów łączymy w bibliotekę, którą później będziemy wykorzystywać do obliczeń. Następnie określamy specyfikację eksperymentu parametrycznego czyli z jakimi wartościami wejściowymi eksperyment ma zostać wykonany. Te specyfikacje wysyłamy do platformy Scalarm, która ułatwia wykonywanie tego typu eksperymentów poprzez wsparcie dla :

1. Określenia wartości parametrów wejściowych, dla których należy uruchomić obliczenia,
2. Zarządzania wykonaniem wielu uruchomień aplikacji z różnymi wartościami parametrów wejściowych,
3. Zbierania i analizy rezultatów obliczeń,

Infrastruktury wykonują naszą symulację oraz monitorują obliczenia, a następnie agregują wyniki symulacji. Poprzez platformę Scalarm otrzymujemy wyniki naszego eksperymentu, teraz sami decydujemy czy wynik jest zadowalający czy zmienimy wartości wejściowe   
i jeszcze raz przeprowadzimy symulację. Polska infrastruktura obliczeniowa daje nam swobodny dostęp do ich zasobów przez co obliczenia wykonywane na dostępnej mocy obliczeniowej są bardzo dokładne i pozwalają nam na wielokrotne skrócenie czasu obliczeń.

## **Przegląd metod**

Zrównoleglenie algorytmów w dzisiejszych czasach jest ważnym elementem programowania. W literaturze czy Internecie można znaleźć wiele przykładów implementacji wielu algorytmów każdy zrównoleglony w inny sposób, porównać  
 z innymi zrównolegleniami oraz wybrać najbardziej odpowiadającą nam wersję. Najprościej zrównoleglić algorytmy, które oparte są na biologii takie jak algorytm genetyczny czy rój cząstek. Przykładowo zrównoleglenie algorytmu genetycznego polega na przypisaniu każdemu wątkowi innej populacji. Jest to tylko jedna z dróg stworzenia tego algorytmu jako algorytmu współbieżnego. W tej pracy zrezygnowano z implementacji równoległej obu tych algorytmów i postanowiono skupić się na algorytmach przedstawionych w rozdziale drugim, które trudniej zrównoleglić, a poprawnie zaimplementowane mogą przynieść znacznie większe korzyści.

# Realizacja oprogramowania

## **Idea zrównoleglenia wybranych metod**

Algorytmy wykonujące tylko jedne obliczenia w jednej iteracji, które później wykorzystujemy do rozwiązywania skomplikowanych równań matematycznych lub opisu zjawisk fizycznych nie rozwiążą tych równań w krótkim czasie. Wykonywanie dwóch obliczeń jednocześnie teoretycznie powinno skrócić czas o połowę, a dodając kolejne jednostki obliczeniowe jesteśmy w stanie jeszcze skrócić ten czas[5]. Algorytmy, które zostały opisane w rozdziale wyżej nie są typowymi algorytmami, które łatwo można zrównoleglić. Ich podstawowa wersja w jednej iteracji liczy tylko wartość dla jednego punktu. Aby zrównoleglić takie algorytmy trzeba było zastanowić się, który moment będzie najlepszy do zrównoleglenia i przyniesie nam najlepsze wyniki. Zrównoleglenie tych algorytmów znacząco przyspieszyło ich czas obliczeń, co było najważniejszym punktem do osiągnięcia.

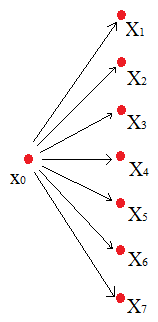
## **Zrównoleglenie metody Hook’a-Jeeves’a**

Algorytm Hooka –Jeevesa można zrównoleglić na kilka sposobów, oto przykładowe:

1. skok próbny wykonywać naraz w kilku kierunkach, przykładowo wykonać skok w lewo i w prawo jednocześnie,
2. tworzyć losowe ścieżki (w różnych kierunkach) w zależności od wartości losowej zmiennej zmieniającej się przy każdej iteracji (dodając do obliczeń kilka punktów w jednym kierunku), przeliczyć wszystkie punkty naraz i wybrać najlepszy,
3. tworzyć skoki w jednym kierunku z pewnymi zadanymi odchyleniami   
   i sprawdzić, w którym punkcie wartość funkcji jest najmniejsza.

