



POLITECHNIKA WARSZAWSKA



WYDZIAŁ MATEMATYKI
I NAUK INFORMACYJNYCH

PRACA DYPLOMOWA INŻYNIERSKA

INFORMATYKA

Efektywna implementacja wybranego algorytmu optymalizacji globalnej

Autorzy:

Paweł Kupidura

Hubert Rosiak

Kacper Trojanowski

Promotor: mgr inż. Michał Okulewicz

Warszawa, wrzesień 2016

.....

podpis promotora

.....

podpis autora

Streszczenie

W niniejszej pracy inżynierskiej prezentujemy rozproszony system do optymalizacji umożliwiający uruchomienie obliczeń na klastrze złożonym z kilku komputerów, którego bazowym silnikiem optymalizacyjnym jest algorytm Particle Swarm Optimization. Program napisany został ze szczególnym uwzględnieniem optymalizacji funkcji ciągłych z benchmarku BBOB jednak umożliwia programiście rozszerzenie funkcjonalności do dowolnego rodzaju funkcji. Algorytm zaimplementowany został w postaci biblioteki w technologii .NET umożliwiającej dopasowanie się do sprzętu, na którym jest uruchamiana wykorzystując wieloprocessorowość lokalnej maszyny oraz umożliwiającej uruchomienie obliczeń rozproszonych na wielu komputerach jednocześnie wykorzystując usługi WCF. Biblioteka pozwala też na wykorzystanie do obliczeń równoległych procesorów graficznych obsługujących technologię CUDA. Jako test działania prezentujemy wyniki testów naszej biblioteki na benchmarku optymalizacyjnym BBOB.

Spis treści

1	Słownik	4
2	Wstęp teoretyczny	6
2.1	Czym jest PSO?	6
2.2	Definicja algorytmu	7
2.3	Dotychczasowy stan badań nad PSO	8
2.4	Istniejące implementacje i zastosowania PSO	10
2.5	Równoległe i rozproszone PSO	11
2.5.1	Stan badań nad równoległym i rozproszonym PSO	12
3	Zaimplementowany system optymalizacyjny	15
3.1	Architektura węzła obliczeniowego	15
3.1.1	MachineManager	15
3.1.2	VCpuManager	16
3.1.3	IPsoController	16
3.2	Implementacja algorytmu	17
3.2.1	Algorytm	18
3.2.2	Cząstka PSO	19
3.2.3	Funkcja celu	21
3.2.4	IPsoManager	22
3.2.5	NetworkNodeManager	23
3.2.6	INodeService	24
3.3	Komunikacja w klastrze obliczeniowym	25

3.3.1	Komunikacja pomiędzy węzłami obliczeniowymi	26
3.3.2	Komunikacja na poziomie algorytmu PSO	27
3.4	Implementacja PSO na GPU	30
3.4.1	Porównanie implementacji PSO na CPU i GPU	30
3.4.2	Krótkie wprowadzenie do architektury CUDA	30
3.4.3	Implementacja algorytmu	31
4	Platforma COCO	37
4.1	Testowanie algorytmów optymalizacyjnych	37
4.2	Wrapper	38
4.2.1	Eksport\import funkcji z języka C	39
4.2.2	Wykorzystanie wrappera	40
5	Testy	42
5.1	Wyniki testów	42
5.1.1	Pojedynczy węzeł obliczeniowy	43
5.1.2	Klaster	47
5.2	Interpretacja wyników	49
6	Wymagania techniczne i instrukcja	53
6.1	Wymagania sprzętowe i programowe	53
6.2	Instrukcja użytkownika	53
6.2.1	Interfejs użytkownika	53

Rozdział 1

Słownik

Algorytm optymalizacyjny - Algorytm rozwiązujący problem optymalizacji.

Benchmark - Sposób oceny jakości sprzętu lub programu komputerowego polegający na uruchomieniu zestawu testów i porównaniu ich ze znanymi standardami.

Black-box optimization - Rodzaj zadania optymalizacyjnego, w którym algorytm optymalizacyjny nie zna postaci funkcji celu, a może jedynie zapytać się o jej wartość w danym punkcie.

CUDA - Platforma programistyczna firmy NVIDIA umożliwiająca wykorzystywanie procesorów graficznych w obliczeniach

Framework - Platforma programistyczna. Szkielet aplikacji definiujący jej ogólną strukturę i mechanizm działania lub dostarczający komponenty wykonujące określone zadania.

Funkcja celu - Funkcja, której ekstremum jest poszukiwane w zagadnieniu optymalizacji.

GPU - Graphics Processing Unit. Wyspecjalizowana jednostka obliczeniowa typu SIMD przystosowana do efektywnego przetwarzania grafiki komputerowej. Może być również wykorzystywana do obliczeń równoległych na dużych zbiorach danych.

Klaster obliczeniowy - Zespół węzłów obliczeniowych połączonych ze sobą siecią, umożliwiający prowadzenie wspólnych obliczeń.

Marshalling - Zmiana sposobu reprezentacji danych w pamięci do formatu wła-

ściwego dla transmisji do innego fragmentu programu. **Metaheurystyka** - Ogólny algorytm do rozwiązywania problemów obliczeniowych, który można dostosować do konkretnego problemu.

Obliczenia rozproszone - Obliczenia prowadzone jednocześnie na wielu jednostkach obliczeniowych, mogących różnić się między sobą architekturą i położeniem geograficznym, ale komunikujących się ze sobą i mogących korzystać ze wspólnych zasobów. Synchronizacja między jednostkami zachodzi w szerszej skali niż w przypadku obliczeń równoległych.

Obliczenia równoległe - Zsynchronizowane obliczenia prowadzone jednocześnie na kilku jednostkach obliczeniowych, zazwyczaj procesorach na jednej maszynie.

Optymalizacja - Zagadnienie znajdowania wartości ekstremalnej (najczęściej minimalnej lub maksymalnej) zadanej funkcji celu.

PSO - Particle Swarm Optimization. Ewolucyjny algorytm optymalizacyjny, wykorzystujący populację cząstek poruszających się w przestrzeni możliwych rozwiązań.

SIMD - Single instruction, multiple data. Jeden z rodzajów architektur jednostek obliczeniowych, w którym wiele strumieni danych przetwarzanych jest w oparciu o pojedynczy strumień rozkazów.

WCF - Windows Communication Foundation. Platforma programistyczna wykorzystywana przy budowie aplikacji korzystających z komunikacji sieciowej.

Wrapper - Zestaw funkcji i klas mających za zadanie pośredniczenie między dwiema różnymi warstwami aplikacji.

Rozdział 2

Wstęp teoretyczny

2.1 Czym jest PSO?

PSO (Particle Swarm Optimization) jest metodą rozwiązywania problemów optymalizacyjnych należącą do klasy metaheurystycznych algorytmów ewolucyjnych, wzorowaną na spotykanym w przyrodzie zachowaniu się rojów i stad zwierząt. Po raz pierwszy przedstawiona została w pracy Eberhardta i Kennedy'ego w 1995 r. [6]. W ogólności polega ona na potraktowaniu potencjalnych rozwiązań zagadnienia optymalizacyjnego jako „cząstek”, które poruszają się w przestrzeni możliwych rozwiązań. Ruch cząsteczek odbywa się w czasie dyskretnym - w każdej iteracji głównej pętli algorytmu każda z cząstek przemieszcza się o wektor prędkości, który zależy od prędkości w poprzedniej iteracji, najlepszego dotychczas znalezionego przez sąsiadów rozwiązania (globalnego), najlepszego rozwiązania znalezionego przez daną cząstkę (lokalnego) oraz od czynnika losowego. Na poziomie praktycznym za przestrzeń rozwiązań najczęściej wybiera się n -wymiarową przestrzeń Euklidesową, zaś zagadnieniem optymalizacyjnym jest znalezienie minimum (maksimum) pewnej funkcji rzeczywistej określonej na pewnym podzbiorze tej przestrzeni. Ze względu na fakt, iż PSO jest jedynie (meta)heurystyką, szczegóły algorytmu i jego parametry można dobierać odpowiednio do konkretnego zastosowania.

2.2 Definicja algorytmu

Klasyczna wersja algorytmu (wg. [6]), zakładająca pełne sąsiedztwo cząstek [pełny graf sąsiedztwa] wygląda następująco:

1. Inicjalizacja cząstek:

- (a) Utwórz cząstki rozmieszczone losowo (według rozkładu jednostajnego) w całej przestrzeni rozwiązań;
- (b) Jako najlepszą znaną pozycję ustaw dla każdej z cząstek jej pozycję startową;
- (c) Zaktualizuj optimum globalne znajdując pozycję cząstki o najmniejszej wartości funkcji celu;
- (d) Nadaj cząstkom losowe prędkości z pewnego zakresu (według rozkładu jednostajnego);

2. Główna pętla algorytmu - dopóki nie został spełniony warunek stopu (określona liczba iteracji lub znalezienie wartości odpowiednio bliskiej znanemu optimum globalnemu), wykonuj:

- (a) Dla każdej z cząstek oblicz jej nową prędkość w następujący sposób:

$$v_{t+1} \leftarrow \omega \cdot v_t + \phi_1 \cdot r_1 \cdot p + \phi_2 \cdot r_2 \cdot g$$

