

VYSOKÉUČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA STROJNÍHO INŽENÝRSTVÍ
ÚSTAV AUTOMATIZACE A INFORMATIKY

FACULTY OF MECHANICAL ENGINEERING
INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

DISTRIBUOVANÉ VÝPOČTY S VYUŽITÍM TECHNOLOGIE ACTIONSCRIPT

TITLE

DISTRIBUTED COMPUTING BY FORCE OF ACTIONSCRIPT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. Petr Minář

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. Radomil Matoušek, Ph.D.

BRNO 2009



ZADÁNÍ ZÁVĚREČNÉ PRÁCE

Vysoké učení technické v Brně, Fakulta strojního inženýrství

Ústav automatizace a informatiky

Akademický rok: 2008/2009

ZADÁNÍ DIPLOMOVÉ PRÁCE

student(ka): Bc. Petr Minář

který/která studuje v **magisterském navazujícím studijním programu**

obor: **Aplikovaná informatika a řízení (3902T001)**

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma diplomové práce:

Distribučované výpočty s využitím technologie ActionScript

v anglickém jazyce:

Distributed Computing by force of ActionScript

Stručná charakteristika problematiky úkolu:

Daná DP se bude zabývat vývojem Flash aplikace demonstrující paralelní zpracování řešené úlohy. V kontextu optimalizace bude implementován paralelní optimalizační algoritmus HC12. Aplikace bude zahrnovat Master/Slave rozhraní s databázovou konektivitou. Výsledek řešení bude demonstrovat „možnosti“ paralelního zpracování i implementovaného algoritmu HC12 na zadaných optimalizačních problémech. Výsledky budou vyhodnoceny na základě statistiky. DP bude součástí řešení VZ Inteligentní systémy v automatizaci.

Cíle diplomové práce:

- Popis zadaných testovacích úloh.
- Praktická Master/Slave aplikace.
- Vyhodnocení výsledků.
- Vytvořit adekvátní uživatelskou dokumentaci.

Seznam odborné literatury:

- Matoušek, R.: Disertační práce (obor technická kybernetika) – Vybrané metody umělé inteligence – implementace a aplikace, VUT Brno, 2004

Vedoucí diplomové práce: Ing. Radomil Matoušek, Ph.D.

Termín odevzdání diplomové práce je stanoven časovým plánem akademického roku 2008/2009.
V Brně, dne 28.11.2008

L.S.

doc. RNDr. Ing. Miloš Šeda, Ph.D.
Ředitel ústavu

doc. RNDr. Miroslav Doupovec, CSc.
Děkan fakulty



ABSTRAKT

Tato diplomová práce se zabývá problematikou návrhu optimalizačního software v jazyku ActionScript 3.0. Navržené aplikace realizují samostatné i distribuované výpočty pomocí implementované optimalizační heuristiky HC12. Pro ověření funkcionality a výkonu byla vybrána sada testovacích úloh. Praktickým výsledkem práce je vytvoření veřejně přístupných klientů v síti internet a zpracovávání dílčích výpočtů s nimi. Klienti jsou trojího druhu dle pravomocí a úloh na nich spočívajících.

ABSTRACT

This Master thesis deals with designing and realization of optimization software in ActionScript 3.0 background. The designed applications realized in terms of computation by standalone and distributed variant and by means of chosen heuristic method called HC12. For verification of a function and performance was select set of testing examples. The main purpose of the work was create public clients in the network internet and calculated partial computing by it. The completed clients are selected by competence and carry on tasks.

Práce vznikla při řešení výzkumného záměru *Intelligentní systémy v automatizaci* podporovaného MŠMT ČR pod registračním číslem MSM 0021630529.

This research has been supported by the Czech Ministry of Education in the frame of MSM 0021630529 Research Intention Intelligent Systems in Automation.

KLÍČOVÁ SLOVA

Distribuované výpočty, optimalizace, HC12, APC, ActionScript.

KEYWORDS

Distributed computing, optimization, HC12, APC, ActionScript.



PODĚKOVÁNÍ

Nemalé poděkování za pomoc a podmětné připomínky patří vedoucímu práce Ing. Radomilovi Matouškovi, Ph.D. Také bych rád poděkoval všem blízkým a přátelům, kteří mi svojí tolerancí vytvořili vhodné prostředí při psaní této závěrečné práce.

OBSAH

Zadání závěrečné práce	ii
Abstrakt	v
Klíčová slova	v
Poděkování	vi
Obsah	viii
Úvod	xi

KAPITOLA 1

OPTIMALIZACE

13

1.1	○ STAVOVÝ PROSTOR	13
1.2	HOROLEZECKÝ ALGORITMUS	13
	HC s VYUŽITÍM BINÁRNÍHO KÓDOVÁNÍ.....	14
	○ BINÁRNÍ MASKA	15
	○ GRAY KÓD	15
	○ PRINCIP ALGORITMU	16
	○ OHODNOCENÍ BINÁRNÍHO ŘETĚZCE.....	17
	○ APC.....	17
	• Zvýšení vzorkování.....	17
	• Omezení rozsahu prohledávání	18

KAPITOLA 2

VÝPOČTOVÉ PŘÍKLADY V DIPLOMOVÉ PRÁCI

21

2.1	ZÁTĚŽOVÝ TEST	21
	○ ALGORITMUS PRO DISTRIBUCI	21
	○ PŘÍKLAD PŘEPOČTU INTERVALU POMOCÍ RYCHLOSTI PC	22
	○ TEST RYCHLOSTI VÝPOČTU	22
2.2	APLIKACE HC PRO OPTIMALIZAČNÍ ÚLOHY.....	23
	○ OBECNÝ ZPŮSOB DISTRIBUCE VÝPOČTU	23
	○ INICIALIZAČNÍ DATA.....	25
	○ XML – SETTINGS	25
2.3	FUNKČNÍ OPTIMALIZACE	27
	○ APLIKACE ÚLOHY FUNKČNÍ OPTIMALIZACE V HC	27
	○ ZPŮSOB DISTRIBUCE VÝPOČTU	30
	○ PARAMETRY DLE XML SOUBORU.	30
2.4	PROBLÉM BATOHU	32
	○ DEFINICE PROBLÉMU	32
	○ MOŽNOSTI APLIKACE PROBLÉMU	33
	○ APLIKACE PROBLÉMU BATOHU V HC	33
	○ ZPŮSOB DISTRIBUCE VÝPOČTU	34
	○ PARAMETRY DLE XML SOUBORU	35
2.5	PROBLÉM OBCHODNÍHO CESTUJÍCÍHO	37
	○ APLIKACE PROBLÉMU TSP V HC.....	38
	○ ZPŮSOB DISTRIBUCE VÝPOČTU	38
	○ PARAMETRY DLE XML SOUBORU	40

KAPITOLA 3 DISTRIBUOVANÉ VÝPOČTY 41

3.1	ÚVOD DO PROBLEMATIKY	41
	○ DISTRIBUCE POMOCÍ SAMOSTATNÝCH KLIENTŮ	41
	• Projekt Folding@Home.....	41
	○ OBECNÝ PRINCIP ROZDĚLENÍ ÚLOH <i>DC</i> SYSTÉMU	43
	○ DISTRIBUCE V RÁMCI JEDNOHO PROGRAMU	43
	○ HARDWAROVÁ NÁROČNOST <i>DC</i> VÝPOČTŮ	44
3.2	APLIKOVÁNÍ <i>DC</i> V PRÁCI.....	45
	○ SAMOSTATNÝ KLIENT	45
	• Skripty pro CMD line	47
	○ SLAVE KLIENT	48
	• Podrobný popis Slave klienta	48
	○ MASTER KLIENT	50
	• Podrobný popis Master klienta.....	50
	• Struktura databáze.....	52

KAPITOLA 4 POUŽITÉ TECHNOLOGIE 53

4.1	ACTIONSCRIPT	53
	○ AS VERZE 3.0	53
	○ ROZHRANÍ <i>AIR</i>	54
4.2	PHP	55
4.3	XML.....	56
4.4	SQL.....	57
4.5	WAMP.....	58

KAPITOLA 5 PRAKTICKÉ TESTY 59

	○ TESTOVACÍ FUNKCE	59
5.1	TEST 1: SAMOSTATNÝ KLIENT PŘI RŮZNÉM ZPŮSOBU NASTAVENÍ	60
5.2	TEST 2: APLIKOVÁNÍ RŮZNÝCH TYPŮ VÝPOČTU.....	61
5.3	TEST 3: SCHOPNOST NALÉZT ŘEŠENÍ	62
5.4	TEST 4: RYCHLOST <i>DC</i> SYSTÉMU DLE MASKY	64
5.5	TEST 5: RYCHLOST <i>DC</i> SYSTÉMU DLE INDIVIDUÁLNÍHO ROZDĚLENÍ	65

Závěr	67
Seznam použité literatury	69
Seznam obrázků.....	71
Příloha.....	72

ÚVOD

Cílem této diplomové práce byl návrh softwarových klientů realizujících jak samostatné, tak distribuované optimalizační výpočty. Jako optimalizačního algoritmu byla při realizaci klientů využita varianta heuristické metody *HC*, dále označená jako *HC12*. Součástí práce bylo seznámení s distribuovanými výpočty, které jsou ve světě běžně používány, jako jsou například projekty *Bionic* a *Home*. Hlavní příklady řešené v diplomové práci jsou z oblasti funkční a kombinatorické optimalizace. Tyto příklady slouží k ověření funkcionality a dalšího použití navržených aplikací v oblasti DC výpočtů.

Teoretická část je rozdělena do několika kapitol, hlavní část teorie je obsažena v kapitole 2, další části jsou začleněny jako úvod do každé kapitoly. Kapitola 2 stručně popisuje optimalizaci a použitou heuristickou metodu spolu s rozšířením o popis principu adaptivního kódování a způsoby kódování generující masky. Kapitola 3 obsahuje konkrétní výpočtové příklady včetně s nastavení úlohy a nastínění použitých algoritmů k distribuci výpočtů. Kapitola 4 prezentuje stručný přehled v současnosti používaných technologií DC systémů ve světě. Poslední teoretická kapitola obsahuje souhrn používaných technologií a softwarových prostředků využitých při tvorbě této práce.

Praktická část představuje vlastní práci autora. V této části byla řešena otázka realizace klientů. Výslední klienti jsou dvojího druhu, spustitelné na *PC* pomocí instalačního souboru nebo přímo ze serveru projektu prostřednictvím internetového prohlížeče.

Při tvorbě klientů byla využita technologie Adobe Flash (případně Adobe AIR) založená na programovacím jazyku ActionScript 3.0. Tato technologie byla zvolena z důvodu možnosti velkého rozšíření na veřejných sítích a dobrým vlastnostem vzhledem k programování robustních internetových aplikací. Pro zachování bezpečnosti technologie Adobe Flash je nutné při komunikaci s DB systémy využít „prostředníka“, kterým byl v tomto případě PHP.

Datová struktura v programech je uložena pomocí strukturovaného souboru XML. Tato forma byla volena kvůli jednotnému programovému přístupu všech klientů, inicializaci proměnných a možnosti kdykoli přerušovat a obnovovat samotné výpočty. Další výhodou XML souboru je také přehledné uložení dat v *DB*.

Samotné distribuované výpočty jsou prováděny prostřednictvím dvou klientů. První klient (Master) ovládá databázový systém, aplikuje algoritmy pro dělení úloh a vypočtené úlohy zpětně zpracovává tak, aby se daly využít k dalšímu zpracování. Druhý klient (Slave) je výpočtová aplikace DC systému s možností správy uživatelského účtu. Posledním jmenovaným klientem vzniklým jako praktická aplikace této diplomové práce je ne-DC orientován a může být provozován samostatně jako jednouživatelská aplikace.

KAPITOLA 1 OPTIMALIZACE

Optimalizace je proces, při kterém se snažíme dosáhnout zlepšení daného problému pomocí procesu úpravy proměnných v matematickém problému (experimentu) k nalezení definovaného extrému výstupu (minimum, maximum). K řešení tohoto procesu jsou v praxi používány různé metody a algoritmy, které se od sebe liší svými vlastnostmi, jako jsou vhodnost použití, časová náročnost výpočtu, dosahovaná přesnost řešení atd. Problém všech těchto procesů po dosažení výsledku je nemožnost s určitostí říci, jestli existuje jen jediný výsledek a jestli je tento výsledek také ten nejlepší možný. Za předpokladu používání slova „nejlepšího“ předpokládáme, že daný problém má víc než jen jedno řešení a tato řešení si nejsou navzájem rovná. Samotná definice nejlepšího možného řešení je relativní k danému problému, proto je akceptovatelná tolerance v hodnotách výsledků. Výběr vhodné metody závisí na konkrétním druhu problému a vlastnostech, které chceme dosáhnout [12].

Charakteristický problém je definován prostřednictvím účelové funkce, požadavkem optimalizace (typ extrému), okrajovými podmínkami pro vstupy do problému, popřípadě dalšími speciálními podmínkami (např. u problému batohu váha věcí nesmí překročit nosnost batohu). Samotná účelová funkce většinou obsahuje více parametrů, u kterých se pomocí úprav snažíme dosáhnout co nejlepšího optima. Omezující podmínky udávají parametrům přesný rozsah, které mohou nabývat. Bývají zadány buď pomocí rozsahu hodnot, nebo v dodatkových funkcích [2].

Mnoho optimalizačních procesů pracuje většinou s diskretním tvarem funkce, proto je nutné před samotným výpočtem požadovanou funkci převést ze spojitého tvaru na vhodný tvar (tzv. vzorkováním). Postup vzorkování spočívá v rozsekání stavového prostoru na dílce (v binárním prostoru je velikost 2^n bitů). Volba velikosti n zásadně ovlivňuje následující výpočet problému, např. při zvolení malé hodnoty bude výsledná hustota příliš řídká k pokrytí všech možných výsledů a lehce se může stát, že nejlepší výsledek bude úplně ztracen. Opačně při volbě příliš velkého n bude výpočet přesný, ale zase neúměrně náročný. Samotné vzorkování se provádí na problému dle omezujících podmínek [2].

○ Stavový prostor

V informační technice je stavový prostor považován za výpočetní model, který obsahuje diskrétní stavy [18]. Konečná množina diskrétních stavů obsahuje všechny stavy konkrétního problému, pro prohledávání tohoto prostoru jsou využívány dvě základní metody:

- **Neinformované metody:** příkladem může být prohledávání do hloubky, do šířky, nevýhoda těchto algoritmů je vyhledávání „hrubou silou“, tedy nejsou příliš šetrné na výpočetní čas.
- **Informované metody:** vybírají bod k expanzi podle různých druhů heuristik (pro každý problém unikátní typ), heuristika je postup při vyhodnocování konkrétních bodů a řídí se tzv. hodnotící funkcí. Kvalita informované metody je pevně spojená s vhodnou volbou ohodnocení bodů, při špatné volbě se nevybírají nejlepší uzly a metoda nedosahuje kýmžných výsledků.

1.1 HOROLEZECKÝ ALGORITMUS

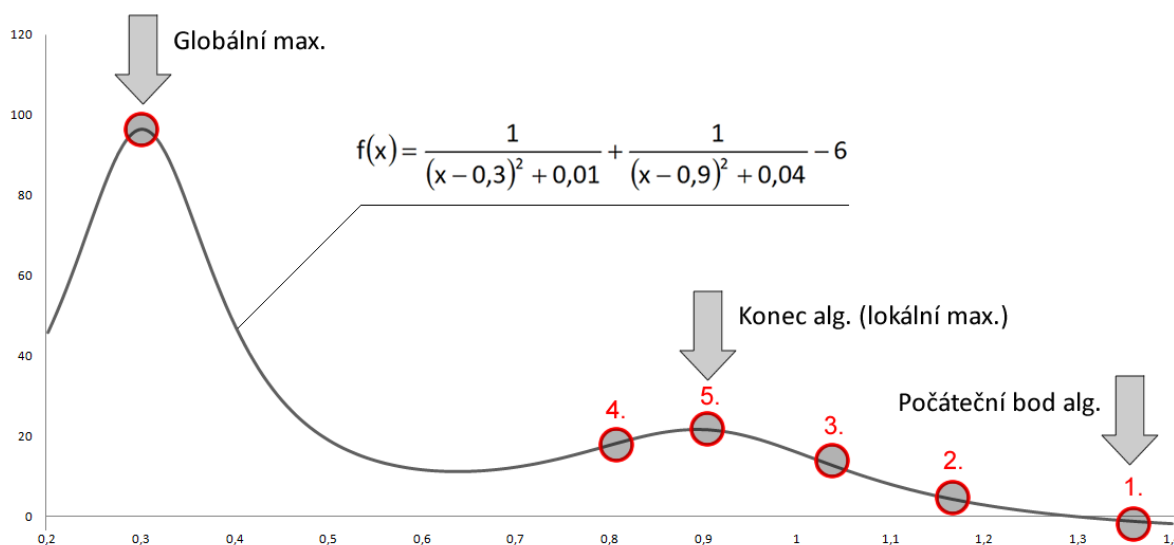
K výpočtu optimalizačních problémů v této diplomové práci byl určen tzv. horolezecký algoritmus (*hill-climbing algorithm* zkráceně *HC*). Tento algoritmus patří do skupiny informovaných metod prohledávání a pod oblast gradientních algoritmů (gradientní algoritmus vždy expanduje ten bod, který byl doposud vyhodnocen pomocí hodnotící funkce jako nejlepší, a vyhodnocuje jeho následovníky [18]). Hlavní výhodou *HC* je lehká implementace a rychlé nalezení výsledku. Princip

algoritmu spočívá v systematickém prohledávání stavového prostoru, první krok je volba inicializačního bodu (většinou volen náhodně), ke kterému se vygeneruje oblast nových bodů (body se generují podle specifického vzorce). V těchto nových bodech se zkoumá, jestli došlo ke zlepšení optimalizačního problému, když je zvolen nový výchozí bod k expanzi, tak se celý proces opakuje, rodič i sourozenci tohoto bodu jsou zapomenuti až do nalezení výsledku. Vhodný příklad využití horolezeckého algoritmu je problém obchodního cestujícího. Kde algoritmus začne ne zcela optimálním řešením a pomocí malých změn se dostává k lepším výsledkům. Ke konci výpočtu algoritmus může vést několik podmínek, ale existují dvě základní [2].

- **Konec 1:** není nalezeno v novém kroku lepší řešení než v předchozím.
- **Konec 2:** omezení iterací pro výpočet algoritmu.

Princip algoritmu je znázorněn na obr. 1 pro funkci $f(x)$ v rozsahu $\langle 0.2, 1.4 \rangle$ s globálním maximem přibližně v bode 0.32.

Hlavní problém horolezeckého algoritmu je, jako u všech gradientních algoritmů, možnost zaseknutí v lokálním extrému. Algoritmus prohledává jen svoje nejbližší okolí, a tedy může nastat případ, kdy algoritmus začne pracovat z bodu umístěného před lokálním extrémem. V tomto případě se body generované z nejbližšího okolí dostanou jen k tomuto extrému, místo za extrémem je vyhodnoceno jako body s horším optimálním řešením. Jak je naznačeno na obr. 1, výsledek bude lokální extrém ale ne globální. Určité řešení tohoto problému je správná volba způsobu generování nových bodů a zvětšení šířky generování nových bodů v okolí. Toto řešení vede mimo jiné ke zvýšení kombinatorické a časové náročnosti algoritmu [3].



OBR. 1: Princip horolezeckého algoritmu

1.2 HC S VYUŽITÍM BINÁRNÍHO KÓDOVÁNÍ

Do praktické části diplomové práce (konkrétně pro výpočet extrému funkce, problému batohu a problému obchodního cestujícího), byla zahrnuta varianta algoritmu HC s binárním generováním nových bodů pro masky až HC1n. Algoritmus patří mezi horolezecké případně heuristické metody prohledávání. Je založen na binárním kódování popřípadě na kódování Gray(ovo).

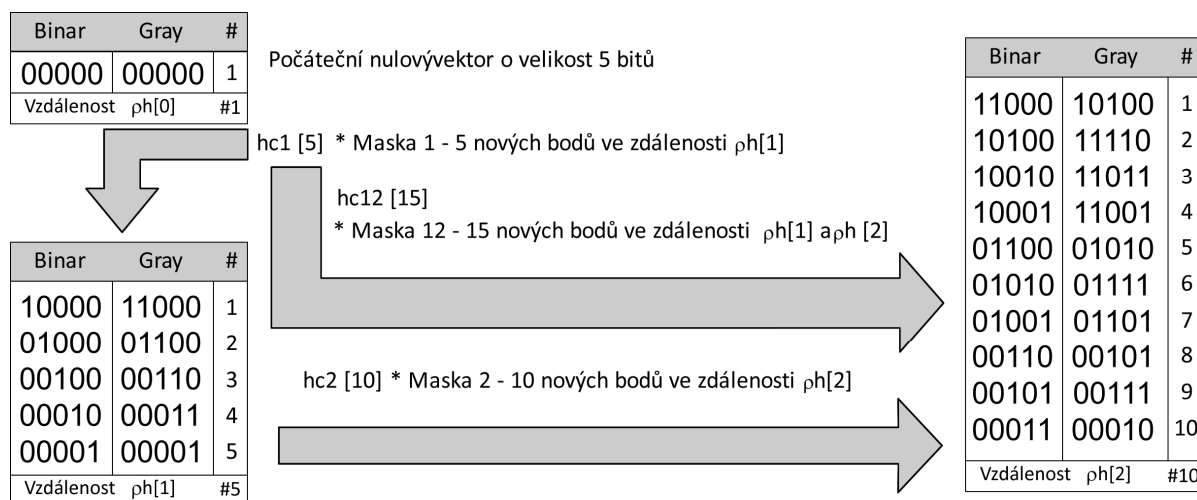
Princip		
Název	G	Vzdálenost
HC1	ph[1]	od původního řešení
HC2	ph[2]	
HC12	ph[1] až ph[2]	
HC1n	ph[1] až ph[n]	
B: generování nových bodů ve vzdálenosti		

V principu je algoritmus *HC* velmi jednoduchý, protože obsahuje jen jednu metodu a to metodu pro generování okolí z konkrétního bodu. Samotná hodnotící funkce ke generovaným bodům je pak obsažena až v konkrétních případech.

OBR. 2: Typy HC podle masky [3].

○ Binární maska

Generování nových bodů se provádí pomocí tzv. masky. Maska se aplikuje na konkrétní binární řetězec a vytvoří nové body v nejbližším okolí. Vzdálenost bodů od sebe závisí na velikosti zvolené masky a konkrétního způsobu kódování. Každý *level* masky přidá jeden invertující bit až do absolutní velikosti binárního řetězce. Způsob generování je znázorněn na obr. 2 pro masku velikosti „12“.



OBR. 3: Maska algoritmu HC12 s velikostí 5 bitů

○ Gray kód

Gray(ovo) kódování je specifické svou vlastností, kdy se mění ve slově jen jeden bit pro změnu mezi dvěma sousedícími slovy. Tato vlastnost byla použita poprvé v roce 1875 v telegrafní technice jejím vynálezcem Émile Baudotem, kvůli zmenšení předávaných zpráv mezi stanicemi. Postup generování nových slov se dá rozdělit na dva algoritmy:

- **Algoritmus 1:** pomocí přímého generování z binárního kódu.
- **Algoritmus 2:** využívání symetrie v *Gray(ovém)* kódu.

Algoritmus 1

- Dvojková číslice přímého binárního kódu s nejvyšší vahou se ponechá beze změny [1].
- Každá následující dvojková číslice přímého binárního kódu se invertuje, když ji v přímém kódu předchází na vyšší váze jednička [1].

Dec	Tabulka Bin [4bit]	Gray [4bit]	Příklad
0	0000	0000	bit s nejvyšší vahou Invertující znak (1) Neinvertující znak (0) Poslední znak (neinvertující) 1101 → 1011 Bin -- Gray bit s nejvyšší vahou se nemění invertovaný znak (předcází 1) invertovaný znak (předcází 1) ne invertovaný znak (předcází 0)
1	0001	0001	
2	0010	0011	
3	0011	0010	
4	0100	0110	
5	0101	0111	
6	0110	0101	
7	0111	0100	
8	1000	1100	
9	1001	1101	
10	1010	1111	
11	1011	1110	
12	1100	1010	
13	1101	1011	
14	1110	1001	
15	1111	1000	

OBR. 4: Převod binárního kódování na Gray(ovo)

Algoritmus 2

Při tvorbě n -ciferného Gray(ova) kódu je používán střídavě prefix „0“ a „1“, a zrcadlení již vytvořeného kódu, jak je patrné z obr. 5. Postup algoritmu se dá rozdělit do čtyř kroků.

- **Krok 1:** provést jednoduchý převod na Gray(ův) kód pro jeden bit.
- **Krok 2:** vytvořit obraz existujícího kódu, a zapsat tento kód v opačném pořadí než původní hodnoty, pod předešlý kód.
- **Krok 3:** přidat prefix „0“ k původním hodnotám a pro hodnoty obrazu použít prefix „1“.
- **Krok 4:** opakovat kroky (2) a (3) dokud nedosáhneme požadované přesnosti Gray(ova) kódu.

Přesnost	Start	Zrcadlení	Princip				
			Prefix	Zrcadlení	Prefix	Zrcadlení	Prefix
1 [bit]	(2) →	0	00	00	000	000	0000
		1	01	01	001	001	0001
		1	11	11	011	011	0011
2 [bit]	(4) →	0	10	10	010	010	0010
				10	110	110	0110
				11	111	111	0111
				01	101	101	0101
3 [bit]	(8) →			00	100	100	0100
					100	100	1100
					101	101	1101
					111	111	1111
					110	110	1110
4 [bit]	(16) →				010	010	1010
					011	011	1011
					001	001	1001
					000	000	1000

OBR. 5: Zrcadlové techniky v Gray(ovém) kódu [1]

○ Princip algoritmu

HC algoritmus s využitím binárního kódování má podobný princip jak původní gradientní HC. Hlavní předností využití binárního kódu (Gray(ova)) je větší variabilita generovaných bodů dle masky a tím možnost překonání lokálních extrémů. Algoritmus HC vytváří matice nových bodů, které jsou relativně velké a tím snižuje rychlost prohledávání matice. Tento problém je patrný hlavně u řetězce s mnoha bity na parametr. Pro příklad, kdy se použije pětice parametrů s 64 bity, se musí vytvořit

matice o velikosti 51360 x 320 pro každé generované okolí. Proto je lepší použít nižší počet bitů na parametr pro získání výsledku a pak využít nějaký z APC algoritmů pro zpřesnění hodnoty optimální funkce až v posledním bodě ve vektoru řešení. Postup algoritmu je následující.

- **Krok 1:** inicializace algoritmu se provádí dvěma způsoby a) hodnotou binárního řetězce zadaného uživatelem, b) řetězcem náhodně vygenerovaným.
- **Krok 2:** ohodnocení řetězce dle jednotlivých problémů.
- **Krok 3:** vygenerování nejbližšího okolí z konkrétního bodu, uložení všech generovaných bodů do matice nových bodů.
- **Krok 4:** ohodnocení všech binárních řetězců v matici nových bodů.
- **Krok 5:** výběr nejlepšího bodu z matice nových bodů z hlediska hledané optimalizace. Př. pro hledání minima bod s nejmenší hodnotou účelové funkce.
- **Krok 6:** vybrané řešení (je to z kroku 4) se uloží do vektoru výsledných hodnot a dále se pokračuje krokem 3.
- **Krok 7:** algoritmus končí, když krok 4 nedokáže najít bod s lepším ohodnocením. Potom prohlásí poslední uložený bod ve vektoru řešení jako výsledek optimalizační úlohy.

