Zrównoleglenie algorytmu PSO w języku JavaScript

Patryk Nizio *@Dyzio* | *26.06.2019*
"Systemy rozproszone i równoległe"

Spis treści

Zadanie **2**

Algorytm PSO **2**
Opis algorytmu PSO 2
Funkcja testowa 2

Równoległość w JavaScript **4**
Język JavaScript - specyfikacja 4
Node.js 4
Sposoby zrównoleglenia w JS 5
Cluster 5
Child Process 5
Worker Threads 5
Web Workers 5

Zrównoleglenie algorytmu PSO **6**
Specyfikacja komputera 6
Implementacja 6
Pomiar i wyniki 8
Analiza wyników 9

Podsumowanie **10**

Bibliografia **10**

Zadanie

Celem zadania było przeprowadzenie pomiarów i zbadanie wydajności algorytmu PSO w języku JavaScript w wariancie szeregowym i równoległym.

Algorytm PSO

Opis algorytmu PSO

Proponowany w 1995 r. przez J. Kennedy'ego i R. Eberharta artykuł „Optymalizacja roju cząstek" stał się bardzo popularny ze względu na jego ciągły proces optymalizacji, pozwalający na różne odmiany algorytmu.

Ideą algorytmu PSO jest iteracyjne przeszukiwanie przestrzeni rozwiązań problemu przy pomocy roju cząstek. Każda z cząstek posiada swoją pozycję w przestrzeni rozwiązań, prędkość oraz kierunek w jakim się porusza. Ponadto zapamiętywane jest najlepsze rozwiązanie znalezione do tej pory przez każdą z cząstek (rozwiązanie lokalne), a także najlepsze rozwiązanie z całego roju (rozwiązanie globalne). Prędkość ruchu poszczególnych cząstek zależy od położenia najlepszego globalnego i lokalnego rozwiązania oraz od prędkości w poprzednich krokach. Poniżej przedstawiony jest wzór pozwalający na obliczenie prędkości danej cząstki.

$$\mathbf{v \leftarrow \omega v + \varphi_l r_l (l-x) + \varphi_g r_g (g-x)}$$

Gdzie:
v - prędkość cząstki
ω - współczynnik bezwładności, określa wpływ prędkości w poprzednim kroku
φ_l - współczynnik dążenia do najlepszego lokalnego rozwiązania
φ_g - współczynnik dążenia do najlepszego globalnego rozwiązania
l - położenie najlepszego lokalnego rozwiązania
g - położenie najlepszego globalnego rozwiązania
x - położenie cząstki
r_l, r_g - losowe wartości z przedziału <0,1>