gdzie: v_{t+1} - wektor prędkości w iteracji $t + 1$,

v_t - wektor prędkości w iteracji t ,

p - wektor łączący obecne położenie cząstki z jej najlepszą znaną pozycją,

g - wektor łączący obecne położenie cząstki z najlepszą globalnie pozycją,

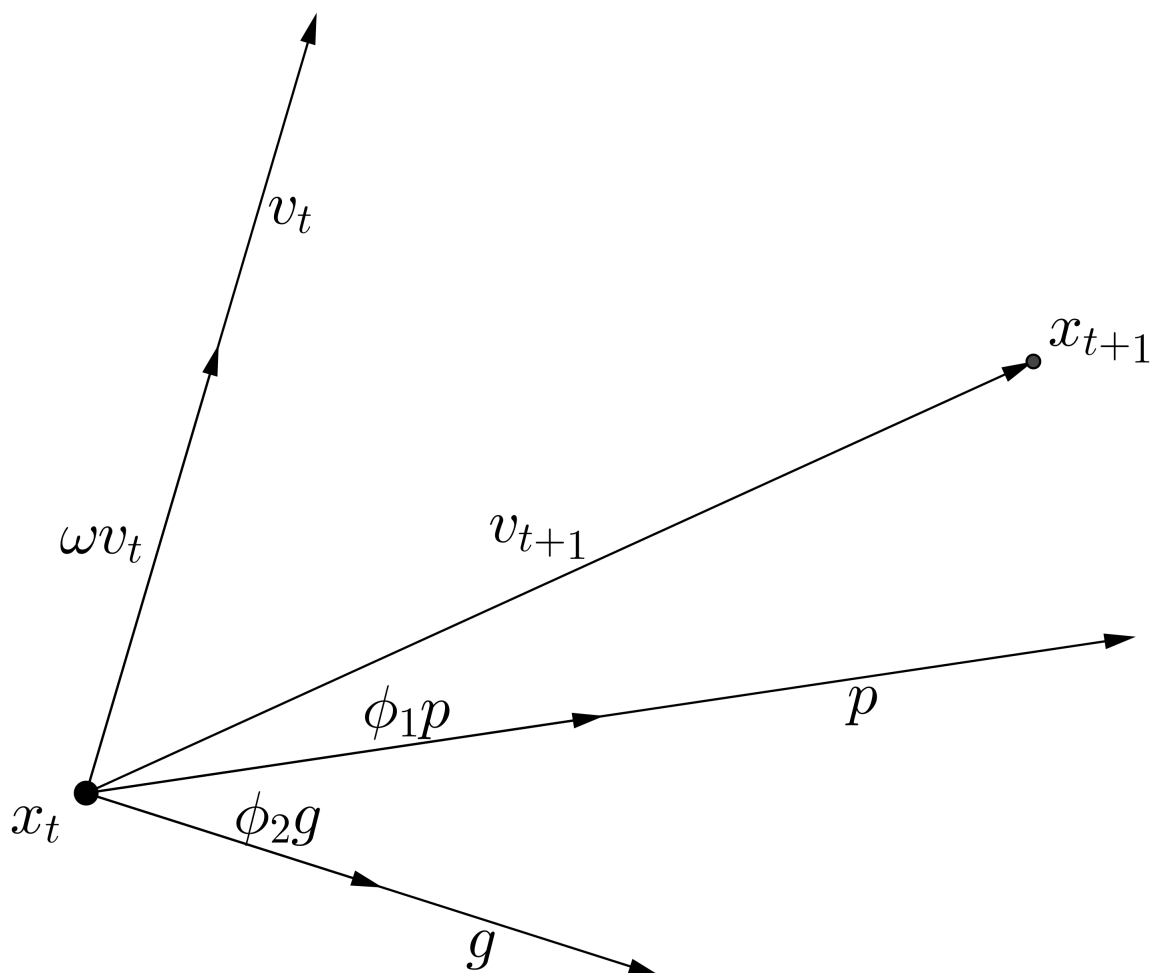
ω - współczynnik bezwładności,

ϕ_1, ϕ_2 - ustalone przez użytkownika parametry (wagi),

r_1, r_2 - liczby z przedziału $[0, 1]$ losowane w każdej iteracji

- (b) Przesuń każdą cząstkę o jej nowy wektor prędkości;

- (c) Dla każdej z cząstek oblicz funkcję celu w nowym położeniu i, jeżeli to możliwe, zaktualizuj najlepszą dotychczasową pozycję cząstki;
- (d) Zaktualizuj optimum globalne;



Rysunek 2.1: Algorytm PSO w dwóch wymiarach

2.3 Dotychczasowy stan badań nad PSO

Od momentu powstania w 1995 r. algorytm ten zyskał popularność i szerokie zastosowanie. Był też tematem licznych prac naukowych i opracowań, mających na celu

próbę jego lepszego zrozumienia i poprawy skuteczności oraz zbadanie zachowania dla wielu różnych klas problemów optymalizacyjnych.

Elastyczność definicji algorytmu sprawia, że wiele jego parametrów można zmieniać i dostosowywać do aktualnych potrzeb. Skuteczność standardowej wersji PSO w zależności od parametrów takich, jak: wielkość roju cząstek, sposób inicjalizacji położenia i prędkości oraz ich aktualizacji w głównej pętli algorytmu czy topologia sąsiedztwa cząstek roju została zbadana w pracy [7]

Dobre podsumowanie stanu badań nad PSO znajduje się w pracy [9], której autorzy wspominają, iż szybko zauważono, że niektóre słabości standardowej, opisanej powyżej wersji algorytmu, można usprawnić, wprowadzając do niego pewne dodatkowe elementy, często inspirowane lub wprost pochodzące od innych algorytmów optymalizacyjnych. Powstały w ten sposób różne warianty i hybrydy algorytmu PSO.

Jedna z prostszych modyfikacji wprowadza parametr „bezwładności ω ” do wzoru wyliczającego zaktualizowaną prędkość każdej cząstki w następujący sposób:

Kontrolowanie bezwładności ma umożliwiać zrównoważenie zdolności eksploatacyjnych i eksploracyjnych algorytmu – duża wartość sprzyja eksplorowaniu większego obszaru, zaś mniejsza – skupienie się na bardziej lokalnym przeszukiwaniu. Bezwładność może też być dynamicznie zmieniana w trakcie wykonywania algorytmu – jej liniowe zmniejszanie się względem liczby iteracji przedyskutowano w [10], zaś właściwy dobór parametrów algorytmu omówiono w [11]. Inne, nieliniowe metody zmiany bezwładności opisano w [12]. W pracy [13] dla odmiany, parametr w ustawiany jest na 0 za wyjątkiem sytuacji reinicjalizacji. W trakcie działania algorytmu można dynamicznie modyfikować także inne jego parametry, jak np. prędkość maksymalną – wariant algorytmu implementujący jej liniowy spadek opisany został w [14].

Duży wpływ na wydajność algorytmu ma też w oczywisty sposób topologia roju cząstek. J. Kennedy (jeden z twórców pierwotnej wersji algorytmu) stwierdził, że mniej liczne sąsiedztwo każdej cząsteczki sprawdza się lepiej w przypadku bardziej złożonych problemów, podczas gdy liczne sąsiedztwo działa lepiej dla pro-

stych problemów [15], [16]. Ciekawym rozwiązaniem jest zaproponowane w [17] dynamicznie dostosowywane sąsiedztwo, które stopniowo powiększa się, aż do momentu objęcia całego roju. W artykule [18] Hu i Eberhart zaproponowali inny sposób dynamicznego modyfikowania sąsiedztwa – w każdej iteracji algorytmu cząsteczka na nowo wybiera sobie za sąsiadów te cząstki, które są jej bliskie według pewnej metryki. Wariant UPSO (Unified Particle Swarm Optimizer) [19] stara się połączyć globalną eksplorację z lokalną eksploatacją. Wprowadzone przez Mendesa i Kennedy'ego [20] Fully Informed PSO różni się od klasycznej wersji tym, że w momencie aktualizacji prędkości cząstki pod uwagę bierze się nie tylko informacje pochodzące od najlepszego sąsiada, ale, z odpowiednimi wagami, również informacje zebrane od pozostałych sąsiadów cząstki. Wagę, jaką przypisuje się informacji od każdej z cząstek sąsiedztwa zależy m.in. od jej aktualnej wartości funkcji celu oraz rozmiaru samego sąsiedztwa. Z kolei fitness-distance-ratio-based PSO (FDR-PSO) [21], wprowadza pewne dodatkowe interakcje między pobliskimi cząstkami podczas aktualizacji prędkości.

Innym sposobem usprawnienia działania algorytmu, który doczekał się licznych opracowań, jest hybrydyzacja PSO, czyli próba połączenia z innymi algorytmami i technikami optymalizacji. Jednym z popularnych pomysłów jest wprowadzenie do PSO operacji znanych z innych algorytmów ewolucyjnych, takich jak selekcja, krzyżowanie i mutacja, w celu zachowania cech najlepszych cząstek roju, bądź też wprowadzenie większej różnorodności do populacji, mające zapobiec zbieżności do lokalnych minimów [22], [23]. Z tego samego powodu wprowadza się różne mechanizmy unikania kolizji między [26], takie jak relokacja cząstek, które znalazły się zbyt blisko siebie [25]. Operatory mutacji są też stosowane do mutowania parametrów algorytmu takich jak opisana wcześniej bezwładność [24]. W pracy [23], rój cząstek dzielony jest na mniejsze subpopulacje, zaś „breeding operator” jest stosowany w obrębie jednej z nich lub też między nimi w celu zwiększenia różnorodności roju. Odpowiedzią na problem zbyt wczesnej zbieżności cząsteczek do minimów lokalnych, które może spowodować ominięcie minimum globalnego, jest wprowadzenie negatywnej entropii [27]. W [28] z kolei, przedstawione zostały techniki, których ce-

lem jest znalezienie jak największej liczby minimów funkcji kosztu poprzez zapobieganie poruszaniu się cząsteczek w kierunku znanych już minimów. Jednym z innych wariantów mających poprawić wyniki otrzymywane przez PSO na funkcjach multimodalnych jest tzw. cooperative PSO (CPSO-H), używające jednowymiarowych rojów do oddzielnego przeszukiwania każdego wymiaru zadanego problemu, których wyniki są następnie w odpowiedni sposób łączone.

2.4 Istniejące implementacje i zastosowania PSO

Wspomniana popularność algorytmów ewolucyjnych, a wśród nich PSO, poskutkowała ich wykorzystaniem zarówno do badań teoretycznych jak i praktycznych zastosowań. Spowodowało to powstanie wielu implementacji tych algorytmów, z których jednak każda posiada swoje wady i ograniczenia. Jednym z podstawowych problemów wielu bibliotek jest narzucona z góry, sztywna reprezentacja obiektów, które chcemy poddać procesowi ewolucji, jak i operatorów ewolucyjnych – często pozwalają one jedynie na korzystanie niewielkiego zbioru predefiniowanych reprezentacji, co zmusza do „spłaszczania” bardziej skomplikowanych struktur danych do typowych postaci, na których operują istniejące biblioteki, takich jak ciągi bitów, czy tablice liczb – podejście takie może znacząco utrudnić zarówno zrozumienie, jak i rozwiązanie problemu.

Jedną z odpowiedzi na opisane wyżej problemy jest napisana w języku C++ open source’owa biblioteka EOlib (Evolving objects library), zawdzięczająca swoją elastyczność podejściu obiektowemu – każda struktura danych jak i każdy operator jest obiektem. Biblioteka zawiera kilka predefiniowanych reprezentacji, ale każdy użytkownik może stworzyć swoje własne „ewoluujące obiekty”, o ile tylko implementują one wymagany interfejs, tzn. zapewniają możliwość inicjalizacji, selekcji osobników oraz ich reprodukcji i mutacji (lub krzyżowania) .

Kolejną zaletą EOlib jest odejście od często stosowanego, ale ograniczającego założenia, iż funkcja celu musi być funkcją skalarną – w bibliotece tej może ona być dowolnego typu.