W niniejszej pracy zrównoleglenie wykonano wykorzystując sposób opisany w punkcie   
b) oraz punkcie c). Obliczenia, jakie zostały wykonane potwierdziły słuszność wybranej drogi i przyniosły bardzo dobre efekty. Ilość tworzonych nowych losowych ścieżek była uzależniona od ilości wymiarów. Następnie punkty te zostały przeliczone na osobnych wątkach i porównane, co pozwoliło przyspieszyć czas obliczeń w porównaniu   
do implementacji tego algorytmu bez modyfikacji. Dokonano zrównoleglenia zarówno kroku próbnego jak i kroku roboczego. Negatywny wpływ na działanie tego wariantu implementacji mógł mieć czynnik losowy, ponieważ nigdy tak naprawdę nie wiadomo, w którym kierunku algorytm osiągnie lepszą wartość i kiedy będzie można zakończyć obliczenia.

Wariant z punktu c), gdzie tworzyliśmy dużo większą ilość punktów, pozwolił na jeszcze większe skrócenie czasów obliczeń.



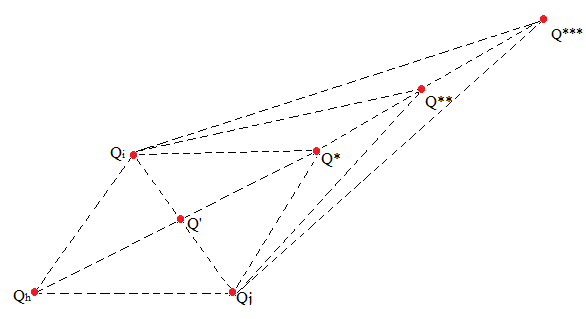
**Rys. 9.** Graficzne przedstawienie zrównoleglenia metody Hook’a Jeeves’a  
skoki z zadanym odchyleniem

Źródło : własne opracowanie

Na rysunku 9 można zobaczyć działanie zrównoleglenia algorytmu Hook’a Jeevesa.   
Łatwo zauważyć jak wygląda odchylenie od wybranego przez nas punktu początkowego.   
Po przeliczeniu wszystkich punktów algorytm porówna wartości funkcji celu i wybierze najmniejszą wartość. Dodanie do przeliczenia tylu punktów, ile jest dostępnych wątków w maszynie równoległej, pozwoli w najlepszy sposób wykorzystać całą moc obliczeniową. Dodanie zbyt dużej ilości punktów już nie będzie przynosiło znacznej poprawy w wydajności, a niewykorzystanie pełnej mocy obliczeniowej wydłuży czas obliczeń danego problemu.

## **Zrównoleglenie metody Nelder’a–Mead’a**

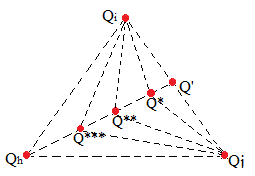
Algorytm Nelder'a - Mead'a został zrównoleglony poprzez jednoczesne przeliczenie wielu punktów, które zostały uzyskane poprzez wielokrotne wykonanie odbicia, ekspansji oraz kontrakcji w jednej iteracji. Wykorzystanie takiego rodzaju zrównoleglenia pozwoliło wykonać wiele obliczeń wartości funkcji w punktach, dzięki czemu czas obliczeń znacznie się zmniejszył. Ilość punktów powstałych przez wielokrotne odbicie, ekspansje oraz kontrakcję podaje użytkownik. W podstawowej wersji tego algorytmu funkcja celu jest liczona osobno dla każdego punktu powstałego np. po odbiciu.



**Rys. 10.** Graficzne przedstawienie wielokrotnego odbicia w algorytmie Nelder’a – Mead’a

Źródło : własne opracowanie

Na rysunku 10 pokazano odbicie poprzez wykonanie trzech skoków w tym samym kierunku. Następnie funkcja oblicza wartość funkcji celu dla wszystkich punktów oraz zwraca ten punkt gdzie jej wartość jest najmniejsza. Jeśli wielokrotne odbicie spowodowało mniejszą wartość funkcji celu wykonujemy wielokrotną ekspansję, czyli wielokrotnie przenosimy się w tym samym kierunku, obliczamy wszystkie punkty i sprawdzamy czy, któryś z nich zwróci minimum. W tym przypadku wielokrotna ekspansja może być także zobrazowana jako rysunek 10. Wybieramy punkt i przenosimy się kilkukrotnie w jednym kierunku.



