VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

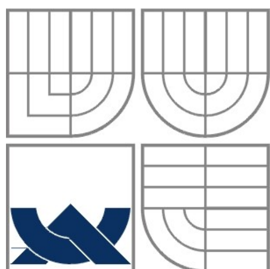
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

EFEKTIVNÍ IMPLEMENTACE GENETICKÉHO ALGORITMU S VYUŽITÍM VÍCEJÁDROVÝCH CPU

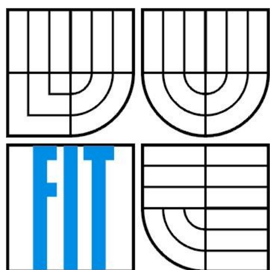
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. MIROSLAV KOUŘIL
AUTHOR

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

EFEKTIVNÍ IMPLEMENTACE GENETICKÉHO ALGORITMU S VYUŽITÍM VÍCEJÁDROVÝCH CPU

THE EFFICIENT IMPLEMENTATION OF THE GENETIC ALGORITHM USING MULTICORE PROCESSORS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MIROSLAV KOUŘIL

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JIŘÍ JAROŠ

BRNO 2010

Abstrakt

Tato práce se zabývá akcelerací pokročilého genetického algoritmu. Pro implementaci byly zvoleny diskrétní i spojitá varianta genetického algoritmu typu UMDA. Hlavní částí akcelerace bylo využití SSE sady. Pomocí této sady byly zrychleny zejména funkce pro výpočet fitness a vzorkování nové populace. Dále byl implementován pseudonáhodný generátor čísel, který také pracuje s SSE sadou. Po této implementaci dosáhla diskrétní varianta algoritmu zrychlení 4,6. Na závěr byly algoritmy upraveny pro využití systému OpenMP, který umožňuje spouštění bloků programu ve více vláknech. Ukázalo se, že pro paralelní zpracování se příliš nehodí spojitá verze algoritmu, neboť její činnost je relativně jednoduchá. Oproti tomu diskrétní verze algoritmu jsou pro paralelizaci velmi vhodné, implementované verze dosáhly celkového zrychlení 4,9 a 7,2.

Abstract

This diploma thesis deals with acceleration of advanced genetic algorithm. For implementation, discrete and continuos versions of UMDA genetic algorithm were chosen. The main part of the acceleration is the utilization of SSE instruction set. Using this set, the functions for calculating fitness and new population sampling were accelerated in particular. Then the pseudorandom number generator that also uses SSE instruction set was implemented. The discrete algorithm reached the speed of up to 4,6 after this implementation. Finally, the algorithms were modified so that the system OpenMP could be used, which enables the running of blocks of code in more threads. The continuous version of algorithm is not convenient for parallelization, because computational complexity of that algorithm is low. In comparison, the discrete versions of algorithm are really appropriate for parallelization. Both the implemented versions reached the total acceleration of up to 4,9 and 7,2.

Klíčová slova

Pokročilé evoluční algoritmy, EDA, UMDA, paralelizace, Amdahlův zákon, Gustafsonův zákon, Flynnova klasifikace, SIMD, SSE, OpenMP

Keywords

Advanced evolutionary algorithms, EDA, UMDA, Parallelization, Amdahl's law, Gustafson's law, Flynn's taxonomy, SIMD, SSE, OpenMP

Citace

Miroslav Kouřil: Efektivní implementace genetického algoritmu s využitím vícejádrových CPU, diplomová práce, Brno, FIT VUT v Brně, 2010

Efektivní implementace genetického algoritmu s využitím vícejádrového CPU

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Jiřího Jaroše. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Miroslav Kouřil
25.05.2010

Poděkování

Rád bych poděkoval panu Jiřímu Jarošovi za vedení diplomové práce a také za jeho podporu, trpělivost, rady a inspiraci při vypracování této diplomové práce.

© Miroslav Kouřil, 2010
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Obsah....1
1 Úvod....3
2 Evoluční algoritmy....4
2.1 Genetické algoritmy....4
2.1.1 Kódování chromozomu....6
2.1.2 Výběr rodičů....6
2.1.3 Tvorba nového jedince....7
2.2 Pokročilé genetické algoritmy....9
2.2.1 UMDA....11
2.2.2 BDMA....13
2.2.3 BOA....15
2.3 Další druhy evolučních algoritmů....15
2.3.1 Evoluční strategie....15
2.3.2 Genetické programování....16
2.4 No free lunch teorém....16
3 Paralelizace programu....17
3.1 Granularita paralelismu....17
3.2 Výpočet zrychlení při paralelizaci....19
3.3 Klasifikace paralelních systémů....21
3.4 OpenMP....23
4 SSE jednotky....25
4.1 Verze SSE jednotek....25
4.1.1 SSE1....25
4.1.2 SSE2....27
4.1.3 SSE3....27
4.1.4 SSE4....28
4.1.5 SSE5....28
4.2 XMM registry....29
4.3 Podpora SSE sady pro jazyk C....31
5 Vybrané problémy pro řešení genetickým algoritmem....32
5.1 Hledání minima spojité funkce....32
5.2 Řešení problému batohu....33
6 Základní implementace programu....34
6.1 Algoritmus pracující s reálnými čísly....34

6.2 Algoritmus pracující s celými čísly....37
6.3 Možnosti akcelerace jednotlivých částí UMDA algoritmu....39
7 Analýza programu....40
7.1 Profilovací nástroj....40
7.2 Diskrétní verze algoritmu s reálnými čísly....40
7.3 Spojitá verze algoritmu s reálnými čísly....41
7.4 Algoritmus s celými čísly....42
7.5 Souhrn výsledků profilování....42
8 Generátor náhodných čísel....43
8.1 Generátor pseudonáhodných čísel....43
8.1.1 Lineární kongruentní generátor....43
8.1.2 SSE implementace generátoru....44
9 Akcelerace genetického algoritmu pomocí SSE jednotek....47
9.1 Generování pseudonáhodných čísel....47
9.2 Výpočet fitness funkce....48
9.2.1 Fitness funkce programu s reálnými čísly....48
9.2.2 Fitness funkce programu s celými čísly....50
9.2.3 Zrychlení programu implementací SSE fitness funkce....52
9.3 Vzorkování nové populace....52
9.3.1 Vzorkovací funkce programu s reálnými čísly využívajícím diskretizaci hodnot....53
9.3.2 Zrychlení programu implementací SSE vzorkování....54
9.4 Souhrn zrychlení dosaženého využitím SSE jednotek....55
10 Paralelizace programu pomocí OpenMP....57
10.1 Postup paralelizace....57
10.2 Zrychlení dosažené paralelizací systémem OpenMP....58
11 Celková akcelerace implementovaných algoritmů....59
12 Závěr....61
12.1 Postup řešení....61
12.2 Dosažené výsledky....61
12.3 Možnosti dalšího vývoje projektu....62
Literatura....63
Seznam příloh....66

1 Úvod

Římský filozof M. T. Cicero prohlásil, že „Dáme-li se vést přírodou, nemůžeme v ničem pochybit“. Přesně této myšlenky se drží evoluční algoritmy. Jejich snahou je co nejlépe napodobit evoluční procesy v přírodě, zejména dědičnost, přirozený výběr, křížení či mutaci.

Při snaze o zlepšení funkce evolučních algoritmů byly vyvinuty pokročilé evoluční algoritmy, které se snaží hledat dobré vlastnosti jedince, které pak zkouší předat do další generace. Zásadním rozdílem oproti klasickým evolučním algoritmům je aktivní hledání dobrých částí v jedinci.

Tyto algoritmy jsou velmi dobré například pro řešení obtížných matematických funkcí, nicméně pokud dokážeme libovolný problém popsat tak, že pro něj dokážeme v počítači vypočítat vhodnost navrhovaného řešení, pak můžeme evolučním algoritmem řešit cokoliv. Problémem se stává fakt, že ne vždy je algoritmem nalezeno řešení nebo je potřeba pro hledání řešení velmi dlouhá doba.

Ke zkrácení potřebného času pro výpočet můžeme využít paralelních prvků v moderních počítačích. Na jejich procesorech bývá několik jader, ve kterých lze současně provádět výpočet. Navíc v rámci každého jádra je dostupná sada instrukcí, která také pracuje paralelně. Cílem této práce je implementace pokročilého evolučního algoritmu, který bude poté paralelizován s využitím vícejádrového procesoru.

Na tomto procesoru bude využita instrukční sada SSE, která umožňuje paralelizaci matematických operací v rámci jednoho jádra. Původně byla sada dostupná jen pro reálná čísla, ale ve vyšších verzích byla přidána i podpora pro celá čísla. Dalším výstupem práce bude porovnání zrychlení výpočtu s celými čísly oproti zrychlení výpočtu s reálnými čísly.

Na závěr bude algoritmus upraven pro využití systému OpenMP, který umožňuje spuštění programu ve více vláknech.

2 Evoluční algoritmy

Evolučními algoritmy nazýváme množinu algoritmů, které k výpočtu zadaného problému využívají procesy, kterými se snaží napodobit principy evoluce v přírodě. Všechny tyto algoritmy mají společnou základní myšlenku, která využívá všeobecné představy o hnacích silách evoluce: vychází z populace jedinců, u kterých můžeme určit jejich kvalitu a dle té jsou pak vybráni rodiče, z nichž se následně vytvoří další populace. Tímto postupem se napodobuje proces evoluce, jak je v současné době chápán.

Neodarwinismus

Jako paradigma evolučního procesu v přírodě považujeme neodarwinismus, který vychází z principů darwinismu [16], Mendelovy genetické teorie [17] a populační genetiky [18].

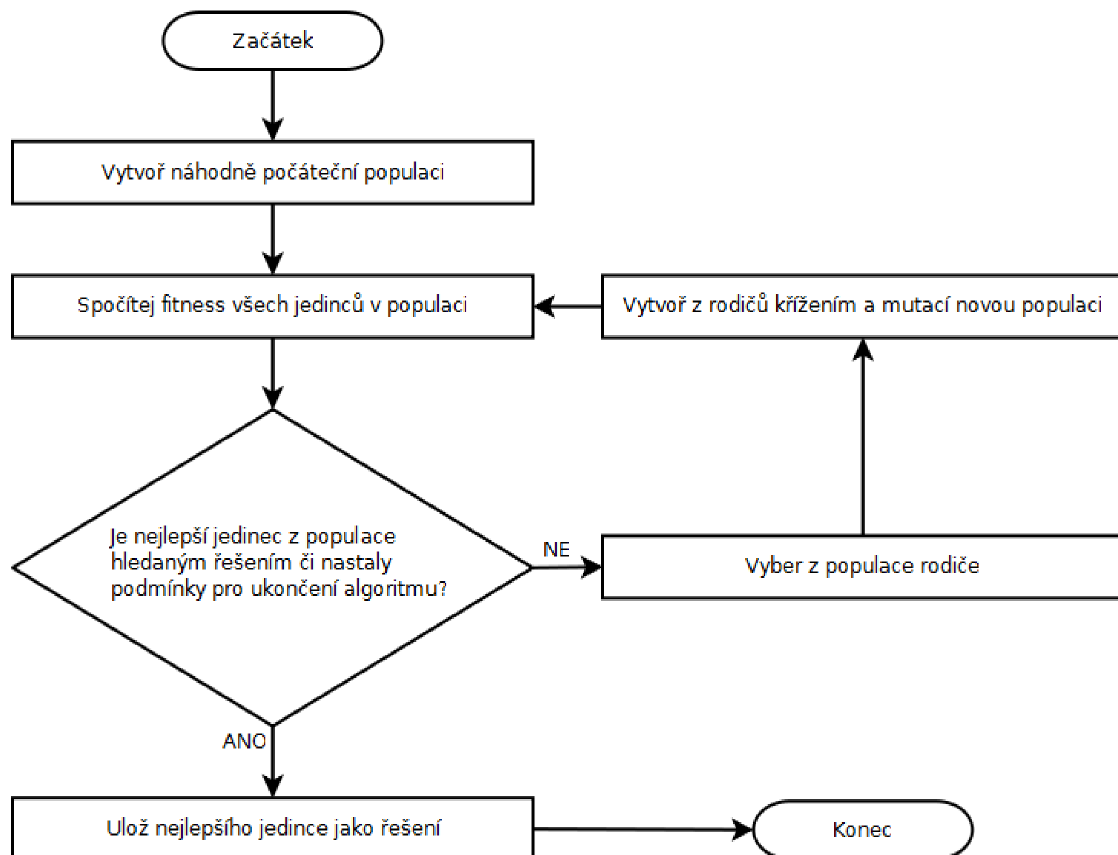
Neodarwinismus [6] předpokládá, že druhová odlišnost jedinců v generaci je způsobena náhodnou mutací genetické informace. Pokud je tato mutace pro jedince výhodná, pak se dále množí a svoji mutovanou genetickou informaci předává na potomky. Naopak pokud daná mutace jedince znevýhodňuje oproti ostatním, pak má tento jedinec problémy s množením a postupně tato mutace z populace vymizí. Fakt, že v populaci přežívají jen jedinci, kteří se dokáží lépe přizpůsobit aktuální situaci, se nazývá selekční tlak. Dále se předpokládá schopnost dědění vloh pro jednotlivé znaky z generace na generaci v nezměněné podobě. Tomuto přenosu se říká tvrdá dědičnost.

Existuje řada dalších teorií, které poukazují na nedostatky neodarwinismu, například na možnosti cílené mutace či dědičnosti vlastností získaných za života jedince (měkká dědičnost). Jiný pohled na evoluci poskytuje například teorie sobeckého genu [20] či teorie zamrzlé plasticity [6].

2.1 Genetické algoritmy

Klasickým představitelem evolučních technik jsou genetické algoritmy [1]. Tyto algoritmy nejčastěji slouží k řešení složitých problémů, u kterých selhávají běžné výpočetní postupy. Základními stavebními prvky genetického algoritmu jsou dědičnost, přirozený výběr, křížení a mutace.

Genetický algoritmus pracuje s populací jedinců, z níž každý je možným řešením daného problému. Z těchto jedinců vybere nejvhodnější, které pak mezi sebou kříží, čímž vznikají noví potomci. Při křížení může s malou pravděpodobností nastat mutace jedince, která pomáhá udržet variabilitu jedinců v populaci. Vývojový diagram klasického genetického algoritmu je na obrázku 2.1.



Obr. 2.1: Vývojový diagram genetického algoritmu

V další části textu bude popsána funkčnost jednotlivých etap genetického algoritmu. Pro přesnější popis je nutné vymezení následujících pojmů:

- *Fitness* – hodnota, která vyjadřuje kvalitu jedince. U genetických algoritmů je zvykem, že čím nižší je fitness, tím je jedinec vhodnější jako řešení.
- *Chromozom* – reprezentace jedince v populaci. V rámci genetického algoritmu se často jedná o číslo, které bitově prezentuje řešení hledaného problému.
- *Gen* – chromozom se dělí na jednotlivé geny. Pokud bereme chromozom jako číslo, pak je genem jeden bit.
- *Alela* – hodnota, kterou může nabýt gen. Při binárním kódování se jedná o hodnotu 0 nebo 1.

Pro správnou funkci genetického algoritmu je velmi důležité vhodné kódování chromozomu, správný výběr rodičů a způsob tvorby nového jedince z populace rodičů. Tyto části budou podrobněji popsány v následujícím textu.

2.1.1 Kódování chromozomu

Při volbě kódování chromozomu musíme navrhnout takový zápis, aby umožnil aplikaci dalších funkcí genetického algoritmu a současně byl co nejjednodušší z hlediska rychlosti výpočtu.

Proto pokud to řešený problém umožňuje, tak volíme binární kódování či práci s polem čísel. Samozřejmě toto není možné vždy, nicméně čím složitější bude chromozom, tím pomalejší bude vlastní výpočet, což způsobí pomalou konvergenci algoritmu.

Klasickým příkladem pro binární kódování je problém batohu [2], kdy hodnotou 0 nebo 1 na pozici řetězce určujeme, zda věc (u které pořadí = pozice hodnoty v řetězci) bude zařazena do batohu či nikoli.

2.1.2 Výběr rodičů

V rámci genetického algoritmu je výběrem rodičů simulován selekční tlak na jedince, kteří se mezi sebou budou množit. Z hlediska funkčnosti algoritmu je potřeba zachovat dva pohledy:

1. Výběr nejlepších jedinců
2. Zachování variability jedinců

Pokud bychom pouze vybrali pro křížení nejlepší jedince z populace, tak může nastat stav, že všichni vybraní jedinci jsou velmi podobní, tudíž algoritmus může lehce uváznout v lokálním řešení. Naopak při absolutně náhodném výběru opomíjíme kvalitní jedince a algoritmus může velmi pomalu konvergovat k nejlepšímu řešení. Proto je třeba najít kompromis a vybrat dostatečně kvalitní jedince s dobrou variabilitou.

Proporcionální selekce

Jedná se o náhodný výběr, ve kterém je pravděpodobnost výběru jedince do populace rodičů přímo úměrná hodnotě jeho fitness. Při tomto postupu je možné, že se v populaci vyskytne jedinec s relativně vysokým fitness, který se velmi rychle rozšíří i mezi ostatní řešení a ztratí se variabilita. Proto se při tomto druhu selekce používají další postupy, které redukují rozdíl mezi kvalitou nejlepšího a nejhoršího jedince. Nejčastěji jsou používány komprimace fitness funkce a sigma škálování. Přesnější popis těchto funkcí v [20].

Tento druh selekce se často nazývá ruletou, neboť postup výběru je podobný točení ruletou, kde velikost jednotlivých sekcí odpovídá kvalitě jedince (velikosti jeho fitness).

Lineární uspořádání

Při tomto postupu je nutné seřadit populaci jedinců dle fitness vzestupně a pak přepočítat pravděpodobnost výběru jedince dle vztahu:

$$P_{lin-rank}(i)=(\frac{2-s}{N})+\left(\frac{2i(s-1)}{N(N-1)}\right) \qquad (2.1)$$

$$i \in 1,2,\ldots..N$$

Exponenciální uspořádání

Toto uspořádání také vyžaduje seřazenou populaci dle fitness, ale pravděpodobnost výběru jedince jako rodiče není rozložena lineárně, ale exponenciálně dle vzorce:

$$P_{(\exp-rank)}(i)=\frac{1-e^{-i}}{c} \qquad (2.2)$$

$$i \in 1,2,\ldots..N$$

$$0<c<1$$

Turnajová selekce

Tato selekce vybere z populace několik jedinců, ze kterých vybere za rodiče jedince s nejlepším fitness. Tento postup opakuje tolikrát, kolik je rodičů.

Vlastní selekce jedinců je velmi jednoduchá a kvalitní z hlediska zachování variability rodičů. Oproti lineárnímu a exponenciálnímu uspořádáni nevyžaduje setřídění populace.