Biblioteka Eolib jest podstawową popularnego frameworka Paradiseo (typu white-box, czyli dającego programiście wgląd w szczegóły implementacji), służącego do tworzenia metaheurystyk. Składa się on z modułów: Paradiseo-EO - obsługującego algorytmy populacyjne, Paradiseo-MOEO – służącego do optymalizacji [wielokryteriowej], Paradiseo-MO – dla problemów z jednym rozwiązaniem, Paradiseo-PEO – narzędzia do tworzenia rozwiązań równoległych i rozproszonych.

2.5 Równoległe i rozproszone PSO

Obliczenia rozproszone polegają na uruchomieniu jednoczesnych obliczeń na więcej niż jednym komputerze. W przypadku algorytmu PSO rozproszenie obliczeń można wykonać na wiele sposobów - na przykład za pomocą jednego wielkiego roju cząstek działającego równoległe na wszystkich maszynach lub też za pomocą większej liczby mniejszych rojów.

Standardowa wersja algorytmu PSO wprost ze swojej natury nadaje się do zrównoleglenia obliczeń - niezależnie czy mówimy o zrównolegleniu na poziomie roju czy części roju cząstek. Obliczanie nowej prędkości dla każdej z cząstek zależy tylko od jej własności i własności sąsiadów z poprzedniej iteracji, zatem w oczywisty sposób potrzeby synchronizacji sprowadzają się do zaktualizowania zbioru cząstek i wyboru globalnego minimum (maximum).

2.5.1 Stan badań nad równoległym i rozproszonym PSO

W niniejszym podrozdziale przedstawiamy obecny stan badań nad zrównolegleniem i rozproszeniem algorytmu PSO na podstawie dostępnej literatury.

Algorytm PSO jest algorytmem w oczywisty sposób nadającym się do zrównoleglenia obliczeń, ze względu na fakt, że poszczególne osobniki populacji (w przypadku PSO – cząstki) są od siebie w mniejszym bądź większym stopniu niezależne i przeprowadzają samodzielne obliczenia. Dodatkowo, dla trudnych problemów optymalizacyjnych o wielu wymiarach, potrzebna liczba cząstek może być bardzo znacząca, co skłania do poszukiwania poprawy wydajności i przyspieszenia obliczeń

właśnie na drodze zrównoleglenia czy rozproszenia. Oczywiście wraz z zaletami programowania równoległego pojawiają się charakterystyczne dla niego problemy, które należy rozwiązać, takie jak: skalowalność, synchroniczna i asynchroniczna implementacja, spójność i komunikacja sieciowa.

W pracach [32] oraz [29] przedstawiono po krótkce niektóre z zaproponowanych do tej pory rozwiązań problemu paralelizacji algorytmu PSO. Większość z nich oparta jest na klastrach komputerów wymieniających się między sobą wiadomościami. Niektóre z implementacji używają popularnego standardu OpenMP. Warto wspomnieć, że analiza przeprowadzona w [33] wskazuje, że pewien rodzaj równoległego PSO, w którym cząstki aktualizuje się grupami, nie zawsze wymagać większej liczby operacji niż implementacja sekwencyjna, w której cząstki aktualizowane są jedna po drugiej, zajmując przy tym mniej czasu.

PSO jest w naturalny sposób równoległe na poziomie algorytmicznym, jednakże zaimplementowanie tej równoległości nie jest już takie oczywiste – wśród głównych problemów, z którymi należy się zmierzyć, są komunikacja i równoważenie obciążenia, przy czym zagadnienia te są ze sobą powiązane. W algorytmie PSO największym kosztem obliczeniowym jest zazwyczaj ewaluacja funkcji celu dla każdej cząstki. Jeżeli ewaluacja ta jest relatywnie kosztowna, koszt komunikacji można zaniedbać i pierwszoplanowym problemem staje się równoważenie obciążenia między węzłami obliczeniowymi. W przeciwnym przypadku względnie niskiego kosztu ewaluacji funkcji celu, koszt komunikacji może dominować obliczenia.

Wśród stosowanych podejść do problemu zrównoleglenia algorytmu PSO można wyróżnić dwa główne: podejście synchroniczne i asynchroniczne. W pierwszym z nich wszystkie procesory czekają na zakończenie ewaluacji funkcji celu dla wszystkich cząsteczek przed przejściem do kolejnej iteracji. Przeprowadzono wiele eksperymentów, które sugerują, że efektywność zrównoleglenia (parallel efficiency) spada wraz z liczbą procesorów i jest daleka od idealnej 100 procent. W celu równoważenia nieefektywności wynikającej z nierównego rozłożenia obliczeń, zaproponowano podejście asynchroniczne - pozwala ono każdemu procesorowi (procesowi, wątkowi) przejść niezależnie do kolejnej iteracji po ukończeniu obliczania funkcji

celu. Wyniki eksperymentów pokazały znaczący wzrost efektywności w porównaniu z wersją synchroniczną.

Jednym ze sposobów zrównoleglenia obliczeń jest wykorzystanie w tym celu procesorów graficznych (GPU). Skupimy się tutaj głównie na procesorach graficznych wspierających architekturę CUDA (Compute Unified Device Architecture) [35] stworzoną przez firmę NVIDIA. Platforma ta zyskała bardzo dużą popularność jako prostsza i bardziej intuicyjna alternatywa dla np. standardu OpenCL.

W artykule [36] opisana została paralelizacja standardowego PSO (SPSO) na GPU z 11-krotnym przyspieszeniem względem CPU. Implementacja ta wykorzystuje topologię pierścienia, a więc umożliwia każdej cząstce komunikację z jedynie dwoma sąsiadami. Dzięki temu nie ma potrzeby równoległego poszukiwania najlepszego sąsiada podczas aktualizacji prędkości, co ogranicza konieczność komunikacji między wątkami. Z kolei rozwiązanie opisane w [34] bazuje na pomysśle utworzenia wątków GPU w liczbie równej liczbie cząstek roju pomnożonej przez liczbę wymiarów (co jednak niesie ze sobą konieczność ograniczenia maksymalnej liczby wymiarów rozważanego problemu optymalizacyjnego), który to w pewnych szczególnych przypadkach daje nawet 50-krotne przyspieszenie względem implementacji sekwencyjnej. Innym ciekawym rozwiązaniem jest zaproponowana w [37] hybrydyzacja algorytmu PSO z algorytmem „pattern search”, która na GPU może osiągnąć lepsze wyniki niż każdy z tych algorytmów z osobna. Artykuł [31] bada zależność między wydajnością algorytmu PSO, a wielkością bloku wątków, uzyskując przy tym 43-krotny zysk.

Rozdział 3

Zaimplementowany system optymalizacyjny

Stworzony przez nas system optymalizacyjny składa się z jednakowych węzłów obliczeniowych, które w celu rozproszenia obliczeń mogą łączyć się w klaster, a przy dostępnych zasobach, są w stanie zrównoleglić obliczenia pomiędzy wiele wątków procesora oraz uruchomić je na karcie graficznej.

3.1 Architektura węzła obliczeniowego

Poniżej przedstawiamy najważniejsze klasy składające się na pojedynczy węzeł obliczeniowy.

3.1.1 MachineManager

Klasa zarządzająca działaniem maszyny, umożliwiającą rozpoznanie dostępnej na komputerze liczby procesorów. Jest obiektem zaimplementowanym według wzorca fasada, odpowiedzialnym za rozpoczęcie obliczeń algorytmu (metoda *StartPsoAlgorithm*) oraz pobranie wyników (metoda *GetResult* zwracająca obiekt klasy *ParticleResult*), zaś za pomocą metody *Register* możemy podłączyć maszynę do klastra obliczeniowego. Obiekt klasy *MachineManager* posiada tablicę *VCpuManager*ów,

których liczba domyślnie odpowiada liczbie wątków procesora, lecz programista może ją ręcznie zmienić.

3.1.2 VCpuManager

Klasa ta ma dwa zadania: po pierwsze uruchamia obliczenia jednego roju cząstek PSO, po drugie, reprezentuje pojedynczy węzeł klastra obliczeniowego i odpowiada za nawiązanie połączenie z resztą klastra. W celu oddzielenia infrastruktury sieciowej od logiki dotyczącej algorytmu PSO wykorzystuje ona interfejs *IPsoManager*, który zarządza warstwą logiczną klastra PSO i umożliwia m.in wywołanie metody *Run*, uruchamiającej obliczenia oraz obiekt klasy *NetworkNodeManager* odpowiedzialny za komunikację pomiędzy węzłami, korzystający z technologii WCF. *VCpuManager* rozpoczyna oraz kończy nasłuchiwanie na portach TCP. Stosując wzorzec obserwator zaimplementowany za pomocą funkcjonalności wydarzeń języka C# obsługuje nowe zapytania przesyłane przez pozostałe węzły zbudowanego klastra.

3.1.3 IPsoController

Jest to interfejs odpowiedzialny za uruchomienie obliczeń algorytmu PSO na podstawie podanych parametrów wykonania.

Następujące informacje konfiguracyjne, zamknięte w klasie *PsoParams*, potrzebne są do uruchomienia algorytmu:

- Epsilon - dokładność obliczeń algorytmu
- FunctionParams - parametry funkcji celu:
- Iterations - maksymalna liczba iteracji do przeprowadzenia na wątku CPU
- IterationsLimitCondition - wartość logiczna stanowiąca o zatrzymaniu algorytmu po wykonaniu danej liczby iteracji

- *ParticleIterationsToRestart* - liczba iteracji bez poprawy położenia pojedynczej cząstki, po której dana cząstka zostanie zrestartowana, czyli jej położenie zostanie wylosowane, a najlepsza znana wartość zapomniana
- *Particles* - lista typów oraz ilości cząstek do wykorzystania
- *PsoIterationsToRestart* - liczba iteracji bez poprawy najlepszego wyniku wszystkich cząstek, po której zakończone zostanie działanie algorytmu
- *TargetValue* - wartość ekstremalna funkcji celu, do której dąży wynik algorytmu
- *TargetValueCondition* - wartość logiczna stanowiąca, czy zatrzymać algorytm po zbliżeniu się na odległość *Epsilon* od *TargetValue*

3.2 Implementacja algorytmu

Algorytm PSO w wariencie uruchamianym na CPU został przez nas zaimplementowany w języku C#, co różni się od wstępnych założeń projektu, w których zakładaliśmy jego implementację w języku C++ oraz opakowanie go klasą w technologii .NET. Początkowo algorytm został napisany zgodnie z założeniami systemu - algorytm w języku C++ natomiast jego opakowanie w języku C++/CLI. Jest to język oparty na C++ pozwalający na proste połączenie kodu natywnego C++ z kodem zarządzanym platformy .NET. Rozwiązanie takie było wystarczające dla tego modułu systemu, lecz okazało się być nadmierną komplikacją w budowie klastra oraz wykorzystaniu benchmarków BBOB.