**Rys. 11.** Graficzne przedstawienie wielokrotnej kontrakcji w algorytmie Nelder’a – Mead’a

Źródło : własne opracowanie

W razie niepowodzenia – wartość funkcji celu po kilkukrotnej ekspansji jest większa – wykonujemy wielokrotną kontrakcję, którą przedstawia rysunek 11.

Algorytm Hook’a - Jeeves’a oraz algorytm Nelder’a - Mead’a zostały zrównoleglone tak, aby przynieść jak najlepsze korzyści z zastosowania ich w skomplikowanych obliczeniach. Wielokrotne testy potwierdziły słuszność wybranej ścieżki implementacji, Oba te algorytmy można zrównoleglić na inne sposoby, ale wybrane drogi implementacji w tej pracy przyniosły oczekiwane rezultaty.

## **Szczegóły implementacji**

Implementacja obu algorytmów została wykonana w języku programowania C#. Język ten jest prosty, nowoczesny oraz zorientowany obiektowo. C# zapewnia przyjazny użytkownikowi interfejs, szybkie wykonywanie aplikacji, wysokie bezpieczeństwo oraz może działać wszędzie tam gdzie jest środowisko .NET bez instalacji pełnego systemu [7]. Badane funkcje są bardzo kosztowne obliczeniowo, że algorytmy zaimplementowane w języku C++ nie przyniosłyby realnego zysku natomiast zaletą C# jest szybka implementacja oprogramowania. Algorytm Hook’a – Jeeves’a oraz Nelder’a Mead’a zostały zrównoleglone tak, aby w zależności od złożoności funkcji oraz ilości punktów do przeliczenia,  
metody mogły zostać wykonane na tylu wątkach ile jest ich dostępnych w maszynie równoległej Do reprezentacji punktów w przestrzeni została wykorzystana klasa OptimizationPoint, która pozwala na przechowywanie punktów w liście, utrzymanie współrzędnych oraz rezultatów obliczeń funkcji celu dla danego punktu, co ułatwia późniejsze porównywanie oraz odnajdywanie najlepszego punktu.

## **Implementacja metody Hook’a–Jeeves’a**

W metodzie Hook’a – Jeeves’a w wersji nierównoległej krok próbny został zaimplementowany w następujący sposób:

*Listing nr. 1:*

|  |
| --- |
| newFx.Inputs[i] -= \_step[i]; // przenosimy się w lewo  Points.Add(newFx); // dodajemy punkt do przeliczenia  EvaluateStep(); // Przeliczenie punktu |

Na listingu 1 można łatwo zauważyć, że w tym przypadku z każdym nowym dodanym punktem przeliczamy go osobno, a następnie wykonujemy operację porównania, aby znaleźć wartość najmniejszą. Krok roboczy został przedstawiony na listingu nr. 2.

*Listing nr. 2:*

|  |
| --- |
| newFx = new Core.OptimizationPoint(G\_Best); for (int i = 0; i < inputCount; i++)  newFx.Inputs[i] += \_lambda \* \_dStart[i]; Points.Add(newFx); //dodajemy punkt do przeliczenia EvaluateStep(); //liczymy punkt |

Krok roboczy pokazany na listingu nr. 2 w wersji podstawowej wykonuje tylko jeden skok i liczy wartość funkcji celu tylko w tym punkcie.

Równoległa wersja tego algorytmu zakłada tworzenie losowych ścieżek ze skoków próbnych oraz skoków roboczych. Następnie wszystkie punkty uzyskane ze ruchów przeliczamy jednocześnie i wybieramy najlepszy punkt gdzie wartość funkcji będzie najmniejsza. Przykład skoku próbnego przedstawiony jest na listingu nr. 3:

*Listing nr. 3:*

|  |
| --- |
| int direction = Helpers.NextInt(0, 1); // funkcja losująca kierunek if (direction == 0){  newFx.Inputs[i] -= \_step[i]; // przenosimy się w lewo   Points.Add(new Core.OptimizationPoint(newFx)); // dodajemy punkt  }  else if (direction == 1){  newFx.Inputs[i] += \_step[i]; // przenosimy się w prawo  Points.Add(new Core.OptimizationPoint(newFx)); // dodajemy punkt  } |

