VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Paralelizace neuronových sítí

Parallelization of Neural Networks

2011

Jakub Rychlý

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava*.

V Ostravě 6. května 2011 .....

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 6. května 2011 .....

Na tomto místě bych rád poděkoval doc. Mgr. Jiřímu Dvorskému, Ph.D. za odborné vedení diplomové práce a také za jeho trpělivost, podporu a rady.

Abstrakt

Tato práce se zabývá paralelizací Kohonenových samoorganizovaných map v prostředí MPI.NET. V práci je popsán úvod do umělých neuronových sítí, na který navazuje popis samoorganizovaných map. Dále jsou rozebrány obecné principy paralelizace s využitím modelu předávání zpráv a probrán standard Message Passing Interface a jeho implementace MPI.NET. Jsou zde popsány a porovnány možné přístupy k paralelizaci Kohonenových samoorganizovaných map. Některé přístupy byly implementovány v prostředí MPI.NET, otestovány na Windows HPC serveru a porovnány mezi sebou z hlediska výkonu.

Klíčová slova: paralelizace, MPI, MPI.NET, umělé neuronové sítě, samoorganizované mapy

Abstract

This thesis deals with parallelization of Kohonen self-organized maps within the MPI.NET environment. The thesis describes an introduction to artificial neural networks and in detail explains self-organized maps. General principles of parallelism using the message passing model, the Message Passing Interface standard and its implementation MPI.NET are also discussed. Various approaches to the parallelization of Kohonen self-organized maps are described and compared to each other. Several approaches have been implemented in MPI.NET, tested on Windows HPC Server and compared to each other in terms of performance.

Keywords: parallelization, MPI, MPI.NET, artificial neural networks, self-organizing maps

Seznam použitých zkratek a symbolů

MPI	–	Message Passing Interface
SOM	–	self-organizing map, samoorganizovaná mapa
SOL	–	Sequential Online Learning
SBL	–	Sequential Batch Learning
POCLNPMW	–	Parallel Online Centralized Learning with Network Partitioning - Master/Worker
PODLNPMW	–	Parallel Online Decentralized Learning with Network Partitioning - Master/Worker
PODLNP	–	Parallel Online Decentralized Learning with Network Partitioning
PBDLDP	–	Parallel Batch Decentralized Learning with Data Partitioning
PBCLNPMW	–	Parallel Batch Centralized Learning with Network Partitioning - Master/Worker
PBDLNPMW	–	Parallel Batch Decentralized Learning with Network Partitioning - Master/Worker
PBHLMW	–	Parallel Batch Hybrid Learning - Mater/Worker

Obsah

Seznam tabulek

Seznam obrázků

Seznam algoritmů

Seznam výpisů zdrojového kódu

1 Úvod

V dnešní době dochází ke stále prudšímu rozvoji informačních technologií. Různá elektronická zařízení doprovází člověka na každém jeho kroku. Do různých informačních systémů proudí čím dál větší množství dat. Ať už tato data pochází z čidel zaznamenávajících výrobní proces, z lékařských přístrojů, ze záznamů banky, nebo z dnes stále více se rozšiřujících sociálních sítí, mohou obsahovat zajímavé informace, které nemusí být na první pohled patrné. Nemusí být patrné ani na druhý pohled, neboť v dnešní záplavě dat není v lidských silách, hledat v těchto datech souvislosti. Proto nabývá na významu strojové dolování informací z dat. K tomuto účelu je možné použít některé umělé neuronové sítě.

Pokud potřebujeme zpracovat enormní množství dat, může to být časově náročné i za použití výpočetní techniky a musíme proto hledat možnosti, jak se dopracovat k výsledkům rychleji. Požadavky na zpracování dat rostou mnohem rychleji než výkon procesorů. Logickým důsledkem je snaha zpracovávat úlohy za pomoci více výpočetních jednotek, což s sebou přináší nové problémy. Úloha musí být vhodně paralelizována, t.j. rozdělena mezi výpočetní jednotky, které ji pak souběžně zpracovávají. Souběžné zpracování musí být určitým způsobem řízeno a synchronizováno. Tyto problémy byly dosud řešeny pouze v superpočítačových centrech. Se zvyšujícím se počtem výpočetních jader v osobních počítačích budou však takovéto záležitosti řešeny stále častěji běžnými programátory.

1.1 Cíl práce

Cílem mé diplomové práce je shrnout a porovnat možnosti paralelní implementace vybraných neuronových sítí. Provedu implementaci několika různých paralelních algoritmů jedné neuronové sítě a provedu testy s cílem určit, který algoritmus je nejvýkonnější, případně k jakým účelům se který algoritmus hodí.

2 Umělé neuronové sítě

Umělé neuronové sítě [1] jsou již dlouhou dobu součástí informatiky. Vznikaly jako modely reálných neuronových sítí v živých organismech, když chtěl člověk pochopit jejich fungování. Původní snahou bylo modelovat chování mozku a nervového systému živých organismů. Později se biologické modely staly základem pro nové metody strojového učení.

2.1 Biologický model neuronu

Neuronové sítě tvoří základ nervového systému organismů. Základní stavební jednotkou tohoto systému je nervová buňka, která je uzpůsobena k příjmu, uchovávání a přenosu informací. Skládá se ze 4 hlavních částí:

1. Dendrity – vedou vzruch směrem k buňce, tvoří vstupy.
2. Axonové vlákno – vede výstupní signál z neuronu směrem k synapsím
3. Synapse – zakončují axonové vlákno a tvoří jakési komunikační rozhraní. Podle procesu učení zesilují nebo zeslabují signál a přivádí ho k dalším neuronům.
4. Tělo neuronu – sčítá signály z okolních neuronů. Takto stanovený vnitřní potenciál vede k vybuzení neuronu.

Biologický neuron je ve skutečnosti podstatně složitější struktura, která stále není dopodrobna prozkoumána. Biologický model neuronu sloužil jako základ pro vytvoření matematického modelu.

2.2 Matematický model neuronu

Matematický model neuronu je základním stavebním prvkem umělých neuronových sítí. Postupem času bylo vyvinuto mnoho různých matematických modelů neuronu. Liší se v závislosti na používaných vstupních datech, v použité matematické funkci, nebo ve složitosti celého modelu.

Neuron je matematický procesor, do kterého vstupuje vektor vstupních signálů a výstupem je skalární výstupní signál. Model neuronu se skládá ze dvou částí. A to z *obvodové* a *aktivační funkce*. Obvodová funkce definuje, jakým způsobem budou uvnitř neuronu zkombinovány hodnoty vstupních signálů. Aktivační funkce definuje, jakým způsobem budou hodnoty vstupních signálů transformovány na výstup. Jedná se tedy o přenosovou funkci.

Nejrozšířenějším modelem neuronu je tzv. *formální neuron*. Na jeho vstupu je vektor $\vec{x}$. Hodnoty jeho složek jsou upraveny podle *váhového vektoru* $\vec{w}$. V tomto modelu se dále vyskytuje *práh neuronu* b. Pokud je vážená hodnota vstupů větší než tento práh, pak je neuron aktivní, jinak je neaktivní. Výstupem neuronu je skalár y.

2.3 Učení neuronové sítě

Podle způsobu učení můžeme rozdělit neuronové sítě do dvou skupin a to na sítě učící se s učitelem a sítě učící se bez učitele.

Při *učení s učitelem* jsou síti předkládána vektorová data ze vstupní množiny a zároveň je předkládána očekávaná hodnota výstupu. Cílem takového učení je změnit synaptické váhy v neuronech tak, aby se minimalizovala chyba mezi reálným a požadovaným výstupem.

Při *učení bez učitele* síť sama nastavuje nové hodnoty synaptických vah neuronů, bez toho, aby byly předkládány jakékoliv očekávané výstupy. Příkladem takovéto sítě je Kohonenova samoorganizovaná mapa, které se budu věnovat ve zbývající části této práce.

2.4 Kohonenova samoorganizovaná mapa

Samoorganizovaná mapa (SOM) [2] je model neuronové sítě představený prof. Teuvo Kohonenem v roce 1982, který umožňuje projekci vysoce dimenzionálních dat na data o nižší dimenzi, nejčastěji na dvourozměrná data, ale prakticky může být výsledná dimenze jakákoliv. Tato projekce vytváří mapu znaků, ve které je zachována topologie dat a která může být užitečná pro detekci a analýzu znaků ve vstupním prostoru. Samoorganizované mapy byly úspěšně použity v různých disciplínách a to včetně rozpoznávání řeči, klasifikaci obrazu a shlukování dokumentů.

Model SOM se skládá ze vstupní a výstupní vrstvy neuronů. Výstupní kompetitivní vrstvu tvoří dvourozměrná mřížka neuronů, které jsou určeny svou pozicí v mřížce a váhovými vektory. Výstupní vrstva může mít různou předem určenou topologii, např. čtvercovou, hexagonální, kruhovou, atd..

SOM je neuronová síť, učící se bez učitele. Je tedy aplikována na data, ve kterých nejsou a priori známy specifické třídy nebo výsledky. Síť sdružuje vstupní vektory se stejnými znaky do skupin a ty zobrazuje jako shluky ve výstupní neuronové vrstvě. SOM tedy může být použita pro porozumění struktuře vstupních dat nebo také k nalezení shluků vstupních záznamů, které mají podobné charakteristiky ve vstupním vektorovém prostoru. Důležitá vlastnost SOM je schopnost zachovat uspořádání vstupních vektorů. Během procesu učení dochází ke kompresi vstupních dat, ale zároveň jsou zachovány jejich topologické vztahy a vzdálenosti. Podobné vstupní vektory jsou tak mapovány na sousední neurony v naučené mapě. Tato samoorganizace je prakticky použitelná ve shlukové analýze, protože poskytuje podrobnosti o vztazích mezi nalezenými shluky.

2.4.1 Výpočet mapy

Existují různé varianty SOM. Dvě z nich budou popsány dále. Jak bylo zmíněno výše, SOM vytváří mapování z n-rozměrného vstupního prostoru do pravidelné dvourozměrné mřížky neuronů. Mějme množinu vstupních vektorů $\vec{x} \in \mathbb{R}^n$ a přidružené váhové vektory $\vec{w}_k \in \mathbb{R}^n, k = 1, \ldots, K$ v neuronech, které jsou uspořádány do pravidelné dvourozměrné mřížky. Dále mějme časový index $t, t = 0, 1, \ldots$, který bude označovat předkládaný vstupní vektor $\vec{x}(t)$ v čase t a váhový vektor $\vec{w}_k(t)$ vypočtený v čase t. Vek-

tory ze vstupní tréninkové množiny jsou během učebního procesu cyklicky předkládány SOM. Jeden průchod vstupními daty se nazývá *epocha*. Počáteční hodnoty váhových vektorů mohou být náhodné vektory, nejlépe z oboru hodnot vstupních dat, nebo je možné váhovým vektorům přiřadit hodnoty K různých vektorů ze vstupních dat. Pro adaptaci váhových vektorů je možné použít následující algoritmy.

2.4.2 On-line algoritmus

V tradičním on-line algoritmu (algoritmus 1) jsou váhové vektory adaptovány po každém předložení vstupního vektoru. Po předložení vstupního vektoru jsou nejdříve spočteny euklidovské vzdálenosti d_k mezi předkládaným vektorem a váhovými vektory.

$$d_k(t) = \|\vec{x}(t) - \vec{w}_k(t)\|^2 \tag{1}$$

Následně je vybrán vítězný neuron, který je často označován jako *best matching unit* (BMU), zde je označen indexem c.

$$d_c(t) = \min d_k(t) \tag{2}$$

Váhové vektory jsou pak adaptovány podle Kohonenova pravidla:

$$\vec{w}_k(t+1) = \vec{w}_k(t) + \alpha(t) h_{ck}(t) \left[\vec{x}(t) - \vec{w}_k(t)\right] \tag{3}$$

kde $\alpha(t)$ je hodnota míry učení v čase t a $h_{ck}(t)$ je funkce okolí. Míra učení určuje míru adaptace váhového vektoru a s postupem učení klesá k nule. Funkce okolí určuje rozsah adaptovaných neuronů $\vec{w}_k(t)$ okolo vítězného neuronu $\vec{w}_c(t)$. Můžeme použít například Gaussovskou funkci okolí

$$h_{ck}(t) = e^{\frac{-\|\vec{r}_k - \vec{r}_c\|}{\sigma(t)^2}} \tag{4}$$

kde $\vec{r}_k$ a $\vec{r}_c$ jsou souřadnice neuronů k a c ve dvourozměrné mřížce. Šířka funkce okolí $\sigma(t)$ klesá během učení od počáteční hodnoty odpovídající velikosti mřížky k nule.

Tento postup je odpovědný za samoorganizaci váhových vektorů. Po předložení každého vstupního vektoru je tedy adaptován vítězný neuron a jeho okolí tak, že adaptované hodnoty jejich váhových vektorů jsou euklidovsky blíže k hodnotě předkládaného vstupního vektoru. Na konci procesu učení váhové vektory v neuronech aproximují distribuční funkci vstupních dat a jejich hodnoty tak mohou být považovány za prototypy reprezentující vstupní data.

2.4.3 Dávkový algoritmus

Online algoritmus adaptuje hodnoty váhových vektorů po předložení každého vektoru ze vstupní množiny. V dávkovém algoritmu (algoritmus 2) jsou váhy neuronů adaptovány jen na konci každé epochy a to podle následujícího vztahu

Algoritmus 1 Online algoritmus

```
Inicializuj váhové vektory w_k
t = 0
for epocha = 1 to N_epoch do
  urči hodnoty α(t) a σ(t)
  for vstup = 1 to N_vstup do
    t = t + 1
    for k = 1 to K do
      spočti vzdálenosti d_k podle rovnice (1)
    end for
    podle rovnice (2) nalezni vítězný neuron c
    for k = 1 to K do
      adaptuj váhové vektory w_k podle rovnice (3)
    end for
  end for
end for
```

$$\vec{w}_k(t_f) = \frac{\sum_{t=t_0}^{t=t_f} h_{ck}(t)\vec{x}(t)}{\sum_{t=t_0}^{t=t_f} h_{ck}(t)} \tag{5}$$

kde t_0 a t_f je začátek a konec aktuální epochy a $\vec{w}_k(t_f)$ jsou váhové vektory neuronů spočtené na konci učební epochy. Sumy jsou počítány průběžně během jednoho průchodu všemi vstupními daty. Vzdálenosti předkládaného vstupního vektoru od váhových vektorů jsou zde počítány podle vztahu

$$d_k(t) = \|\vec{x}(t) - \vec{w}_k(t_0)\|^2 \tag{6}$$

kde $\vec{w}_k(t_0)$ jsou váhové vektory spočtené v předcházející epoše. Vítězný neuron c je pak vybrán stejným způsobem jako v online algoritmu. Také funkce okolí $h_{ck}(t)$ zůstává totožná.

Dávkový algoritmus přináší několik výhod oproti online algoritmu. Jelikož neuronová síť není opakovaně adaptována po každém předložení vstupu, pak nevzniká závislost sítě na pořadí předkládání vektorů ze vstupních dat a nemůže tak dojít k situaci, že vektory předkládané později v tréninkové fázi překryjí výsledky dřívějších vektorů. Také v tomto algoritmu chybí učební koeficient $\alpha(t)$ a nemůže se tak stát, že by učení sítě bylo negativně ovlivněno jeho nevhodnou volbou.

