PARALELISMUS V DIFERENCIÁLNÍ EVOLUCI

Jiří Bůžek
Katedra informatiky a počítačů, Přírodovědecká fakulta, Ostravská univerzita v Ostravě,
jirkabuzek705@gmail.com

Abstrakt

Cílem tohoto článku je představení paralelní implementace algoritmu diferenciální evoluce v programovacím jazyce C#. Paralelní implementace byla porovnána s neparalelní na třiceti testovacích funkcích z CEC14.

Klíčová slova: *Diferenciální evoluce; paralelismus.*

Úvod

Mějme účelovou funkci $f: D \to \mathbb{R};\ -\infty < f(x) < \infty, \forall x \in D, D \subseteq \mathbb{R}^d$, kde D je prohledávaný prostor a $d \in \mathbb{N}$ je dimenze úlohy. Úkolem globální optimalizace je najít takový bod, který je globálním minimem nebo maximem účelové funkce. Pro řešení úlohy nalezení globálního minima máme najít bod $x^* \in D$, pro které platí, že $f(x^*) \leq f(x)$, pro $\forall x, x \in D$. Přitom předpokládejme, že D je souvislá ohraničená oblast $D = \prod_{i=1}^{d} < a_i, b_i >,\ a_i < b_i, i = 1,2, \dots, d$, kde a_i je dolní a b_i je horní hranice.

Diferenciální evoluce, dále jen DE, byla vyvinuta v roce 1995 Stornem a Pricem [1]. Projevila se jako robustní a spolehlivá a stala se velmi používaným evolučním algoritmem pro řešení problému spojité globální optimalizace. DE pracuje s populací Np d-rozměrných vektorů. Každá nová populace se generuje z předchozí, použitím metod mutace, křížení a selekce nad jednotlivými jedinci dokud není nalezeno přijatelné řešení nebo dokud není splněna jiná podmínka ukončení.

Tento článek se věnuje paralelismu v klasické DE, varianta de/rand/1/bin. Ve zprávě [2], na kterou tento článek bezprostředně navazuje, bylo navrženo a implementováno řešení paralelizace pro klasickou DE a její adaptivní varianty. V původním řešení se při generování populace paralelně provádí rutiny mutace, křížení a selekce. Cílem tohoto článku je představení nové implementace v programovacím jazyce C#, která řeší problém False Sharing, jemuž se autor v [2] nevěnoval. Hlavní rozdíl mezi původní a novou implementací je v odlišné interpretaci populace na úrovni dat. V prvních dvou sekcích tohoto článku jsou uvedeny významné problémy, které se vyskytly při paralelizaci algoritmu. V dalších následujících sekcích se nachází popis nové implementace a výsledky experimentálního testování na testovacích funkcích z CEC14 [5].

Race Condidion

Race Condition je chyba ve vícevláknových aplikacích, která nastává při přístupu dvou a více vláken ke sdílené proměnné. Algoritmus s touto chybou často vyprodukuje různý výsledek pro různý běh, i když by měl být pro všechny běhy stejný. Výsledek algoritmu nemůžeme s jistotou předpovídat.

Race Condition ilustruje následující příklad. Přepokládejme, že dvě vlákna současně inkrementují proměnnou ***i***. Tato operace se vykonává v následujících krocích: načtení hodnoty proměnné ***i*** do registru, vlastní inkrementace hodnoty v registru a uložení výsledné hodnoty zpátky do ***i***. Vlákno, které načetlo a inkrementovalo hodnotu proměnné ***i*** v registru, může být

předběhnuto druhým vláknem (preempce). Toto druhé vlákno vykoná všechny tři kroky. První vlákno pak pokračuje ve své činnosti a přepisuje hodnotu proměnné $\boldsymbol{i}$ svou hodnotou, aniž by bralo v potaz, že jeho hodnota byla změněna druhým vláknem. Tento problém lze vyřešit synchronizací přístupu vláken k proměnným.

U varianty de/rand/1/bin se průměrně vygeneruje $d + 4$, náhodných čísel na jedno vyhodnocení účelové funkce. Synchronizace přístupu k jedné instanci generátoru je pro tento obrovský počet vygenerovaných čísel velmi nevýhodná. Proto byl v implementaci nasazen algoritmus Mrg32k3a. Algoritmus Mrg32k3a [3] navržený Pierrem L'ecuyerem s periodou délky $p \approx 2^{191}$ je schopen vytvořit instance, jejichž stavy jsou o Z, $Z = 2^{127}$ kroků vpřed před poslední vytvořenou instancí. Při vytvoření nové instance se generátor překlopí do Z-tého stavu, vypočtením Z-té iterace své funkce vycházející ze stavu posledně vytvořené instance. Mezi dvěma po sobě vytvořenými instancemi leží sekvence 2^{127}náhodných čísel, která je nezávislá na sekvenci druhé instance. Tyto instance splňují podmínku nezávislosti a jejich periody jsou dostatečně dlouhé.

