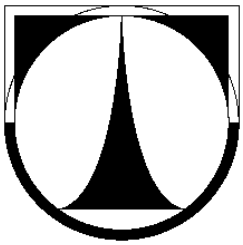
TECHNICKÁ UNIVERZITA V LIBERCI



FAKULTA MECHATRONIKY A MEZIOBOROVÝCH INŽENÝRSKÝCH STUDIÍ

DIPLOMOVÁ PRÁCE

Pavel Paleček

Realizace hierarchického distribuovaného výpočetního systému

Realization of hierarchical distributed computing system

Liberec 2005

Anotace

Paralelizace je jedním z možných směrů zvyšování rychlosti výpočetního systému. Velký rozmach vysokorychlostního internetu, mnohaprocesorových výpočetních systémů (tzv. gridů) i vícejaderných procesorů (včetně jejich hardwarových simulátorů), které jsou již nasazeny nebo na velmi blízkou budoucnost ohlášeny, dává možnost výkonného, ale heterogenního prostředí pro využití k náročným, hlavně matematickým výpočtům.

Cílem práce je navrhnout a realizovat univerzální hierarchický distribuovaný výpočetní systém pro řešení matematických úloh, který bude popsané prostředí využívat.

Annotation

Parallelization is a possible way how to increase the speed of a computing system. A big boom of highspeed internet, multiprocessor computing systems (grids) and multicore processors (including hardware simulators), which have been already applied or which are announced in the very near future, give a possibility of powerful but heterogeneous background for intensive, especially mathematical computations.

The aim of the Diploma Thesis is to design and realize a universal hierarchical distributed computing system for solutions of mathematical tasks which will use described background.

Místopřísežné prohlášení

Místopřísežně prohlašuji, že jsem diplomovou práci vypracoval samostatně s použitím uvedené literatury.

V Liberci dne 1.5.2005 Pavel Paleček

Poděkování

Na tomto místě bych chtěl poděkovat lidem, bez jejichž pomoci by nikdy tato práce nevznikla. Mé díky patří zejména Doc. Janu Cvejnovi, Ph.D. za vedení diplomové práce a mým rodičům za trpělivost.

Seznam použitých zkratek

API - Application Program/Programming Interface

ASCII - American Standard Code for Information Interchange

BFS - Breadth First Search

CPU - Central Processing Unit

DLL - Dynamic Link Library

GUI - Graphical User Interface

HDVS - Hierarchický Distribuovaný Výpočetní Systém

ID - Identification, Identifier

IP - Internet Protocol

ISO - International Standardization Organization

NIC - Network Information Centre

OSI - Open Systems Interconnect/Interconnection

PC - Personal Computer

TCP - Transmission Control Protocol

UDP - User Datagram Protocol

VCL - Visual Component Library

Obsah

1 Úvod, cíl práce

Paralelizace je jedním z možných směrů zvyšování rychlosti výpočetního systému. Velký rozmach vysokorychlostního internetu, mnohaprocesorových výpočetních systémů (tzv. gridů) i vícejaderných procesorů (včetně jejich hardwarových simulátorů), které jsou již nasazeny nebo na velmi blízkou budoucnost ohlášeny, dává možnost výkonného, ale heterogenního prostředí pro využití k náročným, hlavně matematickým výpočtům.

Někdy může být výhodné nebo dokonce nutné rozdělit řešený problém do více vrstev či logických celků – zavést hierarchický model. Pokud se jedná o distribuovaný výpočet s vysokou komunikační náročností, je vhodné rozdělit výpočetní kapacitu tak, aby se uzly v určité technologické doméně (např. jednotlivá jádra procesoru nebo počítače v lokální síti) navenek jevily jako jeden uzel a tím došlo k redukci komunikační režie přes pomalejší infrastrukturu.

Práce se zabývá využitím univerzálního hierarchického modelu řízení komunikace paralelních procesů v prostředí Win32 pro výpočet matematických úloh. Je rozdělena na následující části:

První část obsahuje úvod do problematiky distribuovaných systémů, vysvětluje důvod zavedení hierarchického modelu distribuovaných výpočtů. Vysvětluje pojmy jako distribuované algoritmy, distribuované řízení a data; věnuje se problematice modelu klient – server, zasílaní zpráv a uvádí příklady úloh vhodných pro distribuovaný výpočet.

Druhá část, „Koncepce HDVS“, popisuje zavedení hierarchického modelu nad úroveň výpočetní architektury klient–server tak, aby jednotlivé úrovně mohly autonomně rozhodovat o dalším využití prostředků. Zabývá se návrhem univerzálního rozhraní pro zadání úlohy hierarchickému distribuovanému výpočetnímu systému.

Třetí část, nazvaná „Použité technologie“, pojednává o počítačových sítích a protokolech, které jsou využity pro komunikaci mezi procesy jak v rámci jedné počítačové stanice, tak

pro komunikaci procesů mezi stanicemi vzdálenými. Popisuje způsob zavádění dynamických knihoven (DLL) a využívání jejich funkcí v prostředí Win32. Zdůvodňuje potřebu implementace vláken a popisuje základní metody objektu vlákna TThread z knihovny VCL od firmy Borland®.

Čtvrtá implementační část dokumentuje praktickou implementaci výše popsaného teoretického návrhu s možností zavádět jednotlivé úlohy v podobě dynamických knihoven. Zahrnuje i konkrétní aplikaci úlohy, která realizovaný výpočetní systém využívá.

2 Distribuované systémy

Konkurentní systém složený z mnoha počítačů propojených komunikační sítí je obvykle označován jako distribuovaný systém. Tradiční představa distribuovaného systému s sebou nese i vlastnost značné geografické distribuovanosti, ale ani to nemusí být pravidlem. Hlavními znaky distribuovaného systému jsou neexistence sdílené paměti a globálního časového mechanizmu.

Distribuované systémy dávají možnost rozdělit celkovou zátěž mezi řadu procesorů a tím zvýšit celkovou výkonnost systému. V systému může běžet několik stejných procesů (serverů), které zvyšují rychlost zpracování i spolehlivost. Nejčastějšími důvody pro jejich používání je nižší čas výpočtu ve srovnání s klasickým sekvenčním systémem, vyšší spolehlivost a dostupnost, možnost využití speciálních zařízení a vlastní distribuovaná povaha některých aplikací.

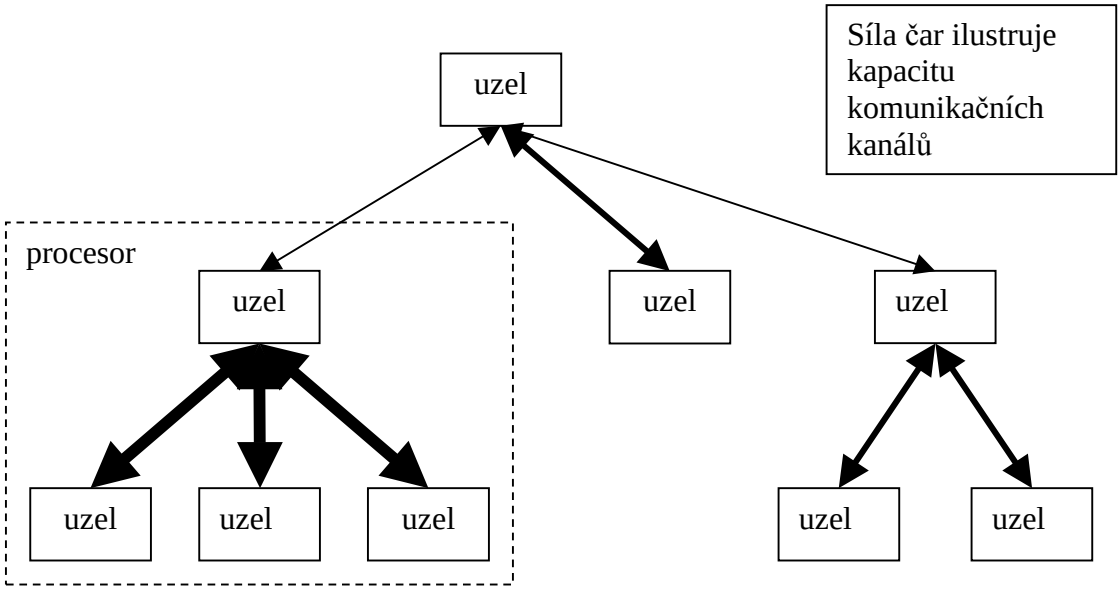
Distribuované výpočty můžeme považovat za zvláštní typ paralelních výpočtů. Paralelní počítání vzniklo jako přirozený důsledek snahy po urychlení především vědecko-technických výpočtů s velkým množstvím dat. Spolupráce mnoha procesorů na jednom výpočtu vyžaduje řízení distribuovaným algoritmem.

V distribuovaném systému řešíme takové problémy, jakými jsou vyrovnání zátěže jednotlivých procesorů, detekce okamžiku, kdy byl výpočet dokončen ve všech procesorech, rozdělení a duplikace dat nebo vznik uváznutí systému, kdy výpočet nemůže pokračovat a nebyl ještě dokončen.

2.1 Hierarchie v distribuovaném výpočtu

Někdy může být výhodné nebo dokonce nutné rozdělit řešený problém do více vrstev či logických celků. Pokud se jedná o distribuovaný výpočet s vysokou komunikační

náročností, je vhodné rozdělit výpočetní kapacitu tak, aby se uzly v určité technologické doméně (např. jednotlivá jádra procesoru nebo počítače v lokální síti) navenek jevily jako jeden uzel a tím došlo k redukci komunikační režie přes pomalejší infrastrukturu (viz Obrázek 2-1).



Obrázek 2-1:Hierarchie v distribuovaném výpočtu

2.2 Distribuované algoritmy

Jsou navrhovány tak, aby byly prováděny mnoha propojenými procesory současně. Části distribuovaného algoritmu mohou běžet současně a nezávisle a každý z procesů pracuje pouze s částí vstupních dat. Celý algoritmus musí pracovat správně, i když jednotlivé procesory a komunikační kanály pracují různými rychlostmi. Problémy, které řešíme v distribuovaném prostředí, mohou zaujmout svojí poněkud záludnou povahou – jejich formulace je obvykle celkem jednoduchá, řešení však musí brát v potaz řadu těžko předvídatelných, komplikovaných a navzájem se ovlivňujících faktorů.

2.3 Příklady úloh

První příklad numerických distribuovaných algoritmů je ilustrací, jak lze sítí se známou a pravidelnou topologií snížit časovou složitost aplikační úlohy, jejíž výsledek sestává z více nezávislých dílčích a vesměs jednoduchých výpočtů a problémem je efektivní dodávání vstupních dat. Druhý uváděný algoritmus ilustruje metodiku, která vede k dynamickému násobenému použití téže funkce při řešení téhož problému, a je v podstatě analogií nasazení více dělníků na tutéž práci.

2.3.1 Násobení matic synchronním pulzováním

Uváděný algoritmus používá pro výpočet každého prvku výsledné matice samostatný procesor. Odpovídající partnerské procesy se identifikují přímo a komunikují synchronně. Pro postupný přenos prvků řádku a sloupce k příslušnému procesoru použijeme pulzující algoritmus (viz 2.6).

Procesoru $P_{1,1}$ postačí pro postupný výpočet $x_{1,1}$ postupně znát prvky $a_{1,1}$ a $b_{1,1}$, poté $a_{1,2}$ a $b_{2,1}$ atd. Tyto hodnoty $P_{1,1}$ může získávat od svých sousedů ve sloupci a řádku v rytmu udávaném pulzujícím algoritmem tak, že každý procesor posune jím pamatovanou hodnotu v každém taktu svému sousedu. Po (n-1) posunech a dílčích výpočtech vyhodnotí $P_{1,1}$ prvek $x_{1,1}$.

Takto jednoduchý postup platí pouze pro procesory počítající diagonální prvky. Pokud matice nepřeuspořádáme, dostane sice každý procesor postupně všechny prvky, které potřebuje, dostane je však v nesprávném pořadí. Proto je před násobením potřeba provést korekci uspořádání matic $\boldsymbol{A}$ a $\boldsymbol{B}$, řádek i matice $\boldsymbol{A}$ se musí cyklicky posunou o i sloupců a sloupec j matice $\boldsymbol{B}$ se musí cyklicky posunout o j řádků.

2.3.2 Paralelní řešení numerické integrace

Druhá ukázka vzoru distribuovaného algoritmu ilustruje možnost spolupráce replikovaných procesů. Řešení problému se urychlí tím, že se problém rozdělí na nezávislé subproblémy, které se řeší souběžně (řada „dělníků“ souběžně pracuje na více subproblémech).

Řešení výpočtu integrálu používá sdílený kanál, ze kterého se čtou zadání dílčích úloh. Počátečně je zde umístěna dílčí úloha, reprezentující celý problém, který se má vyřešit. Násobné procesy, dělníci, si dílčí úlohy z kanálu vybírají a zpracovávají je. Výpočet končí, když jsou zpracovány všechny dílčí úlohy.

Konkrétně je zadána spojitá nezáporná funkce *f(x)* a dvě hodnoty *l* a *r*, pro které platí *l* < *r*. Problém spočívá ve výpočtu plochy ohraničené funkcí *f(x)*, osou *x* a vertikálami procházejícími body *l* a *r*. Úkolem je aproximovat integrál ∫*f(x)* s dolní mezí *l* a horní *r*. Aproximace plochy pod křivkou se provádí rozdělením intervalu <*l*, *r*> na řadu podintervalů a plocha každého podintervalu se aproximuje lichoběžníkem. V našem případě je problém charakterizován pěti hodnotami: levou a pravou mezí *a*, *b*, hodnotou *f(a)* a *f(b)* a plochou lichoběžníka, definovaného těmito čtyřmi údaji (plocha je uváděna proto, abychom se vyhnuli opakovanému výpočtu). Když dělník vypočítá přijatelný výsledek řešení subproblému, zašle ho předákovi kanálem *výsledek*. Program končí v okamžiku, kdy předák zjistí, že byla vypočítána plocha celého intervalu <*l*, *r*>.

