Efektywna implementacja wybranego algorytmu optymalizacji globalnej

Dokumentacja techniczna

Paweł Kupidura, Hubert Rosiak, Kacper Trojanowski

Metryka dokumentu					
Projekt:	Efektywna implementacja wybranych algorytmów optymalizacji globalnej			Firma:	Politechnika Warszawska
Nazwa:	Dokumentacja techniczna				
Temat:	Efektywna implementacja wybranych algorytmów optymalizacji globalnej				
Autor:	Paweł Kupidura, Hubert Rosiak, Kacper Trojanowski				
Plik:	dokumentacja_techniczna_koncowa_kupidura_rosiak_trojanowski.pdf				
Nr. wersji:	1.0.0	Status:	Wersja ostateczna	Data sporządzenia:	2016-01-20
Streszczenie:	Celem dokumentu jest przedstawienie dokumentacji technicznej projektu.				
Zatwierdził:	mgr inż. Michał Okulewicz			Data ostatniej modyfikacji:	2016-01-20

Spis treści

Streszczenie

Dokument ten zawiera wstęp teoretyczny do problemu optymalizacji przy użyciu algorytmu Optymalizacji Rojowej (PSO) oraz opisuje architekturę i funkcjonowanie systemu implementującego powyższy algorytm, umożliwiającego równoległą oraz rozproszoną optymalizację z możliwością wykorzystania do obliczeń karty graficznej. Dodatkowo w dokumencie znajduje się instrukcja dla użtkownika i programisty przedstawiającą proces instalacji i uruchamiania systemu oraz opisującą sposób rozszerzania jego możliwości poprzez dostarczanie własnych typów cząstek PSO oraz optymalizowanych funkcji.

Słownik

SIMT - single-instruction, multiple threads.

Host - CPU.

Kernel - funkcja wykonywana równolegle na GPU.

Device - karta graficzna wspierająca technologię CUDA.

Pamięć globalna GPU (global memory) - pamięć o dużej pojemności znajdująca się na karcie graficznej i dostępna z każdego bloku/wątku oraz trwała przez cały czas działania aplikacji.

Pamięć współdzielona (shared memory) - pamięć niewielkich rozmiarów dzielona przez wszystkie wątki danego bloku i trwała w trakcie wykonywania kernela. Czas dostępu do niej jest kilkadziesiąt razy krótszy niż w przypadku pamięci globalnej.

Funkcja celu – optymalizowana funkcja rzeczywista zmiennej wektorowej.

Węzeł klastra – maszyna, na której przeprowadzane są obliczenia

Wybrany model wytwarzania

Nasz projekt wykonywany był zgodnie z modelem iteracyjnym (przyrostowym) wytwarzania oprogramowania. Dało nam to większą elastyczność wprowadzania zmian w trakcie tworzenia projektu. Sposób łączenia szerokiego wachlarza technologii używanych przez nas przy realizacji zadania powodował trudne do przewidzenia na początku problemy, których rozwiązywanie wymagało wracania do faz projektu poprzedzających moment wykrycia trudności.

Wstęp

Popularność algorytmów optymalizacji stochastycznej/losowej (algorytmy metaheurystyczne / populacyjne / ewolucyjne/ probabilistyczne i in.) oraz zwiększające się możliwości rozpraszania i zrównoleglania obliczeń na sprzęcie powszechnie dostępnym (wielordzeniowe procesory, karty graficzne, ko-procesory obliczeniowe) sprawiają, że posiadanie systemu umożliwiającego wykorzystanie tych zasobów w procesie optymalizacji arbitralnie wskazanej funkcji (procesu biznesowego) jest niezwykle pożądane.

PSO, jako algorytm populacyjny, jest algorytmem szczególnie podatnym na zrównoleglenie obliczeń, gdyż obliczenia doknywane są względnie niezależnie przez każdą cząstkę z osobna i mogą być wykonywane w tym samym czasie. Wymiana informacji na temat najlepszych znalezionych położeń między cząstkami roju jest oczywiście konieczna, lecz sposób jej realizacji może być dość elastyczny. Umożliwia to w prosty sposób rozproszenie obliczeń pomiędzy wiele komputerów (jak również pomiędzy wątki procesora i GPU na jednej maszynie), gdyż cząstka nie musi mieć pełnej informacji o roju – wystarczy, że informacje najintensywniej wymieniane będę w lokalnym sąsiedztwie znajdującym się na jednej maszynie (wątku CPU), zaś komunikacja z pozostałą częścią roju może być rzadsza.