False Sharing

Mějme dvě vlákna, souběžně pracující se dvěma různými proměnnými, které se nacházejí na stejném řádku v L2 cache paměti. Když první vlákno zapisuje do proměnné, je celý sdílený řádek v paměti překlopen do stavu invalid protokolem typu MESI nebo podobným. Druhé vlákno při čtení nebo zápisu do své proměnné pak musí její hodnotu znovu načíst z hlavní paměti, což způsobuje extrémní režii navíc. Tato událost se obecně označuje jako L2 cache miss. Problém False Sharing [4], dále jen FS, se dá vyřešit vyhýbáním se alokací souvislých bloků paměti, přitom můžeme předpokládat, že: dva objekty inicializované bezprostředně po sobě se nacházejí v paměti často blízko u sebe, dvě sousední buňky pole spolu sousedí i v paměti apod.

Profilovací analýzou bylo zjištěno, že problém FS ve výchozí implementaci [2] má velmi negativní dopad na celkový výkon algoritmu. V nové implementaci je populace jedinců pevně rozdělena napříč vlákna v jednorozměrném poli tak, aby každá dvě vlákna nepracovala s jedinci, kteří by se teoreticky mohli nacházet v paměti blízko u sebe. Hlavní populace je logicky oddělena od populace zkusmých vektorů. Souvislý blok každé populace je odsazen 128 bajty od nultého indexu, protože pole v jazyce C# obsahuje meta člen array size (velikost pole). Hodnota meta členu array size je čtena při každé indexaci jakéhokoliv prvku pole, proto by se při zapisování kteréhokoliv vlákna do oblasti blízké nultému indexu (menší než 128 nebo 64 bajtů) projevoval problém FS mezi všemi ostatními vlákny. Mezi jednotlivými zpracovávanými bloky je dále vsunuta mezera délky 128 bajtů tak, aby bylo zamezeno sdílení řádku v cache paměti mezi hraničními jedinci populace. Tabulka 1 zobrazuje výsledky profilovací analýzy pro jeden běh de/rand/1/bin neparalelní, původní paralelní a nové paralelní implementace na testovací funkci Ackley při stejných vstupních parametrech. Původní varianta má počet vykonaných instrukcí procesoru o hodně vyšší než ostatní varianty díky neošetřenému problému FS.

Tabulka 1. Výsledky profilovací analýzy.

	počet vykonaných instrukcí	L2 cache misses
neparalelní implementace	3921E+06	711
výchozí paralelní s FS	4598E+06	70152
nová paralelní bez FS	3728E+06	1782

Vlastní implementace

Po počáteční inicializaci populace se podle předem daného počtu vytvoří pracovní vláka. V každém vlákně je na začátku jeho životního cyklu vypočten interval pracovní oblasti v datové struktuře populace, je inicializována jeho instance generátoru náhodných čísel a proměnné pro dočasné ukládání vygenerovaných zkusmých vektorů a hodnot vyhodnocených účelových funkcí. Potřebný počet alokací objektů je $2 * n$, kde n je počet vytvořených vláken, což velmi snižuje režii spojenou s Garbage Collectorem, která je pro více vláknové aplikace velkou zátěží. Pomocí synchronizačního mechanismu ManualResetEventSlim provádí každé vlákno pro svou pracovní oblast celý cyklus vygenerování nové populace. Po ukončení každého jednoho cyklu jsou vlákna pozastavena, synchronizována a jsou otestovány ukončovací podmínky. V této implementaci se tedy paralelně provádí křížení, mutace, vyhodnocení účelové funkce a selekce. Paralelní zpracovávání statických bloků, oproti obrácenému přístupu uvedeného v [2], je kompromisem mezi problémem FS a prostorovou lokálností dat. Jednotliví jedinci nemusí být mezi sebou odděleni 128 bajty, 128 bajty jsou odděleny pouze zpracovávané bloky. Problém FS způsobuje větší režii než špatná prostorová lokálnost.