2.4.4 Interpretace výsledků

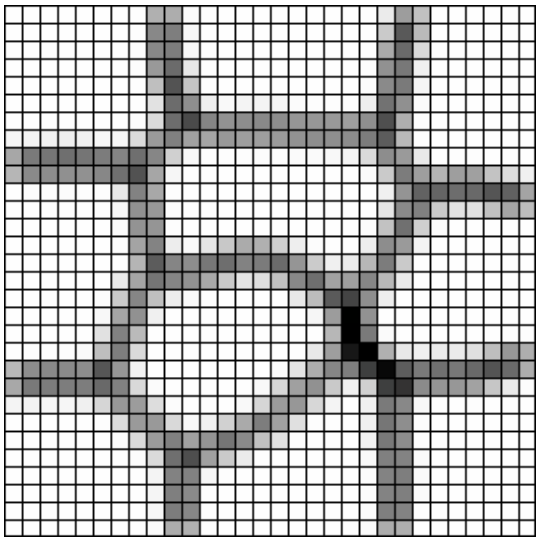
Naučenou SOM můžeme vyhodnocovat několika způsoby. Pokud je počet neuronů v použité síti podstatně menší než počet vstupních dat, pak se na výsledné váhové vektory můžeme dívat jako na středy shluků, které se nachází ve vstupních datech. V tomto pří-

Algoritmus 2 Dávkový algoritmus

```
Inicializuj váhové vektory w_k
t = 0
for epocha = 1 to N_epochy do
  urči novou hodnotu σ(t)
  inicializuj čitatele a jmenovatele v rovnici (5) na nulu
  for vstup = 1 to N_vstup do
    t = t + 1
    for k = 1 to K do
      podle rovnice (6) spočti vzdálenosti d_k
    end for
    pomocí rovnice (2) nalezni vítězný neuron c
    for k = 1 to K do
      přičti hodnoty k čitateli a jmenovateli v rovnici (5)
    end for
  end for
  for k = 1 to K do
    podle rovnice (5) adaptuj váhové vektory w_k
  end for
end for
```

padě musíme dopředu odhadnout počet shluků v datech a zvolit neuronovou síť s odpovídajícím počtem neuronů.

Další možností ve vyhodnocení SOM je využití U-matic [3]. *U-matice* je matice se stejnou topologií jako použitá SOM. Každé číslo v matici náleží jednomu neuronu a odpovídá jeho U-výšce. *U-výška* je definována jako vzdálenost váhového vektoru neuronu od váhových vektorů jeho bezprostředních sousedů. Čím menší je U-výška, tím jsou neurony blíže u sebe, což signalizuje silnější shluk. Pokud hodnoty U-matice vydělíme největší nalezenou U-výškou, pak dostaneme *relativní U-matici*, jejíž hodnoty jsou z intervalu <0;1>. Pokud tomuto intervalu přiřadíme barevný gradient, pak můžeme relativní U-matici snadno graficky zobrazit. Na obrázku 1 je zobrazena naučená SOM. Černá barva zde odpovídá hodnotě 0 a bílá barva hodnotě 1. Z obrázku je zřejmé, že vstupní data obsahovala 10 shluků.



Obrázek 1: Příklad U-matice

3 Paralelní programování a MPI.NET

Učení rozsáhlých SOM, nebo zpracování velkého množství vstupních dat je poměrně časově náročné. Je tedy snaha tyto časové nároky omezit. Nabízejícím se řešením je využití výkonnějšího hardwaru. Je však zřejmé, že výkon sekvenčních počítačů nemůže stále růst, například kvůli fyzikálním limitům. Lepším řešením je využití více výpočetních jednotek, které navzájem komunikují a spolupracují tak na řešení daného problému. Výpočetní náklad se tak rozdělí mezi výpočetní jednotky, úloha může mít větší rozsah a bude dříve dokončena. Tento postup se obecně nazývá *paralelní zpracování*.

V této kapitole si vysvětlíme paralelní programovací techniky založené na modelu předávání zpráv. Dále probereme úvod do MPI, abychom na něj mohli navázat s popisem prostředí MPI.NET, ve kterém jsme implementovali paralelní algoritmy pro výpočet SOM. Z funkcí MPI.NET popíšeme ty, které jsme použili při implementaci. Přiblížíme také jak MPI.NET funguje na pozadí a jaké to má důsledky na rychlost paralelní aplikace.

3.1 Model předávání zpráv

Model předávání zpráv je zobecněním sekvenčního modelu programování, kdy je uvažováno několik instancí sekvenčního modelu. Využívá se několik výpočetních jednotek a každá má svůj paměťový prostor. Na každé výpočetní jednotce běží jedna instance sekvenčního procesu. Pro vyřešení problému procesy spolupracují pouze zasíláním zpráv a nevyužívají sdílenou paměť. Zprávy přenášejí obsahy proměnných z jednoho procesu do proměnných jiného procesu. Předávání zpráv zahrnuje také synchronizaci procesů. Je to velmi obecný přístup, podporovaný většinou paralelních systémů. Existuje více technických realizací modelu předávání zpráv. My se v této práci zaměříme především na standard MPI a jeho implementaci do frameworku .NET.

3.2 MPI

Message Passing Interface (MPI) [4] vznikl z potřeby standardizace modelu předávání zpráv. Před jeho vznikem existovaly pouze různé roztříštěné a navzájem nekompatibilní implementace modelu předávání zpráv. V roce 1992 tak byla ustanovena pracovní skupina, ke které se následně přidali další jednotlivci i organizace. Vzniklo tak MPI fórum, které v roce 1994 vydalo první verzi standardu MPI. Cíli MPI projektu bylo poskytnout přenositelnost zdrojového kódu mezi různými platformami, umožnit efektivní implementaci modelu předávání zpráv, podporovat heterogenní paralelní architektury, poskytnout sémantiku nezávislou na programovacím jazyce. První verze MPI poskytovala rozhraní MPI funkcí pro jazyky C a Fortran. Od verze 2.0 je poskytováno také rozhraní pro jazyk C++. MPI je specifikace jak pro vývojáře, tak pro uživatele knihoven modelu předávání zpráv. MPI tedy nejsou samotné knihovny. Existuje více implementací MPI od více poskytovatelů. Zdrojové kódy by měly být mezi těmito implementacemi přenositelné.

3.2.1 Programování v MPI

Jak již bylo řečeno model MPI je založen na předávání zpráv. Na rozdíl od programování s vlákny má v MPI každý proces svůj paměťový prostor a udržuje si svůj stav. Tento paměťový prostor a stav nemůže být sledován ani ovlivněn jiným procesem. Proto mohou být spolupracující procesy rozmístěny na různých strojích v sítí, nebo dokonce na různých architekturách.

Většina MPI programů je psána s využitím SPMD (Single Program, Multiple Data – jeden program, vícero dat) paralelního modelu. Každý proces paralelní aplikace je tedy instancí stejného programu, ale pracuje s rozdílnou částí dat. Data jsou mezi procesy rozdělena tak, aby stroje, na nichž procesy běží, byly rovnoměrně zatížené. Procesy zpracovávají data v rámci své lokální paměti. Předávání výsledků a synchronizace je pak realizována předáváním zpráv.

MPI umožňuje jedním příkazem spustit stejný program na několika výpočetních uzlech. Spuštěné procesy jsou pak identifikovány pomocí ranku, který slouží také jako adresa při výměně zpráv. Je to celé číslo z intervalu od 0 do P-1, kde P je počet spuštěných procesů. Na základě ranku je pak možné, aby různé procesy stejného programu prováděly různé funkce.

3.3 MPI.NET

MPI.NET [5] je vysoce výkonná a jednoduše použitelná implementace MPI pro prostředí .NET firmy Microsoft. Poskytuje podporu pro všechny jazyky na platformě .NET. Tato implementace byla vyvinuta na Indiana University v USA. MPI.NET vytváří objektovou obálku v řízeném prostředí .NET nad implementací MPI firmy Microsoft v neřízeném prostředí. Přináší tak snadnost programování v řízeném prostředí pro paralelní programování a rozšiřuje model MPI o další vlastnosti, jako je například automatická serializace přenášených objektů.

3.3.1 Komunikátory

Každý netriviální program zpravidla využívá *komunikátor*, který představuje abstrakci komunikačního prostoru, v rámci kterého mohou procesy mezi sebou komunikovat. Proces do něj zahrnutý může komunikovat pouze s procesy v rámci tohoto komunikátoru. Nemůže se stát, aby se promíchaly zprávy mezi různými komunikátory. Komunikátor může obsahovat více procesů a každý proces může být zahrnut do několika komunikátorů. Každý proces má v rámci komunikátoru přiřazen jednoznačný identifikátor, v MPI označovaný jako *rank*. Dvojice komunikátor a rank tedy jednoznačně adresuje proces, se kterým je potřeba komunikovat.

Každý MPI program má od počátku definovány dva komunikátory. Prvním je *self komunikátor*, který obsahuje pouze vlastní proces a je používán velmi zřídka. Druhým je *world komunikátor*, který obsahuje všechny spuštěné procesy. V MPI.NET je přístupný jako Communicator.world. Pokud je potřeba vytvořit nový oddělený komunikační prostor,

je to možné učinit klonováním některého stávajícího komunikátoru, nebo vytvořením komunikátoru, který bude podmnožinou některého stávajícího komunikátoru.

3.3.2 Základní funkce

Prostředí MPI zpravidla potřebuje provést na začátku programu nějaká počáteční nastavení a na jeho konci zase nějaké úklidové operace. Proto podle standardu MPI musí být na začátku programu zavolána funkce MPI_Init, pak mohou být teprve využívány ostatní funkce MPI, a na konci programu je potřeba zavolat funkci MPI_Finalize.

V MPI.NET je prostředí MPI zapouzdřeno do objektu Environment. V rámci jeho konstruktoru je pak volána funkce MPI_Init a v rámci jeho metody Dispose je volána funkce MPI_Finalize. Správnou inicializaci a ukončení prostředí tedy v jazyce C# nejlépe provedeme následující konstrukcí:

```
using ( new MPI.Environment(ref args) ){
    /* všechna další volání MPI funkcí */
}
```

Výpis 1: Inicializace prostředí MPI.NET

V programu často potřebujeme zjistit rank aktuálního procesu. Získáme ho z vlastnosti Rank objektu komunikátoru. Podle tohoto ranku, pak můžeme řídit vykonávání programu.

```
Intracommunicator comm = Communicator.world;
if (comm.Rank == 0)
{
    /*program pro master proces*/
}
else
{
    /*program pro ostatní procesy*/
}
```

Výpis 2: Využití ranku pro větvení programu

Jednou z úloh paralelního programu je rozdělit práci mezi procesy. Aby toto bylo možné, musí program znát, kolik procesů bylo vlastně spuštěno. Počet spuštěných procesů se zjistí z vlastnosti Size objektu komunikátoru.

```
Intracommunicator comm = Communicator.wordl;
int pocetProcesu = comm.Size;
```

Výpis 3: Zjíštění počtu spuštěných procesů

3.4 Dvoubodová komunikace

Dvoubodová komunikace umožňuje přenášet data z jednoho procesu do druhého. Zprávy jsou přenášeny prostřednictvím komunikátorů. Každý komunikátor určuje oddělený komunikační prostor, proto se nemůže stát, že by byla zpráva přenesena napříč komuniká-

tory. Kromě vlastního obsahu nese každá zpráva také celočíselnou značku, která umožňuje uživateli oddělit různé typy zpráv. Pořadí zpráv zůstává zachováno.

Data přenášených zpráv musí být označena také jejich datovým typem. Standard MPI předepisuje vlastní datové typy (MPI_INTEGER, MPI_DOUBLE, ...) pro posílání zpráv. Tyto datové typy pak odpovídají datovým typům v konkrétním programovacím jazyce (pro jazyk C např.: **int**, **double**, ...). Standard MPI také nabízí možnosti, jak v rámci jedné zprávy přenášet data, která jsou složena z více datových typů (např. struktury jazyka C). Pro tento účel slouží takzvané odvozené datové typy. Více o přenosech a datových typech nalezneme ve standardu MPI [4].

MPI.NET, jakožto nadstavba nad implementací standardu MPI, se snaží programátory oprostit od vytváření odvozených datových typů a navíc přidává možnost posílat prostřednictvím zpráv celé objekty. Na tomto místě je však nutné pochopit, jak funguje MPI.NET na pozadí, protože přenos různých druhů dat má různý výkon.

Bez ztráty přenosové rychlosti oproti podkladové implementaci MPI jsou přenášeny hodnotové typy C#. Hodnotové typy se v C# alokují na zásobníku a patří mezi ně základní datové typy (např. **int**, **double**, ...) a struktury. Přenos základních datových typů je na pozadí mapován na přenos základních datových typů MPI. Při přenosu struktur je na pozadí nejprve vytvořen odvozený datový typ MPI a ten je následně uložen do cache a tudíž může být využíván opakovaně. Přenos struktur v MPI.NET je tak pro programátora pohodlnější a rychlostně je srovnatelný s přenosem odvozených datových typů v MPI.

MPI.NET oproti standardu MPI umožňuje přenos objektů. Zde můžeme také využívat polymorfismu a to tak, že odešleme objekt potomka a přijmeme ho jako objekt rodiče. Při přenosu je objekt na pozadí serializován a následně přenášen jako pole bajtů. Třídy, jejichž objekty chceme přenášet mezi procesy, tedy musí být označeny atributem Serializable. Jelikož velikost serializovaných objektů může být různá a standard MPI neumožňuje přenášet pole o předem neznámé velikosti (velikost musí být předem známá jak na straně odesílatele, tak na straně příjemce), přistoupili implementátoři MPI.NET k přenosu této velikosti ve zvláštní hlavičkové zprávě, která předchází zprávu datovou. Vytváření a odesílání hlavičkové zprávy je prováděno transparentně na pozadí a programátor se o to nemusí starat. Programátor by si však měl uvědomit, že odesílání další přidané zprávy na pozadí přináší jistou míru režie, která se negativně podepisuje na přenosové rychlosti.

Odesílání zprávy se provádí pomocí metody Send třídy komunikátoru. Její hlavička je následující:

```
public void Send<T>(
    T value,
    int dest,
    int tag
)
```

Výpis 4: Communicator.Send

T zde označuje přenášený datový typ, dest je rank cílového procesu (musí být rozdílný od zdrojového procesu) , tag je značka, která umožňuje programátorovi určit konkrétní druh zprávy.

Pro příjem zprávy slouží metoda Receive třídy komunikátoru. Volání metody Receive je blokovací, to znamená, že návrat z volání se neprovede dříve, než jsou data přijata. Její hlavička je následující:

```
public T Receive<T>(
    int source,
    int tag
)
```

Výpis 5: Communicator.Receive

T označuje přijímaný datový typ, source je rank procesu, od kterého chceme data přijmout. Pokud chceme přijmout data od jakéhokoliv procesu, můžeme využít konstantu Communicator.anySource. Tag je značka zprávy, kterou chceme přijmout. Pokud chceme přijmout zprávu s jakoukoliv značkou, pak využijeme konstantu Communicator.anyTag.

3.5 Kolektivní komunikace

Kolektivní komunikace je prováděna v rámci skupin, daných komunikátorem. Dvoubodová a kolektivní komunikace je na sobě nezávislá, a to i navzdory použití stejného komunikátoru. Z toho například vyplývá, že kolektivní odeslání není možné přijmout pomocí funkce pro dvoubodový příjem. U kolektivní komunikace se nevyužívají značky.

Nejjednodušší kolektivní operací je *bariéra*, která slouží jako synchronizační mechanismus pro skupinu procesů. Je implementována jako metoda Barrier, třídy komunikátoru. Tuto metodu musí zavolat všechny procesy v rámci komunikátoru. Metoda je blokující. Návrat z volání se provede, až tuto metodu zavolá poslední proces z komunikátoru.