2.4 Distribuované řízení a data

Důvody pro distribuovanost můžeme rozdělit přibližně do dvou hlavních skupin:

1. distribuovanost je danému problému vlastní (např. zpracování dat v místě jejich vzniku, resp. v místě jejich potřeby spojené s centrálním zpracováním),
2. distribuovanost je dána implementací (např. požadavkem na zkrácení doby řešení, na spolehlivosti apod.).

Pokud hovoříme o distribuovaném zpracování, distribuovanost se týká dat nebo řízení, nebo obou těchto aspektů.

2.4.1 Distribuovanost dat

Data mohou být distribuována dojím způsobem:

1. **Replikací dat** – vytvořením kopie množiny dat v každém uzlu, ve kterém ji algoritmus využívá. Problémem duplicity je trvalé zajišťování konzistence všech kopií.
2. **Rozložením dat** – rozložení množiny dat na části, které jsou uchovávány pouze v určitých uzlech a ostatním uzlům jsou dostupné pouze zprostředkovaně, tj. výměnou zpráv.

Obě metody lze kombinovat. Celou databázi je možné např. rozložit a určité její prvky přitom replikovat.

2.4.2 Distribuovanost řízení

Nezávisle na řídicím mechanizmu potřebném pro zajištění konzistence distribuovaných dat lze rovněž distribuovat řízení daného algoritmu. O distribuovaném řízení mluvíme v případě koexistence řady souběžných procesů, mezi nimiž neexistuje žádná pevná hierarchie, tedy neexistuje žádný hlavní algoritmus, který globálně řídí celý systém. Nicméně mohou existovat určité funkce, které mohou být prováděny pouze v určitých uzlech. Potom můžeme potřebovat distribuovaný algoritmus k identifikaci těchto uzlů, pokud nejsou určeny *ad hoc*. Podobně se mnohdy provádějí určitá rozhodnutí formou dosažení dohody (na hodnotě, na společné akci, na vzájemném vyloučení kritických sekcí apod.) mezi skupinou uzlů.

Potřeba distribuovaného řízení je často danému problému vlastní: jestliže má být algoritmus odolný vůči poruchám, potom požadavek na jeho pokračování v degradovaném

režimu vylučuje použití centrálního řízení. Distribuované algoritmy plní jednak základní funkce potřebné ve všech, tj. v paralelních, monoprocesorových i distribuovaných informačních systémech (např. vzájemné vyloučení realizace činnosti neslučitelných v čase), jednak funkce řídicí, které jsou speciální pro distribuované systémy (např. detekce ukončení kooperujících procesů, manipulace s distribuovanými kopiemi dat atd.).

Řízení můžeme distribuovat na různých úrovních. Hrubým paralelizmem rozumíme existenci souběžně prováděných programů, které nekomunikují příliš často. Střední paralelizmus znamená rozdělení procesů na paralelně prováděné procedury nebo bloky. Při jemném paralelizmu se současně provádějí jednotlivé příkazy a komunikace je velmi častá. První dva typy jsou vhodné pro distribuované systémy, třetí typ rozdělení řízení je spíše typický pro paralelní systémy.

2.5 Model klient – server

Typický aplikační program obsahuje řadu procedur, jejichž provádění vyvolává hlavní program. Procedury plní služby požadované hlavním programem. Pokud se takové procedury soustředí do nezávislého programu řídicího procesu v některém uzlu, potom hlavní program žádá takový proces o provedení služby. Hlavní program, **klient**, řídí z hlediska vzájemné interakce aktivní proces, který zadává požadavky. Program soustřeďující v sobě programy služeb, **server**, řídí z hlediska vzájemné interakce pasivní proces, reaguje na iniciační aktivitu klientu. Klient čeká, až je jeho požadavek splněn. Server čeká na požadavek klientu a odpovídajícím způsobem na něj reaguje. Server je obvykle nekonečný proces a často poskytuje služby více klientům. Klient a server mohou působit v různých uzlech sítě, mohou však koexistovat i v jediném uzlu, tj. ve společném adresovém prostoru. Vhodnou univerzální programovací technikou pro implementaci programových systémů s architekturou klient – server je zasílání zpráv.

2.6 Zasílání zpráv – pulzování, vlnový algoritmus

Technice nazývané **pulzování** odpovídá klasická sekvenční grafová technika **prohledávání do šířky**, BFS. Předpis postupu BFS říká, že se nejprve označí uzel, tzv. **iniciátor**, tj. kořen, pak jeho sousedé, pak všichni dosud neoznačení sousedé těchto sousedů atd. Z hlediska pulzování je potřeba pojem „označí se“ nahradit pojmem „vyšle se zpráva“. Generovaný pulz postupuje sítí a zprávy se v určitém okamžiku odrazí a vracejí se zpět k iniciátoru. Sítí se tak šíří **vlny**.

Pulzující algoritmy jsou tedy vhodné především pro iterační algoritmy, ve kterých si uzly opakovaně vyměňují informace tak dlouho, dokud neznají odpověď na zadanou otázku. Při pulzování dochází nejprve k **expanzi**, ve které se informace vyšle do okolí, a potom ke **kontrakci**, při níž daný proces přijme informace z okolí. Pulzující algoritmus nazýváme rovněž **vlnový algoritmus**, protože daná informace se šíří ve vlně od inicializačního uzlu vlny k jeho sousedům, od sousedů k jejich sousedům atd., až zasáhne všechny uzly v sítí, a poté se vrací zpět.

Více informací o problematice distribuovaných výpočtů je možno nalézt v publikacích [1] a [2].

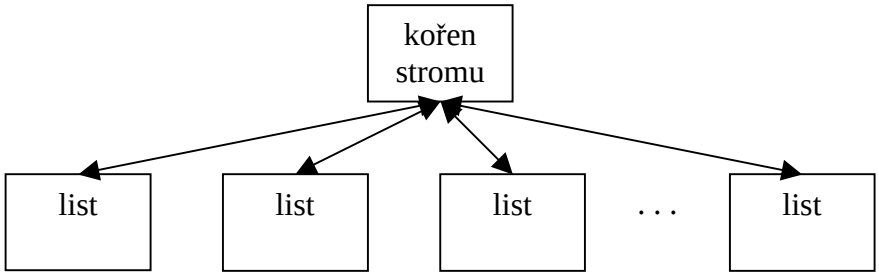
3 Koncepce HDVS

Někdy může být výhodné nebo dokonce nutné rozdělit řešený problém do více vrstev či logických celků. Pokud se jedná o distribuovaný výpočet s vysokou komunikační náročností, je vhodné rozdělit výpočetní kapacitu tak, aby se uzly v určité technologické doméně (např. jednotlivá jádra procesoru nebo počítače v lokální síti) navenek jevily jako jeden uzel a tím došlo k redukci komunikační režie přes pomalejší infrastrukturu. To od výpočetního systému vyžaduje schopnost brát v potaz šířku pásma komunikačních kanálů a výpočetní výkon připojených uzlů (viz Obrázek 2-1). Tuto informaci musí umět zohlednit při generování podúloh. Hierarchický distribuovaný výpočetní systém (HDVS) byl proto realizován jako obecný strom, skládající se z komunikačních-výpočetních uzlů, kde každý uzel je univerzální stavební prvek stromu a je jednoduše nahraditelný.

3.1 Topologie HDVS

Topologii stromu můžeme obecně rozdělit na:

1. Topologii koncipovanou do šířky, kdy převládají uzly zapojené paralelně zapojené v malém množství hierarchických vrstev (viz Obrázek 3-1). Struktura tohoto typu je výhodnější při nižším komunikačním provozu mezi jednotlivými uzly nebo pokud slučování parciálních řešení daného problému není neúměrně výpočetně náročné.
2. Topologii s větším počtem hierarchických vrstev (viz Obrázek 3-2), která je vhodnější při velké komunikační režii, při technologických omezeních nebo je potřeba nad parciálními řešeními provádět časově náročné operace.



Obrázek 3-1:Paralelní zapojení HDVS

3.2 Komunikace mezi uzly HDVS

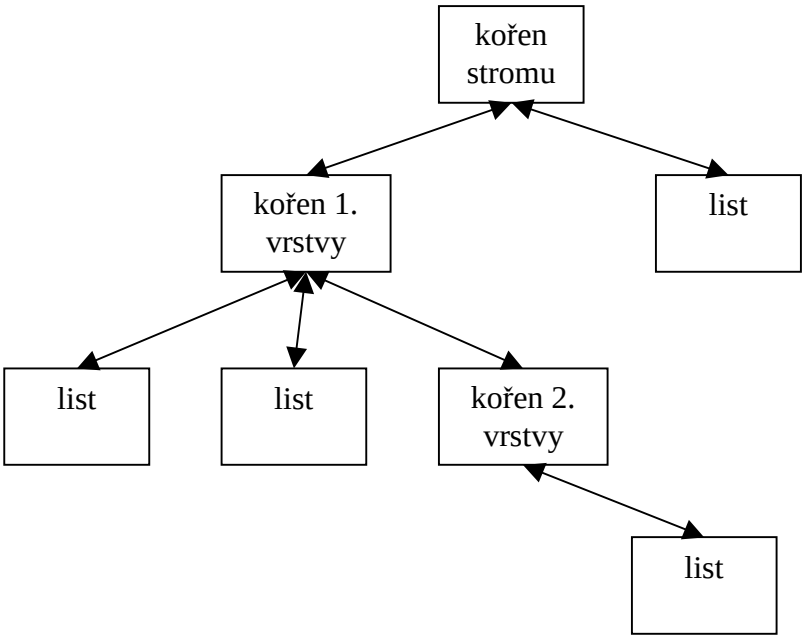
Hrany stromu tvoří síťová komunikace, která je užita jako transportní vrstva pro přenos zpráv mezi uzly a jednotlivými hierarchickými úrovněmi. Ideové schéma stromu ukazuje Obrázek 3-2.

Ke komunikaci v síti uzlů se využívá technika pulzování. Při pulzování dochází nejprve k expanzi, ve které se informace vyšle směrem z vyšších hierarchických vrstev do vrstev nižších, a potom ke kontrakci, při níž každý kořen určité hierarchické vrstvy přijme informace z připojených uzlů. Pulzující algoritmus se nazývá rovněž vlnový algoritmus, protože daná informace se šíří ve vlně od inicializačního uzlu vlny směrem k nižším vrstvám atd., až zasáhne všechny uzly v sítí, a poté se vrací zpět.

3.3 Rozdělení výpočetního výkonu

Heterogenní prostředí, ve kterém HDVS pracuje, způsobené rozdílným výpočetním výkonem zapojených prvků a rozdílnou šířkou pásma komunikační infrastruktury vyžaduje od výpočetního systému schopnost dělit úlohy v poměru odpovídajícím výpočetnímu výkonu připojených uzlů. Každý uzel zapojený do HDVS disponuje hodnotou, která

vyjadřuje jeho výpočetní kapacitu. Hodnotu může nastavit buď uživatel nebo je na jeho žádost určována podle doby trvání posledního výpočtu. Tím dochází k adaptivnímu nastavení vah větví.



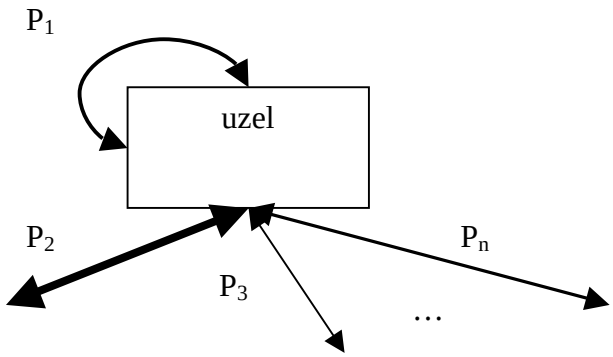
Obrázek 3-2:Hierarchický distribuovaný výpočetní systém

3.4 Třída řešených úloh

Jedná se o uživatelský modul připojitelný přes předepsané komunikační rozhraní k uzlu HDVS. Musí být navržen tak, aby splňoval následující podmínky:

- musí obsahovat funkce předepsaného komunikačního rozhraní,
- musí nést informaci o celé řešené úloze, tzn. zadání úlohy včetně okrajových podmínek,
- musí implementovat funkci zajišťující rozdělení úlohy na podúlohy na základě informace od výpočetního systému o rozložení vah větví (viz Obrázek 3-3),

- musí obsahovat samotnou výpočetní funkci, schopnou provést výpočet pro zadaný interval,
- musí umět partikulární výsledky sečíst.