W listingu nr. 3 można zauważyć, że zostaje wylosowany kierunek, w którą stronę będziemy wykonywać skok. Skoki mogą zostać wykonane naprzemiennie, a ilość skoków zależna jest od ilości zmiennych funkcji. Kolejna czynność, którą należy wykonać to dodać punkty do przeliczenia. Odpowiednia funkcja która przyjmuje listę punktów zwróci nam wartość funkcji celu w najlepszym punkcie oraz współrzędne punktu. Funkcja obliczająca jest przystosowana do pracy na wielu wątkach i każdy punkt jest liczony osobno.

Kolejna z równoległych wersji skoku próbnego:

*Listing nr. 4:*

|  |
| --- |
| // generujemy wiele ruchów w lewo  newFx = new Core.OptimizationPoint(G\_Best); // kopia chwilowa  SinglePointJump(newFx, i, -1 \* l); // skok w lewo z odchyleniem  Points.Add(newFx); // dodanie punku |

W wersji zawartej w listingu 4 skoki próbne wykonywane są z podanym przez użytkownika odchyleniem. Funkcja SinglePointJump jako parametry, przyjmuje punkt bazowy, kierunek oraz długość skoku. Na listingu nr. 5 widoczna jest funkcja SinglePointJump:

*Listing nr. 5:*

|  |
| --- |
| void SinlePointJump(Core.OptimizationPoint point, int input, double dist\_multiplier){  for ( int k = 0; k < this.\_inputCount; ++k){  //przenosimy się w lewo lub prawo  if( k == input)  newFx.Inputs[k] += \_step[k] \* dist\_multipler;  else{  //delikatne losowe odchylenie w innych kierunkach  newFx.Inputs[k] += Helpers.NextDouble(-0.1, 0.1) \* \_step[k] \* dist\_multiplier  }  } } |

Funkcja SinglePointJump zwraca punkty odchylone od punktu bazowego tak jak zostało to przedstawione na rysunku 9. W zależności od ilości parametrów wejściowych tyle odchyleń zostanie wykonanych i tyle punktów zostanie dodanych do przeliczenia każdemu przypisując osobny wątek.

## **Implementacja metody Nelder’a–Mead’a**

W metodzie Nelder’a – Mead’a wykonywane są cztery operacje w każdej iteracji: odbicie, ekspansja, kontrakcja oraz kurczenie simpleksu. Na poniższym listingu zostało przedstawione odbicie niezrównoleglone:

*Listing nr 6:*

|  |
| --- |
| private OptimizationPoint Reflect(OptimizationPoint centerPoint, OptimizationPoint point)  {  return centerPoint + (centerPoint - point) \* \_reflectance;  } |

Funkcja Reflect przyjmuje jako argumenty funkcji środek ciężkości simpleksu oraz punkt odbijany – w tym przypadku najgorszy punkt. Zwraca nam punkt uzyskany po odbiciu.  
 Aby można było wykonywać operacje na współrzędnych wybranych punktów zostały zaimplementowane operatory sumy, różnicy, iloczynu i ilorazu.

*Listing nr 7:*

|  |
| --- |
| public static OptimizationPoint operator + (OptimizationPoint p1, OptimizationPoint p2)  {  return new OptimizationPoint(p1.Inputs.Select((x, i) => x + p2.Inputs[i]).ToList());  } |

Na listingu 6 został pokazany przeładowany operator dodawania dla współrzędnych wybranych punktów.

W wersji równoległej odbicie wygląda następująco:

*Listing nr 8:*

|  |
| --- |
| public List<OptimizationPoint> Reflect(OptimizationPoint centerPoint, OptimizationPoint point, int count) {  List<OptimizationPoint> points = new List<OptimizationPoint>();  for (int i = 0; i < count; ++i) {  points.Add(centerPoint + (centerPoint - point) \* \_reflectance \* (i + 1.5));  }  return points;  } |

Metoda przestawiona na listingu nr. 8 przyjmuje środek ciężkości, punkt odbijany oraz ilość wykonanych odbić. Funkcja zwraca nam listę punktów wykonaną przez wielokrotne odbicie, które następnie trafiają do funkcji obliczającej, która zwróci punkt z najmniejszą wartością funkcji celu oraz jego współrzędnymi.