Další častou kolektivní operací je *broadcast*, který slouží pro rozeslání dat z jednoho procesu všem ostatním procesům v komunikátoru. Tato operace je v MPI.NET implementována jako metoda Broadcast třídy Intracommunicator. Intracommunicator je potomkem třídy Communicator a zapouzdřuje navíc právě většinu kolektivních operací. Metoda Broadcast má následující předpis:

```
public void Broadcast<T>(
    ref T value,
    int root
)
```

Výpis 6: Intracommunicator.Broadcast

T označuje přenášený datový typ, value je na odesílací straně proměnná, jejíž hodnota se bude rozesílat ostatním procesům a na přijímací straně pak proměnná, do které se uloží přijatá hodnota, root je rank procesu, který data odesílá.

Gather je kolektivní operace, která od každého procesu přijme část dat a tyto části pak uloží do pole v cílovém procesu. Ve výsledném poli pak bude na i-té pozici hodnota z pro-

cesu s rankem i. Operace gather je realizována metodou Gather třídy Intracommunicator. Její předpis je následující:

```
public T[] Gather<T>(
    T value,
    int root
)
```

Výpis 7: Intracommunicator.Gather

T označuje datový typ, value je proměnná jejíž hodnota bude odeslána, root je rank cílového procesu, který příjme pole dat, které tato metoda vrací. Metoda vrací pole definovaného typu pouze procesu určenému parametrem root, ostatním procesům vrací **null**.

Inverzní operací ke gather je operace *scatter*. Tato operace rozdělí pole dat ze zdrojového procesu mezi všechny procesy v rámci komunikátoru tak, že proces s i-tým rankem přijme i-tý prvek odeslaného pole. V MPI.NET ji najdeme jako metodu Scatter třídy Intercommunicator. Scatter má následující předpis:

```
Public T Scatter<T>(
    T[] values,
    int root
)
```

Výpis 8: Intracommunicator.Scatter

T označuje datový typ přenášených dat, values je pole, které bude rozděleno mezi procesy, pole musí mít právě tolik prvků, kolik je procesů v komunikátoru. Root je rank procesu, který data rozesílá. Metoda vrací jednu hodnotu typu T ve všech volajících procesech.

Kolektivní operaci *allgather* si můžeme představit jako operaci gather, kterou následuje operace broadcast. Allgather přijímá od každého procesu jednu hodnotu. Tyto hodnoty následně poskládá do pole tak, že na i-té pozici bude hodnota od procesu s rankem i. Výsledné pole pak přijme každý proces v rámci komunikátoru. Tato operace je realizována metodou Allgather třídy Intracommunicator. Operace má následující předpis:

```
public T[] Allgather<T>(
    T value
)
```

Výpis 9: Intracommunicator.Allgather

T označuje datový typ. Value je hodnota odesílaného prvku. Metoda vrací pole prvků definovaného datového typu.

Reduce je kolektivní operace, která zkombinuje hodnoty odeslané procesy do jedné hodnoty umístěné v cílovém procesu. Pro určení způsobu, jakým budou hodnoty zkombinovány, je možné využít některou z předdefinovaných operací (suma, minimum, maximum, ...), nebo vytvořit operací vlastní. V MPI.NET je operace reduce implementována jako metoda Reduce třídy Intracommunicator. Vlastní redukční operaci jí můžeme dodat pomocí delegáta. Předdefinované operace nalezneme jako statické veřejné vlastnosti třídy Operation. Signatura operace Reduce je následující:

```
public T Reduce<T>(
    T value,
    ReductionOperation<T> op,
    Int  root
)
```

Výpis 10: Intracommunicator.Reduce

T je datový typ. Value je odesílaná hodnota. Op je asociativní redukční operace, která musí splňovat signaturu delegáta ReductionOperation<T>. Root je rank procesu, kterému bude vrácen výsledek redukce. Ostatním procesům bude vrácena hodnota **null**. Delegát ReductionOperation je definován takto:

```
public delegate T ReductionOperation<T>( T x, T y )
```

Výpis 11: Delegát ReductionOperation

Poslední kolektivní operací, kterou zde uvedeme je operace *allreduce*. Její funkce odpovídá operaci reduce následovanou operaci broadcast. Výslednou hodnotu tedy dostanou všechny procesy. V MPI.NET se nachází jako metoda Allreduce třídy Intercommunicator . Signatura této operace je následující:

```
pulblic  T Allreduce<T>(
    T value,
    ReductionOperation<T> op
)
```

Výpis 12: Intracommunicator.Allreduce

T je datový typ, value je odesílaná hodnota a op je redukční operace.

4 Paralelizace Kohonenovy samoorganizované mapy

Paralelizace znamená rozdělení úlohy na části, které mohou být vykonávány souběžně. Nejdříve tedy musí být identifikována úloha, kterou chceme zpracovávat a následně části úlohy, které je možné rozdělit a zpracovávat souběžně.

V literatuře [7, 8, 9, 10] se nachází rozličné množství přístupů k paralelizaci SOM. Podle veličiny, která je rozdělována mezi výpočetní jednotky, můžeme rozlišit paralelní algoritmy

- s rozdělením neuronové sítě,
- s rozdělením dat,
- hybridní algoritmy.

Podle algoritmu, ze kterého vychází, bychom je mohli rozdělit na techniky založené na

- online algoritmu,
- dávkovém algoritmu.

4.1 Rozdělení sítě

Každý proces pracuje se všemi vstupními daty, ale jen s části neuronové sítě. Po předložení vstupního záznamu nalezne každý proces v přidělené části sítě neuron, který je vstupnímu záznamu nejblíže. Tyto neurony nazvěme kandidáty na vítěze. Hledání kandidátů na vítěze je prováděno paralelně a po jejich nalezení musí dojít k meziprocesní komunikaci a vybrání jediného vítězného neuronu. Následně pokračujeme v závislosti na použitém algoritmu.

4.1.1 Online algoritmus

Pro paralelizaci rozdělením sítě se často využívá online algoritmus [8]. Nalezli jsme tři způsoby implementace.

První způsob zachycuje algoritmus 3. Využívá jeden programový kód pro všechny procesy. Po nalezení kandidátů na vítěze procesy rozdistribuují tyto kandidáty mezi ostatní procesy. Procesy pak redundantně vyhledávají mezi kandidáty vítězný neuron a podle něj adaptují své části neuronové sítě. Hlavní výhodou tohoto postupu je, že využívá originální Kohonenovo pravidlo a výsledná síť při paralelním zpracování odpovídá výsledné síti ze sekvenčního algoritmu. Nevýhodou je, že v každé iteraci (po předložení každého vstupního vektoru) musí proces komunikovat se všemi ostatními procesy, a to proto, aby se určil vítězný neuron a podle jeho polohy v mřížce se mohly adaptovat části sítě v procesech. Při této komunikaci bude docházet k častému přenosu malých dat, tím

pádem bude zatížená především latencí. Dá se očekávat, že to bude působit jako překážka škálovatelnosti. Při konstantní velikosti neuronové sítě a zvětšujícím se počtu výpočetních uzlů bude zřejmě narůstat množství komunikační režie, což může degradovat zrychlení.

Algoritmus 3 Paralelní online algoritmus s rozdělením sítě

```
Inicializuj váhové vektory w_k identicky ve všech procesech
t = 0
for epocha = 1 to N_epoch do
  urči hodnoty α(t) a σ(t)
  for vstup = 1 to N_vstup do
    t = t + 1
    for k ∈ [neurony přiřazené procesu] do
      spočti vzdálenosti d_k podle rovnice (1)
    end for
    MPI_Allgather zajistí rozdistribuování všech vzdáleností d_k mezi všechny procesy
    podle rovnice (2) nalezni vítězný neuron c
    for k ∈ [neurony přiřazené procesu] do
      adaptuj váhové vektory w_k podle rovnice (3)
    end for
  end for
end for
```

Druhý způsob je znázorněn algoritmem 4. Tento algoritmus je typu master/worker. Worker procesy vyhledávají kandidáty na vítězný neuron v části sítě a posílají je do master procesu. Ten z kandidátů na vítěze vybere globálního vítěze a podle něj aktualizuje neuronovou síť. Příslušné části neuronové sítě pak rozešle do worker procesů. Toto se opakuje v každé iteraci, tedy při každém předložení vstupního vektoru. Tento algoritmus se nejeví příliš vhodný, protože paralelizuje pouze vyhledávání vítězného neuronu a navíc v každé iteraci přenáší velké množství dat v podobě části neuronové sítě.

Algoritmus 5 je opět typu master/worker. V tomto případě jediným úkolem master procesu je vybrat z došlých kandidátů na vítěze globálního vítěze a rozeslat ho zpět do worker procesů. Worker procesy již samy adaptují přidělené části neuronové sítě. Tento algoritmus je podobný algoritmu 3, má tudíž i stejné výhody a nevýhody. Drobný rozdíl je v tom, že globální vítěz není vyhledáván redundantně ve všech procesech, ale pouze v master procesu. Worker procesy však musí čekat, až master proces nalezne globálního vítěze a nemohou mezi tím vykonávat jinou práci. Drobný rozdíl je i v komunikaci. U algoritmu 3 komunikuje každý proces se všemi ostatními. U tohoto algoritmu worker procesy komunikují pouze s master procesem. Komunikace je tedy méně.

4.1.2 Dávkový algoritmus

Z paralelních online algoritmů 4 a 5 můžeme odvodit paralelní dávkové algoritmy.

Algoritmus 4 Paralelní centralizovaný online algoritmus s rozdělením sítě - master/worker

```
{master proces}
Inicializuj váhové vektory w⃗_k
t = 0
for epocha = 1 to N_epochy do
  t = t + 1
  urči hodnoty α(t) a σ(t)
  for vstup = 1 to N_vstup do
    MPI_Gather přijme d_i ze všech worker procesů
    pomocí rovnice (2) nalezni vítězný neuron c
    for k = 1 to K do
      adaptuj váhové vektory w⃗_k podle rovnice (3)
    end for
    rozešli příslušné části adaptované neuronové sítě worker procesům
  end for
end for

{worker procesy}
for epocha = 1 to N_epochy do
  for vstup = 1 to N_vstup do
    for k ∈ [neurony přiřazené procesu] do
      podle rovnice (6) spočti vzdálenosti d_k
    end for
    pomocí rovnice (2) nalezni vítězný neuron c
    MPI_Gather odešle d_c do master procesu
  end for
  přijmi část adaptované neuronové sítě z master procesu
end for
```

Algoritmus 5 Paralelní decentralizovaný online algoritmus s rozdělením sítě - master/worker

```
{master proces}
for epocha = 1 to N_epochy do
  for vstup = 1 to N_vstup do
    MPI_Gather přijmi d_i ze všech worker procesů
    pomocí rovnice (2) nalezni vítězný neuron c
    MPI_Bcast rozešli hodnotu c worker procesům
  end for
end for

{worker procesy}
Inicializuj váhové vektory w_k
t = 0
for epocha = 1 to N_epochy do
  urči hodnoty α(t) a σ(t)
  for vstup = 1 to N_vstup do
    t = t + 1
    for k ∈ [neurony přiřazené procesu] do
      podle rovnice (6) spočti vzdálenosti d_k
    end for
    pomocí rovnice (2) nalezni vítězný neuron c
    MPI_Gather odešle d_c do master procesu
    MPI_Bcast přijme hodnotu c z master procesu
    for k ∈ [neurony přiřazené procesu] do
      přičti hodnoty k čitateli a jmenovateli v rovnici (5)
    end for
  end for
  for k ∈ [neurony přiřazené procesu] do
    adaptuj váhové vektory w_k podle rovnice (3)
  end for
end for
```

Algoritmus 6 Paralelní centralizovaný dávkový algoritmus s rozdělením sítě - master/worker

{master proces}
for epocha = 1 to N_{epochy} **do**
  urči novou hodnotu $\sigma(t)$
  inicializuj čitatele a jmenovatele v rovnici (5) na 0
  for vstup = 1 to N_{vstup} **do**
    MPI_Gather přijme d_i ze všech worker procesů
    pomocí rovnice (2) nalezni vítězný neuron c
    for k = 1 to K **do**
      přičti hodnoty k čitateli a jmenovateli v rovnici (5)
    end for
  end for
  rozešli příslušné části čitatelů a jmenovatelů z rovnice (5) worker procesům
end for

{worker procesy}
Inicializuj váhové vektory
t = 0
for epocha = 1 to N_{epochy} **do**
  for vstup = 1 to Nvstupů **do**
    t = t + 1
    for $k \in$ [neurony přiřazené procesu] **do**
      podle rovnice (6) spočti vzdálenosti d_k
    end for
    pomocí rovnice (2) nalezni vítězný neuron c
    MPI_Gather odešle d_c do master procesu
  end for
  přijmi část čitatelů a jmenovatelů z rovnice (5) z master procesu
  for $k \in$ [neurony přiřazené procesu] **do**
    podle rovnice (5) adaptuj váhové vektory $\vec{w}_k$
  end for
end for

Algoritmus 6 je odvozen z algoritmu 4. Je tedy implementován pomocí master/worker modelu. Neurony sítě jsou rozděleny mezi worker procesy, které nacházejí kandidáty na vítěze ve své části sítě a posílají je do master procesu. Ten nalezne globálního vítěze a akumuluje sumy v rovnici (5). Toto je opakováno pro všechna vstupní data. Po předložení všech vstupních dat master proces odešle příslušné součty z rovnice (5) do worker procesů, které provedením podílu v rovnici (5) adaptují váhové vektory neuronů ve své části sítě. Tento algoritmus není příliš výhodný, protože paralelizuje pouze vyhledávání vítěze a adaptaci neuronů, zato akumulace sum v rovnici (5) není paralelizována vůbec. V každé iteraci se přenáší malá data (kandidát na vítěze) a v každé epoše data poměrně velká (sumy čitatelů a jmenovatelů). Tato komunikace může mít negativní vliv na výkon algoritmu.

Algoritmus 7 je odvozen z algoritmu 5. Opět využívá master/worker model. Mezi worker procesy jsou rozděleny neurony sítě. Worker procesy ve svých částech sítě nacházejí kandidáty na vítěze, které odesílají do master procesu. Ten z doručených kandidátů vybírá globálního vítěze a rozesílá ho do worker procesů, které podle něj akumulují sumy v rovnici (5). Toto se opakuje pro všechna vstupní data. Na konci epochy, po předložení všech vstupních dat, worker procesy adaptují své části neuronové sítě podle rovnice (5). Algoritmus je zatížen stejnou komunikací jako algoritmus 5 a mohl by dosahovat podobných časů výpočtů.

4.2 Rozdělení dat

Algoritmus 8 znázorňuje paralelizaci s rozdělením vstupních dat za použití dávkového algoritmu [8]. Vstupní data jsou rovnoměrně rozdělena mezi procesy a každý proces si udržuje kopii celé neuronové sítě. Výpočet probíhá obdobně jako při sekvenčním algoritmu. Každý proces během učící epochy předkládá své kopii sítě své vstupní vektory, vyhledává vítězný neuron a následně pro všechny neurony akumuluje čitatele a jmenovatele podle rovnice (5). V rámci epochy je to provedeno pro všechna vstupní data přiřazená k procesu. Před koncem epochy je nutné sečíst čitatele a jmenovatele rovnice (5) mezi procesy a výsledek doručit do všech procesů, aby si podle něj mohly aktualizovat své kopie sítě. Z principu tohoto algoritmu je zřejmé, že je zapotřebí méně častá komunikace, než u paralelních online algoritmů. Zprávy se přenáší jen jednou za epochu, avšak jsou objemnější než u online algoritmu s rozdělením sítě. Rozdělení vstupních dat mezi procesy dává větší naději na dobrou škálovatelnost, protože vstupních dat je zpravidla větší množství než neuronů v síti a lépe tak můžeme využít větší množství procesorů.