○ Ohodnocení binárního řetězce

Binární řetězec se ohodnocuje individuálně dle každé úlohy zvlášť. Jedno ale všechny úlohy mají společné, a to dělení řetězce na zadané parametry a posléze z těchto parametrů výpočet námi hledané funkční hodnoty. Základní princip pro každou novou úlohu je takový, aby každý z binárních podřetězců nesl jen jedno číslo, úloha sama pak rozhodne jak s konkrétní hodnotou naložit v příkladu.

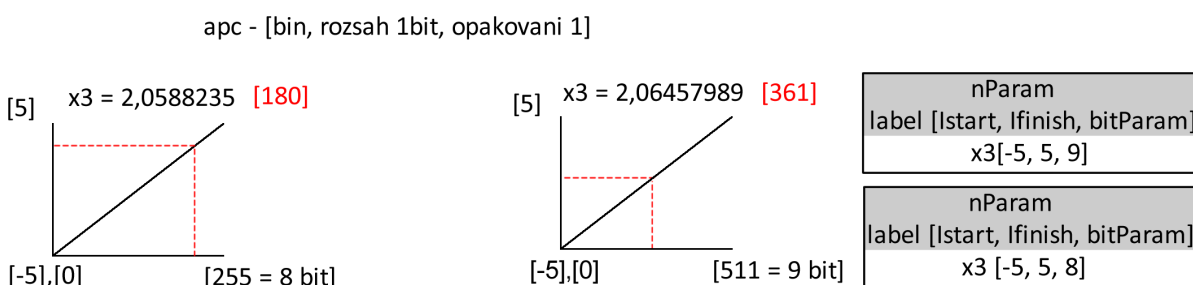
○ APC

Metoda APC (*Adaptive Parameter Coding*) slouží ke zpřesnění dosaženého výsledků pomocí úprav některých základních vlastností parametru. Algoritmus HC při použití APC dosahuje mnohem lepšího času pro nalezení přesného výsledku než při použití již od začátku jemného vzorkování problému. Způsoby aplikace se dají rozdělit na tři druhy dle lineární transformace parametru.

- Zvýšení vzorkování problému.
- Omezení rozsahu prohledávání.
- Kombinace dvou předchozích.

• Zvýšení vzorkování

První možnost, jak aplikovat algoritmus APC do výpočtu, je změna rozsahu pomocí vzorkování, v tomto případě změna velikosti binárního řetězce pro zvolený parametr, viz. obr. 6. V následujícím příkladě je naznačen postup pro změnu parametru x_3 s 8 bity (kombinatorická náročnost 256) na 9 bitů (kombinatorická náročnost 512).



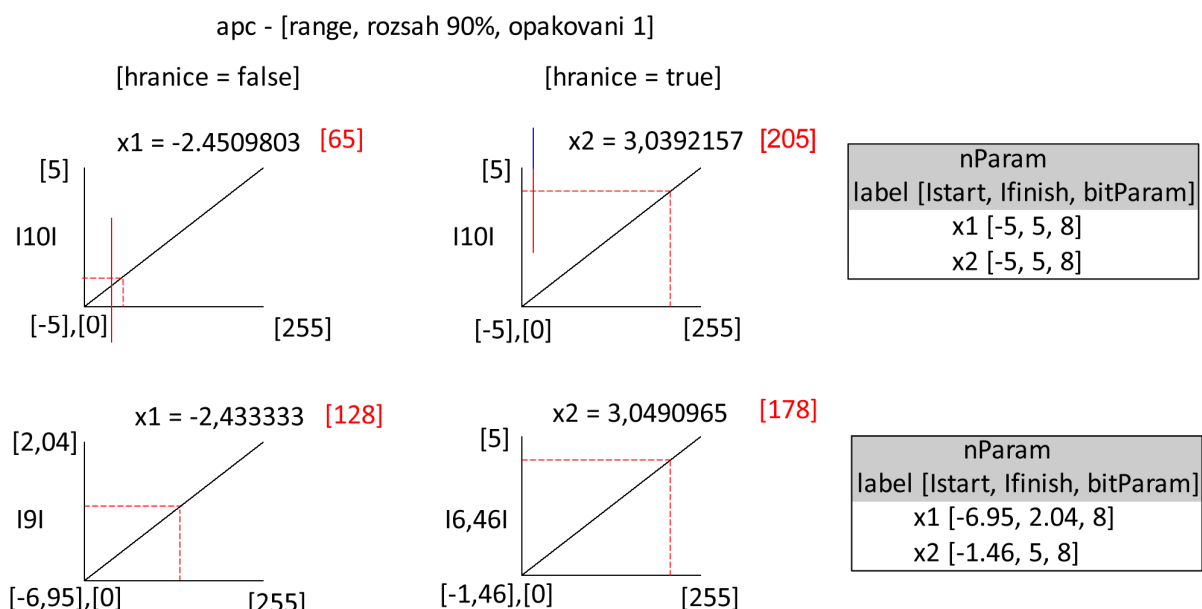
OBR. 6: Metoda zvýšení vzorkování u algoritmu APC

• Omezení rozsahu prohledávání

Další možnost aplikace algoritmu APC je z hlediska prohledávaného prostoru. Zde se vychází z předpokladu, že není nutno pro zpřesňující výpočet využívat celý rozsah původní funkce, ale jen její menší část podle definovaného zmenšení. Princip zmenšení prostoru je s pomocí procent zúžení k původnímu nastavení rozsahu. Tahle část se dělí na dvě odvětví.

- **S použitím hranic původního nastavení:** kontroluje se po zpřesnění, jestli nový rozsah nepřekročí původní nastavení, jestli ano, tak se provede oseknutí té části, které překračuje.
- **Bez požití hranic:** zde se nic nekontroluje a využívá se jen procenta zpřesnění podle definice.

Na následujícím příkladě (obr. 7) je demonstrováno použití obou dvou postupů při zpřesnění rozsahu. Na proměnou x_1 je použit způsob zpřesnění o 90 % bez hlídání hranic, proto na ose x v novém zpřesnění přesahuje původní hodnotu ($-6,95 < -5$). Oproti tomu proměnná x_2 využívá hlídání hranic, osa x by přesahovala původní nastavení, proto se musí zaokrouhlit na původní hodnotu 5.



OBR. 7: Metoda zpřesnění rozsahu u algoritmu APC

Při aplikaci APC algoritmu je nutno použít zpětné lineární transformace pro nové vzorkování se starou funkční hodnotou.

Lineární transformace parametru

Pomocí lineární transformace, viz. vzorec (2.1) a (2.2), se zjišťují funkční hodnoty parametru dle vzorkování (celočíslná hodnota) a nastavených rozsahů.

$$y = kx + I_{start} \quad \left(k = \frac{I_{finish} - I_{start}}{transf} \right) \quad (2.1) \quad (2.2)$$

kde:

I_{start}	min reálného rozsahu]
I_{finish}	max reálného rozsahu]
x	bod vzorkování
$transf$	přesnost parametru (celočíslná hodnota)

Zpětná lineární transformace parametru

Zjišťuje zpětně celočíselnou hodnotu pro nové vzorkování z funkční hodnoty, provádí se vždy při aplikaci APC k zjištění *startBin* k výpočtu, viz. vzorec (2.3).

$$x = \text{round}\left(\frac{y - q}{k}\right) \quad (2.3)$$

KAPITOLA 2 VÝPOČTOVÉ PŘÍKLADY V DIPLOMOVÉ PRÁCI

Tato kapitola je věnována všem výpočtovým příkladům v diplomové práci a jejich aplikaci v samotném DC systému. Příklady se dělí na dva druhy dle principu rozdělení práce. První druh úlohy je pomocí rozdělení ve *Slave* klientech – zátěžový test, distribuce dle dělení masky a ve druhém typu se celý DC algoritmus řídí pomocí *Master* klienta – všechny příklady spadající pod *HC*.

2.1 ZÁTĚŽOVÝ TEST

První příklad pro distribuované výpočty je zátěžový test pomocí fiktivního výpočtu. Fiktivní výpočet je v tomto případě propočet jednoho velkého čísla a kontrola, jestli je nebo není prvočíslo, výpočet byl zvolen kvůli možnosti přistupovat k výpočtu z jakékoliv strany, a tedy flexibilita při distribuci výpočtů. Prvočíslo existuje v případě, že kontrolované číslo je dělitelné pouze jedničkou nebo samo sebou. Samotný výpočet je formou externího souboru *SWF*, tedy je možnost nahrát do programu jiný s podmínkou *distribuvatelnosti* pomocí jedné osy (parametru).

Specialitou zátěžového testu je postup distribuce výpočtového intervalu, kdy tuto činnost neprovádí *Master* klient ale samotní *Slave* klienti, *Master* jen založí úlohu v databázi a o nic dalšího se už nezajímá. Po skončení příkladu si *Slave* klient spočte výslednou statistiku a sám se i odmění bodovým ohodnocením pro uživatele. Tento typ přístupu k distribuci má své výhody i nevýhody.

- **Klady:** není nutno do výpočtu nijak zapojovat *Master* klienta ke kontrole práce, všechny dílčí činnosti spojené s kontrolou souslednosti, vyhodnocení statistiky a samotnou distribuci zvládne *Slave* sám.
- **Zápory:** mnohem větší zatížení databáze při komunikaci mezi klienty navzájem, algoritmus pracuje v časovém úseku každých 10 s, při kontrole ostatních klientů (kvůli bezpečnosti není přímá ale přes prostředníka ve formě *PHP/DB*), bezpečnostní riziko, kdy se *Slave* klientů dávají velké pravomoci zasahovat do *DB* (mazat, vyvážet tabulky).

○ Algoritmus pro distribuci

Princip algoritmu pro komunikaci mezi klienty zapojených do výpočtu příkladu v rámci tzv. zátěžového testu, spolu s postupem předávání rozsahu v distribuovaném intervalu, na kterém běží výpočet.

- **Krok 1:** klient po přihlášení změří rychlost počítače, na kterém je spuštěná aplikace. Způsob měření rychlosti je ve formě kolikrát za 10 s je počítač schopen ověřit číslo 389 167, jestli je prvočíslo. Hodnota rychlosti je ve formě celého čísla **[0, 10 000]**.
- **Krok 2:** první uživatel, který se zapojí do výpočtu, si stáhne kompletní interval práce, ostatní uživatelé už získávají práci zprostředkovaně od jiných klientů podle rychlosti jejich *PC*, viz. příklad přepočtu intervalu.
- **Krok 3:** klient, který požádá o práci u odpojeného klienta (důvodů špatného spojení atd.), počká určitý časový okamžik (25 s), potom odpojeného klienta manuálně odpojí od databáze a sám převezme kompletně jeho práci.
- **Krok 4:** spočítaná práce se ukládá na server každých 5 s a vytvoří se nový interval pro výpočet z rychlosti *PC* tak, aby za časový úsek 1 s bylo spočítáno 20 intervalů.
- **Krok 5:** první klient, který zjistí dokončení celé práce, provede kontrolu hotových segmentů. V případě chyb v segmentech opraví chybné segmenty (přidá chybějící intervaly k výpočtu, opravy přesahy intervalů) a případně přikáže ostatním klientům spočítat přidanou práci. Práce je ukončena, když všechny segmenty sedí se zadáním práce.

- **Krok 6:** po ukončení práce první klient vypočítá statistiky průběhu a přidá bodové ohodnocení uživatelům podle vykonané práce v procentech na intervalu.

○ **Příklad přepočtu intervalu pomocí rychlosti PC**

Výpočet se provádí v bodě 2, kdy jeden klient od druhého žádá novou práci. Předávaný jako první věc vypočítá koeficient práce, a to pomocí jednotlivých rychlostí *PC*, vzorec (3.1) a vybere ten s největším možným koeficientem.

$$kce = (1000 - speed) / 10000 \times |If - Is| \quad (3.1)$$

<i>kce</i>	koeficient práce
<i>speed</i>	rychlost <i>PC</i>
<i>I_s</i>	začátek intervalu
<i>I_f</i>	konec intervalu

Př. klient s rychlostí <115>, intervalem práce <729,12548>, koeficient práce je roven <1045, 9815>.

Po výběru uživatele pro transfer se provádí samotný přepočet nových intervalů pomocí rychlosti *PC* a update databáze. Operaci má na starosti předávající, vzorec (3.2).

$$I^* = floor \left(\frac{(If - Is)}{\left(\frac{speed^* + kce}{kce} \right)} + Is \right) \quad (3.2)$$

$$I_1 = \langle Is, I^* \rangle \quad I_2 = \langle I^* + 1, If \rangle \quad (3.3) \quad (3.4)$$

<i>I*</i>	nový konec intervalu
<i>speed*</i>	rychlost <i>PC</i> předávajícího
<i>I₁</i>	nový interval předávajícího
<i>I₂</i>	nový interval předávaného
<i>floor</i>	zaokrouhlení na nižší celou hodnotu

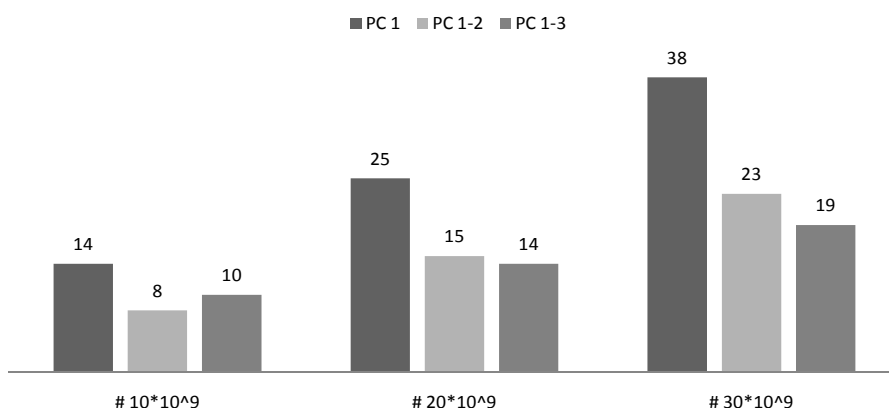
Př. dva klienti (předávaný, předávající) s koeficientem <1045, 9815> a intervalem <729, 12548>. Rychlost předávaného <55>. Nové intervaly se pak rovnají *I*₁ <729, 11957>, *I*₂ <11958, 12548>.

○ **Test rychlosti výpočtu**

Tento test zobrazuje zlepšení času při výpočtu velkého prvočísla v rozsahu $1 \cdot 10^9$ až $3 \cdot 10^9$ v závislosti na počtu samostatných různých jednotek *PC*. Jednotlivá konfigurace počítačů v testu byla následující.

- **PC 1:** Intel mobile core 2 duo T7200 (2 GHz), 4 GB DDR2 RAM.
- **PC 2:** AMD Athlon (tm) XP 2500+ (1,84 GHz), 1,5 GB DDR1 RAM.
- **PC 3:** Intel Pentium 2 (350 MHz), 512 SD RAM.

Zátěžový test



OBR. 8: Měření rychlosti výpočtu pro 1 až 3 různé PC

Při prvním výpočtu a použití jednoho PC byl výsledný čas přibližně 14 minut, po zapojení dalšího počítače (2) se výsledek zlepšil asi o 43 % na 8 min, ale při zapojení posledního (nejpomalejšího PC) bylo zlepšení času záporné než u dvou PC, a to o 29 %. Tohle zpomalení rychlosti lze přičíst ke zvýšené komunikace mezi klienty a nízké rychlosti posledního PC, který tento zápor nedokáže kompenzovat. U výpočtů větších rozsahů se zvýšená komunikace mezi klienty nehraje tak významnou roli a postupně se zvyšuje rychlost výpočtu i s PC (3).

2.2 APLIKACE HC PRO OPTIMALIZAČNÍ ÚLOHY

Stěžejní úlohou v diplomové práci je aplikování vybraných optimalizačních úloh řešených pomocí algoritmu HC s rozšířením o generování nových masek dle binárního řetězce. Vybrané úlohy jsou následující.

- **Funkční optimalizace:** výpočet extrému pro zadané funkce.
- **Problém batohu:** výpočet batohu pro n věcí s opakováním.
- **Problém obchodního cestujícího:** výpočet hodnoty cesty dle zadané sítě bodů.

Na tyto specifikované úlohy bylo nutno dle zadání práce vytvořit algoritmy ohodnocení binárního řetězce v HC a tvorbu algoritmů k distribuci výpočtů pomocí vytvořeného DB systému.

○ Obecný způsob distribuce výpočtu

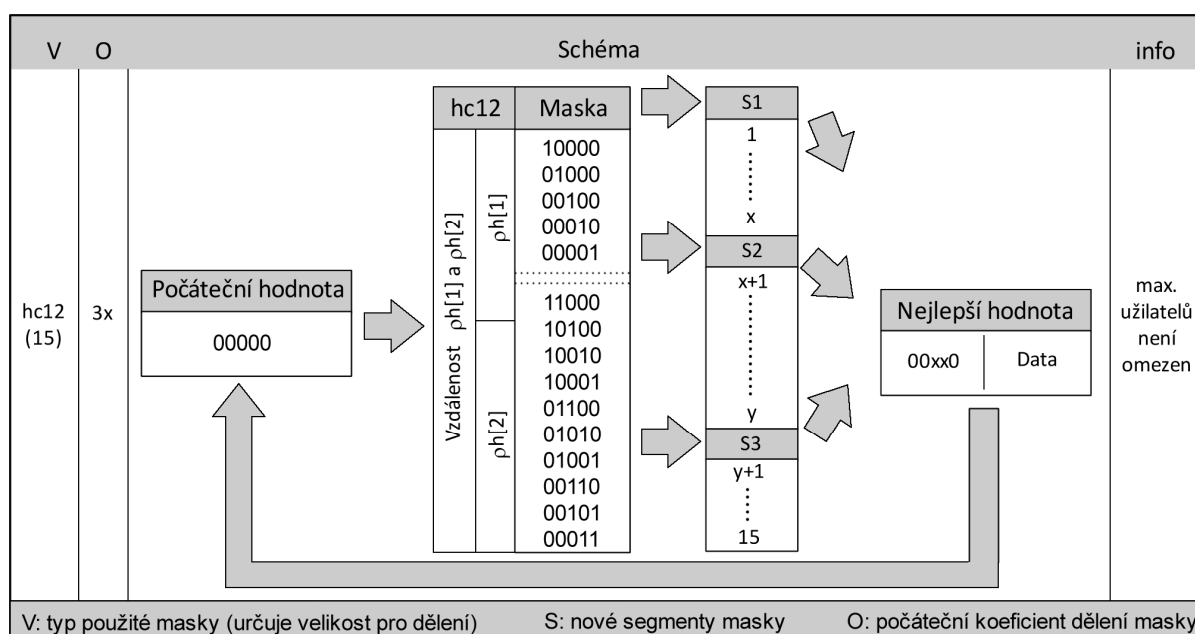
U algoritmu HC se nepracuje jen s jednou osou parametru ale obecně s n dimenzí prostoru, také je neznámý konečný počet iterací, proto nelze použít způsob distribuce z předchozího případu. Pro tento problém tedy byly vytvořeny dva rozdílné principy distribuce speciálně upravené pro účely HC. Každý z těchto principů má své klady i zápory, a proto jsou v této práci obsažené oba dva typy.

- **Princip 1:** distribuce rozsahů generované masky dle rychlosti PC.
- **Princip 2:** konstantní rozdělení stavového prostoru na určený počet dílů.

První princip je pomocí dělení generované masky na stanovené dílce v intervalu hodnot. Tento postup je nejvíce podobný předešlému příkladu (zátěžový test) a lze použít pro každý typ příkladů v HC. Hlavní nevýhoda požití této distribuce je nemožnost aplikování žádný typ APC algoritmů, kvůli změně vlastností parametrů, další zápor je v nutnosti nepřetržitého připojení k DB a komunikací mezi klienty, která je náročná na čas. Použití tohoto postupu je při výpočtů velmi rozsáhlých

generujících masek, kdy už strmě klesá rychlost chodu algoritmu. Samotný princip distribuce se dá rozdělit do několika kroků.

- **Krok 1:** *Master* klient vygeneruje XML data s počátečním nastavením problému a rozdělí masku pro první iteraci na předem definovaný počet dílů. Dále nastaví vlastní parametry distribuce, jako jsou celkový čas práce, část na výpočet jednoho dílce atd. Všechna tato data pak *Master* uloží do vygenerovaných tabulek *DB*.
- **Krok 2:** po přihlášení klientů a výpočtu všech dílů první iterace se klient s nejvyšším pořadovým číslem (*ID*) prohlásí za hlavního klienta a aplikuje princip dle obr. 9. Načte všechny spočtené hodnoty a vybere nejlepší hodnotu podle typu optimalizace (*max*, *min*), výsledná data pak uloží do tabulky výsledků a vygeneruje nové zadání iterace, ale už ne s konstantním dělením masky, ale pomocí koeficientu rychlosti *PC* z každého přihlášeného klienta, viz. vzorec (3.5). Tahle činnost se opakuje vždy po skočení iterace, až po konec výpočtu, viz. konec výpočtu *HC*.
- **Krok 3:** klient s menším *ID* než ostatní klienti se prohlásí za výpočtového klienta, pošle požadavek do *DB* o přidělení intervalu masky a počátečního binárního řetězce a čeká, až proběhne krok 2.



OBR. 9: Princip distribuce HC algoritmu pomocí dělení masky

$$I_{\text{nový}} = \frac{\text{speed}}{\text{speed}_{\text{celková}}} \cdot \text{maska} \quad (3.5)$$

kde:

<u>speed</u>	rychlost <i>PC</i>
<u>speed_{celková}</u>	celkový součet všech rychlostí <i>PC</i>
<u>maska</u>	velikost generované masky

Další možností distribuce je podle konstantního dělení stavového prostoru, u tohoto způsobu se nevyužívá dynamické dělení v průběhu výpočtu, ale *Master* klient při počátku definuje úlohu pomocí algoritmu dělení (pro každý případ unikátní viz. následující text) a vytvoří předem stanovený počet *XML* nastavení a uloží je na server plus další hodnoty (podobně jak u masky celkový čas práce, segmentu atd.). *Slave* klient si pak stáhne jen počáteční nastavení příkladu a spustí klasický výpočet. Po skončení výpočtu uloží hodnoty zpět do *DB*, kde *Master* klient tyto hodnoty zpětně zpracovává. Největší výhoda je minimální komunikace se serverem, kdy se připojení provádí jen na začátku

případu a na konci při ukládání dat. Využití principu je pro ponechání kombinatorické náročnosti původního výpočtu a převedení této přesnosti na vytvořené dílce příkladu, tedy ve výsledku zvýšení přesnosti původního zadání. Princip konstantního dělení se lze rozdělit do tří základních postupů. Kde Postup 1 a Postup 3 jsou identické pro všechny druhy příkladu, jen se mění algoritmus u Postupu 2 z hlediska dělení problému.

- **Postup 1:** prosté opakování základního výpočtu, s nastaveným počtem opakování (definuje i počet maximálně připojených uživatelů).
- **Postup 2:** algoritmus definovaný v každém z příkladů individuálně.
- **Postup 3:** kombinace postupu 1 a postupu 2, každý nový segment je vynásoben počtem opakování, maximální počet uživatelů je roven počtu segmentů vynásobeno opakováním.

○ Inicializační data

Data k načtení externích hodnot a struktura programu byla zvolena pomocí souborů ve formátu XML dat. Pracovní soubor je rozdělen na čtyři části. První část nazvaná *settings* (počáteční nastavení) je podrobné zadání aktuálního optimalizačního problému. Tato část XML souboru jde sestavit ručně pomocí jakéhokoliv textového editoru dle dokumentace z tabulky nastavení, nebo jde vygenerovat v programu po zadáních hodnot v CMD line editoru. Tento typ zadání používá i Master klient k definování zadání DC výpočtů. Další dvě části souboru už generuje klient při inicializaci úlohy. Jsou to oblast popisující podrobně všechny vlastnosti parametrů nazvaná *parameter*, oblast vlastností aktuálního řešeného problému (jméno se mění na základe typu problému) a poslední nejrozsáhlejší oblast generující při běhu výpočtu se nazývá *optCalculation*. V této části je popsán podrobně průběh celého výpočtu, uloženy všechny pomněné potřebné k obnovení již spuštěného výpočtu a výpočet statistické části výpočtu.

Skript CMD line

Načtení dat do sekce *settings* přes příkazový řádek CMD line se provádí následovně, (podrobný popis příkazů dále v textu):

```
1|      optimization{
2|          repeat[50]; // počet run(ů)
3|          find[min]; // typ hledání
4|          mask[2]; // generující maska
5|          funType[test];} // způsob ukládání dat
6|      specification{
7|          funName[problemName]; // optimální hodnota popisu
8|          iniBin[random]; // startovní binární kód
9|          iniType[random]; // způsob generování bin kódu
10|         mCode[BC]; // typ kódování
11|         funPrb[fceOpt];} // typ optimalizace
12|      dc{ // kategorie určená jen pro cmd obsažený v Master klientovy.
13|          chunks[2]; // násobení všech segmentů
14|          workTime[51]; // čas pro celkovou práci
15|          clientTime[42]; // čas pro Slave klienta
16|          priority[50]; // priorita práce
17|          clientText[info for client];} // text pro uživatele
```

○ XML – settings

První část strukturovaného datového zápisu XML je *settings* pro definování aktuálního problému. Jedná se o jedinou část, která se může definovat ručně. Příklad pro testovací funkci a podrobný popis pomněných zápisu dat je následující :

```

1|      <settings label="initial settings" param="">
2|      <funName label="funName: test fce " param=" test fce "/>
3|      <funPrb label="funPrb: fceOpt" param="fceOpt"/>
4|      <funOpt label="funOpt: min" param="min"/>
5|      <funType label="funType: quick" param="quick"/>
6|      <nParam label="nParam: x[0,10,10];y;" param="x[0,10,10];y;"/>
7|      <optFce label="optFce: F(x,y)=x*sin(4*x)+1.1*y*sin(2*y);"
      param="F(x,y)=x*sin(4*x)+1.1*y*sin(2*y);"/>
8|      <iniType label="iniType: set" param="set"/>
9|      <iniBin label="iniBin: 10001110010000100010"
      param="10001110010000100010"/>
10|      <maskLevel label="maskLevel: 2" param="2"/>
11|      <mCode label="mCode: BC" param="BC"/>
12|      <repeat label="repeat: 3" param="3"/>
13|      <apcSelect label="apcSelect: Only the best" param="best"/>
14|      <apcMode label="apcMode: x(range,10,false,1),y(bin,1,3)"
      param="x(range,10,false,2),y(bin,1,3)"/>
15|  </settings>

```

Každý uzel `<node>` v celém rozsahu XML objektu má minimálně dva základní parametry:

- **label:** používá program pro zobrazení dat v interface. Není nutno zadávat uživatelem, vygeneruje si ho sám program při inicializaci.
- **param:** hlavní parametry pro uložení proměnné v programu.