Listing nr. 9 prezentuje ekspansję wykonaną wielowątkowo:

*Listing nr 8:*

|  |
| --- |
| public List<OptimizationPoint> Stretch(OptimizationPoint centerPoint, OptimizationPoint point, int count) {  List<OptimizationPoint> points = new List<OptimizationPoint>();  for (int i = 0; i < count; ++i) {  points.Add(centerPoint + (centerPoint - point) \* \_stretch \* (i + 1.5));  }  return points;  } |

Ekspansja przedstawiona na listingu nr. 9 została zaimplementowana w oparciu o metodę odbicia. W taki sam sposób została napisana czynność wykonująca kontrakcję.  
Każda z operacji różni się w niewielkim stopniu wzorem oraz współczynnikiem.Każda z wykonywanych czynności posiada własny współczynnik. Współczynnik odbicia-*\_reflect*, ekspansji-*\_stretch* i kontrakcji-*compress* może zostać dowolnie ustawiony przez użytkownika.

# Wyniki

## **Specyfikacja sprzętu**

. Każdy z algorytmów został przetestowany na procesorze Intel(R) Core(TM) i5-4210 U CPU @ 1-70 GHZ, który jest oparty o 64-bitową architekturę. Procesor należy do rodziny procesorów niskonapięciowych tzw. ultramobilnych wykonany w procesie technologicznym 22 nm. Charakteryzuje się dwoma rdzeniami oraz czterema wątkami.

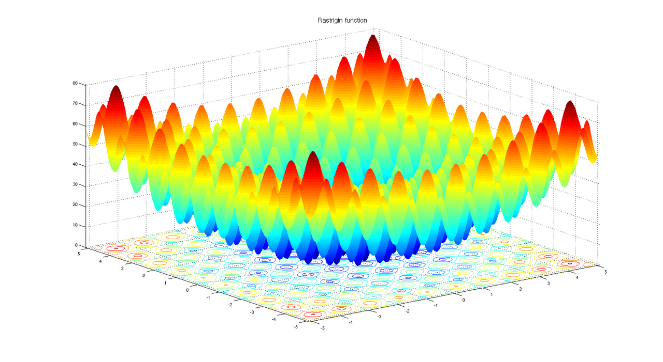
## **Funkcje Testowe**

Algorytmy przedstawione w tej pracy zostały poddane badaniom przy użyciu funkcji testowych, które oceniają cechy algorytmów optymalizacji. Kryteriami zatrzymania optymalizacji dla każdych z algorytmów dla funkcji testowych były:

1. maksymalny dopuszczalny błąd e = 0,01,
2. maksymalna liczba iteracji wynosząca 100.

## **Funkcja testowa Rastrigin**

Funkcja Rastrigin charakteryzuje się wieloma minimami lokalnymi. Z tego powodu często prawdziwe minimum znajdujące się w *f*(0, 0) = 0 nie zostaje odnalezione. Funkcja została przedstawiona na rysunku nr. 12:



**Rys. 12.** Funkcja testowa : Rastrigin  
Źródło : [8]

A jej wzór wygląda następująco:

|  |  |
| --- | --- |
|  | (8) |

Gdzie:  
*n*–liczba parametrów wejściowych

*xi-*współrzędne punktu

W Tabeli nr 1 zostały zestawione obliczenia dla każdego z algorytmów.

**Tabela 1.** Zestawienie wyników obliczeń algorytmów dla badanej funkcji testowej

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Ilość kroków | Liczba wywołań funkcji celu | Wartość | x1 | x2 |
| Metoda Hook'a-Jeeves'a | 65 | 182 | 0,990 | -0,990 | 0,990 |
| Metoda Hook'a-Jeeves'a równolegle | 53 | 1263 | 0,990 | 0,990 | 1,600 |
| Metoda Hook'a-Jeeves'a równolegle - druga wersja | 41 | 1834 | 0,009 | -0,002 | -0,006 |
| Metoda Nelder'a-Mead'a | 88 | 199 | 1,010 | -0,990 | -0,990 |
| Metoda Nelder'a-Mead'a równolegle | 65 | 480 | 0,010 | 0,003 | 0,006 |

Z powyższej tabeli można wywnioskować, że w przypadku funkcji Rastrigin nie zawsze zostaje znalezione minimum globalne. Jest to spowodowane tym, że nasza funkcja posiada wiele minimów lokalnych, a zadeklarowany najmniejszy błąd oraz ilość iteracji nie pozwala na większe przeszukiwanie przestrzeni. W tym przypadku najdokładniejsze wyniki przyniosła druga wersja równoległa algorytmu Hook’a–Jeeves’a.

