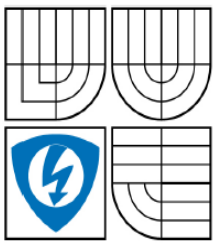


VYSOKÉ UČENÍ
TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

Ústav automatizace a měřicí techniky

# Bakalářská práce

bakalářský studijní obor
**Automatizační a měřicí technika**

***Student:*** Ondřej Mikulín ***ID:*** 106645
***Ročník:*** 3 ***Akademický rok:*** 2009/2010

**NÁZEV TÉMATU:**

## Implementace RapidMineru na výpočetním GRIDu

**POKYNY PRO VYPRACOVÁNÍ:**

Seznamte se s programováním a distribucí úloh na výpočetním GRIDu Alchemi a se základní obsluhou programu RapidMiner pro zpracování dat a modelování. Vytvořte program s rozhraním podobným RapidMineru (např. modulu GridSearch), který umožní distribuci výpočtů na více počítačových stanic. Proveďte několik experimentů a srovnejte dobu výpočtu při různě velkém počtu exekutorů výpočetního GRIDu.

**DOPORUČENÁ LITERATURA:**

Nagel Ch. - kol.: C# 2008. Praha, Computer Press, 2008.

***Termín zadání:*** 8.2.2010 ***Termín odevzdání:*** 31.5.2010

***Vedoucí práce:*** Ing. Petr Honzík, Ph.D.

**prof. Ing. Pavel Jura, CSc.**
*Předseda oborové rady*

**UPOZORNĚNÍ:**

**ABSTRAKT**

Tato práce se zabývá integrací programu RapidMiner do GRIDového prostředí a paralelizací úlohy hledání optimálních parametrů matematických modelů. Vše je realizováno s ohledem na odlišnou časovou náročnost dílčích výpočtů. Výsledkem je aplikace RapidParallel postavená na GPL softwaru.

**KLÍČOVÁ SLOVA**
GRID, paralelní algoritmy, RapidMiner

**ABSTRACT**

The aim of this work is integration of RapidMiner into the GRID environment and parallelization of mathematical model optimization. Different subtask time complexity was considered during realization. Result of this work is application RapidParallel based on GPL software.

**KEYWORDS**

GRID, parallel agorithms, RapidMiner

**Bibliografická citace mé práce:**
MIKULÍN, O. *Implementace RapidMineru na výpočetním GRIDu.* Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2010. XY s. Vedoucí bakalářské práce Ing. Petr Honzík, Ph.D.

## P ROHLÁŠENÍ

„Prohlašuji, že svou bakalářskou práci na téma „Implementace RapidMineru na výpočetním GRIDu" jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.“

V Brně dne: **1. června 2009** ……
podpis autora

## PODĚKOVÁNÍ

Děkuji vedoucímu bakalářské práce Ing. Petru Honzíkovi Ph.D. za účinnou metodickou, pedagogickou a odbornou pomoc a další cenné rady při zpracování mé bakalářské práce.

V Brně dne: **1. června 2009**

……
podpis autora

**OBSAH**

# 1. ÚVOD

Ve vědě a technice se i přes vzrůstající rychlost počítačů stále vyskytují problémy, k jejichž řešení v rozumném čase je potřeba větší výpočetní síla než je momentálně standardně k dispozici. Zlepšování hardwaru počítačů je jednou z cest vedoucích ke zrychlování výpočtů. Levnější a současně dostupnou alternativu představuje použití několika méně výkonných strojů společně řešících jeden konkrétní problém. Nižší nároky na hardware než v předchozím případě jsou vykoupeny větší složitostí paralelně fungujících aplikací. K usnadnění distribuce výpočtů je k dispozici velké množství softwarových prostředků. Mnoho z nich je možné za jistých podmínek využívat zcela zdarma. Ve firmách nebo školách je navíc často procentuální využití počítačů velmi malé. Taková situace umožňuje využít stávající vybavení v době kdy není plně využíváno, bez nutnosti nákupu dalšího. To jsou některé z důvodů vedoucích ke stále vzrůstající popularitě paralelních architektur jakou je například výpočetní GRID.

Tato práce se zabývá možností integrace java aplikace RapidMiner do výpočetního GRIDu. RapidMiner je primárně určen k provádění výpočtů souvisejících s umělou inteligencí. Některé z těchto postupů mohou být při větším množství zpracovávaných dat a sekvenčním provádění algoritmů časově velmi náročné. Jedním z výsledků této práce a zároveň částečnou možností řešení výše zmíněného problému, by měla být aplikace (dále RapidParallel) tvořící mezičlánek spojující software obsluhující výpočetní GRID a RapidMiner. Společně s několika dalšími knihovnami metod by měla poskytnout platformu, na které by mohly jednodušeji vznikat paralelní algoritmy využívající program RapidMiner jako výpočetní jádro.

# 2. OBECNÉ INFORMACE

V této kapitole budou vysvětleny základní, dále využívané pojmy. Jedná se především o definici GRIDu, shrnutí dostupných softwarových prostředků a terminologie paralelního programování. Závěr patří stručnému objasnění pojmu neuronová síť a optimalizaci jejích parametrů.

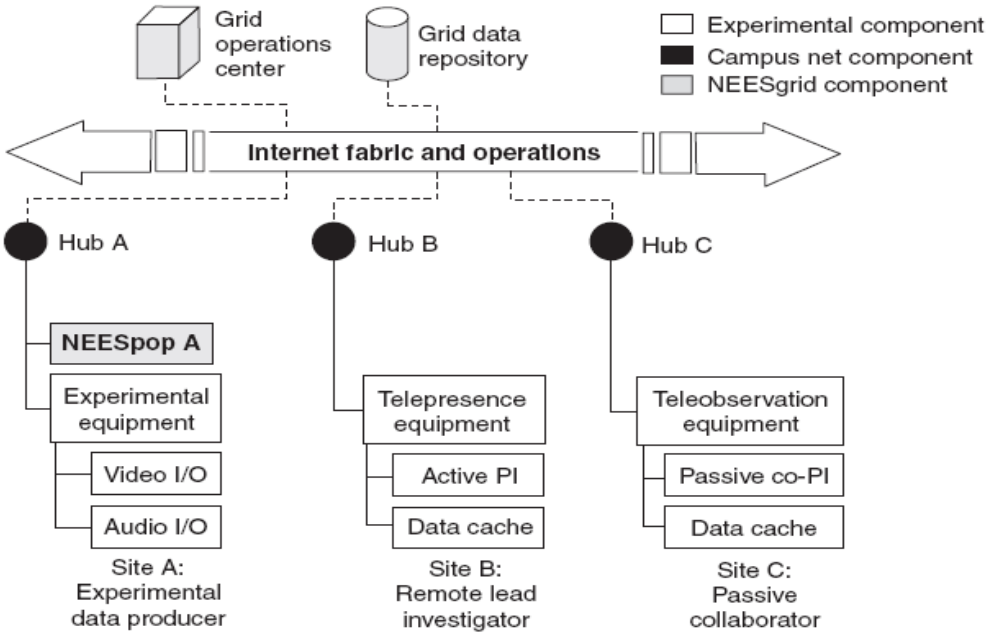
## 2.1 GRID

V definici z [1] je psáno:
*GRID je typ paralelního a distribuovaného systému, který umožňuje výběr, sdílení a shlukování geograficky rozdělených autonomních výpočetních zdrojů, dynamicky za běhu v závislosti na jejich dostupnosti, použitelnosti, výkonu, ceně a požadavcích uživatele.*

**Obrázek 2.1: schéma výpočetního GRIDu , převzato z [7]**

Definici z [1] a základní představu o GRIDu vystihuje obrázek 2.1. Samotný pojem GRID je poměrně široký a rozdílně vykládaný. Stejně rozsáhlé je i spektrum

vlastností této paralelní architektury umožňující mimo jiné sdílení drahých, jinak nedostupných přístrojů, experimentálně získaných dat a také vědomostí. Jsou to atributy, které umožnily existenci mezinárodních projektů jako je NEES GRID sdružující odborníky z oblasti zemětřesení. Vzdálená pracoviště prostřednictvím GRIDu vybudovaného na softwaru Globus toolkit sdílejí vybavení pro testování a simulaci materiálů a struktur odolávajících zemětřesení. Schéma NEES GRIDu je na obrázku 2.2. Dalším známým vědeckým projektem je Worldwide LHC Computing GRID, který ukládá a zpracovává obrovské množství dat získaných z urychlovače částic. Vědci tak mohou spolupracovat bez ohledu na geografické hranice, společně provádět náročné měření nebo složité simulace. GRID není jen výsadou vědy a techniky. O jeho použití se uvažuje i v herním průmyslu, ale i v každodením životě. Do určité míry se jedná o další rozšíření možností internetu tak, aby byla dostupná velká výpočetní síla a obsáhlý úložný prostor pro každého a to jenom ve chvíli, kdy je zrovna potřebuje. V GRIDu je často spatřována budoucnost výpočetních technologií.




**Obrázek 2.2 struktura NEES GRIDu, převzato z [1]**

Výpočetní GRID na VUT je realizován GPL softwarem Alchemi vyvíjeným na univerzitě v Melbourne. Tento framework poskytuje všechny služby typické pro GRID, jako je zajištění bezpečnosti, abstrakce výpočetních a úložných zdrojů. Uživatel nemusí řešit jak se provede jeho výpočet a kde přesně budou uložena jeho data. Několik počítačů se potom z vnějšího pohledu chová jako jedno homogenní zařízení. Alchemi je software typu plug&play. Instalace a konfigurace je poměrně rychlá, jednoduchá a nevyžaduje žádné hlubší znalosti tvorby výpočetních GRIDů, narozdíl od prostředků typu Globus toolkit. Terminologie Alchemi uvedená například v [4] rozeznává dva druhy aplikací, které je možné spouštět na GRIDu:

- **Grid-Thread**

  Tento objektově orientovaný model GRIDové aplikace zpracovává paralelně pouze označenou část zdrojového kódu, která se provede ve vláknech odesílaných a zpracovaných na počítačích s aktivním Executorem (pojem Executor je vysvětlen níže). Je to asi nejvíce používaný přístup. Limitujícím nedostatkem Grid-Threadů je velmi omezená komunikace mezi jednotlivými vlákny a managerem. Přímé zasílání zpráv mezi vlákny není přítomno vůbec. Řešení této skutečnosti je ponecháno na programátorovi.

- **Grid-Job**

  Model Grid-Job je vyšší abstrakcí než předchozí přístup. Uvádí se, že je vhodný zejména při začleňování jiných aplikací do své vlastní. Pro vytvoření paralelně vykonávaného jobu na GRIDu je nutné specifikovat tři údaje. Jsou to vstupní soubory, výstupní soubory a posledním z parametrů je proces. Oproti předchozímu přístupu má programátor menší volnost, což může vést k méně elegantní a flexibilní aplikaci.

Struktura GRIDu na bázi Alchemi uvedená v [4] je reprezentována čtyřmi typy uzlů (nodes) :

- **Manager**

  Stará se o chod celého GRIDu. Na tomto uzlu je nainstalována aplikace RapidParallel. Manager může využívat služeb webové aplikace cross-platform manager k propojení uzlů s různým operačním systémem.