Cel pracy

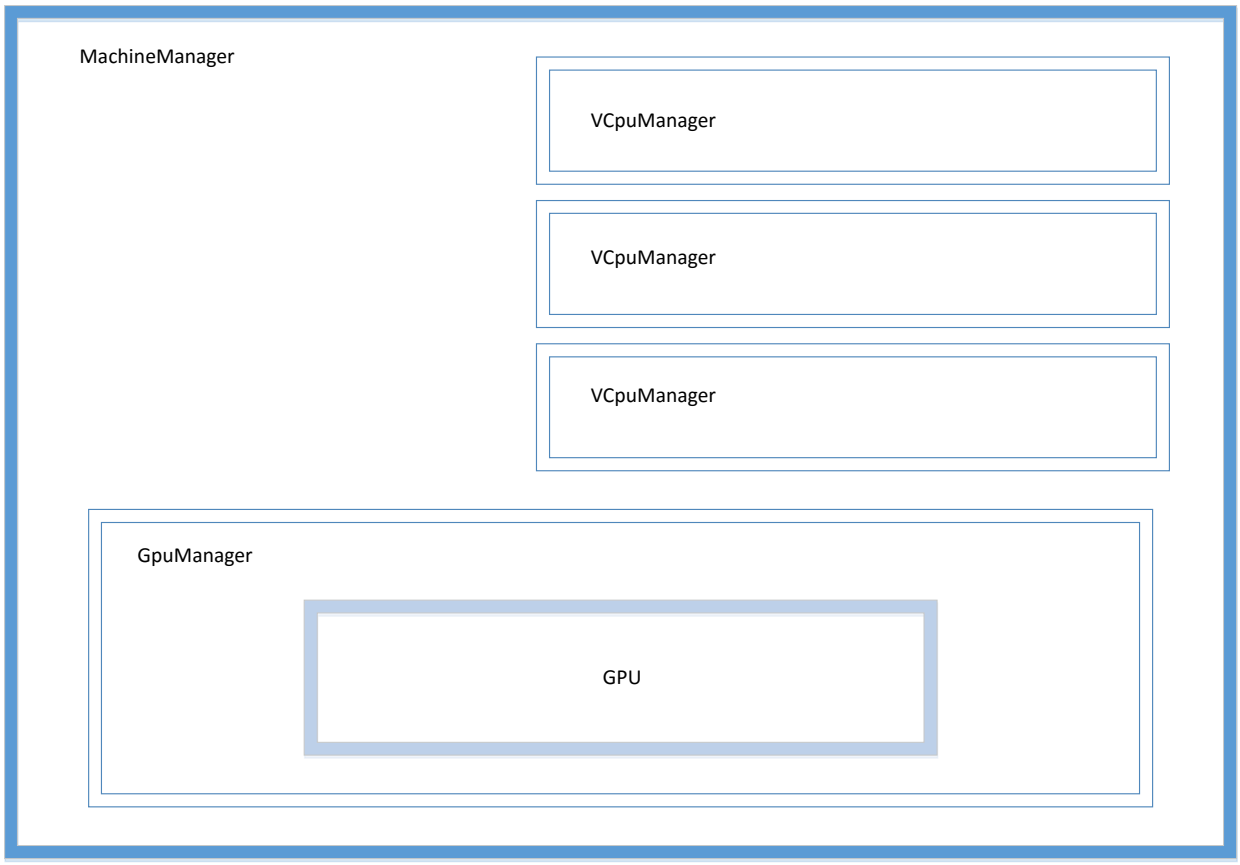
Celem naszej pracy inżynierskiej jest stworzenie biblioteki implementującej algorytm Particle Swarm Optimization (PSO) potrafiącej wykorzystywać cechy sprzętu, takie jak wielowątkowość procesora oraz obecność karty graficznej wspierającej technologię CUDA w celu zwiększenia efektywności obliczeń. Biblioteka dostępna będzie z poziomu środowiska .NET i umożliwi programiście łatwe rozszerzenie jej o własne wersje algorytmu.
Dodatkowo w naszej pracy zaproponujemy architekturę prostego klastra obliczeniowego wykorzystującego specyfikę algorytmu populacyjnego, jakim jest PSO, w celu rozproszenia obliczeń pomiędzy wiele maszyn (komputerów, procesorów) jednocześnie.

Architektura i komponenty systemu

Klaster obliczeniowy

W celu umożliwienia rozproszenia obliczeń pomiędzy wiele maszyn oraz zrównoleglenia ich na jednej maszynie poprzez uruchomienie wielu wątków, w naszej pracy zaproponujemy prostą architekturę klastra obliczeniwego, wykorzysującą do komunikacji technologię Windows Communication Foundation (WCF).
Podstawowym założeniem, którym kierowaliśmy się przy projektowaniu tej części naszej aplikacji było osiągnięcie maksymalnej elastyczności i "decentralizacji" w duchu algorytmu PSO – chcieliśmy uniknąć konieczności tworzenia specjalnego serwera, który pośredniczyłby w komunikacji między węzłami, czy projektowania znacząco róznych jednostek odpowiadających za poszczególne części wykonania algorytmu oraz nie

narzucać sztywnej topologii węzłów – ich podłączanie lub odłącznie od klastra powinno być możliwie proste. Węzłem inicjującym obliczenia może być dowolny z węzłów, zaś wyniki obliczeń przechowywane są jednocześnie na wszystkich węzłach.
Stworzony przez nas klaster składa się z jednakowych węzłów obliczeniowych. Każdy węzeł, implementowany przez klasę `MachineManager`, odpowiada jednemu komputerowi i zarządzą znajdującymi się na nim wirtualnymi procesorami (`VCpuManager`) oraz kartą graficzną (`GPUManager`).