**Rys. 13.** Wykres zależności ilości iteracji od wywołań funkcji celu dla funkcji testowej Rastrigin

Źródło : własne opracowanie

Wykres przedstawiony na rysunku 12 obrazuje, że im więcej wywołań funkcji celu w jednej iteracji, tym wynik jest dokładniejszy, a algorytmy pozwolą obliczyć skomplikowane problemy. W tym przypadku metoda Hook’a–Jeeves’a zrównoleglona drugą metodą przyniosła najlepsze rezultaty.

## **Funkcja testowa Ackley’s**

Kolejnym przykładem jak został pokazany w tej pracy jest funkcja testowa Ackley’s,

której minimum jest w *f*( 0, 0 ) = 0. Jej wykres przedstawiony jest na rysunku 14:

|  |
| --- |
| **Rys. 14.** Funkcja testowa : Ackley's Źródło : [8] |

A jej wzór charakteryzuje się:

|  |  |
| --- | --- |
|  | (9) |

W Tabeli 2 zostały zaprezentowane wyniki algorytmów :

**Tabela 2.** Zestawienie wyników obliczeń algorytmów dla badanej funkcji testowej

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Ilość kroków | Liczba wywołań funkcji celu | Wartość | x1 | x2 |
| Metoda Hook'a-Jeeves'a | 34 | 76 | -0,560 | 0,060 | -0,090 |
| Metoda Hook'a-Jeeves'a równolegle | 28 | 441 | -0,580 | 0,040 | 0,100 |
| Metoda Hook'a-Jeeves'a równolegle - druga wersja | 22 | 871 | -0,640 | 0,100 | 0,001 |
| Metoda Nelder'a-Mead'a | 37 | 101 | -0,310 | -0,030 | -0,120 |
| Metoda Nelder'a-Mead'a równolegle | 25 | 173 | -0,140 | 0,008 | 0,070 |

Ta funkcja testowa nie była skomplikowana i algorytmy obliczyły jej minimum w dużo mniejszej ilości iteracji w porównaniu do funkcji Rastragin. Każdy z algorytmów szybko poradził sobie z zadanym problemem, a wyniki są bardzo do siebie zbliżone co potwierdza słuszność wybranej drogi.

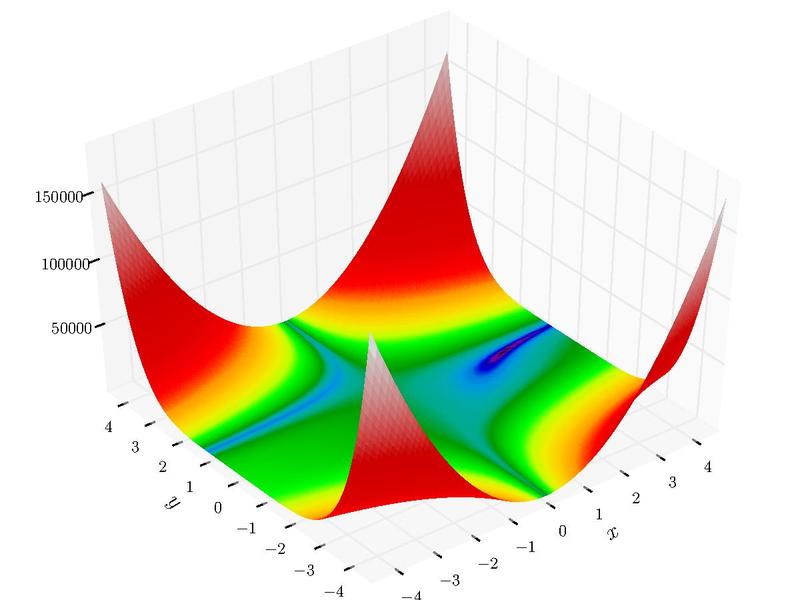
**Rys. 15.** Wykres zależności ilości iteracji od wywołań funkcji celu dla funkcji testowej Ackley’s

Źródło : własne opracowanie

Także w tym przypadku druga wersja równoległa Hook’a Jeeves’a przyniosła największą liczbę wywołań funkcji celu w pojedynczej iteracji, spowodowane jest to tym, że dodajemy bardzo dużą ilość punktów do przeliczenia.

