

Imperialist Competitive Algorithm

KIV/PPR

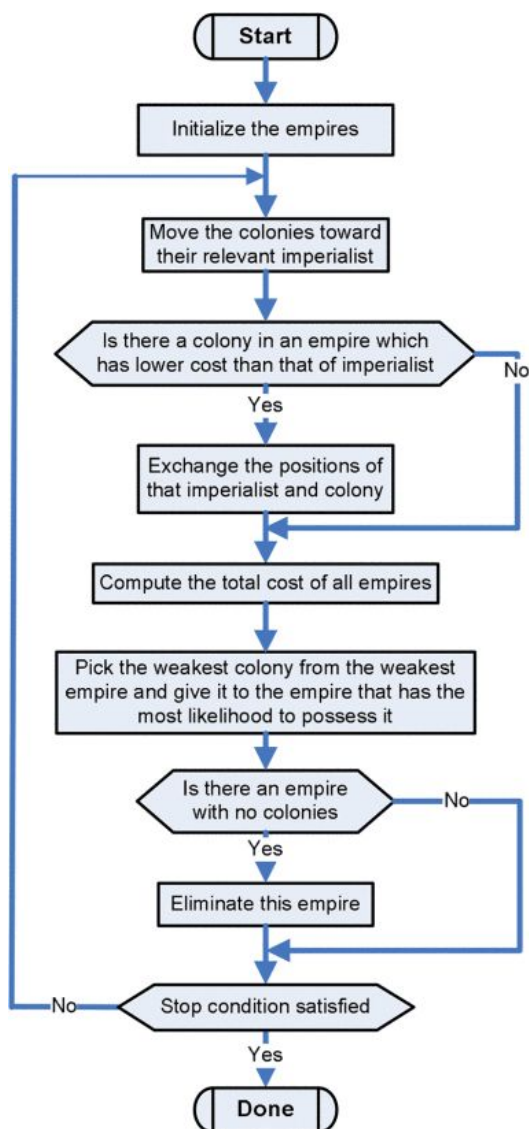
Zadání	3
Algoritmus	4
Pohyb kolonií	4
Migrace kolonií	5
Stop condition	5
Řešení	6
Serial	6
SMP	6
OpenCL	7
Country	7
Imperialist	7
Statistics	7
Výsledky	8
Sphere	8
Schwefel	10
Závěr	13

1. Zadání

Implementujte buď stávající, nebo si navrhnete vlastní, evoluční algoritmus dle specifikovaného rozhraní - viz zdrojové soubory, soubor solver.cpp, funkce `do_solve_generic`. Algoritmus implementujte alespoň ve dvou verzích (SMP, OpenCL nebo MPI), čímž získáte alespoň dva solvery. V souboru descriptor.h si vygenerujte nové GUID pro vaše solvery a zadejte jejich název dle uvedeného vzoru. Nic jiného v tomto souboru neměňte.

2. Algoritmus

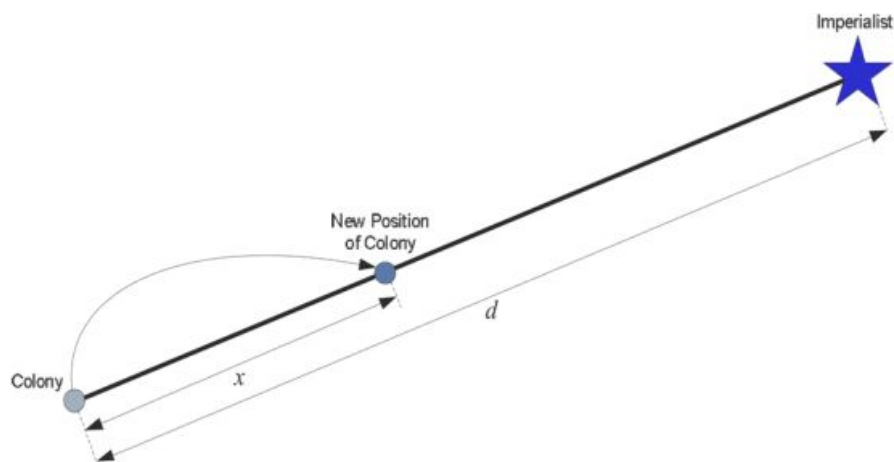
Pro zpracování úlohy jsem zvolil evoluční algoritmus Imperialist Competitive Algorithm (<https://ieeexplore.ieee.org/abstract/document/4425083>). Princip algoritmu (viz obr. 1) spočívá ve vygenerování počáteční populace, která se rozdělí na imperialisty a kolonie. Následně se kolonie pohybují směrem k imperialistům a migrují.