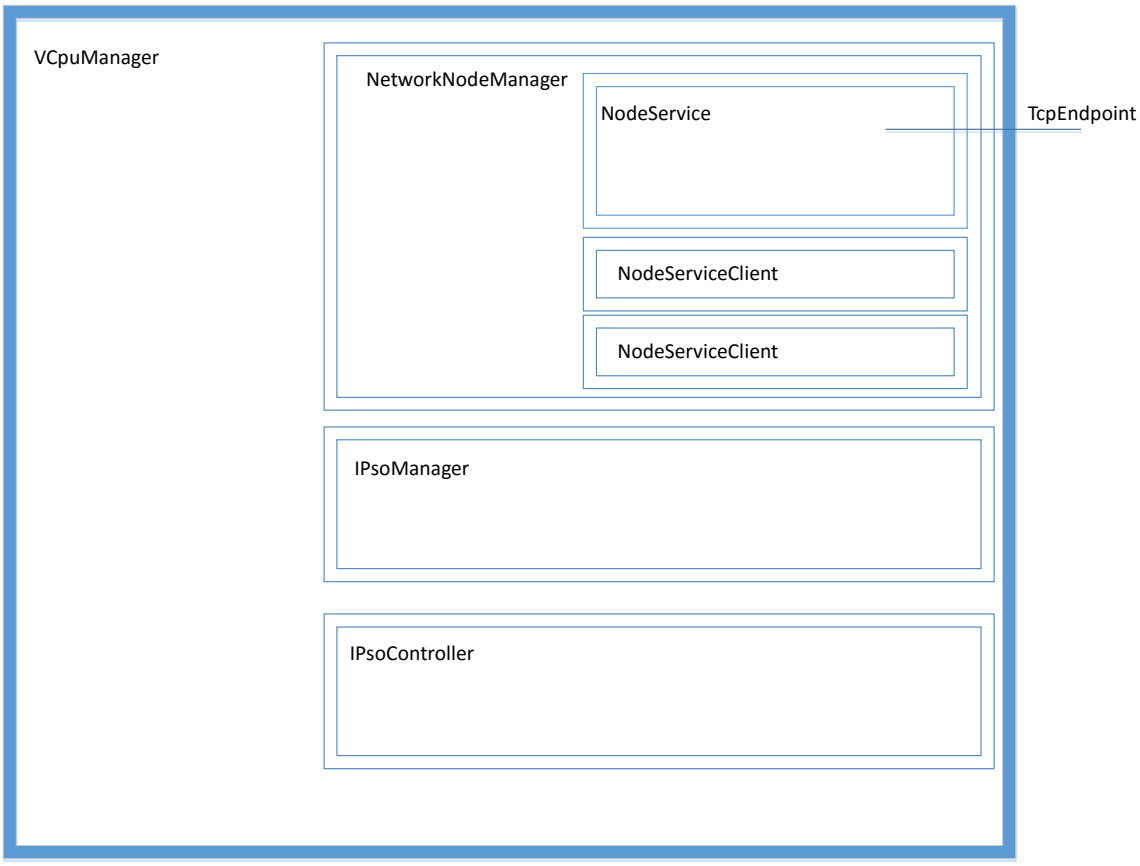


Rys. 1: Machine Manager

Do poprawnego przeprowadzenia obliczeń wystarczy istnienie jednego węzła – każdy kolejny jedynie poprawia wynik uzyskiwany w jednostce czasu, jednakże dodawanie ani usuwanie węzłów nie wpływa na poprawność obliczeń (nie spowoduje pojawienia się błędu).

VCpuManager

Wirtualny procesor (wątek procesora) implementowany jest za pomocą klasy `VcpuManager`, która zawiera w sobie instancje klas `NetworkNodeManager` oraz `PsoRingManager`. Pierwsza z nich odpowiada za stworzenie klastra i komunikację między węzłami, druga zaś realizuje komunikację między cząstkami stworzonymi na różnych VcpManagerach.



Rys. 2: VCpuManager

Klasa `NetworkNodeManager` odpowiada za łącznie się węzłów w klastrze. Stworzony przez nią klaster posiada topologię siatki.

Każdy z węzłów posiada serwis (instancję klasy `NodeService` implementującą interfejs `INodeService`) odbierający wiadomości za pomocą protokołu TCP. Wiadomości wysyłane są za pomocą klasy `NodeServiceClient` również implementującej interfejs `INodeService`.

Za komunikację między cząstkami odpowiada `PsoRingManger` implementujący interfejs `IPsoManager`. Komunikacja cząstek przebiega niezależnie od komunikacji między węzłami klastra – bazuje jedynie na ich numerach identyfikacyjnych. Umożliwia to dostosowanie topologii roju poprzez odpowiednią implementację interfejsu `IPsoManager`. Klasa implementująca ten interfejs jest odpowiedzialna za stworzenie cząstek przekazujących poprzez serwis WCF informacji o najlepszej znanej przez nie pozycji. Po dostarczeniu adresu zdalnego cząstki będącej częścią innego węzła odpytują one ten adres. Tak zaimplementowana komunikacja przebiega jednokierunkowo, gdyż odpytywana cząsteczka odpowiada na wszystkie zapytania, niezależnie od ich źródła i sama może odpytywać cząstkę, która nie ma jej w swoim sąsiedztwie lub, w szczególności, nie odpytywać żadnej cząstki. Zaprojektowana przez nas cząstka może odpytywać wyłącznie jedną sąsiadującą z nią cząstkę, natomiast może być odpytywana przez wiele różnych cząstek. Sąsiedztwa te mogą się zmieniać w trakcie działania algorytmu np. w przypadku awarii pojedynczego węzła.