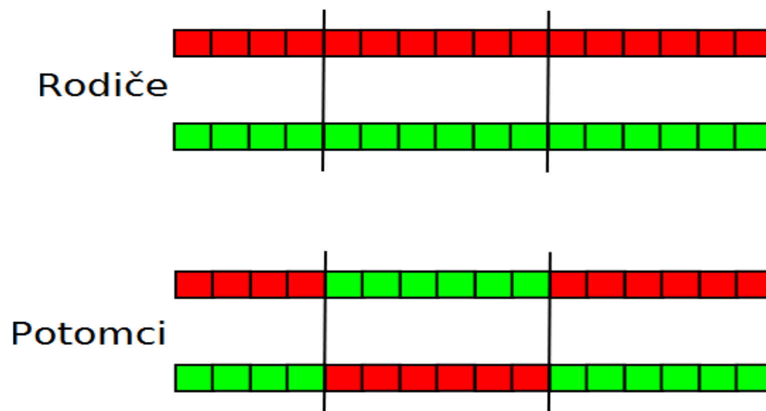
2.1.3 Tvorba nového jedince

Při tvorbě nového jedince se v klasické verzi genetického algoritmu používá křížení rodičů a mutace. Při křížení se snažíme získat výhodné prvky z obou rodičů. Při mutaci se snažíme zachovat variabilitu nově vzniklé populace.

Křížení

Cílem křížení je získat jedince, který bude mít kombinaci vlastností rodičů, která může být jako řešení hledaného problému vhodnější než u jednotlivých rodičů.

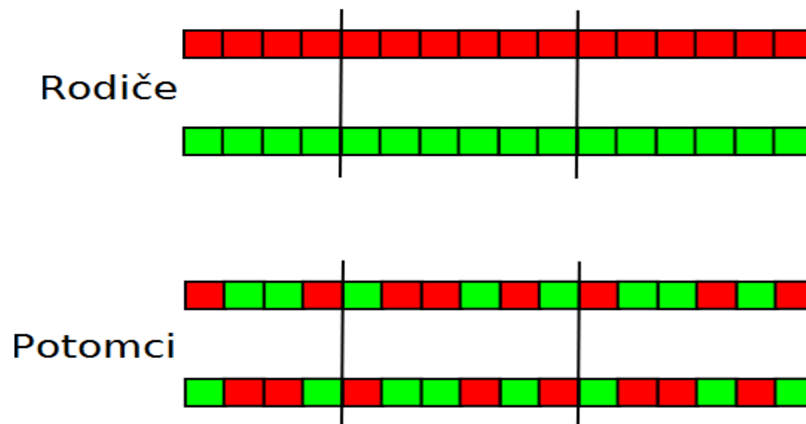
Nejčastější implementací je jedno a vícebodové křížení, při kterém se rekombinace genů chromozomů provádí na jednom (či více) místech [20]. U jednobodového křížení jsou ve vybraném místě části rodičů vzájemně prohozeny. Touto cestou vzniknou dva noví potomci. U vícebodových křížení je logika stejná, jen se vyměňuje více úseků. Příklad dvoubodového křížení je na obrázku 2.2.



Obr. 2.2.: Příklad dvoubodového křížení

Další způsob křížení je uniformní křížení, které se používá pro binární chromozomy. Při uniformním křížení procházíme chromozomy a provádíme výměnu jednotlivých genů na základě pravděpodobnosti p_u.

Nevýhodou tohoto přístupu je časté rozbíjení kvalitních bloků v rámci rodičovského chromozomu, naopak velkou výhodou je vysoká variabilita nově vytvořené populace. Proto se toto křížení využívá při řešení složitých vícerozměrných funkcí s mnoha lokálními extrémy. Často se toto křížení používá, pokud algoritmus předčasně konverguje k lokálnímu extrému. Příklad uniformního křížení je na obrázku 2.3.



Obr. 2.3.: Příklad uniformního křížení

Mutace

Mutace má obvykle velmi malou četnost výskytu, ale je velmi důležitá pro správnou funkčnost genetického algoritmu. Často bývá jedinou cestou, jak se dostat z lokálního minima, ve kterém řešení populace dočasně uvázlo. Princip je velmi jednoduchý – nad každým bitem v celé populaci provedeme náhodný experiment s pravděpodobností úspěchu p_m . Vyjde-li pozitivní výsledek, pak provedeme inverzi příslušného bitu. Příklad mutace bitu a její vliv na hodnotu jedince je uveden na obrázku 2.4.

$$1\mathbf{0}101101(173d) \rightarrow 1\mathbf{1}101101(237d)$$

Obr. 2.4.: Příklad změny hodnoty chromozomu při mutaci jednoho genu

2.2 Pokročilé genetické algoritmy

Možnou slabinou genetického algoritmu je skutečnost, že při křížení jedinců se často rozbíjejí tzv. stavební bloky. Tyto bloky jsou sekce chromozomu, které jsou přínosem pro výsledek řešení. Pokud ovšem při křížení rozpojíme tento blok, fitness hodnota výsledného jedince klesá. Tuto slabou stránku algoritmu je možné řešit několika cestami:

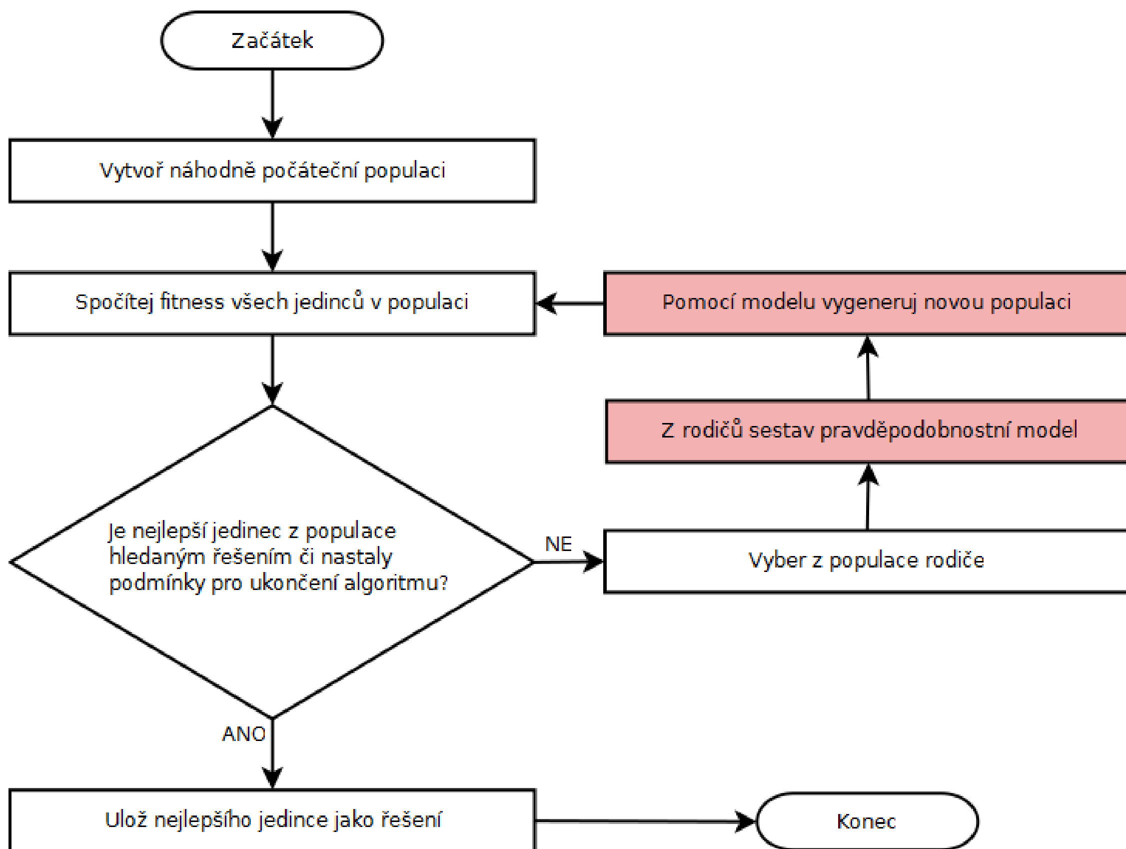
- adaptibilními rekombinačními postupy
- explicitním zjišťováním závislostí
- odhady distribuční funkce

Podrobnější popis prvních dvou řešení lze nalézt v [22]. Algoritmy, které využívají odhady distribuční funkce, se budeme zabývat v následujícím textu. Tyto algoritmy se nazývají EDA (**E**stimation of **D**istribution **A**lgorithm) [21].

EDA

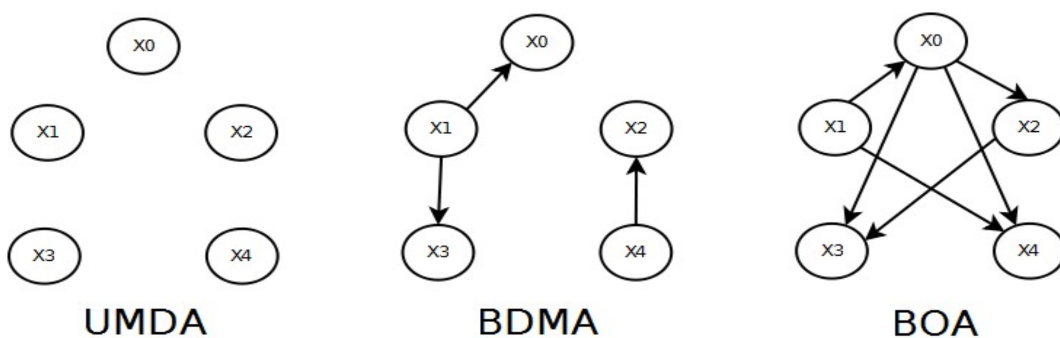
Algoritmy v této skupině se vyznačují globálním pohledem na celou populaci při tvorbě pravděpodobnostního modelu. Model je tvořen na základě statistické analýzy populace. Noví jedinci již nevznikají křížením, ale generováním na základě vypočítaného pravděpodobnostního modelu.

Na rozdíl od standardních genetických algoritmů jsou EDA algoritmy schopny poznat závislosti mezi geny chromozomu a tedy detekovat stavební bloky, což vede k lepší konvergenci algoritmu. Pseudokód algoritmu EDA je uveden na obrázku 2.5. Bloky, které se zásadně liší od klasického genetického algoritmu jsou vyznačeny barevně.



Obr. 2.5: Vývojový diagram EDA algoritmu

Nejdůležitější částí EDA algoritmu je tvorba pravděpodobnostního modelu, ze kterého se bude vzorkovat další generace. Dle předpokládané interakce mezi proměnnými modelu rozlišujeme tři základní druhy pravděpodobnostních modelů. Předpokládaná interakce mezi proměnnými je zobrazena na obrázku 2.6.



Obr. 2.6: Interakce mezi jednotlivými proměnnými pravděpodobnostního modelu

První model (UMDA) se vyznačuje nulovou interakcí mezi geny, druhý model (BDMA) umožňuje podvojné závislosti a třetí model (BOA) pracuje s Bayesovskou sítí a BDe metrikou [23]. Každý z těchto modelů se využívá pro řešení jiné domény problémů.

Matematické vyjádření pravděpodobnosti výskytů jednotlivých jedinců pro obrázek 2.6 je uvedeno ve vzorci 2.3.

$$\begin{aligned} &UMDA: && p(x)=p(X_3)\,p(X_0)\,p(X_2)\,p(X_4)\,p(X_1) \\ &BMDA: && p(X)=p(X_1)\,p(X_3|X_1)\,p(X_0|X_1)\,p(X_4)\,p(X_2|X_4) \\ &BOA: && p(X)=p(X_1)\,p(X_0|X_1)\,p(X_4|X_{0,}X_1)\,p(X_2|X_0)\,p(X_3|X_{2,}X_0) \end{aligned} \tag{2.3}$$

2.2.1 UMDA

UMDA (z anglického názvu **U**nivariate **M**arginal **D**istribution **A**lgorithm) využívá nejjednodušší možný přístup – jednotlivé proměnné jsou nezávislé.

Diskrétní varianta

Nejčastěji se UMDA používá pro řešení diskrétních problémů. Pravděpodobnostní model, dle kterého bude generován nový jedinec, je popsán ve vzorci 2.4.

$$p(X)=\prod_{i=0}^{n-1} p(X_i) \tag{2.4}$$

Pravděpodobnost pro každý gen chromozomu vypočítáme jako četnost výskytů řetězců, které mají hodnotu x_i na i-té pozici v populaci vybraných rodičů, která má mohutnost N. Četnost výskytů hodnoty x_i nazýváme n_i.

$$p(X_i)=\frac{n_i(X_i)}{N} \tag{2.5}$$

UMDA algoritmus je pro svou jednoduchost a rychlost velmi používaný.

Spojitá varianta

Při řešení spojitého problému můžeme použít dva postupy:

1. Diskretizujeme spojitý model, načež můžeme využít diskrétní variantu UMDA algoritmu, jehož výsledek převedeme zpět na spojitý model.
2. Sestavíme model, který bude pro každý gen obsahovat aritmetický průměr x všech i-tých genů rodičů a směrodatnou odchylku σ. Populaci potomků pak generujeme tak, že pro každý gen vybíráme náhodné hodnoty ze vzniklého rozsahu $(x - \sigma, x + \sigma)$.

Diskretizace spojitého modelu

Popis diskretizace bude popsán pro jednorozměrný spojitý problém, ve kterém budeme hledat reálnou hodnotu jedné proměnné. Pro vícerozměrný model je postup shodný, jen musí být aplikován na všechny prohledávané prostory.

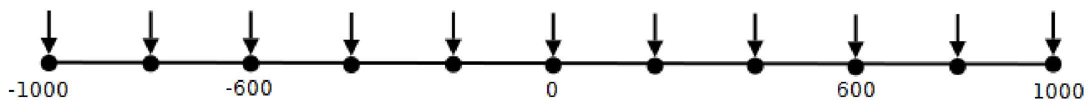
Při zadání problému je nutné zvolit rozsah, ve kterém budeme hledat výsledek funkce. Dále musíme zvolit počet oblastí, které ze spojitého rozsahu vytvoříme.

Jakmile máme navržený formát modelu, můžeme pro populaci rodičů vypočítat pravděpodobnosti, že hodnota rodiče patří do jednotlivých oblastí v rozsahu.

Při generování nového jedince poté vezmeme náhodnou hodnotu, pro kterou určíme, do kterého rozsahu patří. Tuto hodnotu poté zařadíme do genu s pravděpodobností, kterou jsme načetli z rozsahu.

Příklad práce UMDA algoritmu, který pracuje s diskretizací:

1. Zadaný rozsah hodnot je (-1000, 1000). Tento rozsah rozdělíme na 10 oblastí. Formát modelu je zobrazen na obrázku 2.7. Mezery mezi šipkami jsou jednotlivé oblasti modelu.



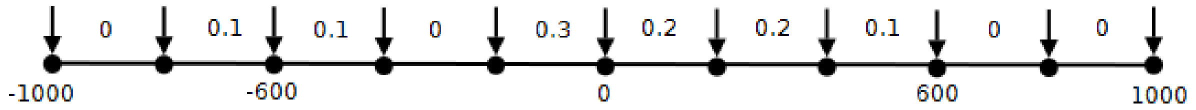
Obr. 2.7: Rozdělení modelu na oblasti

2. Pro populaci rodičů se vypočítají jednotlivé pravděpodobnosti, že číslo spadá do oblasti modelu. Populace rodičů je v tabulce 2.1, výpočet pravděpodobnosti pro rozsah (-200, -0) je ve vzorci 2.6, vypočítaný model je zobrazen na obrázku 2.8.

Tabulka 2.1: Náhodně vygenerovaná populace rodičů

Pořadové číslo rodiče	Hodnota rodiče	Rozsah, do kterého hodnota patří
1	296	(200,400)
2	-111	(-200,0)
3	555	(400,600)
4	110	(0,200)
5	-79	(-200,0)
6	-175	(-200,0)
7	77	(0,200)
8	-358	(-600,-400)
9	245	(200,400)
10	-610	(-600,-800)

$$P_{(-200,0)} = \frac{\textit{Počet výskytů v oblasti}}{\textit{Celkem prvků}} = \frac{3}{10} = 0{,}3 \quad (2.6)$$



Obr. 2.8: Vypočítaný model pro populaci rodičů z tabulky 2.1

Sestavení modelu spojitého řešení

Model pro spojité řešení se skládá ze dvou hodnot:

1. Aritmetický průměr x hodnot rodičů, který je vypočítán dle vzorce 2.7.

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} X_i \quad (2.7)$$

2. Směrodatná odchylka σ, která je vypočítána dle vzorce 2.8.

$$\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^2} \quad (2.8)$$

Hodnota nového jedince je vygenerována z rozsahu $(x - \sigma, x + \sigma)$.

2.2.2 BDMA

BDMA (z anglického názvu **B**ivariate **M**arginal **D**istribution **A**lgorithm) umožňuje kromě nezávislých proměnných počítat i s proměnnými, které jsou podvojně závislé. Tyto závislosti zjišťuje statistickými testy. Pro generování proměnných na závislých pozicích je využíván aparát podmíněné pravděpodobnosti.

Závislost mezi proměnnými

Pro zjišťování závislosti mezi proměnnými se používá Pearsonův chí – kvadrát test (z anglického názvu Pearson's chi-square test) [5].

Tento test se spouští pro všechny možné dvojice parametrů. Výchozí hypotézy vztahu proměnných jsou:

- H_0 – proměnné jsou nezávislé
- H_a – proměnné jsou závislé

Výsledkem testu pro každý pár je číslo, které je nulové, pokud jsou parametry nezávislé. Pro závislé parametry velikost čísla výrazně narůstá. Vlastní výpočet probíhá podle vzorce 2.9.

$$r = \sqrt{\frac{\sum_{i=1}^{n}(x_{1i}-\bar{x}_1)(x_{2i}-\bar{x}_2)}{\sum_{i=1}^{n}(x_{1i}-\bar{x}_1)^2\sum_{i=1}^{n}(x_{2i}-\bar{x}_2)^2}} \qquad (2.9)$$

Popis výrazů použitých ve vzorci:

- x_{1i} – hodnota vybrané proměnné *i*-tého člena populace
- x_1 – aritmetický průměr vybrané hodnoty v populaci
- n – počet jedinců v populaci
- r – výstupní hodnota Pearsonova testu v rozsahu $(0,\infty)$

Při interpretaci výsledku výpočtu je nutné vzít v potaz, že i vysoká hodnota čísla nemusí znamenat závislost mezi proměnnými, neboť může být ovlivněn vysokými hodnotami u jedné z proměnných. Proto se používá testanční statistiky ve vzorci 2.10.

$$t = \frac{r\sqrt{n-2{,}5r}}{\sqrt{1-0{,}5r^2}} \qquad (2.10)$$

Testuje se hypotéza H_0 proti různým alternativám H_a. Vyjde-li $|t|$ větší než odpovídající kvantil Studentova rozdělení, zamítá se H_0 a zkoumané proměnné jsou závislé.