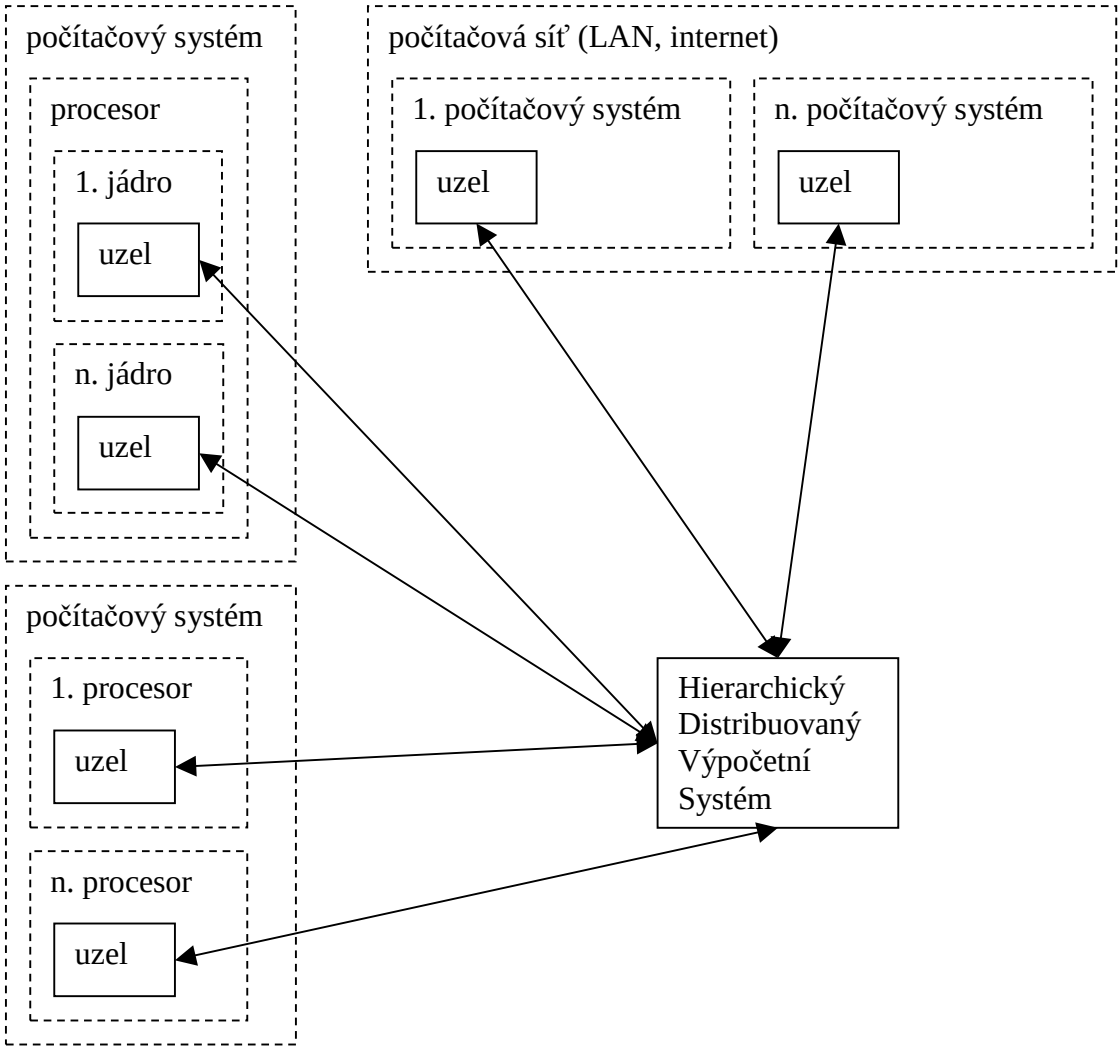


Obrázek 3-3:Zohlednění vah větví při generování podúloh

Obrázek 3-3 ukazuje rozdělení úlohy na podúlohy vlivem vah větví (pozn. podúloha P_1 je řešena výpočetním vláknem uzlu.).

4 Použité technologie

Zvoleným prostředkem pro tvorbu konkrétních aplikací je programovací jazyk C++ a knihovna VCL (Visual Component Library) od firmy Borland®, zapouzdřující širokou škálu funkcí. Především funkce pro síťovou komunikaci, vlákna (threads) a tvorbu jednoduchého grafického uživatelského prostředí (GUI).



Obrázek 4-1:Heterogenní prostředí využité HDVS

Kapitola zdůvodňuje volbu použitých technologií a vysvětluje pojmy a principy, které je nutné znát pro pochopení práce s HDVS.

4.1 Počítačové sítě a protokoly

Heterogennost prostředí (viz Obrázek 4-1), ve kterém by měl hierarchický distribuovaný výpočetní systém pracovat, vyžaduje univerzální komunikační prostředek. Jednotlivé uzly distribuovaného systému mohou být spuštěny vedle sebe v rámci jednoho počítače (např. na procesoru s více jádry či víceprocesorovém systému), v rámci malé lokální počítačové sítě nebo na velkou vzdálenost v prostředí internetu. Nejvhodnějším prostředkem se pro daný problém jeví internetový protokol TCP/IP.

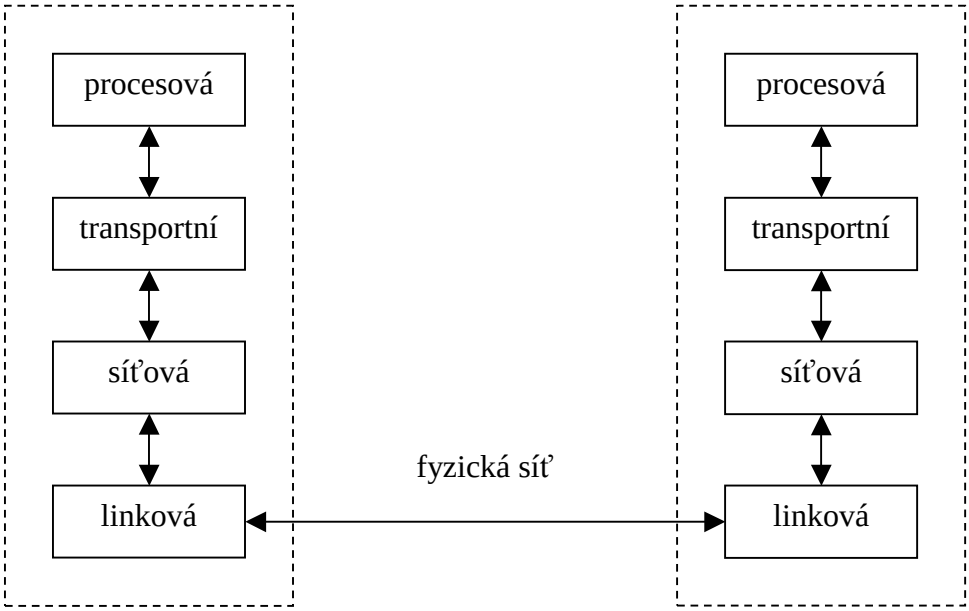
4.1.1 Model OSI, protokoly, vrstvení

Počítače propojené v síti používají ke komunikaci přesně definované protokoly. Protokol je souhrn pravidel a konvencí používaných mezi účastníky komunikace. Vzhledem k tomu, že tyto protokoly mohou být dosti složité, jsou navrženy ve vrstvách, čímž se usnadní jejich používání. V tabulce (Tabulka 4-1) je uveden model OSI.

Číslo vrstvy	Název
1	Fyzická (physical)
2	Spojová (data link)
3	Síťová (network)
4	Transportní (transport)
5	Relační (session)
6	Prezentační (presentation)
7	Aplikační (application)

Tabulka 4-1:Model OSI

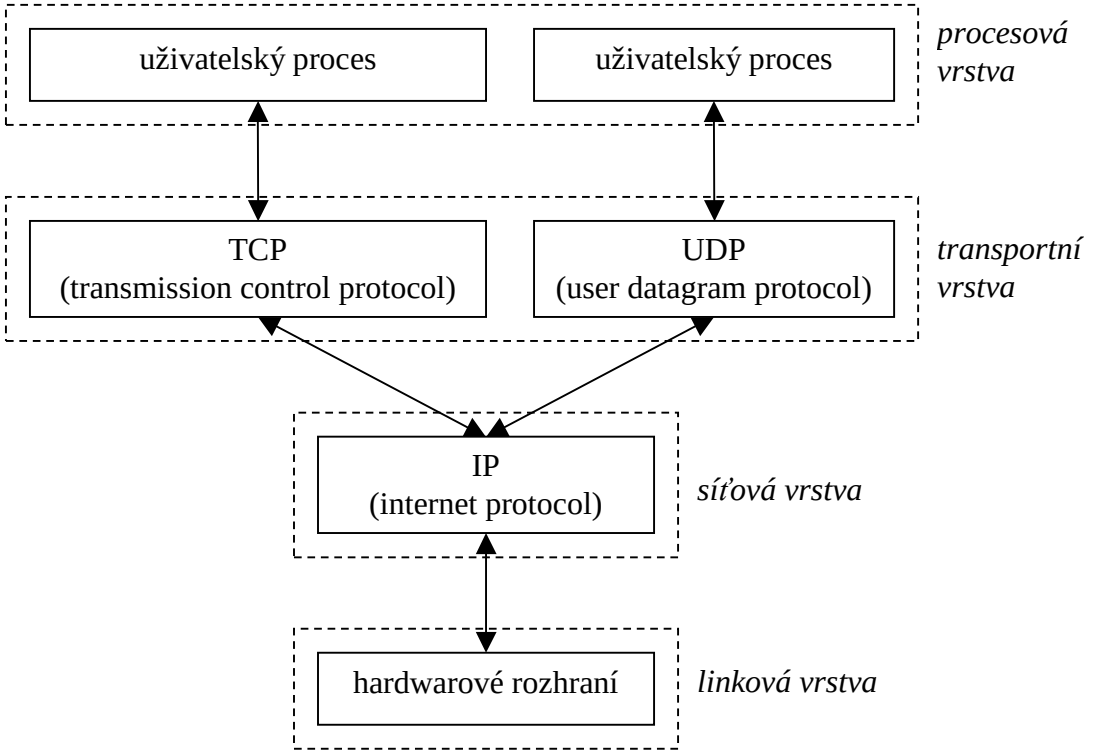
Tento model byl vyvinut v letech 1977 – 1984 a je pouhým návodem, nikoli přesně stanoveným popisem.



Obrázek 4-2:Zjednodušený čtyřvrstvý model propojující dva systémy

Mezi výhody vrstvení patří i to, že umožňuje dobře definovat rozhraní mezi vrstvami, takže změna v jedné vrstvě neovlivní vrstvu přilehlou. Typ protokolu je souborem protokolů z více než jedné vrstvy, někdy je rovněž označován jako rodina protokolů.

Model OSI se někdy zjednodušuje na model čtyřvrstvý (viz Obrázek 4-2). Obrázek 4-3 znázorňuje vrstvy používané protokoly typu TCP/IP.



Obrázek 4-3:Typ protokolů TCP/IP využívající čtyřvrstvý model

4.1.2 Režimy provozu

Služby zaměřené na stálé spojení vyžadují, aby oba aplikační programy ustanovily vzájemné logické spojení dříve, než se uskuteční samotná komunikace. Při tomto utváření spojení dochází k určité prodlevě. Služby zaměřené na stálé spojení se často používají v případě výměny více než jedné zprávy mezi rovnocennými jednotkami. Výměna zaměřená na stálé spojení zahrnuje tři kroky:

- utvoření spojení,
- přenos dat a
- ukončení spojení.

Konverze služby orientované na stálé spojení ve službu bez stálého spojení se nazývá službou datagramů. V případě tohoto typu služby se zprávy nazývají datagramy a jsou přenášeny z jednoho systému do druhého. Vzhledem k tomu, že jednotlivé zprávy se přenášejí nezávisle na ostatních, musí každá zpráva obsahovat informaci potřebnou pro své doručení. V typu protokolu TCP/IP umožňuje TCP virtuální okruh zaměřený na stálé spojení, zatímco UDP nabízí datagramové vybavení bez spojení.

Kontrola sledu informací popisuje vlastnost, která zaručuje, že příjemce obdrží data ve stejném sledu, v jakém je odesílatel odeslal. V síti přepínané pakety je možné, aby dva po sobě jdoucí pakety použily různé směry od zdrojového počítače do cílového a aby tak dorazily na místo určení v jiném pořadí, než byly odeslány.

Kontrola chyb zaručuje, že aplikační program přijme bezchybná data. V případě, že k nějaké modifikaci dojde, musí příjemce požádat odesílatele, aby mu data zaslal znovu. Aby se zabránilo ztrátě dat během přenosu po síti, musí odesílatel měřit čas a poté, co odeslal data a vypršela stanovená lhůta (timeout), musí data poslat znovu. Při přenosu se mohou ztratit nejen zprávy, ale i potvrzení. Pokud k tomu dojde, pošle odesílatel data znovu, takže příjemce obdrží data dvakrát. Příjemce proto musí provést kontrolu dvojího výskytu – duplikace – a pokud již uvedená data obdržel, zprávu ignoruje.

Řízení toku dat zjišťuje, zda odesílatel nezahlcuje příjemce tím, že vysílá zprávy rychleji, než je příjemce může zpracovat. Pokud se toto řízení toku dat nepoužije, může se stát, že příjemce ztratí informaci z důvodu nedostatku zdrojů. Protokol TCP nabízí kontrolu sledu, chyb i toku dat. Naproti tomu UDP žádnou z uvedených služeb neposkytuje.

Služba proudu slabik neklade proudu dat žádná omezení. Konverzí tohoto rysu je služba orientovaná na zprávy, která příjemci uchovává hranice zpráv tak, jak je stanovil odesílatel. TCP je typ protokolu zaměřený na proud slabik, zatímco UDP nabízí ohraničení zpráv.

Spojení s plným duplexem umožňuje v jednom okamžiku obousměrný přenos dat mezi rovnocennými jednotkami. TCP tento plný duplex nabízí.

4.1.3 Model klient-server

Servery se rozdělují na iterativní a konkurentní. Iterativní server ví předem, jak dlouho trvá zpracování každého požadavku, a proces serveru zpracovává každý požadavek sám. Server konkurentní nezná velikost objemu práce potřebné ke zpracování požadavku, a proto spouští pro každý požadavek samostatný proces, který jej zpracuje.

Server provádí následující kroky:

1. Otevře komunikační kanál a informuje místní hostitelský uzel, že je připraven přijmout požadavky klienta na některé ze známých adres.
2. Čeká na požadavek klienta na známé adrese.
3. Iterativní server požadavek zpracuje a odešle odpověď. Tento typ serverů se obvykle používá, když může být požadavek klienta vyřízen v rámci jedné odpovědi ze strany serveru. Konkurentní server vytvoří a spustí nový proces, který má tento požadavek klienta zpracovat. Uvedený nový proces pak zpracuje požadavek klienta, ale nemusí odpovídat na požadavky jiných klientů. Jakmile tento proces ukončí požadovanou činnost, uzavře komunikační kanál a skončí.
4. Vrátí se zpět na krok 2, čeká na požadavek klienta.

Předchozí kroky předpokládají, že systém nějakým způsobem zjistí, že v průběhu zpracování předchozího požadavku klienta přišel další požadavek. V případě konkurentního serveru mohou přijít další požadavky během doby, kdy server vytváří nový proces pro zpracování požadavku klienta. Rovněž se předpokládá, že hlavní proces serveru existuje po celou dobu činnosti hostitelského uzlu – server nikdy neukončí svou činnost, pokud k tomu není donucen.

Proces klienta provádí jinou sérii operací:

1. Otevře komunikační kanál a napojí se na známou adresu určitého hostitelského uzlu (tj. serveru).
2. Odešle serveru zprávu o požadavku služby a přijme odpověď. Tento krok provádí tak dlouho, dokud je třeba.
3. Uzavře komunikační kanál a skončí.