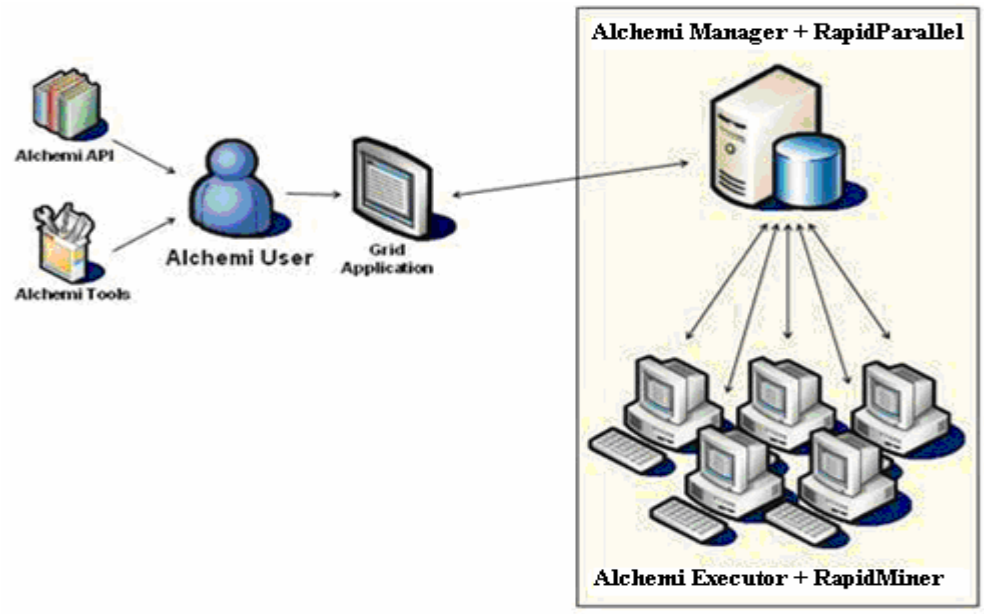
- **Executor**

  Na uzlu typu Executor dochází k spuštění vláken a jobů. Může pracovat v režimu dedicated a nondedicated. Režim dedicated se vyznačuje významným podílem Managera jako elementu zahajujícího výpočet vláken a jobů. V režimu nondedicated řídí výpočet Executor na žádost Managera. Na všech executorských uzlech je nainstalována aplikace RapidMiner.

- **User**

  User je vlastníkem GRIDové aplikace vytvářené za použití .NET Api a Alchemi SDK.

  Strukturu GRIDu podle Alchemi nejlépe vystihuje obrázek 2.3.




**Obrázek 2.3: výpočetní GRID na bázi frameworku Alchemi, převzato z [4]**

Mezi další velmi rozšířený a zajímavý prostředek pro tvorbu paralelních programů patří komunikační protokol MPI. Dnes se jedná o určitý neoficiální standart, který byl úspěšně aplikován v mnohých projektech. V MPI je na každém uzlu paralelní architektury (GRID, cluster) vytvořen proces s částí vstupních dat. Komunikaci mezi uzly umožňují různé druhy knihovních funkcí. Může se jednat o zasílání zpráv point-

to-point mezi dvěma účastníky, lze utvořit předem definovanou skupinu komunikujících uzlů nebo je možné vysílat zprávy pro všechny aktivní procesy. Toto pojetí reflektují i metodologie vytváření paralelních algoritmů. Jednou z nejslavnějších je Fosterův návrh uvedený například ve [3]. Foster postuloval čtyři kroky vývoje paralelního algoritmu:

- **Partitioning**

  V tomto kroku se snažíme nálezt datový nebo funkcionální paralelismus v řešeném úkolu. Datový paralelismus je vlastnost řešeného problému umožňující provádění stejných operací nezávislými výpočetními jednotkami s různými částmi vstupních dat. Funkcionální paralelismus umožňuje realizovat různé operace na stejné části dat. Hledání paralelismu se zpravidla děje intuitivním rozdělováním celé úlohy na podúlohy, které mohou být řešeny samostatně nebo jen s minimální závislostí na výsledcích a vstupních datech jiných podúloh.

- **communication**

  Jak název napovídá, hledáme a implementujeme nejvhodnější způsob komunikace mezi jednotlivými procesy za využití již zmíněných knihovních funkcí.

- **agglomeration**.

  Zde upravujeme doposud navržený paralelní algoritmus takovým způsobem, aby co nejlépe využíval architekturu na níž bude prováděn. To například znamená, že máme-li mnoho podúkolů vzhledem k počtu počítačů na GRIDu, sloučíme je do větší posloupnosti kroků prováděné na jednom uzlu. Na místě je také uvážení velikostí vyměňovaných bloků rozdělených dat mezi uzly v jedné zprávě vzhledem k charakteristikám sítě.

- **mapping**

  V této chvíli by jsme měli dle Fostera zaměřit své úsilí na co nejlepší využití jednotlivých procesorů a snížení meziprocesorové komunikace tím, že namapujeme vzájemně často komunikující procesy na stejný procesor.

Existuje několik doporučení jak kroky partitioning, communication, agglomeration i mapping úspěšně provádět. Fosterův návrh má charakter nízkoúrovňového přístupu se vším co z toho plyne. Vývoj aplikace a její přípravná fáze jsou delší, ale ve srovnání s jinými postupy vykazuje větší rychlost a velmi efektivní využití hardwaru. Výhodou MPI je velký počet uživatelů, diskusních fór, kvalitní oficiální literatury a to, že projekty se stále aktivně rozvíjejí. Některé z idejí Fosterova návrhu je doporučené studovat a možné využít i při použití jiného nástroje než je MPI a jemu podobné, protože srozumitelně vysvětluje podstatu paralelního programování a obtíží s ním spojených. MPI se často používá v součinnosti s OpenMP pro dosažení ještě většího zrychlení. OpenMP je aplikační programové rozhraní pro paralelní programování systémů se sdílenou pamětí jako je třeba vícejádrový procesor. Náznaky MPI se objevují i v posledních iniciativách vývojářů Alchemi.

## 2.2 NEURONOVÉ SÍTĚ

Definice z [2] říká :

*Neuronová síť je výpočetní model používaný v umělé inteligenci. Jejím vzorem je chování odpovídajících biologických struktur. Umělá neuronová síť je struktura určená pro distribuované paralelní zpracování dat. Skládá se z umělých (nebo také formálních) neuronů, jejichž předobrazem je biologický neuron. Neurony jsou vzájemně propojeny a navzájem si předávají signály a transformují je pomocí určitých přenosových funkcí.*

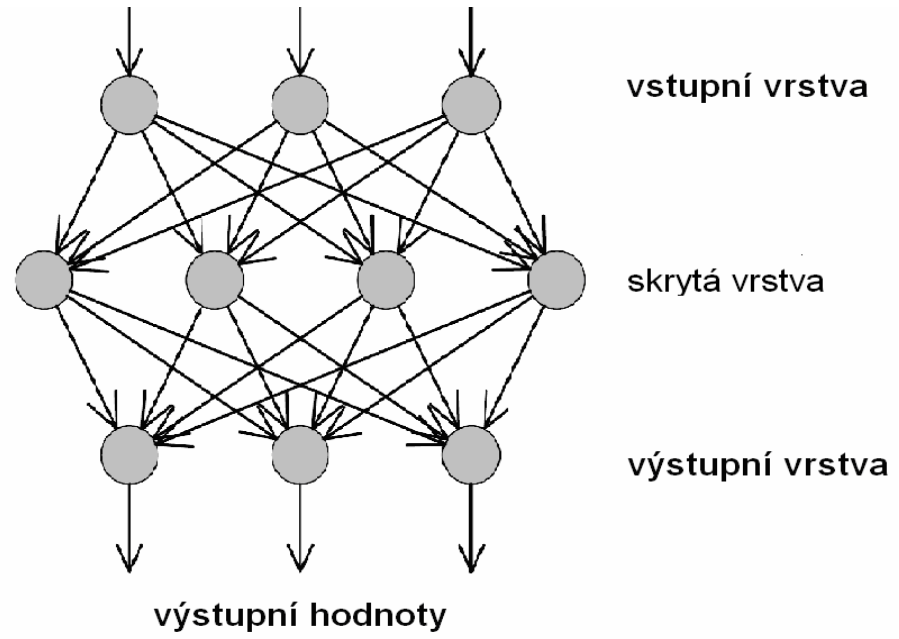
Neuronové sítě jsou nástrojem k řešení problémů v mnoha různých odvětvích lidské činnosti. Využívají se v počítačovém vidění, navigaci robotů, kompresi obrazu, zvuku nebo filtrování spamu a mnoha dalších. Jednou z možných topologií neuronových sítí je vícevrstvá perceptronová síť. Je to dopředná síť bez laterálních spojů mezi neurony. To značí pouze jednosměrné šíření signálu od vstupní vrstvy směrem k výstupní a nepřítomnost spojů mezi neurony téže vrstvy, jak ukazuje obrázek 2.3. K učení je použit iterativní gradientní algoritmus backpropagation minimalizující čtverce chybové funkce $E_p$. Chybová funkce $E_p$ je podle [6]

definována jako součet kvadrátů rozdílů žádaného a skutečného výstupu všech neuronů výstupní vrstvy. Chyba celé tréninkové množiny $E$ je součtem chybových funkcí při použití $p$ vzorů testovací množiny. Symbol $N$ udává počet výstupních neuronů.

$$E_p = \frac{1}{2}\sum_{o=1}^{N}(d_o - y_o)^2 \tag{1}$$

$$E = \frac{1}{2}\sum_{k=1}^{p}\sum_{o=1}^{N}(d_o - y_o)^2 \tag{2}$$

Informace o chybě se z výstupní vrstvy šíří přes vnitřní vrstvy až k vrstvě vstupní. Při tom se v základní verzi upravují jen vstupní váhy neuronů a chyba tréninkové množiny $E$ se zmenšuje.




**Obrázek 2.4: vícevrstvá perceptronová síť**

## 2.3 BRUTE FORCE SEARCH (BFS)

Jednoduchý a obecný způsob řešení úkolu, při kterém se systematicky prochází celý prostor možných řešení. Značnou výhodu je fakt, že pokud řešení existuje, tak bude zaručeně nalezeno. Nedostatkem je velká časová složitost. I pro relativně malé množství dat se tento postup stává nepraktickým.

## 2.4 OPTIMALIZOVÁNÍ NEURONOVÝCH SÍTÍ

Neuronová síť má paradigma tvořené způsobem učení, vybavování, topologií a modelem použitého neuronu. Jejich volba společně s dalšími parametry rozhoduje o výsledné přesnosti vytvořené sítě. Existuje zřejmá snaha zjistit jaká kombinace parametrů povede k nejlepší neuronové síti. Jednou z metod je vyzkoušet všechna možná nastavení a z výsledných sítí vybrat tu nejlepší. Jedná se o BFS algoritmus, který realizuje operátor GridParameterOptimization programu RapidMiner.

Pro zjednodušení uvažujme, že každý parametr má stejný počet hodnot X. Pro N parametrů dostáváme exponenciální asymptotickou časovou složitost $O(X^N)$ optimalizačního algoritmu. Vzroste-li například počet hodnot jediného parametru o deset, zvětší se celkový počet uvažovaných kombinací desetkrát. Pro velké množství parametrů nebo jejich hodnot je počet možných nastavení obrovský. Optimalizace metodou BFS naštěstí vykazuje značný datový paralelismus. Je umožněno dělení hodnot parametrů neuronových sítí na menší celky a jejich samostatné vyhodnocení. Tento problém je možné výhodně vyřešit za použití GRIDu.