Detailní popis výpočtu závislosti je uveden v [5].

Generování hodnot na závislých pozicích

Pomocí testu jsme zjistili, které proměnné jsou závislé. U nezávislých proměnných postupujeme stejné jako u UMDA algoritmu. Pro závislé proměnné si nejprve vypočítáme pravděpodobnost, že se ve dvou jedincích vyskytují obě proměnné současně (vzorec 2.11). Poté vypočítáme podmíněnou pravděpodobnost, která udává pravděpodobnost výskytu první proměnné, pokud se v potomkovi vyskytuje druhá proměnná (vzorec 2.12).

$$p_{i,j} = \frac{n_{i,j}(i,j)}{N} \qquad (2.11)$$

$$p_{i,j}(x_i|x_j) = \frac{p_{i,j}(x_i,x_j)}{p_j(x_j)} \qquad (2.12)$$

Pravděpodobnost generování jedince $X = a_1a_2...a_{n-1}$ je uvedena ve vzorci 2.13, kde R je množina kořenových uzlů a V je množina vrcholů ve stromě závislostí.

$$p(X) = \prod_{j\in R} p(a_j) \prod_{(i,j)\in V\setminus R} p(a_j|a_i) \qquad (2.13)$$

2.2.3 BOA

Vícenásobné závislosti mezi proměnnými je možné pokrýt pomocí BOA (z anglického názvu **B**ayesian **O**ptimization **A**lgorithm), který strukturu problému zakóduje do acyklické Bayesovské sítě. Pro každou proměnou X_i definuje množinu Π_x, ve které jsou uloženy proměnné, na kterých je X_i závislá.

Závislosti mezi problémy hledáme pomocí stejného algoritmu, jako u algoritmu BDMA, jen se zvětšují stupně volnosti, což samozřejmě navyšuje výpočetní složitost algoritmu.

Generování hodnot jedince

Při generování hodnot jedince se využívá podmíněné pravděpodobnosti $p(X_i|\Pi_i)$, která udává závislost proměnné X_i na množině proměnných Π_i. Pro generování nových jedinců populace je poté použito rozdělení pravděpodobnosti:

$$p(X) = \prod_{i=0}^{n-1} p(X_i|\Pi_i) \quad (2.14)$$

2.3 Další druhy evolučních algoritmů

Genetické algoritmy nejsou jedinou aplikací evolučních technik. V této části si uvedeme některé další typy algoritmů, které využívají principy evoluce.

2.3.1 Evoluční strategie

Evoluční strategie jsou vhodné především pro optimalizaci funkcí, pro které nedokážeme určit analytické řešení a klasické gradientní metody konvergují k lokálním extrémům funkce. Nejčastěji jsou využity pro výpočet diskrétních či spojitých parametrů u multidimenzionálních funkcí.

Evoluční strategie vznikla nezávisle na vývoji genetických algoritmů, její první využití byla optimalizace tvaru těles pro snížení turbulence ve větrném tunelu.

V principu jsou evoluční strategie velmi podobné genetickým algoritmům. Hlavním rozdílem je snaha pracovat s reálnými parametry a autoevolucí řídících parametrů, čemuž se genetické algoritmy, pokud je to jen trochu možné, vyhýbají a využívají statické řídící parametry. V evolučních strategiích se hodnoty řídících parametrů po každém cyklu algoritmu upravují, aby výsledkem v další populaci bylo 20 % kvalitnějších jedinců, než jsou v populaci rodičů.

Při křížení využívají evoluční strategie možnosti reálných čísel, proto vedle bodového křížení také umožňují křížení průměrem. Při tomto křížení je gen jedince vypočítán jako aritmetický průměr příslušných genů rodičů.

Každý nově vygenerovaný gen podléhá mutaci, která se určuje velikostí rozptylu od Gaussovy distribuční funkce s nulovou střední hodnotou a zadaným rozptylem. Tento rozptyl je obvykle 0,05.

Při popisu evolučních strategií čerpáno z [1], [2] a [20], podrobnější zpracování v [4].

2.3.2 Genetické programování

Cílem genetického programování není hledat optimální hodnoty parametrů pro řešení problému, ale generovat celé programy v určitém programovacím jazyce, které danou problematiku řeší.

Při genetickém programování pracujeme se spustitelnými strukturami, které můžou mít variabilní délku. Nejčastěji se jedná o programy, které se dají reprezentovat stromy. Uzly těchto stromů obvykle prezentují operace a listy proměnné a konstanty.

Fitness hodnota jedince je vypočítávána na testovací množině dat, pro kterou sledujeme výstupy navrženého programu.

Při křížení se u obou rodičů náhodně určí podstromy, které se vzájemně prohodí. Touto cestou vzniknou vždy dva potomci ze dvou rodičů. Při mutaci se náhodně vybraný podstrom nahradí nově vygenerovaným podstromem.

Podrobnější popis genetického programování v [3].

2.4 No free lunch teorém

Tento teorém pojednává o faktu, že každý stochastický optimalizační algoritmus dosahuje lepších výsledků jak optimalizační algoritmus deterministický [11]. Pokud je deterministický algoritmus lepší, pak je to díky doménově závislým informacím. Pokud tyto informace vložíme do stochastického algoritmu, pak bude opět lepší než deterministický [12].

3 Paralelizace programu

Hlavním důvodem pro paralelizaci programu je snaha o zrychlení jeho vykonávání. K paralelizaci lze přistoupit mnoha způsoby, které se mezi sebou liší úrovní zanoření – granularitou.

3.1 Granularita paralelismu

Granularitou paralelismu rozumíme způsob aplikace paralelismu v programu. Jemnou granularitou rozumíme paralelismus uvnitř instrukcí, které mají velkou režii s vlastním spuštěním paralelního zpracování.

Jako hrubou granularitu bereme paralelismus mezi procesy, který má malou režii spuštění paralelního zpracování, ovšem paralelismus málo využívá. Při rozhodování o granularitě paralelismu tedy musíme najít rovnovážný stav, aby byl paralelismus nejlépe využit při co nejmenší režii spuštění. Při členění granularity budeme postupovat od nejjemnějšího k hrubým metodám paralelismu programu.

Paralelismus uvnitř instrukcí

Jedná se o případ, kdy se může provádění jedné instrukce rozdělit na více paralelních bloků, které lze vykonat současně. Na tomto principu pracuje právě architektura SSE [27], která je například schopna jednou instrukcí zpracovat čtyři čísla v plovoucí řádové čárce s jednoduchou přesností. Možnosti této architektury budou podrobněji probrány ve čtvrté kapitole.

Příklad

ADDPS xmm1, xmm2

Tato instrukce paralelně sečte čtyři čísla (každé velikosti 32b) uložená v zadaných registrech, které mají velikost 128b a výsledek uloží do registru xmm1 .

Paralelismus mezi instrukcemi

Zde je paralelismus prováděn tak, že se vezmou dvě instrukce a zpracují se paralelně. Při takovémto provádění instrukcí nesmí dojít k těmto situacím:

- *datová závislost* – data vstupující do druhé instrukce jsou ovlivněna provedením první instrukce

- *procedurální závislost* – pokud se pracuje s pamětí a není doposud rozhodnuto, kdy bude zahájena další práce s pamětí. Příkladem je instrukce za instrukcí skoku, kterou můžeme provést, až máme vyhodnocenu podmínku skoku.
- *konflikt zdrojů* – obě instrukce potřebují k provedení přístup ke stejnému zdroji, například k ALU

Příklad

ADD r1, r2

ADD r3, r4

Při pokusu o současné provedení těchto dvou instrukcí je jasné, že budou od jedné ALU jednotky zařazeny sekvenčně. Pokud se v počítači nacházejí dvě aritmetické jednotky, pak lze tyto instrukce provést paralelně.

Paralelismus mezi příkazy

Tento paralelismus využívá možnost provést několik příkazů v programu paralelně.

Příklad

Na tomto principu pracují například vektorové počítače, které dokáží efektivně paralelizovat množinu jednoduchých příkazů, například u násobení matic [41].

Paralelismus mezi bloky procesu

V tomto případě se určí bloky programu, které se provádějí paralelně. Jako bloky můžeme chápat například procedury v programu.

Příklad

Pokud při programování využijeme knihovnu OpenMP [10], můžeme bloky programu paralelizovat pomocí direktivy pro překladač.

Paralelismus mezi procesy

V tomto případě chápeme procesy jako celé programy, které se vykonávají autonomně a tudíž mohou být provedeny paralelně.

> ***Příklad***
> *Tento paralelizmus může být také vytvořen pomocí knihovny OpenMP [10].*

3.2 Výpočet zrychlení při paralelizaci

V rámci této diplomové práce budeme využívat faktu, že na moderních procesorech jsou pro každé jádro dostupné jednotky SSE, které umožňují paralelizaci výpočtů. Navíc máme k dispozici procesory s více jádry, tudíž budeme pracovat s granularitou uvnitř instrukcí a mezi bloky procesu.

Zkusme si představit ideální program pro naši paralelizaci, který obsahuje jen výpočty s reálnými čísly s jednoduchou přesností a provádí takové operace, které jsou dostupné v instrukční sadě SSE. Doba provádění tohoto programu by poté byla:

$$T_{sse} = \frac{T}{4} \tag{3.1}$$

Dále bychom pracovali na procesoru, který umožňuje zpracování na n jádrech. Doba provedení programu by poté byla:

$$T_{par+sse} = \frac{T_{sse}}{n} = \frac{T}{4 \cdot n} \tag{3.2}$$

Touto paralelizací bychom dosáhli zrychlení:

$$S_n = \frac{T}{T_{par+sse}} = \frac{T \cdot 4 \cdot n}{T} = 4 \cdot n \tag{3.3}$$

Je zřejmé, že takto ideální program zřejmě neexistuje, neboť do doby vykonání programu musíme započítat:

- režie pro spuštění programu na více jádrech
- přítomnost částí programu, které nelze paralelizovat

Amdahlův zákon

Pan Gene Myron Amdahl formuloval v roce 1967 vztahy, kterými popisoval postup výpočtu zrychlení při paralelizaci částí algoritmu [37]. Tyto vztahy vycházejí z předpokladu, že každý algoritmus je složen ze dvou částí – z části vhodné k paralelnímu zpracování a z části, která se musí vykonat sekvenčně.

Než budeme pokračovat v popisu Amdahlova zákona, tak si popišme zkratky, které se budou ve vzorcích vyskytovat:

- T – čas, za který se provede celý algoritmus
- T_{par} – čas, za který je zpracován celý algoritmus s maximálním využitím paralelizace
- T_p – čas, za který je zpracována paralelní část algoritmu
- T_s – čas, za který je zpracována sériová část
- n – počet procesorů
- $S(n)$ – zrychlení, kterého dosáhneme paralelizací na n procesorech
- β – podíl paralelizované části vůči sériové (T_s / T_s)

Z předpokladu přítomnosti paralelní a sériové části algoritmu můžeme sestavit rovnici

$$T = T_s + T_p \quad (3.4)$$

Paralelní část můžeme rozdělit na více procesorů, pak tedy doba trvání algoritmu bude

$$T_{par} = T_s + \frac{T_p}{n} \quad (3.5)$$

Pokud známe dobu vykonání algoritmu sekvenčně a dobu vykonání algoritmu s využitím paralelizace, pak můžeme spočítat zrychlení získané paralelizací [38]:

$$S(n) = \frac{T}{T_{par}} = \frac{1}{(1-\beta) + \frac{\beta}{n}} \quad (3.6)$$

Zřejmě nejdůležitější věcí, kterou lze vyvodit ze vzorce pro zrychlení je, že nikdy nemůžeme získat větší zrychlení než ($1 / (1 - \beta)$). Například pokud dosáhneme podíl obou částí $\beta = 0{,}9$ (90 % algoritmu je paralelizováno), tak nikdy nemůžeme algoritmus zrychlit více než 10 krát.

Gustafsonův zákon

Gustafsonův zákon říká, že každý dostatečně velký problém může být efektivně paralelizován. Tento zákon úzce souvisí s Amdahlovým zákonem, který stanovuje hranice, do jaké míry může být program paralelizován.

Slabinou Amdahlova zákona je předpoklad, že se sekvenční část nemění s rostoucím počtem procesorů a paralelní část je na procesorech rovnoměrně rozložena, takže s rostoucím počtem procesorů nelze dosáhnout většího zrychlení, než je poměr paralelizovatelné a neparalelizovatelné části.

Zrychlení na sekvenčním stroji paralelizací je vždy rovno 1 a se dá vyjádřit jako:

$$\alpha + \beta = 1 \quad (3.7)$$

Proměnná α je poměr sekvenční části a β je poměr paralelní části vůči celému programu. Pokud sestavíme stejný vzorec pro paralelní počítač, pak pro P procesorů získáme:

$$\alpha + P\beta = 1 \tag{3.8}$$

Ze vzorce 3.7 vyjádříme β, kterou poté dosadíme do vzorce 3.8. Touto cestou vznikne vzorec 3.9, který je vyjádřením Gustafsonova zákona [39].

$$S(P) = P - \alpha(P-1) \tag{3.9}$$

Z tohoto vzorce plyne, že pokud klesá velikost sekvenční části α s rostoucí velikostí programu, pak při zvyšujícím se počtu procesorů P se také zvětšuje dosažené zrychlení, teoreticky až do nekonečna.

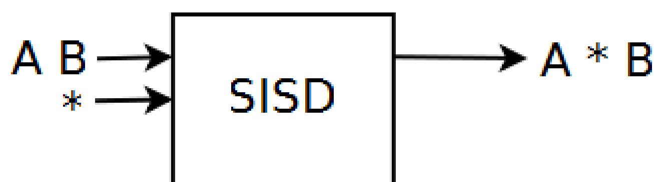
3.3 Klasifikace paralelních systémů

Nejznámější klasifikací paralelních systémů je Flynnova klasifikace z roku 1966 [40]. Tato klasifikace dělí paralelní systémy z hlediska počtu toků instrukcí a počtu toků dat. Na výpočetní proces se tedy můžeme dívat jako na množinu instrukcí (tok instrukcí) zpracovávající množinu dat (tok dat).

Tato klasifikace třídí paralelní systémy dle velikosti obou množin na jednotkové a vícenásobné. Tím pádem vznikly čtyři kategorie. Toto rozdělení je vcelku hrubé a je možné popsat řadu mezitříd, nicméně pro základní pochopení principu paralelismu systému je dostačující.

Jednotlivé třídy Flynnovy klasifikace [42] jsou:

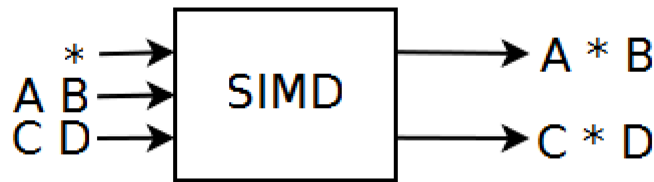
- **SISD** – počítač v této třídě zpracovává jeden tok dat jedním tokem instrukcí. Klasickým představitelem této třídy je počítač s architekturou von Neumann, který má jeden procesor a data uložena v jedné paměti.



Obr. 3.1: Ukázka práce SISD architektury

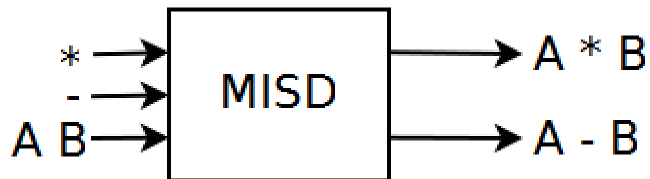
- **SIMD** – v tomto případě je zpracováváno více toků dat jedním tokem instrukcí. Příkladem zástupce této kategorie je vektorový procesor, který daný paralelismus provádí mezi příkazy. Tento procesor je navržen, aby dokázal zpracovat instrukci nad množinou dat. Více informací

o architektuře vektorových procesorů v [41]. Příkladem paralelismu SIMD mezi instrukcemi jsou jednotky MMX a SSE, které budou podrobněji popsány v dalších kapitolách práce.



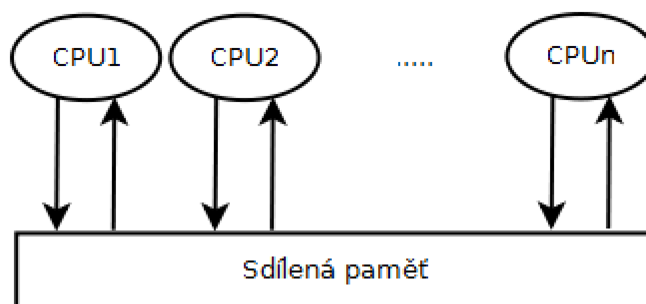
Obr. 3.2: Ukázka práce SIMD architektury

- **MISD** – tato kategorie procesorů není příliš běžná, obvykle je složena ze série procesorů, které zpracovávají společná data dle vlastních programů. Zástupcem této kategorie jsou zřetězené procesory, které jsou lineárně propojené a data jimi postupně prochází. Jsou vhodné pro řešení úloh s proudovým charakterem.



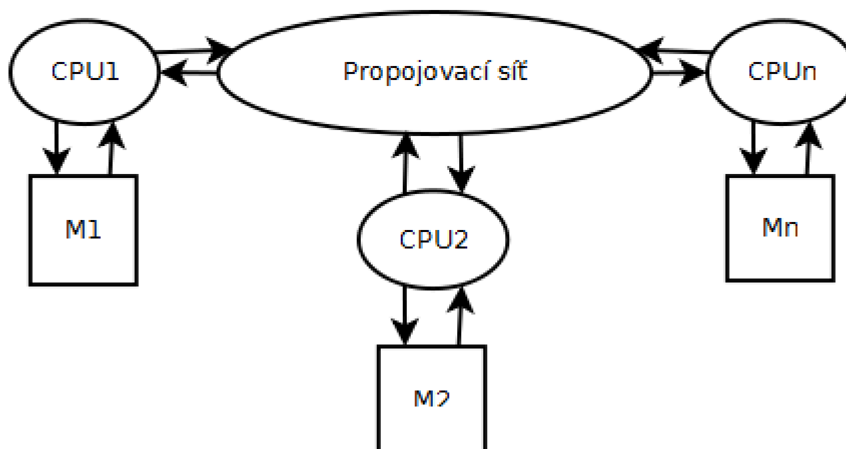
Obr. 3.3: Ukázka práce MISD architektury

- **MIMD** – systém pracující s touto architekturou má množství samostatných procesorů, které samostatně zpracovávají vlastní data. Tyto počítače se nejčastěji dělí dle způsobu propojení jednotek [42] na počítače:
    1. *Se společnou sběrnicí* – skládají se z několika procesorů a sdílené paměti, které jsou propojeny sběrnicí. Každý procesor může mít svou lokální paměť, komunikace mezi procesory probíhá přes sdílenou paměť. Zástupcem tohoto typu je systém 66 IBM Blade serverů po dvou procesorech, který využívá skupina Speech@FIT [13] na Fakultě informačních technologií VUT v Brně.