4.3 Hybridní algoritmus

Jako hybridní je označován algoritmus, který mezi procesy rozděluje jak data, tak i síť. V literatuře se nacházejí hybridní algoritmy založené jak na dávkovém učení [7], tak i na online učení [10]. Tyto algoritmy využívají faktu, že počátečního topologického uspořádání neuronové sítě je dosaženo již po poměrně krátkém učení. Poté již vstupní vektory spadají do určité oblasti sítě. Neuronovou síť tedy můžeme rozdělit na části a k těmto částem přiřadit vstupní vektory, které do nich spadají. Každá část neuronové sítě je pak

Algoritmus 7 Paralelní decentralizovaný dávkový algoritmus s rozdělením sítě - master/worker

```
{master proces}
for epocha = 1 to N_epochy do
  for vstup = 1 to N_vstup do
    MPI_Gather přijme d_i ze všech worker procesů
    pomocí rovnice (2) nalezni vítězný neuron c
    MPI_Bcast rozešle hodnotu c worker procesům
  end for
end for

{worker procesy}
Inicializuj váhové vektory
t = 0
for epocha = 1 to N_epochy do
  urči novou hodnotu σ(t)
  inicializuj čitatele a jmenovatele v rovnici (5) na 0
  for vstup = 1 to N_vstup do
    t = t + 1
    for k ∈ [neurony přiřazené procesu] do
      podle rovnice (6) spočti vzdálenosti d_k
    end for
    pomocí rovnice (2) nalezni vítězný neuron c
    MPI_Gather odešle d_c do master procesu
    MPI_Bcast přijme hodnotu c z master procesu
    for k ∈ [neurony přiřazené procesu] do
      přičti hodnoty k čitateli a jmenovateli v rovnici (5)
    end for
  end for
  for k ∈ [neurony přiřazené procesu] do
    podle rovnice (5) adaptuj váhové vektory w⃗_k
  end for
end for
```

Algoritmus 8 Paralelní dávkový algoritmus s rozdělením dat

```
Inicializuj váhové vektory
t = 0
for epocha = 1 to N_epochy do
  urči novou hodnotu σ(t)
  inicializuj čitatele a jmenovatele v rovnici (5) na 0
  for vstup ∈ [vstupy přiřazené procesu] do
    t = t + 1
    for k = 1 to K do
      podle rovnice (6) spočti vzdálenosti d_k
    end for
    pomocí rovnice (2) nalezni vítězný neuron c
    for k = 1 to K do
      přičti hodnoty k čitateli a jmenovateli v rovnici (5)
    end for
  end for
  MPI_Allreduce sečte sumy z rovnice (5) napříč procesy a součty uloží do každého
  procesu
  for k = 1 to K do
    podle rovnice (5) adaptuj váhové vektory w_k
  end for
end for
```

učena jen příslušnou částí vstupních dat. Výhodou je, že v této fázi je prováděno méně výpočtů (části sítě jsou učeny částmi dat). Nevýhodou je, že tyto algoritmy nepodávají shodné výsledky se sekvenčními algoritmy. Může se totiž stát, že okolí vítězného neuronu přesahuje do sousední části mapy a není v ní adaptováno. Jelikož je adaptované okolí v průběhu učení zmenšováno, stává se to však stále méně často. Příkladem hybridního algoritmu je algoritmus 9.

Algoritmus 9 Paralelní hybridní algoritmus

{master proces}
Rozděl množinu vstupních vektorů
Do každého worker procesu zašli indexy přidělených vstupních vektorů
for epocha = 1 to N_{epochy} **do**
  urči novou hodnotu $\sigma(t)$
  if segmentační epocha **then**
    přijmi histogramy z worker procesů
    rozděl mapu a vstupní vektory
    rozešli adaptovanou mapu, hodnotu $\sigma(t)$, nové indexy přidělených vstupních vektorů a $region_p$ mapy do worker procesů p
  else
    rozešli adaptovanou mapu a hodnotu $\sigma(t)$ do všech worker procesů p
  end if
  přijmi sumy z rovnice (5) od worker procesů p
  for k = 1 to K **do**
    podle rovnice (5) adaptuj váhové vektory $\vec{w}_k$
  end for
end for

{worker procesy}
Inicializuj váhové vektory
t = 0
for epocha = 1 to N_{epochy} **do**
  if segmentační epocha **then**
    spočti histogram a zašli do master procesu
    přijmi adaptovanou mapu, hodnotu $\sigma(t)$, nové indexy přidělených vstupních vektorů a $region_p$ mapy
  else
    přijmi adaptovanou mapu a hodnotu $\sigma(t)$
  end if
  inicializuj čitatele a jmenovatele v rovnici (5) na 0
  for vstup $\in$ [vstupy přiřazené procesu] **do**
    t = t + 1
    for $k \in region_p$ **do**
      podle rovnice (6) spočti vzdálenosti d_k
    end for
    pomocí rovnice (2) nalezni vítězný neuron c
    for $k \in region_p$ **do**
      přičti hodnoty k čitateli a jmenovateli v rovnici (5)
    end for
  end for
  odešli sumy z rovnice (5) do master procesu
end for

4.4 Implementace

Algoritmy popsané v předchozí kapitole byly implementovány v prostředí MPI.NET. Vybrané algoritmy byly implementovány ve více variantách za účelem optimalizace komunikačních nákladů. Tabulka 1 shrnuje implementované algoritmy a jejich varianty.

4.4.1 Paralelní online algoritmus s rozdělením sítě a centralizovaným učením

V dalším textu budeme používat označení *POCLNPMW* (Parallel Online Centralized Learning with Network Partitioning - Master/Worker). Jedná se o implementaci algoritmu 4, který využívá dvourozměrnou obdélníkovou SOM s online učením a rozdělením mapy mezi procesy. Algoritmus je implementován s využitím master/worker modelu. Master proces inicializuje celou síť a rozešle ji worker procesům, aby měly všechny procesy síť shodně inicializovanou. Podle počtu spuštěných procesů rozdělí master proces síť na intervaly a tyto rozešle worker procesům, čímž určí, nad kterou částí sítě bude který worker proces pracovat. Na počátku každé učební epochy jsou zamíchány vstupní vektory, aby se odstranila závislost sítě na pořadí předkládaných vstupních vektorů. V každém procesu však musí být v jednom okamžiku předkládán stejný vektor ze vstupní množiny, proto před započetím učení sítě rozešle master proces worker procesům inicializační hodnotu generátoru pseudonáhodných čísel a tato hodnota je pak používána pro míchaní pole se vstupními vektory ve všech procesech. Učební epocha dále probíhá následovně. Worker procesy pro jeden vstupní vektor naleznou kandidáta na vítězný neuron. To je neuron v rámci přidělené části sítě, jehož hodnota váhového vektoru je euklidovsky nejbližší předkládanému vektoru. Souřadnice v neuronové mřížce a vzdálenosti od vstupního vektoru kandidátů jsou odeslány master procesu, který porovnáním vzdáleností určí vítězný neuron. Z jeho souřadnic určí míry sousednosti pro každý neuron sítě. Na základě míry sousednosti, koeficientu učení a vstupního vektoru master proces adaptuje neurony sítě podle Kohonenova pravidla. Adaptovanou síť pak rozešle worker procesům. Pokračuje se další učební epochou.

4.4.2 Paralelní online algoritmus s rozdělením sítě a centralizovaným učením 2

V dalším textu budeme používat označení *POCLNPMW2*. Tento algoritmus učení je shodný s předchozím. Rozdíl je v implementaci neuronové sítě, která v tomto případě uchovává hodnoty váhových vektorů všech neuronů v jediném jednorozměrném poli. Objekt s neuronovou sítí je používán nejen pro uložení sítě, ale také pro přenos částí sítě mezi procesy při výpočtu. Experimentálně jsme ověřili, že přenos jednorozměrného pole mezi procesy je rychlejší než přenos třírozměrného pole se stejným počtem uložených prvků. Toto zjištění jsme zahrnuli do tohoto algoritmu, abychom zjistili jaký to bude mít dopad na dobu učení neuronové sítě.

4.4.3 Paralelní online algoritmus s rozdělením sítě a decentralizovaným učením - master/worker

V dalším textu budeme používat označení *PODLNPMW* (Parallel Online Decentralized Learning with Network Partitioning - Master/Worker). Toto je implementace algoritmu 5. Před započetím učení master proces rozešle worker procesům inicializační hodnotu pro generátor náhodných čísel, který je používán pro zamíchání vstupních vektorů. V jednotlivých iteracích musí worker procesy síti předkládat stejný vstupní vektor, proto musí být ve všech worker procesech vektory zamíchány shodným způsobem. Master proces inicializuje síť a celou ji pošle do všech worker procesů. Master proces rozdělí síť na oblasti o přibližně stejné velikosti a tyto oblasti přiřadí worker procesům. Worker procesy předkládají své části sítě vstupní vektory a nacházejí kandidáty na vítěze, které posílají do master procesu. Ten pak nalézá celkového vítěze a rozesílá ho worker procesům, ty pak podle něj aktualizují své kopie sítě. Takovéto učební epochy se opakují do předepsaného počtu.

4.4.4 Paralelní online algoritmus s rozdělením sítě a decentralizovaným učením

V dalším textu budeme používat označení *PODLNP* (Parallel Online Decentralized Learning with Network Partitioning). Jedná se o implementaci algoritmu 3. Tento algoritmus je myšlenkou podobný předchozímu, avšak nevyužívá master/worker model. Všechny procesy využívají stejný zdrojový program, který je sloučením funkcionalit předchozích master a worker procesů. Na začátku algoritmu nultý proces rozešle inicializovanou síť ostatním a taktéž základ pro generátor náhodných čísel. Každý proces si pak vypočte nad kterými řádky sítě bude pracovat a to podle počtu spuštěných procesů a vlastního ranku. Je zřejmé, že když v předchozím algoritmu worker procesy odeslaly kandidáty na vítěze do master procesu, musely čekat na výsledek, než mohly začít s prací na vyhledávání nového kandidáta pro nová data. V tomto algoritmu procesy rozesílají kandidáty prostřednictvím funkce allgather, která doručí množiny kandidátů do každého procesu. Procesy pak redundantně určí konečný vítězný neuron a podle něj každý proces aktualizuje svou část mapy.

4.4.5 Paralelní dávkový algoritmus s rozdělením dat a decentralizovaným učením

Dále budeme používat označení *PBDLDP* (Parallel Batch Decentralized Learning with Data Partitioning). Jde o implementaci algoritmu 8, který je založen na sekvenčním dávkovém algoritmu a rozdělení vstupních dat mezi procesy. Všechny procesy běží podle jednoho programu, který využívá kolektivní komunikaci. Mapa neuronů je obdélníková a její kopie je uchovávána v každém procesu. Každý proces zpracovává část vstupních dat. Která data bude proces zpracovávat si vypočte na základě počtu spuštěných procesů a svého ranku. Na počátku učební epochy jsou ve všech procesech inicializovány hodnoty čitatelů a jmenovatelů pro každý neuron na hodnotu 0. Následuje cyklus přes pole vstupních vektorů přidělených procesu. Pro každý vstupní vektor je nalezen vítězný neuron v síti a k čitatelům a jmenovatelům podle rovnice (5) jsou přičteny příslušné hod-

noty, vypočtené ze souřadnic vítězného neuronu a vstupního vektoru. Za tímto cyklem je volána funkce allreduce a to zvlášť pro čitatele a zvlášť pro jmenovatele. Tato funkce prostřednictvím vlastní redukční operace sečte čitatele a jmenovatele ze všech procesů a výsledné součty uloží do každého procesu. Na základě výsledných součtů je pak možné adaptovat neurony sítě. Dále se postup opakuje pro další učební epochy.

4.4.6 Paralelní dávkový algoritmus s rozdělením dat 2

Dále budeme používat označení *PBDLDP2*. Program je shodný s předchozím. Využívá pouze rozdílnou implementaci neuronové sítě, která ukládá hodnoty čitatelů a jmenovatelů společně pro všechny neurony do jednoho jednorozměrného pole, které je na konci každé epochy přenášeno mezi procesy. Tímto se pokoušíme zredukovat čas potřebný pro přenos čitatelů a jmenovatelů. Vycházíme z předpokladu, že jeden přenos, namísto přenosu dvou polí - čitatelů a jmenovatelů, je zatížen poloviční latencí a dále, že přenos jednorozměrného pole je v MPI.NET rychlejší než přenos dvourozměrného pole o stejném počtu prvků.

4.4.7 Paralelní dávkový algoritmus s rozdělením sítě a centralizovaným učením

Dále budeme používat označení *PBCLNPMW* (Parallel Batch Centralized Learning with Network Partitioning - Master/Worker). Je to implementace algoritmu 6. Jedná se o master/worker algoritmus dávkového učení, který na rozdíl od předchozích dvou nerozděluje mezi procesy vstupní data, ale neuronovou síť. Master náhodně inicializuje síť a rozešle ji worker procesům. Následně rovnoměrně rozdělí neuronovou síť mezi worker procesy a rozešle worker procesům indexy prvních a posledních řádků, které budou zpracovávat. Všechny procesy si udržují kopie vstupních dat. Učební epocha probíhá následovně. Na počátku epochy si worker procesy vypočtou koeficient učení α a šířku funkce okolí σ. Následně pro každý vstupní záznam worker procesy vyhledávají vítězný neuron v přidělené části sítě a nalezené kandidáty na vítěze posílají do master procesu. Ten z nich vybírá globálního vítěze a podle něj akumuluje čitatele a jmenovatele dle rovnice (5). Po tom, co jsou takto zpracovány všechny vstupní záznamy, master proces posílá worker procesům matice čitatelů a jmenovatelů. Worker procesy, na základě těchto matic, aktualizují své přidělené části neuronových sítí. Obdobným způsobem se učební epochy opakují až do předepsaného počtu. Aby mohla být výsledná síť uložena, jsou její části nakonec poslány do master procesu.

4.4.8 Paralelní dávkový algoritmus s rozdělením sítě 2

Dále budeme používat označení *PBCLNPMW2*. Tento algoritmus je shodný s předchozím. Využívá rozdílnou implementaci neuronové sítě, která ukládá hodnoty čitatelů a jmenovatelů společně pro všechny neurony do jednoho jednorozměrného pole, které je následně používáno v závěru epochy pro přenos mezi procesy. Tímto způsobem se opět pokoušíme snížit komunikační režii.

4.4.9 Paralelní dávkový algoritmus s rozdělením sítě a decentralizovaným učením

Dále budeme používat označení *PBDLNPMW* (Parallel Batch Decentralized Learning with Network Partitioning - Master/Worker). Toto je implementace algoritmu 7. Jedná se o master/worker algoritmus dávkového učení, který rozděluje mezi procesy neuronovou síť. Algoritmus pracuje následovně. Master vytvoří a náhodně inicializuje síť, tu pak rozešle worker procesům. Master rozdělí neuronovou síť na části, nad kterými budou pracovat worker procesy. Indexy počátků a konců těchto částí zašle worker procesům. Worker procesy si uchovávají všechna vstupní data. Na začátku učební epochy si worker procesy spočtou šířku funkce okolí σ a vynulují matice čitatelů a jmenovatelů dle rovnice (5). Worker procesy postupně předkládají vstupní data své přidělené části neuronové sítě a nacházejí souřadnice kandidáta na vítězný neuron. Kandidáta na vítěze zasílají do master procesu, který nalezne vítězný neuron a jeho souřadnice zašle zpět worker procesům. Ty pak akumulují čitatele a jmenovatele dle rovnice (5) v těch částech matic, které odpovídají přiděleným částem neuronové sítě. Po zpracování všech vstupních dat, worker procesy aktualizují své části neuronových sítí na základě matic čitatelů a jmenovatelů. Učební epochy se tímto způsobem znovu opakují až do požadovaného počtu. Na závěr worker procesy pošlou své části sítě do master procesu, aby je tento mohl uložit.