# 3. IMPLEMENTACE NA GRIDU

Po stručném definování základních pojmů můžeme přistoupit k popisu začleňování programu RapidMiner do výpočetního GRIDu pomocí aplikace RapidParallel psané v programovacím jazyku C# pro .Net 3.0 a vyšší v Microsoft Visual Studiu 2008. Jako výpočetní Alchemi model byl zvolen Grid-Thread. Výběr byl proveden navzdory teoretickým informacím, které by v této situaci jednoznačně favorizovaly použití modelu Grid-Job. Jedním z důvodů rozhodnutí byla již zmíněná malá volnost při programování, nedostatečná dokumentace a minimální množství ukázkových příkladů u druhého z výše jmenovaných modelů.

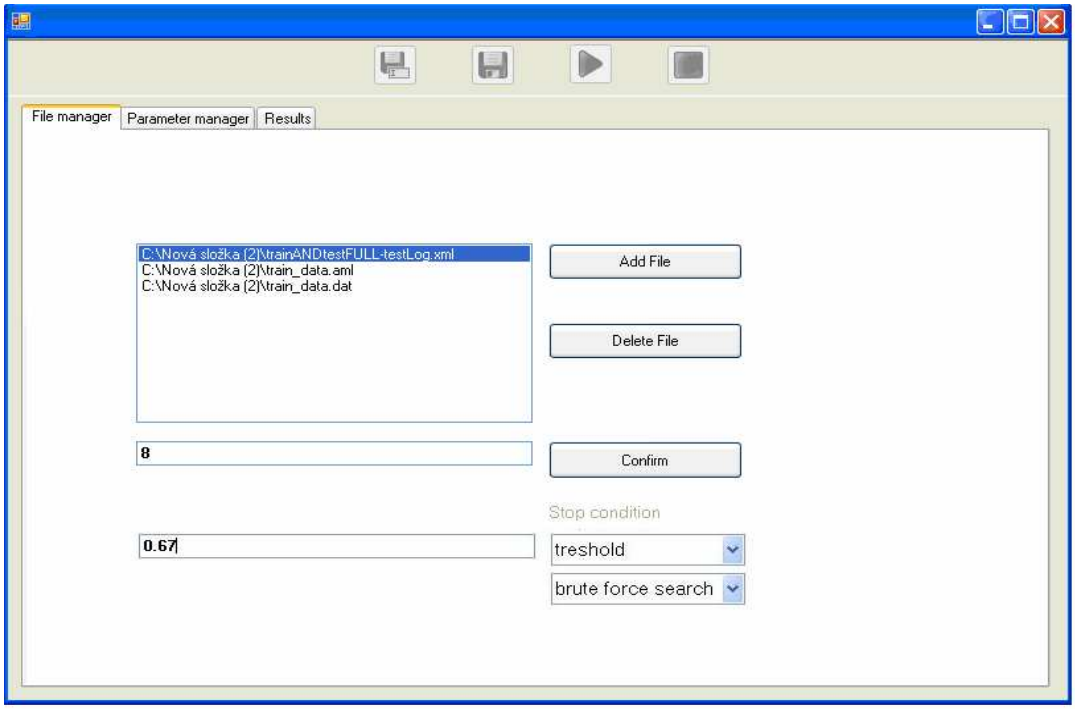
Nyní bude představeno uživatelské rozhraní aplikace RapidParallel a následně funkce, které se za ním skrývají.

## 3.1 UŽIVATELSKÉ ROZHRANÍ APLIKACE RAPIDPARALLEL

Skládá se ze tří částí. Mezi jednotlivými sekcemi je možné přecházet pomocí záložek. Uzamčení a odemčení editačních prvků je stanoveno konkrétním stavem programu a částí rozhraní do které se přechází.

### 3.1.1 File manager

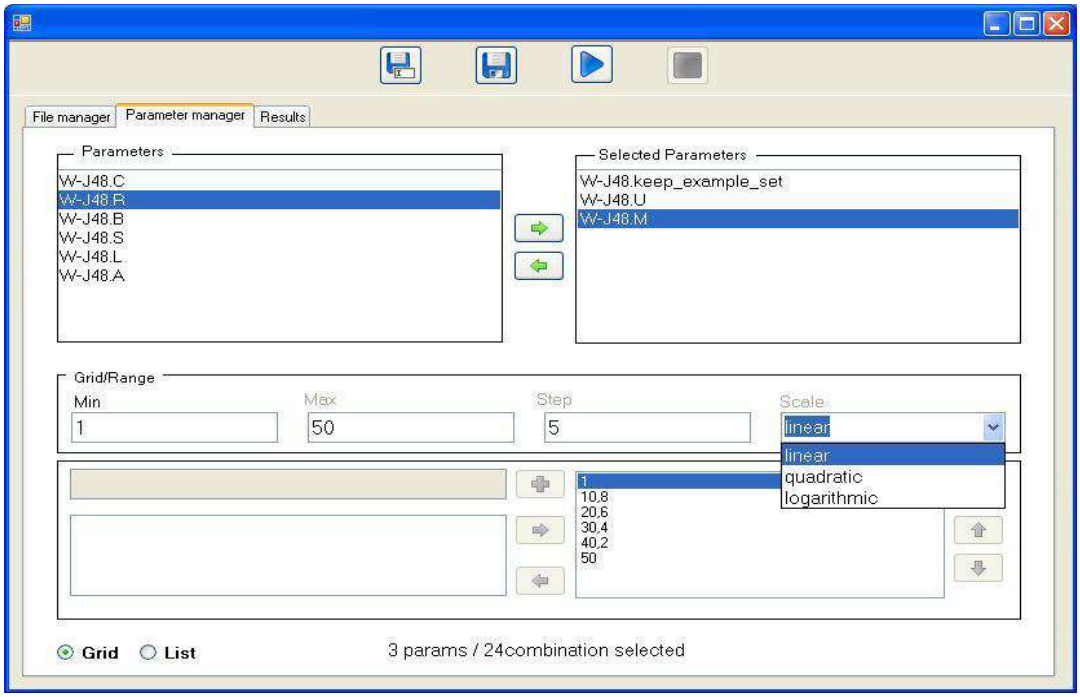
Tato část rozhraní zachycená na obrázku 3.1 je uživateli zobrazena bezprostředně po spuštění aplikace RapidParallel. Vstupní soubory je možné specifikovat v dialogovém okně vytvořeném po stisknutí tlačítka Add File. Jejich vyhledávání usnadňuje filtrování podle přípony xml, aml a dat. Opačnou funkci poskytuje tlačítko Delete File. V roletovém menu lze vybírat z několika algoritmů prohledávání prostoru parametrů. Výpočet je ukončen, když některý z výsledků překročí nastavenou prahovou hodnotu nebo se zpracují všechna vstupní data. Tlačítkem Confirm se potvrdí správnost zadaných údajů. Rozhraní se přepne do části Parameter manager. V této chvíli je možné používat editační prvky v sekci Parameter manager i File manager .



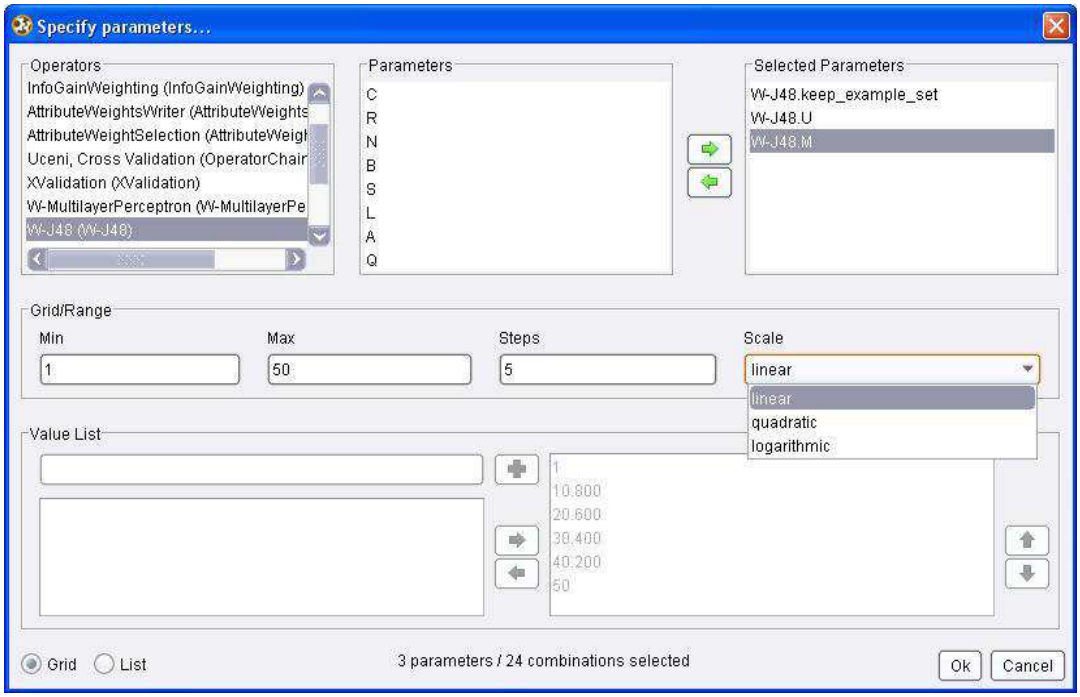

**Obrázek 3.1: uživatelské rozhraní aplikace Rapidparallel- File manager**

### 3.1.2 Parameter manager

Sekce Parameter manager vychází z rozhraní operátoru GridParameterOptimization programu RapidMiner. Pro uživatele s ním obeznámené se usnadní používání aplikace RapidParallel. Mezi možné operace v této sekci rozhraní patří přidání nebo odebrání parametrů načtených z xml souboru, změna jejich hodnot a formátu. Po stisknutí tlačítka play je nutné v dialogovém okně nastavit parametry komunikace Alchemi GRIDu. Jejich zadání a stisknutí tlačítka ok spustí výpočet a zároveň přepne rozhraní do části Results. V této chvíli dochází k uzamčení všech editačních prvků v částech File manager a Parameter manager. Jejich opětovné použití je možné jen po ukončení výpočtu, restartu aplikace nebo použitím tlačítka Stop.