Obr. 3.4: Ukázka práce MIMD architektury se společnou sběrnicí

2. *S propojovací sítí* – tyto počítače jsou složeny z relativně samostatných jednotek, kde každá jednotka má svůj procesor a vlastní paměť. Jednotlivé procesory spolu komunikují předáváním zpráv po propojovací síti. Tím pádem je komunikace pomalejší než u předchozího typu, ale je možné využít mnohem vyšší počet jednotek. Zástupcem toho typu je server IBM SP, který využívá Centrum pro intenzivní výpočty ČVUT [14].



Obr. 3.5: Ukázka práce MIMD architektury s propojovací sítí

3. *S pevnou sítí* – tento systém propojuje množství počítačů sítí tak, aby na nich mohla běžet paralelni úloha. Tento systém je obvykle složen z klasických (uživatelských) počítačů, které jsou propojeny sítí (LAN, WAN), na kterých běží program pro komunikaci mezi procesy programu. Jedním z nejslavnějších zástupců tohoto systému paralelniho zpracování je projekt SETI@home [15], který analyzuje signály z vesmíru a hledá ty, které by mohly být mimozemského původu. Na počátku roku 2010 bylo do tohoto projektu zapojeno více než 2,5 miliónu počítačů z 234 států.

3.4 OpenMP

K paralelizaci programu bude využit systém OpenMP [10]. Jedná se o sadu direktiv pro překladač jazyku C/C++, kterými se ovlivňují způsoby vykonání jednotlivých bloků programu a procedur, které podporují paralelní programování.

Princip činnosti programu s podporou OpenMP standardu je jednoduchý – při překladu se identifikují bloky, které lze vykonávat paralelně. Tyto bloky určuje programátor za pomocí direktiv pro překladač.

Program může pro jazyk C a překladač gcc vypadat například takto:

```
#pragma omp parallel
{
        //blok programu, který vykoná každé jádro
}
```

Podrobnější popis tohoto standardu hledejte v [10], dále v [7] a [8].

V rámci této práce se bude nejvíce využívat direktiva **#pragma omp parallel for**. Tato pragma zajišťuje vykonání jednotlivých iterací cyklu *for* na více jádrech.

Syntax zápisu této pragmy je:

#pragma omp parallel for [*klauzule* [[*,*] *klauzule*]...] *nový řádek*
for-smyčka

Dostupné klauzule jsou:

- *if*(*scalar-expression*) - pokud je podmínka v této klauzuli nepravdivá, pak se paralelizace smyčky for neprovede.
- *num_threads*(*integer-expression*) - vložené číslo určuje, na kolika jádrech se má paralelizace provést.
- *default*(*shared* | *none*) – určuje základní přístup k proměnným ve smyčce. Pokud je použita klauzule *default*(*shared*), pak jsou všechny proměnné ve smyčce sdílené. Pokud je použito *default*(*none*), pak musí být všechny proměnné pro paralelizovanou část určeny výpisem s využitím klauzulí pro jejich identifikaci.
- *private*(*list*) - proměnné v tomto seznamu jsou privátní pro každé jádro, jejich změna se projeví pouze v rámci aktivního jádra.
- *firstprivate*(*list*) - proměnné v tomto seznamu jsou privátní pro každé jádro, ale při vytvoření proměnné z tohoto listu se naplní původní hodnotou, kterou obsahovala před započetím paralelního vykonávání.
- *shared*(*list*) - proměnné určené v tomto listu jsou sdílené pro všechna jádra, která vykonávají paralelní blok programu.
- *copyin*(*list*) - proměnné v tomto listu jsou při vytvoření v jednotlivých vláknech naplněny aktuální hodnotou proměnné z řídícího jádra.
- *reduction*(*operator: list*) - hodnoty proměnných v tomto listu jsou po skončení paralelního bloku uloženy do původní proměnné.

4 SSE jednotky

Operace v plovoucí desetinné čárce byly v procesorech silně problematickým místem vzhledem k rychlosti zpracování. Jako první přišla s řešením v roce 1998 firma AMD s instrukční sadou 3DNow [28]. Tato sada využívala osm 64 bitových registrů. Do každého registru umožňovala vložit dvě čísla v plovoucí řádové čárce s jednoduchou přesností. Dále poskytovala 31 nových instrukcí, které podporovaly SIMD operace.

Odpovědí konkurenční firmy Intel byla v roce 1999 instrukční sada SSE (dříve se tato sada jmenovala MMX-2, KNI) [27].

4.1 Verze SSE jednotek

První implementaci SSE jednotek se objevila v roce 1999. Od té doby bylo vydáno několik verzí, které byly střídavě podporovány procesory Intel nebo AMD. V následujícím textu budou uvedeny jednotlivé známé verze a jejich stručný popis. Informace o procesorech, datech vydaní a prvotní implementace dané verze SSE jednotky získány z [25, 26], pro procesory Intel dále z [24].

4.1.1 SSE1

Jednalo se o první sadu SSE, kterou implementovala firma Intel v roce 1999 pro procesor Pentium 3. Tato sada obsahovala 8 XMM registrů a rozšiřovala instrukční sadu o 70 instrukcí. Tyto instrukce umožňovaly:

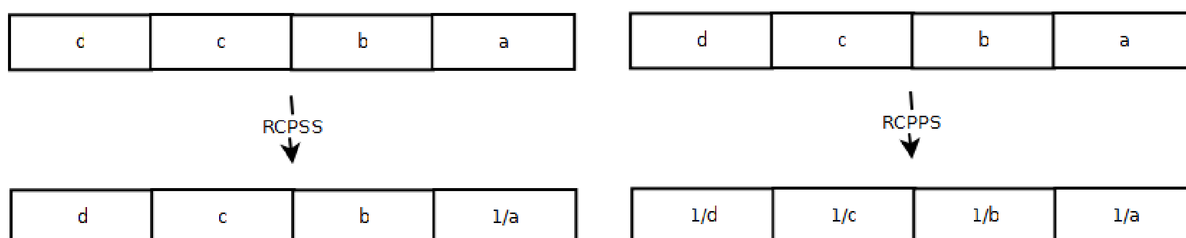
- práci s čísly v plovoucí řádové čárce s jednoduchou přesností
- práci s 64-bitovými čísly z domény MMX
- práci s registrem MXSCR
- řízení ukládání dat do cache paměti a řízení pořadí ukládání dat

Konkurenční výrobce procesorů (firma AMD) přidal podporu pro SSE instrukce v roce 2001 do procesorů Athlon XP (jádro Palonimo) a Duron (jádro Morgan).

Aritmetické instrukce

Většina instrukcí je navržena pro datový typ float, zbytek pracuje s celočíselnými typy z domény MMX. Jsou podpořeny základní matematické operace a přidány například aritmetický průměr, maximum z hodnot či obrácená hodnota. Podle posledních dvou písmen názvu lze u instrukcí zjistit toto:

1. Poslední písmeno udává typ, se kterým se pracuje. V sadě SSE1 se můžeme setkat s písmeny:
    - *s* – reálné číslo s jednoduchou přesností
    - *b* – celé číslo o velikosti 8-bitů
    - *w* – celé číslo o velikosti 16-bitů
2. Předposlední písmeno určuje, zda bude pracovat s celým registrem – písmeno *p,* či jeho částí – písmeno *s*. Příklad tohoto rozdělení pro reálné číslo je na obrázku 4.1.



Obr. 4.1: Rozlišení výsledku instrukce dle koncovky

Logické instrukce

Pro čísla v plovoucí řádové čárce jsou podpořeny operace:

- konjunkce
- negace konjunkce
- disjunkce
- nonekvivalence

Instrukce porovnání

Porovnává dva zadané registry. Pokud je vyhodnocení pravdivé, pak do výsledku ukládá hodnotu 0xffffffff, naopak při nepravdivosti výroku ukládá hodnotu 0x0.

Možnosti porovnání jsou:

- =, <, <=, <>, >, >=
- seřazenost, neseřazenost

Ostatní instrukce

Další instrukce podporují:

- konverzi mezi datovým typem SSE jednotky (čtyři 32 bitová čísla v plovoucí řádové čárce s jednoduchou přesností) a typem MMX (dvě 32 bitová celočíselná čísla).
- podporu práce s registrem MXSCR a uložení/načtení stavu x87 FPU.
- práci mezi bloky paměti a XMM registry.

- přesouvání hodnot v rámci XMM registru a extrakci určené části registru na požadované místo.
- kontrolu cache paměti

4.1.2 SSE2

Tato instrukční sada byla implementována firmou Intel v roce 2001 pro procesor Pentium 4. Stávající SSE instrukční sadu rozšiřuje o 144 nových instrukcí a přidává nové datové typy pro XMM registry. Firma AMD přidala podporu pro instrukční sadu SSE2 poprvé v roce 2003 do procesorů Opteron a Athlon 64.

SSE2 podporuje datové typy vymezené sadou SSE1 a přidává některé nové. Velkou výhodou nové sady je podpora reálných čísel s dvojitou přesností a řady celočíselných typů.

V této verzi lze do XMM registru uložit nově tato čísla:

- dvě reálná čísla s dvojitou přesností
- dvě 64-bitová celá čísla
- čtyři 32-bitová celá čísla
- osm 16-bitových čísel
- šestnáct 8-bitových čísel

Samozřejmě součástí této sady byly instrukce, které umožnily práci s těmito novými datovými typy. Hlavně byly přidány instrukce pro práci s reálnými čísly s dvojitou přesností. Dále byly přidány instrukce:

- pro práci s celočíselnými typy
- nové možnosti využívání cache paměti a řízení kódu

Pro tuto sadu již nebudu podrobně popisovat jednotlivé třídy instrukcí, jak tomu bylo u SSE1. V principu jsou u SSE2 podpořeny stejné operace, jako u SSE1, jen pro nové datové typy. Rozlišení instrukcí dle datových typů již není striktně dle posledních dvou písmen instrukce.

4.1.3 SSE3

Sada SSE3 byla přidána firmou Intel v roce 2004 do procesorů Pentium 4 (Prescott revize). Firma AMD přidala podporu pro SSE3 v roce 2005 do procesoru Athlon 64.
Tato sada obsahovala 13 nových instrukcí, které se dají rozdělit následovně:

- 10 instrukcí pro vylepšení funkčnosti SSE a SSE2 sady
- instrukce akcelerující programování x86 FPU jednotky, která zavádí konverzi reálných hodnot na celá čísla
- dvě instrukce pro vylepšení synchronizace vláken

SSSE3

Tato revize sady SSE3 byla vydána firmou Intel v roce 2006 a obsahovala rozšíření sady SSE3 o 16 nových instrukcí. Firma AMD toto rozšíření instrukcí neimplementovala.

Všechny přidané instrukce byly pro práci s celočíselnými typy.

4.1.4 SSE4

Sada SSE4 byla implementována roku 2008 firmou Intel a obsahovala 54 nových instrukcí. Byla rozdělena na dvě části:

1. SSE4.1 – tato část obsahovala 47 instrukcí a byla implementována na procesorech s kódovým označením Penryn.
2. SSE4.2 – obsahuje zbylých 7 instrukcí sady SSE4 a byla implementována na procesorech s kódovým označením Nehalem.

Sada SSE4 neobsahovala nové datové typy a nově přidané instrukce již nepracovaly s 64-bitovými registry sady MMX.

Zřejmě nejdůležitější přidanou instrukcí byla instrukce PMULLD, která umožňovala násobení 32-bitových celých čísel.

Konkurenční firma AMD z instrukcí sady SSE4 implementovala pouze dvě instrukce (POPCNT, EXTRQ) a přidala 4 nové. Tuto sadu nazvala SSE4a a implementovala ji roku 2007 do procesorů s kódovým označením Barcelona. K implementaci kompletní sady SSE4 zřejmě firma AMD nepřistoupí.

4.1.5 SSE5

Vývoj této sady ohlásila firma AMD v roce 2007 [30]. V roce 2008 byla představena společná vize firem Intel a AMD o tomto rozšíření sad SSE:

- nastavení délky registrů XMM registry YMM, čímž by se dosáhlo 256-bitové délky.
- Zavedení nedestruktivních instrukcí, které ukládají výsledek dvou registrů do třetího. V současném stavu došlo vždy k přepsání hodnot jednoho z registrů
- volný přístup k zarovnání proměnných v paměti

Tato vize byla pojmenována AVX [31]. Proto firma AMD roku 2009 [29] přehodnotila dříve vydané prohlášení o SSE5, které má obsahovat tři sety instrukcí:

1. XOP – hlavní sada instrukcí, která pracuje s XMM a YMM registry. Množina operací by měla být podobná jako u SSE jednotek. Důležitá je podpora nedestruktivních instrukcí s více než dvěma operandy.

2. FMA4 – podpora operace uvedené ve vzorci 4.1.

$$d = a + (b * c) \tag{4.1}$$

3. CVT16 – podpora konverze reálného čísla mezi jednoduchou a dvojitou přesností.

První kompletní implementaci sady SSE5 tedy mají obsahovat procesory vyrobené 32nm technologií s kódovým označením Buldozer, které byly ohlášeny na čtvrtý kvartál roku 2010, ale spíše se předpokládá vydání v roce 2011.

Firma Intel se chystá vizi implementovat v sadě AVX, která by měla být v roce 2011 v procesorech s jádrem Sandy Bridge.

Díky vzájemné dohodě obou firem by měla být kompatibilita mezi oběma sadami zaručena, což je dobrou zprávou pro vývojáře.

4.2 XMM registry

Základní částí SSE sady jsou registry, se kterými pracují instrukce. Tyto registry můžeme rozdělit do dvou skupin:

1. Registry pro uložení čísel, nad kterými probíhají operace
2. Řídící registr, který nastavuje a řídí operace nad registry z první skupiny

Registry pro práci s čísly

Tyto registry jsou pojmenovány XMM*n*, kde *n* je pořadové číslo registru. V základní SSE sadě bylo 8 registrů, později byla sada rozšířena o dalších 8. Každý XMM*n* registr má délku 128 bitů. V současné době umožňuje uložení těchto typů:

- čtyři 32-bitové čísla v plovoucí řádové čárce s jednoduchou přesností
- dvě 64-bitová čísla v plovoucí řádové čárce s dvojitou přesností
- dvě 64-bitová celá čísla
- čtyři 32-bitová celá čísla
- osm 16-bitových celých čísel
- šestnáct 8-bitových celých čísel

Řídící registr

Tento registr se jmenuje MXCSR. Jednotlivé bity registru jsou buď pro nastavení funkce SSE operací, nebo slouží jako příznaky, které se nastavují při práci s čísly v registrech. Popis možných nastavení a příznaků SSE jednotek je v tabulce 4.1.

Tabulka 4.1: Položky řídícího registru MXCSR

Název	Pozice	Popis příznaku
FZ	15	Při podtečení nastavuje číslo na nulu
Round	13,14	Kombinace bitů určuje způsob zaokrouhlení čísla v plovoucí řádové čárce, popis uveden v tabulce 4.2.
PM	12	Vypíná detekci nepřesnosti výpočtu
UM	11	Vypíná detekci podtečení
OM	10	Vypíná detekci přetečení
ZM	9	Vypíná detekci dělení nulou
DM	8	Vypíná 80-bitovou přesnost při práci s double čísly
IM	7	Vypíná detekci neplatné operace s čísly
DAZ	6	Ovlivňuje počítání s čísly, které nelze převést do standardního exponenciálního tvaru (jsou příliš malá). Pokud je zapnutý, tak tato čísla převádí na 0, což zvyšuje rychlost výpočtu, ale snižuje přesnost.
PE	5	Příznak nepřesnosti výpočtu
UE	4	Příznak podtečení
OE	3	Příznak přetečení
ZE	2	Příznak dělení nulou
DE	1	Příznak 80-bitové přesnosti při práci s double čísly
IE	0	Příznak neplatné operace

Tabulka 4.2: Možnosti zaokrouhlení u SSE jednotek

	Bit 14 = 0	Bit 14 = 1
Bit 13 = 0	RN – klasické zaokrouhlení k nejbližšímu	R+ - zaokrouhlení směrem nahoru
Bit 13 = 1	R- - zaokrouhlení směrem dolů	RZ – zaokrouhlení k nule

Registr má velikost 32 bitů, ale bity 16-31 jsou neobsazené a zápis do jejich pozic způsobuje výjimku.

Po restartu je v registru nastavena hodnota 1F80h, což explicitně zakazuje počítání s 80-ti bitovou přesností, která je samozřejmě pomalejší než standardní přesnost.

Registr MXCSR není možné číst ani nastavovat přímo, je třeba jej zkopírovat do paměti, kde se dají hodnoty načíst a upravit a poté tento blok paměti nahrát celý zpět. Pro tyto operace jsou dostupné dvě instrukce:

1. STMXCSR – uloží obsah registru MXSCR do paměti
2. LDMXSCR – načte do registru MXSCR hodnoty z paměti

Jiné instrukce pro registr MXSCR nejsou podporovány.

4.3 Podpora SSE sady pro jazyk C

Pro programování v jazyku C lze zvolit tyto přístupy:

1. Implementace assemblerovských instrukcí přímo přes funkci **_asm**{instrukce}.
2. Využití knihovny, která přímo podporuje instrukce SSE sady stylem 1:1.

Při implementaci jsem se rozhodl pro druhou variantu. Tyto knihovny jsou napsány pod GNU GPL licencí [43], lze je tedy volně využít.

Pro každou verzi SSE sady je vytvořena jedna knihovna. Příslušné verze knihoven SSE sad na sebe navazují. Jejich přehled je uveden v tabulce 4.3.

Tabulka 4.3: Přehled knihoven, které přímo implementují sadu SSE a sadu MMX

Název knihovny	Verze SSE sady	Přímo navazuje na SSE sadu	Direktiva pro gcc překladač
mmintrin.h	Není SSE, ale MMX	---	-mmmx
xmmintrin.h	SSE1	MMX	-msse
emmintrin.h	SSE2	SSE1	-msse2
pmmintrin.h	SSE3	SSE2	-msse3
tmmintrin.h	SSSE3	SSE3	-mssse3
smmintrin.h	SSE4.1	SSSE3	-msse4.1
nmmintrin.h	SSE4.2	SSE4.1	-msse4.2
ammintrin.h	SSEa	SSE3	-msse4a
bmmintrin.h	SSE5	SSE4a	-msse5

5 Vybrané problémy pro řešení genetickým algoritmem

Genetické algoritmy se nejčastěji používají pro optimalizaci parametrů funkcí. Proto bylo nutné vybrat takové problémy, pro které je řešení genetickým algoritmem vhodné a jsou dostatečně výpočetně složité, aby se na nich dalo prezentovat zrychlení dosažené paralelizací. Pro příliš jednoduché funkce by byly náklady spojené s paralelizací vyšší, než dosažené zrychlení.