<název>	popis	formát hodnot
		příklad
funName	Uživatelem zadaný nebo programem generované jméno aktuálního počítaného problému.	Znakový řetězec o max. 255 znaků
		testovací příklad
funPrb	Selekce typu problému pro výpočet optimalizace. Možnost počítání optimální hodnoty funkce, problém batohu nebo TSP.	Znakový řetězec [fceOpt , knapsackOpt , tsp]
		fceOpt
funOpt	Výběr typu hledání globálního extrému. Možnost výběru maxima nebo minima problému.	Znakový řetězec [min. , max.]
		min
funType	Nastavení rychlosti výpočtu ke vztahu k omezení některých parametrů statistiky, jako jsou zjišťování počtu spočítaných hodnot, počet unikátních generovaných hodnot za jeden <i>run</i> a počet všech spočtených hodnot v optimalizaci.	Znakový řetězec [test , quick]
		quick
nParam	Způsob zadávání hodnot pro jednotlivé parametry výpočtu. Mění se od typu optimalizace. Všechny parametry musí být od sebe odděleny středníkem. Způsob zadávání, viz. testovací příklady.	Znakový řetězec udávající jméno parametru a tři hodnoty <rozsah> a počet bitů na parametr [real, real, integer];
		X[0,10,10];y;
optFce	Zadávání hodnot pro specifikaci optimalizačního problému. Mění se také podle typu optimalizace. Konec se definuje středníkem. Způsob zadávání viz testovací příklady.	Znakový řetězec o max 255 znaků.
		F(x,y)=x*sin(4*x)+1.1*y*sin(2*y);
iniType	Způsob zadání prvního binárního řetězce. Možnost měnit mezi náhodně generovaným nebo uživatelem definovaným. Další možnosti se mění podle aktuální úlohy.	Znakový řetězec [random , defined , set]
		set

iniBin	Binární řetězec pro zahájení výpočtu, musí být stejné velikosti jako celková přesnost parametrů.	Znakový řetězec <0,1> 10001110010000100000.
maskLevel	Velikost generující masky pro nové body. Možnost zadávání přesnosti od 1 do maximální přesnosti binárního řetězce. Pzn. čím je větší maska, tím je větší časová náročnost výpočtu.	Celé číslo <1, max. přesnost> 2
mCode	Typ kódování binárního řetězce.	Znakový řetězec [BC, Gray] BC
repeat	Celkový počet opakování optimalizačního výpočtu.	Celé číslo <1, - > 3
apcSelect	Způsob výběru <i>iRun(u)</i> pro dynamické programování. Možnost pro všechny opakování nebo vybere až poslední s nejlepší hodnotou ke vztahu funOpt.	Znakový řetězec [all, best] best
apcMode	Specifikace dynamického programování pro každý parametr. Možnost volby mezi zvětšením vzorkování nebo zpřesnění rozsahu. Každé z APC se ukončuje čárkou, viz. CMD line.	Znakový řetězec udávající jméno parametru a vlastnosti APC yx(vlastnosti); (range,10,false,2),y(bin,1,3)

2.3 FUNKČNÍ OPTIMALIZACE

První aplikace pro algoritmus *HC* byly zvoleny příklady z funkční optimalizace s využitím generované masky dle binárního kódu s velikostí 1 až n . Funkční optimalizace pracuje s matematickými funkcemi v rozsahu n dimenze prostoru, tedy pro n počet parametrů vstupujících do problému, které také udávají omezující podmínky optimalizace, samotný zápis je pomocí obecné matematické funkce. Na tento typ optimalizace lze využít zpřesnění pomocí z některého z typů algoritmu APC.

○ Aplikace úlohy funkční optimalizace v HC

Příklad aplikace je znázorněn pomocí testovací funkce $F_6(x)$ – viz. vzorec (3.6), tato funkce je speciálně vytvořena pro účely testování algoritmů k optimalizaci. Jedná se o příklad, který simuluje problém s mnoha lokálními extrémy a je sestavena v tzv. optimálním vektorovém tvaru. Na této funkci je možno testovat problémy v libovolné dimenzi prostoru řešení v rozsahu E^n . Přesné zadání problému je specifikováno dle obr. 10 [2].

$$F_6(x_i) = A \cdot n + \sum_{i=1}^n (x_i^2 - A \cdot \cos(2 \cdot \pi \cdot x_i)) \quad (3.6)$$

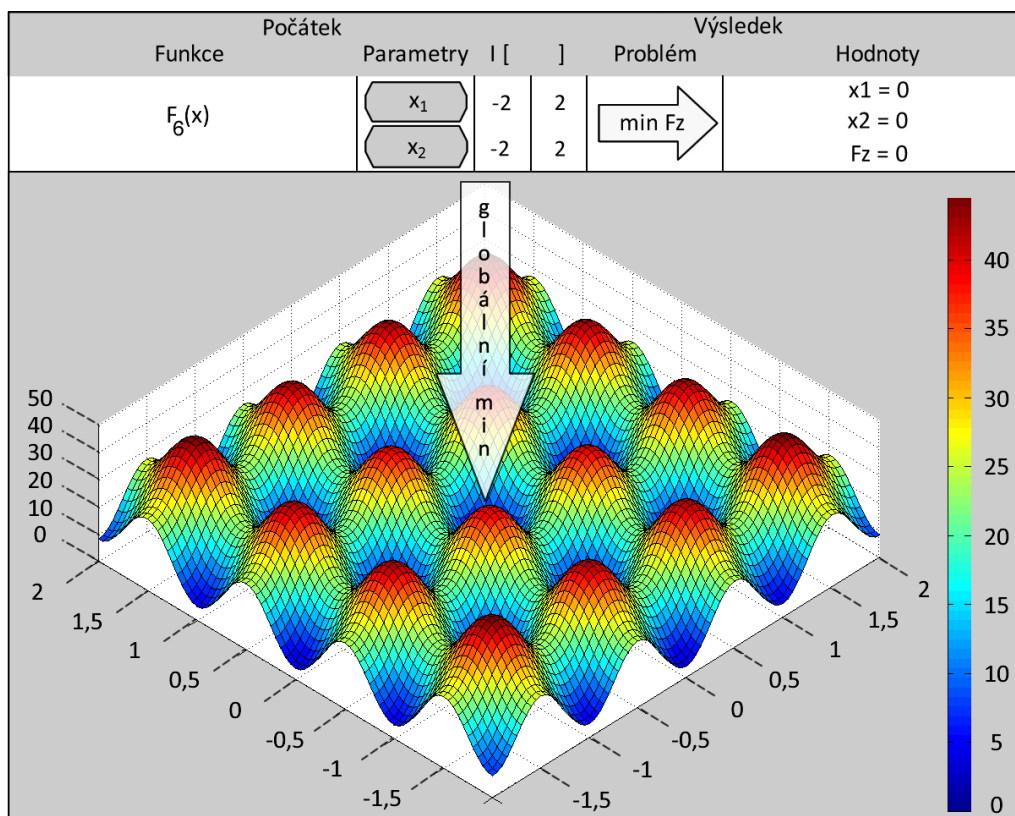
$$F_6(x_i) \mid x_i \in [-5,5], i \in N$$

kde:

A	konstanta 10
n	dimenze prostoru řešení E^n

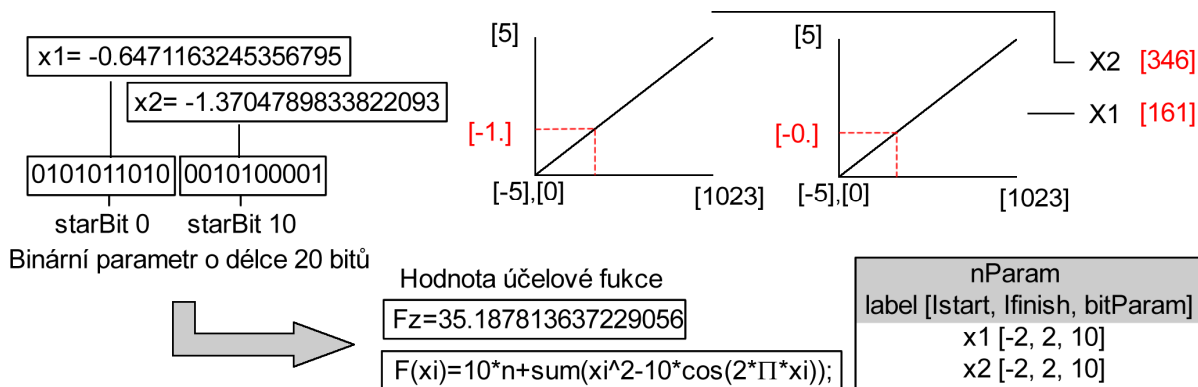
Funkční optimalizace se aplikuje do *HC* pomocí ohodnocení generovaného binárního řetězce. Ohodnocení se provádí dle parametrů a hodnot zadaných uživatelem. Hodnoty obsažené v parametru jsou následující.

- **Istart:** definuje počátek intervalu výpočtu pro parametr.
- **Ifinish:** definuje konec intervalu výpočtu pro parametr.
- **bitParam:** definuje kombinatorickou náročnost, a tedy i přesnost parametru, udává se v počtu bitů na podřetězec.



OBR. 10: 3D graf optimalizační funkce $F_6(x)$

Postup ohodnocení binárního řetězce pro optimalizaci funkce je znázorněn na obr. 11, příklad $F_6(x)$ s parametry x_1 a x_2 . Kompletní algoritmus ohodnocení se dá rozdělit na čtyři základní kroky.



OBR. 11: Princip ohodnocení funkční optimalizace

- Krok 1:** rozložení binárního řetězce na jednotlivé parametry podle přesnosti každého z parametrů.
- Krok 2:** jednotlivé parametry se převedou na celé číslo.
- Krok 3:** pomocí lineární transformace se celé číslo převede na konkrétní hodnotu parametru.
- Krok 4:** z parametrů se vypočítá optimální hodnota příkladu dle řešené funkce.

Zápis funkční optimalizace pomocí skriptu *CMD* line

Aplikace funkční optimalizace obsažené v této diplomové práci zavádí pomocí příkazů z *CMD* line, příkazy specifické pro tento problém jsou kategorie *parameter*, *function* a *apc*, syntaxe je následující:

parameter

<typ>	popis	formát hodnot
		příklad
1.	Zapsání všech hodnot v parametru ve tvaru (<i>název</i> [<i>Istart</i> , <i>Ifinish</i> , <i>přesnost</i>];)	string [real, real, integer]; x1[-2,2,10];
2.	Definování jen velikost rozsahu a přesnosti je zadána předešlým parametrem tvar (<i>název</i> [<i>Istart</i> , <i>Ifinish</i>];)	string [real, real]; x2[-2,2];
3.	Použit jen název parametru, zbytek je zděděný z předešlého parametru, tvar (<i>název</i> ;	string; x3;

function

<typ>	popis	formát hodnot
		příklad
-	Funkce se zavádí prvně pomocí názvu problému, tvar (<i>název</i> = <i>funkce</i> ;	string = funkce; fce(x) = xxx;
$\Sigma(x_i)$	Suma, tvar (sum (<i>název</i> (proměnná)) proměnná=start:finish)	sum(string(char)) char=integer:integer sum(x(i)) i=1:2
$x \wedge y$	Základní operace +-*/%^, příklad tvar (<i>název</i> ^ <i>název</i>)	string ^ string x ^ y
sin(x)	sin, cos, tan, log, abs příklad tvar (<i>sin</i> (<i>název</i>))	string(string) sin(x)
$\cong x$	floor, ceil, round, rand, příklad tvar (<i>rand</i> (<i>název</i>))	string(string) rand(x)
x!	Faktoriál, tvar (<i>název</i> !)	string! x!

apc

<typ>	popis	formát hodnot
		příklad
1.	Typ <i>APC</i> se definuje pomocí funkce mode, tvar (mode[definice1, definice2, ...])	mode [string]; mode [x (definice)];
1a.	Definice - <i>APC</i> pomocí zpřesnění rozsahu při dodržení hranic (<i>název</i> (range , rozsah, hranice, opakování))	string (range, real, boolean, integer) x1(range,10,true,1)
1b.	Definice - <i>APC</i> pomocí zvýšení vzorkování (<i>název</i> (bin , bitů, opakování))	x2(bin, integer, integer) x2(bin,1,1)
2.	Zvolení kdy se bude provádět <i>APC</i> operace se provede až s poslední (nejlepší) hodnotou při každém run(u) se provede operace	select[volba]; select[best]; select[all];

dc

<typ>	popis	formát hodnot
		příklad
1.	specifické rozdělení funkční opt. pro <i>DC</i> (dcSpec [dílčů, název1, název2...])	dcSpec [integer, název]; dcSpec[3,x1,x2];

Příklad $F_6(x)$ z obr. 10 zapsaný pomocí skriptu *CMD* je následující :

```
1| function{
2|     F6(xi)=10*2+sum(x(i)^2-10*cos(2*PI*x(i))); i=1:2;}
3| param{
4|     x1[-2,2,10];
5|     x2[-2,2,10];}
6| apc{
7|     mode[x1(range,10,true,1), x2(bin,1,1)];
8|     select[best];}
9| dc{
10|     dcSpec[3,x,y];}
```

○ Způsob distribuce výpočtu

Specifický postup (2) pro funkční optimalizaci je prováděn pomocí dělení jednoho až n parametrů (n je počet všech parametrů) na definovaný počet dílů, výsledný počet segmentů určuje i počet připojených uživatelů. Postup je znázorněn na příkladě pro Postup 3, viz. obr. 12.

Příklad DC rozdělení

Jako příklad rozdělení výpočtu pro funkční optimalizace použijeme funkci $F_6(x)$ na definičním rozsahu viz. obr. 11, použitý způsob rozdělení je podle postupu 3, dělíme každou osu na dvě části s trojnásobným opakováním. Pro tento případ DC je možno připojit maximálně 12 uživatelů k simultánní práci – vytvoří se 12 prací z toho 4 unikátní.

Fce	D	O	Schéma	S	P	I [	]	Pr	info
$F_6(x)$	2x 2x	3x		1	x1	-2	0	segment 1_a segment 1_b segment 1_c	max. 12 uživatelů
					x2	-2	0		
				2	x1	-2	0	segment 2_a segment 2_b segment 2_c	
					x2	0	2		
				3	x1	0	2	segment 3_a segment 3_b segment 3_c	
					x2	-2	0		
				4	x1	0	2	segment 4_a segment 4_b segment 4_c	
					x2	0	2		
Fce: prohledávaná funkce			I: interval parametru			Pr: název všech vytvořených prací			
P: parametry funkce, nových segmentů			S: nové segmenty pro inikátní parametry			info: počet připojených uživatelů			
D: počet dělení definičního rozsahupro každý parametr			O: počet opakování každého unikátního segmentu						

OBR. 12: Rozdělení výpočtu ve funkční optimalizaci

○ Parametry dle XML souboru.

Zápis jednoho parametru (v našem případě x_2) z příkladu $F_6(x)$ v konečné podobě ve formátu XML v sekci *paramater* využívajícího v programu s podrobným popisem jednotlivých objektů.

```

1|      <parametr label="x1" index="8">
1|      <constant label="constant">
2|          <int label="Int: [0,1023]" intMax="1023"/>
3|          <real label="Re: [2,-2]" reMin="2" reMax="-2"/>
4|          <accurency label="accurency: 10" param="10"/>
5|          <startBit label="startBit: 0" param="0"/>
6|      </constant>
7|      <apc label="apc">
8|          <apcMode label="apcMode: bin" param="bin"/>
9|          <apcBin label="apcBin: 1" param="1"/>
10|          <apcPersent label="apc%: 100" param="100"/>
11|          <apcBoundries label="apcBoundries: false" param="0"/>
12|          <runsApc label="runsApc: 1" param="1"/>
13|      </apc>
14|      <digit label="digitTranslation">
15|          <bin label="Bin: 11011100000" param="11011100000"/>
16|          <int label="Int: 1760" param="1760"/>
17|          <real label="Re: 8.59794821690278" param="8.59794821690278"/>
18|      </digit>
19|  </parametr>

```

<název>	popis	formát hodnot
		příklad
parametr	Název proměnné a index pozice pro výpočet <i>fceOpt</i> (index generuje program při inicializaci).	Znakový řetězec o max. 30 znacích, celočíselná hodnota <i>x1, 8</i>
constant	Defaultní hodnoty nastavení parametru.	
int	Hranice pro osu celočíselných hodnot potřebné převod binárního čísla na celočíselné.	Celočíselná hodnota [2^(délka binárního řetězce)] <i>1023</i>
real	Hranice pro osu reálných hodnot potřebné pro převod celočíselné hodnoty na reálnou hodnotu parametru.	Reálná hodnota rozsahu pro parametr <2³², 2³²> <i>[-2,2]</i>
accurency	Počet bitů na parametr, zadáváno v počátečních hodnotách.	Celočíselná hodnota v rozmezí <0, 64> <i>10</i>
startBit	Index start bitu v binárním řetězci určující počátek parametru.	Celočíselná hodnota v rozmezí <0, délka binárního řetězce-1> <i>0</i>
apc	Konstantní hodnoty dynamického programování.	
apcMode	Definování typu použití dynamického programování APC.	Znakový řetězec podle typu apc, základní rozdělení: [bin, range] <i>bin</i>
apcBin	Hodnoty pro typ APC se zvýšením vzorkování problému (počet navýšení bitů na kolo).	Celočíselná hodnota v rozmezí <0, 64> <i>1</i>
apcPersent	Velikost procent z absolutní hodnoty rozsahu prohledávání o kolik se zpřesní rozsah na kolo.	Celočíselná hodnota v rozmezí <0,100> <i>100</i>
apcBoundries	Specifikování pro předešlý typ z pohledu dodržování hranic.	Znakový řetězec: [true, false] <i>false</i>

runsApc	Počet opakování APC výpočtu při nalezení výsledku.	Celočíselná hodnota v rozmezí <0, 64>
		1
digit	Hodnoty přepočtu z binárního řetězce.	
bin	Binární řetězec vztahující se pro aktuální parametr.	Znakový řetězec <0,1> o velikosti viz accuracy
		11011100000
int	Přepočet binárního řetězce na celočíselnou hodnotu.	Celočíselná hodnota v rozmezí <0, 2 ^{accuracy} -1>
		1760
real	Skutečná pozice parametru v defaultním rozsahu. Konečná hodnota používaná ke konečnému výpočtu.	Reálná hodnota rozsahu pro parametr <-2 ³² , 2 ³² >
		8.59794821690278

2.4 PROBLÉM BATOHU

Problém batohu (*knapsack problem*) je součástí kombinatorické optimalizace a patří do skupiny nejlehčích NP-těžkých problémů (*nondeterministic polynomial-time hard*). Obecně v literatuře existuje mnoho variant tohoto problému, které mají různé nároky na algoritmus řešení. Obecné zadání problému je následující.

- celé číslo n (počet věcí).
- celé číslo M (kapacita batohu).
- konečná množina $V=\{v_1, v_2, \dots, v_n\}$ (hmotnosti věcí).
- konečná množina $C=\{c_1, c_2, \dots, c_n\}$ (ceny věcí).

○ Definice problému

Definice pro problém batohu s maximalizací hodnoty věcí v něm. Problém se dá rozdělit pomocí dvou základních definic:

- **batoh bez opakování:** omezí počet opakování (x_i) na 0,1, tedy všechny objekty v batohu mohou být obsaženy maximálně 1, nebo nebudou v batohu obsaženy vůbec. Matematický zápis je dle vzorce (3.7).

$$\text{Maximum: } \sum_{i=1}^n C_i \cdot x_i; \quad x_i \in \{0,1\} \quad (3.7)$$

$$\text{Podmínka: } \sum_{j=1}^n V_j \cdot x_j \leq M; \quad x_j \in \{0,1\} \quad (3.8)$$

- **batoh s opakováním:** omezí počet opakování (x_i) na definovanou hodnotu (b_i), tedy všechny objekty v batohu můžou maximálně s hodnotou (b_i), nebo nebudou v batohu obsaženy vůbec. Matematický zápis podle vzorce (3.9).

$$\text{Maximum: } \sum_{i=1}^n C_i \cdot x_i; \quad x_i \in \{0, b_i\} \quad (3.9)$$

$$\text{Podmínka: } \sum_{j=1}^n V_j \cdot x_j \leq M; \quad x_j \in \{0, b_j\} \quad (3.10)$$

Pro všechny typy problému musí platit podmínka nosnosti, kdy součet vah věcí v batohu nesmí překročit možnou maximální nosnost batohu, viz. vzorce (3.8) a (3.10).

○ Možnosti aplikace problému

Problém batohu je prototypem maximalizační optimalizace, proto v reálném světě existuje mnoho mutací. Jedna z možných aplikací existuje v prostředí obchodu, kdy základní definice zůstane zachována a jen se změní věci na kupované objekty s parametry (zisk a cena) a koncová podmínka (3.10), sleduje maximální zisk z obchodu. Zadání příkladu:

- celé číslo n (počet akcií).
- celé číslo P (množství peněz).
- konečná množina $C = \{c_1, c_2, \dots, c_n\}$ (cena akcií).
- konečná množina $Z = \{z_1, z_2, \dots, z_n\}$ (zisk akcií).

Další možností aplikace je např. v letecké dopravě (náklad zavazadlovém prostoru) atd.

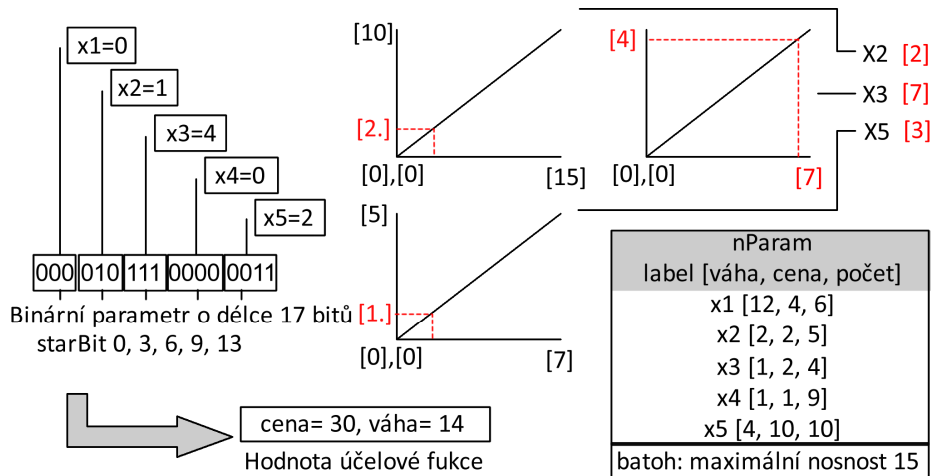
○ Aplikace problému batohu v HC

V této diplomové práci je požit problém batohu s n opakováním a pro m počet předmětů. Pro demonstraci aplikace v HC algoritmu je zvolen příklad batoh 15. Příklad ukazuje optimální rozložení batohu o nosnosti maximálně 15 jednotek a 5 druhů položek s různou cenou a váhou. Počet opakování u předmětu není zvolen stejně kvůli přepočtu a znázornění přesnosti. Přesné zadání pomocí obr. 13.

Batoh	Počátek				Problém	Výsledek	
	Věci	Váha	Cena	Počet		Batoh	Hodnoty
max nosnost 15	x1	12	4	6	max cena	x3	Cena 36. Váha 15.
	x2	2	2	5		x3	
	x3	1	2	4		x3	
	x4	1	1	9		x5	
	x5	4	10	10		x5	

OBR. 13: Optimální rozložení batohu 15

Základem aplikace v HC je ohodnocení binárního řetězce. Tvorba binárního řetězce v problému batohu je tvořena obdobným způsobem jako u funkční optimalizace, neboli rozdělení řetězce podle parametrů. Každá položka v řetězci má přesnost (počet bitů) dle počtu opakování, viz. vzorec (3.11). Přesnost je nastavena na nejbližší vyšší počet bitů, např. pro opakování 5 se musí nastavit 3 bity (až 7 opakování) a pomocí lineární transformace parametru vzorec (2.1), pak přepočítávat pro hodnotu 5. Celková délka řetězce je součet přesností všech parametrů v něm obsažených.



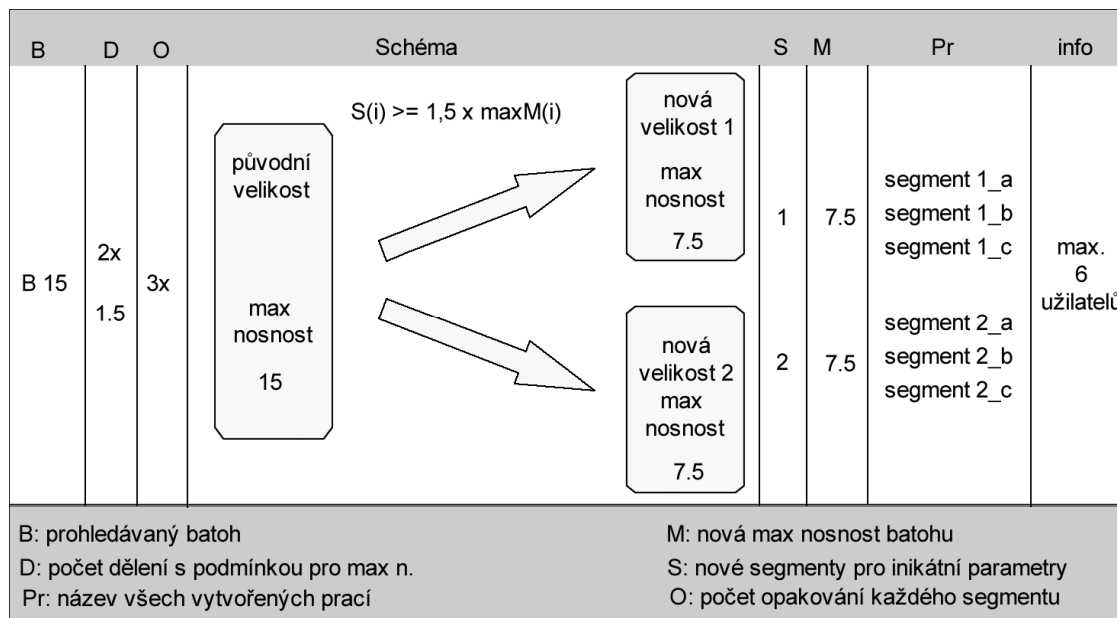
OBR. 14: Princip ohodnocení batohu

○ Způsob distribuce výpočtu

Postup 2 u problému batohu funguje pomocí dělení maximální nosnosti základního batohu na definovaný počet dílů s volitelnou podmínkou nosnosti nového batohu, výsledný počet segmentů určuje i počet připojených uživatelů, viz. příklad DC výpočtů pro batoh obr. 13.

Příklad DC rozdělení

Pomocí příkladu pro batoh 15 je znázorněn postup číslo 3, viz, obr. 15. Aplikace postupu 3 s dělením batohu na 2 části s podmínkou velikosti nového segmentu (nosnost min 1.5 velikosti váhy nejtěžšího předmětu z parametrů) a počtem prostého opakování je roven 3. Výsledný počet nových prací je 6 z toho 2 unikátní.