4.4.10 Paralelní dávkový hybridní algoritmus

Dále budeme používat označení *PBHLMW* (Parallel Batch Hybrid Learning - Master/Worker). Zde se jedná o implementaci algoritmu 9. Tento algoritmus je označován jako hybridní, neboť mezi procesy rozděluje jak síť, tak i vstupní data. Jedná se o master/worker algoritmus. Master nejprve rozdělí vstupní data na stejně velké části a ty poté rozdělí mezi worker procesy. Desetinu předepsaného počtu epoch pak algoritmus funguje jako paralelní dávkový algoritmus. Účelem této fáze je předuspořádat neuronovou síť. Zbylých devět desetin epoch funguje algoritmus jako hybridní. Na počátku hybridní fáze master proces spočte histogram, který obsahuje informaci o tom, který neuron je nejblíže jakému počtu vstupních vektorů. Na základě tohoto histogramu je pak pomocí rekurzivní bisekce rozdělena neuronová síť na obdélníkové oblasti tak, aby do těchto oblasti náleželo pokud možno obdobné množství vstupních vektorů. Oblastí je vytvořeno tolik, kolik je worker procesů. Souřadnice oblastí a indexy vstupních dat do těchto oblastí spadajících jsou zaslány worker procesům. Deset epoch pak worker procesy předkládají přiřazená vstupní data své části neuronové sítě a aktualizují ji pomocí dávkového algoritmu. Každou desátou epochu pak worker procesy spočtou nové histogramy ve své části sítě. Tyto výsledky odešlou do master procesu, který je sloučí do jednoho histogramu a na jeho základě znovu přerozdělí oblasti a indexy vstupních vektorů mezi worker procesy. Toto se opakuje až do předepsaného počtu epoch.

4.4.11 Paralelní dávkový hybridní algoritmus 2

Dále budeme používat označení *PBHLMW2*. Je to variace na předchozí algoritmus. Rozdíl spočívá v odlišné implementaci neuronové sítě. V tomto případě neuronová síť ukládá

hodnoty složek váhových vektorů v jednorozměrném poli a hodnoty čitatelů a jmenovatelů ve druhém jednorozměrném poli. Jedná se o experiment s cílem snížit náklady na meziprocesní komunikaci.

4.4.12 Paralelní dávkový hybridní algoritmus 3

Dále budeme používat označení *PBHLMW3*. Snaha o další zrychlení předchozího algoritmu. Implementace se ve velké míře shoduje s předchozí. Na rozdíl od ní, je zde použito pro přenos histogramu mezi procesy třírozměrné pole typu Integer, namísto dvourozměrného pole typu List.

Algoritmus	Učení	Mezi procesy se dělí	Jednou za iteraci se přenáší	Jednou za epochu se přenáší	Jednou za 10 epoch se přenáší
SOL	online				
SBL	dávkové				
SBL2	dávkové				
PBCLNPMW	dávkové	síť	Winner	část čitatelů a jmenovatelů (objekt)	
PBCLNPMW2	dávkové	síť	Winner	část čitatelů a jmenovatelů (pole)	
PBDLDP	dávkové	data		čitatelé a jmenovatelé (objekt)	
PBDLDP2	dávkové	data		čitatelé a jmenovatelé (pole)	
PBDLNPMW	dávkové	síť	Winner		
PBHLMW	dávkové	síť i data		síť (objekt), část čitatelů a jmenovatelů (objekt)	hranice oblastí, indexy dat, histogram (pole seznamů)
PBHLMW2	dávkové	síť i data		síť (objekt), část čitatelů a jmenovatelů (pole)	hranice oblastí, indexy dat, histogram (pole seznamů)
PBHLMW3	dávkové	síť i data		síť (objekt), část čitatelů a jmenovatelů (pole)	hranice oblastí, indexy dat, histogram (pole)
POCLNPMW	online	síť	Winner	část sítě (pole objektů)	
PODLNP	online	síť	Winner		
PODLNPMW	online	síť	Winner		

Tabulka 1: Přehled implementovaných algoritmů

5 Testy

V této kapitole se zaměřím na ověření funkčnosti implementovaných algoritmů a na porovnání jejích výkonů při různých podmínkách.

Testování bylo prováděno na školním Windows HPC serveru 2008. Tento server je složen ze 3 výpočetních uzlů, které jsou propojeny 1Gb linkou. Každý uzel obsahuje dva čtyř-jádrové procesory a 12GB paměti.

5.1 Ověření správné funkce algoritmů

Abychom dokázali, že implementované algoritmy dávají správné výsledky, nechali jsme každým algoritmem naučit tři sady dat. K výsledným neuronovým sítím jsme spočítali relativní U-matice, které jsme zobrazili jako čtvercové sítě, kde každý čtvereček odpovídá relativní U-výšce jednoho neuronu. Bílá barva čtverečku odpovídá hodnotě 0, černá barva hodnotě 1. Bílé čtverečky tedy označují neurony, které jsou podobné (jejichž váhové vektory jsou euklidovsky blízko) svým sousedům. Na vzniklých obrazcích jsou pak lidským okem rozeznatelné shluky, které byly nalezeny ve vstupních datech.

První sadu dat pro tento test tvořil uměle připravený soubor. Data byla vygenerována tak, aby obsahovala 10 nepřekrývajících se shluků. Každý shluk obsahoval 100 záznamů, které byly tvořeny 3-rozměrnými vektory. Všechny záznamy spadaly do jednotkové krychle.

Výroba dat probíhala následovně. Nejdříve byly definovány konkrétní středy shluků a to na pozicích [0,2; 0,2; 0,2], [0,2; 0,7; 0,2], [0,7; 0,2; 0,2], [0,7; 0,7; 0,2], [0,2; 0,2; 0,7], [0,2; 0,7; 0,7], [0,7; 0,2; 0,7], [0,7; 0,7; 0,7], [0,45; 0,45; 0,45], [0,45; 0,45; 0,9]. Z každého středu pak byly vyráběny vektory tak, že k hodnotě každé souřadnice středu bylo přičteno náhodné číslo z intervalu <-0,001; 0,001>. Z každého středu bylo takto vyrobeno 100 vektorů.

Druhou a třetí sadu dat tvořila data převzatá z projektu Databionic ESOM Tools [11].

Druhá sada obsahovala 1000 3-rozměrných záznamů. Polovina záznamů tvořila jednu kružnici a druhá polovina záznamu druhou kružnici. Roviny kružnic svíraly úhel 90° a kružnice byly v prostoru položené tak, že tvořily dva články řetězu. Viz. obrázek 2a.

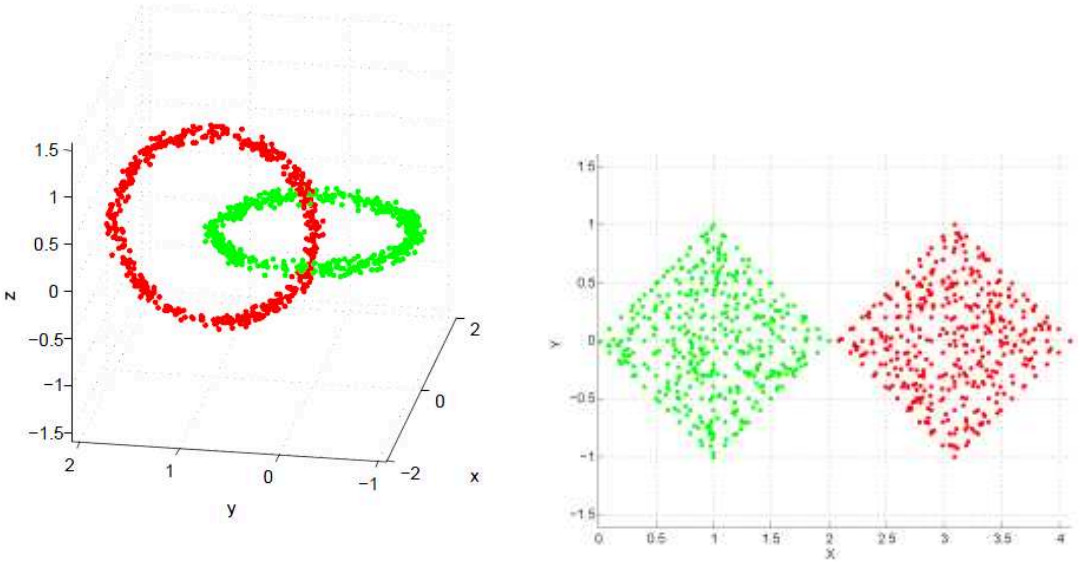
Třetí sadu dat tvořilo 800 2-rozměrných záznamů. Záznamy tvořily dva čtvercové shluky, které se dotýkaly v jednom vrcholu. Viz. obrázek 2b.

Učení probíhalo 1000 epoch a bylo spuštěno na 4 výpočetních jádrech.

Výsledné relativní U-matice vidíme na na obrázku 3. Rozpoznáme zde všech 10 shluků ze vstupních dat. Na obrázcích hybridních algoritmů (Obrázky 3f, 3g, 3h) jsou patrné rovné hranice mezi shluky, které ukazují, že tyto algoritmy nepracují shodně s ostatními algoritmy. Tyto hranice odpovídají hranicím dělení sítě mezi procesy.

U-matice druhé sady dat jsou na obrázku 4. Obrázek 4a je referenční a pochází z [11]. Můžeme vidět, že námi vytvořené U-matice odpovídají referenční. Výjimku tvoří opět hybridní algoritmy (obrázky 4g, 4h, 4i), které v tomto případě naprosto selhaly, čímž se ukázalo, že pro složitější data jsou nepoužitelné.

Třetí sadu dat zobrazuje obrázek 5. Obrázek 5a je opět referenční a pochází z [11]. U-matice většiny algoritmu odpovídají referenční U-matici. I zde se projevila odlišnost



(a) Chainlink (b) TwoDiamonds

Obrázek 2: Zobrazení druhé a třetí sady dat.

hybridní implementace. Na obrázcích 5g, 5h, 5i je patrné, že vstupní data obsahují dva shluky, ale již z nich nevyplývá, že se tyto dva shluky dotýkají.

Těmito testy bylo dokázáno, že algoritmy s výjimkou hybridních fungují podle očekávání. Hybridní algoritmy fungovaly jen s jednoduššími daty a i v těchto případech podávaly zkreslené výsledky. Se složitými daty byly hybridní algoritmy nepoužitelné.

5.2 Testy rychlosti algoritmů

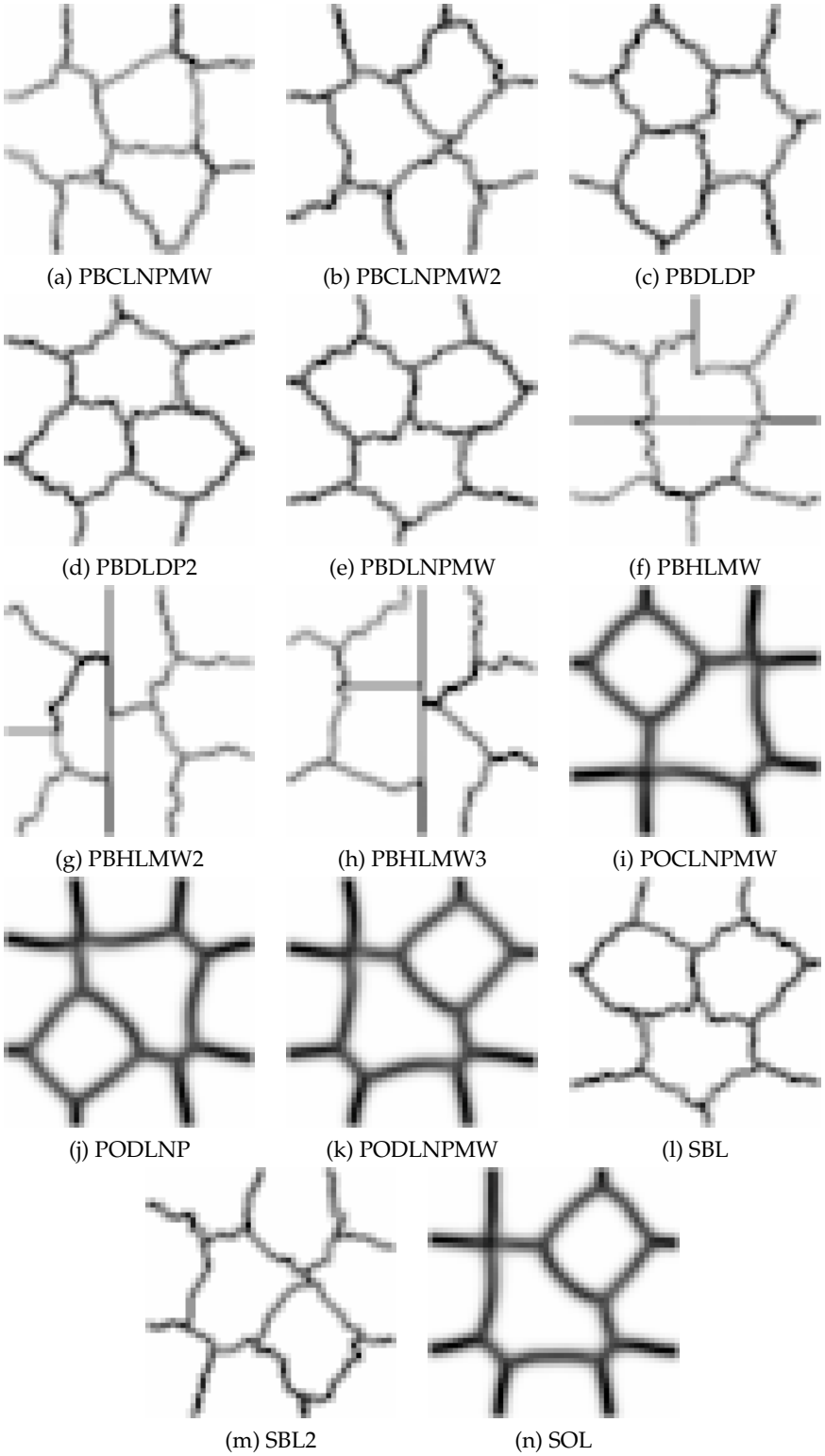
Pro otestování rychlosti výpočtu jsme náhodně vygenerovali tři sady vstupních dat s rozdílným počtem záznamů a s rozdílnou velikostí dimenze vstupního vektoru:

- 100 záznamů s dimenzí 30
- 10000 záznamů s dimenzí 5
- 10000 záznamů s dimenzí 30

Byly použity dvě velikosti SOM:

- 6×6 neuronů
- 50×50 neuronů

Měření bylo provedeno pro všechny kombinace výše popsaných algoritmů, sad vstupních dat a velikostí SOM. Každý paralelní algoritmus byl spuštěn na 2 až 16 jádrech. Na tomto místě je nutné si uvědomit, že pokud byl program spuštěn na 8 nebo méně jádrech, pak běžel v rámci jednoho stroje. Pokud byl spuštěn na 9 nebo více jádrech, pak běžel na