Otevření, které provede server, následně čekající na požadavek klienta, se nazývá pasivní otevření. Naproti tomu klient provádí tzv. aktivní otevření, protože předpokládá, že jej server již očekává.

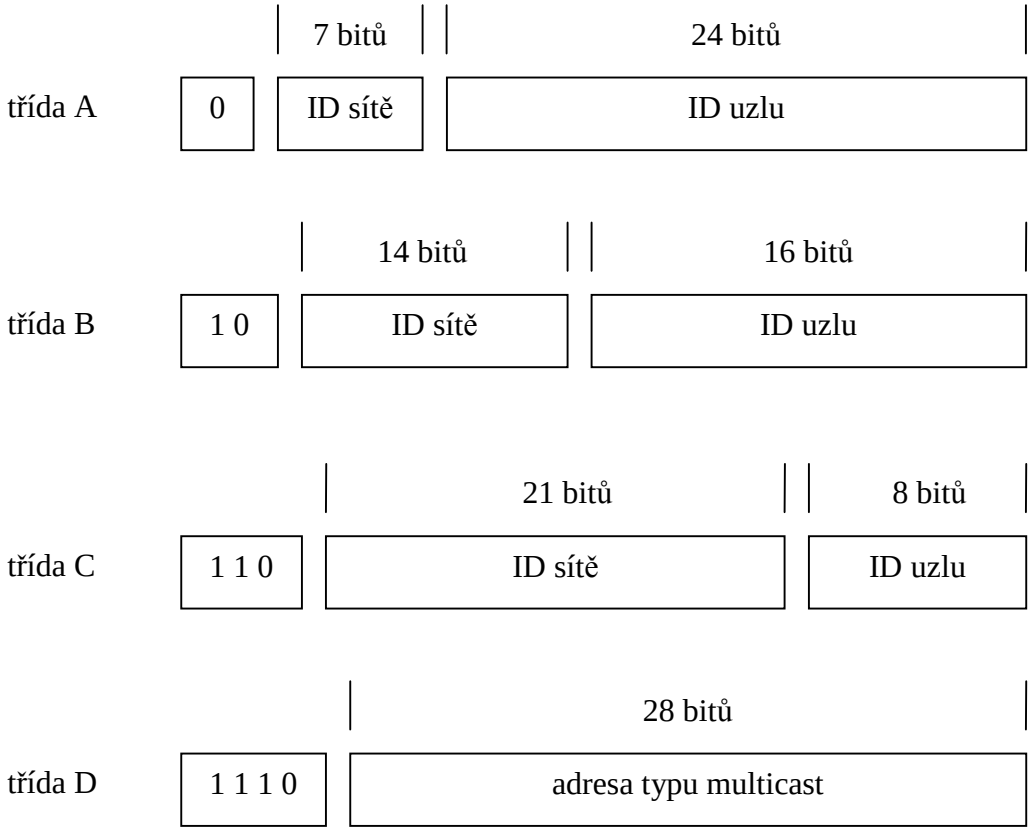
4.1.4 Síťová vrstva IP

Datagramy IP

Vrstva IP poskytuje nespolehlivý systém doručování bez stálého spojení. Systém nemá stálé spojení, protože považuje každý datagram IP za nezávislý na ostatních. Jakékoliv spojení mezi datagramy musí poskytnout vyšší vrstvy. Každý datagram IP obsahuje zdrojovou a cílovou adresu, takže každý datagram může být doručen i směrován nezávisle. Vrstva IP je nespolehlivá z toho důvodu, že nemůže vždy zaručit, že budou datagramy IP doručeny správně nebo doručeny vůbec. I spolehlivost musí zaručit vyšší vrstvy. Vrstva IP vypočítává a verifikuje kontrolní součet, který zahrnuje svou vlastní 20slabičnou hlavičku. Takto může verifikovat pole, která potřebuje prozkoumat nebo zpracovat. Pokud je hlavička IP nalezena s chybou, je zrušena s předpokladem, že vyšší vrstvy protokolu zajistí opakovaný přenos paketu.

Vrstva IP je dále zodpovědná za fragmentaci. Když brána obdrží datagram IP, který je příliš velký na to, aby byl poslán po síti, IP modul jej rozloží na fragmenty a jednotlivě je odešle jako pakety IP. Tyto fragmenty jsou opět složeny do podoby datagramu IP poté, co dorazí do svého cíle. Pokud se některý z datagramů ztratí nebo je zrušen, zruší cílový hostitelský uzel celý datagram.

Vrstva IP nabízí základní podobu řízení toku dat. Když dorazí paket IP do hostitelského uzlu nebo na bránu tak rychle, že je zrušen, modul IP pošle původnímu zdroji zprávu o zničení, čímž systém informuje, že data přicházejí na místo určení příliš rychle.



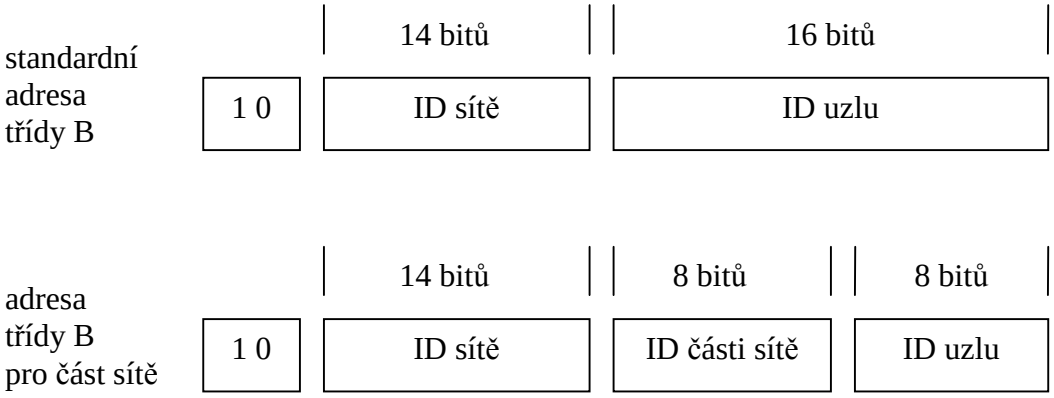
Obrázek 4-4:Formáty adres internetu

Adresy internetu

Adresy internetu zabírají 32 bitů a je v nich zahrnuto jak ID sítě, tak ID hostitelského uzlu. ID hostitelského uzlu se vztahuje k dané síti. Každý hostitelský uzel na síti TCP/IP internet musí disponovat jedinečnou 32bitovou adresou. Adresy TCP/IP v síti internet přiřazuje centrum NIC (Network Information Centre – Síťové informační centrum). 32bitová adresa internetu má jeden z následujících čtyř formátů (viz Obrázek 4-4).

Adresy třídy A se používají pro sítě, které mají mnoho hostitelských uzlů v rámci jedné sítě. Adresy třídy C se používají pro více sítí s méně hostitelskými uzly. Adresy internetu se obvykle zapisují jako čtyři desítková čísla, oddělená tečkami. Každé desítkové číslo

kóduje jednu slabiku 32bitové adresy internetu. Každý datagram IP obsahuje v každé 20slabičné hlavičce IP 32bitovou adresu internetu pro hostitelský uzel a 32bitovou adresu internetu pro cílový hostitelský uzel.



Obrázek 4-5:Adresa internetu třídy B s použitím části sítě

Adresy částí sítí

Každá organizace s adresou internetu kterékoliv třídy může dále jakýmkoliv způsobem rozdělit dostupnou část hostitelského prostoru na adresy, čímž vzniknou sítě.

Obrázek 4-5 představuje rozdělení ID uzlu na ID části sítě a ID uzlu.

Tato situace přináší nový prvek do hierarchie adres internetu a rozšiřuje možnost identifikace v síti na:

- ID sítě
- ID části sítě v rámci dané sítě
- ID hostitelského uzlu v rámci části sítě

4.1.5 Transportní vrstva – UDP a TCP

Tabulka 4-2 porovnává protokoly IP, UDP a TCP z hlediska nabízených služeb.

	IP	**UDP**	**TCP**
zaměřený na stálé spojení	ne	ne	ano
hranice zpráv	ano	ne	ne
kontrolní součet	ne	volitelný	ano
Potvrzování	ne	ne	ano
časová prodleva a opakování	ne	ne	ano
zdvojená detekce	ne	ne	ano
kontrola sledu	ne	ne	ano
řízení toku dat	ne	ne	ano

Tabulka 4-2:Porovnání rysů jednotlivých protokolů IP, UDP a TCP

Čísla portů

Je možné, aby více než jeden uživatelský proces využíval v určitém okamžiku buď TCP nebo UDP. Pro tento předpoklad jsou však zapotřebí určité metody identifikace dat vztahujících se k jednotlivým procesům. TCP i UDP používají k této identifikaci 16bitová celá čísla – čísla portů.

Když se chce uživatelský proces spojit se serverem, musí mít klient k dispozici způsob, kterým by identifikoval požadovaný server. Jestliže klient zná 32bitovou adresu internetu daného hostitelského uzlu, na níž server sídlí, může tento hostitelský uzel kontaktovat. Zvolený proces serveru určí jak TCP, tak UDP pomocí skupiny tzv. známých portů.
Čísla portů v TCP a UDP jsou rezervována v rozsahu od 1 do 255. Do tohoto rozmezí spadají všechny známé porty, které využívají základní služby.

Pětice, která určuje asociaci v protokolu typu internet, se skládá z:

- protokolu (TCP nebo UDP),
- adresy internetu místního hostitelského uzlu (32bitová hodnota),
- čísla místního portu (16bitová hodnota),
- adresy internetu cizího hostitelského uzlu (32bitová hodnota),
- čísla cizího portu (16bitová hodnota).

Více o počítačových sítích a protokolech v publikacích [3],[4] a [5].

4.2 Vlákna v C++ Builderu

C++ Builder poskytuje několik objektů, které usnadňují zápis vícevláknových aplikací. Vícevláknové aplikace jsou aplikace, které obsahují několik souběžných cest provádění. Při použití jednoho vlákna, program musí zastavit všechno provádění, když čeká na dokončení pomalého procesu (např. zpřístupnění souboru na disku, komunikaci s dalšími počítači nebo zobrazování multimédií). CPU čeká, dokud proces není dokončen. S více vlákny, aplikace může pokračovat v provádění dalších vláknen, když jedno vlákno čeká na výsledek pomalého procesu.

Chování programu může být často organizováno do několika paralelních procesů, které pracují nezávisle. Každý z těchto paralelních procesů může být prováděn souběžně pomocí jednoho vlákna. Jednotlivým vláknům lze přiřadit prioritu, čím určíme jak mnoho času CPU bude vlákno používat. Pokud systém, na kterém běží aplikace, má několik procesorů, můžeme zvýšit výkonnost rozdělením práce do několika vláken a spouštět je současně na různých procesorech.

4.2.1 Vytvoření vlákna

K reprezentaci prováděného vlákna použijeme objekt vlákna. Objekty vlákna zjednodušují zápis vícevláknových aplikací zaobalením nejčastějších požadavků na vlákna. Objekty

vláken neumožňují ovládat bezpečnostní atributy nebo velikost zásobníku našeho vlákna. Abychom to mohli provést, musíme použít funkci API Windows **CreateThread**. Objekt vlákna musí být oddděděn od třídy **TThread**.

Hodnota	**Priorita**
tpIdle	Vlákno je prováděno pouze, když je systém nečinný. Windows nepřerušuje provádění ostatních vláken k provedení těchto vláken.
tpLowest	Priorita vlákna je dva body pod normálem.
tpLower	Priorita vlákna je jeden bod pod normálem.
tpNormal	Vlákno má normální prioritu.
tpHigher	Priorita vlákna je jeden bod nad normálem.
tpHighest	Priorita vlákna je dva body nad normálem.
tpTimeCritical	Vlákno získá nejvyšší prioritu.

Tabulka 4-3:Priority vláken

Konstruktor použijeme k inicializaci nové třídy vlákna. Můžeme zde přiřadit implicitní prioritu vlákna a určit, zda vlákno bude automaticky uvolněno po dokončení provádění. Priorita určuje, kolik prostředků vlákno získá, když operační systém plánuje využití času CPU pro všechna vlákna v aplikaci. Vyšší prioritu použijeme pro vlákna zpracovávající kritické úlohy a nižší prioritu pro vlákna provádějící ostatní úkoly. Pro určení priority vlákna nastavíme vlastnost **Priority**. Priorita může nabývat sedmi možných hodnot (viz Tabulka 4-3).

Následující kód ukazuje konstruktor vlákna nejnižší priority, které je prováděno na pozadí (nepřerušuje ostatní aplikace):

```
__fastcall TMyThread::TMyThread(bool CreateSuspended):
TThread(CreateSuspended)
{
  Priority = tpIdle;
}
```

4.2.2 Běh vlákna

Často aplikace provádí jisté vlákno pouze jednou. V tomto případě je nejvhodnější, když objekt vlákna po skončení běhu uvolní sám sebe. To nastane, když vlastnost **FreeOnTerminate** je nastavena na *true*. Jestliže objekt vlákna reprezentuje úlohy aplikace, které jsou prováděny několikrát (např. v reakci na akci uživatele nebo příchodu externí zprávy), pak můžeme zvýšit výkonnost odložením vlákna pro opětovné použití (namísto jeho zrušení a opětovného vytvoření). To provedeme nastavením vlastnosti **FreeOnTerminate** na *false*.

Metoda **Execute** představuje činnost vlákna. Můžeme zde určit, jakým způsobem bude vlákno chápáno aplikací, mimo sdílení stejného procesového prostoru. Zápis vlákna je nepatrně složitější než zápis samostatného programu, neboť se musíme ujistit, že nepřepisujeme paměť používanou jinými vlákny v aplikaci. Na druhé straně, díky sdílení stejného procesorového prostoru s ostatními vlákny, můžeme použít sdílenou paměť pro komunikaci mezi vlákny.

4.2.3 Synchronizace vláken

Vlákna nezajišťují bezpečný přístup k vlastnostem a metodám objektů z hierarchie objektů C++ Builderu. To znamená, že přístup k vlastnostem nebo spouštění metod může způsobit zápis do paměti, která není chráněna před akcemi z ostatních vláken.