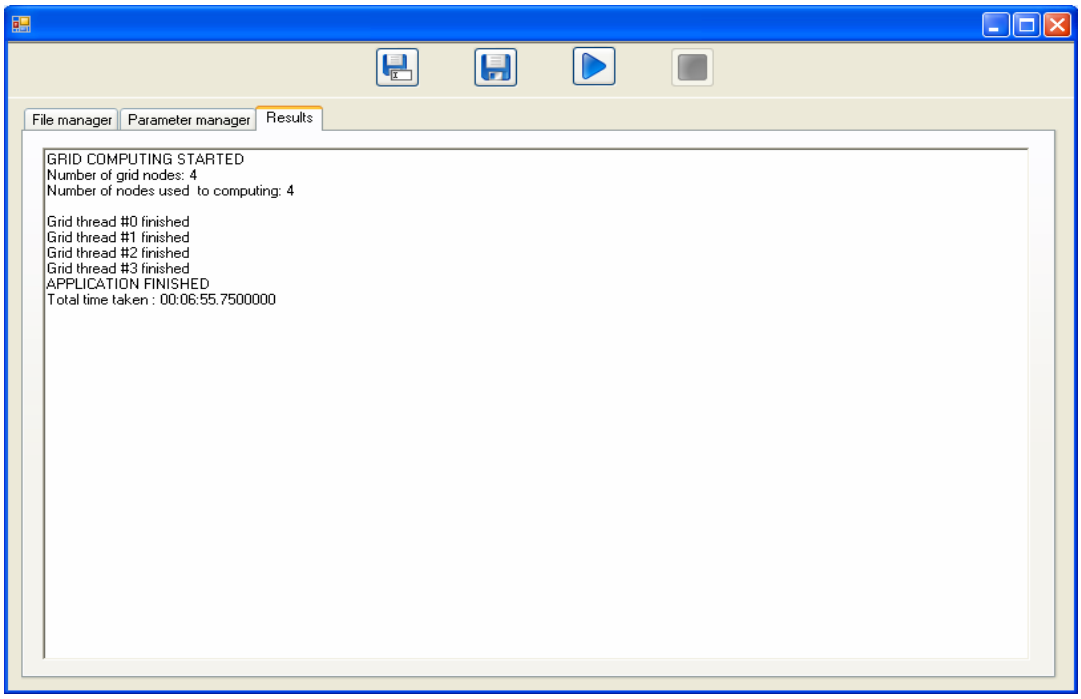

**Obrázek 3.2: uživatelské rozhraní aplikace Rapidparallel- Parameter manager**




**Obrázek 3.3: rozhraní operátoru GridParameterOptimization programu RapidMiner**

### 3.1.3 Results

Zde se zobrazují informace o průběhu výpočtu na GRIDu. Oznamuje se dokončení každého threadu, celé aplikace, využitý počet executorů, celkový čas výpočtu a výsledek optimalizace.




**Obrázek 3.4: uživatelské rozhraní aplikace Rapidparallel- Results**

## 3.2 PROGRAMOVÁ STRUKTURA APLIKACE RAPIDPARALLEL

V této chvíli máme jistou představu o tom, jak aplikace RapidParallel vypadá a jak z uživatelova pohledu funguje. Část programová sestává z několik knihoven, které obsahují metody pro vyhledávání RapidMineru na stanicích s Executory, jeho korektní spuštění a ukončování, zpracování xml souborů, formátování parametrů, operace se stromy a mnohé další. Uživatel se s jejich využitím může soustředit na samotný návrh paralelních algoritmů s různými operátory RapidMineru, jejichž ukázky jsou popsány v této kapitole.

Struktura kódu aplikace vytvářené pomocí Alchemi se dělí do dvou částí:

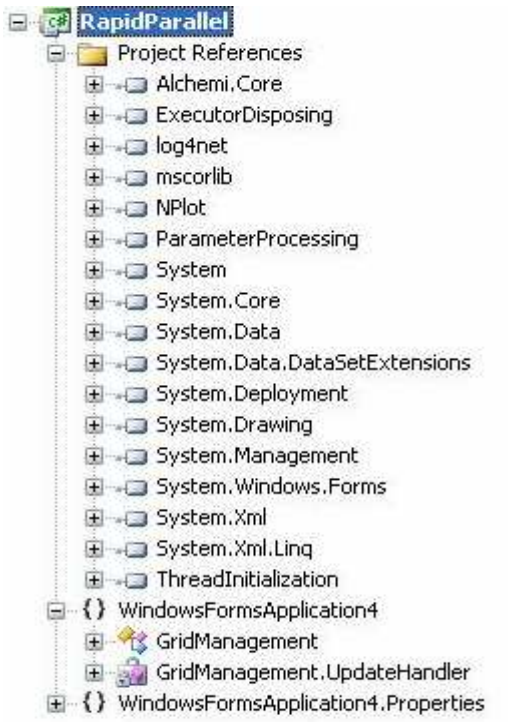
- **Local code**

  Zajistí vytvoření a spuštění GRIDové aplikace.

- **Grid code**

  Provádí se na vzdálených počítačích s aktivním Executorem.

Výčet knihoven využívaných aplikací RapidParallel je ukázán na obrázku 3.5. Význačné metody z ExecutorDisposing, ThreadInitialization (Local code) a ParameterProcessing (Grid code) , které vznikly v rámci této práce, budou popsány na příkladu optimalizování perceptronových sítí v pořadí, v jakém se volají při užívání aplikace RapidParallel. Představeny budou také důležité části knihovny Alchemi.Core.




**Obrázek 3.5 struktura aplikace RapidParallel**

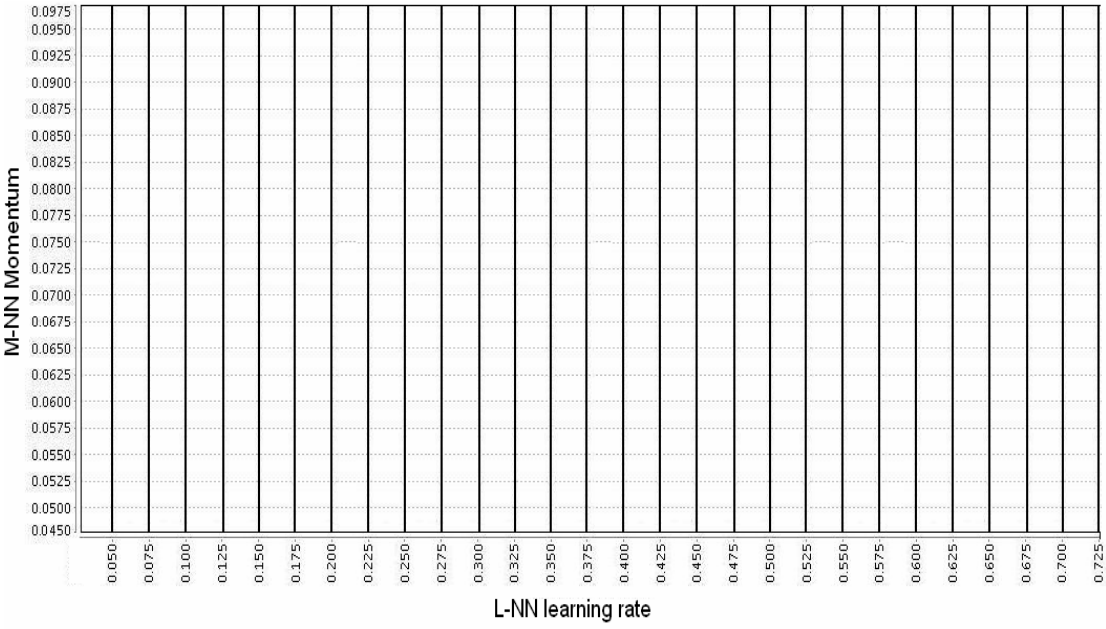
### 3.2.1 Local code

Po spuštění aplikace RapidParallel se vytvoří rozhraní, které implemetuje třída GridManagement. Uživatel je zde vyzván, aby určil nezbytná vstupní data. Po

stisknutí tlačítka Confirm se získají metodou Analyze a technologií LinQ z xml souboru parametry specifické pro využívaný operátor RapidMineru. Uživateli je umožněna jejich základní editace metodami třídy ParameterProcessing. Stiskem tlačítka play se spustí vytváření GRIDové aplikace. Zde se běh programu liší v závislosti na typu distribuce dat a přidělování výpočetních zdrojů na:

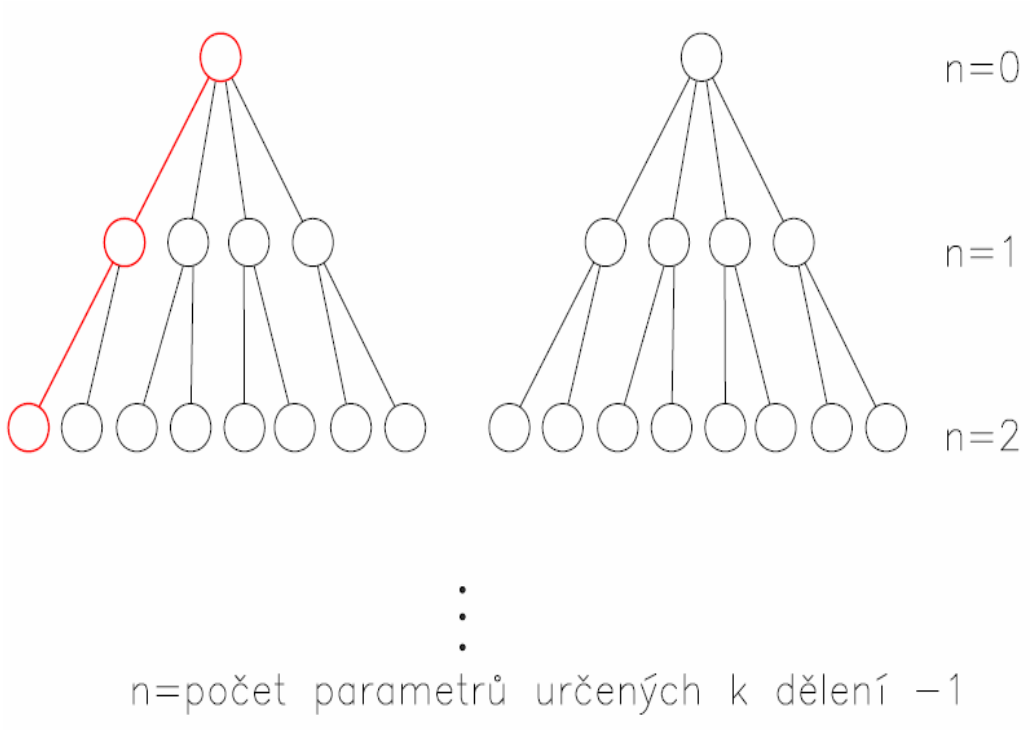
- **brute force search optimization (BFSO)**

Algoritmus paralelně prochází prostor parametrů metodou BFS. Z toho vyplývá způsob dělení dat u nějž musí být nejprve určeno, které parametry budou rozdělené na jednotlivé výpočetní stanice a které zůstanou v původní nedělené podobě. Dále je nutné stanovit vhodný počet počítačů využitých na GRIDu . Tyto a několik dalších údajů vyhodnocuje a vrací metoda GetDividingInfo. Metoda probíhá iterativně dokud není nalezeno optimální řešení. Postupně se zvyšuje počet dělených parametrů, až je dosaženo dostatečně velkého množství kombinací jejich hodnot. Kritériem pro ukončení dělení je výsledek podílu dvou údajů. Čitatel je roven součinu počtů hodnot dělených parametrů. Jmenovatel je počet použitých Executorů na GRIDu, který uživatel zadával po spuštění aplikace. Pokud je výsledek podílu větší nebo roven jedné, výběr dělených parametrů končí. Pokud nedosáhneme uspokojivého počtu kombinací ani dělením předposledního parametru, sníží se počet využitých stanic na GRIDu o jednu. Opakováním tohoto postupu se zajistí maximální, ale ne zbytečně velký počet využitých výpočetních stanic a vhodné množství rozdělených parametrů. Dělení prostoru vzniklé takovým postupem je na obrázku 3.6. Jedna hodnota zůstává vždy nedělena. Zamezí se tím situaci, kdy by RapidMineru byly odeslány parametry s jednou hodnotou. RapidMiner není schopen taková data vyhodnotit a došlo by k jejich ztrátě. Počet vytvořených bloků se může zdát poměrně vysoký, ale výhodou je fakt, že zůstává v průběhu celého výpočtu konstantní a nezvyšuje výpočetní a komunikační náročnost.