Experimentální testování

Paralelní a neparalelní implementace byly otestovány podle CEC14 [5] pro dimenzi úlohy $d = 30$. Velikost populace byla nastavena na $Np = 50$ a řídící parametry F a Cr byly nastaveny na $F = 0.5, Cr = 0.5$. Výsledky testování se v kvalitě nalezeného řešení významně neliší, odlišnosti jsou způsobeny stochastickou povahou algoritmů. Tabulka 2 uvádí průměrné délky trvání jednoho běhu algoritmu pro každou testovací funkci v milisekundách. Symbol $\bar{t}_n$ v tabulce 2 představuje průměrný celkový uběhlý čas jednoho běhu neparalelní varianty a symbol $\bar{t}_p$ představuje tentýž čas pro paralelní variantu. Zrychlení je vypočteno jako podíl $\bar{t}_n$ a $\bar{t}_p$. Z důvodu korektnosti měření u paralelní implementace byly vždy pro každý jeden běh vytvořena dvě vlákna a jejich afinita byla nastavena tak, aby každé vlákno běželo na vlastním fyzickém jádře procesoru. Vzhledem k tomu, že paralelní varianta používá dvě vlákna, lze očekávat, že se zrychlení bude přibližovat ke 2.

Tabulka 2. Výsledky testování.

	$\bar{t}_n$	$\bar{t}_p$	zrychlení		$\bar{t}_n$	$\bar{t}_p$	zrychlení
f1	3190	1620	1.97	f16	3001	1560	1.92
f2	3057	1606	1.90	f17	3599	1854	1.94
f3	2356	1269	1.86	f18	3153	1656	1.90
f4	2424	1289	1.88	f19	19403	9464	2.05
f5	2871	1481	1.94	f20	3144	1652	1.90
f6	83594	40771	2.05	f21	3501	1810	1.93
f7	2925	1531	1.91	f22	6447	3214	2.01
f8	1448	796	1.82	f23	23510	11158	2.11
f9	2738	1447	1.89	f24	15587	7725	2.02
f10	2541	1347	1.89	f25	16974	8375	2.03
f11	3877	2002	1.94	f26	119608	57567	2.08
f12	32251	15665	2.06	f27	116091	56796	2.04
f13	2384	1285	1.86	f28	37509	17933	2.09
f14	2458	1312	1.87	f29	34249	17115	2.00
f15	2917	1514	1.93	f30	21345	10796	1.98

Výsledky měření z tabulky 2 ukazují, že výsledné zrychlení roste s rostoucí časovou náročností vyhodnocení testovací funkce. Nejhoršího zrychlení dosáhla testovací funkce *f8*. Průměrný čas běhu pro funkci *f8* je natolik malý, že se v něm ve větší míře projevují režie spojené se synchronizacemi vláken. U časově náročnějších funkcí například *f25* tyto režie zabírají minimální část naměřeného času.

Závěr

Nová verze paralelní implementace algoritmu de/rand/1/bin minimalizující problém False Sharing je rychlejší a stabilnější než původní verze. Použití metod z knihovny Task Parallel Library je nevhodné pro řešení paralelizace v diferenciální evoluci. Omezením počtu alokací objektů v pracovních vláknech na minimum a použitím vhodně uspořádaných datových struktur je kód optimalizován pro hardwarové platformy s architekturou MIMD.

Zdrojové kódy algoritmu, dokumentace a API jsou dostupné volně ke stažení na adrese http://adaptivedenet.codeplex.com/.

Poděkování

Rád bych touto formou poděkoval panu doc. Ing. Josefu Tvrdíkovi, CSc. za jeho cenné rady, nápady, připomínky a odborné vedení.

Literatura

[1.] STORN, R., PRICE, K. *Differential evolution-a simple and effcient heuristic for global optimization over continuos spaces.* Journal of global optimization, 11(4):341-359,1997.

[2.] BŮŽEK, J. *Paralelismus v diferenciální evoluci a jejích adaptivních variantách.* Zpráva. Ostravká univerzita v Ostravě, Přírodovědecká fakulta, 2013. Dostupné na Internetu: http://www1.osu.cz/~tvrdik/down/files/zprava2_Buzek.pdf.

[3.] L'ECUYER, P., SIMARD, R., CHEN, E. J., KELTON, W. D. *An object-oriented random-number package with many long streams and substream.* Operations Research, 50(6):1073-1075, 2002.

[4.] INTEL. *Avoiding and Identifying False Sharing Among Threads*. Intel Guide for Developing Multithreaded Application. Intel, 2011.

[5.] LIANG, J. J., QU, B. Y., SUGANTHAN, P. N. *Problem Definitions and Evaluation Criteria for the CEC 2014 Special Session and Competition on Single Objective Real-Parameter Numerical Optimization.* Computational Intelligence Laboratory, 2013.

Abstract

The purpose of this article is to introduce a parallel implementation of differential evolution in C# language. Parallel implementation was compared to its sequential version on thirty test functions from CEC14 suite.