Každá aplikace využívající VCL spouští hlavní vlákno VCL. Jedná se o vlákno, které zpracovává komunikaci mezi frontou zpráv Windows a komponentami v aplikaci. Synchronizaci s hlavním vláknem a uživatelským vláknem zprostředkovává metoda **Synchronize**. **Synchronize** zařadí předanou metodu do cyklu hlavního vlákna VCL.
Ne vždy musíme používat hlavní vlákno VCL. Některé objekty jsou navrženy jako vláknově bezpečné. Pro volaní jejich metod není nutné použít metodu **Synchronize**, což zvyšuje výkonnost, neboť není potřeba čekat na vstup do cyklu zpráv hlavního vlákna.

Další informace o třídě **TThread** a jejích metodách je možno nalézt v [6].

4.3 Použití DLL

Aby bylo možné vytvořit obecný výpočetní systém, je nutné oddělit implementaci jednotlivých úloh, určených pro HDVS od samotného výpočetního systému. Dynamicky sestavitelné knihovny (Dynamic Linked Library) umožňují modularizovat aplikace a tak usnadňují aktualizování a opětovné používání jejich funkčnosti. Také pomáhají redukovat paměťové nároky. Pokud několik aplikací používá ve stejnou chvíli stejné funkce z knihovny DLL, získá sice každá aplikace svou vlastní kopii dat, kód je však sdílen.

DLL knihovny jsou moduly, které obsahují funkce a data. DLL jsou zaváděny za běhu aplikace tak, že jsou mapovány do adresového prostoru volajícího procesu. DLL mohou definovat dva typy funkcí:

1. exportované funkce mohou být volány ostatními moduly
2. interní funkce mohou být volány pouze uvnitř DLL, ve které jsou definovány.

Exportované funkce DLL knihovny musí být identifikovány modifikátorem:

```
extern "C" __declspec(dllexport)
```

4.3.1 Dynamické zavedení DLL

K dynamickému zavedení DLL za běhu aplikace se používá funkce Windows API **LoadLibrary**.

```
HINSTANCE LoadLibrary(
  LPCTSTR lpLibFileName
);
```

Parametrem funkce je *lpLibFileName*. Jedná se o ukazatel na nullem ukončený řetězec znaků se jménem DLL souboru. Funkce selže pokud není soubor nalezen. Návratová hodnota představuje handle (rukojeť) na zavedenou knihovnu. NULL hodnota indikuje selhání funkce.

4.3.2 Lokalizace funkcí v knihovně

Konkrétní funkce z DLL knihovny se lokalizuje funkcí **GetProcAdress**.

```
FARPROC GetProcAddress(
  HMODULE hModule,
  LPCWSTR lpProcName
);
```

Funkce má dva parametry. *hModule* je handle (rukojeť) zavedeného DLL modulu. *lpProcName* je ukazatel na NULLem ukončený řetězec znaků se jménem hledané funkce. Návratová hodnota představuje adresu exportované funkce. NULL hodnota indikuje selhání funkce.

4.3.3 Uvolnění DLL z paměti

Uvolnění DLL modulu z paměti zajišťuje funkce **FreeLibrary**.

```
BOOL FreeLibrary(
  HMODULE hLibModule
);
```

Parametrem funkce je *hLibModule*, který představuje handle (rukojeť) zavedeného DLL modulu. Při úspěchu je na výstupu funkce nenulová hodnota

Tvorba a používání DLL je podrobně popsána v [3] a [6].

5 Implementace

Hierarchický distribuovaný výpočetní systém je realizován jako obecný strom, skládající se z komunikačních-výpočetních uzlů. Hrany stromu tvoří síťová soketová komunikace TCP/IP vrstvy, která je užita jako transportní vrstva pro přenos zpráv mezi uzly. Ke každému uzlu se přes předepsané rozhraní připojuje dynamická knihovna, která obsahuje algoritmus řešení konkrétní úlohy.

5.1 Systém zpráv

Jak už bylo vysvětleno výše, přenos zpráv je realizován TCP/IP vrstvou síťového komunikačního modelu. Zprávy jsou složeny z klíčových slov (tokenů), jako oddělovač je použita mezera. Tokeny jsou tvořeny řetězci ASCII znaků tak, aby byly jednoduše čitelné i člověkem. Základní token (prefix), kterým je uvozena každá samostatná zpráva má tvar **HDVS**. Následující tabulky uvádí seznam a popis významu všech tokenů použitých v systému (bez prefixu).

Token	**Příklad použití**	**Význam**
LOGIN <celková kapacita>	LOGIN 5	Přihlášení k nadřazenému uzlu a předání celkové výpočetní kapacity (součet všech výpočetních kapacit podřízených uzlů a kapacity uzlu).
CAP <celková kapacita>	CAP 5	Předání celkové výpočetní kapacity nadřazenému uzlu.
RESULT <číslo vyřešené podúlohy> <celkový výsledek>	RESULT 1 600	Informuje nadřazený uzel o výsledku vyřešené úlohy.
LOGOUT	LOGOUT	Odhlášení uzlu.

Tabulka 5-1:Tokeny zasílané klientem serveru

Tokeny určené pro komunikaci s nadřazeným uzlem (viz Tabulka 5-1). Tokeny určené pro komunikaci s podřazeným uzlem uvádí Tabulka 5-2.

Token	Příklad použití	Význam
OK	OK	Každá zpráva přijatá nadřazeným uzlem je tímto potvrzena.
JOB <název knihovny s úlohou>	JOB mcfe.dll	Nastavuje knihovnu s konkrétní úlohou.
JOB	JOB	Uvolní aktuální úlohu z paměti.
BOUNDS <číslo úlohy> <meze pro úlohu>	BOUNDS 1 300;600	Předání čísla a mezí podúlohy (Meze jsou libovolný vektor).

Tabulka 5-2:Tokeny zasílané serverem klientovi

5.2 Uzel

Je koncipován jako jednoduchý stavový automat (popis stavů viz Tabulka 5-3), který je programován nadřazenými uzly (případně obsluhou) a jimž poskytuje zpětnou vazbu. Jelikož se jedná o univerzální stavební prvek stromu, je jednoduše nahraditelný. V zásadě vykonává tři činnosti:

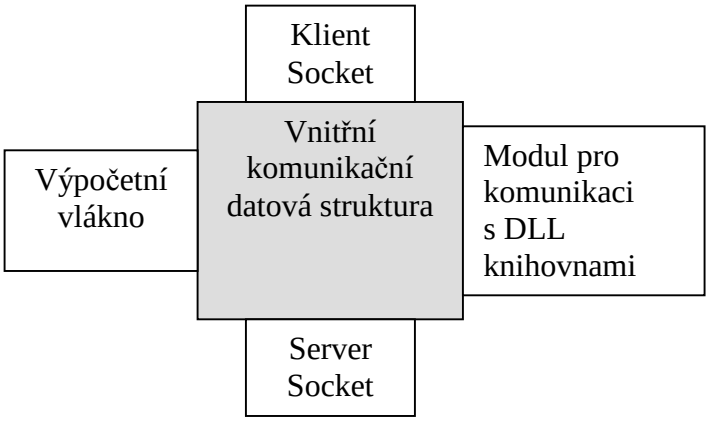
- Zajišťuje přenos informací mezi hierarchickými vrstvami.
- Spravuje seznam podúloh. Tzn. rozděluje úlohy na podúlohy a slučuje partikulární řešení.
- Provádí výpočet zadané úlohy.

Stav	Popis
0	Uzel nově připojený do HDVS. Očekává zaslání informace o řešené úloze.
1	Uzel provádí výpočet podúlohy.
2	Výpočet úlohy byl dokončen, očekává se příjem čísla a mezí další podúlohy.

Tabulka 5-3:Stavy uzlu

Jeho činnost však může být omezena buď na pouhou správu (výkon hostitelského stroje je příliš omezený nebo je vyšší zátěž nežádoucí) či samotný výpočet – případ, kdy se jedná o list stromu).

Skládá se ze čtyř různých částí, komunikujících přes vnitřní statickou datovou strukturu (viz Obrázek 5-1).



Obrázek 5-1:Vnitřní struktura uzlu

5.2.1 Vnitřní komunikační datová struktura

Slouží jako komunikační prostředník mezi jednotlivými částmi uzlu. Komunikační datová struktura je z velké části složena z grafických prvků uživatelského grafického rozhraní. Tím je zamezeno možným nekonzistentnostem mezi vnitřní a vnější reprezentací a snížena paměťová náročnost samotného uzlu. Nevýhodou by mohla být větší procesorová režie. Vzhledem k počtu datových prvků může být zanedbána.

5.2.2 Klient Socket

Jedná se o komunikační kanál spojující uzel s nadřazeným uzlem. Zapouzdřuje v sobě funkčnost TCP/IP socketu a vyššího komunikačního protokolu (viz Tabulka 5-1). Prostřednictvím Klient Socketu je uzel programován a zajišťuje zpětnou vazbu směrem ke kořenu stromu. Dokud není spojení vytvořeno, je považován za kořen celého stromu.

5.2.3 Server Socket

Je tvořen, podobně jako Klient Socket, TCP/IP socketem a vyšším komunikačním protokolem (viz Tabulka 5-2), který používá TCP/IP protokol jako transportní vrstvu. Jedná se o asynchronní (neblokující) server s mechanizmem obsluhy více klientů. K Server Socketu se připojují uzly z nižší hierarchické vrstvy a jsou jím programovány. O list stromu se jedná v případě nulového počtu vytvořených spojení.

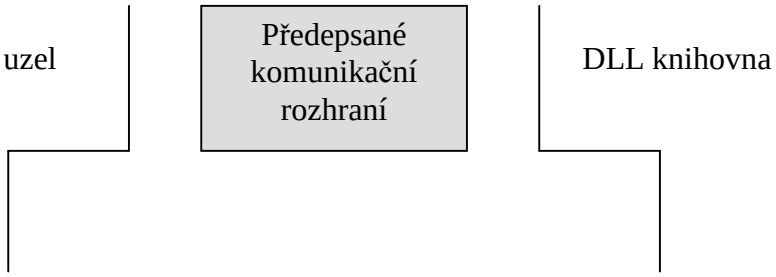
5.2.4 Výpočetní vlákno

V jednovláknovém pojetí problému by uzel při řešení složitých a dlouhotrvajících úloh mohl dospět do stavu, kdy nemá prostor pro vyřizování komunikace v rámci stromu výpočetního systému. Každá podúloha, kterou uzel řeší je vložena do samostatného

výpočetního vlákna a operační systém mu přiřazuje procesorový čas podobně, jako by se jednalo o další proces. V případě dokončení zadané úlohy informuje nadřazený uzel (prostřednictvím Klient Socketu) o jejím výsledku.

5.2.5 Modul pro komunikaci s DLL knihovnami

Každá úloha řešená pomocí HDVS musí být dynamická knihovna (DLL) s předepsaným komunikačním rozhraním. Na žádost obsluhy nebo v hierarchii výše položeného uzlu dojde k natažení knihovny do paměti a její napojení na uzel. Díky Server Socketu modul zajišťuje předání informace o změně načtené knihovny připojeným uzlům. Před ukončením aplikace se stará i o korektní uvolnění knihovny z paměti.



Obrázek 5-2:Standardizovaná komunikace s úlohou

5.3 Komunikační rozhraní mezi HDVS a úlohami

Univerzálnost HDVS spočívá v tom, že není staticky slinkován s jednou nebo několika předem danými úlohami. Každá jednotlivá úloha tvoří samostatnou dynamicky linkovanou knihovnu (DLL). Před začátkem výpočtu je natažena do paměti a po ukončení zase

uvolněna. To však na ní klade nároky v podobě předepsaného komunikačního rozhraní, kterým musí být každá vybavena (viz Obrázek 5-2).

Následující Tabulka 5-4 ukazuje funkce, které tvoří praktické komunikační rozhraní pro napojení úlohy na HDVS. Zároveň je to minimální deklarace DLL knihovny.

`void inicializace(int koren);`
`void deinicializace(int koren);`
`void setKapacity(char *kapacity, char *celkova);`
`int getPocetPoduloh(void);`
`int getPocetVyresenychPoduloh(void);`
`char *getMezePodulohy(int cislo);`
`void setMezeUlohy(char *meze);`
`char *vypocetPodulohy(char *meze);`
`char *sloucitPodulohy(char *scitanec1, char *scitanec2);`
`int getPocetIteraci(void);`
`void setIterace(void);`

Tabulka 5-4:Deklarace funkcí komunikačního rozhraní

Inicializace

Slouží jako konstruktor. Argumentem **koren** HDVS označuje, že se jedná o kořen výpočetního stromu. Typicky použitelné pro definici hodnot, inicializaci případného uživatelského grafického rozhraní.

Deinicializace

Slouží jako destruktor. Argumentem **koren** HDVS označuje, že se jedná o kořen výpočetního stromu. Typicky použitelné pro zpracování výsledku úlohy. Např. zápis do souboru.

setKapacity

Z vektoru **kapacity** a argumentu **celkova** vytvoří seznam kapacit a jim odpovídající seznam mezí pro podúlohy.

getPocetPoduloh

Předává informaci HDVS o počtu podúloh, které je třeba řešit.

getPocetVyresenychPoduloh

Předává informaci HDVS o počtu již vyřešených podúloh.

getMezePodulohy

Předává HDVS vektor s mezemi podúlohy odpovídající argumentu **cislo**. Meze jsou obecný vektor. V rámci jedné úlohy musí být dodržena jednotná konvence pro použitý oddělovač. Pozn. není vhodné používat symboly typické pro např. oddělovač desetinných míst („.“, „,“ apod.).

setMezeUlohy

Nastaví **meze** úlohy, kterou bude uzel řešit. Pro meze platí stejná pravidla, která jsou uvedena výše. Zároveň provádí vynulování hodnoty určující počet již vyřešených podúloh.

vypocetPodulohy

Předává HDVS vektor s výsledkem parciální úlohy vymezené argumentem **meze**. Pro meze i vektor s výsledkem platí stejná pravidla, která jsou uvedena výše.

sloucitPodulohy

Předává HDVS vektor s výsledkem součtu výsledků dvou parciální úloh zadaných argumenty **scitanec1**, **scitanec2**. Pro vektory s výsledkem platí stejná pravidla, která jsou uvedena výše.

getPocetIteraci

Předává informaci HDVS o celkovém počet iterací úlohy.