## **Funkcja testowa Baley’s**

Trzecia z kolei funkcja testowa przedstawiona w tej pracy to funkcja Beale’s posiadająca minimum w *f*( 3, 0.5 ) = 0. Graficzne przedstawienie badanej funkcji znajduje się na rysunku 16.



**Rys. 16.** Funkcja testowa : Beale's  
Źródło : [8]

Wzór funkcji:

|  |  |
| --- | --- |
|  | (9) |

Zestawienie wyników znajduje się w tabeli 3:

**Tabela 3.** Zestawienie wyników obliczeń algorytmów dla badanej funkcji testowej

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Ilość kroków | Liczba wywołań funkcji celu | Wartość | x1 | x2 |
| Metoda Hook'a-Jeeves'a | 42 | 114 | 0,008 | 2,790 | 0,440 |
| Metoda Hook'a-Jeeves'a równolegle | 38 | 559 | -0,002 | 3,035 | 0,498 |
| Metoda Hook'a-Jeeves'a równolegle - druga wersja | 27 | 727 | 0,005 | 3,180 | 0,548 |
| Metoda Nelder'a-Mead'a | 39 | 97 | -0,008 | 2,880 | 0,453 |
| Metoda Nelder'a-Mead'a równolegle | 25 | 141 | -0,002 | 2,930 | 0,491 |

W tym przypadku najlepiej poradził sobie zrównoleglony algorytm Nelder’a – Mead’a który w najmniejszą ilość kroków podał najdokładniejszy wynik. Jest to spowodowane najlepiej obraną ścieżką tworzenia nowych simpleksów dla tego typu funkcji testowych.

**Rys. 17.** Wykres zależności ilości iteracji od wywołań funkcji celu dla funkcji testowej Baele’s

Źródło : własne opracowanie

## **Funkcja testowa Booth’s**

Ostatnia z testowych funkcji ma jedno minimum dla *f* (1, 3) = 0. Rysunek 17 pokazuje wykres funkcji:

|  |
| --- |
| File:Booth's function.pdf  **Rys. 17.** Funkcja testowa : Beale's Źródło : [8] |

Wzór funkcji:

|  |  |
| --- | --- |
|  | (9) |

W Tabeli 4 zostały przedstawione wyniki:

**Tabela 4.** Zestawienie wyników obliczeń algorytmów dla badanej funkcji testowej

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Ilość kroków | Liczba wywołań funkcji celu | Wartość | x1 | x2 |
| Metoda Hook'a-Jeeves'a | 30 | 82 | 0,007 | 1,014 | 2,949 |
| Metoda Hook'a-Jeeves'a równolegle | 25 | 381 | 0,006 | 0,930 | 3,054 |
| Metoda Hook'a-Jeeves'a równolegle - druga wersja | 20 | 486 | 0,004 | 1,013 | 3,017 |
| Metoda Nelder'a-Mead'a | 29 | 63 | 0,005 | 1,012 | 3,000 |
| Metoda Nelder'a-Mead'a równolegle | 23 | 132 | 0,007 | 1,049 | 2,927 |

Każdy z przedstawionych algorytmów obliczył minimum tej funkcji testowej   
w małej ilości kroków – jest to spowodowane nie skomplikowaniem funkcji celu.  
Dokładna wartość minimum nie została odnaleziona, ponieważ maksymalny błąd został globalnie ustawiony na 0,01.

**Rys. 18.** Wykres zależności ilości iteracji od wywołań funkcji celu dla funkcji testowej Baele’s

Źródło : własne opracowanie

## **Podsumowanie**

Każdy z algorytmów w wersji podstawowej oraz zrównoleglonej poradził sobie z funkcjami testowymi. Można zauważyć, że wraz z zwiększoną ilością wywołań funkcji celu w pojedynczej iteracji wartość jest dokładniejsza oraz zostaje odnaleziona w mniejszej ilości kroków. Jednak najważniejszym powodem zrównoleglenia tych algorytmów, było zmniejszenie czasu obliczeń. W tabeli 5 zostały zestawione czasy wykonania poszczególnych algorytmów:

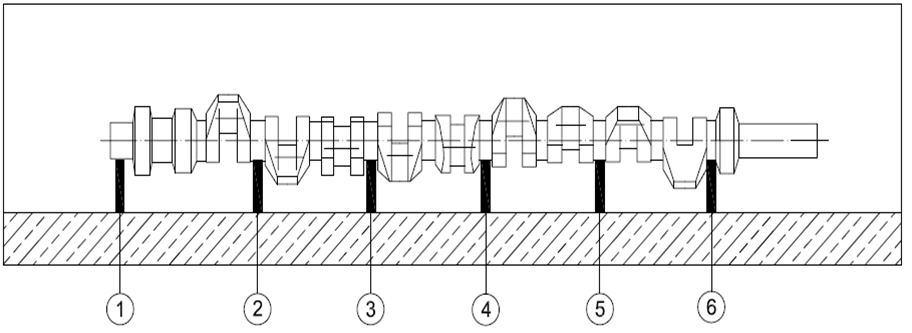
**Tabela 5.** Zestawienie czasów wykonania funkcji testowych

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Funkcja testowa Rastigin | Funkcja testowa Ackley's | Funkcja testowa Baley's | Funkcja testowa Booth's |
| Metoda Hook'a-Jeeves'a | 47 ms | 39 ms | 43 ms | 27 ms |
| Metoda Hook'a-Jeeves'a równolegle | 39 ms | 17 ms | 24 ms | 19 ms |
| Metoda Hook'a-Jeeves'a równolegle - druga wersja | 33 ms | 14 ms | 24 ms | 21 ms |
| Metoda Nelder'a-Mead'a | 88 ms | 67 ms | 41 ms | 23 ms |
| Metoda Nelder'a-Mead'a równolegle | 63 ms | 29 ms | 26 ms | 19 ms |

Z powodu nieskomplikowanych funkcji testowych w przestrzeni dwuwymiarowej czasy algorytmów są bardzo niskie. Każda z wersji równoległych przyniosła skrócenie czasu obliczeń w pewnym stopniu. Z danych przedstawionych w Tabeli 5 można wywnioskować, że zrównoleglenie algorytmów metodami przedstawionymi w rozdziale czwartym przyniosło oczekiwane rezultaty.

## **Problem praktyczny**

Problem praktyczny jakim postanowiono się zająć w tej pracy to proces wytwarzania wału korbowego dla silników łodzi. Jest to skomplikowany proces, który angażuje bardzo dużą moc obliczeniową dlatego optymalizowany jest projekt do symulacji produkcji.



**Rys. 18.** Wał korbowy

Źródło : ??????

Parametry wejściowe jakie zostały brane pod uwagę to:

1. Temperatura początkowa [oC] – zakres: { 783, 957 },
2. Moduł Younga – zakres: {196 200, 239 800},
3. Granica plastyczności [MPa] – zakres: {45, 55}

Natomiast parametrem wyjściowym jest ugięcie wału które powinno być jak najmniejsze.

# Wnioski

# Literatura

[1] Vázquez S., Martín M.J., Fraguela B.B., Gómez A., Rodríguez A., Elvarsson B.B., *Novel parallelization of simulated annealing and Hooke & Jeevessearch algorithms for multicore systems with application to complexfisheries stock assessment models*, Journal of Computational Science, 2016, in print

[2] Luchi F. Renato A. Krohling *Differential Evolution and Nelder – Mead for Constrained Non – linear Integer Optimization Problems*, Procedia Computer Science, Volume 55, 2015, Pages 668 - 677

[3] http://optymalizacja.w8.pl/simplexNM.html, maj 2008

[4] Nelder J. A., Mead R., *A simplex method for function minimization*, Computer Journal, Volume 7, 1965, 308-315

[5] Migallón H. Migallon V. Penades J., *Parallel alternating iterative algorithms with and without overlapping on multicore architectures*, Advances in Engineering Software, Volume 101, 2016, Pages 27-36

[7] Zhang Yu, Jian-Ping A. Chen P. *Research of Hybrid Programming with C#.Net and Matlab*, Physics Procedia, Volume 24, 2012, Pages 1677 – 1681

[8] https://en.wikipedia.org/wiki/Test\_functions\_for\_optimization