GpuManager

Kontrolę nad kartą graficzną sprawuje `GpuManager`. Umożliwia on sprawdzenie dostępności odpowiedniej karty graficznej oraz uruchomienie na niej obliczeń.

PSO

Algorytm PSO zaimplementowany jest w języku C++/CLI oraz CUDA C i wywoływany za pośrednictwem wrappera .NET, czyli prostej warstwy dającej dostęp do algorytmu w środowisku technologii .NET. Wrapper umożliwia ponadto zorientowanie się w dostępnych na maszynie wersjach (tj. hybrydach) algorytmu.
Komponentem na wyższym poziomie, opakowującym zarówno algorytm jak i wrapper jest kontroler (interfejs `IPsoController`), który ma za zadanie rejestrowanie faktów dotyczących architektury maszyny (np. czy można uruchomić obliczenia na karcie graficznej, ile rdzeni ma procesor itp.) oraz obsługiwanie żądań wyższej warstwy.

UI

Prosty interfejs użytkownika umożliwi wybranie funkcji celu spośród podanych wraz z przestrzenią rozwiązań (możliwość wyboru jest ograniczona do kostek n-wymiarowych), a także hybrydy PSO, która ma zostać użyta do wykonania obliczeń. Umożliwi on także zdalne wykonanie obliczeń na innym komputerze, z którym komunikacja nawiązana zostanie poprzez serwis WCF.

Algorytm

Ogólnie o PSO

PSO (Particle Swarm Optimization) jest metodą rozwiązywania problemów optymalizacyjnych należącą do klasy algorytmów metaheurystycznych ewolucyjnych, wzorowaną na spotykanym w przyrodzie zachowaniu się rojów i stad zwierząt. 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 do konkretnego zastosowania. Ogólną postać algorytmu przedstawia schemat blokowy zamieszczony poniżej.

Szczegółowo o PSO

Klasyczna wersja algorytmu PSO (wg. [3]) wygląda następująco:

1. Inicjalizacja cząstek:
   (a) Utwórz cząstki rozmieszczone losowo 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;
2. Główna pętla algorytmu - dopóki nie został spełniony warunek stopu (określona liczba iteracji lub optimum globalne odpowiednio bliskie pewnej znanej wartości):
   (a) Dla każdej z cząstek oblicz jej nową prędkość w następujący sposób:
   $$prędkość \leftarrow \omega * prędkość + \varphi_1 * r_1 * p + \varphi_2 * r_2 * g,$$
   gdzie:
   $$p - wektor\ łączący\ obecne\ położenie\ cząstki\ z\ jej\ najlepszą\ znaną\ pozycją,$$
   $$g - wektor\ łączący\ obecne\ położenie\ cząstki\ z\ najlepszą\ globalnie\ pozycją,$$
   $$\omega, \varphi_1, \varphi_2 - ustalone\ przez\ użytkownika\ parametry\ (wagi)$$
   $$r_1, r_2 - liczby\ z\ przedziału\ [0,1]\ losowane\ w\ każdej\ iteracji$$
   (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;