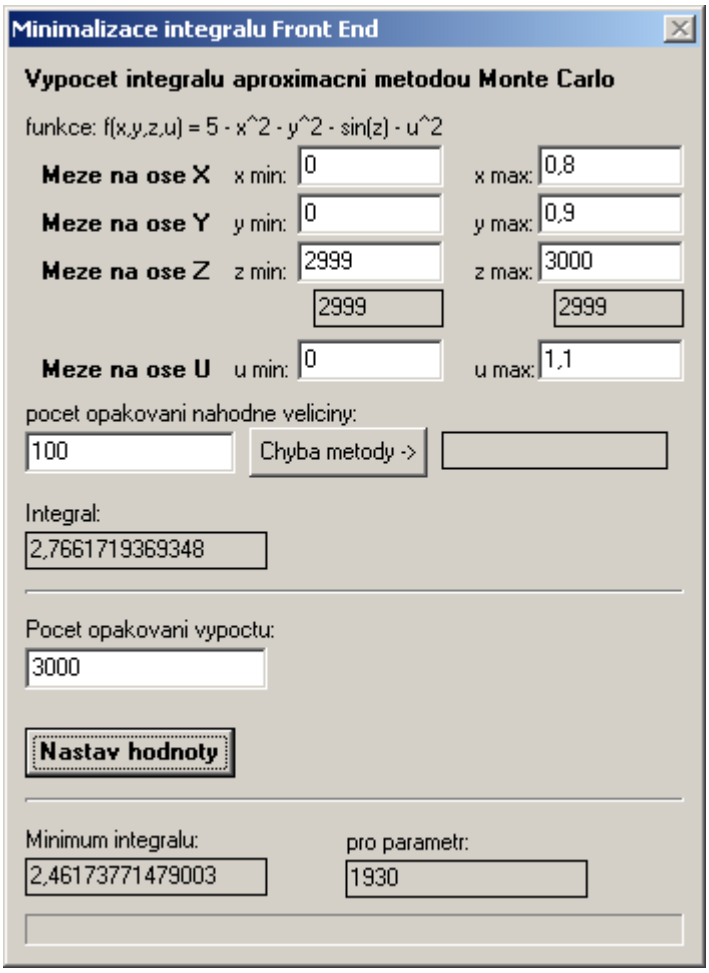
setIterace

Argumentem nastavuje **iteraci** úlohy. Funkce se používá pro přípravu úlohy na další iteraci.

Příklad konkrétní úlohy je uveden v příloze 2.

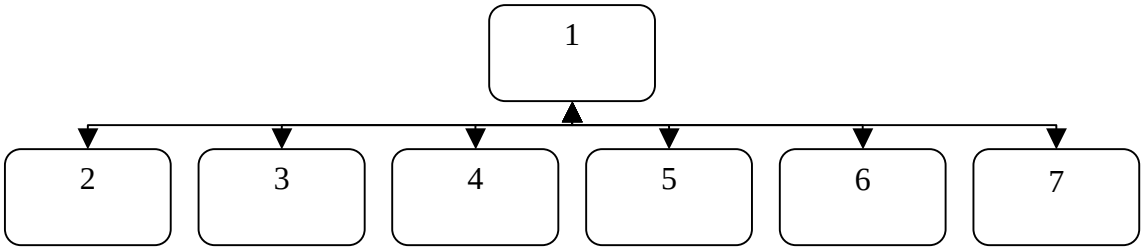
6 Ověření funkčnosti a vlivu topologie HDVS

Funkčnost realizovaného hierarchického výpočetního systému byla ověřena na lokální síti v počítačové laboratoři TUL. Funkčnost a vliv topologie HDVS byla zkoumána na sedmi vybavením shodných PC s předinstalovaným operačním systémem Windows 2000. PC byla spojena ethernetovou sítí se šířkou pásma 100Mbit pomocí aktivního přepínacího síťového prvku (switch). Jako referenční úloha pro výzkum vlivu topologie na rychlost výpočtu byla použita úloha minimalizace integrálu počítaného metodou Monte Carlo (viz příloha 2).

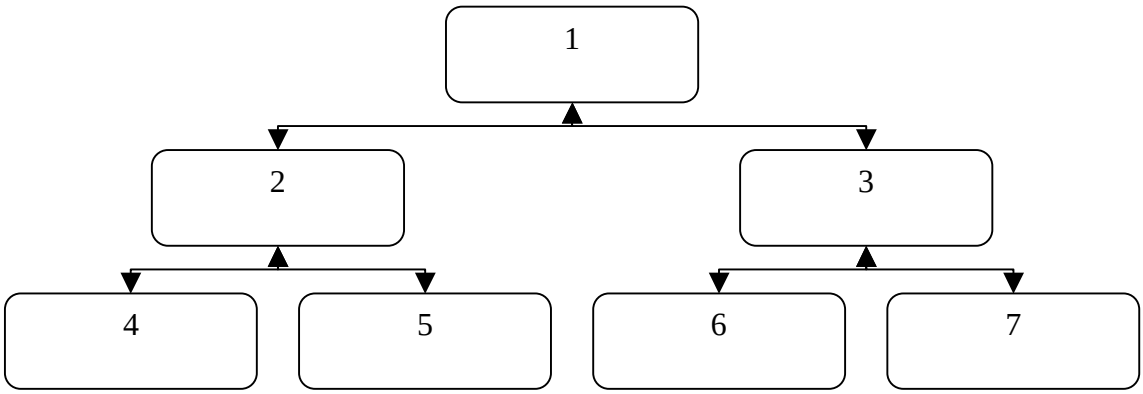


Obrázek 6-1:Uživatelské prostředí realizované úlohy

Podmínky úlohy (viz Obrázek 6-1) byly následující: počet opakování náhodné veličiny pro metodu Monte Carlo $N = 100$, počet iterací úlohy $i = 3000$. Meze na ose z se měnily s počtem iterací $zmin = k$; $zmax = k + 1$; $k = \{0, 1, \dots, 2999\}$.



Obrázek 6-2:Topologie HDVS s jednou hierarchickou vrstvou



Obrázek 6-3:Topologie HDVS se dvěma hierarchickými vrstvami

Doba výpočtu za výše uvedených podmínek v síti s topologií na obrázku (Obrázek 6-2) činila 65,39 [s], v síti s topologií na obrázku (Obrázek 6-3) pak 65,42 [s].

Více o integraci metodou Monte Carlo je možno nalézt v [7] a [8].

7 Závěr

Cílem této práce byla realizace výpočetní architektury typu klient-server v prostředí operačních systémů Windows 2000/XP s využitím nástrojů komunikace mezi procesy obsažených v API Win32. Realizace hierarchického distribuovaného výpočetního systému pro řešení určité třídy problémů, kdy řízení výpočtu probíhá na více úrovních současně, s možností dynamického připojování a odpojování výpočetních uzlů. Návrh a implementace rozhraní umožňujícího zadat úlohu pro realizovaný obecný výpočetní systém pomocí procedur v knihovně DLL. Dále pak demonstrace systému na několika zvolených úlohách.

V průběhu řešení diplomové práce jsem se seznámil s funkcemi a vlastnostmi síťového komunikačního protokolu TCP/IP, objektu pro vícevláknové TThread a možnostmi zavádění a využití funkcí dynamických knihoven DLL.

Realizoval jsem hierarchický distribuovaný výpočetní systém a ověřil jeho funkčnost na konkrétní úloze řešící výpočet minima integrálu čtyřdimenzionální funkce metodou Monte Carlo. Zároveň jsem provedl výzkum vlivu topologie sítě realizovaného HDVS na rychlost výpočtu úlohy. Proti očekávání nebylo dosaženo lepšího výsledku na vícevrstvé topologii. Důvodem může být množství připojených výpočetních uzlů v kombinaci s malý počtem hierarchických vrstev HDVS. Očekávané zrychlení by se projevilo na mnohem rozsáhlejší síti, než kterou jsem měl k dispozici.

Použitá literatura

[1] Motyčková, L.: Distribuované systémy, výpočty v sítích. Science, 1997

[2] Cvejn, J.: Distributed Computation under Microsoft Windows NT. ECMS 2003. 6th International Workshop on Electronics, Control, Measurements and Signals. Technical University of Liberec, Czech Republic, June 2-4, 2003

[3] Microsoft Developer Network Library

[4] Žitník, J.: Komunikace paralelních procesů v prostředí DEC DMS Server Clients. Technická univerzita v Liberci, 1999

[5] Satrapa P.: Počítačové sítě. Technická univerzita v Liberci, Katedra aplikované informatiky

[6] Borland C++ Builder Help

[7] Limpouch J.: Integrace metodou Monte Carlo. České vysoké učení technické v Praze, Fakulta jaderná a fyzikálně inženýrská, 2000

[8] Mathews J. H., Fink K.: Monte Carlo Integration. California State University Fullerton, Department of Mathematics, 2005

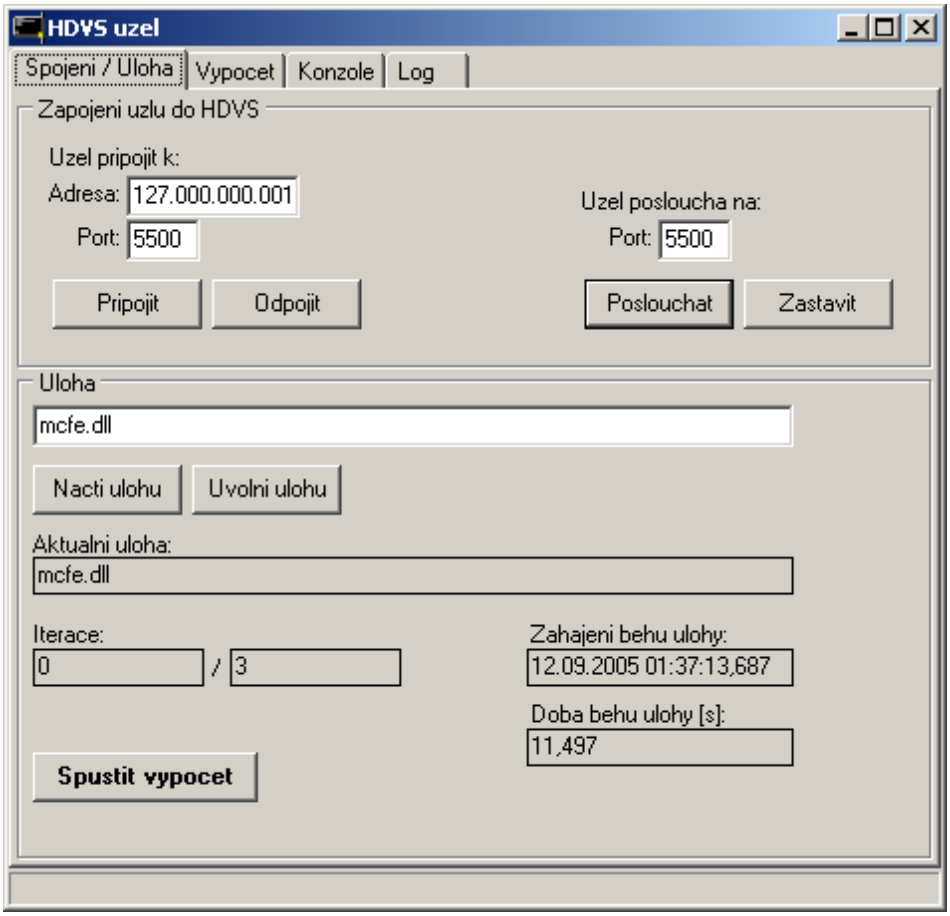
Přílohy

Seznam příloh

1. Popis uživatelského rozhraní Hierarchického distribuovaného výpočetního systému
2. Příklad úlohy pro Hierarchický distribuovaný výpočetní systém
3. CD-ROM se všemi zdrojovými kódy Hierarchického distribuovaného výpočetního systému a vytvořeným příkladem.

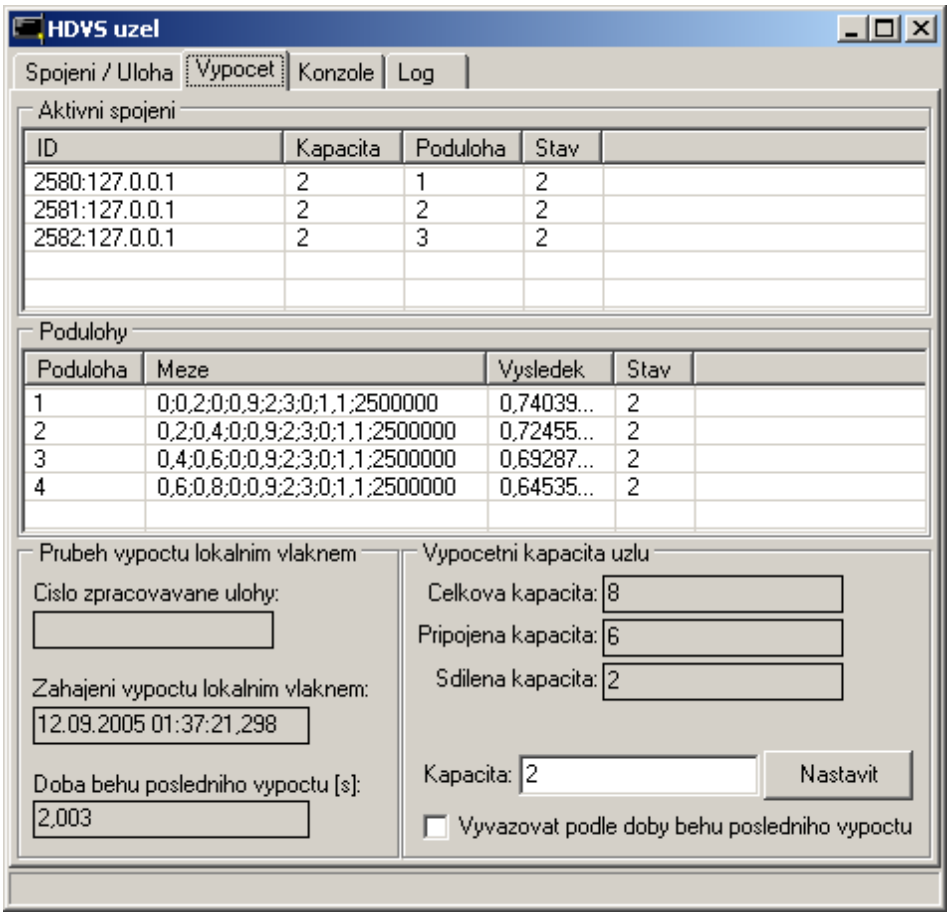
Popis uživatelského rozhraní Hierarchického distribuovaného výpočetního systému

Uživatelské prostředí HDVS je složeno z elementárních ovládacích a zobrazovacích prvků, které nabízí knihovna VCL.