W pierwszym przypadku utrudnieniem była konieczność użycia klas udostępniających usługi WCF w natywnym kodzie C++, aby z perspektywy cząstek komunikacja między zdalnymi węzłami przebiegała w sposób jednolity do interakcji z cząstkami lokalnymi, następnie opakowanie tych cząstek w klasy języka C#.

Drugi przypadek konfliktował z jednym z założeń, na podstawie którego programista mógł definiować funkcje celu w języku C#. W celu spełnienia tego przekazywaliśmy funkcje z maszyny wirtualnej .NET do środowiska natywnego w celu ich

wywołania przez cząstki. Wykorzystanie benchmarków wymagałoby dodatkowo odwrotnego opakowywania funkcji napisanych w C i rodziło dodatkowe komplikacje z odtwarzaniem funkcji dla zdalnych węzłów.

3.2.1 Algorytm

Zaimplementowanie algorytmu w języku C# rozwiązało powyższe problemy oraz dodatkowo umożliwiło dużo przejrzystsze przekazywanie parametrów algorytmu oraz ułatwiało rozszerzanie algorytmu o nowe typy cząstek. Sam algorytm stworzony został przez nas w taki sposób, aby był jak najbardziej generyczny ze względu na rodzaj cząstek, przez co jest odpowiedzialny jedynie za przeprowadzenie odpowiednich kroków algorytmu, których dokładna implementacja zawarta jest w cząstkach. W pętli sprawdzane są warunki stopu oraz liczone iteracje od ostatniej poprawy algorytmu, jeśli któryś z warunków zostanie spełniony, lub iteracje bez poprawy osiągną ustalony limit działanie algorytmu zostaje przerwane.

```
public IState<double[],double[]> Run(CancellationToken token)
{
    foreach(var particle in _particles)
    {
        var j = randomParticleIndex();
        particle.Transpose(_fitnessFunction);
        particle.UpdateNeighborhood(_particles);
        particle.InitializeVelocity(_particles[j]);
    }
    var _currentBest = GetCurrentBest();
    _globalBest = new ParticleState(_currentBest.Location,
    _currentBest.FitnessValue);
    while (_conditionCheck())
    {
        foreach (var particle in _particles)
        {
```

```
        particle.Transpose(_fitnessFunction);
    }
    foreach (var particle in _particles)
    {
        particle.UpdateVelocity(_globalBest);
    }
}
return _fitnessFunction.BestEvaluation;
}
```

3.2.2 Częstka PSO

W naszym systemie cząstki są implementacją interfejsu `IParticle`, który zawiera właściwości potrzebne do identyfikacji cząstki, określenia jej położenia obecnego i najlepszego dotychczas odwiedzonego oraz metody aktualizujące sąsiedztwo cząstki, jej prędkość i położenie. Od implementacji cząstek zależy topologia roju oraz sposób w jaki poruszają się po przestrzeni poszukiwań.

StandardParticle

W celu dokładniejszego opisu cząstki przedstawimy przykładową cząstkę zbudowaną zgodnie z założeniami SPSO. Rozszerza ona abstrakcyjną klasę `Particle`, która zawiera elementy wspólne pomiędzy różnymi wersjami cząstek. Topologia jest ustalana poprzez funkcję `UpdateNeighborhood`, która wybiera sąsiadów, z którymi się komunikuje dana cząstka.

```
public override void UpdateNeighborhood(IParticle[] allParticles)
{
    Neighborhood = allParticles
        .Where(particle => particle.Id != Id)
        .ToArray();
}
```

Funkcja `UpdateVelocity` aktualizuje prędkość cząstki uwzględniając najlepsze położenia swoje oraz cząstek sąsiednich. W danym przykładzie, aby uzyskać wzrost wydajności algorytmu pojedyncza cząstka nie szuka najlepszej pozycji sąsiadów tylko otrzymuje globalną najlepszą wartość jako parametr funkcji. Jest to wartość identyczna z najlepszą wartością sąsiedztwa cząstki, gdyż w tym przypadku jest nim cały rój.

```
public override void UpdateVelocity(IState<double[], double[]> globalBest)
{
    // 1. get vectors to personal and global best
    var toPersonalBest = Metric
        .VectorBetween(CurrentState.Location, PersonalBest.Location);

    var toGlobalBest =
        Metric
            .VectorBetween(CurrentState.Location, globalBest.Location);

    var phi1 = RandomGenerator.RandomVector(CurrentState.Location.Length, 0, 1);
    var phi2 = RandomGenerator.RandomVector(CurrentState.Location.Length, 0, 1);

    // 2. multiply velocity by Omega and add toGlobalBest and toPersonalBest
    Velocity = Velocity.Select((v, i) => v * Constants.OMEGA + phi1[i] * toGlobalBest + phi2[i] * toPersonalBest);
}
```

Funkcja `Transpose` jest odziedziczona z klasy `Particle`, ponieważ jest wspólna dla stosowanych przez nas typów cząstek. Zmienia ona obecne położenie cząstki licząc przy tym liczbę iteracji bez poprawy wartości funkcji celu. Jeśli przekroczy ona ustalony w konfiguracji algorytmu limit, cząstka jest restartowana, czyli losowane jest jej położenie początkowe oraz zapominane najlepsze dotychczas rozwiązanie.

```
public virtual void Transpose(IFitnessFunction<double[], double[]> function)
{
    double[] newLocation;
```

```
    if (_sinceLastImprovement == _iterationsToRestart)
    {
        newLocation = randomLocation()
        _sinceLastImprovement = 0;
    }
    else
    {
        newLocation = GetClampedLocation(CurrentState.Location + Velocity);
    }
    var newVal = function.Evaluate(newLocation);
    var oldBest = PersonalBest;
    CurrentState = new ParticleState(newLocation, newVal);

    if (Optimization.IsBetter(newVal, PersonalBest.FitnessValue) < 0)
    {
        PersonalBest = CurrentState;
        _sinceLastImprovement = 0;
    }
    else
    {
        _sinceLastImprovement++;
    }
}
```

3.2.3 Funkcja celu

Funkcje celu są obiektami implementującymi interfejs `IFitnessFunction`. Udostępniają one liczbę przeprowadzonych ewaluacji funkcji, najlepszą ewaluację oraz informacje o wymiarze przestrzeni poszukiwań i wartości funkcji. Funkcje są tworzone przez fabrykę funkcji `FunctionsFactory` na podstawie przekazanych do niej parametrów. W celu stworzenia odpowiedniej funkcji klient musi przekazać odpow-

wiednią nazwę funkcji. Jest to szczególnie ważne w przypadku funkcji benchamrkowych, gdyż w nazwie zakodowany jest rodzaj funkcji oraz jej odpowiednia instancja. Dodatkowo fabryka funkcji umożliwia zapisanie danej funkcji w cache'u. Konieczność zastosowania pamięci tymczasowej była wynikiem integracji systemu z biblioteką COCO, która wymagała użycia jednej instancji funkcji celu w całym systemie.

Parametry te przekazywane są do metody Run.

PsoController

Klasa PsoController implementująca IPsoController kontroluje wykonywanie obliczeń algorytmu PSO za pomocą obiektu `_algorithm` klasy PsoAlgorithm oraz obiektu `_cudaAlgorithm` klasy GenericCudaAlgorithm.

Na podstawie informacji zawartych w obiekcie PsoParams, PsoController tworzy cząstki PSO (obiekty implementujące interfejs IParticle) oraz funkcję celu (obiekt implementujący interfejs IFitnessFunction), które służą następnie do uruchomienia właściwych obliczeń, co następuje w funkcji StartAlgorithm wywoływanej z funkcji Run:

```
_algorithm.Run(CancellationToken token)
```

W przypadku, gdy na maszynie dostępna jest karta graficzna spełniająca wymagania, na podstawie dostarczonych parametrów, w funkcji PrepareCudaAlgorithm przygotowywane jest również wywołanie algorytmu na GPU, zaś samo uruchomienie następuje w funkcji Run.

Obliczenia na CPU jak i GPU wykonywane są asynchronicznie, z możliwością przerwania ich w dowolnym momencie.

3.2.4 IPsoManager

Interfejs IPsoManager odpowiada za komunikację pomiędzy rojami cząstek PSO znajdującymi się na różnych węzłach klastra obliczeniowego. Interfejs składa się z następujących metod:


```
void UpdatePsoNeighborhood(  
NetworkNodeInfo[] allNetworkNodes,  
NetworkNodeInfo currentNetworkNode);  
  
Uri[] GetProxyParticlesAddresses();  
  
ProxyParticle[] GetProxyParticles();  
  
event CommunicationBreakdown CommunicationLost;
```

Komunikacja pomiędzy rojami cząstek zostanie opisana w dalszej części pracy.

PsoRingManager

PsoRingManager odpowiada za logiczne połączenie rojów na różnych węzłach obliczeniowych w pierścień (każdy rój posiada dwa roje sąsiednie).

3.2.5 NetworkNodeManager

Jest to klasa odpowiadająca za infrastrukturę sieciową klastra. Najważniejszymi polami klasy są:

```
private readonly int _tcpPort;  
private ServiceHost _tcpHost;  
public NodeService MyNodeService { get; set; }  
public List<NodeServiceClient> NodeServiceClients
```

Po otwarciu za pomocą metody StartTcpNodeService serwisu _tcpHost nasłuchującego na porcie _tcpPort na połączenie TCP, NetworkNodeManager gotowy jest na przyjęcie połączenia TCP od innych węzłów.

Interfejs sieciowy udostępniany przez węzeł obliczeniowy przedstawia klasa NodeService opisana poniżej.

Komunikacja w drugą stronę zachodzi przy użyciu obiektów klasy `NodeServiceClient`, które odpowiadają obiektom `NodeService` znajdującym się na zewnętrznych węzłach.

3.2.6 `INodeService`

Jest to interfejs sieciowy dla węzła obliczeniowego.

```
[ServiceContract]
public interface INodeService
{
    [OperationContract]
    void UpdateNodes(NetworkNodeInfo[] nodes);

    [OperationContract]
    NetworkNodeInfo[] Register(NetworkNodeInfo source);

    [OperationContract]
    void Deregister(NetworkNodeInfo brokenNodeInfo);

    [OperationContract]
    void StartCalculation(
        PsoParameters parameters, NetworkNodeInfo mainNodeInfo);

    [OperationContract]
    ParticleState StopCalculation();

    [OperationContract]
    void CalculationsFinished(
        NetworkNodeInfo source, ParticleState result);

    [OperationContract]
    void CheckStatus();
}
```

```
}
```

Dokładniejszy schemat komunikacji między węzłami klastra zostanie dokładniej opisany w późniejszym rozdziale.