Rys. 3: Schemat blokowy algorytmu PSO

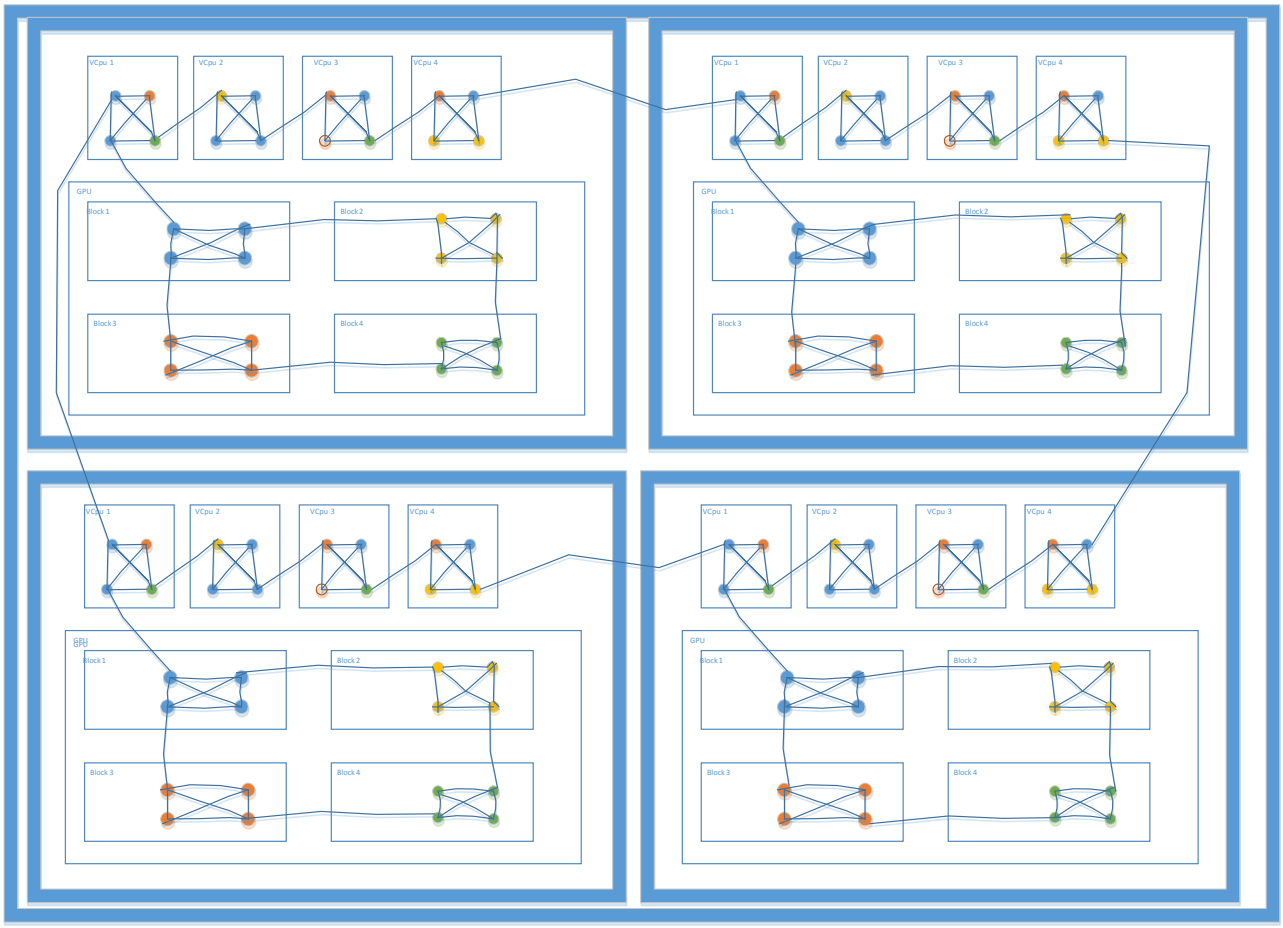
Hybrydyzacja PSO

Poszczególne warianty algorytmu różnić się mogą sposobem podejmowania decyzji w każdej iteracji. Ważnym aspektem naszego projektu jest możliwość hybrydyzacji PSO, czyli połączenia ze sobą różnych wariantów algorytmu, tak aby zachowanie części cząstek różniło się od zachowania pozostałych przy utrzymaniu komunikacji między nimi. Hybrydyzacja daje większą elastyczność i umożliwia wykorzystanie zalet różnych wersji algorytmu.
Jednym z podstawowych założeń naszego rozwiązania jest umożliwienie komunikacji pomiędzy różnymi typami cząstek, tak, aby cząstka nie miała wiedzy na temat typu cząstek, z którymi się komunikuje, gdyż wszystkie implementują wspólny interfejs.

Nasz program umożliwi użytkownikowi dodanie własnej hybrydy algorytmu, poprzez stworzenie klasy implementującej wspomniany wcześniej interfejs, co zaprezentowaliśmy implementując cząstkę klasycznego PSO.

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 równolegle na wszystkich maszynach lub za pomocą większej liczby mniejszych rojów.
Wybranym przez nas podejściem jest potraktowanie wszystkich cząsteczek na wszystkich maszynach jako jednego wielkiego roju, gdzie każda cząstka może potencjalnie komunikować się z każdą inną. Przykładowy schemat sąsiedztwa cząstek znajduje się na poniższej grafice.
Od sytuacji idealnej różni go to, że cząstki na różnych komputerach/procesach nie muszą być zsynchronizowane, co jednak nie przeszkadza im wymieniać między sobą informacji na temat najlepszych położeń.