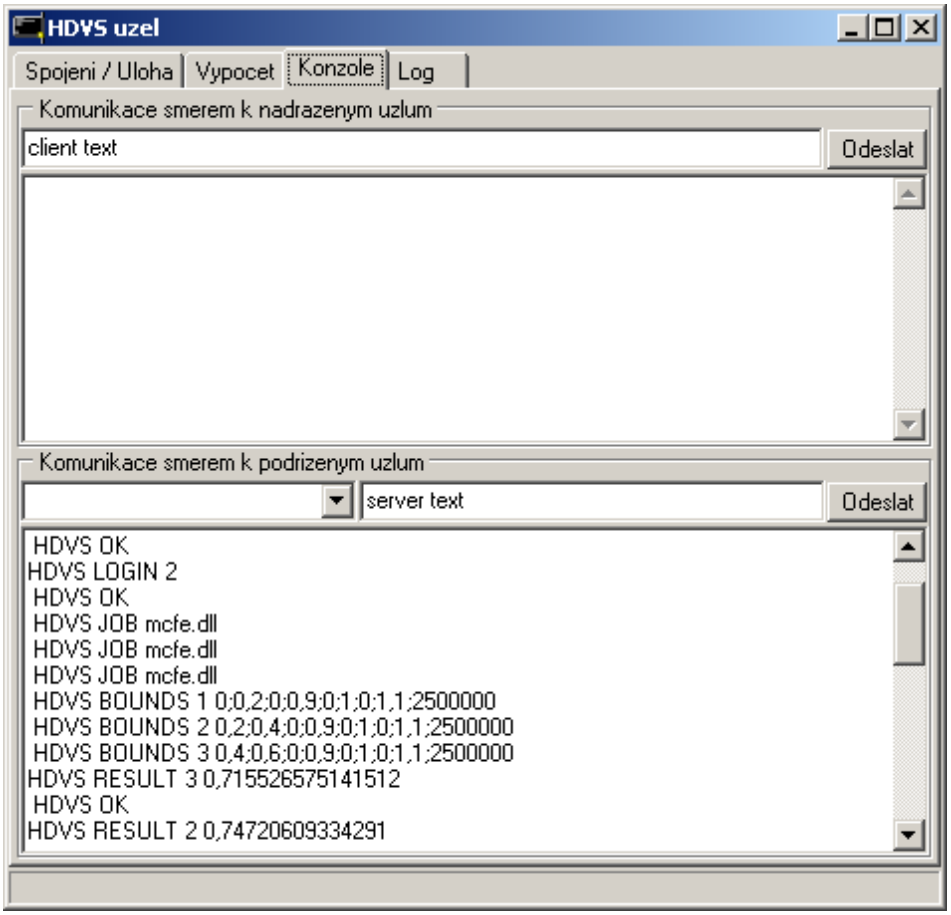
Záložka „**Spojení / Úloha**“ je rozdělena na dvě části:
V první části je možno nastavit připojení uzlu do HDVS zadává se IP adresa a port uzlu, ke kterému se úloha připojí. Připojení a odpojení uzlu se provádí tlačítky „Připojit“ a „Odpojit“. Pokud bude uzel sloužit jako kořen HDVS či jedné z hierarchických vrstev, musí se spustit jeho serverová část tlačítkem „Poslouchat“. Server začne poslouchat na zadaném portu. Tlačítkem „Zastavit“ se serverová část ukončí.

Druhá část „Úloha“ nastavuje název úlohy (dynamické knihovny) pro její zavedení do paměti (tlačítkem „Načti úlohu“) nebo uvolnění z paměti „Uvolni úlohu“. Tlačítkem „Spustit výpočet“ se zahájí výpočet zavedené úlohy. O právě zavedené úloze informuje pole „Aktuální úloha“. Pole „Zahájení běhu úlohy“ a „Doba běhu úlohy“ časovou značkou informuje uživatele o zahájení výpočtu respektive o době jeho trvání.

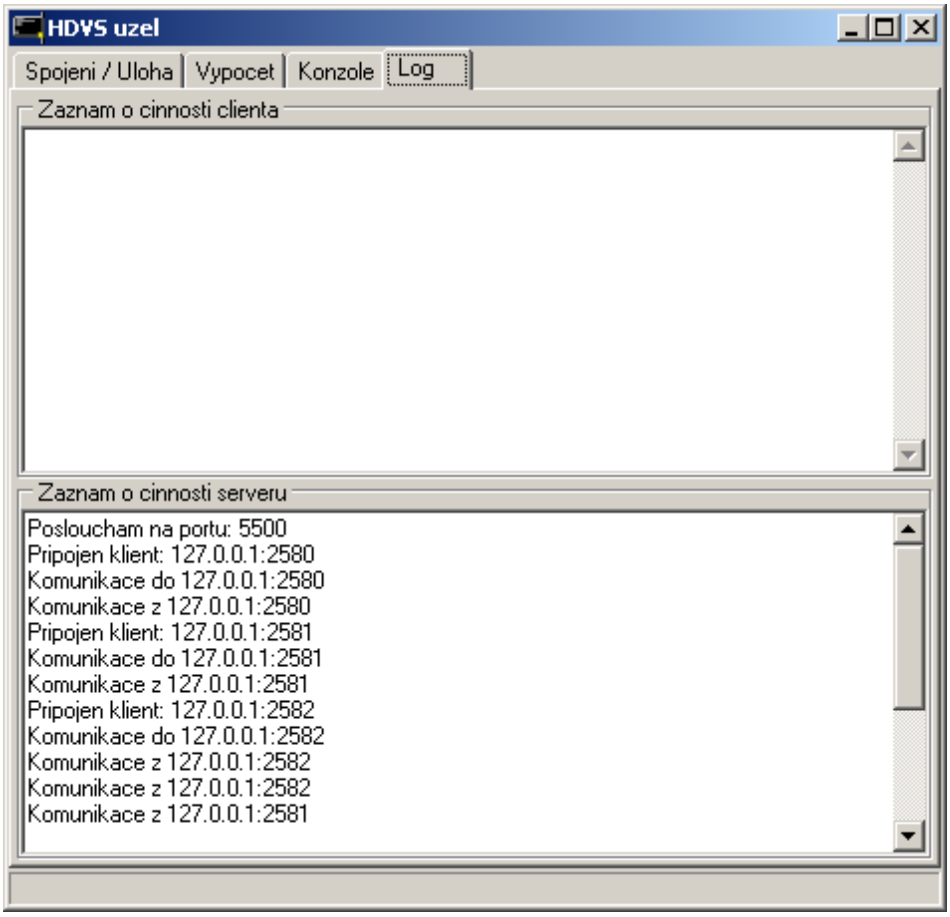


Záložka „**Výpočet**“ je rozdělena na čtyři, převážně pouze informativní části. Část nazvaná „Aktivní spojení“ informuje uživatele o počtu a parametrech připojených poduzlů. Část „Podúlohy“ obsahuje zásobník všech podúloh, které pro daný uzel zpracovávaná úloha vygenerovala. Část „Průběh výpočtu lokálním vláknem“ zobrazuje informace o čísle a průběhu výpočtu, který provádí samotný uzel. Čtvrtá, interaktivní část „Výpočetní kapacita uzlu“ informuje o poskytované výpočetní kapacitě uzlu a připojených poduzlů. V poli „Kapacita“ může uživatel zadat požadovanou výpočetní kapacitu podle svého

uvážení. Specifickou vlastnost má hodnota „0“, která způsobí, že se uzel nebude na výpočtu podúloh podílet. V tomto případě jeho funkce spočívá pouze v generování úloh a slučování výsledků z připojených poduzlů. Automatické adaptivní nastavení výpočetní kapacity uzlu se provádí funkcí „Vyvažovat podle doby běhu posledního výpočtu“.



Záložka „**Konzole**“ zobrazuje průběh komunikace mezi klientem a serverem na vyšším komunikačním protokolu. Uživatel může do komunikace aktivně vstupovat.



Záložka „**Log**“ podrobně zaznamenává průběh komunikace transportní TCP/IP vrstvy a je určena pro kontrolu běhu systému.