Cílem práce je porovnat možnosti zrychlení pro celá i reálná čísla, proto byly vybrány dva problémy:

1. Hledání minima spojité funkce
2. Řešení problému batohu

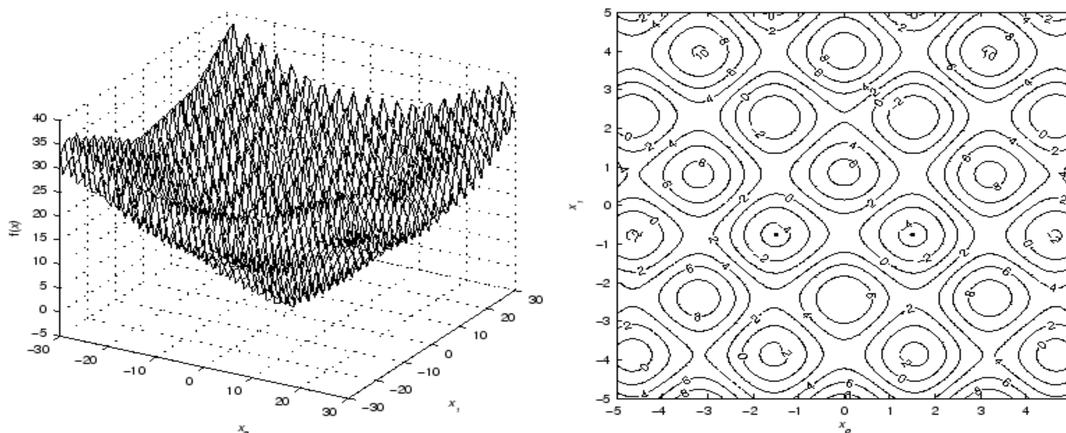
5.1 Hledání minima spojité funkce

Jedná se o nalezení reálných parametrů, pro které je výsledná hodnota funkce minimální. Záměrně jsem vybral funkci, která má velké množství lokálních extrémů a klasický genetický algoritmus by pravděpodobně předčasně konvergoval k nesprávnému výsledků.

Ackleyho funkce

Tato funkce se vypočítá dle vzorce 5.1 [2]. Na obrázku 5.1 je zobrazen průběh funkce pro dvě dimenze ve 3D prostoru a výřez, který zobrazuje dvě hodnoty minima funkce.

$$f_g(\vec{x}) = \sum_{i=1}^{D-1} \left(e^{-0,2} \sqrt{x_i^2 + x_{i+1}^2} + 3(\cos(2x_i) + \sin(2x_{i+1})) \right) \quad (5.1)$$



Obr. 5.1: Průběh Ackleyho funkce pro dvě dimenze [9]

V pravé části obrázku 5.1 jsou tečkami zobrazeny dvě minima funkce. Přesné pozice a hodnoty minim funkce jsou:

1. Pro dvě dimenze
   - pozice $(x_{1,} x_2)=(\mp 1{,}5096201, -0{,}7548651)$
   - hodnota $f(x)=-4{,}5901016$
2. Pro pět dimenzí
   - pozice
     $(x_{1,} x_{2,} x_{3,} x_{4,} x_5)=(\pm 1{,}517, -1{,}1151, -1{,}1096, -1{,}1038, -0{,}7471)$
   - hodnota $f(x)=-13{,}379575$

5.2 Řešení problému batohu

Pro testování algoritmu s celými čísly jsem vybral problém batohu [2]. Máme batoh, jehož nosnost je C. Dále máme množinu n věcí, z nichž i-tá věc má váhu w_i, cenu v_i a proměnnou x_i, která udává přítomnost věci v batohu.

Pro jednotlivé položky platí:

- $i = 1,2,...,n$ – index věci
- w_i, $v_i > 0$ – každá věc má nenulovou váhu i cenu
- x_i ε {0,1} – přítomnost věci v batohu

Celý problém batohu je tedy optimalizační funkcí, v níž se snažíme maximalizovat výpočet ze vzorce 5.2 při dodržení podmínky stanovené ve vzorci 5.3.

$$\sum_{i=1}^{n} v_i x_i \qquad (5.2)$$

$$\sum_{i=1}^{n} w_i x_i \leqslant C \qquad (5.3)$$

6 Základní implementace programu

Pro ověření navržených postupů jsem zvolil implementaci pokročilého genetického algoritmu UMDA, který byl podrobně popsán v druhé kapitole této práce. Sada SSE byla primárně navržena pro práci s reálnými čísly, ale sada SSE2 poskytla širokou podporu pro celočíselné operace, proto jsem se rozhodl implementovat dvě varianty algoritmu:

1. Algoritmus pro hledání reálných parametrů spojité funkce, která má množství lokálních extrémů. Ve druhé kapitole jsou podrobně popsány dva postupy řešení spojitého problému pomocí UMDA. Oba tyto postupy jsem implementoval s cílem ukázat možnosti paralelizace tvorby jednotlivých modelů a vzorkování jedinců, porovnání rychlosti konvergence obou řešení a kvality jejich řešení.
2. Algoritmus pro řešení problému batohu, který pracuje bitově s celočíselnými typy.

6.1 Algoritmus pracující s reálnými čísly

Tento algoritmus hledá parametry, při kterých je hodnota Ackleyho funkce (kap. 5.1) minimální. Nejprve byla implementována diskrétní varianta, poté i spojitá. Tyto varianty UMDA jsou popsány v kapitole 2.2.1.

Obě varianty se od sebe liší stochastickým modelem, ostatní procedury jsou stejné. V dalším textu budu nejprve popisovat funkce, které mají diskrétní i spojitý algoritmus shodné.

Reprezentace jedince

Základní implementace algoritmu počítá Ackleyho funkci pro dvě dimenze. Pro každou dimenzi je uloženo číslo v plovoucí řádové čárce s jednoduchou přesností (*float*), které udává hodnotu parametru. Dále je v jedinci uložena jeho fitness hodnota (*float*). Pro lepší přehlednost je jedinec uložený ve struktuře *tPoint*.

```
typedef struct point
{
        float x;
        float y;
        float fitness;
} tPoint;
```

Generování počáteční populace

Počáteční populace je generována ve funkci *create_population*. Všichni jedinci jsou vygenerováni náhodně v rozsahu určeném konstantami v hlavičkovém souboru.

Výpočet fitness

Pro výpočet fitness zadané populace slouží funkce *compute_fitness*, která uloží vypočítanou fitness do příslušné proměnné v každém jedinci.

Výběr rodičů

Rodiče jsou vybíráni turnajovou selekcí, při níž jsou náhodně vybráni dva jedinci z populace a ten, který má lepší fitness, je uložen do populace rodičů. O tento výběr se stará funkce *selection_parents*.

Sestavení stochastického modelu

V této proceduře se obě varianty algoritmu liší, proto musí být popsány samostatně. Obě pracují se strukturou *tSample*, která je uložená v proměnné *model*. Tato proměnná je naplněna vypočítanými hodnotami ve funkci *create_stochastic_model*.

Diskrétní varianta

Spojitý rozsah je diskretizován na několik sekcí a pro každou sekci je vypočítána pravděpodobnost, že číslo z populace rodičů spadá do rozsahu sekce. Proměnná, která charakterizuje sekci, je uložena ve struktuře:

```
typedef struct sample {
        float xprobably;
        float yprobably;
} tSample;
```

V hlavičkovém souboru je definován počet sekcí, meze jsou vypočítány za běhu programu. Například pro deset sekcí pro rozsah (-1000, 1000) je rozsah první sekce (-1000, -800), druhé (-800, -600) atd. Celkem je tedy pro tento příklad uloženo deset struktur *tSample*.

Spojitá varianta

Ve spojité variantě je model charakterizován střední hodnotou a směrodatnou odchylkou. Pro každou dimenzi je uložena struktura:

```
typedef struct sample {
        float x_mean;           //střední hodnota
```

```
        float x_deviation;        //směrodatná odchylka
        float y_mean;
        float y_deviation;
} tSample;
```

Vzorkování nové populace

Toto je druhá sekce, v níž se diskrétní a spojitý algoritmus zásadně liší. Funkce pro vzorkování nové populace na základě vytvořeného modelu se jmenuje *sample_new_generation.*

Diskrétní varianta

Vzorkovací funkce pro každou dimenzi řešení provádí následující pseudokód:

1. Vygeneruj náhodné číslo z rozsahu hodnot, ve kterém hledáme řešení
2. Urči, do které sekce modelu vygenerované číslo patří
3. Vygeneruj náhodnou pravděpodobnost a porovnej s pravděpodobností ve vybrané sekci
   - pokud je experiment úspěšný, nebo nastala mutace, pak zařaď číslo do potomka a pokračuj pro další dimenzi. Jakmile jsou vygenerovány všechny dimenze, algoritmus ukonči.
   - jinak návrat do bodu 1

Spojitá varianta

Tato varianta je mnohem jednodušší, neboť pro každou dimenzi se pouze rozhoduje, zda nastane mutace nebo bude potomek generován standardní cestou:

1. Nastala mutace – vygeneruj náhodné číslo z rozsahu, pro který hledáme řešení.
2. Nenastala mutace - vygeneruj náhodné číslo z rozsahu, který je určen hodnotami modelu pro danou dimenzi.

Porovnání přístupu ke vzorkování

Na první pohled je zřejmé, že spojitý přístup je velmi jednoduchý a výpočetně rychlý, neboť pro každou dimenzi je proveden jeden experiment s pravděpodobností a na základě jeho výsledku je vygenerována hodnota nového jedince.

Oproti tomu je u diskrétní implementace vysoká pravděpodobnost, že vygenerované číslo nebudeme zařazovat do populace a budeme celý cyklus opakovat. V mezním případě by se mohlo stát, že číslo nebude nikdy vygenerováno a algoritmus bude neustále cyklit v této smyčce. Při

úvahách o možných řešeních tohoto problému jsem navrhl dva přístupy, jak možnosti nekonečného cyklu zamezit:

1. Po každém neúspěšném pokusu o zařazení čísla zvětšit pravděpodobnost všech sekcí v modelu o hodnotu *p*. Tímto postupem by nastalo maximálně *1/p* pokusů, neboť poslední pokus by byl vykonán s polem, v němž by každá sekce měla pravděpodobnost ≥1. Nevýhodou tohoto řešení je nutnost vytváření pracovní kopie modelu a přičítání hodnoty *p* ke každé sekci modelu.
2. Po každém neúspěšném pokusu zmenšit počet sekcí *s* v modelu o 1 a přepočítat pravděpodobnosti. V mezním případě by se vykonalo *s-1* experimentů. Nevýhodou tohoto řešení je opět nutnost pracovní kopie modelu a výpočetní složitost přepočtu modelu.

6.2 Algoritmus pracující s celými čísly

Tento algoritmus hledá podmnožinu z množiny věcí, která se vejde do batohu, přičemž součet cen vybraných věcí je maximální.

Pro toto řešení bude využita diskrétní UMDA. Seznam věcí, které budou zařazeny do batohu, bude uložen binárně v celém čísle.

Původně jsem chtěl implementovat více verzí tohoto algoritmu, které by se lišily bitovou velikostí používaného čísla. Toto bylo po domluvě s vedoucím práce vynecháno, neboť ALU pracuje s 32 bitovými čísly a podpora SSE jednotek je pro 32 bitový typ velmi dobrá. Pro ostatní typy již SSE jednotky neposkytují tolik dostupných operací, proto by režie s konverzemi přesáhla zisk z využití SSE.

Reprezentace jedince

Každý jedinec populace se skládá z pole celých čísel, které obsahuje tolik čísel, aby se do něj bitově vešel seznam všech věcí (určený hodnotou konstanty *THINGS*), a celého čísla, ve kterém je uložena fitness hodnota jedince. Jedinec je uložený ve struktuře tPoint. Konstanta *SIZE_OF_NUM* určuje bitovou velikost čísla(např. 32). Každý bit čísla určuje, zda bude věc vložena do batohu či nikoli. Začíná se prvním bitem prvního čísla, který reprezentuje první věc v seznamu věcí.

```
typedef struct Child
{
        unsigned long int num[(THINGS / SIZE_OF_NUM) + 1];
        int fitness;
} tChild;
```

Generování počáteční populace

Při tvorbě první populace je do každého čísla jedince uloženo náhodné celé číslo. O tuto tvorbu se stará funkce *create_population*.

Výpočet fitness

Postupně se bitově prochází jedinec a do pomocných proměnných se ukládá celková váha jedince a součet cen věcí, které zařazuje do batohu. Pokud váha přesáhne limit batohu, pak je fitness hodnota jedince nulová. Pokud je váha věcí jedince vhodná, pak je fitness hodnota jedince určena součtem cen jednotlivých věcí. Proto algoritmus hledá jedince s maximální hodnotou fitness. Výpočet fitness v programu provádí funkce *compute_fitness*.

Výběr rodičů

Rodiče jsou vybíráni turnajovou selekcí, při níž jsou náhodně vybráni dva jedinci z populace a ten, který má lepší fitness, je uložen do populace rodičů. O tento výběr se stará funkce *selection_parents*.

Sestavení stochastického modelu

Model je uložen jako pole pravděpodobností, z nichž index každé buňky odpovídá indexu věci v seznamu věcí. Model je vytvořen jako struktura *tSample*:

```
typedef struct Sample
{
        float prob[THINGS];
} tSample;
```

Hodnoty každé buňky určují pravděpodobnost, že tato věc byla v populaci rodičů zařazena do batohu. Výpočet hodnot modelu provádí funkce *create_stochastic_model.*

Vzorkování nové populace

Vzorkování nového jedince je prováděno po jednotlivých bitech, které určují zařazení věcí do batohu. Každá věc je s pravděpodobností určenou hodnotou ve stochastickém modelu zařazena do jedince. Pokud nastane mutace, pak je stav věci negován oproti dříve vybrané hodnotě. Toto vzorkování provádí funkce *sample_new_generation*.

6.3 Možnosti akcelerace jednotlivých částí UMDA algoritmu

Důležitým krokem je identifikace částí algoritmu, které jsou vhodné pro paralelizaci. Vývojový diagram implementovaného algoritmu je na obrázku 2.5. Podle tohoto popisu bude rozhodnuto, na které bloky je nutné se při implementaci zaměřit. Základní bloky UMDA algoritmu jsou:

- Tvorba počáteční populace – tento blok se vykoná pouze při startu algoritmu, proto není potřeba jej paralelizovat.
- Výpočet fitness populace – jelikož se fitness vypočítává pro každého potomka každé generace, je tento blok kódu pro paralelizaci velmi vhodný. K paralelizaci lze využít akceleraci výpočtu pomocí SSE sady i spuštění ve více vláknech, neboť výpočty hodnot jednotlivých jedinců jsou vzájemně nezávislé.
- Výběr rodičů – tento blok je spuštěn jednou pro každou generaci. Výběr je prováděn turnajovou selekcí, proto nelze urychlit pomocí SSE sady, ale můžeme výběr provádět paralelně ve více vláknech.
- Výpočet stochastického modelu – tento blok je spuštěn jednou pro každou generaci. Výpočet modelu je prováděn matematickými operacemi, které lze paralelizovat pomocí SSE sady. Pro vícedimenzionální modely lze spustit výpočet paralelně, nicméně vyžaduje přístup každého jádra k modelu, ve kterém se ukládají vypočítané hodnoty.
- Vzorkování nové populace – tento blok je vykonáván pro vytvoření potomků v každé generaci, proto je vhodné jej paralelizovat. V rámci vzorkování není možné využít SSE sady, ale je velmi vhodné spustit vzorkování ve více jádrech.

V následující kapitole budu zjišťovat skutečné náročnosti jednotlivých bloků algoritmu a na základě výsledků vyberu části algoritmu, které budu paralelizovat.

7 Analýza programu

Před přístupem k paralelizaci algoritmu je nutné zjistit, které části programu jsou pro paralelizaci vhodné. Je zřejmé, že nemá smysl paralelizovat část kódu, ve které je vykonávaný program 5 % času, neboť režijní náklady spojené s paralelizaci přesáhnou zisk, který paralelizací získáme.

Měření bude prováděno pro 10000 generací po 1000 jedincích, z nichž je vždy vybíráno 100 rodičů. Pravděpodobnost mutace je 5 %.

7.1 Profilovací nástroj

Ke zjištění četnosti spuštění jednotlivých funkcí algoritmu využijeme profilovací nástroj gprof [32], který je dostupný pro kompilátor *gcc*. Samotné profilování se skládá ze dvou kroků:

1. Měření programu – při spuštění se do souboru *gmon.out* ukládají informace o běhu programu
2. Analýza měření – analyzuje se soubor *gmon.out* a jsou zobrazeny informace o jednotlivých blocích programu

Překlad programu

Pro ukládání informací o běhu programu je nutné použít při překladu parametr *-pg*, pro analýzu naměřených dat se používá program *gprof*.

Ostatní parametry použité při překladu jsou:

gcc-4.4 GA_norm.c -lm -std=gnu99 -o prog -Wall -pedantic -pg

Při vlastním překladu nebyl využit parametr *-O3*, který optimalizuje výsledný kód, neboť jednou z optimalizací je, že všechny funkce jsou typu *inline*, tudíž je program sestaven jako jeden blok bez volání funkcí. Toto sestavení lze profilovat, ale nejsou zřejmé počty volání jednotlivých funkcí a jejich časová náročnost. Proto byl při profilování tento parametr odstraněn. Podrobný popis ostatních parametrů v [33].

7.2 Diskrétní verze algoritmu s reálnými čísly

Získaný profil této implementace je uložen v tabulce 7.1. Nejnáročnější částí je dle předpokladu vzorkování nové populace - funkce *sample_new_generation*. Časově druhou nejnáročnější funkcí je

generování náhodného čísla – funkce *frand* a třetí v pořadí je výpočet fitness - funkce *compute_fitness*.