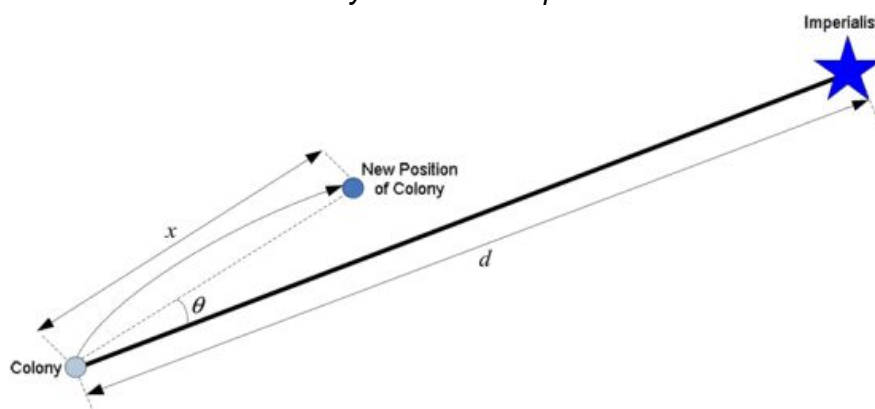
Obr.1: Imperialist Competitive Algorithm

2.1. Pohyb kolonií

Pohyb kolonií se sestává ze dvou částí. Zprvée pohyb kolonie k imperialistovi (obr. 2) a zadruhé výchylka pohybu (obr. 3). Tyto dvě složky jsem implementoval jako $U(0,1)\{V\}$, kde U je vektor náhodných veličin s uniformním rozdělením a $\{V\}$ je směrový vektor.



Obr.2: Pohyb kolonie k imperialistovi



Obr.3: Výchylka pohybu

Z důvodu častého uvíznutí v lokálním minimu v závislosti na počáteční vygenerované populaci jsem s pravděpodobností $1/(2 \cdot \text{velikost populace})$ jsem pro kolonii vygeneroval zcela náhodou novou pozici. Toto řešení přineslo značné zlepšení konvergence do globálního minima.

2.2. Migrace kolonií

Po pohybu kolonie slabších imperií migrují k silnějším. V původní verzi mé implementace jsem migraci řešil dle pravděpodobností migrace k danému imperiu (silnější imperialista má větší pravděpodobnost). Ale při větším počtu imperialistů se stávalo, že několik imperialistů se v síle vyrovnalo a pouze si kolonie navzájem předávali.

Jelikož cílem bylo aby na konci zůstal pouze jeden imperialista, přistoupil jsem k řešení kdy nejslabší imperialista odevzdá svou nejslabší kolonii nejsilnějšímu imperialistovi.

Pokud imperialista ztratí všechny své kolonie, stává se z něj kolonie a je přiřazen nejsilnějšímu imperialistovi.

2.3. Stop condition

Podmínkou zastavení algoritmu je dosažení maximálního počtu generací nebo pokud posledních N řešení je stejných (tolerance $\pm 10^{-k}$). Ideální nastavení se jeví 100 posledních řešení s odchylkou $\pm 10^{-9}$.

3. Řešení

3.1. Serial

Výpočet sériové verze řeší třída *ICA*.