NodeService

Klasa ta stanowi implementację interfejsu sieciowego węzła obliczeniowego.

NodeServiceClient

Jest to klasa, za pomocą której obiekt implementujący `INodeService` jednego węzła wykonuje operacje na interfejsie sieciowym innego węzła. Technologia WCF umożliwia zaimplementowanie klienta usług sieciowych w prosty sposób - wystarczy dokonać następującej inicjalizacji:

```
Address = new EndpointAddress(tcpAddress);  
Binding = new NetTcpBinding(SecurityMode.None);  
ChannelFactory = new ChannelFactory<INodeService>(Binding);  
INodeService Proxy = ChannelFactory.CreateChannel(Address);
```

Na obiekcie Proxy można w tej chwili wykonywać wszystkie operacje, które implementuje interfejs sieciowy węzła o adresie tcpAddress.

3.3 Komunikacja w klastrze obliczeniowym

Komunikacja między węzłami przebiega na dwóch opisanych poniżej poziomach. Fizyczny sposób połączenia węzłów w klastrze jest oddzielony od logicznego powiązania rojów PSO, dzięki czemu system umożliwia zaimplementowanie różnych topologii rojów bez konieczności ingerencji w strukturę połączeń pomiędzy węzłami.

3.3.1 Komunikacja pomiędzy węzłami obliczeniowymi

Klaster obliczeniowy jest oparty na topologii pełnej siatki, w której każdy węzeł połączony jest z wszystkimi pozostałymi i dołączany jest do sieci niezależnie od pozostałych. Każdy z węzłów jest równoważny wszystkim pozostałym, to znaczy nie istnieje centralny serwer zarządzający pracą sieci, dzięki czemu obliczenia mogą być zapoczątkowane przez dowolny z nich.

Komunikacja realizowana jest za pomocą usług WCF korzystających z protokołu sieciowego TCP. Modułem odpowiadającym za komunikację na poziomie klastra jest w każdym węźle `INodeService`, który, jako serwis sieciowy, opisany jest atrybutem `[ServiceContract]`. Udostępnia on następujące usługi, z których każda posiada atrybut `[OperationContract]` i można ją wywołać zdalnie używając `NodeTcpClient'a`.

`UpdateNodes` -

`Register` - dodaje węzeł wywołujący zdalnie tę usługę do listy znanych węzłów węzła, który wykonuje usługę.

`Deregister` - usuwa węzeł wywołujący zdalnie tę usługę z listy znanych węzłów węzła ją wykonującego.

`StartCalculation` - nakazuje węzłowi rozpoczęcie obliczeń na podstawie podanych parametrów algorytmu.

`StopCalculation` - nakazuje przerwanie obliczeń i zwrócenie informacji o ich dotychczasowym wyniku.

`CalculationsFinished` -

Typowy schemat komunikacji pomiędzy węzłami jest przedstawiony poniżej.

[[[Diagram sekwencyjny]]]

Awaria

Obliczenia są niezależne od połączeń pomiędzy węzłami klastra, dzięki czemu po awarii jednego z nich obliczenia nie są przerywane, a topologia sąsiedztwa rojów algorytmu PSO jest dostosowywana do dostępnych w danym momencie węzłów.

3.3.2 Komunikacja na poziomie algorytmu PSO

Komunikacja pomiędzy rojami PSO na różnych węzłach obliczeniowych zarządzana jest przez interfejs IPsoManager, który posiadając informacje o połączonych jednostkach, definiuje sąsiedztwa pomiędzy nimi. W przypadku naszego klastra implementuje on strukturę pierścienia, w której każdy węzeł posiada dwóch sąsiadów. Połączenia pomiędzy rojami działającymi wewnątrz jednej maszyny, ale korzystające z różnych wątków procesora przebiegają w ten sam sposób co połączenia między różnymi maszynami. Wszystkie wątki, a dokładniej VCpuManager'y wszystkich maszyn tworzą jeden pierścień. Komunikacja pomiędzy rojami jest dwustronna. Z perspektywy cząstek wewnątrz roju, sąsiedni rój jest reprezentowany przez tzw. cząstkę proxy, która również implementuje interfejs IParticle, przez co nie wymaga oddzielnego traktowania - komunikacja z zewnętrznymi rojami jest przezroczysta dla innych cząstek i samego algorytmu.

Cząstka proxy posiada odpowiadającą jej, zespoloną z nią cząstkę [[[proxy?]]] z innego roju.

[[[Jakiś obrazek]]]

Cząstka proxy nie wykonuje ewaluacji funkcji celu, a przekazuje jedynie najlepsze znane położenie cząstki z nią zespolonej pomiędzy rojami. Do własnego roju podaje najlepszy wynik znany cząstce proxy z roju sąsiedniego, natomiast do niej przekazuje informacje od lokalnej cząstki. Odbywa się to w momencie, gdy cząstka proxy pytana jest o najlepsze znane sobie położenie, jednakże nie za każdym razem odpytywana jest cząstka z sąsiedniego roju - dzieje się tak tylko co pewną liczbę iteracji określoną parametrem RemoteCheckInterval, który domyślnie przyjmuje wartość 200, a wprowadzony został w celu ograniczenia spadku wydajności, który zauważyliśmy, gdy komunikacja pomiędzy rojami zachodziła zbyt często.

```
public override ParticleState PersonalBest
{
    get
    {
        if (_getBestCounter == RemoteCheckInterval)
```

```
        {  
            _getBestCounter = 0;  
            _proxyManager.GetRemoteBestState();  
        }  
        _getBestCounter++;  
        return _proxyManager.GetBestState();  
    }  
}
```

Poza informacjami przekazywanymi przez cząstki proxy, po zakończeniu obliczeń każdy z węzłów rozsyła do pozostałych informację o najlepszym znalezionym przez siebie wyniku.

Cząstka proxy komunikuje się z innymi cząstkami proxy za pomocą jeszcze jednej warstwy pośredniczącej - ProxyManagera.

ProxyManager jest odpowiednikiem klasy NetworkNodeManager dla komunikacji między cząstkami - i podobnie jak NetworkNodeManager posiadający interfejs sieciowy w postaci INodeService, tak ProxyManager posiada IParticleService, który jest interfejsem sieciowej komunikacji między cząstkami i udostępnia jedną operację:

GetBestState - zwraca najlepszą znaną cząstce pozycję, czyli najlepszą znaną wartość wśród cząstek jej sąsiedztwa.

Wykrywanie awarii

ProxyManager jest jednocześnie miejscem, w którym zachodzi wykrywanie awarii. W przypadku, gdy cząstka sąsiedniego roju, z którym próbujemy się połączyć, wielokrotnie nie odpowiada na zapytanie, zgłaszane jest wydarzenie Communication-Breakdown, a jednocześnie jako wynik zapytania zwracana jest domyślna wartość.

```
try  
{  
    var s = _particleClient.GetBestState();  
    _particleService.UpdateBestState(s);  
    _communicationErrorCount = 0;  
}
```

```
        return s;
    }
    catch
    {
        _communicationErrorCount++;
        if (CommunicationBreakdown != null && _communicationErrorCount ==
        {
            CommunicationBreakdown();
        }
        return new ParticleState();
    }
}
```

3.4 Implementacja PSO na GPU

3.4.1 Porównanie implementacji PSO na CPU i GPU

Implementacja na CPU, w uproszczeniu, wygląda następująco:

```
while(condition) {  
    foreach particle in particles:  
        particle.Transpose  
  
    foreach particle in particles:  
        particle.UpdateVelocity  
}
```

Zatem dopóki nie zostanie pewien warunek zakończenia, dla każdej cząstki aktualizujemy jej pozycję, a następnie dla każdej cząstki aktualizujemy jej prędkość.

Z kolei implementację algorytmu dla GPU, również w uproszczeniu, można przedstawić jako:

```
while(condition) {  
    RunTransposeKernel(particles)  
    RunUpdateVelocityKernel(particles)  
}
```

Różnica polega na tym, że aktualizacja prędkości lub pozycji odbywa się w jednym kroku dla wszystkich cząstek. Technologia CUDA umożliwia nam działanie w właśnie takim modelu - wykonanie tej samej instrukcji dla wielu danych wejściowych (SIMD - Single Instruction Multiple Data).

3.4.2 Krótkie wprowadzenie do architektury CUDA

Opis części programu na GPU wymaga krótkiego wprowadzenia do organizacji wątków i pamięci w architekturze CUDA.

Podstawową jednostką obliczeniową w architekturze CUDA jest pojedynczy wątek GPU. Dzięki dostępowi do tysięcy wątków na procesorze graficznym możliwe

jest stosowanie modelu SIMT (Single Instruction Multiple Threads). Każdy wątek ma dostęp do swojej lokalnej pamięci wątku.

Wątki zorganizowane są w wielowymiarowe bloki wątków. W zależności od wersji architektury, blok może zawierać 512 lub 1024 wątków. Poza grupowaniem wątków każdy blok wątków ma swój segment pamięci zwany pamięcią dzieloną bloku. Dostęp do tej pamięci jest dużo szybszy niż do pamięci globalnej, do której dostęp ma dowolny wątek obliczeniowy. [35]

3.4.3 Implementacja algorytmu

Implementacja wsparcia dla obliczeń na procesorze graficznym składa się z dwóch modułów: NativeGPU oraz ManagedGPU. Integracja między tymi modułami, napisanymi w różnych językach programowania, odbywa się poprzez bibliotekę managedCuda.

managedCuda

Biblioteka ta umożliwia na wykorzystanie potencjału kart graficznych wspierających technologię CUDA z poziomu platformy .NET. Daje ona dostęp do informacji o dostępnym sprzęcie, ułatwia zarządzanie pamięcią karty graficznej oraz umożliwia wykonywanie kerneli CUDA z poziomu kodu C#.

NativeGPU

Moduł zawiera implementację algorytmu PSO oraz implementacje optymalizowanych funkcji w języku CUDA.

Zaimplementowane funkcje nie są wykorzystywane bezpośrednio przez pozostałe moduły. Biblioteka managedCuda wykorzystuje format pośredni PTX. Pliki z kernelami kompilowane są do tego pośredniego formatu, a następnie są wczytywane do pamięci trakcie uruchomienia obliczeń z poziomu C#. W ten sposób możemy wykorzystać te funkcje w prosty sposób z poziomu reszty systemu.