OBR. 15: Rozdělení výpočtu pro batoh

$$a = \text{ceil}(\log_2(x + 1)) \quad (3.11)$$

kde:

a	přesnost parametru
x	počet opakování předmětu v úloze
ceil	zaokrouhlení čísla na nejbližší vyšší

Zápis batohu pomocí skriptu *CMD* line

Specifické vlastnosti batohu jsou definovány pomocí skriptů, příkazy specifické pro tento problém jsou kategorie *parameter*, *function* a *dc*, syntaxe je následující:

parameter

<typ>	popis	formát hodnot
		příklad
1.	Zapsání všech hodnot v parametru ve tvaru (<i>název</i> [<i>váha</i> , <i>cena</i> , <i>opakování</i>];)	string [real, real, integer];
		x1[12,4,10];
2.	Definování jen velikost rozsahu a přesnost je zadána předešlým parametrem tvar (<i>název</i> [<i>váha</i> , <i>cena</i>];)	string [real, real];
		x2[2,2];
3.	Použit jen název parametru, zbytek je zděděný z předešlého parametru, tvar (<i>název</i> ;))	string;
		x3;

function

<typ>	popis	formát hodnot
		příklad
-	Problém batoh se definuje, tvar (<i>název</i> = <i>nosnost</i> ;))	string = real;
		knapsack = 15;

dc

<typ>	popis	formát hodnot
		příklad
-	Specifické rozdělení batohu pro <i>DC</i> , tvar (<i>dcSpec</i> [<i>počet</i>];)	string [integer];
		dcSpec[2];

Zápis plného znění zkušební příkladu pro batoh 15 do *CMD* line dle parametrů z příkladu obr. 13 :

```
1| function{
2|     knapsack = 15;}
3| param{
4|     x1[12,4,6];
5|     x2[2,2,5];
6|     x3[1,2,4];
7|     x4[1,1,9];
8|     x5[4,10,10]}
9| dc{
10|     dcSpec[2]; }
```

○ Parametry dle *XML* souboru

Popis hodnot parametru pro problém batohu s příkladem a podrobným popisem je znázorněn v následující tabulce.

```

1|      <parameters label="parameters">
2|      <parametr label="x2">
3|      <constant label="constant">
4|          <int label="Int: [0,7]" intMax="7"/>
5|          <real label="Re: [0,6]" reMax="6"/>
6|          <weigh label="weigh: 12" param="12"/>
7|          <price label="price: 4" param="4"/>
8|          <accurency label="Accurency: 3" param="3"/>
9|          <startBit label="Start bit: 0" param="0"/>
10|      </constant>
11|      <digit label="digitTranslation">
12|          <bin label="Bin: 101" param="101"/>
13|          <int label="Int: 5" param="5"/>
14|          <real label="Re: 3" param="3"/>
15|          <price label="price: 12" param="12"/>
16|          <weigh label="weigh: 15" param="15"/>
17|      </digit>
18|      </parametr>
19|      <parmNum label="paramNum: 2" param="2"/>
20|      <binLength label="binLength: 6" param="6"/>
21|  </parameters>

```

<název>	popis	formát hodnot
		příklad
parametr	Název položky předmětu.	Znakový řetězec o max. 30 znacích X2
int	Hranice pro osu celočíselných hodnot pro převod binárního čísla na celočíselné.	Celočíselná hodnota [2 ^{^(délka binárního řetězce)}] 1023
real	Hranice pro osu reálných hodnot pro převod celočíselné hodnoty na reálnou hodnotu (skutečný možný počet opakování předmětu) parametru.	Reálná hodnota počtu opakování o max. velikosti 1023 6
weigh	Hodnota váhy pro předmět.	Reálná hodnota <bez omezení> 5
price	Hodnota ceny pro předmět.	Reálná hodnota <bez omezení> 4
accurency	Počet bitů na parametr (přepočteno podle počtu opakování na nejbližší vyšší hodnotu), počítáno programem.	Celočíselná hodnota v rozmezí <0, 64> 3
real	Vypočtená skutečná hodnota počtů opakování předmětu v batohu.	Reálná hodnota počtu opakování o max. velikosti 1023 3
price	Celková cena všech předmětů stejného typu v batohu.	Reálná hodnota <bez omezení> 12
weigh	Celková váha všech předmětů stejného typu v batohu.	Reálná hodnota <bez omezení> 15

2.5 PROBLÉM OBCHODNÍHO CESTUJÍCÍHO

Problém spočívající ve výpočtu hodnoty cesty mezi množinou měst se nazývá *traveling salesman problem* (zkráceně *TSP*) a úkolem algoritmu je najít nejmenší možnou cenu (vzdálenost cesty mezi jednotlivými městy pro zadanou trasu pohybu), jedná se tedy o minimalizační problém. V teorii grafů je tento typ úlohy znázorněn pomocí *hamilton*(ovské) kružnice tzv. nalezení nejkratší cesty v grafu. Problém *TSP* je diskrétní kombinatorická optimalizace a patří mezi *NP* plné úlohy, stejně jako u batohu existuje mnoho variant problému.

- celé číslo n (počet měst).
- typ úlohy pro *TSP* (bude dále definováno).
- konečná množina $V=\{v_1, v_2, \dots, v_n\}$ (souřadnice měst).

V praxi je řešen problém *TSP* pouze přibližně pomocí heuristických metod, genetickými algoritmy tak, aby výsledek byl dosažitelný v rozumném čase, ale za cenu omezení požadavku na vyhledání přesného řešení problému. Problém *TSP* je nedeterministický polynomiální problém. Nedeterministický problém je ten, kdy *ND* počítač umožňující v každém kroku rozvětvit výpočet na libovolný počet větví, by mohl začít v některém „městě“ rozdělit propočít délky trasy na tolik větví, kolik z města vede silnic, a v každém z cílových měst postupovat stejně s výjimkou tras vedoucích do již navštívených měst. Tak by prohledal všechny možné trasy v n výpočetních krocích, pokud počet měst činí n , a rozvětvil by se maximálně do $(n-1)!$ větví [9].

Problém *TSP* lze rozdělit pomocí mnoha hledisek, jedno z těch základních je rozdělení dle požitých cest v grafu, tím se *TSP* dělí na dva způsoby.

- **Symetrický problém TSP:** takový typ *TSP*, kdy je vzdálenost mezi dvěma městy rovna v jakémkoliv směru. Příkladem takového typu je i neorientovaný graf. Symetrický problém omezí možné výsledky na polovinu existující [9].
- **Asymetrický problém TSP:** typ, kdy vzdálenost mezi dvěma městy není rovná vzdálenost z druhého směru průchodu grafem. Příkladem je orientovaný graf a v reálném životě použití jednosměrných cest. Tenhle typ *TSP* není obsažen v této diplomové práci [9].

Další možností, jak rozdělit problém, je podle pořadí a množství navštívených měst v cestě výpočtu:

- **S opakováním:** je typ problému, kdy se může při výpočtu cesty navštívit jednotlivé město více než jen jednou, není omezen počet návštěv.
- **Bez opakování:** prohledávaná cesta je omezena jen na jedno navštívení města v jednom okruhu.
- **S návratem:** počátek a konec výsledné cesty musí být vždy ve stejném městě.
- **Bez návratu:** začátek ani konec cesty není nějak omezen nebo předem určen žádnými pravidly.

V diplomové práci byly zvoleny čtyři základní rozdělení *TSP* podle druhu navštívených měst.

- **Problém 1:** počáteční ani koncové město není specifikováno, může se začít odkudkoliv, každé město je navštíveno jen jednou.
- **Problém 2:** obdoba problému 1, s vytvořením kruhové cesty, kdy první město je i zároveň i finální město.
- **Problém 3:** kdy uživatel už při inicializaci problému definuje pořadí jednoho až n měst, kde n je maximum počet bodů, problém bez opakování.
- **Problém 4:** obdoba problému 3 s vytvořením kruhové cesty viz problém 2.

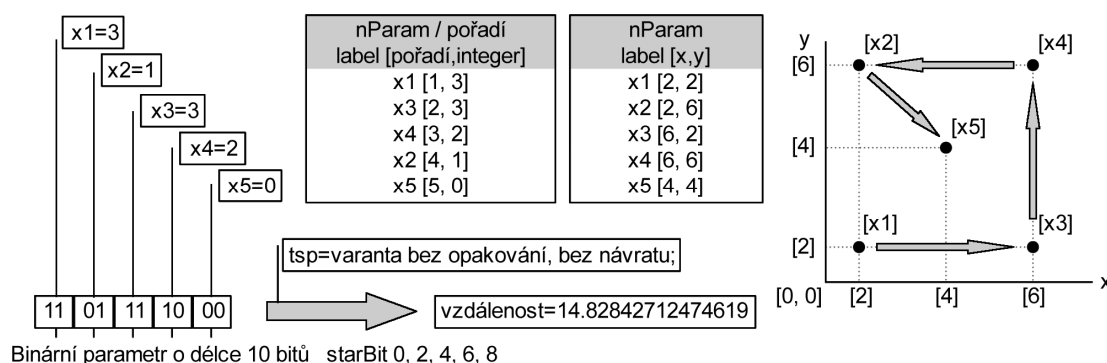
○ Aplikace problému TSP v HC

Máme problém, který obsahuje pět měst x_1, x_2, x_3, x_4, x_5 dle definice na obr. 16 a úkolem je najít nejmenší délku cesty mezi nimi pro problém 1. Maximální počet možných řešení pro pět měst je 120 cest mezi body. Optimálních cest na tomto příkladě existuje více, jedno možné řešení je naznačeno na obrázku.

Počátek		Města		Výsledek			
Mapa		x	y	Problém	Pořadí	Hodnoty	
	[6]	x1	2	2	bez opakování min cesta bez návratu	x4	Cesta 13.65685424949238
		x2	2	6		x5	
		x3	6	2		x3	
	[2]	x4	6	6		x1	
	[2]	x5	4	4		x2	

OBR. 16: Optimální řešení pro TSP T5

Aplikace pro HC probíhá pomocí ohodnocení generovaných řetězců. Binární řetězec je rozdělen na parametry (města) a každý jeho prvek obsahuje pořadí města tak, aby se žádné z měst neopakovalo ve výsledné smyčce. Přesnost parametru je vypočtena pomocí vzorce (3.12). Byl použit příklad TSP bez opakování měst. Vyhodnocení výsledného pořadí měst je znázorněno na obr. 17. a provádí se pomocí seřazování celočíselných hodnot binárního řetězce.



OBR. 17: Princip ohodnocení TSP

- **Krok 1:** provedou se kroky 1 a 2 z funkční optimalizace.
- **Krok 2:** celočíselné hodnoty z binárního řetězce se seřadí sestupně, pak výsledná tabulka seřazených hodnot udává i pořadí navštívených měst.

$$a = \text{ceil} \left(\frac{\text{ceil}(\log_2 x!)}{x} \right) \quad (3.12)$$

kde:

a	přesnost parametru
x	počet měst v úloze
ceil	zaokrouhlení čísla na nejbližší vyšší

○ Způsob distribuce výpočtu

Postup 2 pro problém TSP je prováděn pomocí rozdělení původní mapy na menší celky v zájemně propojených pomocí jednoho města. Pro tento postup definuje maximální počet měst na jednu smyčku. Algoritmus začne u prvního města v pořadí podle definování v CMD line, spočte vzdálenosti všech měst k referenčnímu městu a vybere nejbližší definovaný počet měst, ze kterých posléze

vytvoří smyčku (nastaví prvnímu městu jako výchozí město a poslední město ve smyčce jako koncové město). Další smyčka začíná tam, kde předchozí skočila (nastaví poslední město z předešlé smyčky jako referenční a opakuje postup znovu). Poslední smyčka obsahuje zbylý počet měst na mapě. Lze distribuovat jen výpočty pro problém 1 a 2 a výsledné počítané segmenty jsou ve formátu Problému 3. Postup je znázorněn pomocí příkladu s postupem 3, viz. obr. 18.

Příklad DC rozdělení

Distribuce výpočtu z příkladu obr. 16 je prováděna pomocí postupu 3 *TSP* problému. Dělíme mapu na smyčky o maximálním počtu měst ve smyčce, je stanoveno na 4, celková mapa byla rozdělena na výsledné 2 smyčky o 4 a 2 městech. Počet opakování 3, tedy maximální počet připojených uživatelů 6.

tsp	D	O	Schéma	S	M	Pr	info
tsp 5	4	3x	mapa 1 S1: x1 (start), x5, x3, x2 (konec) — : vzdálenost S2: x2 (start), x4 (konec) — : smyčka mapa 2	1	4	segment 1_a segment 1_b segment 1_c	max. 6 uživatelů
				2	2	segment 2_a segment 2_b segment 2_c	
B: prohledávaný tsp problém			M: počet měst ve smyčce				
D: maximální počet měst ve smyčce			S: nové segmenty pro inikátní parametry				
Pr: název všech vytvořených prací			O: počet opakování každého segmentu				

OBR. 18: Rozdělení výpočtu pro TSP problém

Zápis batohu pomocí skriptu CMD line

Specifické vlastnosti *TSP* problému jsou definovány pomocí skriptů *CMD*, příkazy specifické pro tento typ jsou z kategorie *parameter*, *function* a *dc*, syntaxe je následující:

parameter

<typ>	Popis	formát hodnot
		příklad
1.	Zapsání plné definice města, tvaru (název[souřadnice X, souřadnice Y, pořadí];)	string [+real, +real, integer]; x1[2,2,1];
2.	Definice polohy bez určení pořadí, tvar (název[souřadnice X, souřadnice Y];)	string [+real, +real]; x2[2,2];

function

<typ>	popis	formát hodnot
		příklad
-	Definice typu <i>TSP</i> problému, tvar (název = typ;)	string = integer; tsp = 1;

dc

<typ>	Popis	formát hodnot
		příklad
-	Specifické rozdělení <i>TSP</i> problému dle měst pro <i>DC</i> , tvar (dcSpec [počet];)	string [integer]; dcSpec[4];

Plný zápis příkladu pomocí skriptů v *CMD* line je následující:

```

1| function {
2|     problem = 1; }
3| param {
4|     x1[2,2];
5|     x2[2,6];
6|     x3[6,2];
7|     x4[6,6];
8|     x5[4,4]; }
9| dc {
10|     dcSpec[2]; }

```

○ Parametry dle XML souboru

Popis jednoho města v *TSP* optimalizace pomocí *XML* dokumentu používajícího programy, vlastnosti města jsou z příkladu, viz. obr. 16 podrobný popis v následující tabulce.

```

1| <parametr label="x1" index="0">
2|   <constant label="constant">
3|     <sourX label="axis X: 2" param="2"/>
4|     <sourY label="axis Y: 2" param="2"/>
5|   </constant>
6|   <digit label="digitTranslation">
7|     <bin label="Bin: 101" param="101"/>
8|     <int label="Int: 5" param="5"/>
9|     <place label="place: 4" param="4"/>
10|   </digit>
11| </parametr>

```

<název>	popis	formát hodnot
		příklad
parametr	Název bodu na mapě.	Znakový řetězec o max. 30 znacích <i>x1</i>
sourX	Udává souřadnice města pro osu X k absolutnímu bodu.	Kladná reálná hodnota <kladné hod. bez omezení> <i>2</i>
sourY	Udává souřadnice města pro osu Y k absolutnímu bodu.	Kladná reálná hodnota <kladné hod. bez omezení> <i>2</i>
place	Hodnota pořadí města ve smyčce cesty od 1 do počet měst.	Celočíselná hodnota v rozmezí <1, počet měst> <i>4</i>

KAPITOLA 3 DISTRIBUOVANÉ VÝPOČTY

Je obecně známo, že spojování sil při práci je mnohem výhodnější než práci odvést sám. Proto byly vyvinuty v počítačové technice tzv. distribuované výpočty (z anglického slova –*Distributed Computing* zkráceně *DC*). Příkladem aplikování *DC* je využití sítě internet, kdy ve světě je geograficky rozmístěno obrovské množství velmi silných počítačů, které v drtivé většině svého života nevyužívají ani zlomek svého výpočetního potenciálu. Proto se *DC* snaží tyto počítače spojit do jedné výpočetní sítě a v době nečinnosti *PC* využít jeho výkon pro společné úlohy.

3.1 ÚVOD DO PROBLEMATIKY

Prvním spuštěným *DC* výpočtem se stal projekt společnosti *Distributed.net*. Tahle společnost jako první využila myšlenku vytvoření jednoho velkého virtuálního počítače pomocí sítě internet, a to na přelomu 1997/1998, kdy vydali klienta pro výpočet *Golomb Ruler*, neboli lámáním šifer tzv. hrubou silou [4].

Dalším *DC* projekt byl *GIMPS (Great Internet Mersenne Prime Search)*, který vypočítává tzv. Mersenova prvočísla. Mersenovo prvočíslo nazýváme takové číslo, které je celočíselnou mocninou 2 zmenšené o jedničku. Do roku 2006 bylo známo 44 M. prvočísel. Pomocí tohoto projektu bylo vypočítané 9 M. prvočísel. M. prvočísla velké mají uplatnění v kryptografii.

○ Distribuce pomocí samostatných klientů

První možností jak uplatnit *DC* výpočty je pomocí volně šiřitelných klientů, kdy uživatel má jen rámcové znalosti o probíhajícím výpočtu a jediná jeho starost je stažení výpočtového programu. Uživatel nemusí nijak nastavovat výpočet stačí se připojit do té konkrétní *DB*, kterou si vybere a vše ostatní zařídí klient sám. Tento typ distribuce je ve světě velice populární, neznámější projekty jsou *Home* a *Bionic*.

● Projekt Folding@Home.

Folding@Home je projektem Stanfordské Univerzity, která se nechala inspirovat dobrými výsledky předchozích *DC* projektů a vydali klient pro výpočet bílkovin. Nahradil tak původně zamýšlené použití super počítače. Superpočítač prakticky pracuje skoro na stejném principu jako *DC*. Super *PC* je uspořádání stovek procesorů spojených pomocí velmi rychle sítě [5].

Folding (česky skládání) je skládání bílkovin v prostoru. Každá bílkovina se skládá v řadu mikrosekund a výpočet jedné nanosekundy průměrnému *PC* zabere asi jeden týden, 3 roky jedna mikrosekunda a desetiletí pro výpočet jedné bílkoviny. Proto tento projekt byl vybrán jako ideální pro zapojení ve výpočtech pomocí *DC* klientů.

Použití grafických karet

Projekt *folding@Home* v roce 2006 použil jako první v historii ve výpočtu *DC* sílu grafických karet. Stanfordská Univerzita na tomto projektu spolupracovala se společností ATI, ve spolupráci bylo využito tzv. *pixel shader(ů)* pro práci s aritmeticko-logických jednotkách grafického čipu. Dne 2. října 2006 byl vypuštěn klient v beta verzi a za 9 dnů veřejného testování se přiblížil k hranici 31 tera *FLOPS*. Tohoto výsledku dosáhl při použití cca 450 grafických karet (ATI Radeon X1900). Při tomto testu se potvrdil záměr, že využití *GPU* pro výpočet s plovoucí desetinnou čárkou mnohonásobně převyšuje výkon procesoru. V době testu to bylo přibližně 70. násobek výkonu. V současné době je klient ve verzi pro více typů karet a hlavně byla zakomponována podpora Direct x 10. Pomocí Direct x 10 jsou *GPU* mnohem výkonnější a stoupá jejich účinnost při *DC* výpočtech kvůli zakomponování pokročilých funkcí, jako je třeba *Shader Model*. Velký problém využití *GPU* je absence vhodných ovládacích prvků [5].

Stanfordská Univerzita vyvinula vlastní rozhraní *Brook*, kvůli neexistenci jednotného rozhraní pro přístup k čipu grafické karty pro různé firmy (ATI vs nvidia), který je rozšířením programovacího jazyka C. Pomocí spolupráce ATI na vývoji lze přistupovat s technologií *Brook* až k tzv. *CTM (Close to metal)* rozhraní pro *GP-GPU*. Konkurenční společnost nVidiase tohoto projektu nezúčastní a používá svoje vlastní rozhraní *Compute Unified Device Architecture* (zkráceně *CUDA*). Trvalý problém ve využití GPU pod systémem Linux je neexistence vhodného ovladače, zatím byl vydán jen ovladač od společnosti nVidia s názvem *GPGPU CUDA*, a to roku 2007.

GPU mají na rozdíl od *CPU* architekturu s mnoha paralelními jádry. Na každém z těchto jader je možno provádět stovky výpočtů. Dalším kladem pro využití *GPU* k výpočtu je integrování služby *folding* do *PhysX PowerPack(u)* nVidia, který přináší lepší časy při výpočtu fyzikálních modelů. Pro paralelní výpočty přizpůsobené na chod v grafických kartách, může jejich výkon přinést velmi dobré výsledky [5].

Brook GPU

Runtime klient vyvíjený na Stanfordské univerzitě, jako rozšíření programovacího jazyka *steam* od společnosti Ati. Jeho hlavní přínos je v podobě využití moderních grafických akceleratorů pro ne-grafické operace, jako jsou výpočty s pohyblivou desetinou čárkou. Využití *Grafics processing unit* (zkráceně *GPU*) ve výpočtu se také někdy nazývá jako *General Purpose Graphics Processing Unit* (zkráceně *GP GPU*). *Brook* využívá čím dál tím více programů OpenGL v1.3+, DirectX v9+ nebo AMD „Close to metal“. Největší výhodou *Brook(u)* je jeho otevřenost s licencí spadající pod *GPL (General public license)*. Kompatibilita rozhraní *Brook* zahrnuje i grafické karty intel integrované na základní desce. Ale kvůli nízkým výkonům těchto karet se zatím nevyužívají k paralelním výpočtům [19].

CUDA GPU

Je stejně jak *Brook* paralelní výpočtový systém pro grafické karty, tentokrát vyvíjený u společnosti nVidia. *CUDA* je postavena na jazyce C. Dokáže spolupracovat ale i s programy, jako je Python, Fortran a Java. Poslední dobou se snaží nVidia otevřít architekturu co nejvíce softwarovým vývojářům. *CUDA* také obsahuje rozšíření pro výpočet fyzikálních jevů *PhysX*. *SDK CUDA* byla prve vydána 15. února 2007 [20].

Plastation 3

Folding@home jako první v březnu 2007 vydal klienta pro herní konzoly PS3. Kvůli multi-jádrovému procesoru *cell* obsaženém uvnitř konzole, který je navržen speciálně pro složité matematické výpočty a přesahuje výkon 10 -20 x normálního stolního počítače (2007), a grafickému výkonu se staly konzole důležitou součástí projektu *folding@home*. I když PS3 klienti tvoří k dnešnímu dni asi jen 15 % všech klientů, tak jejich výpočtová síla je kolem 35 % ze všech klientů v projektu *folding@home*. Projekt počítá, že dosáhne s každou samostatnou konzolou cca 20 GigaFlops, to je rovno asi 50 000 zapojených přístrojů v měřítku petaFlops. Momentálně je asi v provozu 150 000 PS3 klientů (2008) [6].

Datový tok.

Hranice jednoho *petaFLOPS* (tisíc miliard operací s plovoucí čárkou za sekundu) byla prolomena dne 16. září 2007. Bylo to vůbec poprvé, kdy tahle hranice byla pokořena a je také zapsána v Guinnessově knize rekordů. Pro představu v té době měl nejlepší super počítač výkon asi 0,478 *petaFLOPS*.

Projekt Bionic

Bionic (Berkeley Open Infrastructure for Network Computing) je projektem univerzity v Berkley v Kalifornii. Původně vznikl jako rozšíření projektu *Seti (Search for Extra-Terrestrial Intelligence)*, tedy projektu, který se zabýval vyhledáváním mimozemského života pomocí prohledávání prostoru s využitím radiového signálu. Po velkém úspěchu *Seti* se uvažovalo o použití *DC* i pro jiné vědecké činnosti ale bez nutnosti neustálého vytváření nového manageru, a tak aby *DC* výpočty zaujaly co

nejvíce lidí. Proto se vyvinulo otevřené rozhraní *Bionic*, které v sobě slučuje mnoho různých vědeckých projektů od biomedicíny až po lámaní šifer silou. Dne 22. 6. 2004 se jako první projekt integroval v systému *bionic* právě *seti@home*.

Různé projekty v systému *bionic* se snaží přilákat co nejvíce aktivních počítačů různými akcemi. Jednou z nich je i možnost registrace jména a soutěžení v různých ligách s ostatními uživateli, kdo odvede nejvíce procent práce na tom daném projektu. Čeští uživatelé patří k těm neaktivnějším v systému *bionic* i *folding@Home*, České republiky momentálně paří 8. místo z celého světa. Další možnosti soutěžení je liga týmů, které sdružuje uživatele a sčítá jejich hotovou práci. V ČR je oficiální tým *CNT (Czech National Team)*, momentálně má asi 6600 členů a je 6. nejvýkonnějším týmem z 75000 registrovaných týmů v systému. V České republice je nejvíce lidí zapojeno do výzkumu vesmíru (př. *Einstein*, *Seti*, *Cosmology*), na 2. místě je biologie, až na konci je kryptografie a matematika. Celosvětově je zapojeno do systému *bionic* více než 1,4 milionu lidí a z toho asi 18 000 českých uživatelů [4].

○ Obecný princip rozdělení úloh DC systému

V DC systémech existuje mnoho druhů distribuce, záleží na konkrétním programu, typu úlohy a možnostech sítě, ve kterých je momentálně distribuce zapojená. Všechny možnosti ale mají společná tři základní pravidla.

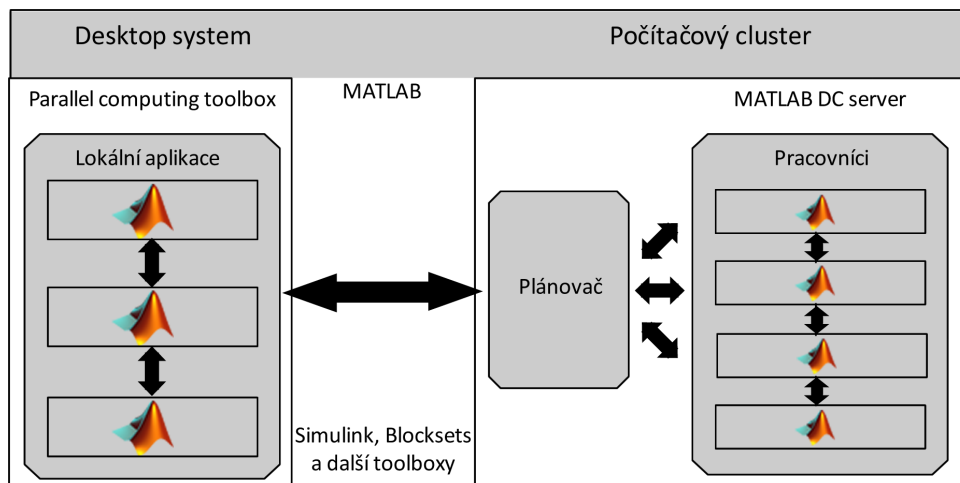
- Výpočty, které se dají rozdělit na mnoho samostatných dílů (tzv. pracovních jednotek *WU work units*). Po rozdělení na díly jsou pak doručeny k jednotlivým uživatelům (třeba pomocí internetu) projektu. Na konečném PC tato jednotka může zabrat od několika *kB* až po desítek *MB*. Od toho se i určuje doba zpracování jednotlivých *WU*. Po výpočtu na příslušném PC je daná *WU* odeslána zpět do mateřské databáze, kde se zpětně jednotky začleňují do zdrojového projektu. Příkladem takové to distribuce je *Seti*, *Einstein* a další [4].
- Dalším způsobem distribuce DC je pomocí odesílání celkových modelů problému. Tyhle modely se liší často jen vstupním nastavením. Taková to distribuce je náročnější jak na prostor na pevném disku, tak na čas výpočtu. Odesílání vypočteného modelu probíhá buď po částech, nebo až po dokončení celé struktury modelu. Příkladem je *Climate prediction* v *Bionic* [4].
- Poslední ze základních možností distribuce je pomocí skládání modelu na uživatelském PC. Tento systém využívá zatím pouze *Rosetta* a *RALP*. Z modelu se obdrží v tomto případě model bílkoviny a počáteční sada instrukcí. U takového projektu se dá nastavit čas výpočtu a v tomto čase se vytváří náhodné způsoby složení bílkovin a výpočet jejich vlastností. Odesílání probíhá pomocí nastaveného času a záleží jen na výkonu to konkrétního PC, kolik se stihne výpočtů za jednotku času [4].