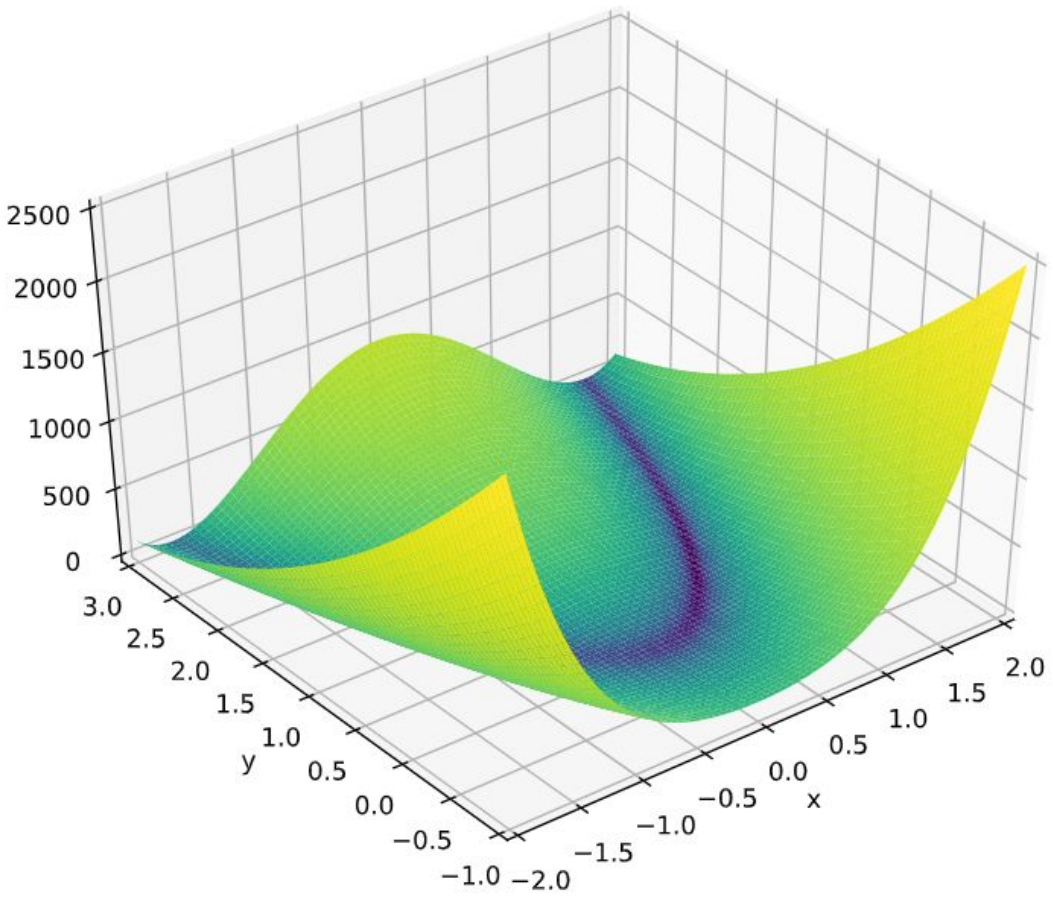
Funkcja testowa

Do testowania wydajności algorytmu wykorzystano funkcję Rosenbrocka, w przestrzeni tej funkcji algorytm PSO szukał optymalnego rozwiązania. Funkcja została wprowadzona w 1960 roku przez Howarda H. Rosenbrocka w 1960 r. przez swój charakterystyczny kształt nazywana bywa doliną Rosenbrocka lub funkcją banana Rosenbrocka. Globalne minimum jest w długiej, wąskiej, parabolicznej płaskiej dolinie. Znalezienie doliny jest banalne. Zbieżność z globalnym minimum jest jednak trudna.

Funkcja jest zdefiniowana przez wzór:

f(x,y) = (a - x)^2 + b*(y + x^2)^2

Wykres funkcji wygląda następująco:



Implementacja dla a=1, b=100:

```
Math.pow(1-x[0],2) + 100*Math.pow(x[1]-x[0]*x[0],2)
```

Równoległość w JavaScript

Język JavaScript - specyfikacja

Javascript jest jednowątkowy, oznacza to, że jeden wątek obsługuje pętlę zdarzeń. Historycznie JavaScript był językiem który miał obsługiwać proste animację na stronach WWW. Nie było konieczne używanie bardziej złożonych mechanizmów gdyż sama pętla zdarzeń bardzo dobrze obsługiwała zdarzenia asynchroniczne.

W przypadku starszych przeglądarek cała przeglądarka udostępnia jeden wątek między wszystkimi kartami. Nowoczesne przeglądarki poprawiły to, wykorzystując proces na instancję witryny lub różne wątki na karcie. Mimo, że dedykowane wątki poprawiły szybkość reakcji stron internetowych, nadal nie ma możliwości, aby każda karta obsługiwała wiele uruchomionych jednocześnie skryptów.

Przez wiele lat nie były potrzebne bardziej skomplikowane mechanizmy zrównoleglenia gdyż język służył do obsługi animacji na stronie, obsługi prostych eventów i walidacji formularzy. Intensywny rozwój technologii webowych oraz samego języka (rozwój Node.js i specyfikacji ES6+) wprowadził nowe możliwości związane ze zrównolegleniem kodu JS, a także obsługę wielowątkowości.

Node.js

Node.js to wieloplatformowe środowisko uruchomieniowe o otwartym kodzie do tworzenia aplikacji typu server-side napisanych w języku JavaScript. Przyczynił się do stworzenia paradygmatu „JavaScript everywhere”. Umożliwia programistom tworzenie aplikacji w obrębie jednego języka programowania zamiast polegania na odrębnych po stronie serwerowej. Został stworzony w 2009 roku przez wyodrębnienie silnika Chrome V8 i stworzenia środowiska uruchomieniowego języka poza przeglądarką.