Tabulka 7.1: Profil diskrétní verze algoritmu s reálnými čísly

Název funkce	**Popis funkce**	**Počet volání**	**% z celkového času**
sample_new_generation	*Vzorkuje novou populaci*	10000	83,16
frand	*Náhodné číslo z určeného rozsahu*	182755649	10,34
compute_fitness	*Výpočet fitness jedinců*	10000	5,01
create_population	*Tvorba náhodné populace*	1	0,86
selection_parents	*Výběr rodičů*	10000	0,47
create_stochastic_model	*Tvorba pravděpodobnostního modelu*	10000	0,08
get_best_parent	*Hledání nejlepšího rodiče*	10000	0,08

7.3 Spojitá verze algoritmu s reálnými čísly

Získaný profil této implementace je uložen v tabulce 7.2. Oproti diskrétní variantě se značně zmenšila náročnost vzorkování nové populace. Nejnáročnější je výpočet fitness - funkce *compute_fitness*, druhá je funkce vracející náhodné číslo ze zadaného rozsahu - funkce *frand_range* a třetí je vzorkování nové populace - funkce *sample_new_generation*.

Tabulka 7.2: Profil spojité verze algoritmu s reálnými čísly

Název funkce	**Popis funkce**	**Počet volání**	**% z celkového času**
compute_fitness	*Výpočet fitness jedinců*	10000	67,09
frand_range	*Náhodné číslo ze zadaného rozsahu*	18999538	13,92
sample_new_generation	*Vzorkuje novou populaci*	10000	11,39
selection_parents	*Výběr rodičů*	10000	5,06
create_stochastic_model	*Tvorba pravděpodobnostního modelu*	10000	1,27
frand	*Náhodné číslo z rozsahu hledání*	1002462	0,63
create_population	*Tvorba náhodné populace*	1	0,63
get_best_parent	*Hledání nejlepšího rodiče*	10000	zanedbatelné

7.4 Algoritmus s celými čísly

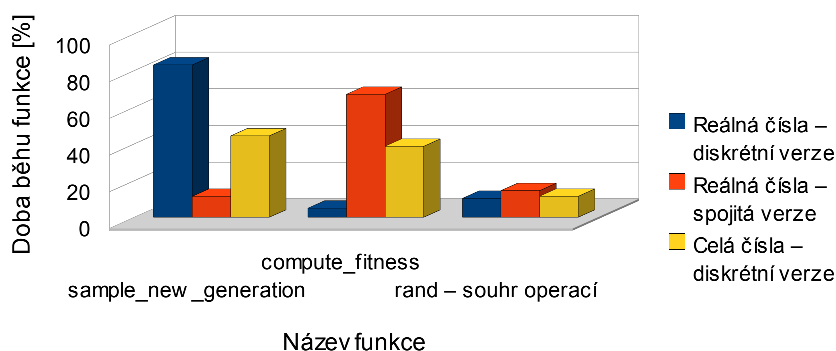
Získaný profil této implementace je uložen v tabulce 7.3. Nejnáročnější částí této implementace je vzorkování nové populace - funkce *sample_new_generation*, neboť se vzorkuje po bitech, což je výpočetně náročné. Druhá je funkce pro výpočet fitness - funkce *compute_fitness* a třetí výpočet náhodné pravděpodobnosti - funkce *rand_prob*.

Tabulka 7.3: Profil celočíselného algoritmu řešícího problém batohu

Název funkce	**Popis funkce**	**Počet volání**	**% z celkového času**
sample_new_generation	*Vzorkuje novou populaci*	10000	44,38
compute_fitness	*Výpočet fitness jedinců*	10000	38,8
rand_prob	*Vrátí náhodné číslo v rozsahu (0,1)*	399989504	11,54
get_random_things	*Vygeneruje seznam věcí*	1	3,59
create_stochastic_model	*Tvorba pravděpodobnostního modelu*	10000	1,44
selection_parents	*Výběr rodičů*	10000	0,25
get_best_parent	*Hledání nejlepšího rodiče*	10000	zanedbatelné
create_population	*Tvorba náhodné populace*	1	zanedbatelné
get_things	*Nastaví seznam věcí*	1	zanedbatelné
print_child	*Srozumitelně vytiskne jedince*	1	zanedbatelné

7.5 Souhrn výsledků profilování

Při pohledu na jednotlivé výsledky lze říci, že časově nejsložitějšími operacemi jsou vzorkování nové populace, výpočet fitness hodnoty jedince a generování náhodného čísla. Náročnost těchto operací je zobrazena v grafu na obrázku 7.1. Na tyto vybrané funkce je třeba se zaměřit při celkové akceleraci algoritmu.



Obr. 7.1: Porovnání náročnosti vybraných funkcí implementovaných algoritmů.

8 Generátor náhodných čísel

Každý genetický algoritmus pracuje s využitím náhodného prvku, který mu slouží k rozhodování o dalších krocích.

V nejlepším případě bychom použili fyzikální generátor náhodných čísel, který vlastní hodnotu náhodného čísla odvozuje z náhodných jevů. Například lze digitalizovat šum na polovodičovém přechodu či atmosférický šum, nebo podle zákonů kvantové fyziky je pravděpodobnost rozpadu atomového jádra izotopu přesně 50 %, což je nejlepší možné rozložení pro generování náhodného čísla.

Bohužel tyto generátory nejsou buď dostupné, nebo nezvládají rychlostní požadavky genetického algoritmu. Proto jsme nuceni využít generátor pseudonáhodných čísel.

8.1 Generátor pseudonáhodných čísel

Generátor pseudonáhodných čísel je deterministický program, který inicializujeme náhodnou hodnotou, například systémovým časem. Tomuto údaji se říká náhodné semínko.

Pro určení kvality generátoru se sledují následující parametry:

- Rozdělení čísel – zda se generují čísla z celého rozsahu hodnot generátoru
- Nezávislost čísel – zda jsou jednotlivé členy posloupnosti vzájemně nezávislé
- Výskyt posloupností – zda se v generované řadě neopakují různé posloupnosti

Nejčastější implementací pseudonáhodného generátoru je lineární kongruentní generátor.

8.1.1 Lineární kongruentní generátor

Výpočet čísla v generátoru je prováděn dle vzorce 3.1 [34].

$$X_{n+1} = (aX_n + b) \bmod m \tag{3.1}$$

Popis proměnných použitých ve vzorci:

- A,b,m – vhodně zvolené konstanty
- X_0 – náhodné semínko

Generátor generuje čísla v rozsahu $0 \leq x_i < m$. Generovaná čísla budou v celém rozsahu hodnot právě tehdy, pokud budou splněna pravidla:

1. b a m jsou nesoudělná čísla
2. $(a - 1)$ je dělitelné všemi prvočíselnými faktory čísla m
3. $(a - 1)$ je násobek m, jestliže m je násobek 4

Tabulka 8.1: Některé často používané nastavení konstant generátoru

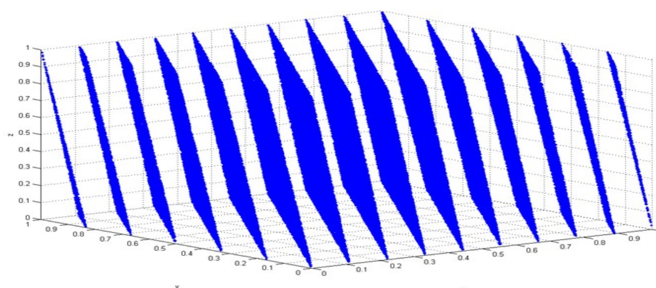
Zdroj	*a*	*b*	*m*
Numerical recipes	1664525	1013904223	2^{32}
glibc	22695477	1	2^{32}
Borland delphi	134775813	1	2^{32}
MS Visual studio	214013	2531011	2^{32}
Park & Miller	16807	0	$2^{32} - 1$
Java API	25214903917	11	2^{48}

Lineární kongruentní generátor je nejpoužívanější způsob generování, neboť je dostatečně kvalitní a současně jednoduchý na implementaci. Pro správnou funkčnost je samozřejmě nutné správně nastavit parametry a, b, m, jinak se můžou projevit závislosti mezi generovanými čísly.

Typickým představitelem takovéto chyby je implementace RANDU, která měla nastavené parametry:

- $a = 65539$
- $b = 0$
- $m = 2^{31}$

Na první pohled byla generovaná čísla pseudonáhodná, ale při zobrazení v tří dimenzionálním grafu se zjistilo, že všechna čísla jsou generovaná v 15-ti dvou dimenzionálních rovinách (obr. 8.1).



Obr. 8.1.: Možné důsledky nastavení špatných konstant generátoru [35]

8.1.2 SSE implementace generátoru

Při analýze programu jsem zjistil, že rychlost generování pseudonáhodných čísel je pro výslednou rychlost algoritmu kritická, neboť se jedná o nejčastěji volanou položku, která se využívá při výběru rodičů a při tvorbě nových jedinců ze stochastického modelu.

Z tohoto důvodu jsem hledal možnosti, jak je možné zrychlit generování pseudonáhodných čísel při zachování rozsahu hodnot standardního generátoru, který je implementován v knihovně *stdlib.h* pro jazyk C. Při hledání jsem se samozřejmě zaměřil na možnost využití SSE jednotek.

Hledanou implementaci generátoru jsem našel v [36]. Tuto implementaci lze používat i upravovat bez omezení.

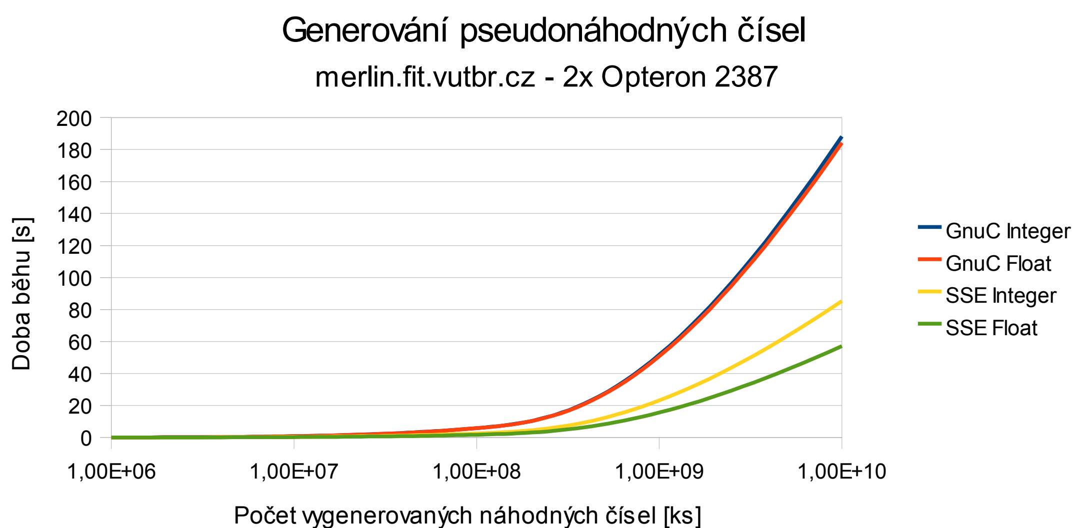
Problémem této implementace byla nekompatibilita mezi překladačem gcc a Intel® Compiler 7.0 při zarovnávání adres proměnných. Dále nebyl dodržen způsob načtení hodnoty dalšího pseudonáhodného čísla, které se vracelo přes parametr funkce, nikoli jako její návratová hodnota.

Uvedené problémy jsem vyřešil a přidal funkci *frand_sse*, která vrací pseudonáhodné číslo jako pravděpodobnost v rozmezí (0,1).

Popis generátoru včetně testů jeho kvality jsou v [36]. Po implementaci jsem provedl časové porovnání se standardním *GnuC* generátorem. Naměřené časy jsou v tabulce 6.2 a 6.3. Pro lepší přehlednost jsem výsledky zobrazil do grafů na obrázcích 8.2 a 8.3.

Tabulka 6.2: Naměřené časy na serveru merlin.fit.vutbr.cz

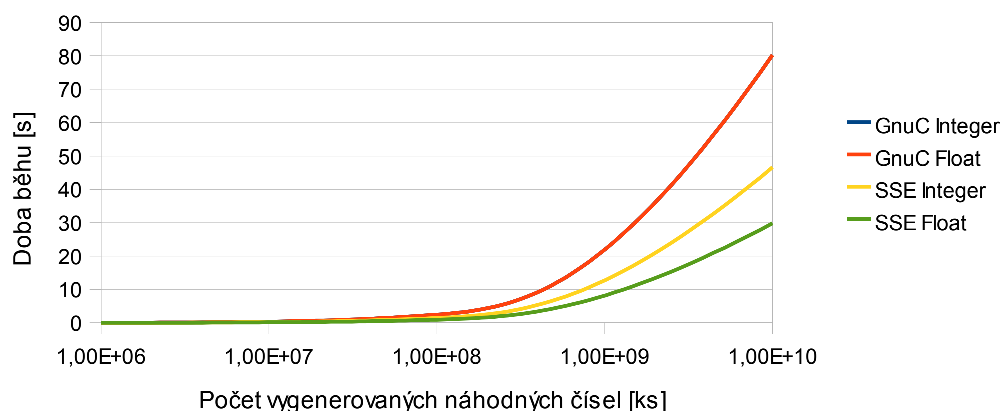
merlin.fit.vutb.cz – 2 x Opteron 2389				
	gnuC [s]		*fastRand [s]*	
Opakování	*GnuC Integer*	*GnuC Float*	*SSE Integer*	*SSE Float*
1,00E+06	0,02	0,02	0,02	0
1,00E+07	0,2	0,19	0,08	0,06
1,00E+08	1,94	1,92	0,83	0,57
1,00E+09	19,27	19,13	8,34	5,7
1,00E+10	188,24	184,22	85,23	57,15



Obr. 8.2: Graf časů generování čísel na serveru merlin.fit.vutbr.cz

Tabulka 6.3: Naměřené časy na serveru pcjaros-gpu.fut.vutbr.cz

pcjaros-gpu.fit.vutbr.cz – Core i7				
	gnuC [s]		*fastRand [s]*	
Opakování	*Integer*	*Float*	*Integer*	*Float*
1,00E+06	0,01	0	0	0
1,00E+07	0,08	0,08	0,05	0,03
1,00E+08	0,81	0,81	0,46	0,3
1,00E+09	8,02	8,02	4,66	2,99
1,00E+10	80,24	80,26	46,57	29,8



Obr. 8.3: Graf časů generování čísel na serveru pcjaros-gpu.fit.vutbr.cz

V grafech je zřejmé, že nejrychlejší je SSE implementace generování pseudonáhodného čísla v rozsahu (0,1), které dosáhla zrychlení 2,7 v porovnání se standardním *GnuC* generátorem. Při generování celého čísla v rozsahu (0, 2^{31}) jsme SSE implementací dosáhli zrychlení 1,7 oproti *GnuC* generátoru.

9 Akcelerace genetického algoritmu pomocí SSE jednotek

V této kapitole se budeme zabývat akcelerací vytvořených programů s využitím SSE jednotek. Ke zrychlení bude také využit SSE generátor pseudonáhodných čísel, který byl popsán v kapitole 8.

V sedmé kapitole byla vybrána tři stěžejní místa, která je potřeba zrychlit. Těmi jsou:

1. Generování pseudonáhodných čísel
2. Výpočet fitness funkce
3. Vzorkování nové populace

9.1 Generování pseudonáhodných čísel

Do programů byl implementován SSE generátor, který je popsán v kapitole 8. Generátor obsahuje tři funkce:

- *srand_sse* – inicializuje generátor a vkládá náhodné počáteční číslo (semínko)
- *rand_sse* – vrací pseudonáhodné číslo v rozsahu (0-2^{31})
- *frand_sse* – vrací pseudonáhodné číslo v rozsahu (0,1)

Při implementaci do programu stačilo nahradit používanou funkci *rand* funkcí *rand_sse*. Pokud se z náhodného čísla vypočítávala pravděpodobnost, pak tato sekce byla nahrazena funkcí *frand_sse*.

Implementací tohoto generátoru bylo dosaženo výrazného zrychlení pro diskrétní UMDA, neboť se v algoritmu velmi často pracuje s náhodnými čísly. Menší zrychlení spojitého algoritmu je způsobeno menší potřebou generování náhodných čísel, tudíž se vlastní zrychlení neprojeví tak výrazně. Přehled dosažených zrychlení je v tabulce 9.1.

Tabulka 9.1: Zrychlení algoritmů implementací SSE generátoru

Typ algoritmu	***Doba běhu původně [s]***	***Doba běhu s random_sse.h [s]***	***Dosažené zrychlení***
Reálná čísla – diskrétní verze	24,26	8,96	2,7
Reálná čísla – spojitá verze	1,82	1,4	1,3
Celá čísla – diskrétní verze	23,09	10,83	2,1

9.2 Výpočet fitness funkce

V implementovaných programech, které pracují s reálnými čísly, je hodnota fitness funkce vypočítána ze vzorce 5.1. Tento vzorec obsahuje klasické matematické operace, které jsou podpořeny v SSE sadě. Jedinou výjimkou je výpočet goniometrických funkcí. Tento výpočet musí být proveden funkcemi ze standardní C knihovny *math.h*.

V programu s celými čísly se určuje suma vah a cen věcí, které chceme vložit do batohu. Tyto operace lze také provést pomocí SSE sady.

9.2.1 Fitness funkce programu s reálnými čísly

Jelikož máme dostupné místo pro čtyři reálná čísla, pak je nutné vybrat princip, kterým k paralelizaci výpočtu přistoupíme:

1. Paralelizace jednotlivých výpočtů v rámci potomka
2. Paralelizace současnou prací se čtyřmi potomky

Rozhodl jsem se pro druhou variantu, proto jsem v každém cyklu vypočítal fitness hodnotu pro čtyři potomky.

Malou slabinou tohoto výpočtu je absence goniometrických funkcí, které nejsou v SSE sadě podpořeny. Proto je nutné tyto hodnoty předpočítat sekvenčním algoritmem, než jsou vloženy do SSE registrů.