○ Distribuce v rámci jednoho programu

Další možností aplikování DC systému je v nástavbách výpočtových programů. Dobrým příkladem tohoto typu je *PCT (Parallel Computing Toolbox)* a *DCS (Distributed Computing Server)* v programu *MATLAB*, obě komponenty dokáží řešit složité matematické operace jak v prostředí *Simulink* a dalších, tak v *MATLAB(u)* samotném. Výpočet lze paralelizovat na multiprocesorové stanice a počítačové *cluster(y)*. Schéma činnosti komponent je na obr. 19.

- **PCT:** je komponenta zpravující až osm paralelních procesů na lokálním PC. Instalace komponenty probíhá podobným způsobem jak u ostatních komponent, samotné ovládání procesů je pomocí příkazu `pmode: start`, po kterém se spustí paralelní algoritmy. V okně *Parallel Command Window* lze pak sledovat průběh jednotlivých procesů [13].

- **DCS:** dokáže spolu s předešlou komponentou vytvořit paralelní výpočty i v rámci sítě počítačů. Komponenta *PCT* vytvoří úlohy a ty jsou pak přeneseny do *DCS* (tento způsob se používá i k otestování správného nastavení paralelizace úlohy před finálním spuštěním na výpočetním clusteru). *DCS* podporuje zapojení 8, 16, 32, 96, 128 a 256 procesorů zároveň [13].

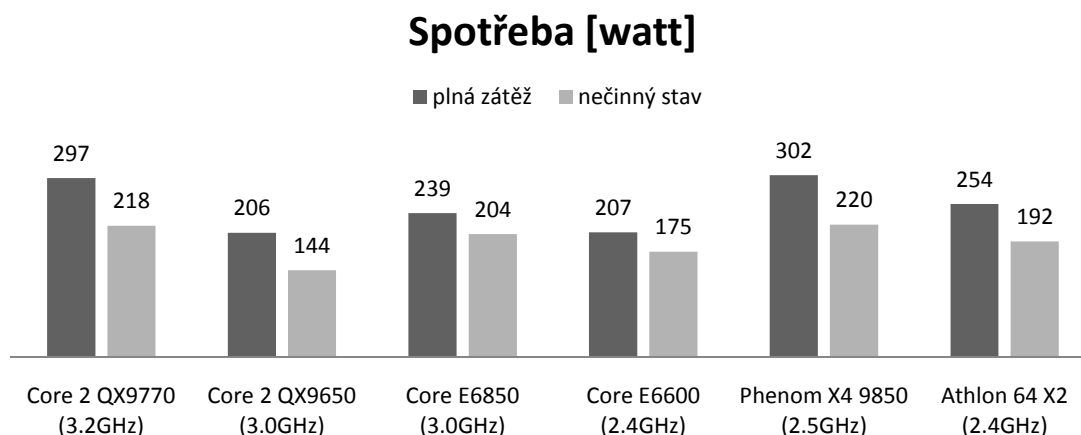


OBR. 19: Distribuce pomocí Matlab toolbox

○ Hardwarová náročnost DC výpočtů

Jelikož jsou *DC* výpočty zaměřeny převážně na použití *CPU*, popřípadě *GPU*, tak se výtečně hodí i jako test výkonu celého počítače. Hlavně matematické *DC* operace využívající i maximální množství *RAM* paměti jsou v hodné. Příkladem takového testování může být tester od skupiny *CNT*, který spolehlivě ověří, jak se dané *PC* bude chovat v *DC* výpočtech. Někdy jsou tyto testy mnohem přesnější než komerční testy pro měření výkonu [4].

Nevýhody při práci v *DC* je vytěžující procesor (grafické jádro) po nepřetržitou dobu na stanovený počet procent. Proto stoupá jejich spotřeba, teplota, popřípadě hluk z ventilátoru. Při vytížení procesoru na hranici 100 % stoupne spotřeba oproti nečinnému stavu asi o 30 %. Další nevýhodou může být trvalý přístup k síti internet (např. datový tok ve všech projektech *Bionic* se pohybuje maximálně do několika *MB* za den). Obecně vzato výkon *PC* a tedy i spotřebu energie ovlivňuje pět základních údajů. V následujícím obr. 20 je graf spotřeby pro vybrané typy procesorů v závislosti na zátěži.

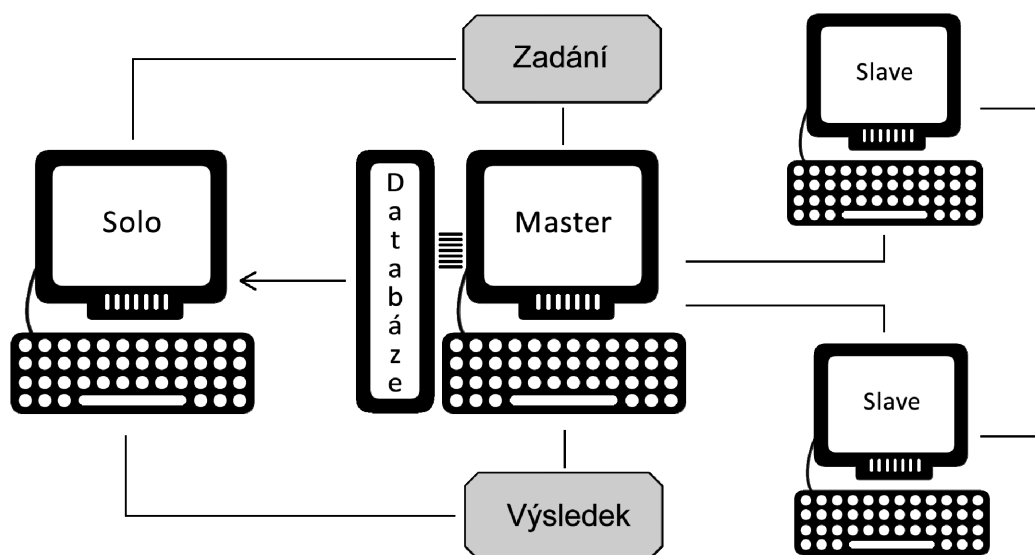


OBR. 20: Spotřeba pro různé CPU [14]

- počet procesorů (jader) obsažených v *PC*, popřípadě počet jader poskytnutých přímo pro výpočet.
- počet procent celkového výkonu přiřazený k *DC* výpočtu.
- doba zapojení výpočtu *DC*.
- výkon procesoru (takt, rychlost sběrnice).
- množství a rychlost paměti *RAM*.

3.2 APLIKOVÁNÍ DC V PRÁCI

Programová část diplomové práce je zaměřena na vývoj softwarových prostředků pro aplikaci *HC* algoritmu podle postupu vysvětleném v kapitole 3 a distribuované výpočty obecně. Programy byly vytvořeny celkem tři typy podle způsobu uplatnění. Jsou to programy „*Samostatný klient*“, „*Slave-klient*“ a „*Master klient*“. Pro všechny tři aplikace bylo využito vývojové prostředí *Flex* využívající objektový programovací jazyk *AS 3.0* a pro vybrané části s rozšířením o nástavbu *AIR*. Rozložení práce v projektu pomocí klientů je znázorněno na schématu, viz. obr. 21 *Master* klient a *Slave* jsou programy přímo zapojené do *DC* výpočtů, kdy *Master* přijme zadání, rozdává práci a vyhodnotí výsledek. *Samostatný* klient zadání zpracuje sám s delším časem výpočtu, ale větší pravděpodobností lepšího dosaženého řešení (zapříčiněno algoritmy rozdělení).



OBR. 21: Schéma rozdělení klientů

○ Samostatný klient

První část praktické části je *Samostatný* klient. Tento software byl naprogramován jako vývojové prostředí s aplikací příkladů pro *HC* algoritmus s využitím binárního (Gray(ova)) kódování. Příklady použité v programu pracují dle postupu z kapitoly 3 a jsou to obchodní cestující, problém batohu a funkční optimalizace. Program pracuje v *off-line* módu, a proto není zahrnut do distribuovaných výpočtů, jeho účel je výpočet programů dle definičního zadání a sběr dat z výpočtu příkladů. Modul je napsán jako desktop(ová) aplikace (spustitelná na *PC* pomocí *EXE* souboru). Hlavní silou programu je dobrá práce s *XML* daty, kdy program ukládá veškeré proměnné do tohoto formátu a je tedy schopný kdykoliv obnovit již jednou spuštěný výpočet, z externího souboru. Další předností programu je reprezentace výsledů pomocí vypočítané statistiky nebo graficky prostřednictvím grafů hodnot. Pro program byl vyvinut i tzv. *command* line (*CMD* line) k zadávání typu příkladu a specifikování způsobu výpočtu skrze okna editoru. Interface programu a ovládání se dá rozdělit na několik oblastí konkrétně na 5, viz. příloha 2.

- **Část 1:** informační část, k zobrazení aktuálního problému s popisem postupu výpočtu (výpis procent hotového výpočtu, počet iterací a aktuální „run“). Část 1 dále obsahuje tlačítka pro zapnutí/vypnutí výpočtu a spuštění skriptu v CMD line,
- **Část 2:** lišta menu pro ovládání celé aplikace, která se dělí na položky:
 - **File:** obsahující funkce jako načtení výpočtu, uložení výpočtu (ve formátu TXT, XML, XLS), uložení masky výpočtu a uzavření aplikace.
 - **Computation:** položka pro ovládání výpočtu příkladu z načtených dat, spouštění, zastavování a reset výpočtu, vypočítání statistiky pro zatím spočítané „run(y)“ tahle konkrétní položka se provede vždy při skončení výpočtu a při načtení nových dat,
 - **Command line:** řídí editor CMD line, spouštění napsaných skriptů, mazání editoru a načtení předdefinovaných příkladů.
 - **Tools:** spouštění externích aplikací jako jsou výpočet funkce pro n parametrů, konverze mezi číselnými typy a analýza binárního kódu z pohledu konkrétního příkladu, aplikace budou podrobně probrány dále v této kapitole.
 - **Properties windows:** přepínání mezi třemi základními okny programu, které jsou okno „Properties tree“ pro vizualizaci veškerých dat používaných programem. „Trace window“, kde se ukládají informace v průběhu výpočtu a „Previous projects“ se seznam všech automaticky uložených předešlých výpočtů ve složce programu. Projekty se automaticky ukládají po dokončení výpočtu a obsahují časový kód v názvu souboru ve formě (počet milisekund od data 1.1. 1970).
 - **Help:** místo se seznámením s programem a stručné info o něm.
- **Část 3:** editor CMD line pro psaní skriptů určující výpočet příkladu.
- **Část 4:** místo k zobrazení oken z položky Properties windows.
- **Část 5:** obsahující grafické ztvárnění výpočtů prostřednictvím grafů hodnot a základní nastavení grafu. Do grafu se ukládají aktuální funkční hodnoty pro každý „run“ (v podobě přímky) a počty unikátních výsledků z každého „run(u)“ (v podobě sloupového zobrazení).

Function probe

Function

Parameters

change actions fce :&3#3+&3#5
 Action 8. - var1: &3#3 var2: &3#5 action: +
change actions fce :&3#6

check info :Syntax success: rightparen 2 = 2
 leftparen.

actions :8
 Result: -2.4217679849853706

Function probe pracuje ve smyslu imaginární kalkulačky, kdy se nastaví typ funkce (funkční zápis je stejný jak u funkční optimalizace) a parametry (zápis *název[hodnota]*;) do ní vstupující a vypočítá hodnotu funkce. V komponentě typu *textArea* je kromě výsledku i znázorněno dekodování funkce a parametrů z datového typu *string* a výpis kroků nutných k výpočtu funkce.

Compute: provede samotnou analýzu a výpočet funkce v bodech.

Convert tool

Binar number

Gray coding

Decimal number

Složí k převodu mezi binárním číslem a desítkovou soustavou, při převodu se automaticky převádí i na *Gray(ovo)* kódování. Maximální délku binárního řetězce určuje celková přesnost v aktuálním příkladě.

Analyse tool

01110010100001111110

x1

x2

parmNum: 2

binLength: 20

binDecode: 0000000000-0000000000

fce value: 20.58274651891598

Random bin
Analyse
Set as iniBin

Analyse tool je určena k analýze binárního kódu podle pravidel v jednotlivém příkladě. Analýza probíhá pomocí dekodování z binárního čísla na skutečné vlastnosti jednotlivých parametrů dle aktuálního problému a podle těchto vlastností spočítá hodnotu příkladu.

Random bin: generuje náhodné binární číslo v celkové přesnosti.

Analyse: provede vlastní dekodování binárního čísla.

Set as iniBin: nastaví zvolené binární číslo jako výchozí bod k výpočtu.

• Skripty pro CMD line

Skripty slouží k nastavení hodnot ve výpočtu problému a naplnění XML souboru daty v sekci *settings*, jsou rozděleny do sedmi hlavních kategorií podle typu dat, které ovlivňují. Každá kategorie má několik funkcí rozdělených dle hodnot, které spravují. Způsob práce skriptů je pomocí analýzy datového typu *string*, kdy se text dělí od základní kategorie a potom až na funkční hodnoty. Kategorie jsou tyto:

- **parameter:** data určený pro definici parametrů, speciální pro každý typ úlohy zvlášť,
- **function:** nastavení typu problému k optimalizaci, obdobně jako *parameter* se definuje pro každou úlohu zvlášť,
- **apc:** data vztahující se k APC a způsob výběru výsledku,
- **optimization:** zpřesňuje data určené k optimalizaci,
- **specification:** dodatečné informace pro výpočet,
- **xmlLoad:** nahrání externích XML soubor z HTML adresy (využívá hlavně *Slave-klient*),
- **dc:** typ distribuce výpočtů a dodatečné informace kolem, částečně se definuje pro každou úlohu zvlášť, používá jen *Master-klient*.

optimization

<typ>	popis	formát hodnot
		Příklad
repeat	počet opakování konkrétní úlohy, tvar (repeat [počet];)	repeat [integer+];
		repeat[50];
find	typ extrému funkční hodnoty pro vyhledání, tvar (find [typ];), typ (max, min)	find [string];
		find[min];
mask	velikost generující masky od řádu 1 do <i>n</i> velikosti přesnosti, tvar (mask [velikost];)	mask [integer+];
		mask[min];
funType	způsob zaznamenávání dodatečných informací, tvar (funType [typ];), typ (test, quick)	funType [string];
		funtype[test];

specification

<typ>	Popis	formát hodnot
		příklad
funName	název počítané úlohy, tvar (funName [názevPříkladu];)	funName [string];
		funName[příklad_6];
iniBin	počáteční binární kód výpočtu, tvar (iniBin [kód];), typ (random, "binKód")	iniBin [string];
		iniBin[random];
iniType	určuje jestli při začátku výpočtu bude generovaný náhodný začátek nebo uživatelem vytvořený, tvar (iniType [typ];), typ (random, set)	iniType [string];
		iniType[random];

mCode	způsob kódování generující masky, tvar (mCode [typ];), typ (BC, Gray)	mCode [string]; mCode[BC];
funPrb	typ konkrétní úlohy výpočtu, tvar (funPrb [typ];), typ (fceOpt, knapsackOpt, tspOpt)	funPrb [string]; funPrb[fceOpt];

xmlLoad

<typ>	Popis	formát hodnot
		příklad
load	načtení externího XML souboru z definované adresy, tvar (load [adresa];)	load [string]; funName[html...];

dc

<typ>	Popis	formát hodnot
		příklad
chunks	počet opakování všech segmentů, tvar (chunks [počet];)	chunks [string]; chunks[3];
dcSpec	specifické rozdělení úlohy definuje se pro každou úlohu zvlášť	dcSpec [string]; dcSpec[-];
workTime	určuje celkový čas po který bude úloha viditelná pro klienty, tvar (workTime [čas];), čas [hod]	workTime [real+]; workTime[50];
clientTime	určuje celkový čas na jeden segment pro klienta, tvar (clientTime [čas];), čas [min]	clientTime [real+]; clientTime[35];
priority	nastavení priority pro konkrétní úlohu, tvar (priority [rozsah];), rozsah (0, 50, 100)	priority [integer+]; priority[50];
clientText	dodatečný info text pro klienty, tvar (clientText [text];)	clientText [string]; clientText[info];

○ Slave klient

Zbývající dva programy v této diplomové práci se už týkají samotných *DC* výpočtů, jsou to konkrétně *Master* klient a *Slave* klient. *Slave* klient se využívá v *DC* jako „výpočtový“ člen pro zpracování konkrétních zadaných příkladů, bez možnosti změny parametrů výpočtu. Dle filozofie *DC* výpočtů je výhodné zapojit do práce co nejvíce samostatných uživatelů, proto bylo zvoleno populární prostředí *flash* aplikace (AS 3.0). Naprogramované aplikace jsou dvojího druhu jednak spustitelné v okně prohlížeče bez nutnosti instalace, nebo aplikace na bázi desktop(ového) programu spustitelný pomocí vytvořeného instalátoru. Kvůli využití první možnost spuštění v okně prohlížeče prostřednictvím souboru *SWF* bylo nutno odebrat širší podporu *XML* souborů (ukládání dat na disk *PC* atd.).

Samotné výpočty lze spustit jen registrovaným uživatelům, možnost registrace je i pomocí *Slave* klienta. Po přihlášení uživatele se změří aktuální rychlosti *PC*, pomocí výpočtu velkého prvočísla (používá se u příkladů typu zátěžový test) a zpřístupní se modul výběru aktivní práce ze serveru.

● Podrobný popis Slave klienta

Celý program aplikace je rozdělen na pět položek rozdělených v záložkách. Všechny položky se zpřístupní až po úspěšném přihlášení uživatele, podrobný popis položek dále textu (obrázky jsou uloženy v příloze).

- **Záložka 1:** přihlášení/odhlášení registrovaného uživatele, hned po přihlášení zde proběhne test rychlosti PC po dobu 10 sekund s výpočtem kontroly prvočísla pro hodnotu 389 167,
 - jméno a heslo uživatele, nastavení adresy domácího serveru
- **Záložka 2:** registrace/update uživatele, v případě nového uživatele možnost registrace do databáze,
 - nepovinné parametry: jméno, příjmení, email, město, pohlaví a země,
 - povinné: uživatelské jméno, heslo v případě Slave klienta lze registrovat jen Slave uživatele,
 - pro přihlášeného uživatele zobrazení podrobných dat (datum reg., počet bodů z výpočtu, typ uživatele),
 - registrace/změna uživatele, žádost o smazání uživatele (tuhle činnost má právech jen Master klient,
- **Záložka 3:** výběr typu práce pro výpočet podle různých parametrů, zobrazení základních dat zadaných Master klientem,
 - všechny volné práce na serveru podle typu
 - ovládání výběru dat ze serveru, dodatečné informační prvky – status práce atd, připojení k práci, obnovení seznamu,
 - informace týkající se vybrané práce, rozdílné podle typu příkladu (přenášených dat z DB),
- **Záložka 4:** výpočtová část s oknem aktuálního výpočtu s podrobnými informacemi o komunikaci s databází a postupu výpočtu, okno je jiné na základě typu výpočtu (optimalizace/zátěžový test), podrobný popis dále v kapitole,
 - **zátěžový test (výpočtová část):** okno obstarává samotný výpočet komunikací mezi ostatními klienty přes rozhraní PHP,
 - okno se základními informacemi, které obsahuje počet výpočtů za sekundu – algoritmus reguluje velikost intervalu ke zpracování tak, aby se co nejvíce přiblížil hodnotě 20 výpočtů za sekundu, počet aktuálně připojených klientů ve stejné práci, přibližný čas výpočtu zrovna probíhající práce (není to celkový čas od začátku práce zadaného Masterem), rychlost počítače v době posledního předání intervalu na server
 - informace o komunikaci s databází zrovna zpracovávaného výpočtu – čas, kdy se naposledy provedla aktualizace databáze, údaj o výpočtu (hodnota kontrolovaného čísla), začátek a konec přiděleného intervalu k výpočtu, velikost rychlosti klienta, který žádá o přidělení práce, ID klienta žádajícího o přidělení práce, dodatečné informace k výpočtu, případně hlášení o chybách v komunikaci se serverem,
 - obsahuje data o ukončených výpočtech. Informace se skládá z pořadového čísla výpočtu a velikosti spočítaného intervalu. Tahle část dále obsahuje základní ovládací prvky modulu – start/stop a zobrazování dat,
 - poslední část udává v jakém pořadí se prováděli skončené výpočty v tzv.: thread,
 - **zátěžový test (statistická část):** se zpustí až po dokončení všech segmentů v daném úkolu (první klient který zjistí konec se prohlásí za Mastera uzavře úkol)
 - sestupné seřazení všech skončených intervalů z tabulky *_segments (* název práce), hodnota ID udává číslo řádku v tabulce pro interval,
 - okno s výpisem statistiky jako je (procenta mé odvedené práce, součet všech intervalů, součet vlastních intervalů),
 - legenda pro vizualizaci spočtených dat,
 - grafická vizualizace práce přihlášeného uživatele proti spočtené práci ostatních uživatelů,

- **HC výpočet:** pro zpracování z některého příkladu obsahujícího algoritmus HC se využívá zmenšený modul ze Samostatného klienta. Modul byl oproti Samostatného klienta změněn hlavně při implementaci XML exportu, který byl úplně odstraněn,
 - základní ovládací prvky (start/stop), náhled počtu hotových procent, doba do konce času pro klienta, pořadí aktuálního segmentu,
 - graf nejlepších hodnoty právě počítaného run(u) a název problému,
- **Záložka 5:** možnost spuštění externích programů ve formě SWF v okně Slave klienta

Kompatibilita externích aplikací

Pro spouštění externích *flash*(ových) aplikací v 5. části *Slave*-klienta lze použít jak programy psané v AS 3.0, tak i AS 2.0 a lze mezi nimi navázat i komunikaci. Jazyk AS 3.0 není zpětně kompatibilní, se starší verzí 2.0 v rámci funkce *LocalConnection* lze toto omezení obejít. Komunikace může být provedena jen v případě, že externí program je kompilovaný v souboru *swf* a na obou stranách je spuštěná funkce *LocalConnection*. Příklad využití funkce *LocalConnection* ze zdrojového AS 3.0 s komunikací funkcí v AS 2.0.

```

1|      // AS 3.0 kód
2|      private function LoadExt():void{
3|          incoming_lc.connect("lc_example");
4|          incoming_lc.client = this;
5|          loader_img.source = "ext_app/example.swf";
6|      public function methodToExecute(param:String):void{
7|          Trace.text += param;    // Trace text
8|          incoming_lc.close();
9|      // AS 2.0 kód
10|      var outgoing_lc:LocalConnection = new LocalConnection();
11|      window_mc.close_btn.onPress = function():Void{
12|          outgoing_lc.send("lc_example", "methodToExecute", "Trace text");

```

○ Master klient

Poslední a stěžejní program je *Master* klient zajišťující samotný chod *DC* systému prostřednictvím *DB*. Přihlásit se do programu mohou jen tzv. *Master* uživatelé, *Master* uživatel se může registrovat jen pomocí jiného *Mastera* buď upgradem *Slave* uživatele, nebo vytvoření nového s plným počtem pravomocí. Klient má absolutní kontrolu nad všemi částmi databáze proto není spouštěn kvůli bezpečnosti přímo z okna prohlížeče ale lokálně pomocí *EXE* souboru. *Master* má tři základní povinnosti.

- **Povinnost 1:** zadávání nových úloh do systému pomocí rozdělovacích algoritmů (v případě *HC* příkladů, zátěžový test ukládá jen prvotní nastavení).
- **Povinnost 2:** kontrola všech operací probíhajících v databázi, sledování času pro každou práci a uzavírání již ukončených prací.
- **Povinnost 3:** pro uzavřené práce provést stažení dat ze serveru a možnost uložení těchto dat do externích *XML* souborů popřípadě vytvoření výpisů ve formátu *XLS*, *TXT*.

Program je spouštěn pomocí instalačního souboru na *PC*, proto je zde zakomponována plná podpora *XML* rozhraní podobně jak u *Samostatného* klienta. Celá aplikace je rozdělena na čtyři části.

• Podrobný popis Master klienta

Podobné dělení programu jak u *Slave* klienta, ale tentokrát ve čtyřech záložkách. Pro nepřihlášeného (*Master*) klienta je zpřístupněna jen první záložka (login), další až po úspěšném přihlášení uživatele.