Środowisko to jest intensywnie rozwijane. W czerwcu 2018 roku (wersja 10.5.0) dodano obsługę wielowątkowości jako eksperymentalna funkcjonalność. Wprowadzenie obsługi wielowątkowości nie była łatwa gdyż mechanizm nie powinien zmieniać natury języka i powinien być kompatybilny z istniejącymi mechanizmami.

2018-06-20, Version 10.5.0 (Current), @targos

Notable Changes

- crypto:
  - Support for `crypto.scrypt()` has been added. #20816
- fs:
  - BigInt support has been added to `fs.stat` and `fs.watchFile`. #20220
  - APIs that take `mode` as arguments no longer throw on values larger than `0o777`. #20636 #20975 (Fixes: #20498)
  - Fix crashes in closed event watchers. #20985 (Fixes: #20297)
- Worker Threads:
  - Support for multi-threading has been added behind the `--experimental-worker` flag in the `worker_threads` module. This feature is *experimental* and may receive breaking changes at any time. #20876

https://github.com/nodejs/node/blob/master/doc/changelogs/CHANGELOG_V10.md

Sposoby zrównoleglenia w JS

Problem równoległości możemy podzielić na dwie kategorie w zależności od środowiska. W przypadku wymagających obliczeń, animacji po stronie przeglądarki (klienta) używamy mechanizmu Web Workerów. W przypadku środowiska Node.js możemy wykorzystać klastry oraz procesy potomne.

Cluster

Pojedyncza instancja Node.js działa w jednym wątku. Aby skorzystać z systemów wielordzeniowych, użytkownik czasami chce uruchomić klaster procesów Node.js, aby obsłużyć obciążenie. Moduł klastrów umożliwia łatwe tworzenie procesów potomnych, które wszystkie współdzielą porty serwera. Możemy rozwidlić (fork) proces główny na wiele procesów potomnych (zazwyczaj mających jedno dziecko na procesor). W tym przypadku dzieci mogą dzielić port z rodzicem (dzięki komunikacji między procesami lub IPC).
Mechanizm ten został użyty w omawianej implementacji algorytmu PSO.
Więcej: https://nodejs.org/api/cluster.html

Child Process

Możemy rozwidlić proces, główny proces może komunikować się z procesem potomnym poprzez wysyłanie i odbieranie zdarzeń. Żadna pamięć nie jest udostępniana. Wszystkie wymienione dane są „klonowane”, co oznacza, że zmiana ich na jednej stronie nie zmienia jej po drugiej stronie. Problem z asynchronicznym przesyłaniem i odbieraniem danych możemy rozwiązać poprzez funkcję zwrotne (callback) lub oznaczenie funkcji jako asynchroniczna (async). W przypadku wykorzystania tego rozwiązania mamy jednak problem z wykorzystaniem dużej ilości pamięci gdyż powstanie nowego procesu to kolejna duplikacja danych a samo kopiowanie danych może trochę potrwać.
Więcej: https://nodejs.org/api/child_process.html

Worker Threads

Jest to nowa funkcjonalność wprowadzona w wersji 10.5.0, obecnie jest oznaczona jako eksperymentalna.
Worker Threads mają izolowane konteksty. Wymieniają informacje z głównym procesem za pomocą przekazywania wiadomości, żyją w tym samym procesie, więc zużywają znacznie mniej pamięci niż procesy potomne. Tak samo jak w przypadku wszystkich asynchronicznych akcji możemy wykorzystać elementy języka JavaScript takie jak promises, async, callback w celu synchronizacji danych.
Więcej: https://nodejs.org/api/worker_threads.html

Web Workers

Web Workers to sposób na uruchamianie skryptów w wątkach w tle. Wątek roboczy może wykonywać zadania bez zakłócania interfejsu użytkownika. Dane są przesyłane między głównym wątkiem a pracownikami (Web Workers) za pośrednictwem wiadomości. Ponieważ pracownicy pracują na osobnym wątku niż główny wątek wykonawczy, można wykorzystać worker do uruchamiania wymagających zadań obliczeniowych bez tworzenia blokujących instancji.
Wiecej:
https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers

Zrównoleglenie algorytmu PSO

Specyfikacja komputera

Badania przeprowadzono na komputerze z czterordzeniowym procesorem:
Intel(R) Core(TM) i5-7300U CPU @ 2.60GHz