Metody třídy *ICA*:

- *gen_population()* - vygeneruje počáteční populaci
- *evolve()* - iterace algoritmu
- *move_all_colonies()* - pohyb kolonií
- *move_colony()* - přesun kolonie k imperialistovi
- *migrate_colonies()* - migrace kolonií
- *write_solution()* - zapíše výsledné řešení
- *calc_fitness()* - spočítá fitness funkci kolonie
- *calc_fitness_imp()* - spočítá fitness funkci imperialisty ($\text{imp_cost} = \text{imp_cost} + 1/\text{col_size} * \text{mean}(\text{cost_of_colonies})$)
- *calc_fitness_all()* - spočítá fitness funkci všech kolonií

Pomocné metody třídy *ICA*:

- *gen_double()* - vygeneruje náhodné číslo (double)
- *gen_vector()* - vygeneruje vektor náhodných čísel
- *vector_add()* - sečte 2 vektory
- *vector_sub()* - odečte 2 vektory
- *vector_mul()* - vynásobí 2 vektory
- *get_min()* - vrátí kolonii s nejmenší fitness funkcí
- *get_max()* - vrátí kolonii s největší fitness funkcí
- *print_vector()* - vypíše vektor
- *print_population()* - vypíše celou aktuální populaci

3.2. SMP

Výpočet sériové verze řeší třída *ICA_smp*, která dědí od třídy *ICA*. Paralelní výpočty jsou řešeny pomocí knihovny Intel® Threading Building Blocks (tbb). Jsou využívány funkce *parallel_for* a *parallel_reduce* a *concurrent_vector*

Metody třídy *ICA_smp*:

- *gen_population()* - paralelní generování kolonií a rozřazení kolonií mezi imperialisty
- *evolve()* - iterace algoritmu (paralelní zpracování imperialistů)
- *move_all_colonies()* - paralelní pohyb kolonií
- *calc_fitness()* - paralelní počítání fitness funkcí kolonií
- *calc_fitness_imp()* - spočítá fitness funkci imperialisty (*parallel_reduce* pro součet fitness kolonií)
- *get_min()* - *parallel_reduce* pro získání kolonie s nejmenší fitness funkcí
- *get_max()* - *parallel_reduce* pro získání kolonie s největší fitness funkcí

3.3. OpenCL

Výpočet sériové verze řeší třída *ICA_opencil*, která dědí od třídy *ICA_smp*. Tato třída implementuje paralelní výpočty třídy *ICA_smp* přes tbb a zároveň přidává paralelní výpočty vektorů na GPU přes OpenCL. Paralelní výpočty na GPU jsou prováděny až při dimenzi problému 100 a větší z důvodu nákladné režie na provedení výpočtu. Při malé dimenzi problému režie značně převyšuje urychlení výpočtu.

Metody třídy *ICA_opencil*:

- *init()* - inicializuje platformu, zařízení, kontext, frontu a program pro výpočty na GPU
- *move_colony()* - spočítá pohyb kolonie k imperialistovi na GPU
- *vector_op()* - provede vektorový výpočet na GPU
- *vector_add()* - sečte 2 vektory
- *vector_sub()* - odečte 2 vektory
- *vector_mul()* - vynásobí 2 vektory
- *shrRoundUp()* - spočte nejmenší velikost NDRange v závislosti na velikosti a počtu work-group

3.4. Country

Struktura *Country* reprezentuje kolonii, tj. vektor, fitness a indikátor zda se jedná o kolonii nebo imperialistu.

3.5. Imperialist

Struktura *Imperialist* reprezentuje imperialistu, uchovává *Country* reprezentující imperialistu, vektor kolonií a fitness celého impéria.

3.6. Statistics

Třída *Statistics* slouží k měření doby běhu výpočtu, rychlosti konvergence a uložení statistik ve formě souboru .csv.

4. Výsledky

Algoritmu jsem porovnával na funkcích Sphere a Schwefel na dimenzích 8, 30, 100, 500 a 1000 a populaci 7, 15, 25, 40, 60 a 100. Zastavovací podmínka je u funkce Sphere nastavena na 100 posledních řešení s odchylkou $\pm 10^{-9}$. U funkce Schwefel byla vzhledem k časové náročnosti výpočtů při vyšších dimenzích problému omezena na 10 posledních řešení s odchylkou $\pm 10^{-6}$ (nastavení na 100 řešení s odchylkou $\pm 10^{-9}$ pro dimenzi 1000 a populaci 100 dosahovalo u sériové verze čas běhu několik hodin). To se výrazně projevilo na přesnosti výsledného řešení a výrazně zkrátilo čas běhu programu. Pro testovací účely probíhá u varianty OpenCL výpočet na GPU i v případě nižších dimenzí problému.