- **Záložka 1:** přihlášení/odhlášení registrovaného uživatele, hned po přihlášení zde proběhne test rychlosti PC po dobu 10 [s] s výpočtem kontroly prvočísla pro hodnotu 389 167,
 - jméno a heslo uživatele, jméno a nastavení adresy domácího serveru, spouštění a vypínání automatické kontroly hotové práce s nastavením časové periody (kontroluje se čas pro celkovou práci, po překročení se práce uzavírá).
- **Záložka 2:** správa prací zrovna uložených na serveru i tvorba nových zadání pomocí parametrů,
 - okno pro vizualizaci všech prací z databáze, okno se mění podle vybraného typu příkladu (HC/zátěžový test),
 - seznam všech prací v DB rozdělených dle typu, s možností obnovení seznamu a mazání vybraných částí,
 - strukturu rozdělení vybrané práce na segmenty, uložená v komponentě *tree*,
 - podrobný popis informací o aktuální práci a vybraném segmentu rozdělení, *progress* výpočtu s výpisem dosažených hodnot pro každý segment, cpu čas výpočtu (sečteny všechny časy výpočtu ze skončených segmentů), reálný čas výpočtu (čas od zahájení výpočtu), poslední část zobrazuje data podobně jak Samostatný klient pomocí XML struktury celého segmentu se všemi vlastnostmi dědicími z toho a vykresluje graf pro ní,
 - volitelné parametry k update práce, spolu s ovládacími prvky – uložení dat na harddisk (možnost volby XML / TXT / XLS typ souborů), aktualizování dat, vytvoření kopie vybraného segmentu, ovládání statutu práce (uzavírání / otevření), smazání vybraného segmentu,
 - základní informace o zrovna počítaném problému (segmentu) v podobě parametrů příkladu, grafické bannery udávající postup práce (časový, počet hotové práce),
 - tvorba nového optimalizačního příkladu pomocí HC algoritmu,
 - CMD line pro psaní skriptů se zadáním nového příkladu, stejný princip jak u Samostatného klienta, odemčena funkce dc viz popis CMD line,
 - uložené testovací příklady pro každý typ úlohy, spuštění skriptu,
 - tvorba nového zadání pro zátěžový test,
 - seznam všech prací v DB rozdělených dle typu, s možností obnovení seznamu a mazání vybraných částí,
 - parametry nového zátěžového testu,
- **Záložka 3:** ovládání všech uživatelských účtů, změna dat a přidělování privilegií pro vstup do Master klienta,
 - seznam všech uživatelů registrovaných v DB, s možností obnovení seznamu a mazání vybraných uživatelů,
 - nepovinné parametry: jméno, příjmení, email, město, pohlaví a země,
 - povinné: uživatelské jméno, heslo,
 - pro přihlášeného uživatele zobrazení podrobných dat (datum reg., počet bodů z výpočtu), typ uživatele (Master klient má právo registrovat nebo změnit Slave uživatele na Mastera),
 - registrace/změna uživatele, žádost o smazání uživatele,
- **Záložka 4:** všeobecný databázový manager k zobrazení všech dat uložených v kompletní databázi, manager obsahuje i funkce pro změnu dat pomocí SQL příkazů,
 - přihlášení do DB přes uživatelské jméno, heslo uložené na serveru (phpAdmin), adresa skriptu PHP, název serveru,
 - seznam všech uložených databází v systému serveru, tvorba a mazání databáze
 - seznam všech uložených tabulek ve vybrané databázi, mazání vybrané tabulky a tvorba nové pomocí parametrů,

- generovaný SQL příkaz pro získání dat z vybrané tabulky, příkaz je možno jakkoliv měnit dle SQL syntaxe, dodatečné informace ze systému,
- výsledná data získaná z předešlého SQL dotazu ve formě tabulky (data nelze ručně měnit).

• Struktura databáze

Struktura databáze v systému *DC* obsahuje minimálně 2 tabulky *aaa_users* a *aaa_work*. Tahle část je konstantní a neměnná ostatní tabulky s daty pro příklady jsou generovány pomocí *Master* klienta. Tabulka *aaa_users* obsahuje data všech uživatelů plus informace o zrovna připojené práci, v *aaa_work* jsou uložena data všech příkladů s názvy generovaných tabulek (údaj pro *Slave* klienta). Ostatní struktura se liší podle typu založeného příkladu, jedna věc je společná pro všechny typy, název tabulky, mají koncovku ve tvaru *_* milisekund od roku 1970 (univerzální čas), aby byla zaručeno unikátní jméno pro každou novou práci. Struktura *DB* je v příloze 1.

- **Zátěžový výpočet:** generuje jednu tabulku s názvem příkladu, kde jsou uloženy všechny nevypočítané intervaly, a další dvě s přídatkem *_segmets* a *_finish*. Tabulka *_segmets* obsahuje hotové intervaly s ID uživatele, *_finish* tabulka je vytvořena až na konci výpočtu s přepočítanými body na každého uživatele.
- **Příklad HC v1:** pro všechny druhy příkladů *HC* algoritmu rozděluje práci *Master* klient, první tabulka s názvem problému (typ optimalizace) obsahuje individuální nastavení dle algoritmů rozdělení. Druhá uchovává výsledná data. Data do tabulky se mohou ukládat jen o velikosti max 40 000 znaků proto některé větší výsledky je třeba rozdělit na několik datových struktur, *Master* má pak za úkol takto rozdělený úkol zas spojit dohromady.
- **Příklad HC v2:** poslední možností distribuce je pomocí dynamického rozdělení generující masky, metoda je pro všechny příklady v *HC* identická. Tento typ distribuce vytvoří dvě tabulky, jednu s inicializačními hodnotami a druhou pro ukládání výsledných hodnot iterací v podobě binárního kódu a hodnoty účelové funkce. *Master* klient zpětně tyto hodnoty ukládá do strukturovaného souboru *XML* dle zásad popsaných v kapitole 3.

Komunikace AS 3.0 s DB

Pro získávání dat z *DB* musí *flash* aplikace použít prostředníka v podobě *server-side* programovacího nástroje, v tomto případě jazyk *PHP*. Komunikace mezi *DB* je zakázána kvůli bezpečnosti, kdy aplikace je spuštěná ve *flash player(u)*, který má omezen přístup na externí zdroje dat. Samotné zapojení prostředníka probíhá pomocí objektu *urlVariables* na straně AS 3.0 a ve funkci *post* pro *PHP*. Následující kód je příkladem použití postupu, *AC* pošle text „hello“ do *PHP*, ten přidá „world“ ke stringu a pošle jej zpátky do *AC*, kde výsledek vypíše.

```
1| var urlVariables:URLVariables = new URLVariables();// as 3.0 kód
2| urlVariables.test = "hello";
3| var loader:URLLoader = new URLLoader();
4| var request:URLRequest = new URLRequest("testPHP.php");
5| request.method = URLRequestMethod.POST;
6| request.data = _urlVariables;
7| loader.load(request);
8| $test=$_POST['test']; // PHP kód
9| $test += ' world';
10| echo '&exportDATA='. $test;
11| Trace (URLLoader(event.target).exportDATA); // as 3.0 kód
```


KAPITOLA 4 POUŽITÉ TECHNOLOGIE

Softwarové prostředky využívané v této diplomové práci jdou rozdělit na čtyři základní technologie. První pro programování vrchní části aplikace (*Slave* klient, *Master* klient a *Samostatný* klient) byl použit AS 3.0 a další pro zprávu dat z databáze, kontorně SQL a skript komunikující se serverem PHP. Poslední technologie je k práci s výslednými soubory v XML formátu.

4.1 ACTIONSCRIPT

S prvními verzemi *flash(e)* se objevila potřeba interaktivně řídit animaci, kvůli tomu vznikl programovací jazyk *ActionScript* (zkráceně AS) od společnosti *Macromedia*. V té době se k podobnému účelu na webu hojně používal *JavaScript*, proto se *ActionScript* vydal v jeho stopách. Nejdříve šlo jenom o jednoduché příkazy k zastavení animace, přesunu na snímek apod. Později vyšla novější verze s označením 2.0, která už v sobě zahrnovala mnoho pokročilých vlastností, jako implementace objektového programování atd. AS 2.0 je možné používat k vytváření her, interaktivních prezentací a web prezentací. Aplikace napsané v AS jsou uloženy v souboru s koncovkou SWF. [15].

Soubor typu SWF je nejrozšířenější forma uložení *flash* aplikace, v takovém formátu má velmi malou velikost, ale k jeho běhu je nutný *flashPlayer*, lze přehrát v prohlížeči tímto kódem [15].

```
1| <embed src="game.swf" quality="high" width="500" height="500"
   type="application/x-shockwave-flash"
   pluginspage="http://www.macromedia.com/go/getflashplayer" />
```

Ke tvorbě vrchní aplikační vrstvy byla využita 3. verze jazyk AS, který také jako předešlé verze vychází ze standardizované verze programovacího jazyka *Java skript*, AS 3.0 je plně objektově orientovaný programovací jazyk, rozšiřující možnosti předešlých verzí dnes už pod hlavičkou *Adobe systems*, o interaktivní prvky a lepší ovládání videa atd., je vhodný pro tvorbu interaktivních webových prezentací a programů (za pomoci rozšíření AIR).

○ AS Verze 3.0

AS ve verzi 3.0 plně vyhovuje specifikaci danou standardem *ECMAScript*. Jazyk mnohem více zaměřená na objektové programování než jeho předchůdci, se kterými není zpětně kompatibilní. Poslední verze přináší jako první začlenění rozšíření AIR (možnost psát *desktop*(ové) aplikace). Vývoj verze AS byl zaměřen na dosažení těchto cílů:

- **Bezpečnost:** jazyk klade velkou váhu na bezpečnost používání dynamických skriptů v síti internet.
- **Jednoduchost:** AS 3 byl vyvinut s požadavkem co možná nejvíce zpřehlednit a zjednodušit programování v AS 1 a AS 2.
- **Rychlost:** verze 3. je mnohem rychlejší než jeho předchůdci (verze 2.0 a níž), kvůli přepracované objektové struktuře.
- **Kompatibilita:** jazyk podporuje kompatibilitu (AS 2.0 jde importovat do AS 3.0, ale zpětně tento proces nefunguje). Z důvodu postavení jazyka na standardech skriptovacího prostředí ECMA se rozšířily vlastnosti a použitelnost oproti AS 2.0. Velkým kladem nové verze je použití velké podpory XML souborů a práce s nimi [16].

○ Rozhraní AIR

AIR rozhraní je všestranný programovací nástroj pro různé systémy, momentálně existuje podpora pro systémy *windows*, *mac* a *linux*, který spojuje existující programovací jazyky do jednoho celku (*AS*, *HTML*, *JavaScript*). Myšlenka AIR je přivést internetové aplikace pro *desktop*(ové) požití.

- **Klady:** aplikace spouštěné prostřednictvím AIR (kompilované ve spustitelném souboru *EXE*) mají přístup na lokální souborový systém (ukládání, měnění souborů atd.), lepší práce s XML daty a vyšší rychlost skriptů.
- **Zápory:** nutnost kompilovat soubory s možností digitálního podpisu. Naproti tomu normální *flash* aplikace se spouští přímo v okně prohlížeče bez nutnosti dodatečné instalace.

K aplikaci ve formě *EXE* souboru je přibalen také přehrávač (výsledný soubor má o cca 1 MB víc), takže k běhu programu není potřebný externí *flashPlayer*, hodí se k publikaci her a prezentací na *CD*, nebo *DVD* [15].

Kompatibilita verzí

Podpora dopředné kompatibility (nahrání *SWF* se starší verzí do *AS 3.0* je zaručena). Není povoleno používat funkce vytvořené v *AS2* v programu psaném v *AS3*. V tomto případě je nutno se spolehnout jen a pouze na *LocalConnection* třídu, která umožňuje komunikaci mezi dvěma *SWF* soubory. Zde bariéra různých verzí *ActionScript(u)* odpadá. V následující tabulce je znázorněný typ kompatibility pro starší verze *AS* k *AS 3* a také podporované typy *flashPlayer(u)* [16]. Typy podporovaných operací jsou následující, viz. obr. 22.

Funkce		Prostředí	
Load swf kompilovaný pro .. Obsahuje AVM Spouští ac v swf	Runtime FlashPlayer ..		
	FP 7	FP 8	FP 9
	<= 7	<= 8	<= 9
	AVM 1 ac 1.0 a ac 2.0	AVM 1 ac 1.0 a ac 2.0	AVM 1 a AVM 2 ac 1.0 , ac 2.0 a ac 3.0
FP >= 9 Dokáže pracovat s .. Dokáže kombinovat ..	Kód vytvořený v ..		
	ac 2.0		ac 3.0
	ac 1.0 a ac 2.0 ac 1.0 a ac 2.0 **		ac 1.0, ac 2.0 a ac 3.0 ac 3.0 ***
	** ac 3.0 pouze přes objek LocalConnestion *** ac 1.0 a ac 2.0 pouze přes objek LocalConnestion		

OBR. 22: Kompatibilita předchozích verzí AS [16]

Syntaxe jazyka

Syntaxe jazyka *AS 3.0* je velmi podobná syntaxi *AS 2.0*, rozdíl je v ukládání základních objektů a volání *API* pro tvorbu tříd. Následující kód ukazuje rozdíl mezi verzemi na stejném příkladu volání textového pole.

```

1| // as 3.0
2| var greet:TextField = new TextField();
3| greet.text = "Hello World";
4| this.addChild(greet);
5| // as 2.0
6| createTextField("greet", 0, 0, 0, 100, 100);
7| greet.text = "Hello, world";

```

Software pro AS 3

Vývoj skriptů AS bylo počito programovací prostředí Adobe Flex 3.0 s podporou Adobe AIR. Vývojové prostředí Flex je více zaměřeno na programování internetových aplikací běžících na platformě flash player než na grafickou stránku animací, pro které byl flash původně navržen. Od 3. verze program podporuje i SDK AIR, jež přináší výhody flash aplikací i pro desktop(ové) aplikace a rozšiřuje podporu k ukládání a posílání souborů, které internetové aplikace kvůli bezpečnosti zakazovaly.

4.2 PHP

Prostředníkem v komunikaci s databází a aplikační vrstvou programu byl zvolen jazyk PHP (původně známý pod Personal Home Page), jenž vznikl v roce 1996, od té doby prošlo velkými změnami a nyní tato zkratka znamená Hypertext Preprocessor. PHP je to skriptovací programovací jazyk, který pracuje na straně serveru a je určen pro programování dynamických stránek. PHP se stalo velmi oblíbeným především díky jednoduchosti použití a tomu, že kombinuje vlastnosti více programovacích jazyků a nechává tak vývojáři částečnou svobodu v syntaxi [8]. Nejčastěji se PHP začleňuje jako součástí kódu stránek HTML, XHTML.

Webová stránka s prvky PHP má nejčastěji koncovku .php. Avšak je možné použít i .php3, php4, php5 a phtml. Stávající verze PHP je 5. Nejlépe je používat koncovky .php [17].

Funkce POST / GET

Z důvodu potřeb v rámci této práce je PHP využito jen pro přenos dat pomocí funkce POST a ovládání databáze pomocí SQL příkazů. Metodou POST se přenáší především objemná data a také citlivá data, u kterých je nevhodné, aby někdo viděl jejich obsah. Tato metoda se také používá při přenášení dat z formulářů. To nejen proto, že u některých z nich je nevhodné, aby byl vidět jejich obsah, ale také proto, že metoda POST je "od přírody" určena k přenosu větších objemů dat. Při použití sesterské metody GET se v URL stránky vždy objeví sekvence `jmenopromenne=hodnota` – pro případ odesílání formuláře s větším množstvím položek by byla nejen vidět některá citlivá data, ale také by adresa URL byla velmi dlouhá a některé údaje by se vůbec nemusely přenést [7].

Hlavní vlastnosti jazyka PHP

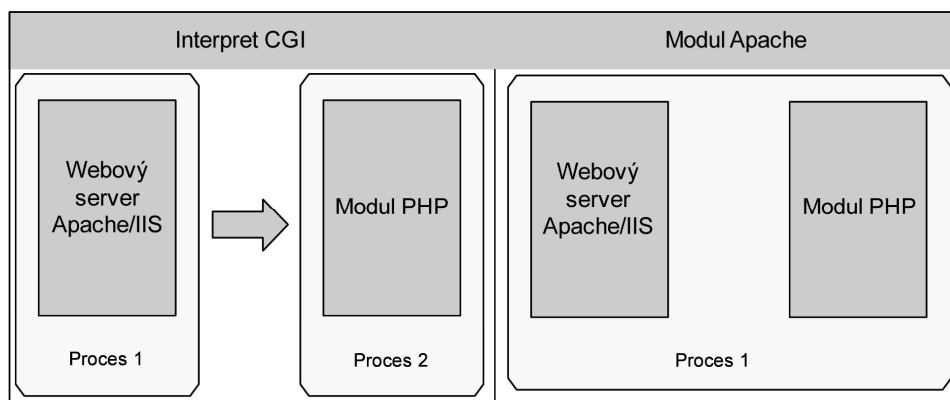
Výběr několika základních vlastností programovacího jazyka PHP.

- **Vlastnost 1:** Dynamický typový jazyk nespecifikuje přesně typ proměnné při deklaraci, ale až při přiřazení se provede rozhodnutí.
- **Vlastnost 2:** Kvůli dynamickému typovému systému má PHP dva druhy operátorů porovnání, první `,==` kontroluje obsah proměnných a `,===` kontroluje datový typ proměnných.
- **Vlastnost 3:** Jazyk je multi-systemový, lze jej spustit na různých systémech (windows, linux, mac), proto je to jeden z nejuniverzálnějších programovacích jazyků pro internet.
- **Vlastnost 4:** PHP je obsažen ve volné distribuci a lze k jeho editaci použít jakýkoliv textový editor [8].

PHP CGI / modul pro Apache

PHP se může zpracovávat (kompilovat) jako samostatný interpret CGI nebo jako součástí serveru Apache. V případě použití způsobu interpret CGI je po každém spuštění PHP vytvořen na serveru nová instance ke zpracování interpretu. Tento postup má za následek snížení rychlosti výkonu serveru. Připojení jako interpret vznikají i problémy s bezpečností (havárie kvůli chybě ve skriptu). Pomocí samostatných interpretů lze spouštět PHP s jinými identifikátory uživatelů [8].

Když je skript PHP zkompileováno v modulu pro *Apache*, tak celý proces běží ve stejném adresném prostoru jak sám server, není nutno vytvářet nové instance problému, vyšší výkon zpracování. Modul *Apache* má i jiné přednosti jako je trvalé připojení k databázi, spuštění probíhá vždy s uživatelem nastaveným jak *Apache* (výchozí uživatel je *nobody*). Způsob spouštění pomocí modulu *Apache* byl využit v této práci. Schéma je znázorněno na obr. 23.



OBR. 23: Způsob interpretace PHP [8]

Příklad zdrojového kódu

Příklad proměnné získané pomocí funkce POST z externího zdroje dat, v našem případě HML stránky a výpis pomněné pomocí PHP.

```
1| // HTML kód
2| <form action="process.php" method="POST">
   <input type="text" name="MyVar1">Hello World
   <input type="submit"></form>
3| // PHP skript
4| $myvar1 = $_POST['MyVar1'];
5| echo $myvar1;
```

Software pro PHP

Pro psaní skriptů PHP v této diplomové práci bylo využito programu Adobe Dreamweaver CS4, program je určen ke tvorbě internetových prezentací pomocí různých prostředků jako je CSS, ASP, PHP atd. Původně byl vyvíjen firmou *Macromedia*, v současné době spadá pod Adobe systems. Vývojové prostředí lze spustit jak pod *Windows* nebo *Mac* operačními systémy.

4.3 XML

Extensible Markup Language (zkráceně *XML*) je podporovaný W3C (*The World Wide Web Consortium*) jako standard pro značkovací jazyk. Definuje obecnou datovou strukturu s přehledným upořádání pomocí *tag(ů)*. Struktura jazyka byla vyvinuta z jazyků jako je *HTML*, *SGML*. Data uložená ve struktuře jsou ve formátu *string* a jsou obklopena *tag(y)* (s popisky zpřesňující data). Základní jednotka v datech je nazvaná *element*. Hlavní výhodou XML je struktura v tzv. metamarkup, to znamená neexistence přesně definovaných *tag(ů)* a *element(ů)*, vše je vytvářeno uživatelem. Soubory v XML jsou rozdělit na dvě základní odvětví [11].

- **XML zaměřené na data:** takové XML je používáné hlavně pro přenos dat, může to být výpis z databáze, nebo třeba pro ukládání *metadat* digitálního obsahu. Tento typ XML je použito v diplomové práci – ukládání dat při výpočtu a export dat z databáze.
- **XML zaměřené na dokumenty:** tento typ dokumentů se nejvíce podobá *SGML* typu dat. Jsou odvozeny od struktury určitých dat, které ukládají jako jsou knihy (použití kapitol), uživatelské manuály atd. Dokumentový formát je primárně určen pro čtení dokumentů než ukládání dat.

Příklad zdrojového kódu

XML specifikuje přesnou syntaxi, která se při psaní dat musí dodržet, jak jsou *element(y)* definovány v *tag* datech, jak *tag* vypadá, jaké názvy *tag(ů)* a *element(ů)* jsou přípustné, kde jsou umístěny atributy atd. XML struktura se velmi podobá HTML struktuře, ale obsahuje zásadní změny. Následující příklad ukazuje způsob zaznamenání dat ve fiktivní skladovací DB, Označení dat pomocí *tag(ů)* a parametry popisující každý unikátní předmět v DB [11].

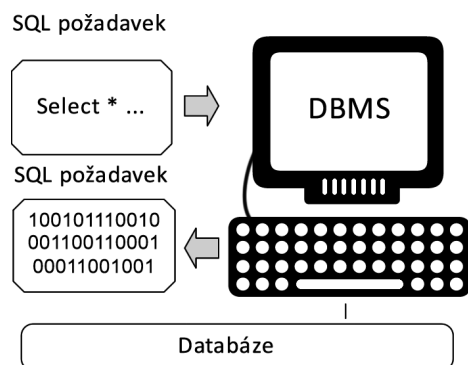
```
1| <?xml version="1.0"?>
2| <product barcode="2394287410">
3|   <manufacturer>Verbatim</manufacturer>
4|   <name>DataLife MF 2HD</name>
5|   <quantity>10</quantity>
6|   <size>3.5"</size>
7|   <color>black</color>
8|   <description>floppy disks</description>
9| </product>
```

Software pro XML

Hlavní výhodou struktury je její otevřenost vycházející z SGML, tedy ke psaní kódu lze využít libovolný textový editor. Lze použít i mnohem lepší a speciálně určený software pro tvorbu, ale ten je mnohdy placen. Pro účely diplomové práce byl využit volně šiřitelný program XMLmind 4 personal edition. Pro možnost vizualizace XML je vhodný jakýkoliv internetový prohlížeč (Internet explorer).

4.4 SQL

Structured Query Language (zkráceně SQL) je nástroj k organizování, získávání a zprávu dat uchovaných v počítačové databázi. Ze své podstaty je SQL jazyk pro interakci s databází. Podporovaný typ DB se nazývá *relational DB*. První standardizace pomocí ANSI v roce 1986, o něco později 1987 byla přijata pravidla pomocí ISO.



Princip činnosti SQL znázorňuje obr. 24. Počítačový systém v našem případě vlastní DB, která v sobě sdružuje konkrétní data. Program ovládající DB je *database management systém* (zkráceně DBMS). Vyzvednutí dat z DB se volá SQL požadavek, DBMS dotaz zpracuje, vyzvedne data a pošle zpátky uživateli. Tento proces se nazývá *db query* [10]. SQL má víc povinností než jen získávání dat (nejdůležitější funkce jazyka), ale ovládá i všechny funkce DBMS, jako jsou:

OBR. 24: Princip činnosti SQL [10]

- **Definice dat:** pomocí SQL uživatel definuje struktura a organizaci uložených dat v DB a vztahy mezi nimi.
- **Získávání dat:** SQL zpřístupní uživateli nebo programu cestu k získání požadovaných dat z DB.
- **Manipulace s daty:** zprostředkování uživateli nebo programu zprávu uložených dat jako je update DB, mazání starých dat a přidání nových.
- **Ovládání přístupu:** omezuje přístup ke všem privilegiím v DB jen konkrétním uživatelům.
- **Sdílení dat:** koordinuje sdílení dokumentů všem uživatelů, tak aby si navzájem nevadily.
- **Integrita dat:** SQL definuje integritu DB, chrání ji proti narušení v důvodu nesprávných zásahů uživatele [10].

Syntaxe SQL

Příklad syntaxe jazyka *SQL*, který provede výběr z tabulky (Book) a výsledek zapíše ve formě tabulky řazený pomocí názvu (title), začleněno do *PHP* kódu.

```
1|      // SQL query
2|      $myvar1 = 'SELECT Book.title, count(*) AS Authors FROM Book JOIN
      Book_author ON Book.isbn = Book_author.isbn GROUP BY
      Book.title';
3|      //PHP kód
4|      mysql_query($myvar1) or die('&query=fail');
```

4.5 WAMP

V diplomové práci k činnosti severu byl zvolen volně šířitelný program WAMP, aplikace je balení tří nezávislých programů, které spolu tvoří hlavní strukturu serveru. Spolupráce těchto programů v rámci WAMP umožňuje spouštět dynamické stránky v síti internet nebo v lokální síti. Program WAMP se skládá z těchto částí:

- **Apache:** hlavní program spravující činnost serveru, spustitelný pod různými operačními systémy (linux, windows, Unix, Mac atd).
- **MySQL:** program pracující na straně serveru podporující práci dat s několika zdroji zároveň, MySQL používá pro svou činnost databázový jazyk SQL. MySQL je napsáno v jazyku C++.
- **PHP:** interpret dynamického jazyka upravující skripty PHP, fungující dle principu z obr. 24.

KAPITOLA 5 PRAKTICKÉ TESTY

V této kapitole budou popsány provedené série výpočtů testovacích funkcí v algoritmu *HC* kvůli porovnání výkonu vytvořených programů. Testy budou jak pomocí distribuovaných výpočtů tak i bez nich a jsou zaměřeny na měření schopnosti nalezení nejlepšího řešení, dobu výpočtu při požití různých nastavení základní funkce (změna přesnosti, použití *APC*, atd.). Závěrem této kapitoly budou srovnány distribuované výpočty oproti klasickému výpočtu v *samostatném* klientovy. Testováno bylo provedeno pro každou úlohu se 100 násobným opakováním, v tabulce je vždy 10 vybraných hodnot.

○ Testovací funkce

K účelům testování byly využity různé funkce. V rámci funkční optimalizace byla zvolena funkce $F_6(x)$ viz. vzorec (3.6) a $F_7(x)$ viz. vzorec (6.1) pro různý počet parametrů.

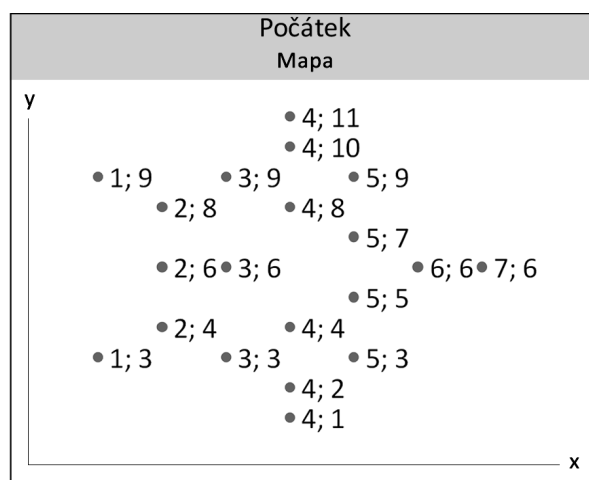
$$F_7(x) = \sum_{i=1}^n \left(-x_i \cdot \sin(\sqrt{|x_i|}) \right) \quad (6.1)$$

$$F_7(x) \quad x_i \in [-500, 500], i \in N$$

Problém obchodního cestujícího je zastupován rozložením bodů T20 o 20 městech s kombinační náročností a přibližně $2,433 \cdot 10^{18}$. Problém batohu je ve formě batoh1000 o 20 unikátních věcech při celkové počtu 241 a maximální nosnosti batohu 1000 jednotek.