Implementacja

Kod wątku rodzica, tutaj odbywa się rozwidlenie procesu na osobne procesory.

```
const cluster = require('cluster');
const fs = require('fs');
const crypto = require("crypto");
const numCPUs = require('os').cpus().length;
const sessionID = crypto.randomBytes(10).toString('hex');
const functionType = "Rosenbrock"
const PSO_PARTICLES_NUM = 100;

// main thread
if (cluster.isMaster) {
 for (let i = 0; i < numCPUs; i++) {
      console.log(`worker #${i} started`);
      let worker = cluster.fork();
      worker.send({sessionID:sessionID, cpuID: i});
 }
}
```

Kod klastra:

```
if(cluster.isWorker){
   // CSV row object: sessionID,iterations,time,cpu,cpuID,value,point,findType,functionType
   // const msgObj = {...}          // Create  msgObj object [...]

   // Receive messages from the master process.
   process.on('message', function(msg) {
      msgObj.sessionID = msg.sessionID;
      msgObj.cpuID = msg.cpuID;
      console.log(msg)
   });

   const pso = require("./particle-swarm-optimization");
   const config =  {
      inertiaWeight: 0.4,
      social: 0.8,
      personal: 0.2,
      pressure: 0.5
   }

   const optimizer = new pso.Optimizer();
   optimizer.setOptions(config)

   optimizer.setObjectiveFunction(function (x, done) {
      setTimeout(function () {
      // done(Math.pow(x[0],2) + Math.pow(x[1],2));
      done(-(Math.pow(1-x[0],2) + 100*Math.pow(x[1]-x[0]*x[0],2)));
```

```
        },  0);
      }, { async: true }
    );

    let t0 = Date.now();
    const initialPopulationSize = PSO_PARTICLES_NUM;
    const domain = [new pso.Interval(-100, 100), new pso.Interval(-100, 100)];
    optimizer.init(initialPopulationSize, domain);
    let iterations = 0;
    const maxIterations = 10000;
    let lastBestResult = null;

    function saveResult(msg){
        // Create  msgObj object from msg  [...]
      fs.appendFile("./result.csv", msgRow, (err) => {
        if(err) {
          return console.log(err);
        } else {
          console.log(msgRow);
          process.exit(0);
        }
      });
    }

    function loop() {
      if(lastBestResult && (Math.abs(0 - optimizer.getBestFitness()) < 1e-4 )){
        // Create  msgObj object [...]
        saveResult(msgObj);
      } else if (iterations >= maxIterations) {
        // Create  msgObj object [...]
        saveResult(msgObj);
      } else {
        iterations++;
        lastBestResult = optimizer.getBestFitness();
        optimizer.step(loop);
      }
    }
    loop();
}
```