Výpočet Ackleyho funkce pro dvě dimenze je možné rozdělit do dvou sekcí. Výsledná hodnota funkce potom vznikne jejich součtem. První sekce je zobrazena ve vzorci 9.1, druhá sekce ve vzorci 9.2.

$$s_1 = e^{-0,2}\sqrt{x_i^2 + x_{i+1}^2} \qquad (9.1)$$

Vzorec 9.1 je v programu vypočítán tímto kódem:

```
        //výpočet čísla e^-0,2 a jeho uložení do všech částí XMM registru
numpower = _mm_set1_ps(pow(M_E, -0.2));
        //cyklu, který bere vždy čtyři potomky
for (i = 0; i < POPULATION; i = i + 4)
{               //uložení hodnot čtyř potomků do XMM registru
        pom1 = _mm_set_ps(population[i].x, population[i+1].x,
                        population[i+2].x, population[i+3].x);
                //provede druhou mocnicu vloženého čísla
        pom1 = _mm_mul_ps(pom1, pom1);
        pom2 = _mm_set_ps(population[i].y, population[i+1].y,
                        population[i+2].y, population[i+3].y);
```

```
pom2 = _mm_mul_ps(pom2, pom2);
        //součet vypočítaných druhých mocnin čísel
pom1 = _mm_add_ps(pom1, pom2);
        //odmocnina vypočítaného součtu
pom1 = _mm_sqrt_ps(pom1);
        //násobení vypočítaného čísla konstantou  e^-0,2
sec1 = _mm_mul_ps(pom1, numpower);
```

$$s_2 = 3(\cos(2x_i) + \sin(2x_{i+1})) \quad (9.2)$$

Vzorec 9.2 je v programu vypočítán tímto kódem:

```
        //uložení konstanty 3 do všech polí XMM registru
num3 = _mm_set1_ps(3.0);
        //cyklu, který bere vždy čtyři potomky
for (i = 0; i < POPULATION; i = i + 4)
{
                //uložení vypočítaných goniometrických funkcí do pomocných proměnných
        pom1 = _mm_set_ps(cos(2*(population[i].x)), cos(2*(population[i+1].x)),
                        cos(2*(population[i+2].x)), cos(2*(population[i+3].x)));
        pom2 = _mm_set_ps(sin(2*(population[i].y)), sin(2*(population[i+1].y)),
                        sin(2*(population[i+2].y)), sin(2*(population[i+3].y)));
                //součet hodnot
        pom1 = _mm_add_ps(pom1, pom2);
                //násobení výpočtu konstantou 3
        sec2 = _mm_mul_ps(pom1, num3);
        .........
```

Jelikož jsou obě sekce uloženy v XMM registru, tak zbývá provést součet obou sekcí a jejich uložení do proměnných s fitness jednotlivých potomků:

```
        //proměnná, přes kterou budeme načítat hodnoty z XMM registrů
float cislo[4];
        //součet vypočítaných sekcí
fit = _mm_add_ps(sec1, sec2);
        //uložení hodnot z XMM registru do zadaného pole reálných čísel
_mm_storeu_ps((float *)cislo, fit);
        //vlastní uložení výpočtů k jednotlivý potomkům
population[i].fitness = cislo[3];
population[i+1].fitness = cislo[2];
```

```
population[i+2].fitness = cislo[1];
population[i+3].fitness = cislo[0];
```

9.2.2 Fitness funkce programu s celými čísly

Tato funkce byla implementována opačným způsobem, než tomu bylo pro algoritmus s reálnými čísly – byla zvolena paralelizace v rámci jedince. Tento postup byl zvolen zejména kvůli nutnosti bitových posunů pro každý bit potomka. Tyto posuny nejsou přímo implementovány sadou SSE, proto bylo nutné použít náhradní řešení využívající násobení.

Komentovaný zdrojový kód výpočtu fitness hodnoty u problému batohu:

```
        //tuto proměnnou využívám k rotaci bitů v masce, pokud vektor (1,2,4,8) vynásobím vektorem
        (16,16,16,16), pak je to stejný efekt, jako kdyby proběhl u všech čísel bitový posun o čtyři pozice.
rotate = _mm_set1_ps(16);
        //proměnná pro zjištění, zda je v registru 0 nebo jiné číslo. Nutné pro možnost využít SSE instrukci
        _mm_cmpgt_epi32
zero = _mm_setzero_si128();

for (k = 0; k < POPULATION; k++)
{
        //vymazání proměnných, které jsou využity pro každého potomka. Instrukce vytvoří nulový vektor.
 act_price = _mm_setzero_si128();
 act_weigh = _mm_setzero_si128();
 mask = _mm_setzero_si128();
        //tato proměnná slouží k indexaci věci v seznamu věcí
 position = 0;
        //program pracuje s čísly, jejichž bitová velikost je uložena v SIZE_OF_NUM. Pokud je více jak 32 věcí.
        Pokud je například 80 věcí a 32 bitové číslo, pak je seznam uložen ve třech číslech. Tato for smyčka
        projde všechna uložená čísla.
 for (j = 0; j < ((THINGS / SIZE_OF_NUM) + 1); j++)
 {
        //uložení celého čísla, které udává, zda věci zaradíme do batohu či nikoli
  act_num = _mm_set1_epi32(population[k].num[j]);
        //v tomto cyklu se po bitu projde celé uložené číslo. Zpracovávají se vždy čtyři bity v jednom průchodu
   for (i = 0; i < SIZE_OF_NUM; i = i + 4)
  {
   if (position < THINGS)
   {
```

```
        //pokud jsme na začátku čísla, musíme si inicializovat masku. Tato maska určuje, ze budeme pracovat s
        bity číslo (1,2,3,4)
      if (i == 0)
        mask = _mm_set_epi32(1,2,4,8);
      else
      {
        //Tato část zajišťuje posun masky na další čtyři bity. Jelikož SSE jednotky nepodporují přímo násobení
        celých čísel, musí se nejprve převést na reálné, pak provést násobení a poté převést zpět na celé.
        mult = _mm_cvtepi32_ps(mask);
        mult = _mm_mul_ps(mult, rotate);
        mask = _mm_cvttps_epi32(mult);
      }
        //Vymaskování příslušných bitů, na které ukazuje maska
      res = _mm_and_si128(act_num, mask);
        //Určení, které z čísel obsahuje číslo a které nulu. Pokud bylo některé pole číslo, pak je nahrazeno
        číslem -1(v hexa 0xFFFFFFFF), jinak zůstane nula. Toto nám umožňuje vymaskování čísel, které
        nemají být do batohu zařazeny.
      res = _mm_cmpgt_epi32(res, zero);
        //Načtení cen a vah jednotlivých věcí
      thingsPrice = _mm_set_epi32(thingsList[position].price,thingsList[position+1].price,
                              thingsList[position+2].price,thingsList[position+3].price);
      thingsWeigh = _mm_set_epi32(thingsList[position].weigh,thingsList[position+1].weigh,
                              thingsList[position+2].weigh,thingsList[position+3].weigh);
        //Vymaskování hodnot, které nemají být do batohu zařazeny a přičtení zbylých do proměných, ve kterých
        je uložená průběžná váha a cena věcí
      thingsPrice = _mm_and_si128(thingsPrice, res);
      thingsWeigh = _mm_and_si128(thingsWeigh, res);
      act_price = _mm_add_epi32(act_price, thingsPrice);
      act_weigh = _mm_add_epi32(act_weigh, thingsWeigh);
        //Posun indexu na další čtyři věci
      position = position + 4;
}}}
        //Načtení vypočítaných vah do pole celých čísel
  _mm_storeu_si128((__m128i*)ret, act_weigh);
        //Test, zda vybrané věci vlezou do batohu
  if ((ret[0] + ret[1] + ret[2] + ret[3]) <= BAGSIZE)
  {
        //Věci se do batohu vejdou, načtení jejich ceny a uložení této hodnoty jako fitness jedince
```

```
    _mm_storeu_si128((__m128i*)ret, act_price);
    population[k].fitness = ret[0] + ret[1] + ret[2] + ret[3];
  }
  else
        //Váha věcí je větší než nosnost batohu, fitness jedince je nulová
    population[k].fitness = 0;
```

9.2.3 Zrychlení programu implementací SSE fitness funkce

Po implementaci výpočtů fitness hodnot pomocí SSE sady jsem provedl měření doby běhu jednotlivých algoritmů. Přehled výsledků je v tabulce 9.2.

Diskrétní implementace algoritmu s reálnými čísly byla zrychlena jen nepatrně, neboť výpočet fitness funkce zabíral jen 5 % původního času výpočtu. Oproti tomu u ostatních implementací se zrychlení výpočtu fitness funkce pomocí SSE sady projevilo výrazněji. U spojité verze bylo dosaženo zrychlení o 20 % a u algoritmu s celými čísly bylo dosaženo zrychlení 16 %.

Tabulka 9.2: Zrychlení algoritmů implementací SSE sady při výpočtu fitness

Typ algoritmu	***Doba běhu původně [s]***	***Doba běhu s akcelerovanou fitness funkcí [s]***	***Dosažené zrychlení***
Reálná čísla – diskrétní verze	8,96	8,72	1,03
Reálná čísla – spojitá verze	1,4	1,17	1,2
Celá čísla – diskrétní verze	10,83	9,35	1,16

9.3 Vzorkování nové populace

Pro implementaci SSE sady je vhodné jen vzorkování u algoritmu pracujícím s reálnými čísly pomocí diskretizace hodnot.

U ostatních algoritmů je implementace SSE sady nevhodná, neboť:

- Algoritmus pracující s reálnými čísly pomocí spojitého modelu při vzorkování jen rozhoduje, zda nastane mutace (náhodné číslo z celého rozsahu), nebo bude generováno číslo z rozsahu stochastického modelu.
- Algoritmus pracující s celými čísly pracuje po jednotlivých bitech (není přímá podpora SSE pro bitové posuny) a pro každý bit generuje pravděpodobnost, kterou porovnává s pravděpodobností v modelu. Při mutaci vkládá do potomka méně pravděpodobnou

hodnotu z modelu. SSE sada podporuje operace porovnání, ale v tomto případě není možné souhrnné vyhodnocení a snaha o jeho implementaci by zrušila zrychlení získané využitím SSE sady.

9.3.1 Vzorkovací funkce programu s reálnými čísly využívajícím diskretizaci hodnot

Tato funkce pro každé vygenerované číslo určí, do které sekce z rozsahu algoritmu patří a poté toto číslo s pravděpodobností udanou ve stochastickém modelu zařadí jako potomka do populace.

Komentovaný kód vzorkování nového jedince:

```
        //výpočet rozsahu jedné sekce modelu a jeho uložení do XMM registru
oneStep = abs(RANGE) / SAMPLES;
step = _mm_set1_ps(oneStep);
        //Uložení minimálního rozsahu výpočtu. Tato hodnota je uložena kvůli potřebě výpočtu indexu do pole,
        slouží k posunu indexu do kladných hodnot
min = _mm_set1_ps(abs(MIN));

for (i = 0; i < POPULATION; i++)
{
        //Příznaky, že příslušná hodnota byla zařazena do potomka
  xflag = 0;
  yflag = 0;
        //Dokud nejsou oba potomci vygenerováni, pak opakuj
  while ((xflag == 0) || (yflag == 0))
  {
        //Vygenerování náhodných čísel a jejich uložení do XMM resistru
    random_numbers[0] = frand();
    random_numbers[1] = frand();
    random_numbers[2] = frand();
    random_numbers[3] = frand();
    num = _mm_load_ps(random_numbers);
        //Výpočet indexu, který odpovídá příslušným číslům. Jelikož je to jedna z nejnáročnějších operací v této
        funkci a je malá pravděpodobnost, že hned první číslo bude přijato, tak se počítají vždy čtyři čísla, která
        se poté sekvenčně projdou.
    num = _mm_add_ps(num, min);
    num = _mm_div_ps(num,step);
    indexes = _mm_cvtps_epi32(num);
```

```
_mm_storeu_si128((__m128i*)ind, indexes);
        //Průchod všemi vygenerovanými čísly
    for (j = 0; j < 4; j++)
    {
     if (xflag == 0)
     {
        //Test, zda zařadit číslo do potomka nebo zda nastává mutace
      if ((model[ind[j]].xprobably > frand_sse()) || (frand_sse() < MUTATION))
      {
       draft[i].x = random_numbers[j];
       xflag = 1;
      }
     }
     if (yflag == 0)
     {
      if ((model[ind[j]].yprobably > frand_sse()) || (frand_sse() < MUTATION))
      {
       draft[i].y = random_numbers[j];
       yflag = 1;
}}}}}
```

9.3.2 Zrychlení programu implementací SSE vzorkování

Jak bylo uvedeno výše, tak byla implementováno využití SSE sady jen do vzorkování algoritmu s reálnými parametry, který využívá princip diskretizace. Výsledek zrychlení této implementace je v tabulce 9.3.

Po implementaci bylo zjištěno zrychlení algoritmu o 65%, což je velmi dobrý výsledek. Toto zrychlení je způsobeno faktem, že při vzorkování je nutné určovat oblast, do které vygenerované číslo spadá a pak toto číslo zařadit do potomka s pravděpodobností v modelu. Právě výpočet indexu oblasti bylo možné paralelizovat pomocí SSE sady, tudíž se současně zpracovávají čtyři čísla, což výrazně zrychluje provádění vzorkování.

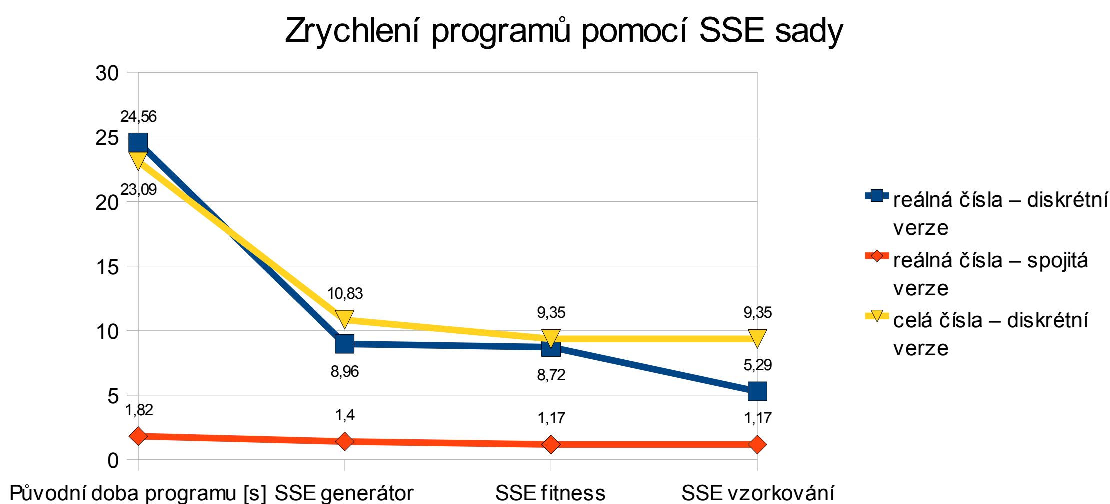
Tabulka 9.3: Zrychlení algoritmu implementací SSE sady při vzorkování algoritmu

Typ algoritmu	*Doba běhu původně [s]*	*Doba běhu s akcelerovanou funkcí pro vzorkování [s]*	*Dosažené zrychlení*
Reálná čísla – diskrétní verze	8,72	5,29	1,65

9.4 Souhrn zrychlení dosaženého využitím SSE jednotek

Hlavním důvodem zrychlení algoritmů byla implementace SSE pseudonáhodného generátoru čísel, který přímo podporuje celá čísla i vrácení pravděpodobnosti. Tyto operace jsou v rámci genetického algoritmu velmi časté, proto každé zrychlení v této sekci je na výsledné době běhu programu znatelné.

Dalšího zrychlení bylo dosaženo implementací SSE sady do výpočtu fitness funkce a do vzorkování nové populace. Vývoj zrychlování jednotlivých algoritmů je vynesen do grafu na obrázku 9.1. Pro lepší představu o vývoji zrychlení byly zobrazeny u bodů grafu přesné hodnoty, neboť například pro spojitý model není vývoj zrychlení z grafu dobře čitelný.



Obr. 9.1: Zrychlení programu pomocí SSE sady

V grafu je zřejmé, že u všech implementací bylo dosaženo implementací SSE sady velmi dobré zrychlení.

Nejvýraznější zrychlení bylo dosaženo u diskrétních variant algoritmu. U těchto algoritmů bylo dosaženo celkové zrychlení 4,62 a 4,36. Takto vysoké hodnoty je způsobeny faktem, že bylo možno

rozčlenit paralelní části na nezávislé výpočty, které mohou probíhat současně v SSE sadě. Navíc implementace SSE pseudonáhodného generátoru zrychlila i sekvenční část algoritmu.

U spojitého algoritmu s reálnými čísly bylo dosaženo zrychlení 1,55. Relativně nízké zrychlení je způsobeno faktem, že jedinou vhodnou částí pro implementaci SSE sady byl výpočet fitness hodnoty jedince. Ostatní části algoritmu nebyly pro paralelizaci vhodné. Algoritmus byl samozřejmě zrychlen i implementací SSE pseudonáhodného generátoru.

Z výsledků je zřejmé, že diskrétní implementace algoritmu UMDA jsou velmi vhodné pro paralelizaci pomocí sady SSE a k využití implementovaného SSE pseudonáhodného generátoru.

10 Paralelizace programu pomocí OpenMP

Paralelizace pomocí systému OpenMP je poslední částí, kterou se budeme snažit vytvořené algoritmy zrychlovat.

Při plánování postupu paralelizace je nutné zahrnout, že výpočet může probíhat na počítači, který má mnoho jader, ale plánovač operačního systému využití všech jader neumožní. Přesněji je umožní jen fiktivně, přičemž všechny paralelní části budou sekvenčně vykonány na jednom jádru.

Z tohoto důvodu jsem se rozhodl, že jednotlivé bloky programu rozdělím na menší celky, které bude jednoduché paralelizovat a přitom neporuším logickou stavbu programu.

10.1 Postup paralelizace

Při paralelizaci vycházím z hodnot, které jsem zjistil při profilování jednotlivých algoritmů. Z naměřených hodnot je zřejmé, že nejvhodnějšími funkcemi pro paralelizaci jsou výpočet fitness hodnoty a vzorkování nových jedinců ze stochastického modelu.

Tyto funkce jsem upravil tak, že zpracovávají vždy jen určenou část jedinců. Průchod všemi jedinci je poté proveden ve *for* smyčce.

Dalším problémem paralelního zpracování bylo generování náhodných čísel pro jednotlivá vlákna. Pokud by vlákna pracovala s jedním generátorem, pak by je čekání na jednotlivá čísla velmi zdržovalo. Z tohoto důvodu byly vytvořeny funkce *load_seed*(*číslo vlákna*) a *store_seed*(*číslo vlákna*), které ukládají a načítají data generátoru pro každé vlákno. Konstanta *BLOCKSIZE* určuje počet jedinců, které se bude zpracovávat v jednom vlákně výpočtu. Pro jasnou představu o funkci kódu přikládám část zdrojového kódu, který provádí výše popsaný kód:

```
#pragma omp parallel for shared(draft) firstprivate(model)
for (j = 0; j < POPULATION; j = j + BLOCKSIZE)
{
  load_seed(omp_get_thread_num());
  sample_new_generation(draft, &model, j, j + BLOCKSIZE);
  store_seed(omp_get_thread_num());
}
```

Procedura *omp_get_thread_num()* je součástí systému OpenMP a vrací číslo jádra, na kterém je program vykonáván.

Tento kód zajistí, že jednotlivé cykly for smyčky budou vykonány paralelně a výsledky budou uloženy do sdílené proměnné *draft*. Proměnná *model* slouží pouze ke čtení hodnot, proto je v klauzuli *firstprivate*, která ji učiní privátní pro jednotlivá vlákna a uloží do ní hodnoty, které obsahovala při zahájení paralelního zpracování.