Kod podzielony jest na pliki zawierające kernele będące częścią algorytmu PSO, implementacje optymalizowanych funkcji oraz pomocnicze funkcje do operacji na wektorach i macierzach.

update_velocity_kernel.cu

Zawiera kernel odpowiadający za wykonanie aktualizacji prędkości cząstek (pierwszy krok w głównej pętli algorytmu PSO). W przeciwieństwie do kernela, który aktualizuje lokalizacje cząstek, ten kernel jest niezależny od optymalizowanej funkcji ponieważ nie ewaluje on optymalizowanej funkcji, przez co ma stały zestaw argumentów.

Pliki dla optymalizowanych funkcji

Każdy z plików odpowiada różnej optymalizowanej funkcji i zawiera w sobie implementację tej funkcji w języku CUDA, kernel który inicjalizuje wartości specyficzne dla danej funkcji celu (współczynniki, macierze obrotów) oraz implementację kernela odpowiadającego za wykonanie aktualizacji lokalizacji cząstek (drugi krok w głównej pętli algorytmu PSO). Kernel aktualizujący lokalizacje różni się dla różnych optymalizowanych funkcji ze względu na różne parametry jakie te funkcje przyjmują.

bbob_generators.cuh

Zawiera implementacje przekształceń na wektorach i macierzach, które używane są w benchmarku BBOB, a także wiele prostych funkcji pomocniczych do działań na wektorach które były przydatne do implementacji funkcji celu lub algorytmu PSO.

ManagedGPU

Moduł ten udostępnia API w języku C# pozwalające na uruchomienie obliczeń na karcie graficznej oraz synchronizację obliczeń na GPU z obliczeniami na CPU.

ManagedGPU.GpuController

Klasa odpowiadająca za wstępną konfigurację obliczeń na GPU oraz umożliwiająca sprawdzenie możliwości maszyny obliczeniowej. Metody udostępniane przez tę klasę:

- **AnySupportedGpu** - sprawdza, czy na maszynie jest dostępna karta graficzna wspierająca technologię CUDA w odpowiedniej wersji. Dostęp do informacji na temat sprzętu umożliwia biblioteka `managedCuda`.
- **Setup** - korzystając z otrzymanych parametrów tworzy reprezentację algorytmu (`CudaAlgorithm`) oraz cząstkę proxy (`CudaParticle`) która może być wykorzystana do synchronizacji obliczeń CPU i GPU. Zwrócona para obiektów jest wystarczająca do uruchomienia obliczeń na GPU samodzielnie lub w synchronizacji z CPU.

ManagedGPU.CudaAlgorithm

Klasa która reprezentuje algorytm optymalizujący pewną funkcję. Jest to klasa abstrakcyjna - zawiera główną pętlę algorytmu i metody umożliwiające kontrolę wykonania algorytmu. Ważne metody:

- **Initialize** - metoda która przygotowuje zasoby do uruchomienia algorytmu na GPU. Wczytuje do pamięci odpowiedni dla optymalizowanej funkcji moduł PTX z funkcjami CUDA oraz alokuje pamięć na procesorze graficznym. Algorytm musi być zainicjalizowany przed uruchomieniem.
- **Run** - uruchamia obliczenia na karcie graficznej. Parametrem wywołania tej metody jest token, który służy do przerywania obliczeń na GPU na żądanie.

Metoda ta zawiera w sobie główną pętlę algorytmu PSO na GPU w której wywoływane są kernele CUDA

- `Dispose` - metoda zwalniająca zasoby zaalokowane przez instancję algorytmu. Powinna zostać wywołana po zakończeniu obliczeń.

ManagedGPU.CudaParticle

Klasa która umożliwia synchronizację między rojem PSO na CPU i rojem na GPU. Rozszerza ona klasę **Particle**, zatem może być traktowana jako zwykła cząstka z poziomu algorytmu CPU - umieszczenie takiej cząstki w roju na CPU symuluje sąsiedztwo między cząstkami na CPU i na GPU.

Jak widać na schemacie ??, wykonywaniem obliczeń na GPU zarządza wątek CPU.

Przed uruchomieniem obliczeń musi dojść do inicjalizacji algorytmu - kernele CUDA zostaną załadowane do pamięci oraz zostanie zaalokowana i zainicjowana pamięć na karcie graficznej.

Uruchomienie algorytmu powoduje wystartowanie pętli, która działa przez zadaną liczbę iteracji lub dopóki nie zostanie wysłany sygnał z zewnątrz aby przerwać obliczenia (sygnał wysyłany jest przez Token będący parametrem metody uruchamiającej algorytm).

W każdej iteracji tej pętli uruchamiane są dwa kernele: kernel aktualizujący prędkości cząstek oraz kernel aktualizujący pozycje cząstek. Po wykonaniu tych kroków algorytmu PSO może dojść do synchronizacji z CPU - synchronizacja zachodzi co ustaloną liczbę iteracji na GPU (w przypadku naszych testów była to empirycznie wyznaczona wartość 100 iteracji).

Organizacja cząstek na GPU

W roju cząstek na GPU stosujemy topologię pierścienia - każda z cząstek ma 2 sąsiadów, cząstkę o indeksie o jeden mniejszym i cząstkę o indeksie o jeden większym. Upraszcza to kernele po stronie CUDA i jest spójne z przyjętą topologią w pozostałej części systemu.

Przy wykonaniu kerneli CUDA każda cząstka obsługiwana jest przez jeden wątek. Potencjalnie jest możliwe zwiększenie stopnia zrównoleglenia poprzez delegowanie pojedynczego wątku na obsługę pojedynczego wymiaru pozycji/prędkości cząstki, ale taka organizacja wątków nie jest możliwa dla wszystkich optymalizowanych funkcji. Dlatego, dla spójności w naszej implementacji, przyjęliśmy jeden wątek GPU na jedną cząstkę PSO.

Synchronizacja CPU i GPU

Roje na GPU i CPU synchronizowane są poprzez specjalny typ cząstki CudaParticle. Cząstka tego typu wygląda różnie w zależności od strony ją obserwującej.

Jeśli cząstka `CudaParticle` znajdzie się w roju cząstek na CPU, zapytanie o jej cechy (np. lokalizację) zwróci cechy pewnej cząstki z roju na GPU (z czasu ostatniej synchronizacji).

Od strony algorytmu na GPU cząstka ta ma cechy najlepszej cząstki z roju na CPU. Jednak wątek zarządzający algorytmem na GPU nie korzysta bezpośrednio z cząstki `CudaParticle` - zamiast tego `CudaParticle` i odpowiadający jej `CudaAlgorithm` współdzielą obiekt `CudaProxy` poprzez który udostępniają swój stan i pobierają stan z drugiej strony.

Synchronizacja odbywa się co ustaloną liczbę iteracji algorytmu na GPU - w iteracji synchronizacji stan pierwszej cząstki z cząstek GPU zostaje pobrany z pamięci GPU i przekazywany jest do cząstki `CudaParticle`, a następnie stan najlepszej cząstki z CPU nadpisuje stan pierwszej cząstki na GPU.

Dodawanie własnych funkcji optymalizacyjnych

Architektura naszego rozwiązania pozwala na dodawanie nowych funkcji do optymalizacji na GPU.

Aby dodać nową funkcję należy:

- Dodać implementację funkcji w języku CUDA wraz z odpowiednim dla niej kernelem aktualizującym pozycję cząstki.
- Rozszerzyć klasę `CudaAlgorithm` dla implementowanej funkcji rozszerzając inicjalizację pamięci o dodatkowe współczynniki dla funkcji celu (jeśli potrzebne) oraz nadpisać wywołanie kernela aktualizującego pozycję cząstki.
- Dodać nową klasę do rejestru algorytmów `ManagedCuda.CudaAlgorithmFactory` z odpowiednim identyfikatorem (identyfikatory 1-24 zarezerwowane są dla funkcji z benchmarku BBOB).

Rozdział 4

Platforma COCO

4.1 Testowanie algorytmów optymalizacyjnych

Jednym z celów naszej pracy było przetestowanie jakości implementacji algorytmu PSO na stworzonym przez nas klastrze obliczeniowym. Do tego zadania postanowiliśmy wybrać platformę COCO [3].

Benchmarkowanie algorytmów optymalizacyjnych polega w skrócie na uruchomieniu algorytmu na przygotowanym wcześniej zbiorze problemów i zebraniu oraz przedstawieniu wyników. Nie jest to jednak tak trywialne zadanie, na jakie mogłoby wyglądać na pierwszy rzut oka m.in. ze względu na częstą trudność w dokładnej interpretacji otrzymanych wyników czy też porównaniu ich z innymi.

Odpowiedzią na te trudności jest „COCO framework” umożliwiający zautomatyzowanie procedury benchmarkowania algorytmów optymalizacyjnych. Ideą przyświecającą twórcom COCO było stworzenie środowiska zapewniającego wszystkie niezbędne do przeprowadzenie testów funkcjonalności oraz możliwość porównania danych i wyników zebranych przez różne zespoły uczonych na przestrzeni lat używających tego frameworka.

Funkcje do benchmarkowania, których używa COCO są jawne dla użytkownika, jednakże sam algorytm, który poddajemy testom, nie ma o nich żadnej wiedzy (działają na zasadzie black-box). Lista wszystkich funkcji używanych przez COCO dostępna jest w [3], jednakże my ograniczyliśmy się do pierwszych 24 funkcji (funkcje

bez szumu). Funkcje te są wybrane w ten sposób, aby dało się w jasny sposób zinterpretować na nich działanie algorytmu optymalizacyjnego. Dodatkowo nie posiadają one żadnych sztucznych regularności, które mogłyby zostać wykorzystane przez algorytm oraz są skalowalne ze względu na wymiar.

Co bardzo istotne, cały framework używa tylko jednej miary jakości algorytmu – tzw. runtime, czyli liczby ewaluacji funkcji celu potrzebnej do osiągnięcia zadanego wyniku, czyli znalezienia wartości funkcji celu odpowiednio bliskiej wartości optymalnej. Zalety takiego podejścia są opisane w [4]. Dodatkowo nie zakłada się ustalonego z góry maksymalnego budżetu, określonego liczbą ewaluacji funkcji celu – testy są „budget free”.

COCO framework składa się z zestawu benchmarków napisanych w języku C wraz z modułami odpowiedzialnymi za zbieranie (logowanie) wyników, ich obróbkę (skrypty Python przygotowujące odpowiednie wykresy) oraz prezentację danych (dokumenty html oraz pliki pdf generowane na podstawie przygotowanych szablonów LaTeX). Dodatkowo twórcy frameworka przygotowali interfejsy w językach C/C++, Java, Matlab/Octave, Python (stan w grudniu 2016) zapewniające obsługiwanie napisanych przez użytkownika algorytmów optymalizacyjnych w wymienionych językach. Niestety w chwili pisania niniejszej pracy, nie był dostępny oficjalny interfejs do języka C#, dlatego też zmuszeni byliśmy stworzyć własny.