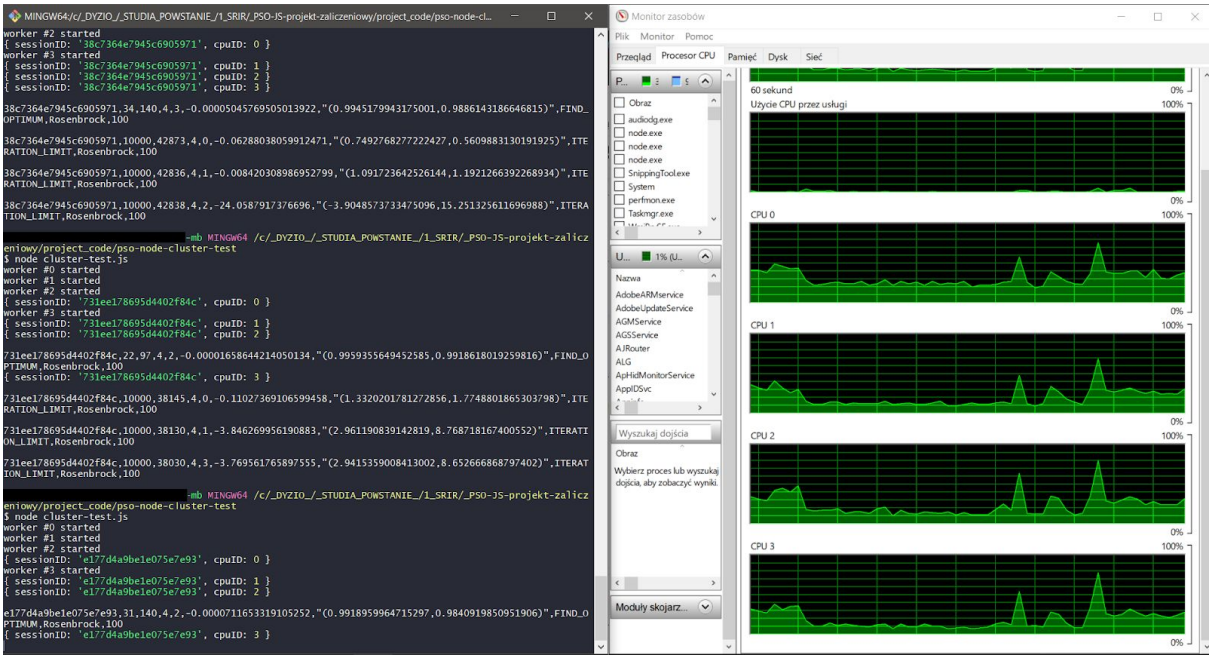
Pomiar i wyniki

Wyniki zapisywane były bezpośrednio do pliku CSV.

	A	B	C	D	E	F	G	H	I	J
1	sessionID	iterations	time	cpu	cpuID	value	point	findType	functionType	particlesNum
2	361a5d98173ad56266f0	23	459	1	0	-0.0000943165435963258	(1.002246973	FIND_OPTIMUM	Rosenbrock	100
3	8d835138dc3f8dc34ebf	44	832	1	0	-0.0000403399889751311 4	(1.004349337	FIND_OPTIMUM	Rosenbrock	100
4	f379494be6c7b4e7084a	87	1604	1	0	-0.0000960007010400673 5	(0.990406332	FIND_OPTIMUM	Rosenbrock	100
5	6ab49c400d3f703dde9d	16	288	1	0	-0.0000488703816768982 6	(1.003638249	FIND_OPTIMUM	Rosenbrock	100
6	5958ac5719c39a248052	21	380	1	0	-0.0000517690960301774 4	(1.000334235	FIND_OPTIMUM	Rosenbrock	100
7	f519e0b0c202267a3c16	24	461	1	0	-0.0000049291910084939 24	(1.001328380	FIND_OPTIMUM	Rosenbrock	100
8	1da35bff076683288dbb	33	689	1	0	-0.0000057235712564029 11	(0.992864183	FIND_OPTIMUM	Rosenbrock	100
9	942070d466de5a5a29d4	25	479	1	0	-0.0000373799311446783	(1.006094049	FIND_OPTIMUM	Rosenbrock	100
10	0081a2ef0de25f12ee82	23	456	1	0	-0.0000339176545526239 4	(1.004624167	FIND_OPTIMUM	Rosenbrock	100
11	737764f794f368caa988	30	571	1	0	-0.0000668240181672124 1	(0.994311635	FIND_OPTIMUM	Rosenbrock	100
12	137337ad6c610211c267	25	470	1	0	-0.0000731985530996109 4	(1.003405829	FIND_OPTIMUM	Rosenbrock	100
13	888850778dc6a272844f	17	320	1	0	-0.0000086572596058405 85	(0.998013259	FIND_OPTIMUM	Rosenbrock	100
14	5f7fe81dcd36cfad3bce	20	159	1	0	-0.0000643008509577580 8	(1.004865024	FIND_OPTIMUM	Rosenbrock	100
15	1499456d215a50d90421	25	432	1	0	-0.0000891399947228330 4	(0.992745546	FIND_OPTIMUM	Rosenbrock	100
16	e9ff65d58a1229220b93	47	901	1	0	-0.0000488971400800380 94	(0.993127982	FIND_OPTIMUM	Rosenbrock	100
17	7504bf4d4f9f4c30b4e4	22	395	1	0	-0.0000199977265818964 94	(0.997519496	FIND_OPTIMUM	Rosenbrock	100
18	b1bce8b6250665862fd4	32	604	1	0	-0.0000049012011286701 424	(1.001685079	FIND_OPTIMUM	Rosenbrock	100
19	41a66e9c09b7af987078	26	480	1	0	-0.0000843486910621380 6	(0.990988557	FIND_OPTIMUM	Rosenbrock	100
20	86215e0ce0b110999a6b	305	5592	1	0	-0.0000767036388054794 3	(0.993154690	FIND_OPTIMUM	Rosenbrock	100
21	c285be98e8ade7b574c8	20	393	2	0	-0.0000892832843228063 5	(0.994202620	FIND_OPTIMUM	Rosenbrock	100
22	bbd2a41bbeda4ea22afb	12	229	2	1	-0.0000372557209699158 8	(1.003829452	FIND_OPTIMUM	Rosenbrock	100
23	b11710ce332eb6f6d7e3	39	686	2	0	-0.0000029984147619291 15	(0.999957862	FIND_OPTIMUM	Rosenbrock	100
24	8b0c352932f54b2ba747	24	459	2	1	-0.0000147966327560747 67	(1.003743413	FIND_OPTIMUM	Rosenbrock	100
25	ba7b088aaa3edc1cd932	35	620	2	0	-0.0000603067600480289 4	(0.993614748	FIND_OPTIMUM	Rosenbrock	100
26	08da4f4268b7d9c0b661	25	449	2	1	-0.0000178679779776418 71	(0.997203903	FIND_OPTIMUM	Rosenbrock	100
27	08da4f4268b7d9c0b661	30	582	2	0	-0.0000056541919794871 81	(0.998914062	FIND_OPTIMUM	Rosenbrock	100
28	a0c7548734a6c387b497	24	367	2	1	-0.0000703653582411428	(1.002779224	FIND_OPTIMUM	Rosenbrock	100
29	ee97151494985cb0f5e1	48	873	2	1	-0.0000946608916153495 6	(0.990273430	FIND_OPTIMUM	Rosenbrock	100
30	2f74313fe4909fd45529	23	144	2	0	-0.0000705797682159041 8	(1.000744333	FIND_OPTIMUM	Rosenbrock	100
31	d046a0fd3df218702bbc	31	137	2	0	-0.0000608257712898221 4	(0.992550606	FIND_OPTIMUM	Rosenbrock	100
32	d046a0fd3df218702bbc	24	117	2	1	-0.0000374947509616410 34	(1.004070622	FIND_OPTIMUM	Rosenbrock	100
33	99315037d90a0e4b7d87	25	118	2	0	-0.0000843133463320565 8	(1.001349890	FIND_OPTIMUM	Rosenbrock	100
34	99315037d90a0e4b7d87	46	201	2	1	-0.0000509309286440122 35	(0.995743673	FIND_OPTIMUM	Rosenbrock	100
35	9ea142f9e1a1bcb66a83	21	108	2	1	-0.0000978158274105645 8	(1.001939014	FIND_OPTIMUM	Rosenbrock	100
36	60bbecf88d5e313b5647	27	434	3	0	-0.0000094189520157446 07	(1.000309406	FIND_OPTIMUM	Rosenbrock	100
37	60bbecf88d5e313b5647	28	275	3	2	-0.0000768493453219158 9	(0.992712197	FIND_OPTIMUM	Rosenbrock	100