Batoh	Počátek				Věci	V	C	P
	Věci	V	C	P				
max nosnost 1000	x1	12	4	8	x11	9	6	30
	x2	2	2	5	x12	5	2	25
	x3	1	2	9	x13	1	13	5
	x4	1	1	9	x14	1	9	9
	x5	4	10	10	x15	10	10	11
	x6	8	10	16	x16	8	5	16
	x7	4	4	8	x17	14	6	13
	x8	5	5	10	x18	2	5	20
	x9	4	9	8	x19	4	7	12
	x10	8	6	7	x20	2	6	10

OBR. 25: Problém batohu příklad B1000



OBR. 26: TSP příklad T20

Testovací sestava

Měření výpočtu běžících v DC systému, konkrétně test 4 a test 5 byla testována na jedno až třech různých *PC*, zbývající testy byly měřeny na *PC* 1.

- **PC 1:** Intel mobile core 2 duo T7200 (2 GHz), 4 GB DDR2 RAM.
- **PC 2:** Intel core 2 Q9400 (2.66 GHz), 4GB DDR2 RAM.
- **PC 3:** AMD Athlon (tm) XP 2500+ (1,84 GHz), 1,5 GB DDR1 RAM.

5.1 TEST 1: SAMOSTATNÝ KLIENT PŘI RŮZNÉM ZPŮSOBU NASTAVENÍ

Při průběhu samostatného klienta lze spustit funkci k ukládání některých dodatečných informací výpočtu jako je sledování všech unikátních bodů, počítání všech navštívených bodů atd. Tato funkce je užitečná pro statistické údaje, ale má za následek zpomalení výpočtu. V normálním stavu se zaznamenávají jen výpočetní stavy a dosažené koncové body v iteraci. Poslední varianta je bez konvertoru rovnic ve funkční optimalizaci, kdy se matematické funkce do programu zavádějí pomocí datového typu *string* a tento konvertor je pak převede na sérii instrukcí v datovém typu *array*, při odstranění konvertoru a napsání úlohy tzv. „natvrdo“ do zdrojového kódu odpadá v každém výpočtovém kroku procházení funkčních instrukcí, změna rychlosti je nejvyšší pro složité funkce s velkou generující maskou. Test je prováděn na funkci $F6(x)$ s 2 parametry ve střež kombinacích (10, 12, 14 bitů na parametr).

Nastavení:

```
nParam = 2, nBitParam = -, funName = F6, funOpt = min, nParam = <-5,5>
nRun = 100, funType = -, HC = 12, mode = BC
```

Průběh 1 (10 nBitParam)

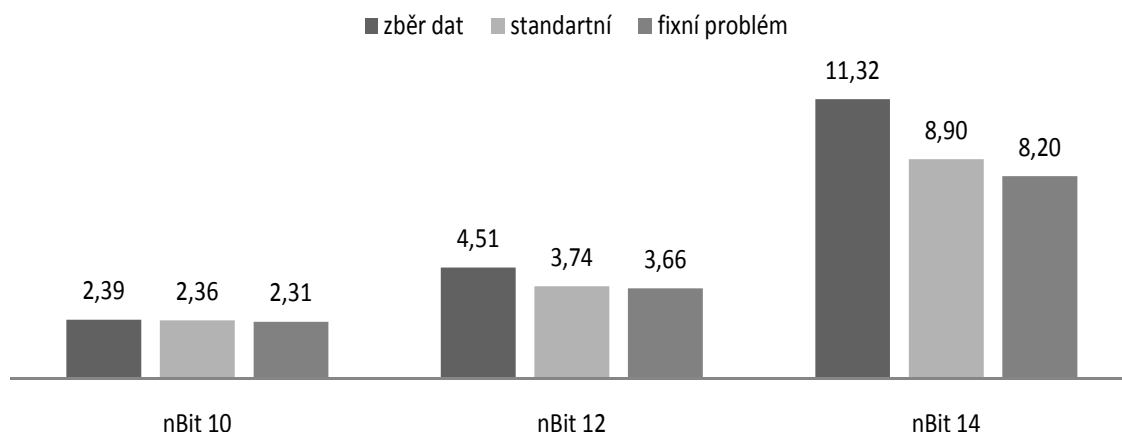
číslo měření	1	2	3	4	5	6	7	8	9	10	průměr
sběr dat	2,97	1,69	1,16	3,07	2,45	2,44	2,63	2,32	2,83	2,32	2,39
standardní	2,51	2,05	1,56	3,11	2,57	2,19	3,15	2,13	1,58	2,72	2,36
fixní problém	2,53	2,64	2,66	1,63	2,82	1,63	2,14	2,16	2,20	2,72	2,31

Průběh 2 (12 nBitParam)

číslo měření	1	2	3	4	5	6	7	8	9	10	průměr
sběr dat	4,88	3,70	4,69	4,74	4,76	4,92	4,80	4,80	4,83	2,96	4,51
standardní	2,29	3,77	4,72	3,91	3,14	3,92	3,81	3,92	4,77	3,17	3,74
fixní problém	4,33	3,63	2,82	2,91	2,99	3,52	5,04	2,86	4,29	4,25	3,66

Průběh 3 (14 nBitParam)

číslo měření	1	2	3	4	5	6	7	8	9	10	průměr
sběr dat	10,20	11,71	8,64	12,19	12,00	12,15	10,57	14,26	10,81	10,71	11,32
standardní	10,26	10,31	7,85	8,11	6,91	11,17	9,39	10,60	6,48	7,90	8,90
fixní problém	8,70	10,19	9,87	8,56	9,58	4,85	7,21	6,22	7,23	9,61	8,20



OBR. 27: Graf Změny rychlosti v samostatném klientu

5.2 TEST 2: APLIKOVÁNÍ RŮZNÝCH TYPŮ VÝPOČTU

Další test porovnává rozdíly mezi různými nastaveními funkční optimalizace. První běh je standardní nastavení s fixními parametry, v následujících už jsou použity algoritmy APC, konkrétně první je z hlediska změny vzorkování a druhý změny rozsahu problému s hlídáním hranic. Ve výsledku lze vidět, že použití APC je časově výhodnější pro nalezení konečného řešení než fixní nastavení. Nejhorší v průměrném čase dopadl APC 2, a to kvůli velkému množství iterací na jeden run (zde má nejlepší hodnotu, kvůli nejnižšímu nBitParam).

Nastavení

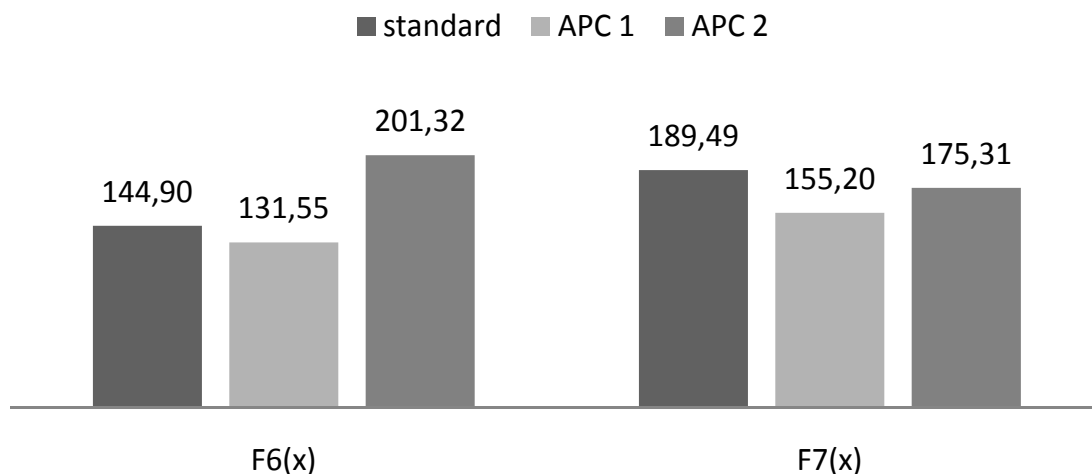
```
F6(x): nBitParam = 14, funName = F6, funOpt = min, nParam = <-5,5>
F7(x): nBitParam = 14, funName = F7, funOpt = min, nParam = <-500,500>
APC1 : nBitParam = 8, binRun = 3, binUpgrade = 2
APC2 : nBitParam = 10, rangeRun = 3, range% = 10, checkBoundries = true
      nParam = 5, nRun = 100, funType = quick, HC = 12, mode = BC
```

F6(x)

bestFitness	max.	min.	průměr	medián	směr. od.	čas
standard	14,41349	0,00009	5,26808	4,28647	4,02733	144,90
APC 1	15,91943	0,00009	5,55246	5,62552	3,35228	131,55
APC 2	14,74955	0,00011	4,22816	3,97987	3,06400	201,32
iterace	max.	min.	průměr	směr. od.	medián	čas
standard	23	10	16,53	2,66	17	8,77
APC 1	26	16	20,53	2,10	21	6,41
APC 2	60	18	38,72	9,52	29	5,20

F7(x)

bestFitness	max.	min.	průměr	medián	směr. od.	čas
standard	-1273,77725	-2094,91431	-1856,44715	-1860,56377	215,37725	189,49
APC 1	-1385,41015	-2094,91087	-1908,22153	-1976,47323	192,71103	155,20
APC 2	-1530,37222	-2094,91254	-1919,10238	-1976,45920	152,59771	175,31
statistika	max.	min.	průměr	směr. od.	medián	čas
standard	35	13	18,17	3,74	152,60	10,43
APC 1	39	13	20,99	3,87	19	7,39
APC 2	44	18	26,13	4,18	30	6,71



OBR. 28: Graf Průměrné rychlosti výpočtu F6(x) a F7(x)

5.3 TEST 3: SCHOPNOST NALÉZT ŘEŠENÍ

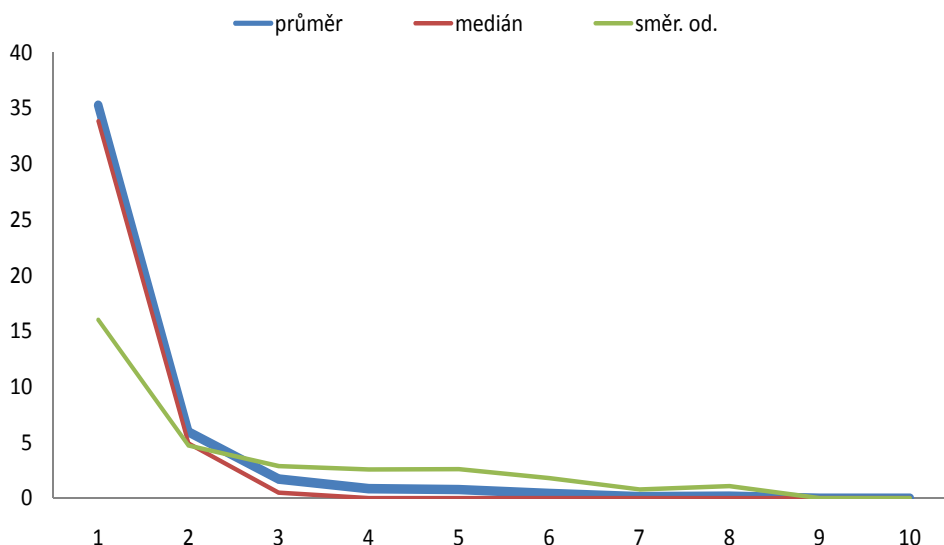
Test 3 se zabývá schopností nalézt optimální řešení pro konkrétní typy úloh, viz kapitola 3, první běh je funkční optimalizace funkce $F6(x)$, batoh b1000 a TSP t20. Každý z výpočtů byl prováděn pro 100 opakování a do výsledné tabulky byly uloženy statistické údaje v průběhu iterací. Výsledkem testu je graf kde na osy x jsou iterace a na ose y je průměr, medián a statistická odchylka.

Nastavení

$F6(x)$: nParam = 2, nBitParam = 14, funOpt = min, nParam = <-5,5>
T20: problem = 3, funOpt = min, nBit = 58
B1000: problem = 1000, funOpt = max, nBit = 82
nRun = 100, funType = quick, HC = 12, mode = BC

F6(x)

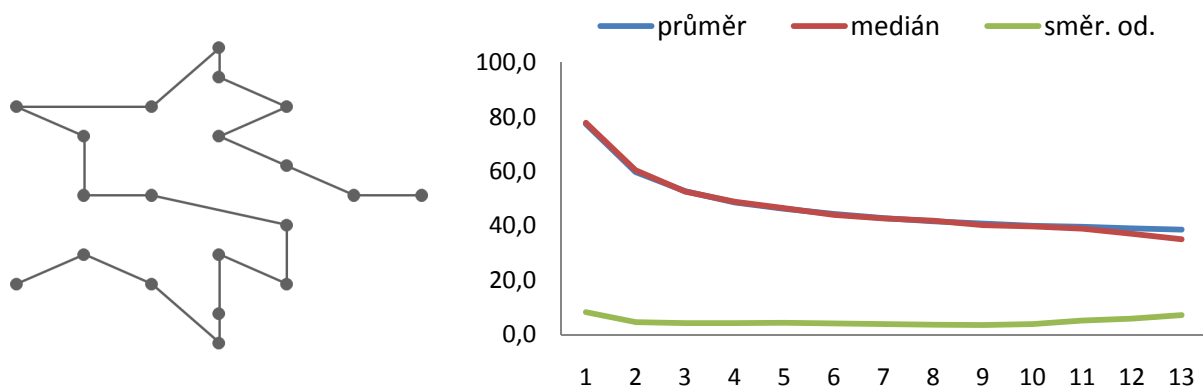
iterace	1	2	3	4	5	6	7	8	9	10
max.	87,483	21,886	14,019	14,014	14,014	13,999	5,9995	5,9995	5E-06	9E-07
min	2,0655	0,0081	0,0002	9E-07	2E-07	2E-07	2E-07	2E-07	2E-07	2E-07
průměr	35,286	5,9414	1,7117	0,8598	0,7878	0,4384	0,135	0,1935	1E-06	6E-07
medián	33,838	4,9032	0,4832	0,0071	0,0004	4E-05	3E-06	2E-06	2E-07	6E-07
směr. od.	16,022	4,7267	2,8772	2,5756	2,6094	1,809	0,7896	1,0775	2E-06	5E-07



OBR. 29: Graf nalezeného řešení $F6(x)$

T20

iterace	1	2	3	4	5	6	7	8	9	10	11	12	13
max.	94,3	70,2	62,2	59,9	58,4	58,1	56,2	55,5	53,9	52,8	52,4	51,8	51,5
min	58,0	48,5	42,6	40,3	37,2	35,6	33,8	33,6	34,2	32,5	30,0	32,2	32,8
průměr	77,5	59,8	52,8	48,6	46,2	44,4	42,8	41,7	40,8	39,9	39,6	39,0	38,6
medián	77,9	60,5	52,7	48,8	46,5	44,1	42,8	42,0	40,3	39,8	39,0	37,1	35,1
směr. od.	8,2	4,6	4,2	4,3	4,4	4,2	3,9	3,8	3,6	3,9	5,2	5,9	7,2

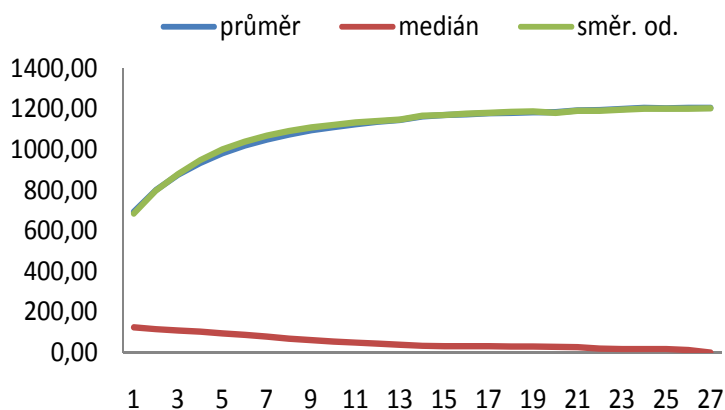


OBR. 30: Graf nalezeného řešení T20 s řešením

B1000

iterace	1	2	3	4	5	6	7	8	9
max.	931	1073	1133	1151	1163	1183	1193	1199	1219
min	404	554	625	685	735	777	819	859	899
průměr	692,60	799,50	875,35	934,31	980,89	1018,78	1048,97	1074,51	1094,80
medián	123,77	115,24	108,11	101,99	94,34	86,33	78,01	68,86	60,87
směr. od.	685,50	798,50	880,50	949,50	1001,00	1039,50	1069,50	1091,50	1109,00
iterace	10	11	12	13	14	15	16	17	18
max.	1233	1235	1236	1204	1221	1223	1225	1232	1240
min	938	974	1004	1034	1063	1076	1088	1098	1107
průměr	1110,94	1124,28	1136,83	1145,54	1163,16	1169,70	1174,04	1178,50	1181,16
medián	54,58	48,83	43,78	39,02	33,70	31,80	31,96	31,01	29,53
směr. od.	1122,50	1134,00	1142,00	1149,00	1167,00	1171,00	1177,00	1181,50	1187,00
iterace	19	20	21	22	23	24	25	26	27
max.	1241	1236	1245	1221	1233	1236	1229	1221	1204
min	1116	1125	1134	1165	1170	1174	1177	1192	1204
průměr	1184,25	1184,65	1193,21	1194,39	1200,29	1204,85	1203,13	1204,60	1204,00
medián	28,92	27,63	26,35	18,63	17,81	17,57	16,59	12,01	0,00
směr. od.	1189,00	1182,00	1192,50	1191,50	1196,50	1202,00	1202,50	1203,00	1204,00

Věci	P	Věci	P	Věci	P
x2	5	x8	10	x15	11
x3	8	x9	8	x16	14
x4	5	x10	7	x17	2
x5	10	x11	30	x18	20
x6	16	x13	5	x19	12
x7	7	x14	9	x20	10



OBR. 31: Graf nalezeného řešení B1000 s řešením

5.4 TEST 4: RYCHLOST DC SYSTÉMU DLE MASKY

Test 4 pojednává o měření rychlosti algoritmu *DC* dle dělení generující masky, celý test je prováděn postupně pro tři různé úlohy v rámci *HC* algoritmu. V každém z úloh je srovnání zlepšení času výpočtu oproti příkladu výpočtu v *samostatném* klientovi, tedy bez nutnosti zasahovat do *DB* a *DC* systém pro jedno až tři nezávislé *PC*. Vliv *DB* je patrný na rozdíl jednoho *PC* a *samostatném* klienta, kdy by měli dosahovat podobných hodnot, ale kvůli komunikaci v síti vzniká časové prodlení, tohle prodlení se zmenšuje se zvyšující se náročností výpočtu, jak jde vidět na ostatních případech, je tedy i vhodné používat tento typ *DC* distribuce.

Nastavení

F6(x): nParam = 2, nBitParam = 14, funOpt = min, nParam = <-5,5>
T20: problem = 3, funOpt = min, nBit = 58
B1000: problem = 1000, funOpt = max, nBit = 82
nRun = 100, funType = quick, HC = 12, mode = BC

F6(x)

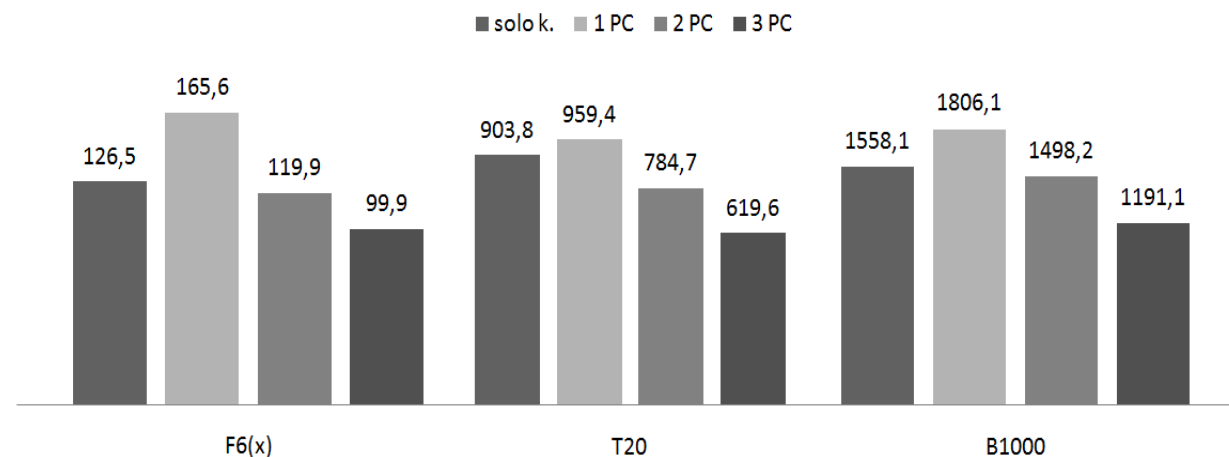
měření	1	2	3	4	5	6	7	8	9	10	průměr
sam. k.	112,61	133,28	108,69	125,18	125,10	115,13	135,44	141,81	142,87	124,73	126,5
1 PC	149,46	148,33	199,52	203,73	144,90	149,82	149,43	150,33	182,31	178,43	165,6
2 PC	112,61	125,10	108,69	125,18	141,82	83,42	117,50	125,69	131,13	124,73	119,9
3 PC	70,94	108,36	98,13	81,25	101,26	98,46	120,36	95,32	115,38	109,38	99,9

T20

měření	1	2	3	4	5	6	7	8	9	10	průměr
sam. k.	812,52	755,19	784,98	985,64	792,42	876,75	1038,20	961,98	946,18	1083,79	903,8
1 PC	955,25	1021,89	867,55	914,34	1203,31	897,87	1005,82	819,12	862,29	1046,34	959,4
2 PC	641,38	795,51	587,89	819,87	854,46	749,72	865,13	859,35	779,54	893,78	784,7
3 PC	648,42	593,71	635,82	521,96	695,65	764,46	533,76	653,55	610,23	538,85	619,6

B1000

měření	1	2	3	4	5	6	7	8	9	10	průměr
sam. k.	1658,4	1981,1	1376,0	1533,3	1762,7	1578,7	1469,7	1637,3	1462,9	1121,2	1558,1
1 PC	1964,5	2003,9	1948,6	1753,1	1856,3	1711,8	1899,5	1702,2	1566,3	1654,9	1806,1
2 PC	1616,1	1807,6	1324,2	1590,3	1400,5	1353,4	1391,1	1480,7	1436,9	1580,8	1498,2
3 PC	1058,3	1141,3	1233,3	1378,9	1042,7	1397,3	1154,8	1264,9	1040,0	1199,2	1191,1



OBR. 32: Graf snížení výpočtového času při dělení masky

5.5 TEST 5: RYCHLOST DC SYSTÉMU DLE INDIVIDUÁLNÍHO ROZDĚLENÍ

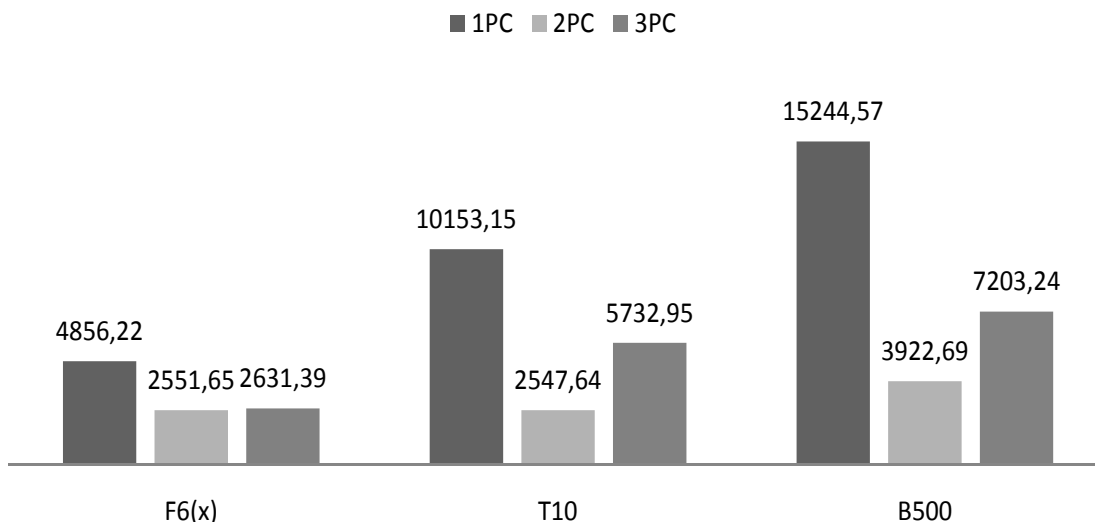
Test číslo pět je zaměřen na testování rychlosti druhého *DC* principu v rámci *HC* algoritmu, konkrétně konstantní rozdělení pro každé úlohy zvlášť pomocí *Master* klienta. Kvůli smyslu tohoto *DC*, kde se nedělí výpočet, ale použitá přesnost je přenášena na vytvořené segmenty, byly testovací úlohy (B1000 a T20) zmenšeny na polovinu svých bodů. Naměřené hodnoty jsou pro celkový čas výpočtu práce a průměrný čas na jeden vytvořený segment pro jednotlivé *PC*. Z hodnot lze vidět, že tento princip je náchylný na tzv. „nejslabší článek řetězu“, kde první měření je čas celé práce potřebný k výpočtu na jednom *PC*, druhé měření je výpočet prvního a druhého *PC*, kdy druhé *PC* je asi čtyřnásobně rychlejší než první, a tedy si postupně zaregistruje tři ze čtyř vytvořených segmentů. Poslední měření je pomocí třech *PC*, kde jedno hodně rychlé a dvě čtyřnásobně pomalejší a zde dochází právě ke zpomalení času v důsledku registrace práce pro všechny *PC* a ve výsledku je výsledek horší než u druhého měření.

Nastavení

F6(x): nParam = 2, nBitParam = 14, funOpt = min, binRun = 3,
binUpgrade = 2 nParam = <-5,5>
T10: problem = 1, funOpt = min,
x1[12,4,8];x2[2,2,5];x3[1,2,9];x4[1,1,9];x5[4,10,10];x6[8,10,16];
x7[4,4,8];x8[5,5,10];x9[4,9,8];x10[8,6,7];
B500: problem = 500, funOpt = max,
x1[2,2];x2[2,4];x3[2,6];x4[4,2];x5[4,4];x6[4,6];x7[6,2];x8[6,4];
x9[6,6];x10[8,2];x11[8,4];x12[8,6];

Individuální DC

měření	Celkový čas práce			Průměrná rychlost na segment		
	1 PC	2 PC	3 PC	1. PC	2. PC	3. PC
F6(x)	4856,22	2551,65	2631,39	1220,70	388,53	1342,77
T10	10153,15	2547,64	5732,95	2535,46	757,27	2594,83
B500	15244,57	3922,69	7203,24	3836,16	1154,95	3660,31



OBR. 33: Graf potřebného času při individuálního DC

ZÁVĚR

Cílem této diplomové práce bylo vytvoření internetových klientů pro distribuované výpočty s využitím variant algoritmu *HC*, resp. varianty *HC12* pro různé typy úloh. Ke splnění práce bylo potřeba získat dovednosti v programování robustních internetových aplikací a práci s *DB*. V této práci byl jako základ použit programovací jazyk *ActionScript 3.0*, dále *SQL* ke komunikaci s databázovými systémy, *XML* kvůli ukládání dat jak z *DB*, tak k vytvoření celé datové struktury programů. V neposlední řadě bylo nutno zakomponovat *PHP* jazyk kvůli nutnosti přímé komunikace mezi programy. Jak již bylo naznačeno v úvodu, celá práce je rozdělena na tři základní části.