Rys. 4: Topologia klastra

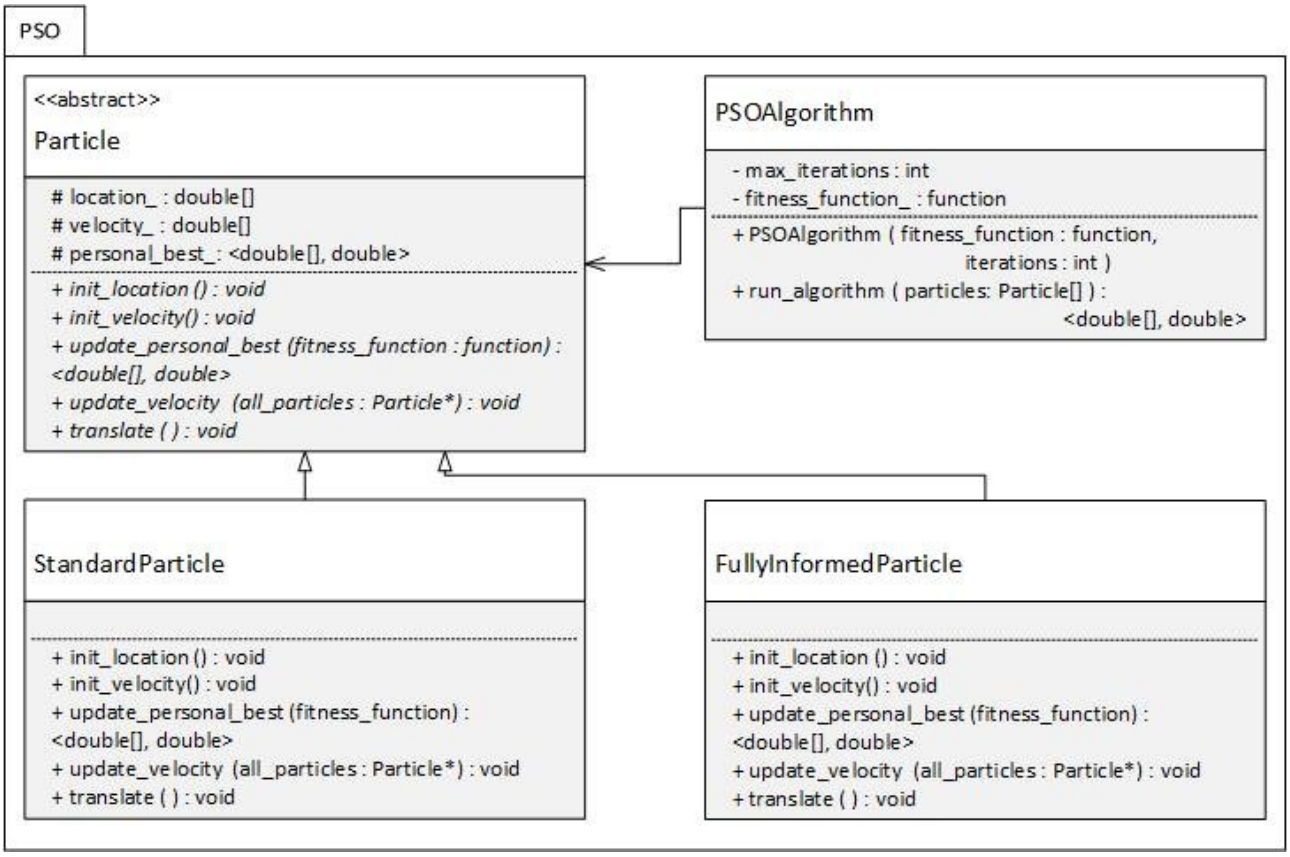
Szczegółowy opis implementacji

Algorytm 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).

Implementacja na CPU

Implementacja algorytmu na CPU prezentuje klasyczne podejście do problemu. Uruchamina na pojedynczym rdzeniu CPU jest punktem referencyjnym dla wydajności pozostałych wersji algorytmu.
Hybrydyzacja algorytmów PSO zaimplementowanych na CPU osiągana będzie poprzez obiektowe modelowanie systemu. Odpowiedzialność za sposób wykonania poszczególnych kroków algorytmu PSO spoczywać będzie na cząsteczkach dziedziczących po abstrakcyjnej klasie `Particle`. Konkretna implementacja takiej cząstki odpowiadać będzie danemu wariantowi PSO. Z punktu widzenia samego algorytmu, nie jest istotne w jaki sposób cząstki wykonują operacje potrzebne do uzyskania wyniku, stąd w celu

uruchomienia jego wykonania, potrzebne będzie jedynie dostarczenie zestawu cząstek obiektowi `PSOAlgorithm` będącemu odpowiedzialnym za obsługę algorytmu, w tym wykonanie jego kroków, zakończenie obliczeń oraz zwrócenie wyniku.
W celu rozszerzenia funkcjonalności dostarczonej przez nas biblioteki programista posiada możliwość dodawania wariantów algorytmu PSO. Może osiągnąć to poprzez zaimplementowanie dwóch klas – pierwszej dziedziczącej po klasie `PSO::Particle` oraz drugiej, opakowującą klasę poprzednią `Wrapper::Particle`. Tak skonstruowane klasy pozwolą na stworzenie odpowiednich cząstek w środowisku .NET (klasa opakowująca), a następnie wykonanie algorytmu używając metod z klasy napisanej w natywnym języku C++.



Rys. 5: Diagram klas reprezentujących cząstki PSO

Implementacja na CUDA

Nasza implementacja algorytmu PSO na procesory graficzne musi uwzględniać wszystkie ich ograniczeenia, aby uzyskać wymaganą efektywność. Jednym z ograniczeń ("bottleneck") jest szybkość transferu danych pomiędzy GPU a CPU, która jest znacznie mniejsza od szybkości transferu w samym GPU, nie mówiąc już o CPU, tak więc uwzględniająca ten problem wersja algorytmu powinna dążyć do minimalizacji transferu danych między tymi jednostakami.

Zrównoleglenie obliczeń uzyskujemy, mówiąc w dużej ogólności, poprzez przypisanie jednej cząstki do jednego wątku i wykonywanie obliczeń przy tym założeniu. Ograniczona sprzętowo liczba wątków z jednej strony, a potencjalnie bardzo duża

wielkość roju cząstek nie są problemem, gdyż zarządzanie wątkami jest zaimplementowane sprzętowo na GPU i zoptymalizowane.

Pierwszym etapem algorytmu jest skopiowania wszystkich niezbędnych parametrów zadania (takich jak wielkość roju czy liczba iteracji) do pamięci globalnej GPU.

Inicjalizacja położeń i prędkości roju cząstek następuje poprzez wywołanie kernela, w którym jeden wątek będzie odpowiadał za nadanie początkowego stanu pojedynczej cząstki, który będzie zapisywał do pamięci globalnej (na GPU). Do uzyskania losowych wektorów położeń i prędkości używamy biblioteki CUDA cuRAND.

Główna pętla algorytmu wykonywana jest na CPU do momentu osiągniecia warunku stopu. W każdej jej iteracji wywoływane są kernele odpowiadające poszczególnym etapom algorytmu, po których stan roju zostaje zapisany do pamięci globalnej.