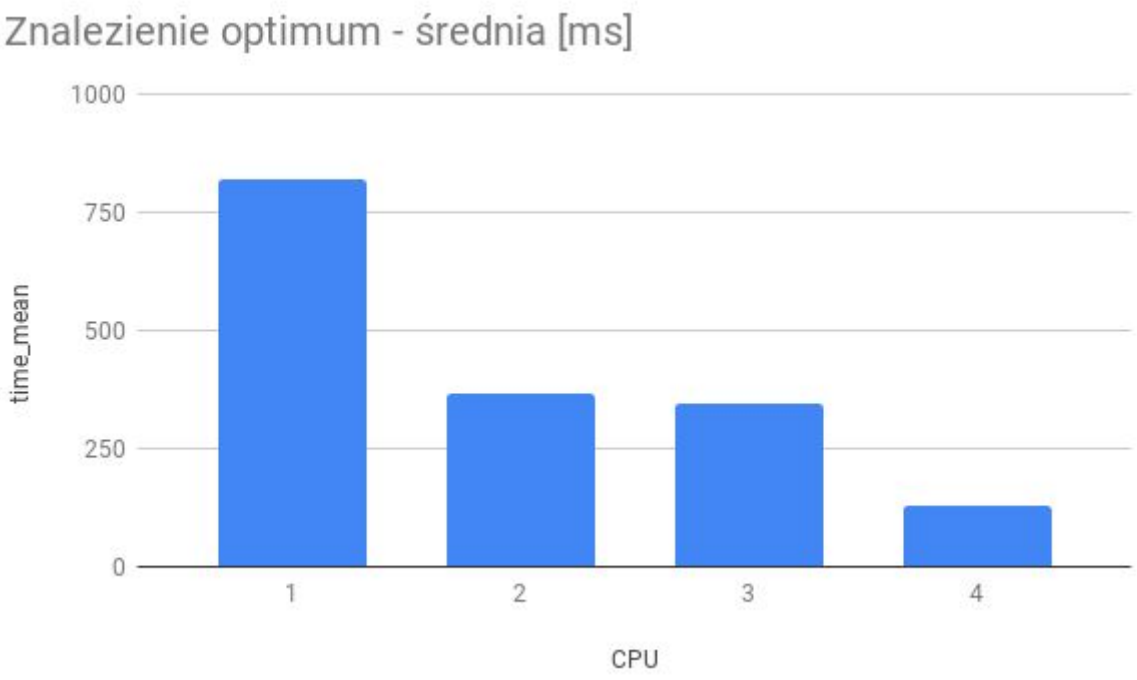
Arkusz CSV dostępny pod adresem: https://raw.githubusercontent.com/Dyzio18/smell-piece-of-code/master/pso-node-js-cluster/result.csv