4.1. Sphere

type	problem_size	population_size	generations	fitness	time[ms]
serial	8	7	2649	1.04184	1678
serial	8	15	956	2.19618	1097
serial	8	25	4386	0.112359	8991
serial	8	40	5369	0.045053	17068
serial	8	60	5063	0.18233	24239
serial	8	100	5513	0.006928	43593
smp	8	7	1408	5.39633	503
smp	8	15	3925	0.256291	3044
smp	8	25	5864	0.098117	8669
smp	8	40	4803	0.060306	13455
smp	8	60	4371	0.067433	21824
smp	8	100	6065	0.007372	69113
openCL	8	7	101	133.846	910
openCL	8	15	101	102.052	1978
openCL	8	25	101	46.0456	3390
openCL	8	40	101	18.8603	5598
openCL	8	60	101	61.1071	8764
openCL	8	100	101	14.3066	15873

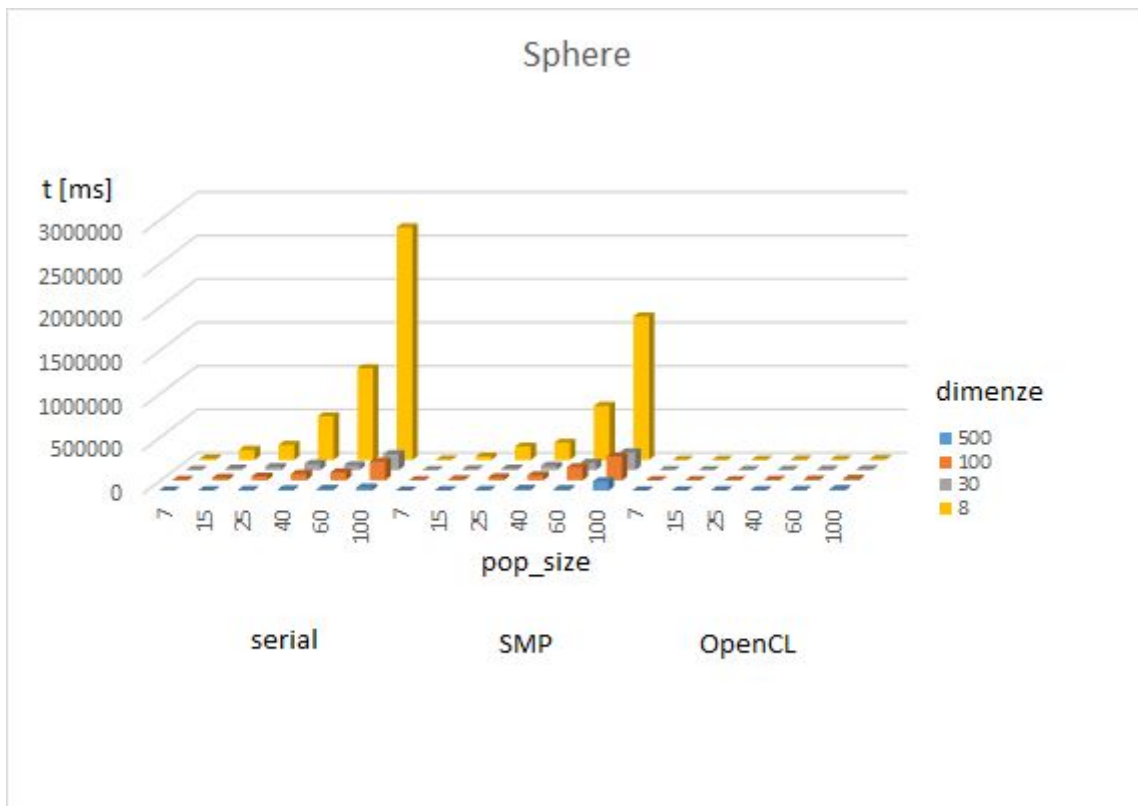
Sphere: dimenze 8

type	problem_size	population_size	generations	fitness	time[ms]
serial	30	7	3027	61.4913	1919
serial	30	15	13855	33.2207	17128
serial	30	25	19505	19.1521	41314
serial	30	40	25075	14.8224	88165
serial	30	60	23779	11.60350	123650
serial	30	100	12860	23.2302	112705
smp	30	7	7037	66.6745	2600
smp	30	15	17488	24.0673	14113
smp	30	25	17916	21.4258	27453
smp	30	40	28950	9.43665	81510
smp	30	60	38033	5.48640	194342
smp	30	100	28856	8.86447	332060
openCL	30	7	101	584.283	900
openCL	30	15	101	521.399	2021
openCL	30	25	101	410.656	3520
openCL	30	40	101	584.579	5795
openCL	30	60	101	401.917	8975
openCL	30	100	101	470.393	16020