**Obrázek 3.6: dělení prostoru parametrů pro BFS algoritmus**

V případě příliš malého počtu hodnot a tím způsobené nemožnosti paralelizace, se výpočet na GRIDu nezahájí. Provede se pouze oznámení negativního stavu uživateli. Z dělených parametrů se při úspěchu předchozího algoritmu vytvoří strom metodou CreateTree. Tato datová struktura umožňuje efektivní a transparentní rozdělení hodnot parametrů na jednotlivé stanice za běhu programu. Na obrázku 3.7 je znázorněn příklad vytvářeného stromu.




**Obrázek 3.7: strom vytvořený z dělených parametrů. První dělený parametr (n=0) má dvě hodnoty, druhý (n=1) čtyři hodnoty a třetí parametr (n=2) má hodnoty dvě.**

Po specifikaci souborů odesílaných na executoské stanice pomocí třídy Alchemi EmbeddedFileDependency, nám již nic nebrání ve vytvoření a spuštění GRIDové aplikace typu on the fly. To znamená, že se nejprve vytvoří a spustí určitý počet threadů. V našem případě rovný počtu stanic využitých k optimalizaci. Další thready se mohou vytvořit až po ukončení předchozích. Každý thread obsahuje unikátní kombinaci hodnot dělených parametrů odpovídající jedné konkrétní větvi ze stromu dělených parametrů (v obrázku 3.7 zvýrazněno červenou barvou). Hodnoty nedělených parametrů jsou shodné pro všechny stanice.

Optimalizování neuronových sítí pro různé hodnoty parametrů může trvat rozdílnou dobu. Zvoleným druhem GRIDové aplikace označované také jako multiuse společně se způsobem přidělování vstupních dat threadům „větev po větvi“, se do jisté míry tento nepříznivý jev potlačí a je tím zajištěn nezbytný load balancing. Zmíněný

algoritmus tak zaručuje relativně rovnoměrné zatížení jednotlivých výpočetních stanic.

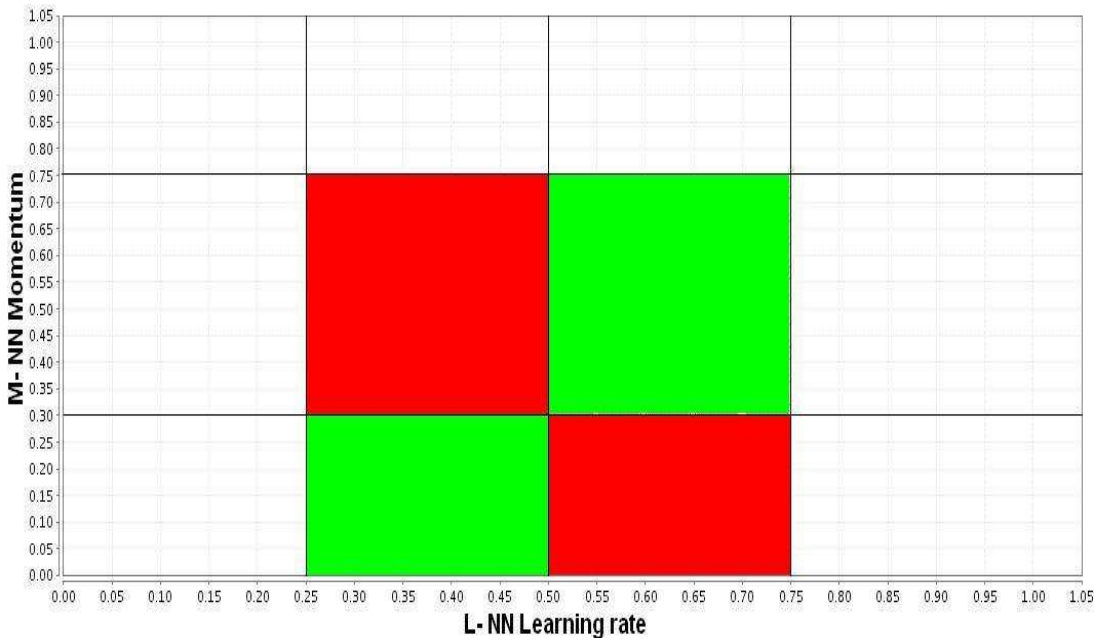
Po ukončení výpočtu všech threadů už zbývá jen vybrat nejlepší neuronovou síť ze všech, které byly na managerskou stanici přijaty ze stanic executorských. Výsledné soubory jsou uloženy do složky obsahující uživatelem specifikovaný xml soubor.

- **dynamic data scheduling (DDS)**

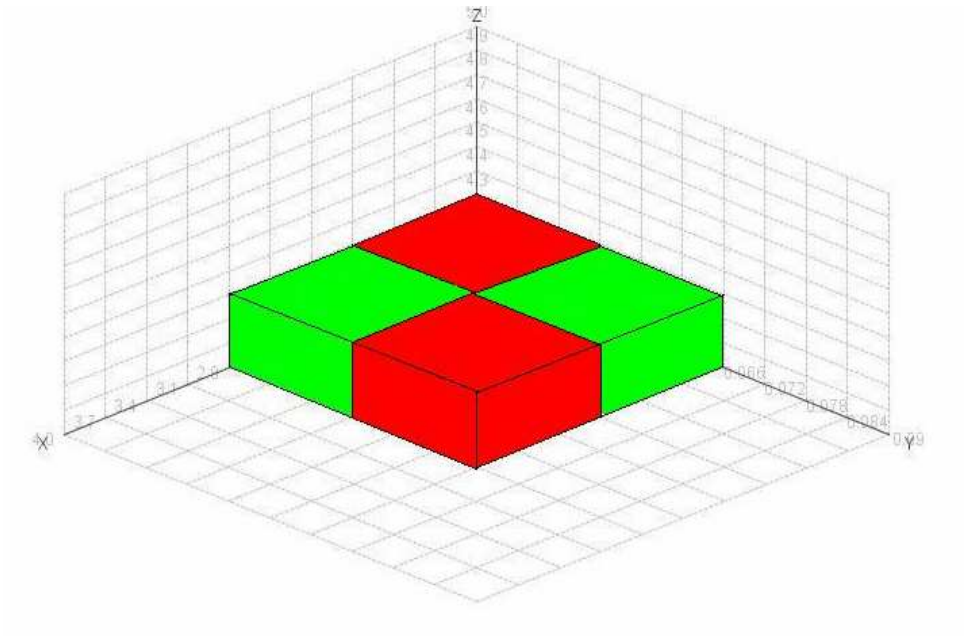
Předchozí algoritmus prokázal ve srovnání se svou sekvenční podobou znatelné zrychlení, jak bude ukázáno v následující kapitole. Přesto se nejedná o nejpropracovanější způsob a existuje možnost mnoha úprav zlepšujících rychlost optimalizace. První odlišností a zároveň vylepšení vlastností algoritmu DDS vůči BFSO spočívá v jiném dělení prostoru parametrů. Množiny prvků stejných vlastností často utvářejí shluky. Pro další zpracování by bylo vhodné prostor parametrů dělit na oblasti obsahující co nejvíce prvků z jednoho takového útvaru. Můžeme tím získat homogennější bloky dat z hlediska nějakého sledovaného parametru např. přesnosti neuronové sítě. Druhým rozdílem a výrazným využitím předchozí nové vlastnosti je průběžné sledování hodnot přesností sítí patřících jednotlivým blokům. Na základě tohoto pozorování dostává každý z bloků přidělen různý výpočetní čas. Oblasti vykazující lepší vlastnosti dostanou více výpočetního času a naopak.

Základní myšlenou při implementaci algoritmu bylo rozdělit prostor parametrů na počet dílů rovný množství počítačů na GRIDu, otestovat několik nastavení z každé části a po určitém počtu iterací sloučit bloky s nízkou dosaženou přesností do jednoho celku a bloky s vysokou přesností rozdělit. Dělení a slučování mělo probíhat tak, aby celkový počet bloků zůstal stále roven původnímu počtu dílů. Přidělování výpočetního času by bylo závislé na procentuálním podílu přesnosti dosažené v bloku dat ze sumy přesností všech bloků. Takto navržený algoritmus však nelze realizovat. Bloky dat jsou kombinacemi parametrů. Při sloučení dvou bloků mohou vzniknout kombinace hodnot parametrů nacházející se v jiných blocích. Ty by musely být také přidány do nově vznikajícího bloku a to vždy není možné ani

výhodné. Tento restriktivní fakt je přehledně dokumentován v 2D a 3D grafech na obrázcích 3.8 a 3.9. RapidMiner bohužel neposkytuje sofistikovanější možnost ohraničování vstupních hodnot, kterým by bylo možno určitý prostor parametrů přesně vymezit. Formování bloků dat je omezené jistými pravidly, které zabraňují existenci algoritmu DDS v této formě.




**Obrázek 3.8: slučování bloků dat v 2D. Zelené bloky dat značí hodnoty, které chceme spojit. Červené bloky jsou naopak ty, jenž jsou nechtěně přidány také.**




**Obrázek 3.9: slučování bloků dat v 3D. Zelené bloky dat značí hodnoty, které chceme spojit. Červené bloky jsou naopak ty, jenž jsou nechtěně přidány také. Pro jednoduchost jsou vykresleny pouze slučované bloky.**

Řešení nepříjemného problému spočívá ve sloučení bloků nikoliv na úrovni dat, pomocí spojování větví stromu, kterým jsou data interně reprezentována, ale spíše ve vytvoření určitého logického seskupení. Vytvoříme několik pomyslných skupin bloků dat a budeme je spojovat přidáváním do některé ze skupin na základě dosažených výsledků. Nejlepší blok dat rozdělíme na několik částí a otestujeme. Velikosti skupin a jim přidělený výpočetní čas volíme v obrácených poměrech. Tímto postupem se vyhneme našemu problému se slučováním bloků při zachování informace o měnícím se prostoru parametrů a zároveň je zaručeno podrobnější prozkoumávání nejlepších oblastí dat, jak bylo původně zamýšleno.