(a) PBCLNPMW (b) PBCLNPMW2 (c) PBDLDP

(d) PBDLDP2 (e) PBDLNPMW (f) PBHLMW

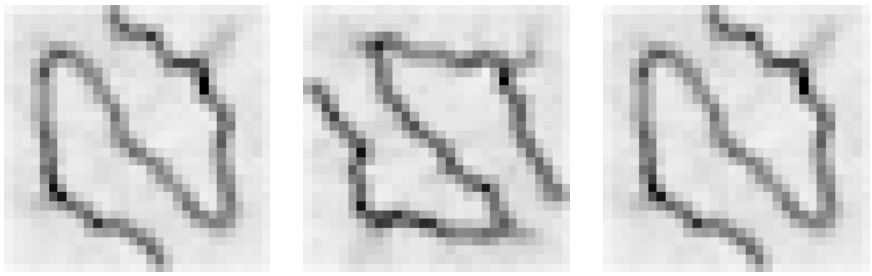
(g) PBHLMW2 (h) PBHLMW3 (i) POCLNPMW

(j) PODLNP (k) PODLNPMW (l) SBL

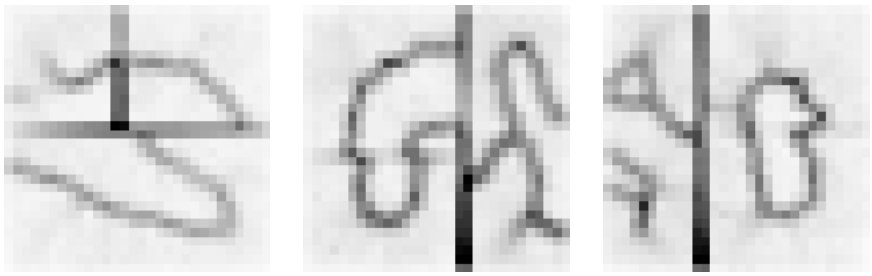
(m) SBL2 (n) SOL

Obrázek 3: Vizualizace U-matic neuronových sítí - oddělené shluky

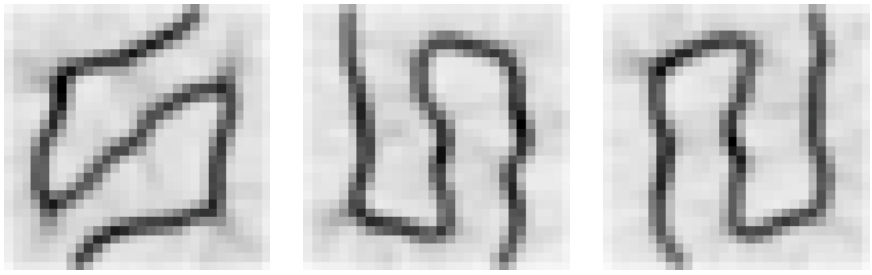
(a) U-matice (b) PBCLNPMW (c) PBCLNPMW2



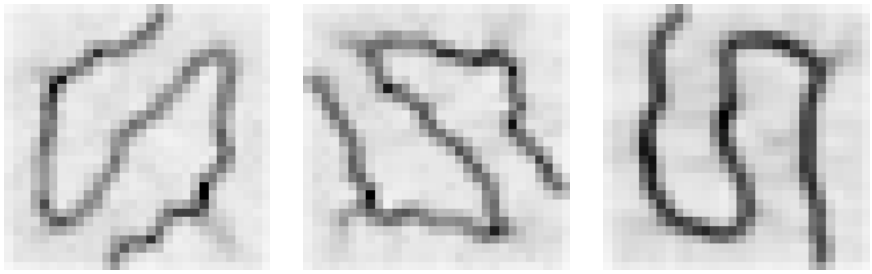
(d) PBDLDP (e) PBDLDP2 (f) PBDLNPMW



(g) PBHLMW (h) PBHLMW2 (i) PBHLMW3

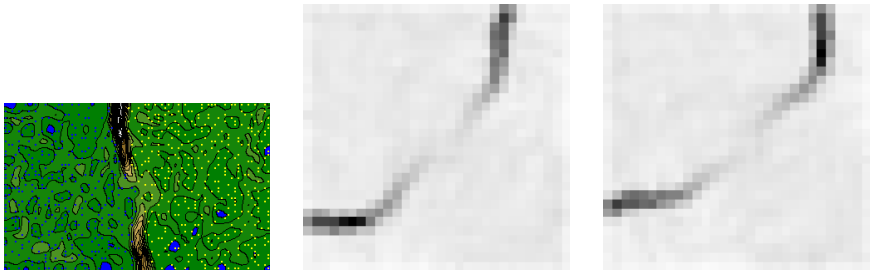


(j) POCLNPMW (k) PODLNP (l) PODLNPMW



(m) SBL (n) SBL2 (o) SOL

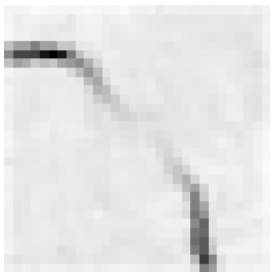
Obrázek 4: Vizualizace U-matic neuronových sítí - Chainlink



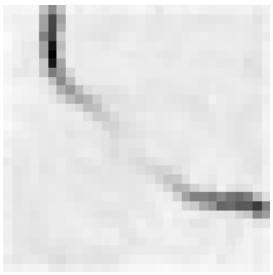
(a) U-matice

(b) PBCLNPMW

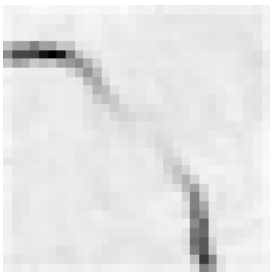
(c) PBCLNPMW2



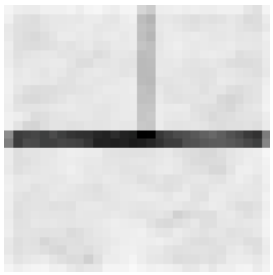
(d) PBDLDP



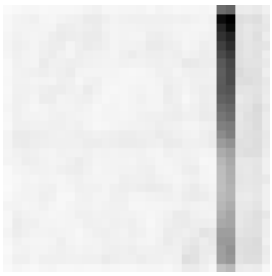
(e) PBDLDP2



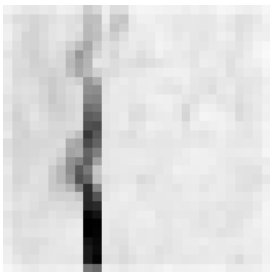
(f) PBDLNPMW



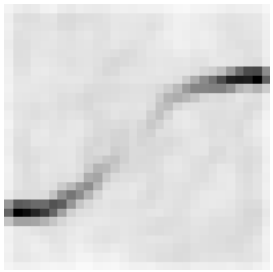
(g) PBHLMW



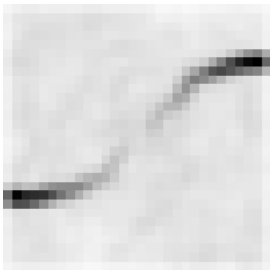
(h) PBHLMW2



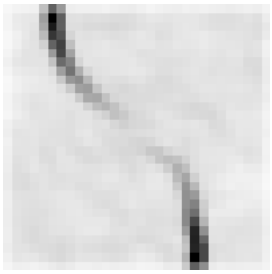
(i) PBHLMW3



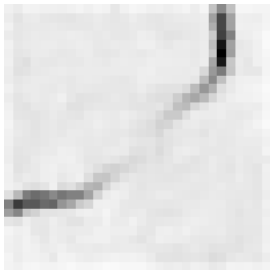
(j) POCLNPMW



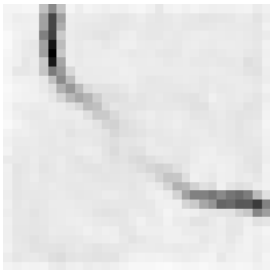
(k) PODLNP



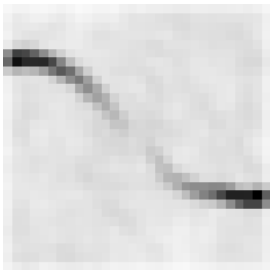
(l) PODLNPMW



(m) SBL



(n) SBL2



(o) SOL

Obrázek 5: Vizualizace U-matic neuronových sítí - Two Diamonds

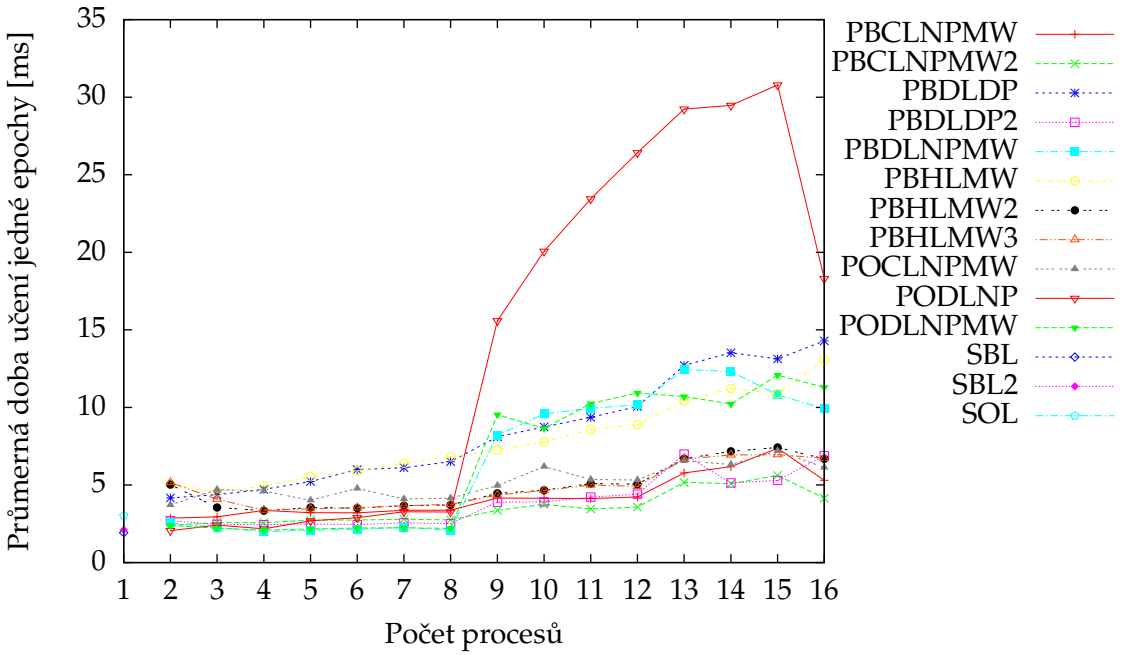
více výpočetních uzlech a bylo nutné komunikovat po síti, což komunikaci zpomalilo. Programy byly spuštěny samostatně tak, aby nesoupeřily o prostředky stroje (sběrnice, paměť) s jinými programy. Do měření byl započítán pouze čas učení. Nebylo tedy započítáno načítání dat, inicializace mapy, ukládání mapy, apod.. U většiny algoritmů bylo provedeno 10 epoch učení. U hybridních algoritmů bylo provedeno 100 epoch učení. Naměřené časy byly vyděleny počtem učebních epoch, čímž vznikly průměrné časy na jednu epochu. Tyto výsledky měření jsou znázorněny v grafech na obrázcích 6–11 a odpovídajících tabulkách 2–7.

Z grafu 6 a tabulky 2 vidíme, že malou síť o velikosti 6×6 neuronů nemá cenu učit 100 záznamy o dimenzi 30, neboť to jsou stále malá data na to, abychom při výpočtu profitovali z paralelizace. Nejlepšího času bylo dosaženo se sekvenčním dávkovým algoritmem SBL. Všechny paralelní algoritmy měly v tomto případě horší časy, neboť z důvodu malé velikosti SOM a malého objemu vstupních dat u nich převládla komunikační režie nad časem stráveným výpočty.

Při testu s neuronovou sítí o velikosti 50×50 neuronů učenou 100 záznamy o dimenzi 30 již některé paralelní algoritmy dosáhly zrychlení výpočtu. Lepších časů než nejlepší sekvenční algoritmus SOL dosáhly algoritmy PODLNP, PODLNPMW, PBDLNPMW. Tyto algoritmy dělí mezi procesy neuronovou síť. Jelikož v tomto testu bylo použito málo vstupních záznamů, dalo se očekávat, že v něm neuspějí algoritmy s dělením dat, což se také potvrdilo.

Další test používal malou síť 6×6 neuronů a učil jí daty o 10000 záznamech s dimenzí 5. Dostatek dat a nedostatek neuronů by měl zvýhodňovat algoritmy s rozdělením dat a znevýhodňovat algoritmy s rozdělením sítě, což se potvrdilo. Paralelní algoritmy s rozdělením sítě dosahovaly horších časů než sekvenční algoritmy. V tomto testu se osvědčily algoritmy PBDLDP a PBDLDP2. S rostoucím počtem procesů se zrychlovalo učení. To platilo až do 12 procesů. Od 13 do 16 procesů se již učení nezrychlovalo. Algoritmus PBDLDP2, který využívá pro přenos pole, byl rychlejší než algoritmus PBDLDP, využívající pro přenos objekty. Rychlejší než sekvenční algoritmy byly také paralelní hybridní algoritmy. Jejich naměřené časy však neškálovaly s rostoucím počtem procesů.