Příklad úlohy pro Hierarchický distribuovaný výpočetní systém

Následující výpis zdrojového kódu uvádí jako příklad úlohy pro HDVS úlohu výpočtu minima integrálu vypočteného aproximační metodou Monte Carlo.

```
//---------------------------------------------------------------------------

#include <vcl.h>
#include <windows.h>
#include <time.h>
#include <math.h>
#include "fe.h"
#pragma hdrstop
//---------------------------------------------------------------------------

// funkce slouzi jako komunikaceni rozhrani mezi ulohou a HDVS
extern "C" __declspec(dllexport)void inicializace(int koren);
extern "C" __declspec(dllexport)void deinicializace(int koren);
extern "C" __declspec(dllexport)void setKapacity(char *kapacity,
                                                 char *celkova);
extern "C" __declspec(dllexport)int getPocetPoduloh(void);
extern "C" __declspec(dllexport)int getPocetVyresenychPoduloh(void);
extern "C" __declspec(dllexport)char *getMezePodulohy(int cislo);
extern "C" __declspec(dllexport)void setMezeUlohy(char *meze);
extern "C" __declspec(dllexport)char *vypocetPodulohy(char *meze);
extern "C" __declspec(dllexport)char *sloucitPodulohy(char *scitanec1,
                                                      char *scitanec2);
extern "C" __declspec(dllexport)int getPocetIteraci(void);
extern "C" __declspec(dllexport)void setIterace(int iterace);
//---------------------------------------------------------------------------

// deklarace funkci zavislych na konkretni uloze
void nastavHodnoty();
//---------------------------------------------------------------------------
```

```
// deklarace globalnich promennych nutnych pro funkcnost ulohy
int pocetPoduloh, pocetVyresenychPoduloh,Iterace,pocetIteraci;
AnsiString ansiMezisoucet, ansiVysledek;
TStringList *ansiSeznamKapacit, *ansiSeznamMezi;

// deklarace globalnich promennych zavislych na konkretni uloze
double Xmin, Xmax, Ymin, Ymax, Zmin, Zmax, Umin, Umax;
double xmin, xmax, ymin, ymax, zmin, zmax, umin, umax;
int N, n;
//---------------------------------------------------------------------------
int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason,
                         void* lpReserved)
{
  return 1;
}
//---------------------------------------------------------------------------
void inicializace(int koren)
{
  //slouzi jako konstruktor
  //typicky pouzitelne pro definici hodnot

  pocetPoduloh = 0;
  pocetVyresenychPoduloh = 0;
  ansiSeznamKapacit = new TStringList;
  ansiSeznamMezi = new TStringList;
  Iterace = 0;
  pocetIteraci = 3;



  //vypocet 4dimenzionalniho integralu
  //aproximacni metodou Monte Carlo
  //
  //f(x,y,z,u) = 5 - x^2 - y^2 - sin(z) - u^2

  //vytoreni instance a zobrazeni uzivatelskeho frontendu
  //(pokud se jedna o koren)
  FormFE = new TFormFE(Application, &nastavHodnoty);
  if(koren) FormFE->Show();
```

```
  //celkove meze integralu
  Xmin = 0; Xmax = 4.0/5.0;
  Ymin = 0; Ymax = 9.0/10.0;
  Zmin = 0; Zmax = 1.0;
  Umin = 0; Umax = 11.0/10.0;

  //N - pocet opakovani nahodne veliciny pro celou ulohu
  N = 10000000;

  //inicializace hodnot ve formulari
  FormFE->EditXMin->Text = FloatToStr(Xmin);
  FormFE->EditXMax->Text = FloatToStr(Xmax);
  FormFE->EditYMin->Text = FloatToStr(Ymin);
  FormFE->EditYMax->Text = FloatToStr(Ymax);
  FormFE->EditZMin->Text = FloatToStr(Zmin);
  FormFE->EditZMax->Text = FloatToStr(Zmax);
  FormFE->EditUMin->Text = FloatToStr(Umin);
  FormFE->EditUMax->Text = FloatToStr(Umax);

  FormFE->EditN->Text = IntToStr(N);
  FormFE->EditPocetOpakovani->Text = IntToStr(pocetIteraci);
  FormFE->ProgressBar1->Max = pocetIteraci;

  //inicializace mezi integralu
  //meze podulohy
  xmin = Xmin; xmax = Xmax;
  ymin = Ymin; ymax = Ymax;
  zmin = Zmin; zmax = Zmax;
  umin = Umin; umax = Umax;
  //pocet opakovani nahodne veliciny
  n = N;
}
//---------------------------------------------------------------------------
void deinicializace(int koren)
{
  //slouzi jako destruktor, argumentem koren HDVS oznacuje,
  //zda se jedna o koren vypocetniho stromu
  //typicky pouzitelne pro zpracovani vysledku ulohy
  //(napr. zapis do souboru)
```

```
  int handle;

  delete ansiSeznamMezi;
  delete ansiSeznamKapacit;

  if(koren == 1){
    handle = FileCreate("vysledek.txt");
    FileWrite(handle, FormFE->EditMinimum->Text.c_str(),
              FormFE->EditMinimum->Text.Length());
    FileWrite(handle, "\n",1);
    FileWrite(handle, FormFE->EditParametr->Text.c_str(),
              FormFE->EditParametr->Text.Length());
    FileClose(handle);
  }
  //zruseni instance uzivatelskeho frontendu
  delete FormFE;
}
//---------------------------------------------------------------------------
void setKapacity(char *kapacity, char *celkova)
{
  //vytvori seznam kapacit a jim odpovidajici seznam mezi

  AnsiString ansiKapacity, ansiCelkovaKapacita, ansiMeze;
  AnsiString ansiXMax, ansiYMax, ansiZMax, ansiUMax, ansiN;
  int pos, i;
  double fragmentx, fragmenty, fragmentz, fragmentu, fragmentn;


  ansiKapacity = kapacity;
  ansiCelkovaKapacita = celkova;

  //smazani seznamu kapacit a mezi
  ansiSeznamKapacit->Clear();
  ansiSeznamMezi->Clear();

  //lexikalni analyza
  while(ansiKapacity.Length()>0){
    pos = ansiKapacity.Pos(";");
```

```
  if(pos > 0){
    ansiSeznamKapacit->Add(ansiKapacity.SubString(1,pos - 1));
    ansiKapacity.Delete(1,pos);
  }
  else{
    ansiSeznamKapacit->
      Add(ansiKapacity.SubString(1,ansiKapacity.Length()));
    ansiKapacity.Delete(1,ansiKapacity.Length());
  }
}

//pocet poduloh
pocetPoduloh = ansiSeznamKapacit->Count;

fragmentx = (xmax - xmin) / StrToFloat(ansiCelkovaKapacita);
fragmentn = (double)n / StrToFloat(ansiCelkovaKapacita);

//vytvoreni seznamu mezi odpovidajici kapacitam
for(i = 0; i < pocetPoduloh; i++){
  ansiMeze = "";
  if(i == 0){
    //x
    ansiXMax =
      FloatToStr(xmin + StrToFloat(ansiSeznamKapacit->Strings[i]) *
                 fragmentx);
    ansiMeze = FloatToStr(xmin)+ ";" + ansiXMax;
  }
  else{
    //x
    ansiMeze = ansiXMax + ";";
    ansiXMax =
      FloatToStr(StrToFloat(ansiXMax) +
                 StrToFloat(ansiSeznamKapacit->Strings[i]) *
                 fragmentx);
    ansiMeze = ansiMeze + ansiXMax;
  }
  ansiN = IntToStr((int)(StrToFloat(ansiSeznamKapacit->Strings[i]) *
                 fragmentn));
  //y,z,u
```

```
    ansiMeze = ansiMeze + ";" + FloatToStr(ymin)+ ";" +
               FloatToStr(ymax) + ";" + FloatToStr(zmin)+ ";" +
               FloatToStr(zmax) + ";" + FloatToStr(umin)+ ";" +
               FloatToStr(umax) + ";" + ansiN;
    ansiSeznamMezi->Add(ansiMeze);
  }
}
//---------------------------------------------------------------------------
int getPocetPoduloh(void)
{
  //vraci pocet poduloh

  return pocetPoduloh;
}
//---------------------------------------------------------------------------
int getPocetVyresenychPoduloh(void)
{
  //vraci pocet jiz vyresenych poduloh

  return pocetVyresenychPoduloh;
}
//---------------------------------------------------------------------------
char *getMezePodulohy(int cislo)
{
  //vraci meze podle cisla
  //meze jsou obecny vektor, v uloze musi byt dodrzena jednotna konvence
  //pro pouzity oddelovac
  //pozn. neni vhodne pouzivat symboly typicke pro napr. oddelovac
  //desetinnych mist ("." "," apod.)

  return ansiSeznamMezi->Strings[cislo].c_str();
}
//---------------------------------------------------------------------------
void setMezeUlohy(char *meze)
{
  //nastavi meze ulohy
  //meze jsou obecny vektor, v uloze musi byt dodrzena jednotna konvence
  //pro pouzity oddelovac
  //pozn. neni vhodne pouzivat symboly typicke pro napr. oddelovac
```

```
//desetinnych mist (".", "," apod.)
//resetuje pocet jiz vyresenych poduloh

int oddelovac;
AnsiString ansiMeze;

//reset poctu vyresenych poduloh
pocetVyresenychPoduloh = 0;

ansiMeze = meze;
//x
oddelovac = ansiMeze.Pos(";");
xmin = StrToFloat(ansiMeze.SubString(1,oddelovac-1));
ansiMeze = ansiMeze.Delete(1,oddelovac);
oddelovac = ansiMeze.Pos(";");
xmax = StrToFloat(ansiMeze.SubString(1,oddelovac-1));
ansiMeze = ansiMeze.Delete(1,oddelovac);
//y
oddelovac = ansiMeze.Pos(";");
ymin = StrToFloat(ansiMeze.SubString(1,oddelovac-1));
ansiMeze = ansiMeze.Delete(1,oddelovac);
oddelovac = ansiMeze.Pos(";");
ymax = StrToFloat(ansiMeze.SubString(1,oddelovac-1));
ansiMeze = ansiMeze.Delete(1,oddelovac);
//z
oddelovac = ansiMeze.Pos(";");
zmin = StrToFloat(ansiMeze.SubString(1,oddelovac-1));
ansiMeze = ansiMeze.Delete(1,oddelovac);
oddelovac = ansiMeze.Pos(";");
zmax = StrToFloat(ansiMeze.SubString(1,oddelovac-1));
ansiMeze = ansiMeze.Delete(1,oddelovac);
//u
oddelovac = ansiMeze.Pos(";");
umin = StrToFloat(ansiMeze.SubString(1,oddelovac-1));
ansiMeze = ansiMeze.Delete(1,oddelovac);
oddelovac = ansiMeze.Pos(";");
umax = StrToFloat(ansiMeze.SubString(1,oddelovac-1));
//n
n = StrToInt(ansiMeze.SubString(oddelovac + 1,
```

```
                                    ansiMeze.Length()-oddelovac));

}
//---------------------------------------------------------------------------
char *vypocetPodulohy(char *meze)
{
  //slouzi k vypoctu podulohy
  //meze i vysledek jsou obecne vektory, v uloze musi byt dodrzena
  //jednotna konvence pro pouzity oddelovac
  //pozn. neni vhodne pouzivat symboly typicke pro napr. oddelovac
  //desetinnych mist ("." "," apod.)

  int oddelovac, i, n1;
  double x1,x2,y1,y2,z1,z2,u1,u2;
  double xi,yi,zi,ui;
  double f,v, suma = 0;
  AnsiString ansiMeze;
  time_t t;

  ansiMeze = meze;
  //x
  oddelovac = ansiMeze.Pos(";");
  x1 = StrToFloat(ansiMeze.SubString(1,oddelovac-1));
  ansiMeze = ansiMeze.Delete(1,oddelovac);
  oddelovac = ansiMeze.Pos(";");
  x2 = StrToFloat(ansiMeze.SubString(1,oddelovac-1));
  ansiMeze = ansiMeze.Delete(1,oddelovac);
  //y
  oddelovac = ansiMeze.Pos(";");
  y1 = StrToFloat(ansiMeze.SubString(1,oddelovac-1));
  ansiMeze = ansiMeze.Delete(1,oddelovac);
  oddelovac = ansiMeze.Pos(";");
  y2 = StrToFloat(ansiMeze.SubString(1,oddelovac-1));
  ansiMeze = ansiMeze.Delete(1,oddelovac);
  //z
  oddelovac = ansiMeze.Pos(";");
  z1 = StrToFloat(ansiMeze.SubString(1,oddelovac-1));
  ansiMeze = ansiMeze.Delete(1,oddelovac);
  oddelovac = ansiMeze.Pos(";");
```

```
  z2 = StrToFloat(ansiMeze.SubString(1,oddelovac-1));
  ansiMeze = ansiMeze.Delete(1,oddelovac);
  //u
  oddelovac = ansiMeze.Pos(";");
  u1 = StrToFloat(ansiMeze.SubString(1,oddelovac-1));
  ansiMeze = ansiMeze.Delete(1,oddelovac);
  oddelovac = ansiMeze.Pos(";");
  u2 = StrToFloat(ansiMeze.SubString(1,oddelovac-1));
  //n
  n1 = StrToInt(ansiMeze.SubString(oddelovac + 1,
                                   ansiMeze.Length()-oddelovac));

  //"objem" integracni oblasti
  v = (x2-x1)*(y2-y1)*(z2-z1)*(u2-u1);
  //inicializace generatoru pseudonahodnych cisel
  srand((unsigned) time(&t));
  for(i = 0; i < n1; i++){
    //generovani pseudonahodnych velicin v danych intervalech
    xi = x1+ ((double)rand()/((double)(RAND_MAX)+(double)(1))) * (x2-x1);
    yi = y1+ ((double)rand()/((double)(RAND_MAX)+(double)(1))) * (y2-y1);
    zi = z1+ ((double)rand()/((double)(RAND_MAX)+(double)(1))) * (z2-z1);
    ui = u1+ ((double)rand()/((double)(RAND_MAX)+(double)(1))) * (u2-u1);
    suma = suma + 5 - xi*xi - yi*yi - sin(zi) - ui*ui;
  }

  f = suma / (double)n1;

  ansiVysledek = FloatToStr(v * f);

  //inkrementuje pocet jiz vyresenych poduloh
  pocetVyresenychPoduloh++;

  return ansiVysledek.c_str();
}
//---------------------------------------------------------------------------
char *sloucitPodulohy(char *scitanec1, char *scitanec2)
{
  //soucet dvou poduloh
  //scitance i vysledek jsou obecne vektory, v uloze musi byt dodrzena
```

```
  //jednotna konvence pro pouzity oddelovac
  //pozn. neni vhodne pouzivat symboly typicke pro napr. oddelovac
  //desetinnych mist ("." "," apod.)

  AnsiString ansiScitanec1, ansiScitanec2;

  ansiScitanec1 = scitanec1;
  ansiScitanec2 = scitanec2;
  if(ansiScitanec1.IsEmpty()) ansiScitanec1 = "0";
  if(ansiScitanec2.IsEmpty()) ansiScitanec2 = "0";
  ansiMezisoucet = FloatToStr(StrToFloat(ansiScitanec1) +
                              StrToFloat(ansiScitanec2));

/*
  //zapsat aktualni hodnotu integralu
  FormFE->EditIntegral->Text = ansiMezisoucet;

  if(Iterace == 0){
    //pocatecni podminky
    FormFE->EditMinimum->Text = ansiMezisoucet;
    FormFE->EditParametr->Text = "0";
  }
*/

  return ansiMezisoucet.c_str();
}
//---------------------------------------------------------------------------
int getPocetIteraci(void)
{
  //vraci celkovy pocet iteraci ulohy

  return pocetIteraci;
}
//---------------------------------------------------------------------------
void setIterace(int iterace)
{
  //nastavuje iteraci ulohy

  Iterace = iterace;
```

```
//zapsat aktualni hodnotu integralu
FormFE->EditIntegral->Text = ansiMezisoucet;

if(Iterace == 1){
  //pocatecni podminky
  FormFE->EditMinimum->Text = ansiMezisoucet;
  FormFE->EditParametr->Text = StrToInt(Iterace - 1);
}
if(Iterace > 1){
  //minimum
  if(StrToFloat(FormFE->EditIntegral->Text) <
    StrToFloat(FormFE->EditMinimum->Text)){
    FormFE->EditMinimum->Text = FormFE->EditIntegral->Text;
    //pro
    FormFE->EditParametr->Text = StrToInt(Iterace - 1);
  }
}
if(Iterace > 0 && Iterace < pocetIteraci){
  //zmena mezi osy Z
  FormFE->EditZMin->Text =
    FloatToStr(StrToFloat(FormFE->EditZMin->Text) + 1);
  FormFE->EditZMax->Text =
    FloatToStr(StrToFloat(FormFE->EditZMax->Text) + 1);
  FormFE->EditParametrZ1->Text = StrToInt(Iterace);
  FormFE->EditParametrZ2->Text = StrToInt(Iterace);

  //prevzeti hodnot z formulare
  Zmin = StrToFloat(FormFE->EditZMin->Text);
  Zmax = StrToFloat(FormFE->EditZMax->Text);

  //meze podulohy
  zmin = Zmin; zmax = Zmax;
}

if(Iterace < pocetIteraci){
  FormFE->ProgressBar1->Position = Iterace + 1;
}
else{
```

```
    FormFE->ProgressBar1->Position = 0;
  }
}
//---------------------------------------------------------------------------
void nastavHodnoty(){
  //funkci vola formular

  //prevzeti hodnot z formulare
  Xmin = StrToFloat(FormFE->EditXMin->Text);
  Xmax = StrToFloat(FormFE->EditXMax->Text);
  Ymin = StrToFloat(FormFE->EditYMin->Text);
  Ymax = StrToFloat(FormFE->EditYMax->Text);
  Zmin = StrToFloat(FormFE->EditZMin->Text);
  Zmax = StrToFloat(FormFE->EditZMax->Text);
  Umin = StrToFloat(FormFE->EditUMin->Text);
  Umax = StrToFloat(FormFE->EditUMax->Text);

  N = StrToFloat(FormFE->EditN->Text);
  pocetIteraci = StrToFloat(FormFE->EditPocetOpakovani->Text);
  FormFE->ProgressBar1->Max = pocetIteraci;

  //reset hodnot
  FormFE->EditIntegral->Text = "";
  FormFE->EditMinimum->Text = "";
  FormFE->EditParametr->Text = "";
  FormFE->EditParametrZ1->Text = "0";
  FormFE->EditParametrZ2->Text = "0";

  //meze podulohy
  xmin = Xmin; xmax = Xmax;
  ymin = Ymin; ymax = Ymax;
  zmin = Zmin; zmax = Zmax;
  umin = Umin; umax = Umax;

  //pocet opakovani nahodne veliciny
  n = N;
}
//---------------------------------------------------------------------------
```

Prohlášení

Byl jsem seznámen s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 o právu autorském, zejména § 60 (školní dílo) a § 35 (o nevýdělečném užití díla k vnitřní potřebě školy).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé práce a prohlašuji, že souhlasím s případným užitím mé práce (prodej, zapůjčení, apod.).

Jsem si vědom toho, že užít své diplomové práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů vynaložených univerzitou na vytvoření díla (až do jejich skutečné výše).

V Liberci 1.5.2005 Pavel Paleček