Dělení prostoru parametrů na počet bloků stanovený za běhu aplikace je prvním krokem ke konkrétní úspěšné realizaci algoritmu. Z povahy věci není možné rozdělit n-rozměrný prostor na libovolné množství stejných částí. Dobrou aproximaci zaručují metody findDividingConstants, createTreeForDynamicSearch a treeModification. První z jmenovaných metod se pokusí nalézt vhodné dělící

konstanty pro každý parametr. Výpočet probíhá s uvážením velikostí dělených parametrů a tak, aby všechny bloky obsahovaly srovnatelné množství hodnot. Parametr s mnoha hodnotami bude rozdělen na početnější skupiny. U méně obsáhlých parametrů je tomu naopak. Pro další zpřesnění jsou dělící konstanty upravovány zvětšováním nebo zmenšováním v závislosti na odchylce aktuálně získané velikosti bloků od požadované hodnoty. Pokud není dosaženo zlepšení v několika po sobě jdoucích krocích nebo se zvětšuje a zmenšuje stále stejná konstanta, zpřesňování končí. S využitím informací z předešlého kroku vytvoříme strom v metodě createTreeForDynamicSearch, který je možno procházet od kořene k listům i směrem opačným. Dosavadní postup není dostačující z důvodu neodstranitelné nepřesnosti při výpočtu dělících konstant. Strom je dodatečně modifikován v metodě treeModification. Postupně dochází ke slučování a dělení větví, až je dosaženo požadované topologie stromu. Nejdříve se zkouší sloučení nesousedních větví. To jsou ty, které mají rozdílné hodnoty alespoň dvou parametrů. Pokud nenajdeme vhodnou dvojici nesousedních větví, spojíme větve sousední, kde je pravděpodobnost úspěšného sloučení největší. Částečně se tím zabrání rozdělení prostoru známé z algoritmu BFSO. Metoda treeValidation ověřuje, že nově vzniklý strom obsahuje všechny kombinace a nedošlo ke ztrátě dat. Test začíná vytvořením nového elementárního stromu s uzly obsahujícími vždy pouze jednu hodnotu. Každá větev představuje jednu konkrétní kombinaci parametrů. Kopii testovaného stromu upravíme do stejného tvaru jako má strom elementární. Vzájemným porovnáním všech větví obou stromů snadno zjistíme duplicitu nebo nepřítomnost nějaké kombinace parametrů .

Uvažujme nyní jednoduchý dvourozměrný případ, kdy první parameter má 26 hodnot a druhý 21 hodnot. Zvolme požadované rozdělení do 10 bloků. X značí množství kombinací, které by mělo být přiděleno pro každý blok. Výpočet vedoucí ke správnému dělení parametrů je následující:

1. Ve výchozí rovnici definujeme vztah mezi dělícími konstantami *a* a *b* jejichž hodnoty hledáme a požadovanou velikostí vytvořeného bloku dat:

$$X = a \bullet b \tag{3}$$

2. K řešení rovnice 3 sestavíme dvě rovnice. První z nich s číslem 4 vyjadřuje vzájemný poměr dělících konstant *a* a *b*, který je stejný jako poměr počtů jejich hodnot. Z druhé rovnice s číslem 5 spočítáme požadovanou velikost bloků.

$$\frac{počet\_hodnot\_druhého\_parametru}{počet\_hodnot\_prvního\_parametru} \bullet a = \frac{21}{26} \bullet a = b \tag{4}$$

$$X = \frac{počet\_kombinací}{počet\_bloků} = \frac{26 \bullet 21}{10} = 54,6 \tag{5}$$

3. Do rovnice 3 dosadíme z 4 a 5 a získámé 6. Po několika úpravách dostáváme výsledky 7 a 8:

$$a \bullet b = a \bullet \frac{21}{26} \bullet a = \frac{21}{26} a^2 = X \tag{6}$$

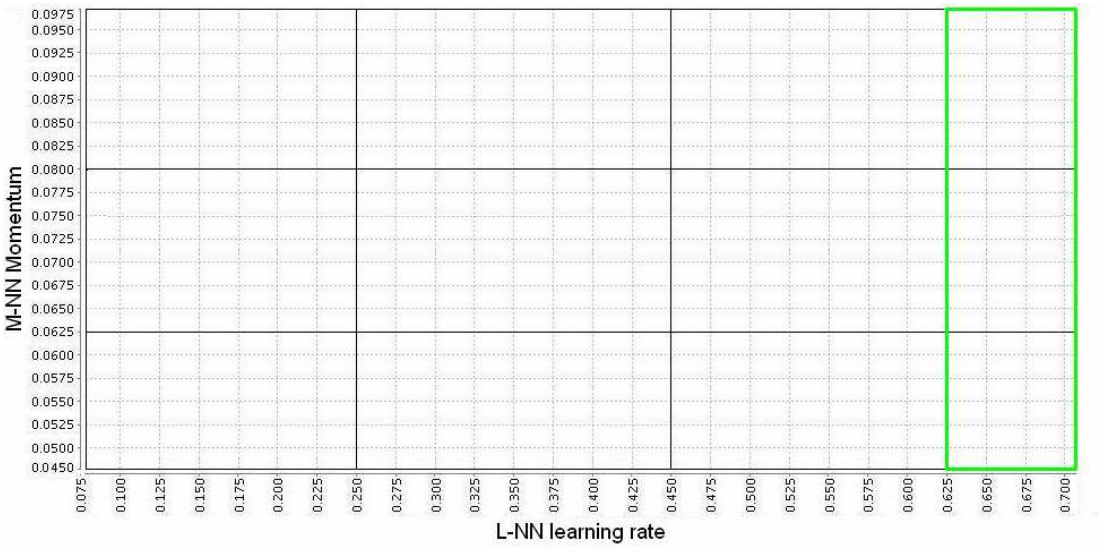
$$a = \sqrt{\frac{26}{21} \bullet X} = a = \sqrt{\frac{26}{21} \bullet 54,6} \doteq \underline{\underline{8}} \tag{7}$$

$$b = \frac{21}{26} \bullet a = \frac{21}{26} \bullet 8 \doteq \underline{\underline{6}} \tag{8}$$

Hodnota *b* se ve funkci FindDividingConstatnts pro zvýšení přesnosti dělení inkrementuje o jedničku a tedy *b* =7. Dělením prostoru parametrů pomocí konstant *a* a *b*, získáme 12 bloků s průměrnou velikostí 45,5 kombinací. Po úpravě funkcí treeModification dosáhneme naprosté shody s požadavkem 10 bloků s průměrem 54,6 kombinací na blok. V takto jednoduchém příkladě se nejedná o příliš výrazný rozdíl. Počet bloků se liší jen o 20% a velikost o 16,7%. Pro více parametrů jsou nepřesnosti nesrovnatelně větší. Kdybychom v ukázkovém příkladu přidali 4 parametry s počtem hodnot 26, 24, 22 a 25, vzniklo by po dělení konstantami 64 bloků. Rozdíl v množství činí 630% a průměrná velikost se od žádané liší o 84%. Metoda treeModification i v takovém případě zajistí správně rozdělený prostor. Pro

více parametrů je výpočet analogický. Vždy hledáme vztah n-tého parametru k prvnímu jako v rovnici 4 a získanou hodnotou násobíme rovnici 6. Prostý výpočet dělících konstant odmocninou by byl v některých případech možný, ale nezohledňoval by velikosti dělených parametrů.

Algoritmus dělení prostoru parametrů v DDS se snaží o dosažení maximální shody průměrné velikosti bloků s požadovanou hodnotou. Je proto možné, že počet bloků nebude vždy úplně přesný, jelikož není primárně optimalizován. Rozpor se zamýšleným typem dělením se vyskytuje zpravidla pouze v okrajových částech prostoru parametrů, jak ukazuje obrázek 3.10 vytvořený s použitím údajů získaných v předchozím příkladu.




**Obrázek 3.10: dělení prostoru parametrů pro DDS. Bloky s černým orámováním vznikly použitím dělících konstant. Zelený obdélník ohraničuje bloky sloučené v metodě treeModification.**

V této fázi je vše potřebné připraveno a můžeme začít s vlastním výpočtem. Několik počátečních cyklů programu slouží k získání hrubé představy o potenciálu jednotlivých bloků. Otestujeme vždy několik náhodně nebo přesně zvolených kombinací parametrů všech bloků a uložíme nejlepší dosažené výsledky. Bloky následně roztřídíme do několika skupin podle informací získaných v předchozím

kroku. V našem případě byl zvolen poměr počtu prvků skupin 10:30:60 a přidělovaný výpočetní čas odpovídající počtu využívaných Executorů danou skupinou na GRIDu v poměru obráceném 60:30:10. Malému procentu nejlepších oblastí je přiřazen značný výpočetní čas. Takto tvrdé podmínky jsou zapříčiněny tím, že každé zapnutí RapidMineru v batch modu a načtení vstupních dat, může trvat až jednotky sekund. Vzhledem k tomu, že do jisté chvíle počet bloků oproti původně zamýšlenému návrhu narůstá a RapidMiner se zapíná častěji, je přísné přidělování výpočetního času nezbytností. Počet skupin i přidělovaný čas se vhodně přepočítá, když počet bloků dat klesne pod určitou hranici. V dalších iteracích probíhá obdobný postup. U bloku, který aktuálně dokončil dílčí výpočet je znovu zjištěno, zda-li nedosáhl lepších výsledků než blok ve skupině s větším výpočetním časem. Jestli ano, tak se tyto dva bloky ve skupinách vymění. Blok s doposud nejlepšími sledovanými parametry je vždy rozdělen na dvě části, které jsou rovněž otestovány a zařazeny do skupin.

### 3.2.2 Grid code

Do skupiny Grid code patří třídy InitThread a ExecutorDisposing odvozené z třídy Gthread.

V třídě InitThread je potřeba pro spuštění výpočtu jednotlivých vláken (Grid-threads) překrýt metodu Start rodičovské třídy Gthread z knihovny Alchemi.Core. Kód vepsaný do této metody bude jako první prováděn na executorských stanicích. Metoda Start třídy InitThread při svém běhu nejprve nalezne složku, ve které je nainstalován Alchemi Executor a vytvoří zde novou složku Executor Files pro ukládání souborů přijatých z managerské stanice, výstupních souborů RapidMineru a několika dalších, esenciálních pro správnou funkci aplikace RapidParallel. K nalezení cesty cílové složky nám poslouží informace získané od procesu Alchemi.ExecutorExec, který přísluší aktivnímu Executorovi na daném uzlu GRIDu.

Jako další se volá metoda GetFilePath vyhledávající správný dávkový soubor rapidminer.bat, kterým se spouští program RapidMiner na executorských stanicích.

Určení jeho lokace začíná nalezením všech logickýh disků počítače a prohledáním celé adresářové struktury každého z nich. Cesta k nalezenému souboru se uloží do textového souboru ve složce, kde je nainstalován Alchemi Executor. Tato informace zajistí, že se nemusí časově náročná metoda opakovat znovu při dalším spuštění aplikace RapidParallel. Pokud je soubor poškozen nebo uložená cesta již není platná, probíhá vyhledávání znovu. Aplikace RapidParallel je schopná z více nalezených souborů rapidminer.bat odfiltrovat ty, které jsou zkopírovány mimo složky kde jsou ostatní programové části RapidMineru a tudíž nefunkční.

Následuje spuštění programu RapidMiner v batch módu a uložení id jeho rodičovského procesu do souboru. Procesu java.exe odpovídajícího programu RapidMiner je přiřazena nízká priorita, aby ho bylo možné kdykoliv rychle ukončit a zároveň mohl běžet na pozadí bez rušení uživatele, který může být u počítače přítomen v souladu s myšlenkou v úvodu o využití volné výpočetní kapacity ve firmách nebo školách. Po ukončení optimalizace se nalezne lokálně nejlepší neuronová síť a společně s dalšími soubory se odešle na managerskou stanici.