Sphere: dimenze 30

type	problem_size	population_size	generations	fitness	time[ms]
serial	500	7	8459	5309.18	14122
serial	500	15	30947	4186.09	116390
serial	500	25	26754	4160.29	172565
serial	500	40	47486	3660.75	502072
serial	500	60	65700	3420.23	1053693
serial	500	100	100000	3081.72	2677175
smp	500	7	6445	5513.69	4813
smp	500	15	24649	4307.56	38419
smp	500	25	55981	3598.9	153110
smp	500	40	41892	3658.85	197613
smp	500	60	77917	3251.59	621658
smp	500	100	100000	2992.3	1649365
openCL	500	7	51	1127.44	483
openCL	500	15	51	1108.42	1038
openCL	500	25	51	1130.77	1771
openCL	500	40	51	1111.48	2998
openCL	500	60	51	1112.42	4689
openCL	500	100	51	1084.05	8284

Sphere: dimenze 500



Graf Sphere

4.2. Schwefel

type	problem_size	population_size	generations	fitness	time[ms]
serial	8	7	29	-1491.4	21
serial	8	15	69	-1481.88	137
serial	8	25	101	-1652.21	258
serial	8	40	124	-1917.18	463
serial	8	60	134	-1898.55	722
serial	8	100	242	-2305.79	1944
smp	8	7	59	-1603.09	36
smp	8	15	114	-1534.87	95
smp	8	25	104	-2063.96	178
smp	8	40	250	-2007.48	743
smp	8	60	81	-2144.39	442
smp	8	100	95	-1761.29	1174
openCL	8	7	11	-777.187	108
openCL	8	15	11	-806.332	231
openCL	8	25	11	-965.251	410
openCL	8	40	11	-1385.23	701
openCL	8	60	11	-1663.13	1059
openCL	8	100	11	-1135.35	1850