4.2 Wrapper

Stworzony przez nas wrapper miał za zadanie umożliwić wywołanie funkcji biblioteki COCO (CocoLibrary.c) napisanych w języku C z poziomu naszej aplikacji w języku C#. Przy jego tworzeniu wzorowaliśmy się na dostępnym wrapperze dla języka Java wykorzystującym Java Native Interface (JNI), czyli framework umożliwiający komunikację z kodem napisanym w C/C++.

Pierwszym zadaniem było utworzenie klas języka C# odpowiadających strukturom języka C, które były wykorzystywane w bibliotece COCO. Te klasy, to:

Problem - odpowiada strukturze *coco_problem_t* i zawierająca dane konkretnego

problemu optymalizacyjnego.

Suite - odpowiada strukturze *coco_suite_t* i odpowiada całemu zestawowi problemów optymalizacyjnych.

Observer - odpowiada strukturze *coco_observer_t*, która służy do zbierania danych w czasie wykonywania testów.

Benchmark - klasa opakowująca, zawierająca w sobie obiekt klasy **Suite** oraz obiekt klasy **Observer**, umożliwiającą pobranie kolejnego problemu optymalizacyjnego z zestawu.

Klasy te stanowią interfejs benchmarku dla naszego programu - tworzone oraz inicjalizowane są w głównej pętli procedury testującej.

Między powyżej opisanymi obiektami a funkcjami z *CocoLibrary.c* pośredniczy jeszcze jedna warstwa, będąca prawdziwym wrapperem i zamknięta w klasie **CocoLibraryWrapper**.

4.2.1 Eksport\import funkcji z języka C

Jedną z części klasy **CocoLibraryWrapper** stanowią zaimportowane funkcje z *CocoLibrary.c*. Import funkcji jest konieczny, aby móc wywoływać je z poziomu języka C#. Na przykładzie jednej z funkcji zaprezentujemy sposób ich importu, który wymaga następującej deklaracji:

```
[DllImport (
    "CocoLibrary.dll",
    CallingConvention = CallingConvention.Cdecl)]

unsafe static extern char*
coco_problem_get_name(struct_pointer_t problem);
```

Umożliwia ona import funkcji o sygnaturze

```
const char *coco_problem_get_name(const coco_problem_t *problem)
```

znajdującej się w pliku *CocoLibrary.c*. Jak widać, aby zaimportować funkcję, należy wskazać jej źródło (w tym przypadku bibliotekę *CocoLibrary.dll*) oraz sposób wywołania (w tym przypadku *Cdecl*, czyli konwencja właściwa dla języka C).

Dodatkowo importowana funkcja musi zostać opatrzona modyfikatorami *unsafe* (umożliwiający korzystanie ze wskaźników w języku C#) oraz *extern* (wskazująca, że implementacja funkcji znajduje się w innym miejscu).

Sam import funkcji w naszej aplikacji nie wystarczy, aby uzyskać do nich dostęp. Konieczny był również ich eksport z samej biblioteki *CocoLibrary.c*, aby po jej skompilowaniu do *CocoLibrary.dll*, były one widoczne na zewnątrz. Eksport funkcji dokonuje się w następujący sposób:

```
__declspec(dllexport)
const char *coco_problem_get_name(const coco_problem_t *problem);
```

Zaimportowanych funkcji nie wywołujemy bezpośrednio, a za pomocą jeszcze jednej warstwy pośredniczącej, z której to korzystają już opisane wcześniej klasy **Problem**, **Suite**, **Observer** i **Benchmark**. Dodatkowa warstwa była konieczna, aby zapewnić, że funkcje eksportowane z *CocoLibrary.dll* wywoływane będą we właściwy sposób, jak np.:

```
public static unsafe String cocoProblemGetName(long problemPointer)
{
    char* str = coco_problem_get_name(problemPointer);
    return Marshal.PtrToStringAnsi((IntPtr)str);
}
```

gdzie należy dokonać odpowiedniego marshallingu argumentów

4.2.2 Wykorzystanie wrappera

Wrapper wykorzystany został w naszym programie oczywiście do uruchomienia zestawu testów benchmarka COCO. Inicjalizacja odpowiednich klas dokonywana jest w następujący sposób:

```
var suite = new Suite("bbob", "year: 2016", "dimensions: " + dims);  
var observer = new Observer("bbob", observerOptions);  
var benchmark = new Benchmark(suite, observer);
```

zaś w głównej pętli programu kolejny problem do optymalizacji pobierany jest z obiektu klasy **Bechmark**:

```
Problem = benchmark.getNextProblem()
```

Rozdział 5

Testy

5.1 Wyniki testów

Poniżej znajdują się wyniki testów naszej biblioteki przy użyciu platformy COCO. Zestawiamy ze sobą z jednej strony wyniki otrzymane przy użyciu pojedynczego węzła obliczeniowego, zarówno z wykorzystaniem GPU, jak i bez niego, z drugiej zaś wyniki głównego węzła obliczeniowego klastra mającego do dyspozycji 2 i 4 jednostki obliczeniowe, a zatem mogącego wykonać około 2- i 4-krotnie większą liczbę ewaluacji funkcji, ale nie korzystającego z obliczeń na kartach graficznych. Porównanie rezultatów wskazuje lepsze jakościowo rozwiązanie korzystające z obliczeń równoległych. Dodatkowo porównujemy ze sobą wyniki działania węzła obliczeniowego korzystającego z roju standardowych cząstek z węzłem, którego połowa populacji stanowią cząstki naładowane (z opisanej wcześniej wersji algorytmu PSO). Jednym z usprawnień względem podstawowej wersji algorytmu, które wprowadziliśmy do naszego rozwiązania, aby poprawić jego jakość są restarty algorytmu, gdy najlepsze znalezione rozwiązanie nie poprawi się przez odpowiednio długą liczbę iteracji oraz analogiczne restarty pojedynczych cząstek, które w przypadku braku poprawy znalezionej osobistej optimum, rozpoczynają poszukiwania od nowa w losowym punkcie dziedziny. Wyznaczona eksperymentalnie liczba iteracji, po których następują restarty 40 i 100 iteracji dla, odpowiednio, restarów cząstki i restarów globalnych.

5.1.1 Pojedynczy węzeł obliczeniowy

Rój złożony ze standardowych cząstek

Rysunek 5.1: Pojedynczy węzeł obliczeniowy

Rysunek 5.2: Wyniki roju 40 standardowych cząstek

Rój korzystający z GPU i złożony z cząstek standardowych i naładowanych

Rysunek 5.3: Struktura węzła obliczeniowego. Rój złożony z 40 cząstek na CPU oraz 640 cząstek na GPU

Rysunek 5.4: Wyniki roju 20 cząstek standardowych i 20 cząstek naładowanych oraz 640 cząstek na GPU

5.1.2 Klaster

Struktura klastra testowego

Rysunek 5.5: Struktura klastra testowego

Klaster złożony z 2 węzłów obliczeniowych

Rysunek 5.6: Wyniki klastra z 2 węzłami obliczeniowymi

Klaster złożony z 4 węzłów obliczeniowych

Rysunek 5.7: Wyniki klastra z 4 węzłami obliczeniowymi

5.2 Interpretacja wyników

Otrzymane wyniki standardowej wersji PSO na pojedynczym węźle obliczeniowym są zgodne z oczekiwaniami - algorytm radzi sobie dobrze z prostymi funkcjami,

takimi jak Linear Slope i szybko znajduje optimum z dużą dokładnością, jednakże dla funkcji trudnych, zwłaszcza w większym wymiarze, PSO osiąga bardzo niską skuteczność.

Porównanie wyników standardowego PSO z wynikami uzyskanymi po zastąpieniu połowy cząstek roju cząstkami naładowanymi oraz użyciu pomocy GPU wskazuje niestety na jedynie niewielką poprawę działania, widoczną jedynie dla niektórych funkcji.

Rysunek 5.8: Porównanie wersji z cząstkami naładowanymi i GPU (kółka) z wersją podstawową PSO

Znaczącą poprawę jakości działania algorytmu udało nam się uzyskać dopiero przy użyciu klastra obliczeniowego. Zgodnie z przewidywaniami stopień poprawy jest większy dla większej liczby węzłów obliczeniowych.

Rysunek 5.9: Porównanie pojedynczego węzła obliczeniowego (kółka) z klaster o 4 węzłach

Rozdział 6

Wymagania techniczne i instrukcja

6.1 Wymagania sprzętowe i programowe

Do uruchomienia aplikacji niezbędny jest komputer z systemem operacyjnym Windows z platformą .NET 4.5. W celu uruchomienia obliczeń na procesorze graficznym wymagana jest karta graficzna wspierająca wersję (compute capability) 2.0 architektury CUDA.

6.2 Instrukcja użytkownika

6.2.1 Interfejs użytkownika

Interfejs użytkownika pozwala na uruchomienie obliczeń dla jednej z zestawu predefiniowanych funkcji celu, włączając w to funkcje z benchmarku BBOB, używając zaimplementowanych typów algorytmu PSO. Umożliwia on odpowiednie dobranie parametrów funkcji celu oraz algorytmu. Za jego pomocą możliwym jest również utworzenie klastra obliczeniowego na wielu komputerach. Wybór oraz parametryzacja funkcji, algorytmu oraz konfiguracja klastra są kontrolowane przez odpowiednie pliki konfiguracyjne.