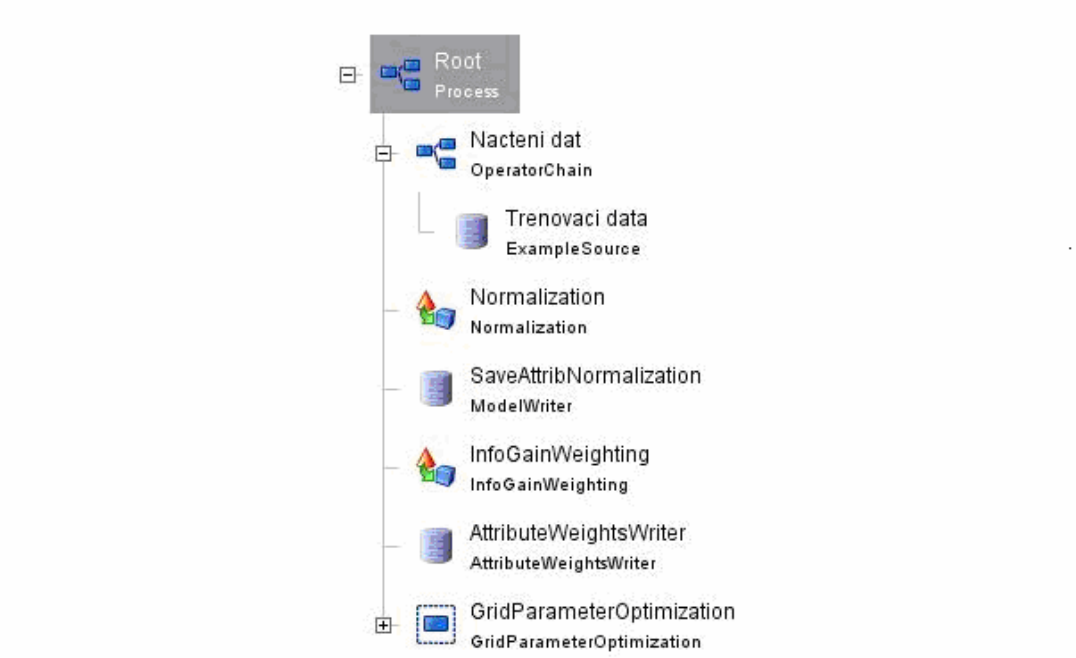
V případě, že se uživatel rozhodne aplikaci RapidParallel ukončit ještě před dokončením optimalizace na executorských stanicích, musí na nich být ukončen program RapidMiner. Metoda pro tento úkon je obsažena ve třídě ExecutorDisposing odvozené z třídy Gthread. Jěště před voláním jejích instancí zastavíme všechny prováděné thready na Executorech metodou Stop a metodou Clear z knihovny Alchemi.Core smažeme všechny vlákna, které obsahuje objekt třídy GApplication představující GRIDovou aplikaci. Pak vytvoříme počet objektů třídy ExecutorDisposing rovný množství aktivních Executorů. Ukončování se zahájí voláním jejich metody Start z Managera, která na všech stanicích vyhledá vhodný proces s názvem java a ukončí ho. Shodné označení procesu mají všechny spuštěné aplikace napsané v programovacím jazyku java. Abychom ukončili jen ten správný, bylo v metodě Start třídy InitThread uloženo id procesu spouštějícího RapidMiner do textového souboru. Při ukončování stačí nalézt všechny procesy se jménem java a ukončit ten, jehož rodičovský proces má id shodné s číslem uloženým v souboru.

Priorita procesu Alchemi.ExecutorExec je nastavena na high, aby ukončování proběhlo bez větší časové prodlevy. Alternativou by mohlo být umístění ukončovacího kódu do bloku finally v metodě Start třídy InitThread a ukončení RapidMineru v případě, že stav threadu je aborted nebo stopped.

Pokud chceme provádět operace, které by se daly nazvat „úklidem“ na Executorech, můžeme úspěšně používat uvedené funkce z knihovny Alchemi.Core. Nemůžeme je použít v případě, že právě dokončený thread splňuje ukončovací podmínku a my chceme provést posloupnost metod Stop nebo Dispose a Clear z Alchemi.Core. Reakcí na volaní prvních dvou metod v takzvaných event handlerech, jakým je i metoda zpracovávající ukončený thread, je zablokování GRIDové aplikace. Funkčním postupem je nepracovat s metodami přímo pro GRIDovou aplikaci, ale spíše s knihovnami Alchemi Managera jako je IManager. Zde se nacházejí rozličné metody, kterými ji můžeme vymazat nebo zastavit. Tento krok ve spojení s vytvořením nové GRIDové aplikace starající se o úklid po předešlé instanci třídy GApplication zaručí splnění požadovaného cíle.

# 4. VÝSLEKDY EXPERIMENTU

K demonstraci vlastností obou algoritmů prohledávání prostoru parametrů byla zvolena optimalizace vícevrstvé perceptronové sítě. Na obrázcích 4.1 a 4.2 je znázorněna grafická podoba programu zpracovávaná RapidMinerem. Obrázek 4.1 popisuje předzpracování trénovacích dat.




**Obrázek 4.1: grafická podoba programu v RapidMineru- předzpracování dat**

Význam jednotlivých uzlů stromu na obrázku 4.1 je podle [5] následující:

- **Root**

  Musí být v kódu vždy přítomen a to pouze jednou. Pomocí něj je možné definovat a používat globální proměnné.

- **Nacteni dat**

  Načte trénovací data ze souboru s příponou .aml .

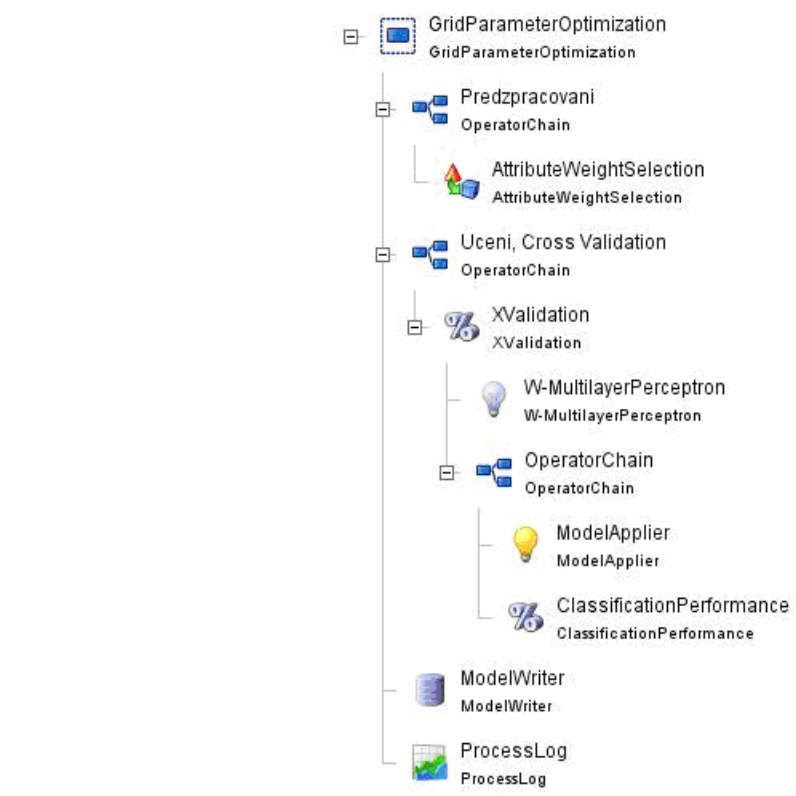
- **Normalization**

  Upravuje rozsah hodnot trénovacích dat. Je možné použít různé transformace nebo prosté upravení hodnot do intervalu s definovanou minimální a

maximální hodnotou. Nové hodnoty jsou v uzlu SaveAttribNormalization uloženy do souboru s příponou .mod.

- **InfoGainWeighting**
  Provádí ohodnocení významnosti parametru na základě výpočtu informačního zisku pro separaci tříd za předpokladu, že by tento parametr byl použit k dělení. Uzel AttributeWeightsWriter zapíše váhy všech atributů do souboru s příponou .wgt

Na obrázku 4.2 je graficky zobrazen průběh optimalizace parametrů vícevrstvé perceptronové sítě v RapidMineru.




**Obrázek 4.2 podoba programu v RapidMineru- optimalizace**

Význam a funkce uzlů na obrázku 4.2 jsou podle [5] následující:

- **GridParameterOptimization**

  Jedná se o algoritmus typu BFS. Operátor vrací optimální kombinaci hodnot všech parametrů, které byly specifikovány jako jeho vstupní data. K tomuto účelu slouží jeho vnitřní řetězec dalších operátorů.
- **AttributeWeigthSelection**

  Vybírá ze všech atributů vstupních dat získaných z uzlu InfoGainWeighting takové, které splňují určitou podmínku. Může se například jednat o filtraci atributů na základě minimální hodnoty váhy.
- **Xvalidation**

  Realizuje metodu odhadu přesnosti modelu cross validation, při níž je trénovací množina rozdělena na několik částí. Určitý počet částí slouží k vytvoření modelu a zbylý k jeho testování. RapidMiner nabízí velké množství způsobů dělení trénovací množiny.
- **MultilayerPerceptron**

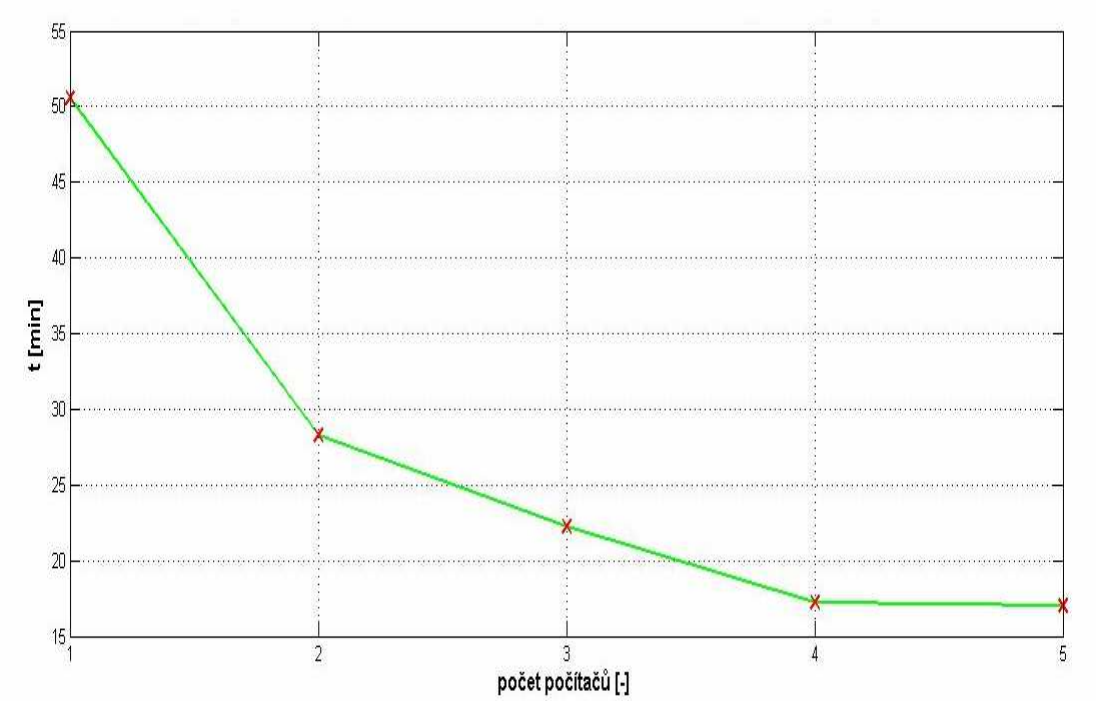
  Jedná se o uzel reprezentující vícevrstvou perceptronovou síť. Vstupem je část trénovací množiny určená k vytvoření modelu.
- **ClassificationPerformance**

  Tento operátor vyhodnotí výslednou přesnost.
- **ModelWriter**

  Ukládá vytvořený model do souboru s názvem FinalModel_NN.mod.
- **ProcessLog**

  Ukládá označené proměnné programu do souboru. V našem případě jsou to testované kombinace parametrů perceptronových sítí a jejich přesností.