Dodatkowo przedstawiam screen z uruchomienia algorytmu i wykresy zużycia by pokazać obciążenie na każdym procesorze.



Analiza wyników

Poniżej wykres przedstawiający uśrednioną szybkość uzyskania rozwiązania w milisekundach według ilości procesów.



Wariant szeregowy	2	3	4
819,5789	365,5333	346,0000	129,9000
przyspieszenie	2,2421456	2,3687253	6,3093068
efektywność	1,1210728	0,7895751	1,5773267

Podsumowanie

Obliczenia zostały zrównoleglone poprzez uruchomienie algorytmu PSO na osobnych procesorach. Alternatywą dla tego rozwiązania jest zrównoleglenie samego algorytmu w miejscu obliczania wartości cząstek.

Zastosowane rozwiązanie nie jest doskonałe jednak obliczenia na większej ilości procesorów charakteryzują się lepszymi wynikami. Najpewniej było to spowodowane przez to że przy większej ilości prób istnieje większe prawdopodobieństwo wylosowania cząstek w lepszych pozycjach do znalezienia rozwiązania. Najszybciej znaleziono rozwiązanie przy czterech procesorach - 66ms, następnie przy trzech procesorach - 105 ms, dwóch - 108 ms, najszybsze rozwiązanie w wariancie szeregowym wynosiło 159 ms.

Dodatkowo warto zaznaczyć że algorytm PSO nie determinuje znalezienia rozwiązania. Warunkiem końcowym było znalezienie rozwiązania do 0.0001 lub maksymalna ilość iteracji (do 10 000). W wielu przypadkach algorytm nie znajdywał optymalnego rozwiązania w dopuszczalnej ilości iteracji, zrównoleglenie algorytmu pozwala zwiększyć szanse na uzyskanie dopuszczalnego wyniku.

Bibliografia

Poniżej lista stron która pomogła w napisaniu sprawozdania:

- Strona projektu Node.js: https://github.com/nodejs/node
- https://www.ii.uni.wroc.pl/~prz/2011lato/ah/opracowania/roj_czast.opr.pdf
- http://adrianton3.github.io/pso.js/docs/pso.html
- http://www.red3d.com/cwr/boids/
- https://blog.logrocket.com/node-js-multithreading-what-are-worker-threads-and-why-do-they-matter-48ab102f8b10/
- https://stackoverflow.com/questions/40028377/is-it-possible-to-achieve-multithreading-in-nodejs
- Artykuł nt. wielowątkowości Node.js (opis problematyki) https://blog.logrocket.com/node-js-multithreading-what-are-worker-threads-and-why-do-they-matter-48ab102f8b10/
- https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers
- Porównanie cluster.fork() i child_process.fork() https://stackabuse.com/setting-up-a-node-js-cluster/
- Repozytorium z wykonanym kodem: https://github.com/Dyzio18/smell-piece-of-code/tree/master/pso-node-js-cluster