V první části byly obsaženy vybrané testovací úlohy řešené s využitím *HC* algoritmu s různou parametrizací úlohy i **řešiče**. Dále byly v této části vytvořeny algoritmy k distribuci samotných výpočtů. Realizovaná distribuce výpočtů je dvojího druhu: první typ distribuce rozděluje řešení stejné optimalizační úlohy na více procesů (segmentů), s cílem urychlit výpočet, druhý typ distribuce rozděluje řešení dané úlohy na podúlohy, s cílem zpřesnit a paralelizovat řešení úlohy.

Druhá část práce je věnována programové oblasti, konkrétně naprogramování všech výpočtových klientů. Celkově byly vytvořeny tři typy klientů v závislosti na účelu použití, jsou to *Slave*, *Master* a *Samostatný* klient. *Slave* je výpočtový uživatelský klient používaný ke zpracování distribuovaných výpočtů v rámci sítě internet, *Master* klient se používá ke zprávě *DB* systému a ovládání *DC* výpočtů. Poslední z programů je *Samostatný* klient, jehož účel je klasický výpočet všech úloh a vyhodnocení statistických dat v průběhu výpočtu, spolu s grafickým ztvárněním výsledů. Jako jediný program není zapojený do *DC*. Všechny programy jsou vytvořené v technologii *flash* ve spojení s *AIR* platformou ke spouštění pomocí instalačního souboru, jen *Slave* klient je ve dvou verzích jak *AIR*, tak klasická *flash* aplikace, kvůli otevření v okně internetového prohlížeče. Celá datová struktura programů je vystavěna okolo *XML* dat s ohledem na možnost kdykoliv zastavit, uložit a zpětně obnovit výpočet.

Poslední část práce prezentuje výsledky z řešení diskutovaných optimalizačních úloh jak v rámci samostatných, tak distribuovaných výpočtů. Celkově z testů vyplývá, že použití distribuovaných výpočtů snížilo dle předpokladů výsledný čas výpočtu.

Vytvořená diplomová práce obsahuje veškeré body zadání. Předložené řešení plně dostačuje v oblasti řešené problematiky v rámci práce. Výsledkem jsou vyhotovené programy a algoritmy pro distribuci výpočtů. V rámci další práce je umožněn rozvoj výsledných programů, a to v podobě vylepšení použitých algoritmů rozdělení a správy komunikace mezi klienty.

SEZNAM POUŽITÉ LITERATURY

- [1] MATOUŠEK, Radomil. *METODY KÓDOVÁNÍ*. [s.l.], 2006. 96 s. Oborová práce. FRVŠ 3420/2006. Dostupný z WWW: <<http://www.uai.fme.vutbr.cz/~matousek/TIK/>>.
- [2] MINAŘÍK, Jan. *Evoluční algoritmy : matlab gate toolbox*. Brno, 2007. 65 s. Vedoucí diplomové práce Radomil Matoušek. VUT FSI ÚAI.
- [3] MATOUŠEK, Radomil. HOROLEZECKÝ ALGORITMUS A ÚLOHA CELO. PROG. In *Ústav přístrojové techniky, AV ČR*. [s.l.] : [s.n.], [200-?]. s. 2.
- [4] VYKOUŘIL , Dušan. *BOINC - počítače všech zemí, spojte se* [online]. 2008 [cit. 2009-01-20]. Dostupný z WWW: <<http://pctuning.tyden.cz/>>.
- [5] MICHÁLEK , Lukáš, VYKOUŘIL , Dušan . *Folding@home* [online]. 2008 [cit. 2008-02-04]. Dostupný z WWW: <<http://www.czechnationalteam.cz/>>.
- [6] Folding@home . *Folding@home PS3 FAQ* [online]. 2008 [cit. 2009-02-05]. Dostupný z WWW: <<http://folding.stanford.edu>>.
- [7] ROZSYPAL, Petr . *Metody POST a GET* [online]. 2006 [cit. 2009-03-29]. Dostupný z WWW: <<http://php.interval.cz/clanky/metody-post-a-get/>>.
- [8] CASTAGNETTO, Jesus, et al. *Programujeme PHP : profesionálně*. [s.l.] : [s.n.], 2002. 650 s. ISBN 80-7226-310-2.
- [9] L. APPLGATE, David, et al. *The Traveling Salesman Problem..* [s.l.] : [s.n.], 2007. 606 s. ISBN 0691129932.
- [10] GROFF, James R., WEINBERG, Paul N. *SQL : The Complete Reference*. [s.l.] : Osborne, 1999. 994 s. ISBN 0072118458.
- [11] HAROLD, Eliot Rusty , MEANS, Scott . *XML in a Nutshell : 2nd Edition*. [s.l.] : O'Reilly, 2002. 634 s. ISBN 0-596-00292-0.
- [12] HAUPT, Randy L., HAUPT, Sue Ellen. *PRACTICAL GENETIC*. JOHN WILEY. 2nd enl. edition. [s.l.] : Wiley interscience, 2004. 261 s. ISBN 0-471-45565-2.
- [13] Matlab toolbox. *Humusoft*. 2008, č. 1, s. 1-1. Dostupný z WWW: <www.humusoft.cz>.
- [14] Intel Core 2 Extreme QX9770 Performance Preview. *Hardware*. 2008, no. 1, s. 1-5. Dostupný z WWW: <<http://hothardware.com/>>.
- [15] ActionScript základy. *Forum*. 2006, č. 1, s. 1-3. Dostupný z WWW: <<http://programujte.com/>>.
- [16] AS 3.0 overview. [online]. 2007. 2007 [cit. 2009-04-15]. Dostupný z WWW: <www.adobe.com>.
- [17] Php. *Tvorba webu*. 2004, č. 3, s. 1-2. Dostupný z WWW: <www.tvorba-webu.cz>.
- [18] RUSSEL, S.; NORVIG, P.. *Artificial Intelligence: A Modern Approach*. 2. vyd. New Jersey, USA : Prentice Hall, 2003. ISBN 0-13-790395-2. s. 59-136.
- [19] *Brook* [online]. 2008. 2008 [cit. 2009-04-20]. Dostupný z WWW: <www.ati.com>.
- [20] *nVidia* [online]. 2008. 2008 [cit. 2009-04-21]. Dostupný z WWW: <www.nVidia.com>.
- [21] MATOUŠEK, R. Hill Climbing and 0/1 Knapsack Problem. In *Proceedings of the 8th International Conference on Soft Computing MENDEL 2002*. Brno: FSI, VUT v Brně, 2002. s. 325-328. ISBN: 80-214-2135-5.
- [22] MATOUŠEK, R. GA-HC: A Hybrid Gentic Algorithm. In *Proceedings of the 10th Fuzzy Colloquium in Zittau*. Zittau: -, 2002. s. 239-244. ISBN: 3-9808089-2-0.

SEZNAM OBRÁZKŮ

OBR. 1:	PRINCIP HOROLEZECKÉHO ALGORITMU.....	14
OBR. 2:	TYPY HC PODLE MASKY [3].	15
OBR. 3:	MASKA ALGORITMU HC12 S VELIKOSTÍ 5 BITŮ.....	15
OBR. 4:	PŘEVOD BINÁRNÍHO KÓDOVÁNÍ NA GRAY(OVO)	16
OBR. 5:	ZRCADLOVÉ TECHNIKY V GRAY(OVÉM) KÓDU [1]	16
OBR. 6:	METODA ZVÝŠENÍ VZORKOVÁNÍ U ALGORITMU APC.....	17
OBR. 7:	METODA ZPŘESNĚNÍ ROZSAHU U ALGORITMU APC.....	18
OBR. 8:	MĚŘENÍ RYCHLOSTI VÝPOČTU PRO 1 AŽ 3 RŮZNÉ PC	23
OBR. 9:	PRINCIP DISTRIBUCE HC ALGORITMU POMOCÍ DĚLENÍ MASKY.....	24
OBR. 10:	3D GRAF OPTIMALIZAČNÍ FUNKCE $F_6(x)$	28
OBR. 11:	PRINCIP OHODNOCENÍ FUNKČNÍ OPTIMALIZACE	28
OBR. 12:	ROZDĚLENÍ VÝPOČTU VE FUNKČNÍ OPTIMALIZACI	30
OBR. 13:	OPTIMÁLNÍ ROZLOŽENÍ BATOHU 15.....	33
OBR. 14:	PRINCIP OHODNOCENÍ BATOHU	34
OBR. 15:	ROZDĚLENÍ VÝPOČTU PRO BATOH.....	34
OBR. 16:	OPTIMÁLNÍ ŘEŠENÍ PRO TSP T5.....	38
OBR. 17:	PRINCIP OHODNOCENÍ TSP	38
OBR. 18:	ROZDĚLENÍ VÝPOČTU PRO TSP PROBLÉM.....	39
OBR. 19:	DISTRIBUCE POMOCÍ MATLAB TOOLBOX.....	44
OBR. 20:	SPOTŘEBA PRO RŮZNÉ CPU [14]	44
OBR. 21:	SCHÉMA ROZDĚLENÍ KLIENTŮ.....	45
OBR. 22:	KOMPATIBILITA PŘEDCHOZÍCH VERZÍ AS [16].....	54
OBR. 23:	ZPŮSOB INTERPRETACE PHP [8]	56
OBR. 24:	PRINCIP ČINNOSTI SQL [10]	57
OBR. 25:	PROBLÉM BATOHU PŘÍKLAD B1000	59
OBR. 26:	TSP PŘÍKLAD T20.....	59
OBR. 27:	GRAF ZMĚNY RYCHLOSTI V SAMOSTATNÉM KLIENTU.....	60
OBR. 28:	GRAF PRŮMĚRNÉ RYCHLOSTI VÝPOČTU $F_6(x)$ A $F_7(x)$	61
OBR. 29:	GRAF NALEZENÉHO ŘEŠENÍ $F_6(x)$	62
OBR. 30:	GRAF NALEZENÉHO ŘEŠENÍ T20 S ŘEŠENÍM.....	63
OBR. 31:	GRAF NALEZENÉHO ŘEŠENÍ B1000 S ŘEŠENÍM	63
OBR. 32:	GRAF SNÍŽENÍ VÝPOČTOVÉHO ČASU PŘI DĚLENÍ MASKY.....	64
OBR. 33:	GRAF POTŘEBNÉHO ČASU PŘI INDIVIDUÁLNÍHO DC	65



PŘÍLOHA

Příloha 1. Struktura databáze

Příloha 2. Samostatný klient

Příloha 3. Slave klient (A)

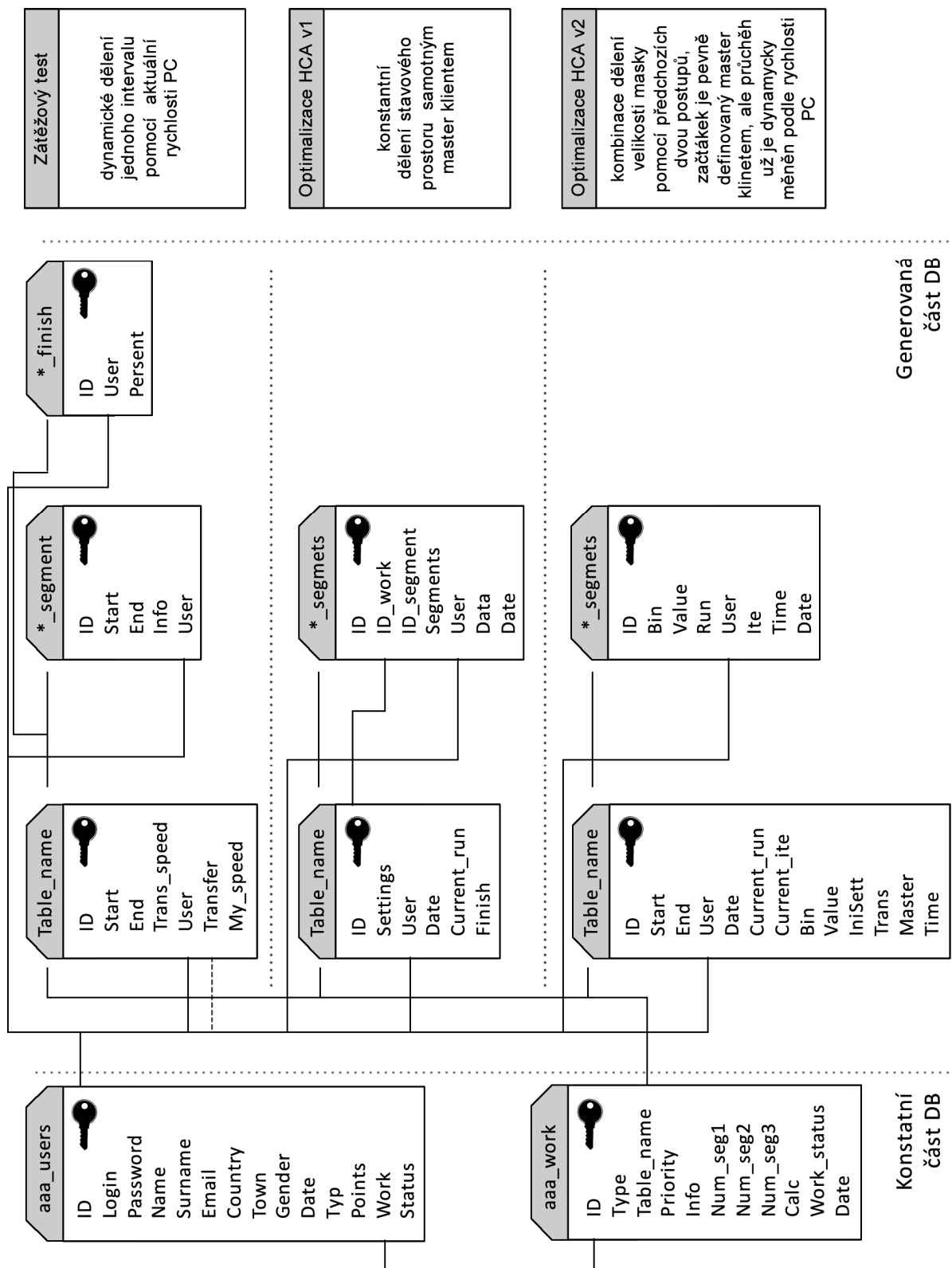
Příloha 4. Slave klient (B)

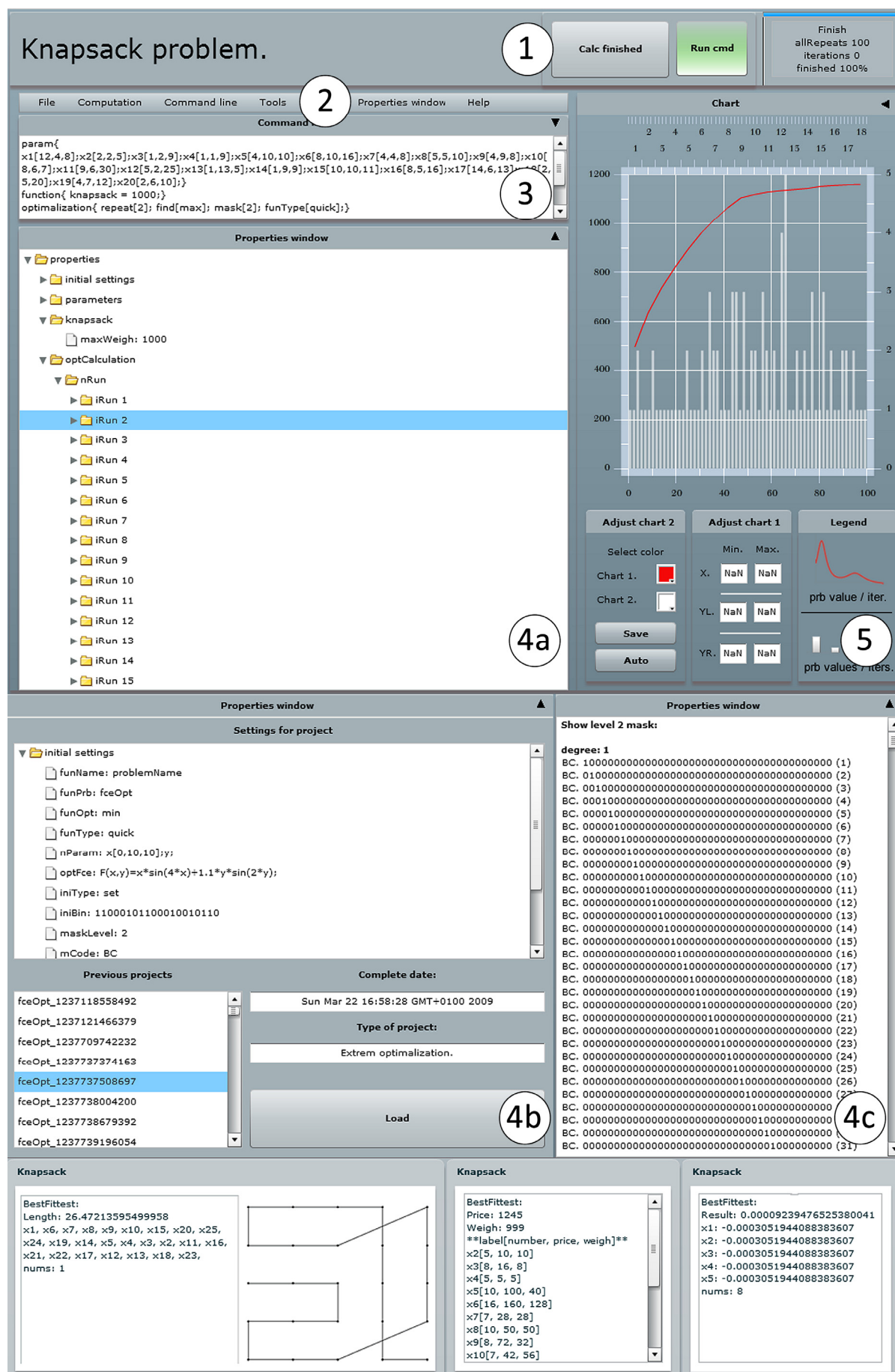
Příloha 5. Master klient (A)

Příloha 6. Master klient (B)

Příloha 7. CD s daty

- CD 1. Slave klient (web)
- CD 2. Slave klient (AIR)
- CD 3. Master klient (AIR)
- CD 4. Samostatný klient (AIR)
- CD 5. PHP skripty
- CD 6. Elektronická verze diplomové práce





Přihlašovací obrazovka



Username **1**

Password

Server url Set server

Login

Registrace/úprava uživatele

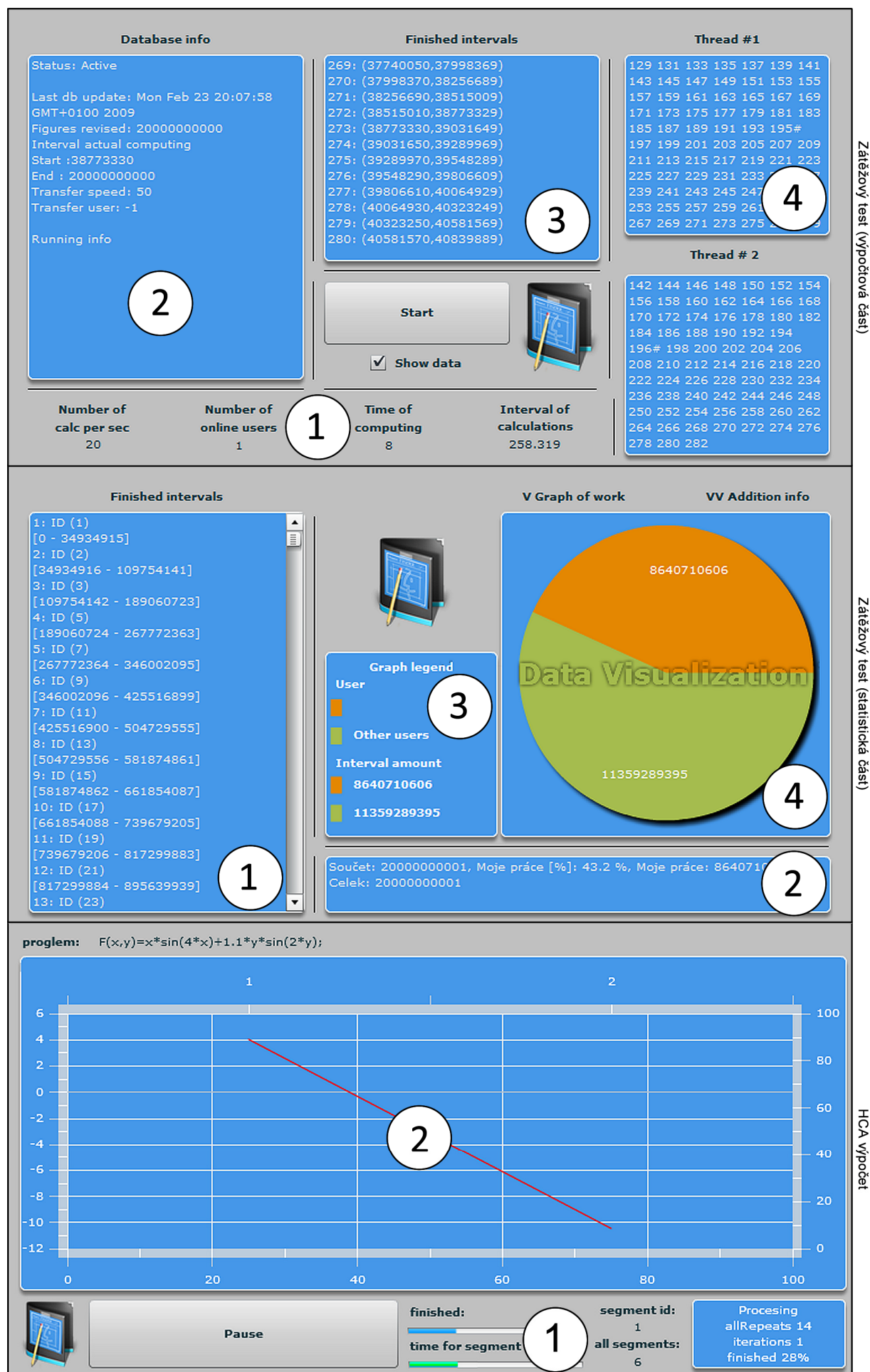
Secondary information		Primary information
1 Name name Surname surname Email email@email.cz Town town Gender Male v Country Czech Republic v	4 Request to delete account Update	User user1 2 Password testing Confirm password testing Account info Date of register Sun Apr 5 21:16:26 GMT+0200 2009 User type Master Points 385 3

Zobrazení všech prací

All available work		Verdict
hotTest fceOpt hotTest knapsackOpt fceOpt 1	Date of work foundation Fri Oct 31 13:09:02 GMT 0100 2008 Start interval: 0 End interval: 20000000000 Online use ☐ 2 Work status Finish Priority <input type="range"/> Refresh Connect to work	work isn't finish 3 Description vypočet prvočísla 50000000000 v intervalu 2 - 20000000000.

Status bar

User user1 1 Actual work - Status Waiting	RollOver on graph 2 more information.	
---	---	--



Přihlašovací obrazovka

User: **user1** Logout

Server name: localhost **1** Set server

Server address: koulelose.gn Start settings

Check work every: 3600 [second]

Manager uživatelských účtů

All registered users

user1
user2

1

Refresh
Delete

2

Secondary information

Name
name

Surname
surname

Email
email@email.cz

Town
town

Gender
Male

Country
Czech Republic

5

Register a new user
Update the user

Primary information

User
user2 **3**

Password
testing

Confirm password
testing

Account info

Date of register
Mon Apr 6 12:32:13 GMT+0200 2009

User type
Master

Points
15 **4**

Obecný db manager

1) Connect to Server

User:
user1 **1**

Password:

Connect

2) List of Databases

information_schema
mysql
testing_db

Create database
my_db
Create DB

Create
Delete database
testing_db
Delete DB

Delete

Database:
Number of DBs: **2**

3) List of Tables

aaa_users
aaa_work
fcept_1240741142577
fcept_1240741142577_segmen
fcept_1240751707242
fcept_1240751707242_segmen
knapsackopt_1240751692015

Create table
CREATE TABLE %s (ROWID INT
PRIMARY KEY AUTO_INCREMENT

Field name
Data type
CHA

Add field Clear query

Table name

Create table Delay **3**

Table:
Number of TBs:

4) Advanced SQL query

SELECT * FROM aaa_work

Execute

onConnectClick
success
///
initDBList
success
3
testing_db
///
initTableList
13
aaa_work

SQL:
Number of rows:
Number of colms: **4**

5

Manage all works | Create a hca opt | Create a test DC

All available work

hotTest

fceOpt

hotTest

knapsackOpt

fceOpt

Refresh

Delete

allWorks

Progress

Full info

DB Information for work: fceOpt_1240751707242

Time of report: Mon Apr 27 09:30:11 GMT+0200 2009

Open work

/*-----DB info-----*/

DB id: 184

Computing problem: Extrem optimization

Work priority: medium

Working segmets table: fceOpt_1240751707242

Finished segmets table: fceOpt_1240751707242_segments

FunName: problemName

Numbers of segments: 6

Numbers of finished segments: 1

Numbers of unique segments: 3

Date of initialization: Sun Apr 26 15:15:07 GMT+0200 2009

Date of expire: Mon Apr 27 17:37:03 GMT+0200 2009

Time for each segmet: 42[min]

Expiration time (work) [h]

26.35562916666667

Expiration time (client) [min]

42

Priority

Info for clients

Info for client

Save: XML

Computing new result

Update work

Duplicate segment

Work: open

Delete segmet

1

2

3

4

5

Fce: $F(x,y)=x*\sin(4*x)+1.1*y*\sin(2*y);$

nParam: x[0,10,10];y;

settings: user defined values [mask: 2; repeat: 50; funOpt: min]

Time to end 69%

Finished segments 17%

Manage all works | Create a hca opt | Create a test DC

All available work

hotTest

fceOpt

hotTest

knapsackOpt

fceOpt

Refresh

Delete

Description

vypocet prvocisla 5000000000 v intervalu 2 - 2000000000.

work isn't finish

2

Name of table

prace2

Waiting segments

Whole number

20000000000

User id

End interval

20000000000

Start interval

0

☒ Finish work

Update

Manage all works | Create a hca opt | Create a test DC

param{ x[0,10,10];y;}

function{ F(x,y)=x*sin(4*x)+1.1*y*sin(2*y);}

apc{ mode{x(range,10,false,1), y(bin,1,3)}; select{best};}

optimization{ repeat[50]; find{min}; mask[2]; funType{quick};}

specification{ funName{problemName}; iniBin{random}; iniType{random}; mCode{BC}; funPrb{fceOpt};}

dc{ chunks[2]; dcSpec[3,x,y]; workTime[51]; clientTime[42]; priority[50]; clientText{info for client};}

1

2

tsp

run

Zpráva práce v rámci HCA

Zpráva práce v rámci zatěžovacího testu

Tvorbu nové práce (HCA příklady)