10.2 Zrychlení dosažené paralelizací systémem OpenMP

Po implementaci navrženého postupu jsem opět provedl měření doby běhu programu. Naměřené výsledky jsem zapsal do tabulky 10.1.

Tabulka 10.1: Porovnání doby běhu algoritmu po implementaci systému OpenMP

Typ algoritmu	***Doba běhu s využitím SSE sady [s]***	***Doba běhu s využitím systému OpenMP [s]***	***Dosažené zrychlení***
Reálná čísla – diskrétní verze	5,29	5,03	1,05
Reálná čísla – spojitá verze	1,4	0,84	1,66
Celá čísla – diskrétní verze	10,83	3,23	3,35

Po implementaci systému OpenMP bylo dosaženo u diskrétní verze s reálnými parametry jen velmi malého zrychlení. Důvodem takto malého zrychlení je fakt, že k paralelizaci byla vhodná jen funkce pro výpočet fitness. Ostatní funkce jsou již relativně jednoduché a dostatečně rychlé, proto zrychlení získané paralelizací převýšila režie se spuštěním ve více vláknech.

U spojité verze algoritmu s reálnými čísly bylo zrychlení pomocí systému OpenMP lepší než u diskrétní varianty. V tomto algoritmu byl paralelizován výpočet fitness a výběr rodičů do nové populace a dosažené zrychlení bylo 1,66.

Nejvhodnějším algoritmem pro paralelizaci systémem OpenMP byla diskrétní varianta algoritmu, která pracuje s celými čísly. V tomto algoritmu bylo možné paralelizovat výpočet fitness hodnoty a vzorkování nové populace. Zrychlení po implementaci systému OpenMP bylo 3,35.

11 Celková akcelerace implementovaných algoritmů

V tabulce 11.1 jsou zobrazeny jednotlivé časy, které odpovídají postupu při paralelizaci implementovaných algoritmů. Dosažené časy jsou také zobrazeny v grafu na obrázku 11.1.

Z průběhu zrychlování časů po jednotlivých krocích je jasné, že nejvhodnější pro zrychlení paralelizací jsou diskrétní varianty algoritmů, neboť lze paralelní zpracování a SSE sadu využít při vzorkování nové populace, což je společně s výpočtem fitness funkce nejnáročnější část diskrétního algoritmu. U těchto algoritmů bylo dosaženo zrychlení 4,89 a 7,15.

Spojitá varianta algoritmu byla také celkově zrychlena, ale ne tak výrazně, jako diskrétní implementace. Tento rozdíl pramení z relativní jednoduchosti algoritmu, který využívá spojitosti zkoumané funkce a tím pádem si velmi zjednodušuje jak výpočet stochastického modelu, tak i vzorkování nových jedinců. Hlavní části zrychlující tuto variantu algoritmu jsou výpočet fitness a pseudonáhodný generátor čísel, které jsou akcelerovány pomocí SSE sady. Celkové bylo dosaženo zrychlení 2,17.

Pro lepší přehlednost byla v tabulce použita následující pojmenování:

T - základní doba běhu programu

T_r - doba běhu programu po přidání SSE pseudonáhodného generátoru čísel

T_f - doba běhu programu po implementaci fitness funkce pomocí sady SSE

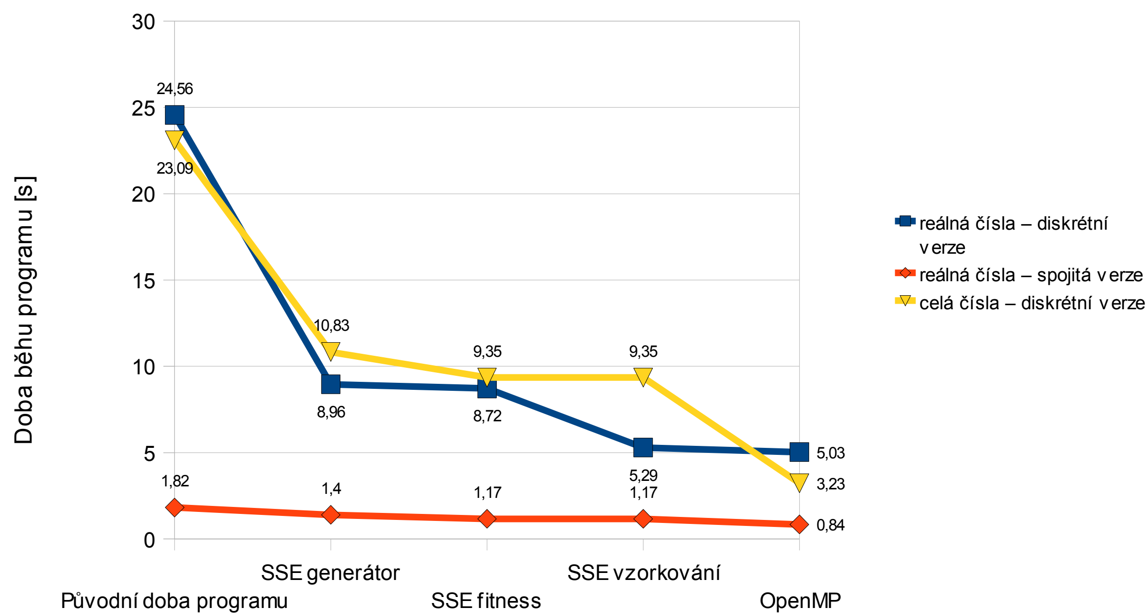
T_v - doba běhu programu po implementaci vzorkovací funkce pomocí sady SSE (implementováno pouze u diskrétního algoritmu s reálnými čísly)

T_o - doba běhu programu po implementaci systému OpenMP

S - celkové dosažené zrychlení algoritmu

11.1: Přehled vývoje doby běhu programu po jednotlivých krocích a dosažené zrychlení

Verze algoritmu	T [s]	T_r [s]	T_f [s]	T_v [s]	T_o [s]	S
Reálná čísla – diskrétní	24,56	8,96	8,72	5,29	5,03	4,89
Reálná čísla – spojitá	1,82	1,4	1,17	1,17	0,84	2,17
Celá čísla – diskrétní	23,09	10,83	9,35	9,35	3,23	7,15



Obr 11.1: Graf vývoje doby běhu programu po jednotlivých krocích

12 Závěr

V této práci jsem se zabýval akcelerací pokročilého genetického algoritmu [20] s využitím vícejádrových procesorů a instrukční sady SSE [27]. Práci lze shrnout v následujících krocích – nejprve uvedu postup řešení (sekce 12.1) a shrnu dosažené výsledky (sekce 12.2). Na závěr vymezím možnosti dalšího vývoje projektu (sekce 12.3).

12.1 Postup řešení

Pro implementaci byl vybrán pokročilý genetický algoritmus *UMDA* [21]. Pomocí tohoto algoritmu jsem řešil dva vybrané problémy – optimalizace parametrů Ackleyho funkce [2] a řešení problému batohu [2]. Tyto problémy byly zvoleny tak, aby byly implementovány dvě varianty algoritmu. První varianta řeší problém pomocí reálných čísel, druhá využívá celých čísel. Toto rozčlenění bylo provedeno kvůli porovnání výkonosti *SSE* sady při práci s reálnými a celými čísly. Algoritmus pracující s reálnými čísly optimalizoval spojitou funkci, proto byl poté implementován také ve dvou variantách, z nichž první pracovala na principu diskretizace spojité funkce na diskrétní model a druhá sestavovala spojitý model [21].

Poté byly implementované algoritmy profilovány [32]. Bylo zjištěno, že nejnáročnějšími částmi algoritmů jsou vzorkování nové populace, výpočet fitness a generování pseudonáhodných čísel.

Funkce pro výpočet fitness funkce a vzorování nové populace byly poté implementovány pomocí *SSE* sady.

Ke zrychlení generování pseudonáhodných čísel byla využita SSE implementace generátoru [36], kterou jsem upravil tak, aby byla přeložitelná v programu *gcc*. Tuto implementaci jsem doplnil o generování reálného čísla v rozsahu (0,1), kterou lze využít při pokusech s pravděpodobností.

Posledním krokem akcelerace algoritmů bylo využití systému *OpenMP* [10]. Implementované programy jsem upravil tak, aby jednotlivé funkce prováděly v blocích, které jsou vzájemně nezávislé. Tyto bloky jsem poté nastavil pro provádění ve více vláknech.

12.2 Dosažené výsledky

V práci jsem implementoval pokročilé genetické algoritmy, které optimalizovaly parametry Ackleyho funkce a řešily problém batohu. Tyto algoritmy jsem upravil pro využití SSE sady a systému OpenMP. Zrychlení dosažené těmito úpravami záleželo na vhodnosti algoritmu pro paralelizaci. Nejméně vhodná byla implementace optimalizace parametrů Ackleyho funkce

s využitím spojitého modelu, neboť je relativně jednoduchá. I přes toto omezení bylo dosaženo zrychlení 2,17.

Mnohem vhodnější pro paralelizaci je implementace algoritmu, který pracuje s diskrétním modelem. Při této implementaci se projevilo, že není zásadní rychlostní rozdíl mezi instrukcemi SSE sady, které pracují s celými čísly a instrukcemi, které pracují s reálnými čísly. Malým omezením je nedostupnost některých operací, jako například výpočet goniometrických funkcí. Tyto výpočty musí být prováděny sekvenčně.

Výrazné zrychlení přinesla implementace SSE pseudonáhodného generátoru. Tento generátor byl k dispozici jen pro systém Windows, proto jsem jej upravil i pro práci v operačním systému Linux. Dále jsem do tohoto generátoru přidal funkci pro generování reálného čísla v rozsahu (0,1), což je v genetických algoritmech velmi častá operace. Implementovaný generátor jsem porovnal s generátorem, který je standardně implementován v jazyce C. Implementovaný SSE generátor dosáhl zrychlení 2,7.

Celkově diskrétní varianty algoritmu dosáhly zrychlení 4,89 a 7,15. Menšího zrychlení dosáhl algoritmus, který pracoval s reálnými čísly. To je způsobeno zejména výše uvedenou absencí goniometrických funkcí a faktem, že vzorkovací funkce algoritmu nešla kvalitně paralelizovat, neboť po implementaci SSE sady bylo vzorkování nové populace relativně rychlé a paraleliazaci brzdila nutnost pracovat se sdílenou proměnnou populace.

12.3 Možnosti dalšího vývoje projektu

První možností je implementace SSE sady do genetických algoritmů, které jsou využívány v rámci výzkumných projektů, zejména se zaměřením na fitness funkci. Dalším krokem je zařazení SSE generátoru pseudonáhodných čísel, který může zrychlit ostatní části algoritmu, jako například tvorbu nové generace či výběr rodičů.

Další možností je výzkum sady AVX [29, 30, 31], která je následníkem SSE sady. Tato sada bude umožňovat ještě větší paralelizaci výpočtu, neboť bude například umožňovat současný výpočet osmi reálných čísel s jednoduchou přesností. V principu se budou pro tuto sadu využívat stejné postupy, které byly navrženy v rámci této práce.

Literatura

[1] Kvasnička, V., Pospíchal, J., Tiňo, P.: *Evolučné algoritmy*. Vydavatelství STU Bratislava, 2000, str. 215, ISBN: 80-227-1377-5

[2] Zelinka, I., Oplatková, Z., Šeda, M., Ošmera, P., Včelař, F.: *Evoluční výpočetní techniky – principy a aplikace*. Nakladatelství BEN – technická literatura, 2009, str. 534, ISBN: 978-80-7300-218-3

[3] Koza, John R.: *Genetic programming: on the proramming of computers by mean of natural selection*. Cambridge, MA: The MIT Press, 1992, str. 819, ISBN: 0-262-11170-5

[4] Beyer, Hans-Georg: *The theory of evolution strategies*. Springer, 2001, str. 355, ISBN: 3-540-67297-4

[5] Meloun, M., Militký, J.: *Kompendium statistického zpracování dat*. Vydavatelství ACADEMIA, 2002, str. 764, ISBN: 80-200-1008-4

[6] Flegr, J.: *Zamrzlá evoluce aneb je to jinak, pane Darwin*. Nakladatelství ACADEMIA, 2008, str. 326, ISBN: 978-80-200-1526-6

[7] Quinn, M. J.: *Parallel Programming in C with MPI and OpenMP*. McGraw Hill, 2004, str. 529, ISBN: 0072822562

[8] Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., Menon, R.: *Parallel programming in OpenMP*. Morgan Kaufmann, 2000, str. 230, ISBN: 1-55860-671-8

[9] *Basic functions: Ackley's function* [online]. Poslední revize 2010-05-21 [cit. 2010-05-03]. <http://www.it.lut.fi/ip/evo/functions/node14.html>

[10] *OpenMP* [online]. Poslední revize 2010-05-03 [cit. 2010-03-01]. <http://openmp.org/wp>

[11] *No Free Lunch Theorems* [online]. Poslední revize 2010-05-21 [cit. 2010-03-01]. <http://www.no-free-lunch.org>

[12] Whitley, D., *No Free Lunch: 1995 – 2009*. In proceedings of Genetic and Evolutionary Computation Conference, GECCO 2009, Montréal, Canada, 2009, Page 3057

[13] *Speech@FIT* [online]. Poslední revize 2010-05-21 [cit. 2010-03-01]. <http://speech.fit.vutbr.cz>

[14] *Centrum pro intenzivní výpočty ČVUT* [online]. Poslední revize 2010-05-21 [cit. 2010-03-10]. <http://www.civ.**cvut**.cz>

[15] *Seti@home* [online], Poslední revize 2010-05-21 [cit. 2010-03-10]. <http://setiathome.berkeley.edu>

[16] Darwin, Ch.: *O vzniku druhů přirozeným výběrem*. Nakladatelství ACADEMIA, 2007, str. 579, ISBN: 80-200-1492-6

[17] Orel., V.: *Gregor Mendel a počátky genetiky*. Nakladatelství ACADEMIA, 2003, str. 240, ISBN: 80-200-1082-3

[18] Relichová, J.: *Genetika populací*. Masarykova univerzita v Brně, 1997, str. 175, ISBN: 80-210-1542-X

[19] Dawkins, R.: *The Selfish gene*. Oxford University press, 2006, str. 360, ISBN: 0-19-929114-4

[20] Schwartz, J., Sekanina, L.: *Aplikované evoluční algoritmy* – Studijní opora pro předmět EVO, Fakulta informačních technologií, Vysoké učení technické v Brně, 2006, str. 101

[21] Očenášek, J.: *Parallel Estimation of Distribution Algorithms*. Disertační práce, Fakulta informačních technologií, Vysoké učení technické v Brně, 2001, str. 154

[22] Mitchell, M.: *An Introduction to Genetic Algorithms (Complex Adaptive Systems)*. The MIT Press, 1998, str. 221, ISBN: 0262631857

[23] Heckermann, D., Geiger, D., Chickering, M.: *Learning Bayesian networks: The combination of knowledge and statistical data*. Technical Report MSR-TR-94-09, Microsoft Research, Redmond, WA, 1994

[24] *Extending the World's Most Popular Processor Architecture* [online]. Poslední revize 2010-05-21 [cit. 2010-03-19].<ftp://download.intel.com/technology/architecture/new-instructions-paper.pdf >

[25] *Intel* [online]. Poslední revize 2010-05-21 [cit. 2010-03-19]. <http://www.intel.com>

[26] *AMD* [online]. Poslední revize 2010-05-21 [cit. 2010-03-19]. <http://www.amd.com>

[27] *Streaming SIMD Extensions* [online]. Poslední revize 2010-05-21 [cit. 2010-03-19]. <*http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions*>

[28] *3DNow! Technology manual* [online]. Poslední revize 2010-05-21 [cit. 2010-03-19]. <http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/21928.pdf >

[29] *AMD64 Architecture Programmer's Manual* [online]. Poslední revize 2010-05-21 [cit. 2010-03-19]. <http://support.amd.com/us/Processor_TechDocs/43479.pdf>

[30] *AMD to Boost Performance for Everyday Compute-Intense Multimedia, Security and High Performance Computing Applications through New Extensions to x86 Instruction Set* [online]. Poslední revize 2010-05-21 [cit. 2010-03-20]. <http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543_15008~119091,00.html>

[31] *Advanced Vector Extensions* [online]. Poslední revize 2010-05-21 [cit. 2010-03-20]. <http://en.wikipedia.org/wiki/Advanced_Vector_Extensions>

[32] *GNU gprof* [online]. Poslední revize 2010-05-21 [cit. 2010-04-08]. <http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html>

[33] *GCC Command-Line Options* [online]. Poslední revize 2010-05-21 [cit. 2010-04-08]. <http://tigcc.ticalc.org/doc/comopts.html>

[34] Entacher, K.: *A collection of classical pseudorandom number generators with linear structures – advanced version* [online]. Poslední revize 2010-05-21 [cit. 2010-04-10]. <http://crypto.mat.sbg.ac.at/results/karl/server/server.html>

[35] *RANDU* [online]. Poslední revize 2010-05-21 [cit. 2010-04-10]. <http://en.wikipedia.org/wiki/RANDU >

[36] Owens, K., Parikh, R.: *Fast Random Generator on the Intel® Pentium® 4 Processor* [online]. Poslední revize 2010-05-21 [cit. 2010-02-01]. <http://software.intel.com/en-us/articles/fast-random-number-generator-on-the-intel-pentiumr-4-processor/ >

[37] Amdahl, G., M.: *Validity of the single-processor approach to achieving large scale computing capablites*. In AFIPS Conference Proceedings, volume 30, Atlantic City, New Jersey, USA, 1967. AFIPS. Str. 483 – 485

[38] Zhang, T., Kang, T., Lipsky, L.: *On The Performance of Parallel Computers: Order Statistics and Amdahl's Law* [online]. Poslední revize 2010-05-21 [cit. 2010-04-20]. <http://www.eng2.uconn.edu/~lester/papers/amdl.pdf>

[39] Gustafson, J., L.: *Reevaluating Amdahl's Law* [online]. Poslední revize 2010-05- 21 [cit. 2010-05-03]. <http://mprc.pku.edu.cn/courses/architecture/autumn2005/reevaluating-Amdahls-law.pdf >

[40] Flynn, M.: *Some Computer Organizations and Their Effectiveness*, IEEE Trans. Comput., Vol. C-21, pp. 948, 1972.

[41] Lasoň, M.: *Vektorové počítače* [online]. Poslední revize 2010-05-21 [cit. 2010-04-20]. <http://homel.vsb.cz/~las03/pa/index.html>

[42] Jaja, J.: *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992, ISBN: 0-201-54856-9

[43] *GNU General Public License* [online]. Poslední revize 2010-05-21 [cit. 2010-04-20]. <http://www.gnu.org/licenses/gpl.html>

Seznam příloh

Příloha 1. CD