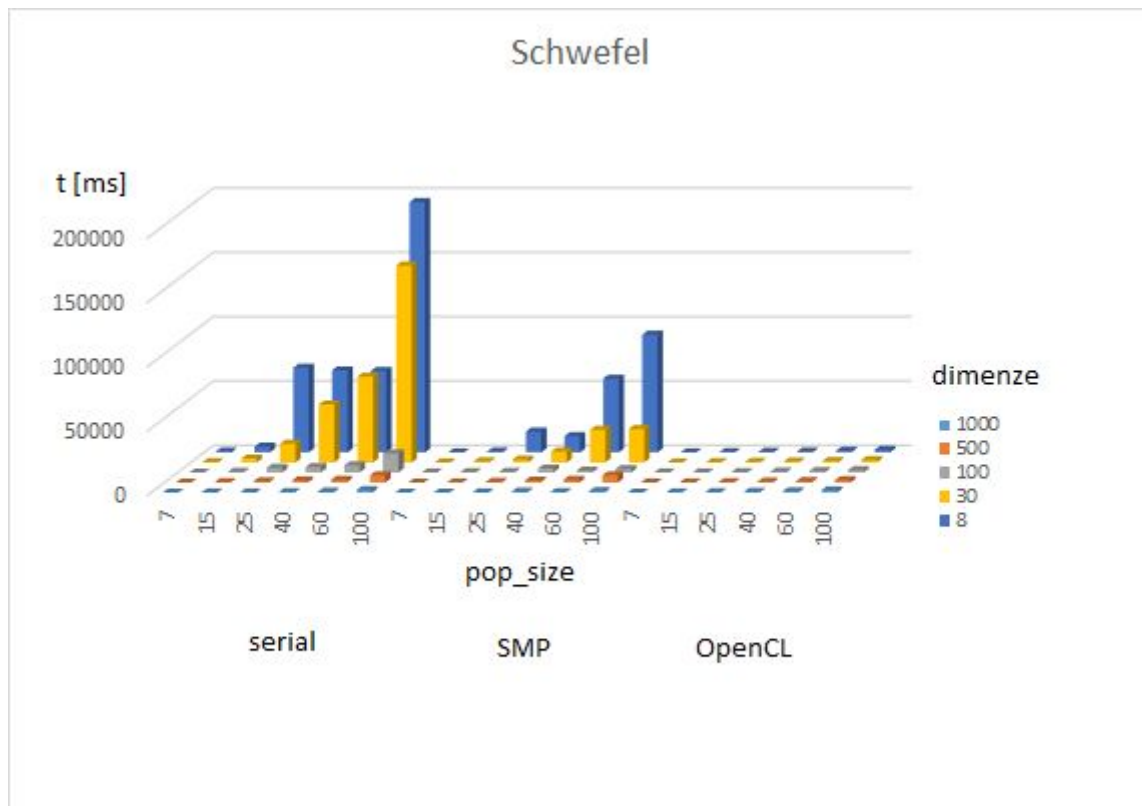
Schwefel: dimenze 8

type	problem_size	population_size	generations	fitness	time[ms]
serial	30	7	42	-3756.78	27
serial	30	15	240	-4602.45	464
serial	30	25	361	-4752.12	1022
serial	30	40	482	-5970.38	1773
serial	30	60	402	-6339.26	2185
serial	30	100	631	-5396.22	5580
smp	30	7	32	-3420.58	27
smp	30	15	307	-4514.78	265
smp	30	25	299	-5288.41	516
smp	30	40	507	-6281.07	1557
smp	30	60	406	-5913.95	2158
smp	30	100	476	-6259.47	5676
openCL	30	7	11	-1759.17	98
openCL	30	15	11	-1466.9	234
openCL	30	25	11	-2386.5	420
openCL	30	40	11	-2172.36	702
openCL	30	60	11	-2576.13	1075
openCL	30	100	11	-2609.02	1885

Schwefel: dimenze 30

type	problem_size	population_size	generations	fitness	time[ms]
serial	500	7	95	-16414.9	298
serial	500	15	683	-26739	2973
serial	500	25	2140	-44142.5	14526
serial	500	40	4060	-44588.5	44944
serial	500	60	3975	-53725.8	66639
serial	500	100	5457	-55261.4	152996
smp	500	7	170	-20130.4	164
smp	500	15	490	-32193.4	891
smp	500	25	684	-32211.5	2166
smp	500	40	1531	-39828.2	8415
smp	500	60	2743	-50028.2	25361
smp	500	100	1389	-45030.1	25949
openCL	500	7	11	-6971.08	100
openCL	500	15	11	-7792.45	219
openCL	500	25	11	-10423.3	411
openCL	500	40	11	-9830.33	711
openCL	500	60	11	-15126.6	1109
openCL	500	100	11	-10786.2	1904

Schwefel: dimenze 500



Graf Schwefel

5. Závěr

Algoritmus jako takový jsem se snažil optimalizovat aby byl co nejúčinnější. Během implementace jsem zjistil, že lépe funguje jednodušší implementace algoritmu, než je popsána v původní práci. Ale zároveň si myslím, že existují lepší implementace či rozšíření algoritmu ICA, které jsou popsány v jiných pracech.

Dle dosažených výsledků mohu konstatovat, že SMP verze algoritmu je vždy rychlejší než sériová. OpenCL verzi nemohu řádně zhodnotit, protože provedenými měřeními jsem zjistil chybu v algoritmu pro OpenCL.

Navzdory tomu u OpenCL mohu říci, že dle zkoušek během implementace byla OpenCL verze kvůli vysoké režii pomalejší pro malé dimenze problému, ale u vyšších dimenzí dosahuje výrazného urychlení oproti sériové i SMP verzi.

Hodně také záleží na počáteční vygenerované populaci. Algoritmus tíhnul ke konvergenci do lokálního minima. Přidáním náhodného členu se konvergence značně zlepšila.