Obliczenie funkcji celu następuje przy założeniu jednego wątku na cząstkę. Jeżeli otrzymany wynik jest lepszy od dotychczas najlepszego dla danej cząstki, wątek zastępuje stary wynik nowym. Stan roju znów zostaje zapisany do pamięci globalnej.

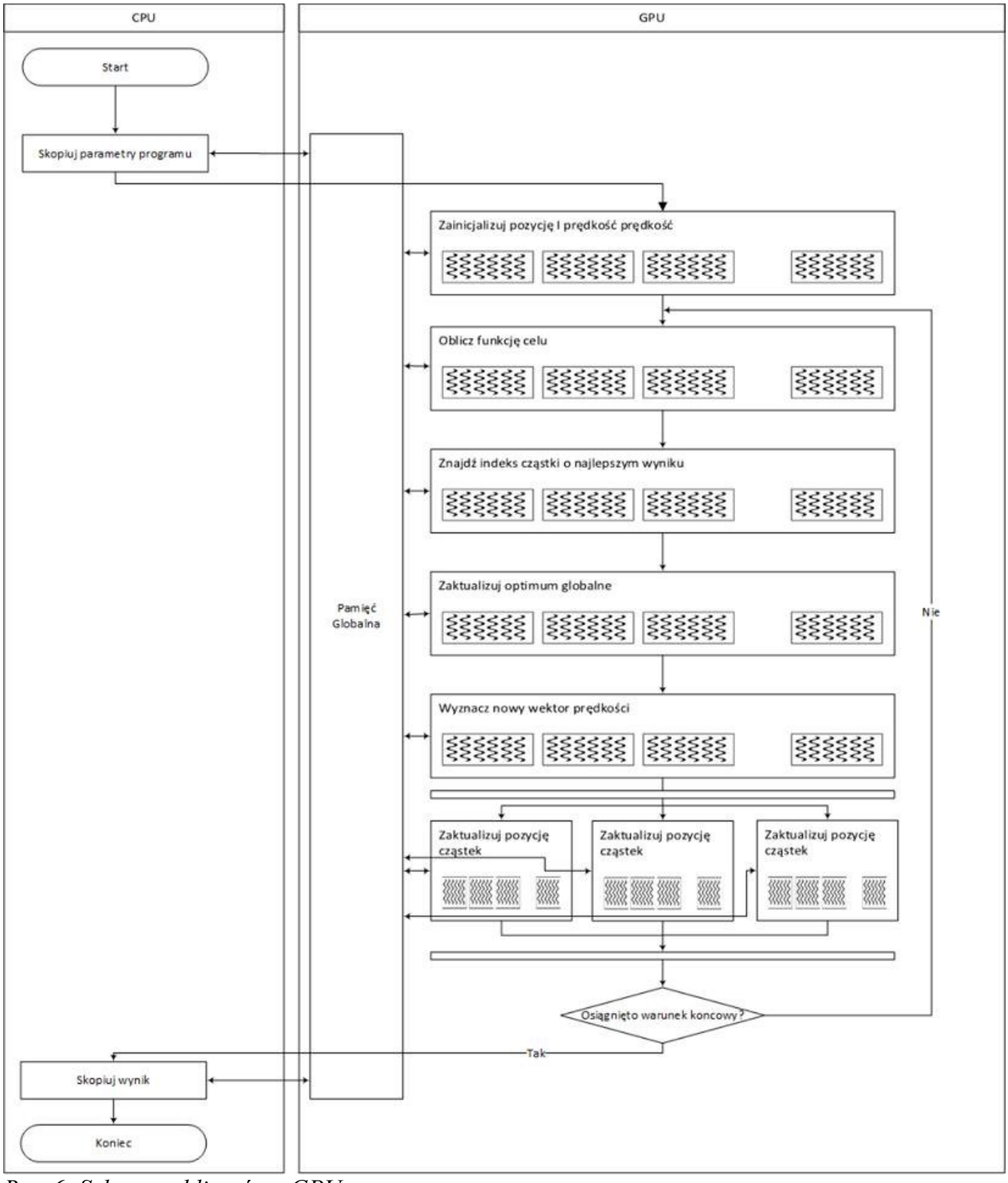
Wyszukanie najlepszego wyniku globalnego wykonany zostanie następująco: najpierw znajdujemy wątek-cząstkę z najlepszym wynikiem w każdym z bloków za pomocą redukcji opartej na drzewie (tree-based parallel reduction), a dopiero kolejny kernel znajduje wśród wyników poprzedniego element globalnie najlepszy, używając tej samej metody. Podział wyszukiwania na dwa etapy jest konieczny ze względu na brak komunikacji między blokami; wynik zapisany zostaje w pamięci globalnej.

Aby zaktualizować optimum globalne, dla znalezionego indeksu najlepszego globalnego wyniku wykorzystamy po jednym wątku do przeniesienia wartości w każdym wymiarze położenia optimum globalnego.

Najtrudniejszym obliczeniowo etapem będzie wyliczenie nowego wektora prędkości dla każdej cząstki. Każdy z typów cząstek będzie miał osobną implementację kernela wyliczającego nową prędkość. Kernele te będą wywoływane równolegle. W obliczeniach każdej cząstce będzie odpowiadał jeden wątek. W każdym z bloków będą znajdował się cząstki jednego typu.