Klaster obliczeniowy

Plik konfigurujący klaster obliczeniowy jest napisany w języku XML. Korzeniem dokumentu jest element o nazwie NodeParameters. Dziećmi elementu głównego są elementy:

1. NrOfVCpu - wartość liczbową definiującą liczbę węzłów, które zostaną utworzone przez program. Gdy przypiszemy mu wartość -1 obliczenia zostaną uruchomione na liczbie wątków dopasowanej do procesora.
2. IsGpu - wartość logiczna włączająca/wyłączająca obliczenia na GPU
3. Ip - adres interfejsu sieciowego maszyny, z którym będą mogły połączyć się pozostałe wątki klastra
4. Ports - tablica integerów z numerami portów, na których nasłuchiwać będą usługi WCF
5. PeerAddress - element zawierający adres węzła, do którego aplikacja powinna się podłączyć. Powinien zostać pominięty w przypadku, gdy użytkownik nie ma zamiaru łączyć się z innymi stacjami

Przykładowy plik konfiguracyjny klastra:

```
<?xml version="1.0" encoding="utf-8"?>
<NodeParameters xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <NrOfVCpu>1</NrOfVCpu>
  <IsGpu>false</IsGpu>
  <Ip>25.123.241.71</Ip>
  <Ports>
    <int>8000</int>
  </Ports>
  <PeerAddress>net.tcp://25.123.244.149:8000/NodeService</PeerAddress>
</NodeParameters>
```

Algorytm PSO

Plik konfigurujący algorytm PSO oraz funkcję celu również napisany jest w języku XML. Korzeniem dokumentu jest element o nazwie PsoParameters. Dziećmi elementu głównego są elementy:

1. `Particles` - tablica definiująca liczbę oraz rodzaj cząstek użytych do obliczeń
2. `IterationsLimitCondition`, `Iterations` - wartość logiczna określająca czy warunek stopu ze względu na liczbę iteracji powinien być zastosowany oraz liczba iteracji do wykonania
3. `TargetValueCondition`, `TargetValue`, `Epsilon` - wartość logiczna określająca czy warunek stopu ze względu na osiągnięcie optimum powinien być zastosowany, optimum funkcji celu do wykonania oraz dokładność z jaką powinno być osiągnięte
4. `FunctionParameters` - definiujące parametry funkcji celu. Dziećmi tego elementu są:
 - (a) `FitnessFunctionType` - typ funkcji celu. Id funkcji w przypadku funkcji z benchmarku BBOB
 - (b) `Dimension` - wymiar funkcji
 - (c) `Coefficients` - współczynniki funkcji
 - (d) `SearchSpace` - tablica zawierająca dolną i górną granicę dziedziny poszukiwań

Przykładowy plik konfiguracyjny algorytmu PSO:

```
<?xml version="1.0" encoding="utf-8"?>
<PsoParameters xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Particles>
    <ParticlesCount>
      <ParticleType>Standard</ParticleType>
      <Count>6</Count>
    </ParticlesCount>
  </Particles>
  <IterationsLimitCondition>true</IterationsLimitCondition>
  <Iterations>1000</Iterations>
  <TargetValueCondition>false</TargetValueCondition>
  <TargetValue>0</TargetValue>
  <Epsilon>0</Epsilon>
  <FunctionParameters>
    <FitnessFunctionType>quadratic</FitnessFunctionType>
    <Dimension>2</Dimension>
    <Coefficients>
      <double>1.0</double>
      <double>1.0</double>
    </Coefficients>
    <SearchSpace>
      <DimensionBound>
        <Min>-5</Min>
        <Max>5</Max>
      </DimensionBound>
      <DimensionBound>
        <Min>-5</Min>
        <Max>5</Max>
      </DimensionBound>
    </SearchSpace>
  </FunctionParameters>
</PsoParameters>
```

Bibliografia

- [1] M. Keijzer, J. J. Merelo, G. Romero, M. Schoenauer, *Evolving Objects: a general purpose evolutionary computation library*, <http://geneura.ugr.es/jmerelo/habilitacion2005/papers/53.pdf>
- [2] *Evolving Objects homepage*, <http://eodev.sourceforge.net/>
- [3] *COCO homepage*, <http://coco.gforge.inria.fr>
- [4] N. Hansen, A. Auger, O. Mersmann, T. Tusar, D. Brockhoff, *COCO: a platform for comparing continuous optimizers in a black-box setting*, <https://arxiv.org/pdf/1603.08785.pdf>
- [5] R. C. Eberhart, J. Kennedy, *A new optimizer using particle swarm theory*, Proc. 6th Int. Symp. Micromachine Human Sci., Nagoya, Japan, 1995, pp. 39-43
- [6] J. Kennedy, R. C. Eberhart, *Particle swarm Optimization*, Proc. IEEE Int. Conf. Neural Networks, 1995, pp. 1942-1948
- [7] Maurice Clerc, *Standard particle swarm optimisation. From 2006 to 2011*, http://clerc.maurice.free.fr/pso/SPSO_descriptions.pdf
- [8] Randy Olson, *Visualization of two dimensions of a NK fitness landscape*, 2013, URL https://commons.wikimedia.org/wiki/File:Visualization_of_two_dimensions_of_a_NK_fitness_landscape.png, [Online; dostęp grudzień 2016]

- [9] J. J. Liang, A. K. Qin, P. N. Suganthan, S. Baskar, *Comprehensive learning particle swarm optimizer for global optimization of multimodal functions*, IEEE Transactions on Evolutionary Computation, Vol. 10, No. 3, June 2006
- [10] Y. Shi, R. C. Eberhart, *A modified particle swarm optimizer*, Proc. IEEE Congr. Evol. Comput., 1998, pp. 69-73
- [11] Y. Shi, R. C. Eberhart, *Parameter selection in particle swarm optimization*, Proc. 7th Conf. Evol. Programming, New York, 1998, pp. 591-600
- [12] Y. Shi, R. C. Eberhart, *Particle swarm optimization with fuzzy adaptive inertia weight*, Proc. Workshop Particle Swarm Optimization, Indianapolis, IN, 2001, pp. 101-106
- [13] A. Ratnaweera, S. Halgamuge, H. Watson, *Self-organizing hierarchical particle swarm optimizer with time varying accelerating coefficients*, IEEE Trans. Evol. Comput., vol. 8, pp. 240-255, Jun. 2004
- [14] H. Y. Fan, Y. Shi, *Study on Vmax of particle swarm optimization*, Proc. Workshop Particle Swarm Optimization, Indianapolis, IN, 2001
- [15] J. Kennedy, *Small worlds and mega-minds: Effects of neighborhood topology on particle swarm performance*, Proc. Congr. Evol. Comput., 1999, pp. 1931-1938
- [16] J. Kennedy, R. Mendes, *Population structure and particle swarm performance*, Proc. IEEE Congr. Evol. Comput., Honolulu, HI, 2002, pp. 1671-1676
- [17] P. N. Suganthan, *Particle swarm optimizer with neighborhood operator*, Proc. Congr. Evol. Comput., Washington, DC, 1999, pp. 1958-1962
- [18] X. Hu, R. C. Eberhart, *Multiobjective optimization using dynamic neighborhood particle swarm optimization*, Proc. Congr. Evol. Comput., Honolulu, HI, 2002, pp. 1677-1681
- [19] K. E. parsopoulos, M. N. Vrahatis, *UPSO-A unified particle swarm optimization scheme*, Lecture Series on Computational Sciences, 2004, pp. 868-873

- [20] R. Mendes, J. Kennedy, J. Neves, *The fully informed particle swarm: Simpler, maybe better*, IEEE Trans. Evol. Comput., vol. 8, pp. 204-210, Jun. 2004
- [21] T. Peran, K. Veeramachaneni, C. K. Mohan, *Fitness-distance-ratio based particle swarm optimization*, Proc. Swarm Intelligence Symp., 2003, pp. 174-181
- [22] P. J. Angeline, *Using selection to improve particle swarm optimization*, Proc. IEEE Congr. Evol. Comput., Anchorage, AK, 1998, pp. 84-89
- [23] M. Lovbjerg, T. K. Rasmussen, T. Krink, *Hybrid particle swarm optimizer with breeding and subpopulations*, Proc. Genetic Evol. Comput. Conf., 2001, pp. 469-476
- [24] V. Miranda, N. Fonseca, *New evolutionary particle swarm algorithm (EPSO) applied to voltage/VAR control*, Proc. 14th Power Syst. Comput. Conf., Seville, Spain, 2002. [Online]. Available: <http://www.psc02.org/papers/s21pos.pdf>
- [25] M. Lovbjerg, T. Krink, *Extending particle swarm optimizers with self-organized criticality*, Proc. Congr. Evol. Comput., Honolulu, HI, 2002, pp. 1588-1593
- [26] T. M. Blackwell, P. J. Bentley, *Don't push me! Collision-avoiding swarms*, Proc. IEEE Congr. Evol. Comput., Honolulu, HI, 2002, pp. 1691-1696
- [27] X. Xie, W. Zhang, Z. Yang, *A dissipative particle swarm optimization*, Proc. Congr. Evol. Comput., Honolulu, HI, 2002, pp. 1456-1461
- [28] K. E. Parsopoulos, M. N. Vrahatis, *On the computation of all global minimizers through particle swarm optimization*, IEEE Trans. Evol. Comput., vol. 8, pp. 211-224, Jun. 2004
- [29] V. Roberge, M. Tarbouchi, *Comparison of parallel particle swarm optimizers for graphical processing units and multicore processors*, International Journal of Computational Intelligence and Applications, Vol. 12 No. 1 (2013)
- [30] G. A. Laguna-Sanchez, M. Olguin-Carbajal, N. Cruz-Cortes, R. Barron-Fernandez, J. Alvarez-Cedillo, *Comparative study of parallel variants for a particle*

- swarm optimization algorithm implemented on a multithreading GPU*, J. Appl. Res. Technol. 7(3) (2010) 292-309
- [31] M. Cardenas-Montes, M. Vega-Rodriguez, J. Rodriguez-Vazquez, A. Gomez-Iglesias, *Effect of the block occupancy in GPGPU over the performance of particle swarm algorithm*, Proc. 10th Int. Conf. Adaptive and Natural Computing Algorithms, Vol. 1 (Springer Berlin, Heidelberg, 2011), pp. 310-319
- [32] Y. Hung, W. Wang, *Accelerating parallel particle swarm optimization via GPU*, 2012, Optimization Methods and Software, 27:1, 33-51, <http://dx.doi.org/10.1080/10556788.2010.509435>
- [33] M. Clerc, *Particle swarm optimization*, ISTE Publishing Company, Newport Beach, CA, 2006
- [34] L. Mussi, S. Cagnoni, *Particle swarm optimization within the CUDA architecture*, 2009, Available: <http://www.gpgpgpu.com/gecco2009/1.pdf>.
- [35] NVIDIA Corporation, *NVIDIA CUDA Programming Guide*, Version 2.3.1, 2009
- [36] Y. Zhou, Y. Tan, *GPU-based parallel particle swarm optimization*, IEEE Congress on Evolutionary Computation, 2009, pp. 1493-1500
- [37] W. Zhu, J. Curry, *Particle swarm with graphics hardware acceleration and local pattern search on bound constrained problems*, IEEE Swarm Intelligence Symposium (SIS '09), 2009, pp. 1-8

Warszawa, dnia

Oświadczenie

Oświadczamy, że pracę inżynierską pod tytułem: „Implementacja algorytmu PSO z zastosowaniem technologii CUDA”, której promotorem jest mgr inż. Michał Okulewicz, wykonaliśmy samodzielnie, co poświadczamy własnoręcznymi podpisami.

.....