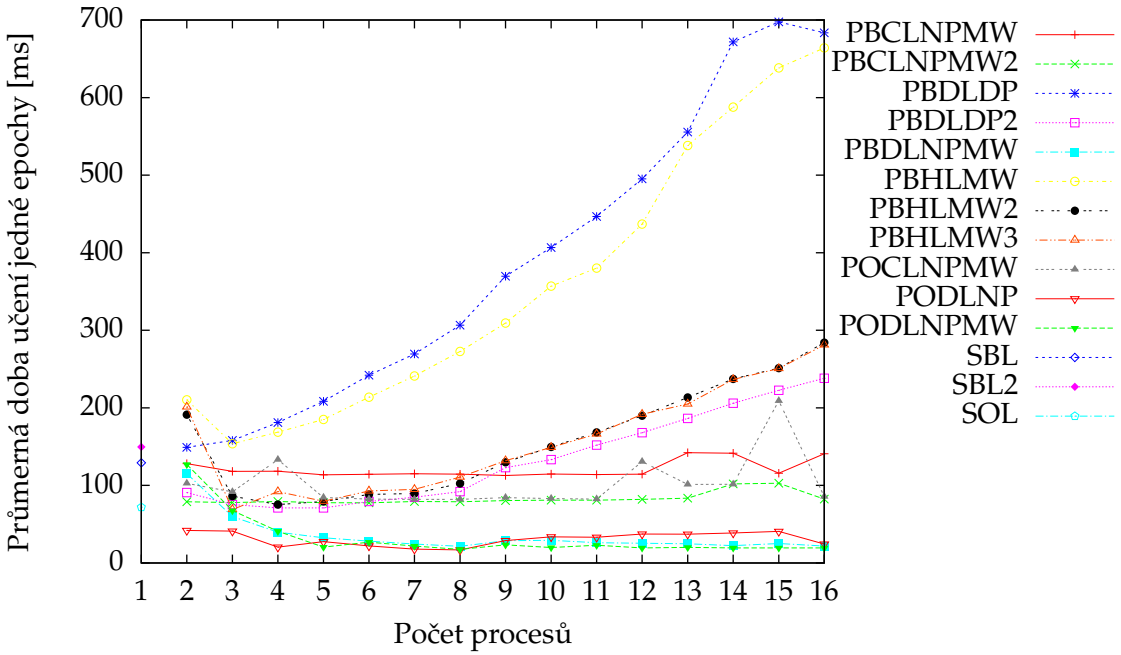
Při použití neuronové sítě 50×50 neuronů jsou již obdobné výsledky jak pro 10000 záznamů s dimenzí 5, tak pro 10000 záznamů s dimenzí 30. V obou případech již máme dostatek prostoru pro algoritmy s dělením sítě i dělením dat. Nejlépe pracují algoritmy PBDLDP a PBDLDP2. Čas výpočtu se snižuje se zvyšujícím se počtem spuštěných procesů. Při spuštění 9 procesů trvá učení déle než při použití 8 procesů, což je způsobeno nutnou komunikací po síti. Opět se projevuje, že přenášení polí mezi procesy je rychlejší než přenášení objektů. Algoritmus PBDLDP2 proto dosahuje lepších časů než algoritmus PBDLDP. PBDLDP2 také lépe škáluje při spuštění 9 až 16 procesů a při 16 spuštěných procesech dosahuje nejkratšího času učení. Algoritmy PBDLNPMW, PODLNP, PODLNPMW jsou také použitelné, avšak dosahovaly o něco horších časů. Čas výpočtu se u těchto algoritmů snižoval od 2 do 8 spuštěných procesů. U 9 a více spuštěných procesů, kdy se začala využívat pomalá komunikace po síti, se čas výpočtu značně prodloužil, což se dalo ostatně očekávat. Komunikace po síti zvýšila latenci a na tu jsou tyto algoritmy citlivé, protože přenášejí v každé iteraci souřadnice vítězného neuronu. Hybridní algo-



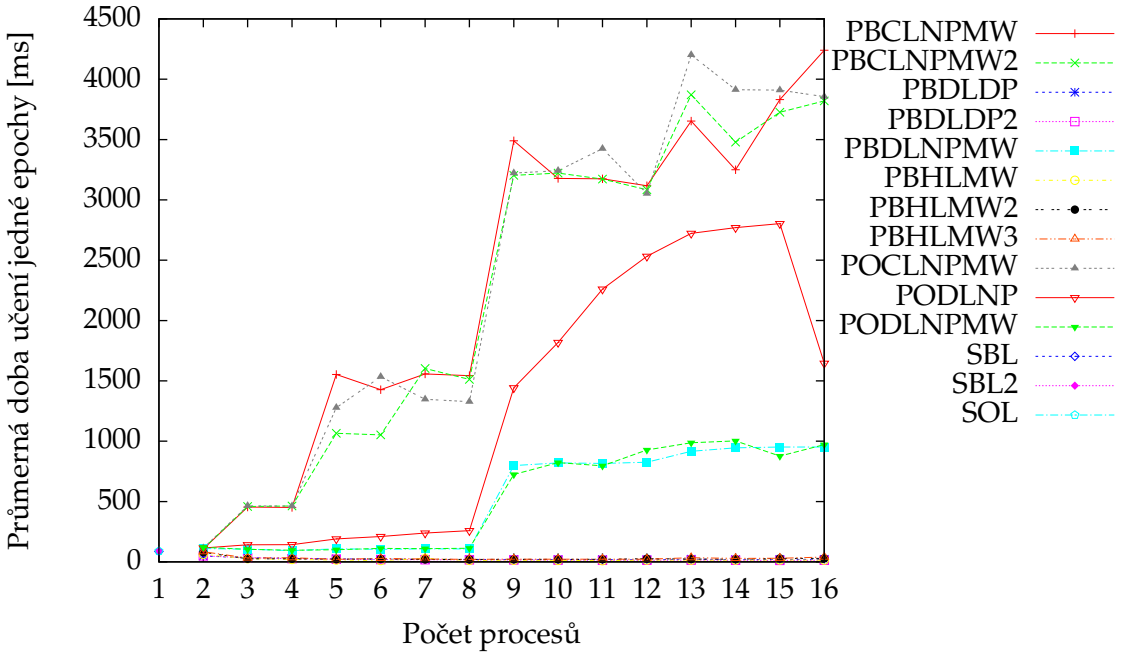
Obrázek 6: Závislost času výpočtu na počtu procesorů. (Data 30×100; síť 6×6)

ritmy dosahovaly lepších časů než algoritmy sekvenční, avšak čas výpočtu u nich neškáloval s počtem využitých procesorů. Naměřené časy neopisují žádnou smysluplnou funkci a jeví se nahodilé. To je pravděpodobně způsobeno neoptimálním rozdělováním neuronové sítě a vstupních dat mezi procesy v rámci synchronizačních epoch. Použití rekurzivní bisekce pro rozdělování sítě se tak v praxi neosvědčilo.

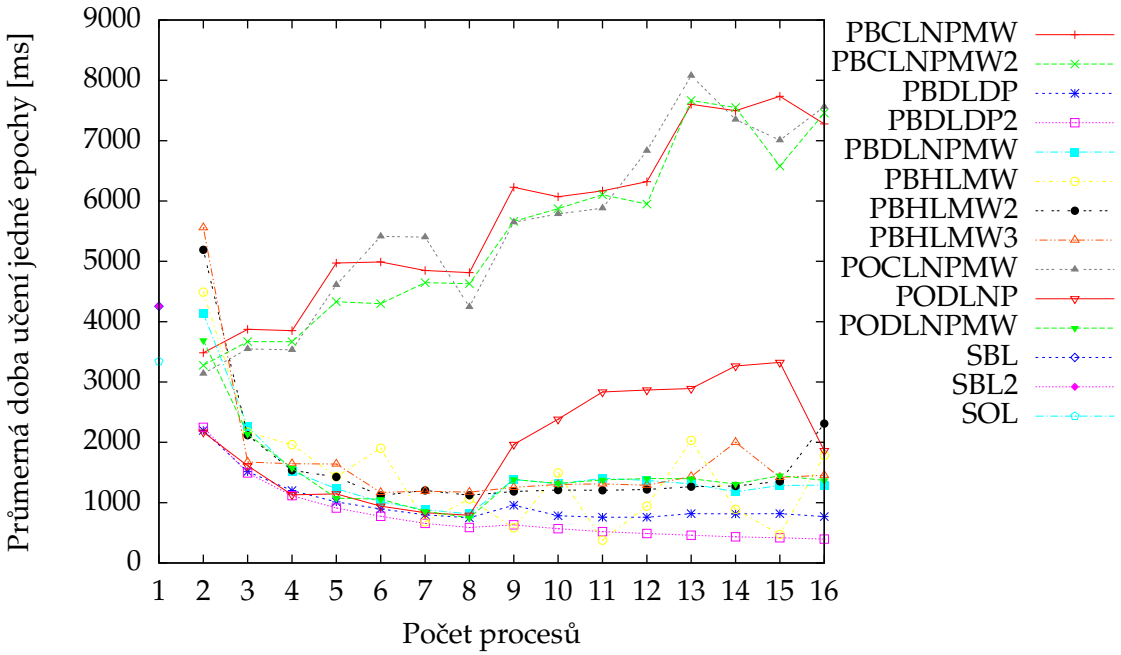
Ve všech testech se ukázalo, že algoritmy s centralizovaným učením (PBCLNPMW, PBCLNPMW2, POCLNPMW) nejsou v praxi příliš použitelné. Při použití dvou procesů byly tyto algoritmy rychlejší než algoritmy sekvenční, avšak s rostoucím počtem procesů rostl jejich čas výpočtu. Tento výsledek se dal očekávat, neboť master proces, který zabezpečoval centralizované učení, musel mít evidentně víc práce než ostatní worker procesy. S rostoucím počtem použitých procesů zde znatelně přibývá komunikační režie, avšak výhoda z paralelního vyhledávání BMU je jen minimální.



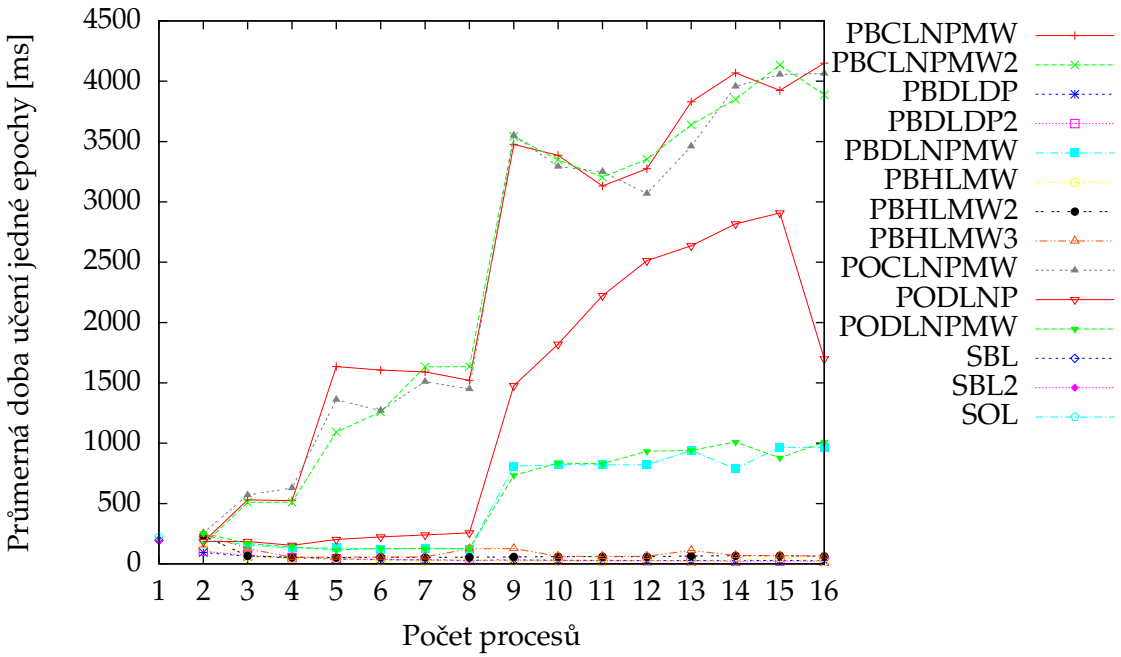
Obrázek 7: Závislost času výpočtu na počtu procesorů. (Data 30×100; síť 50×50)



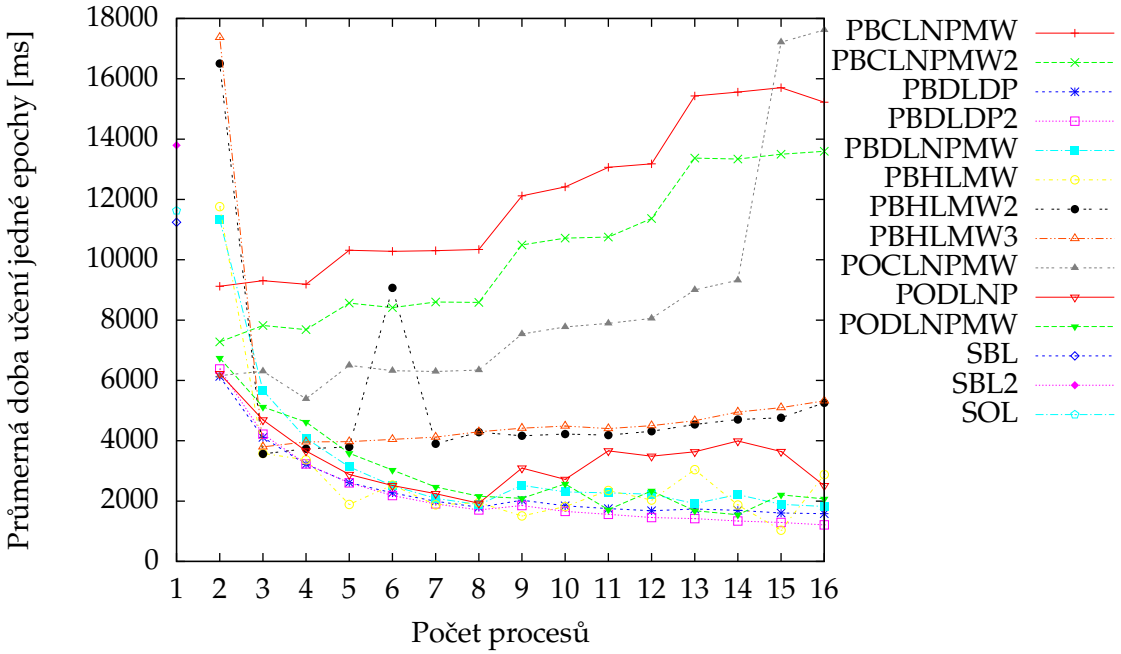
Obrázek 8: Závislost času výpočtu na počtu procesorů. (Data 5×10000; síť 6×6)



Obrázek 9: Závislost času výpočtu na počtu procesorů. (Data 5×10000; síť 50×50)



Obrázek 10: Závislost času výpočtu na počtu procesorů. (Data 30×10000; síť 6×6)



Obrázek 11: Závislost času výpočtu na počtu procesorů. (Data 30×10000; síť 50×50)

6 Závěr

Cílem práce bylo zmapovat možnosti paralelizace neuronových sítí, konkrétně Kohonenových samoorganizovaných map. Byl zde probrán úvod do problematiky neuronových sítí. Podrobně byly popsány samoorganizované mapy a sekvenční i paralelní algoritmy k jejich učení. Tyto algoritmy byly implementovány a otestovány.

Pro implementaci paralelních algoritmů bylo zvoleno prostředí MPI.NET, které je implementací standardu Message Passing Interface použitelnou ve frameworku Microsoft .NET. Během testování se ukázalo, že přenosová rychlost jednotlivých funkcí MPI.NET je závislá na přenášeném typu. MPI.NET sice umožňuje pohodlné přenášení objektů, ale pokud se snažíme dosáhnout maximální rychlosti programů, pak musíme pro přenos používat struktury a pole, čímž vlastně přicházíme o výhody tohoto frameworku. MPI.NET bychom doporučili jako rychlý prototypovací nástroj pro implementaci paralelních algoritmů, ale pro reálné nasazení bychom v budoucnu dali přednost Microsoft MPI.

Implementované algoritmy byly otestovány jak z hlediska správné funkce, tak z hlediska rychlosti. Testování probíhalo na Microsoft Windows HPC serveru 2008. Správná funkce algoritmů byla ověřena výpočtem U-matic a jejich porovnáním s referenčními U-maticemi z projektu Databionic ESOM Tools. Paralelní implementace byly otestovány spuštěním na 2 až 16 procesorech za použití různých velikostí neuronových sítí a různých objemů vstupních dat. Takto naměřené časy jsou zde prezentovány ve formě tabulek a grafů.

Dle očekávání špatně dopadly výsledky paralelních algoritmů s centralizovaným učením. Potvrdilo se, že tyto algoritmy není vhodné implementovat v prostředí předávání zpráv. Jako nejpoužitelnější se ukázaly paralelní algoritmy s dávkovým učením a rozdělením dat, které dosahovaly nejlepších časů výpočtů a dobře škálovaly s počtem procesorů. Obstojně pracovaly i algoritmy s rozdělením sítě a to jak s dávkovým učením, tak i s online učením. Bohužel jen maximálně na 8 procesorech, které byly součástí jednoho výpočetního uzlu. Pokud byly tyto algoritmy spuštěny na více procesorech a muselo tak dojít k síťové komunikaci mezi uzly, začal se u nich prodlužovat čas výpočtu. Toto chování bylo také předvídatelné, neboť tyto algoritmy používají velmi častou komunikaci. Zklamáním byly výsledky hybridních algoritmů. U-matice vzešlé z hybridních algoritmů neodpovídaly U-maticím ze sekvenčních algoritmů. Navíc u složitějších dat nebylo možné na těchto U-maticích rozeznat shluky v datech obsažené. Časy výpočtu u těchto algoritmů byly sice menší než u sekvenčních variant, ale neškálovaly s počtem procesorů.