Ostatni etap algorytmu, czyli aktualizowanie pozycji możemy dokonać przyjmując za jeden wątek jeden wymiar wektora położenia cząstki.

Cząstki na GPU pogrupowane są w mniejsze roje, każdy z nich odpowiada jednemu blokowi wątków GPU. Wybrana cząstka z każdego roju (bloku) komunikuje się z cząstką z sąsiedniego roju w pierścieniu. Dodatkowo, jedna z cząstek z jednego roju na GPU komunikuje się z rojem CPU.



Rys. 6: Schemat obliczeń na GPU

Hybrydyzacja PSO na CUDA

Hybrydyzacja, czyli współdziałanie różnych rodzajów cząstek, musi uwzględniać fakt, że każdy blok wątków może wykonywać tylko jeden zestaw instrukcji, toteż wszystkie wątki-cząstki w jednym bloku są tego samego typu. Każdy rodzaj cząstek ma swój własny kernel wyliczający prędkość.

Instrukcja użytkownika

Interfejs użytkownika pozwala na uruchomienie obliczeń dla jednej z zestawu predefiniowanych funkcji celu 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.

Funkcja celu

Plik konfigurujący funkcję celu powinien zawierać 4 linie. W pierwzsej z nich musi znaleźć się nazwa wybranej funkcji, w drugiej podany jest wymiar dziedziny funkcji celu, w trzeciej współczynniki funkcji oddzielone przecinkiem, natomiast w ostatniej zdefiniowana jest przestrzeń poszukiwań poprzez podanie kolejno, oddzielonych przecinkiem: ograniczenia dolnego i górnego dla każdego z wymiarów. Na przykład:

```
Quadratic
3
1,1,1
-10,10,-10,10,-10,10
```

Algorytm PSO

Plik konfigurujący algorytm PSO ma strukturę podobną do pliku funkcji celu. W jego przypadku powinno zostać podanych 6 linii. Pierwszą z nich jest wartość logiczna czy stosowany będzie limit iteracji, następująca po niej jest liczba iteracji. Trzecia, czwarta oraz piąta linia są odpowiednio: czy stosujemy warunek zbieżności do wartości celu, wartość celu, dokładność wyniku. Ostatnia linia jest liczbą cząsteczek. Na przykład:

```
true
1000
false
0
0
100
```

Klaster Obliczeniowy

Plik konfigurujący klaster obliczeniowy składa się z 4 linii. W pierwszej definiujemy liczbę wirtualnych procesorów, w kolejnej podajemy numery portów oddzielonych przecinkiem, na których węzeł nasłuchuje. Przedostatnią linią jest adres IP stacji na której uruchomiony zostaje klaster. Ostatnia linia jest opcjonalna, podajemy ją w przypadku, gdy mamy podłączyć się do istniejącego już klastra.

```
2
8889, 8890
192.168.123.122
net.tcp://192.168.123.132:1233/NodeService
```

Platforma

Platformą, którą wykorzystywać będziemy tworząc projekt będzie Visual Studio 2015/2013 z rozszerzeniem Nsight będącym częścią pakietu CUDA Toolkit 7.5 udostępnionego przez firmę NVIDIA.
Do uruchomienia aplikacji niezbędny jest komputer z systemem operacyjnym Windows z platformą .NET 4.5. Aby móc wykorzystać możliwość dokonywania obliczeń na karcie graficznej niezbędne jest posiadanie tzw. CUDA-capable GPU, czyli jednej z kart firmy nVidia spośród wymienionych w https://developer.nvidia.com/cuda-gpus.

Źródła

Techniczne

1. CUDA Toolkit Documentation v7.5: http://docs.nvidia.com/cuda/index.html
2. CUDA C Programming guide: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
3. CUDA Runtime API: https://docs.nvidia.com/cuda/cuda-runtime-api/index.html
4. CUDA Math API: https://docs.nvidia.com/cuda/cuda-math-api/index.html
5. cuBLAS: https://docs.nvidia.com/cuda/cublas/index.html
6. cuRAND: https://docs.nvidia.com/cuda/curand/index.html
7. Thrust: https://docs.nvidia.com/cuda/thrust/index.html
8.MAGMA (Matrix Algebra for GPU and Multicore Architectures): http://icl.cs.utk.edu/projectsfiles/magma/doxygen
9. CUDA-capable GPUs: https://developer.nvidia.com/cuda-gpus

Naukowe

1. R. Mendes, J. Kennedy, and J. Neves. The fully informed particle swarm: Simpler, maybe better. IEEE Transactions on Evolutionary Computation, 8(3):204–210, 2004.

2. Vincent Roberge, Mohammed 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).

3. Clerc, M. (2012). "Standard Particle Swarm Optimisation" (https://hal.archives-ouvertes.fr/file/index/docid/764996/filename/SPSO_descriptions.pdf). HAL open access archive.

4. You Zhou, Ying Tan. GPU-based Parallel Particle Swarm Optimization. Senior Member, IEEE

5. Nadia Nedjah, Rogério de M. Calazan, Luiza de Macedo Mourelle, Chao Wang. Parallel Implementations of the Cooperative Particle Swarm Optimization on Many-core and Multi-core Architectures. DOI 10.1007/s10766-015-0368-3