První experiment provedený na GRIDu spočíval v optimalizování perceptronových sítí metodou BFSO bez ukončovací podmínky. Byl proveden postupně třikrát na jednom až pěti počítačích shodného typu, přičemž počítač s Alchemi Managerem zastupoval i funkci Alchemi User. Jako vstupní parametry byly použity hodnoty uvedené v tabulce 4.2 a dosažená maximální přesnost byla 0.68. Průměrné časy jsou uvedeny v tabulce 4.1 a vyneseny v grafu na obrázku 4.3.




**Obrázek 4.3 průběh závislosti času výpočtu na počtu využitých PC pro BFSO bez ukončovací podmínky**

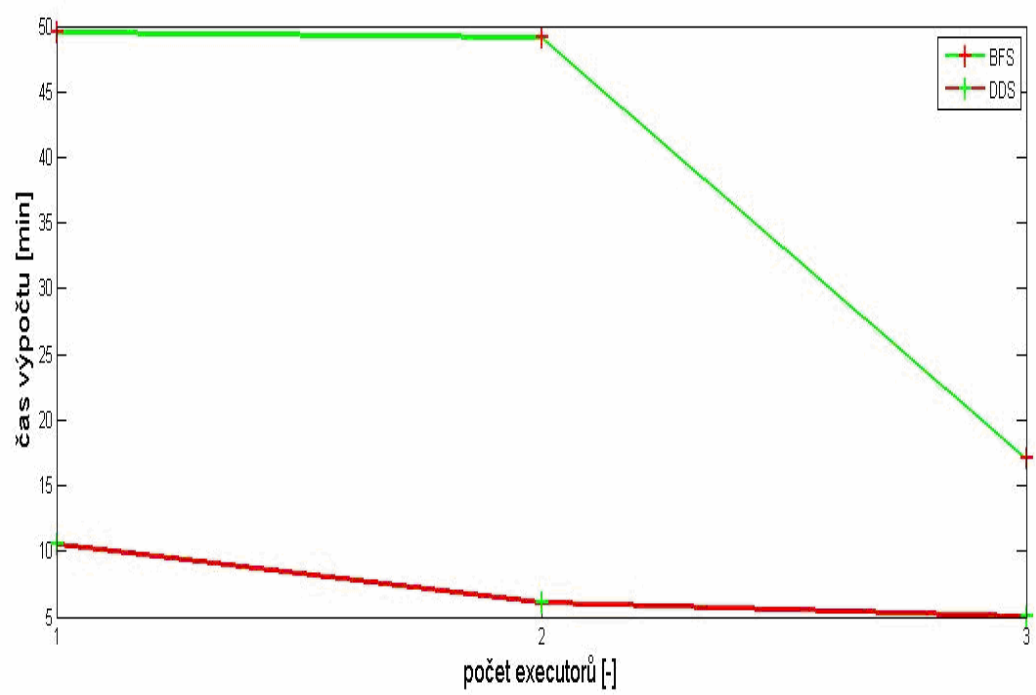
| Počet exekutorů | T[min:s] |
|---|---|
| 1 | 50:54 |
| 2 | 28:25 |
| 3 | 22:23 |
| 4 | 17:23 |
| 5 | 17:02 |

**Tabulka 4.1: čas potřebný k výpočtu při daném množství použitých Executorů pro BFSO bez ukončovací podmínky**

| Parametry | Hodnoty |
|---|---|
| AtributeWeight selection | 1;2;3;4;5;6;7 |
| M- NN momentum | 0.06; 0.32; 0.7 |
| L- NN learning rate | 0.05;0.275;0.5 |

**Tabulka 4.2: optimalizované parametry a jejich hodnoty pro první experiment**

Druhá testovací úloha měla za cíl vzájemné porovnání algoritmů BFS a DDS. Maximální počet Executorů byl z technických důvodů omezen na 3. Vstupní hodnoty v podobě intervalů jsou opět uvedeny v tabulce 4.3 a výsledné průběhy v obrázku 4.4. Průměrné časy jsou uvedeny v tabulkách 4.4 a 4.5. Jako ukončovací podmínka bylo zvoleno dosažení přesnosti  větší nebo rovné 0.63.




**Obrázek 4.4  průběh závislosti času výpočtu na počtu využitých PC pro srovnání BFSO a DDS**

| **Parametry** | **Intervaly hodnot** [min,max, počet hodnot] |
|---|---|
| AtributeWeight selection | [5;10;2] |
| M- NN momentum | [0.06; 0.7;7] |
| L- NN learning rate | [0.05;0.5;7] |

**Tabulka 4.3:  optimalizované parametry a jejich hodnoty pro druhý experiment**

| **Počet exekutorů** | **Čas** [min:s] |
|---|---|
| 1 | 49:32 |
| 2 | 49:06 |
| 3 | 17:03 |

**Tabulka 4.4: čas výpočtu pro daný počet Executorů a algoritmus BFSO s prahovou hodnotou 0.63**

| **Počet exekutorů** | **Čas** [min:s] |
|---|---|
| 1 | 10:36 |
| 2 | 6:04 |
| 3 | 5:02 |

**Tabulka 4.5: čas výpočtu pro daný počet Executorů a algoritmus DDS s prahovou hodnotou 0.63**

# ZÁVĚR

V této práci byly diskutovány přínosy, návrh a softwarové prostředky paralelního programování. Za využití frameworku Alchemi byla vytvořena paralelní aplikace RapidParallel. Jejím hlavním cílem je usnadit vývoj paralelních algoritmů využívajících RapidMiner jako výpočetní jádro. Byly vytvořeny dva příklady takových algoritmů pro prohledávání prostoru parametrů. První z nich, byl velice jednoduchý, ale přesto s velmi dobrými výsledky. Druhý, propracovanější postup dosáhl ještě několikanásobně kratšího času výpočtu.

Optimalizování parametrů perceptronových sítí metodou BFS je časově náročný úkol, jak bylo ukázáno výsledky experimentu. Použitím GRIDu je možné výpočet podstatně zrychlit nebo ve stejném čase prověřit více nastavení modelu. Z obrázku 4.3 je patrné výrazné zkrácení času optimalizace. Pro 5 počítačů je dosaženo více než trojnásobného zrychlení. Omezujícím faktorem je rozdílná délka trvání optimalizace v závislosti na konkrétní kombinaci hodnot parametrů. Tento nepříznivý jev byl částečně potlačen metodou rovnoměrného rozložení zátěže na jednotlivé výpočetní stanice, spočívající ve vhodném dynamickém přidělování vstupních dat. Výpočetní čas se poměrně rychle ustálil na hranici zhruba odpovídající času výpočtu nejnáročnější optimalizované kombinace parametrů. Konvergence času výpočtu ke konstantní hodnotě je pravděpodobně nakonec nevyhnutelná pro libovolné konečné množství hodnot parametrů a počítačů. Pro větší množství hodnot parametrů však bude tento jev nastávat při podstatně větším množství použitých executorských stanic než v uvedeném zjednodušeném případě.

V druhém experimentu došlo ke srovnání algoritmů BFSO a DDS. Čas výpočtu BFSO byl za použití jednoho i dvou počítačů stejný. Je to zapříčiněno tím, že pro zvolené hodnoty bylo v obou případech dělení prostoru parametrů shodné. Blok, který se prohledával za použití jednoho počítače jako první, splnil ukončovací podmínku. Stejná situace nastala při použití dvou počítačů, což vyústilo ve stejný výpočetní čas. Z obrázku 4.4 je patrná velká rychlost algoritmu DDS vůči BFS.

Algoritmus BFS dělí nejmenší možný počet parametrů nutný pro paralelizaci, aby minimalizoval nároky na komunikaci. Při optimalizaci se pak testuje poměrně velké množství vstupních dat najednou. Algoritmus dynamic data scheduling naopak postupně optimalizuje malé bloky dat a je tak schopen rychleji nálezt řešení s vlastnostmi specifikovanými uživatelem. Rychlejší prohledávání perspektivnějších bloků dat pomocí DDS se na zrychlení výpočtu výrazně projeví až při větším množství dat a vyšších nárocích na přesnost výsledku.

Na závěr několik slov k Alchemi. Podle tvůrců je programování paralelních aplikací s využitím Alchemi poměrně jednoduché a flexibilní. S tímto faktem nelze polemizovat už jen proto, že autoři poskytují celý framework zdarma v plně funkční verzi. Bohužel nepříliš podrobná dokumentace občas způsobuje, že není hned zpočátku jasné, jak přizpůsobit „ohebné" Alchemi k splnění požadovaného cíle a je nutné vyvíjet jistou dávku invence a detektivní práce při pátrání v rozsáhlých knihovnách. S touto vyjímkou se opravdu jedná o vhodný nástroj pro vývoj paralelních algoritmů s výsledky srovnatelnými s jinými podobnými projekty.

Někdy bývá diskutováno použití programovacího jazyku C#, v kterém jsou Alchemi a RapidParallel napsány. Vytvořené programy nejsou ve srovnání například s jazykem C stejně rychlé, ale v tomto případě to není tolik omezující. Aplikace RapidParallel využívá jako engine pro řešení úloh umělé inteligence program RapidMiner. Část psaná v C# probíhá vzhledem k výpočtům RapidMineru zanedbatelně krátký čas. Vývoj programů na platformě .NET je navíc velmi pohodlný a rychlý s možností využít mnohé knihovní metody.

## SEZNAM OBRÁZKŮ

## SEZNAM TABULEK

## SEZNAM ZKRATEK

| | |
|---|---|
| BFS | brute force search, obecná metoda prohledávání prostoru řešení |
| BFSO | brute force search optimization, algoritmus pro paralelní optimalizaci parametrů matematických modelů metodou brute force search |
| DDS | dynamic data scheduling, algoritmus pro paralelní optimalizování matematických modelů využívající dynamické přidělování dat |

## SEZNAM LITERATURY

[1] FOSTER, Ian . KESSELMAN, Carl. *The Grid2: Blueprint for a New Computing Infrastructure.* Elsevier, 2004. 748s. ISBN: 1-55860-933-4

[2] Neuronová síť, Dostupné z: http://cs.wikipedia.org/wiki/Neuronov%C3%A1_s%C3%AD%C5%A5

[3] QUINN, Michael J. *Parallel Programming in C with MPI and OpenMp.* 1.vyd. : Singapore: McGraw-Hill, 2004. 500s. ISBN 007-282256-2

[4] HAUPT, Daniel. *Uživatelský manuál pro verzi Alchemi 1.0*. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2009.

[5] *RapidMiner 4.4* [online]. c2009, [cit. 2009-12-10]

[6] Backpropagation neural network, Dostupné z: http://www.learnartificialneuralnetworks.com/backpropagation.html

[7] Grid computing in 30 seconds, Dostupné z: http://www.gridcafe.org/grid-in-30-sec.html