Při případných dalších experimentech by bylo vhodné otestovat algoritmy na větších neuronových sítích a objemnějších datech. A to zejména ty algoritmy, které úspěšně pracovaly pouze do 8 procesů. Na větší úloze by se mohlo projevit zrychlení algoritmů i navzdory pomalejší komunikaci po síti mezi výpočetními uzly.

7 Reference

[1] VONDRÁK, Vít. *Umělá inteligence a neuronové sítě.* 2. vyd. Ostrava : VŠB - Technická univerzita Ostrava, 2002. 139 s. ISBN 80-7078-949-2.

[2] KOHONEN, Teuvo. *Self-Organizing Maps.* 3rd ed. New York : Springer-Verlag, 2001. xx, 501 s. ISBN 3-540-67921-9.

[3] ULTSCH, Alfred. Maps for the Visualization of high-dimensional Data Spaces. In *Proc. 4th Int. Workshop Self-Organizing Maps.* Kyushu : [s.n.], 2003. s. 225-230.

[4] *MPI: A Message-Passing Interface Standard : Version 2.2* [online]. [s.l.] : Message Passing Interface Forum, 4.9.2009 [cit. 2011-03-07]. Dostupné z WWW: <http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf>.

[5] *MPI.NET : High-Performance C# Library for Message Passing* [online]. c2004-2011, last modified: 6-Oct-2008 [cit. 2011-03-07]. Dostupné z WWW: <http://osl.iu.edu/research/mpi.net/>.

[6] GREGOR, Douglas; LUMSDAINE, Andrew. Design and implementation of a high-performance MPI for C# and the common language infrastructure. In . *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming.* New York, NY, USA : ACM, 2008. s. 133-142. Dostupné z WWW: <http://doi.acm.org/10.1145/1345206.1345228>. ISBN 978-1-59593-795-7.

[7] SILVA, Bruno; MARQUES, Nuno. A Hybrid Parallel SOM Algorithm for Large Maps in Data-Mining. In [online]. [s.l.] : [s.n.], [2007] [cit. 2011-03-06]. Dostupné z WWW: <http://ssdi.di.fct.unl.pt/~nmm/MyPapers/SM2007.pdf>.

[8] LAWRENCE, R. D.; ALMASI, G. S.; RUSHMEIER, H. E. A Scalable Parallel Algorithm for Self-Organizing Maps with Applicationsto Sparse Data Mining Problems. In *Data Mining and Knowledge Discovery*. Hingham, MA, USA : Kluwer Academic Publishers, 1999. s. 171-195. ISSN 1384-5810.

[9] LUENGO, F.; COFINO, A. S.; GUTIÉRREZ, J. M. GRID oriented implementation of Self-Organizing Mpas for Data Mining in Meteorology. In *Grid Computing*. [s.l.] : Springer-Verlag, 2004. s. 163-170. Dostupné z WWW: <http://www.meteo.unican.es/files/pdfs/2004_Luengo_LNCS.pdf>. ISBN 978-3-540-21048-1.

[10] YANG, Ming-Hsuan; AHUJA, Narendra. A data partition method for parallel self-organizing map. *In International Joint Conference on Neural Networks, 1999.*. [s.l.] : Institute of Electrical & Electronics Enginee, 2001. s. 1929-1933. ISBN 0-7803-5529-6.

[11] *Databionic ESOM Tools* [online]. c2005-2006 [cit. 2011-05-01]. Dostupné z WWW: <http://databionic-esom.sourceforge.net/index.html>.

A Naměřené časy

Algoritmus	**Doba učení jedné epochy v ms v závislosti na počtu procesorů**															
Procesů	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
PBCLNPMW		2.865	2.95	3.363	3.211	3.203	3.373	3.375	4.164	4.145	4.125	4.232	5.774	6.18	7.368	5.308
PBCLNPMW2		2.4	2.575	2.569	2.731	2.738	2.795	2.751	3.374	3.753	3.46	3.578	5.164	5.087	5.611	4.142
PBDLDP		4.167	4.376	4.724	5.228	5.992	6.109	6.507	8.106	8.746	9.367	10.06	12.71	13.52	13.13	14.29
PBDLDP2		2.702	2.471	2.438	2.484	2.449	2.561	2.509	3.898	3.912	4.225	4.413	6.992	5.131	5.295	6.886
PBDLNPMW		2.553	2.274	1.999	2.089	2.161	2.259	2.099	8.201	9.583	9.951	10.17	12.45	12.31	10.82	9.916
PBHLMW		5.045	4.55	4.935	5.536	5.917	6.398	6.807	7.274	7.774	8.59	8.892	10.4	11.22	10.88	13.05
PBHLMW2		5.026	3.555	3.348	3.547	3.511	3.667	3.714	4.459	4.658	5.092	5.051	6.686	7.17	7.419	6.69
PBHLMW3		5.156	4.086	3.381	3.462	3.536	3.661	3.755	4.353	4.646	5.003	4.906	6.671	6.937	6.954	6.72
POCLNPMW		3.712	4.7	4.608	4.014	4.781	4.101	4.15	4.957	6.19	5.358	5.319	6.558	6.322	7.208	6.133
PODLNP		2.051	2.415	2.172	2.691	2.89	3.273	3.257	15.59	20.07	23.44	26.42	29.23	29.47	30.79	18.3
PODLNPMW		2.399	2.219	2.093	2.169	2.224	2.241	2.201	9.537	8.649	10.25	10.93	10.7	10.22	12.08	11.29
SBL	1.922															
SBL2	2.106															
SOL	3.023															

Tabulka 2: Závislost doby výpočtu na počtu procesorů. (100 záznamů o dimenzi 30, síť 6×6 neuronů)

Algoritmus	**Doba učení jedné epochy v ms v závislosti na počtu procesorů**															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
PBCLNPMW		128.1	118.2	118.2	113.7	114.4	115.1	114.5	112.8	114.6	113.9	114.5	142.2	141.5	115.7	140.9
PBCLNPMW2		78.91	77.86	79.29	77.78	77.62	79.29	78.94	80.49	81.03	81.02	82.09	83.62	102	102.8	82.45
PBDLDP		149	158.1	181.1	208.4	242.1	269.4	306.6	369.8	406.7	446.7	495.2	555.6	671.8	697.6	683.5
PBDLDP2		90.87	75.39	71	71.06	79.28	84.78	91.92	123	133.3	152	168.1	186.4	206	222.6	238.1
PBDLNPMW		115.3	59.76	39.69	32.54	28.17	24.5	21.43	28.64	29.25	26.11	25.41	24.8	22.42	25.06	21.76
PBHLMW		210.3	153.8	168.7	185	213.7	241.1	272.8	309.3	357	380.2	436.9	538.3	587.6	638.3	664.1
PBHLMW2		191.2	86.11	75.29	79.34	88.36	89.63	102.4	129.8	149.6	168.3	189.9	213.4	237.4	251.2	284
PBHLMW3		201.1	69.03	91.97	79.53	92.89	94.97	110.8	131.9	149.2	166.4	191.8	204.9	236.6	250.7	281.2
POCLNPMW		102.6	91.73	133.2	84.34	82.42	81.97	81.99	84.33	83.04	82.44	130.6	101.3	101.4	209.1	85.62
PODLNP		41.89	41.13	20.51	27.52	22.09	17.86	16.89	28.9	33.67	33.25	37.37	37.06	38.73	40.78	24.4
PODLNPMW		127	67.62	40.89	20.45	26.89	21.92	17.49	23.41	19.87	22.78	19.45	20.03	19.41	19.35	19.47
SBL	129.1															
SBL2	149.6															
SOL	71.84															

Tabulka 3: Závislost doby výpočtu na počtu procesorů. (100 záznamů o dimenzi 30, síť 50×50 neuronů)

Algoritmus	**Doba učení jedné epochy v ms v závislosti na počtu procesorů**															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
PBCLNPMW		107	453.9	450.5	1553	1427	1558	1543	3489	3178	3174	3117	3653	3250	3831	4241
PBCLNPMW2		110.4	458.7	462.7	1066	1053	1603	1512	3203	3223	3172	3084	3871	3479	3726	3820
PBDLDP		48.67	33.98	28.3	23.83	21.71	20.79	20.26	19.6	18.77	18.41	18.14	21.77	19.08	19.56	20.73
PBDLDP2		49.5	33.78	26.24	21.87	19.08	17.42	16.5	15.34	14.69	13.38	13.04	13.9	13.43	11.89	13.62
PBDLNPMW		110	104.9	92.51	108.5	103.9	108.5	107.2	797.5	820	815.5	826.2	916.3	945.2	951.4	951.6
PBHLMW		62.38	26.51	19.81	15.81	10.71	27.97	12.29	13.53	12.56	10.43	14.04	14.3	14.2	14.96	14.45
PBHLMW2		79.24	25.1	25	22.1	24.66	20	19.74	20.54	21.09	20.89	26.29	23.23	22.41	29.27	28.4
PBHLMW3		84.39	31.86	33.82	20.49	28.92	23.67	20.84	21.63	22.04	22.12	22.66	33.95	28.21	29.92	39.68
POCLNPMW		113.9	464.1	462.1	1277	1534	1347	1328	3222	3242	3425	3052	4201	3913	3910	3855
PODLNP		115.4	141.7	142.8	190.3	210.4	238.3	259.2	1443	1819	2261	2533	2723	2770	2804	1646
PODLNPMW		121	103.6	96.48	100.9	111.7	109.1	113.1	724.6	822.2	796.3	926.9	987.2	1002	877.1	971.8
SBL	86.69															
SBL2	91.32															
SOL	89.75															

Tabulka 4: Závislost doby výpočtu na počtu procesorů. (10000 záznamů o dimenzi 5, síť 6×6 neuronů)

Algoritmus	**Doba učení jedné epochy v ms v závislosti na počtu procesorů**															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
PBCLNPMW		3484	3874	3851	4972	4990	4848	4813	6228	6072	6168	6319	7602	7497	7736	7278
PBCLNPMW2		3277	3671	3669	4331	4298	4648	4629	5662	5874	6103	5949	7666	7549	6580	7459
PBDLDP		2198	1515	1199	1012	895.7	798.7	750	957.7	782.3	757.7	759	817	813.1	818.8	768.7
PBDLDP2		2247	1493	1117	910.2	774.8	657.1	588.7	631.7	567.5	520	487.5	460.7	433.5	417.4	395.4
PBDLNPMW		4142	2263	1523	1232	1012	884.8	813.5	1384	1314	1404	1369	1311	1180	1285	1292
PBHLMW		4491	2168	1962	1431	1902	680.3	1061	583.9	1492	378.5	942	2031	889.6	472.8	1782
PBHLMW2		5190	2118	1537	1426	1119	1202	1124	1185	1209	1205	1219	1265	1275	1353	2310
PBHLMW3		5558	1674	1644	1640	1170	1188	1174	1254	1302	1311	1289	1432	2000	1419	1454
POCLNPMW		3140	3548	3533	4609	5415	5401	4245	5648	5785	5881	6833	8078	7350	7009	7563
PODLNP		2169	1611	1130	1146	940.9	833.1	792.1	1964	2382	2834	2868	2892	3265	3326	1860
PODLNPMW		3690	2138	1575	1075	1056	856.9	750.5	1383	1318	1377	1399	1397	1308	1448	1368
SBL	4256															
SBL2	4257															
SOL	3343															

Tabulka 5: Závislost doby výpočtu na počtu procesorů. (10000 záznamů o dimenzi 5, síť 50×50 neuronů)

Algoritmus	**Doba učení jedné epochy v ms v závislosti na počtu procesorů**															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
PBCLNPMW		181.1	531.1	524.4	1635	1606	1590	1520	3476	3386	3131	3275	3828	4068	3925	4150
PBCLNPMW2		157.5	509.8	511.8	1092	1259	1633	1635	3543	3348	3205	3354	3639	3850	4135	3887
PBDLDP		93.68	62.94	49	40.29	34.74	31.6	29.34	32.39	30.13	28.56	27.31	27.72	26.21	28.47	26.84
PBDLDP2		104.9	70.03	53.44	43.53	36.67	31.87	28.66	31.31	28.71	26.32	25.37	24.81	22.64	22.96	22.48
PBDLNPMW		209.5	155.2	131.4	131.7	122	128	122.6	809	822.5	820.1	821.1	938.6	789.5	966.7	965.1
PBHLMW		160	57.15	44.91	46.07	23.98	21.79	40.3	22.48	22.47	15.6	24.2	19.46	24.89	51.5	19.11
PBHLMW2		233.7	66.35	53.89	52.47	54.06	54	55.67	57.86	59.57	59.67	59.77	65.17	67.17	67.91	63.37
PBHLMW3		247.7	126.2	55.16	56.86	56.15	56.93	124.7	127.2	62.83	62.14	62.93	111.3	69.11	67.25	65.67
POCLNPMW		255.6	570.5	628.9	1361	1270	1510	1449	3550	3293	3250	3069	3461	3957	4056	4065
PODLNP		184.2	186.1	154	202.7	223.6	239.9	256.9	1477	1821	2223	2513	2636	2817	2908	1699
PODLNPMW		254.3	169.4	139.3	116.2	128.6	125.9	126	733.6	835	830.1	933	941	1009	878.3	1008
SBL	195.5															
SBL2	191.5															
SOL	225.6															

Tabulka 6: Závislost doby výpočtu na počtu procesorů. (10000 záznamů o dimenzi 30, síť 6×6 neuronů)

Algoritmus	**Doba učení jedné epochy v ms v závislosti na počtu procesorů**															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
PBCLNPMW		9122	9308	9186	10315	10278	10304	10342	12119	12417	13066	13182	15433	15559	15704	15225
PBCLNPMW2		7276	7823	7684	8565	8416	8602	8588	10492	10718	10753	11366	13369	13340	13497	13594
PBDLDP		6132	4111	3219	2613	2272	1996	1794	2030	1845	1753	1681	1738	1693	1604	1581
PBDLDP2		6380	4218	3233	2604	2185	1905	1708	1851	1658	1560	1453	1420	1339	1289	1214
PBDLNPMW		11328	5669	4082	3136	2511	2092	1883	2524	2304	2273	2229	1917	2216	1892	1821
PBHLMW		11762	3582	3368	1888	2534	1874	1950	1508	1819	2353	2035	3041	1872	1035	2877
PBHLMW2		16504	3566	3735	3795	9068	3901	4283	4163	4224	4185	4316	4543	4705	4763	5258
PBHLMW3		17372	3798	3960	3971	4046	4123	4298	4417	4487	4403	4499	4663	4950	5099	5323
POCLNPMW		6159	6303	5392	6502	6322	6296	6344	7538	7775	7893	8055	9003	9320	17211	17620
PODLNP		6227	4693	3661	2881	2515	2244	1928	3097	2717	3664	3490	3633	3993	3644	2513
PODLNPMW		6745	5118	4622	3587	3028	2467	2161	2089	2569	1715	2339	1678	1546	2205	2074
SBL	11243															
SBL2	13795															
SOL	11628															

Tabulka 7: Závislost doby výpočtu na počtu procesorů. (10000 záznamů o dimenzi 30, síť 50×50 neuronů)